Real-Time Systems (2023) 59:664–704
https://doi.org/10.1007/s11241-023-09411-3

# A formal framework to design and prove trustworthy memory controllers

**Felipe Lisboa Malaquias**[1] · **Mihail Asavoae**[2] · **Florian Brandner**[1]

Accepted: 25 October 2023 / Published online: 14 November 2023


**Abstract**
In order to prove conformance to memory standards and bound memory access latency, recently proposed real-time DRAM controllers rely on paper and pencil proofs, which can be troubling: they are difficult to read and review, they are often shown only partially and/or rely on abstractions for the sake of conciseness, and they can easily diverge from the controller implementation, as no formal link is established between both. We propose a new framework written in Coq, in which we model a DRAM controller and its expected behaviour as a formal specification. The trustworthiness in our solution is two-fold: (1) proofs that are typically done on paper and pencil are now done in Coq and thus certified by its kernel, and (2) the reviewer's job develops into making sure that the formal specification matches the standards—instead of performing a thorough check of the mathematical formalism. Our framework provides a generic DRAM model capturing a set of controller properties as proof obligations, which all implementations must comply with. We focus on properties related to the assertiveness that timing constraints are respected, every incoming request is handled in bounded time, and the DRAM command protocol is respected. We refine our specification with two implementations based on widely-known arbitration policies—*First-in First-Out* (FIFO) and *Time-Division Multiplexing* (TDM). We extract proved code from our model and use it as a "trusted core" on a cycle-accurate DRAM simulator.

**Keywords** DRAM · Memory controller · Coq · Formal proof assistant

✉ Felipe Lisboa Malaquias
flisboa@telecom-paris.fr

Mihail Asavoae
mihail.asavoae@cea.fr

Florian Brandner
florian.brandner@telecom-paris.fr

[1] LTCI, Télécom Paris, Institut Polytechnique de Paris, Palaiseau, France

[2] CEA List, Université Paris-Saclay, Palaiseau, France

## 1 Extended version

This paper is an extension of a paper published at the *30th International Conference on Real-Time Networks and Systems* (RTNS'22) entitled *A Coq Framework For More Trustworthy DRAM Controllers* Lisboa Malaquias et al. (2022). The main additions are: a section where we discuss in greater detail and through examples how paper-and-pencil latency-analysis/proofs can lead to untrustworthiness (Sect. 2.1); a section where we present how the framework can be used to model controller semantics, such as *memory consistency models* (Sect. 6); and a section where we discuss how exactly our framework enhances trust w.r.t DRAM controllers design (Sect. 8). Furthermore, we extend and re-work several parts of the text of the original paper: textual descriptions are more detailed, code definitions previously omitted due to space constraints are shown, and new figures and tables are added with the goal of easing comprehension.

## 2 Introduction

Multi-core architectures pose a challenge for critical and mixed-criticality systems due to resource contention on the main memory. Memory Controllers (MCs) have to deal with a fundamental trade-off: ensuring predictable behaviour for critical (hard) requests, while providing good bandwidth (BW) for non-critical (soft) requests Guo and Pellizzoni (2017).

On one side of the spectrum, commercial-off-the-shelf (COTS) DRAM controllers employ a series of performance-driven optimisations that degrade predictability, such as reordering and bundling queued requests. As an example, take one of the most commonly used algorithms in high-performance controllers, *First-Ready First-Come-First-Serve* Rixner et al. (2000) (FR-FCFS), which prioritises row-hits over row-misses. Requests can be re-ordered, and hence, a critical request could suffer starvation due to the prioritisation of a non-critical request. As a consequence, this type of arbitration algorithm is poorly suited for memory controllers in real-time systems—as timing analysis gets more complex and memory accesses latencies can be difficult to bound.

On the other side of the spectrum, Real-Time DRAM controllers often adapt conservative techniques, offering predictability at the cost of decreased average-case performance. However, as the need for performance and computational demands in real-time system quickly grows Akesson et al. (2020), research has made significant progress in proposing predictable DRAM controllers at minimum performance cost in recent years (Mirosanlou et al. 2021, 2020; Valsan and Yun 2015; Guo and Pellizzoni 2017; Paolieri et al. 2009; Hassan et al. 2015; Li et al. 2014; Wu et al. 2013; Jalle et al. 2014; Ecco and Ernst 2015; Reineke et al. 2011; Schranzhofer et al. 2011; Yun et al. 2015).

Nonetheless, additionally to latency bounds, memory controller designers have to ensure that the DRAM commands generated by the MC and sent to the device

respect the timing constraints established by the *Joint Electron Devices Engineering Councils* (JEDEC). Taking this into account, the mathematical proofs that ensure conformance to the JEDEC standard and bound memory access latency can be lengthy, complex, and lack readability. These issues are presented in greater detail in Sect. 2.1.

Formal methods come in handy when dealing with such problems. Increasing the trust in the designed system is delegated to specialised tools, which, depending on the approach, can do different things. The two most popular approaches are model checking and deductive verification. The former performs exhaustive exploration of a mathematical model. The latter can be used to generate mathematical proof obligations that are part of the formal specification of a system (a property also known as the Curry-Howard isomorphism Sørensen and Urzyczyn (2006)); therefore, every implementation of the specification must provide proofs that these obligations are met.

While model checking provides greater automation, deductive verification has better expressivity Shankar (2018). In the context of Real-Time Memory Controllers, the expressivity provided by the deductive approach brings benefits:

1. If one convinces oneself that the formal specification of a system and the stated properties really correspond to the addressed problem—here DRAM controllers—then one can become agnostic to the actual implementations and the inner mechanisms of the proofs, as long as they are accepted by the tool's kernel;
2. Different than model checking, deductive verification allows modelling of more complex systems, since computation does not dependent on the number of possible states. In other words, more often than not, model checking approaches do not scale well for models with large/unbounded number of elements and have to apply abstractions to keep the state space manageable.

More specifically, we use Coq,[1] an *interactive theorem prover*. It allows its user to write specifications, implement them, and discharge proof obligations through the aid of proof scripts. We choose Coq over other prove assistants for a variety of reasons—largely based on the points made by Chlipala (2022) in the book *Certified Programming with Dependent Types*:

1. Coq's specification language supports *Higher-Order* logic—which allows us to enjoy the benefits of conventional functional programming languages.
2. It is based on dependent types—which allows us to include references to programs inside of types.
3. Coq is based on an easy-to-check and small kernel to check the correctness of proofs (according to the "de Bruijn" criterion).
4. It supports coding new proof manipulations, and thanks to the third point, these new manipulations cannot be incorrect.

[1] https://coq.inria.fr/.

The trustworthiness that our model provides is two-fold: 1) paper-and-pencil proofs are replaced by Coq-written proofs, which are certified by Coq's kernel, and 2) since proofs are presented as artefacts, reading and reviewing designs introduced within the framework is simpler: instead of performing a thorough check of the underlying mathematical formalism, one just needs to check if the specification accurately captures what is described in the standard. In Sect. 8, we discuss in detail how exactly our framework impacts *trust*.

***Novel Contributions***

- We propose a framework written in Coq that offers a higher degree of trust w.r.t DRAM controllers design. It contains a generic and reusable model of memory controllers in the form of a formal specification that can be refined through actual implementations. The specification carries correctness criteria as proof obligations.
- We refine the specification with two instances: one based on *First-In First-Out* (FIFO) arbitration and the other on *Time-division multiplexing* (TDM). For both, proof obligations are met and certified by Coq's kernel. The proofs written for these two instances serve as a model for future implementations and uses of the framework.
- The framework cleanly separates the specifications and proofs from the implementation, which allows us to extract executable code from Coq that can be used as a "root of trust" in a cycle-accurate DRAM simulator.

***Paper Organisation***

Section 2.1 reviews recently proposed Real-Time Memory Controllers in order to highlight the issues that motivate our work, Sect. 2.2 presents how the DRAM communication protocol, the related timing constraints, and the worst-case latency bounds have been modelled with the aid of formal methods, and Sect. 2.3 highlights how Coq-based solutions have recently gained ground in the design of trustworthy hardware. Section 3.1 presents a concise background on DRAM systems and Sect. 3.2 presents Coq's *Type Classes*, a key feature used in the development of our framework. Sections 4 and 5 present our contributions: while the former introduces the formal specification and the proof obligations in it, the latter shows how we refine the specification, derive latency bounds and resolve proof obligations. Section 6 shows how the framework can be used to model controller semantics. Section 7 discusses how we extract Haskell code from our model and embedded it into an existing cycle-accurate DRAM system simulator. Finally, Sect. 8 discusses our framework's implications on *trust*, and Sect. 9 concludes the paper by revisiting our contributions and presenting future research directions.

## 3 Related work

### 3.1 Real-time memory controllers & Trustworthiness

Latency-analysis (or timing analysis) has always been a key element of any work introducing new Real-Time hardware components. This is because the latency

introduced by the hardware logic has to be upper-bounded for it to be accounted in a task's *Worst Case Execution Time* (WCET). Two important components that introduce significant latency in a system are the memory and the memory controller. Historically, timing analyses w.r.t these components, along with proofs of conformance to the JEDEC standards, have been done on paper-and-pencil, which can be hard to deal with, in more than one aspect.

We analysed the most-often cited Real-Time memory controllers in literature regarding the length of latency analysis/proofs (Ecco and Ernst 2015; Guo and Pellizzoni 2017; Hassan et al. 2015; Jalle et al. 2014; Li et al. 2014; Mirosanlou et al. 2020, 2021; Paolieri et al. 2009; Reineke et al. 2011; Schranzhofer et al. 2011; Valsan and Yun 2015; Wu et al. 2013; Yun et al. 2015). In each work, latency analysis takes from 30% up to 50% of the total space of the paper. Although the content and decision procedures of proofs might be of interest for fields such as mathematics or physics, we advocate for the point of view that hardware design should present them merely as artefacts. Therefore, the space that these proofs take in the papers could be better used – to include details about implementations, experiments, results, and other engineering aspects, for example. Approaching the problem through computer-aided formal methods allows us to properly treat proofs as artefacts and hide the underlying mathematical formalism from the reader.

Moreover, these analyses are often presented for the simplest of cases, leaving out essential details. As an example, Mirosanlou et al. (2021) only derives static *Worst Case Latency* (WCL) for read requests, briefly arguing that the analysis for write requests is very similar, which is therefore omitted. Work by Ecco and Ernst (2015) proceeds in the same way, omitting the proof of a Lemma based on the similarity argument.

In other work, the timings analysis is based on assumptions that reduce the set of valid scenarios. For instance, work by Guo and Pellizzoni (2017) describes their timing analysis as being only valid for a subset of DDR3 devices.[2] Work by Wu et al. (2013) assumes that the task under analysis runs non-preemptively on its assigned core, arguing that the analysis could be easily extended if the maximum number of preemptions is known (although the claim is not supported and details are not given).

Furthermore, works may base themselves on different sets of assumptions. It is therefore hard for users of the design, readers, and reviewers to keep track of the assumptions within the paper, often scattered throughout and/or presented as side notes. It is even harder to compare the set of assumptions that validate different Real-Time memory controller designs. This issue is identified and addressed in a survey by Guo et al. (2018), in which the set of assumptions for a dozen Real-Time memory controllers is made explicit. As an example, in order to compare the WCL analysis performed by Ecco and Ernst (2015), the authors of the survey had to perform a new auxiliary analysis applying the common assumption on the arrival of requests used in related work.

[2] Although a footnote states that the analysis is still applicable if the right parameters are selected, the claim is not supported.

Although *we do not deem wrong* nor contest the author's choices in each mentioned work, we do see the points made in the paragraphs above as *possible sources of untrustworthiness*. These points are resumed below:

- Proofs are not machine-checked, i.e., checking that the analysis is correct still depends on human labour by the authors themselves and through peer-review. The inherent difficulty, length, and incompleteness of the presented proofs makes peer-reviewing a challenging task requiring expert knowledge. Regular users and readers also have the deal with these complex proofs—which should rather be presented as artefacts.
- Works often base themselves on different sets of assumptions. Since these assumptions are often not highlighted or made explicit, it is difficult to keep track of the assumptions within the work itself, and to compare assumptions between different approaches.
- The fact that there is no formal link between system implementation and the mathematical abstractions used to perform timing analysis may also introduce untrustworthiness.

### 3.2 DRAM & formal methods

Jung et al. (2019) model the DRAM timing protocols with timed Petri Nets and generate executable code that can be used for simulation-based validation. However, as their approach only takes the device's timing constraints into consideration, it cannot handle system properties, such as the worst-case latency or the protocol correctness.

Li et al. (2016) use timed automata (TA) models of a memory controller to derive the *Worst-Case Bandwidth* (WCBW) and *Worst-Case Response Time* (WCRT) through the Uppaal model checker Larsen et al. (1997). However, when performing worst-case analysis, in order to keep the state-space limited, they assume that each requestor has at most one outstanding request, i.e., they constrain how requests arrive in the system. Moreover, getting convinced that the TA models reflect the actual DRAM states is not straightforward, since the models are complex and written by hand. We come back to this point in Sect. 8, where we debate how exactly our framework provides more trust.

Hassan and Patel (2017) use *Linear Temporal Logic* (LTL) formulas to specify the correctness of memory controller models. However, the specification is not used to prove the actual implementation correct. Instead, counterexamples are obtained from simplified models of an implementation by bounded model checking, which are then used to build a test bench for the validation of the implementation. The considered models might also slightly diverge from the actual implementation. The obtained test bench might thus be incomplete or contain false positives/negatives.

### 3.3 Uses of Coq in related problems

Coq, and other similar theorem provers, such as Isabelle/HOL Nipkow et al. (2002) and Fstar,[3] have been used successfully to model and prove the correctness of complex systems. PROSA Cerqueira et al. (2016), for instance, proposes a framework to model real-time task scheduling (including task properties, scheduling policies, etc.), which allows to prove scheduling policies correct under a given set of conditions using Coq. Guo et al. (2019) go even a step further by connecting PROSA's formal model to an actual real-time operating system RT-CertiKOS—thus proving the scheduler implementation correct.

In 2020, Bozhko and Brandenburg (2020) built a Coq framework, leveraging previous results from PROSA, to formally reason about Response Time Analysis (RTA), and more specifically, the ubiquitous principle of *busy-window*. The authors motivate their work by identifying a lack for commonality and formality in literature, claiming that "*the general idea [of the busy-window principle] has become part of the real-time folklore, spread across many papers, where it is frequently redeveloped from scratch, using ad-hoc notation and problem-specific definitions*". Moreover, they advocate that "*papers introducing novel RTA should not start from first principles, but rather build on a well understood and general foundation [...]*". This reasoning is on par with the arguments we present in this work, which aims at providing a formal foundation for memory controller design. In 2022, Maida et al. (2022) extended the results from Bozhko and Brandenburg (2020) by connecting the foundational RTA analysis to a larger formal system used to produce "*strong and independently checkable evidence of temporal correctness*".

Several research groups and companies are promoting the use of Coq for the development of trustworthy hardware. Researchers at Google have developed a plug-in replacement of the cryptographic core of the OpenTitan[4] *silicon root of trust*. The hardware is implemented in Cava[5]—a *Domain-Specific Language* (DSL) based on Lava Bjesse et al. (1998) and embedded within Coq. Cava allows to leverage Coq's proof-engine and to derive SystemVerilog code. Kami, a similar system, was first developed by Choi et al. (2017) and later adopted by SiFive. It has its roots in Bluespec,[6] which also serves as the target language derived from Kami. The verification procedure in Kami is based on proving that modules *refine* a given specification based on *trace* inclusion, i.e., the specification defines traces of observable events which have to be respected by its implementation—an approach similar to the one we adopt in this paper.

Following the same rule-based design approach, the more recently proposed DSLs Koîka Bourgeat et al. (2020) and Choi et al. (2022) extend the bases set by the Kami project. The former does so by introducing semantics that can be used to prove performance properties, e.g., that a pipelined system indeed behaves like a

[3] https://www.fstar-lang.org/.
[4] https://opentitan.org/.
[5] https://github.com/project-oak/silveroak.
[6] https://bluespec.com/.

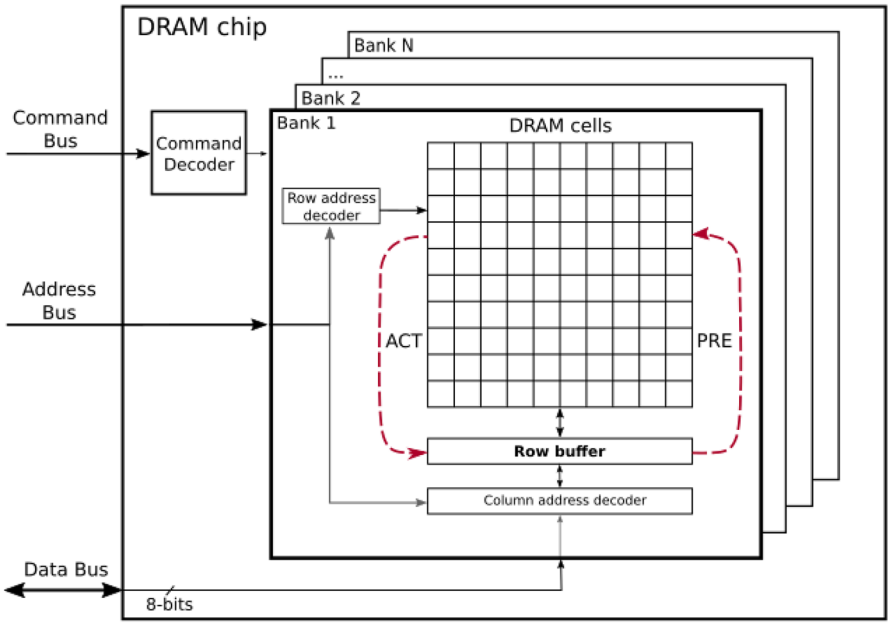


**Fig. 1** DRAM chip structure

pipelined system. The latter proposes a method to prove the *serializability* property of cache-coherence protocol.

## 4 Background

### 4.1 DRAM systems

We centre the discussion around DDR4 devices, as they can be seen as super-sets of older technologies (although sometimes we refer to DDR3 devices as well). The topmost logical unit in a DRAM module is a rank, followed by bank-groups and banks. Each bank is composed of a grid of memory cells, a page-sized buffer—the row-buffer—and sense amplifiers. The overall structure of a DDR3 chip is depicted in Fig. 1 (for DDR4 the $N$ banks in the figure would represent a single bank-group).

DRAM controllers are typically organised in two parts: *front-end* and *back-end*. In the front-end, three things take place: (1) The *Address Mapping*, which determines how a request's address is mapped to the corresponding rank, bank-group, bank, row, and column. (2) The *Scheduling Policy*, responsible for establishing the order in which requests will be serviced. (3) The *Command Generation*, responsible for generating the needed commands for each request, based on the status of the targeted bank. In the back-end, the generated commands go through an arbitration policy and are sent to the device's command bus—possibly in a different order.

**Table 1** JEDEC timing constraints for a DDR3 and a DDR4 device

| Symbol | Description | DDR3-1600K (in data bus clock cycles) | DDR4-2400U (in data bus clock cycles) |
|---|---|---|---|
| Exclusively intra-bank | | | |
| $t_{RCD}$ | ACT to WR/RD | 11 | 18 |
| $t_{RP}$ | PRE to ACT | 11 | 18 |
| $t_{RC}$ | ACT to ACT | 39 | 57 |
| $t_{RAS}$ | ACT to PRE | 28 (min), 9xtREFI (max) | 39 (min), 9x tREFI (max) |
| $t_{WL}$ | WR to data bus transfer | 8 | 12 |
| $t_{RL}$ | RD to data bus transfer | 11 | 18 |
| $t_{RTP}$ | RD to PRE | 6 | 9 |
| $t_{WR}$ | WR data to PRE | 12 | 15 |
| Intra and inter-bank | | | |
| $t_{RD-to-WR}$ | RD to WR | 9 | 12 |
| $t_{WTR}$ | End of WR transaction to RD | 6 | s=3, l=9 |
| $t_{WR-to-RD}$ | WR to RD | 18 | s=19,l=25 |
| $t_{BURST}$ | Data bus transfer | 4 | 4 |
| $t_{CCD}$ | WR-to-WR or RD-to-RD | 4 | s=4, l=6 |
| Exclusively inter-bank | | | |
| $t_{RRD}$ | ACT to ACT | 5 | s=7, l=8 |
| $t_{FAW}$ | Four ACT window | 24 | 30 |

In the device name, the number represents the device's speed in MHz in the letter is the speed grade

There are several types of commands, but for conciseness, only the four related to servicing requests are explained: (1) *Activate* (ACT) commands are responsible for loading the content of a row into the row-buffer, that acts like a small cache for the bank. (2) *Column Address Strobe* (CAS) can be either reads (RD) or writes (WR). If it is a RD, then the command selects a column address and transfers the corresponding data from the row-buffer to the data bus. The WR commands do the opposite, choosing a column address from the data bus and writing it in the targeted bank's row-buffer. Bear in mind that CAS commands can only be issued if there is a row currently present in the row-buffer, i.e., if an ACT commands has been previously issued. (3) If a request aims at a different row, the data present in the row-buffer needs to be written back to the DRAM's cell matrix. This is because loading rows, i.e., reading the content of a DRAM cell is a destructive process. This re-write procedure is done by *Precharge* (PRE) commands. (4) The capacitors in the DRAM cells have to be recharged using *Refresh* (REF) commands, which have to be issued periodically. Similar to existing work (Li et al. 2014; Jalle et al. 2014; Yun et al. 2015; Ecco and Ernst 2015; Hassan et al. 2015; Li et al. 2016; Guo and Pellizzoni 2017), refresh-induced delays are not considered in this work at the moment—these delays are best taken into consideration as additional interference (Bhat and Mueller 2011; Wu et al. 2013).

The JEDEC standards establish several constraints on how far apart in time different commands must be. These constraints are summarised in Table 1. Note that

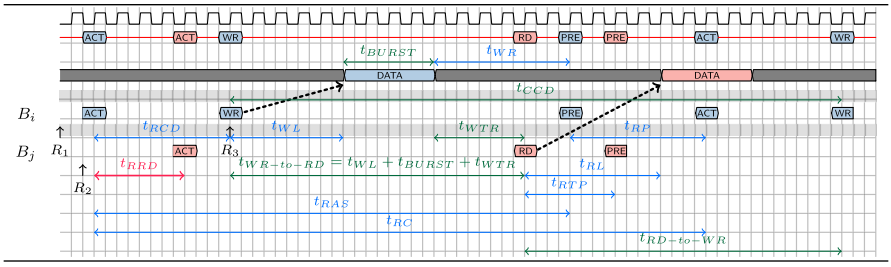


**Fig. 2** Example of timing constraints for a DDR3-800E device. Commands and data: Bank $i$ ($B_i$), Bank $j$ ($B_j$). Constraints: Exclusively intra-bank, Inter- and intra-bank, Exclusively inter-bank

some constraints only apply to different banks, some only to the same bank, and some to all banks. For DDR4 devices, some constraints can have two values, with identifiers $s$ (short) and $l$ (long). The former applies to commands aiming at banks of different bank groups, the latter to banks in the same group. Note also that the constraint $t_{RAS}$ is the only one to impose both upper and lower bounds.

Next, we discuss rows-hits/misses and row-buffer/page policy.[7] If a request, aiming at a given row, finds its data already in the row-buffer, it is said to be a *row-hit*. If, however, the targeted row is different than the one in the row-buffer, it is said to be a *row-miss*. One of the key aspects of (real-time) DRAM controllers is how to handle the row-buffer. One strategy is to keep rows loaded in the row-buffer for as long as possible. This way, row-hits will enjoy small latencies. This is known as the *open-page policy*. Since the controller leaves the bank in an unknown state, the timing analysis employed to bound the worst-case latency grows in complexity. Another strategy is to treat every request as a row-miss, issuing the same PRE-ACT-CAS sequence for every request. This is known as the *closed-page policy*. It offers the benefit of simplified timing analysis at the cost of average-case performance.

Figure 2 illustrates a command scheduling scheme on a DDR3-800E device. The clock signal depicted in the figure is the data bus clock. The second line from top to bottom corresponds to the command bus (NOP commands are not shown). The third line from top to bottom is the data bus. The two lines below that correspond to the commands issued to banks $i$ and $j$, respectively.

Consider the two banks $i$ and $j$ ($B_i$ and $B_j$) to be idle at the initial time instant (left-most clock cycle in the figure). Moreover, consider that the system is serving three distinct requests: $R_1$ is a write to bank $i$, $R_2$ a read to bank $j$, and $R_3$ again writes in bank $i$. The controller issues commands for both $R_1$ and $R_2$ concurrently, always respecting the timing constraints between commands. Note that $R_1$ and $R_2$ could be

[7] The terms row-buffer policy and page policy are used interchangeably.

either row-misses or hits, while $R_3$ is a row-miss, since it has to issue a PRE. Except from *Refresh* related constraints, all timing constraints are depicted in the figure.[8]

### 4.2 A brief introduction to Coq

In the rest of the work we will heavily rely on *Type Classes*. While presenting details about Coq's proof engine is out of the scope of this work, we will try to give an intuitive explanation of these type classes and their use.

A type class in Coq is, in some regards, similar to abstract classes or interfaces from object-oriented programming languages—it allows us to specify a set of members. Listing 1 shows an example of a container—similar to the Container[9] and Collection[10] concepts in C++ or Java, respectively.

Listing 1: Interface specification for a container/collection.

```
Class container_t {T C : Type} := mkContainer {
 append : T → C → C;  (* append element to container *)
 contains : T → C → bool; (* is element in container? *)
 size : C → nat;  (* get number of element in container *)
 app_inc : forall (x : T) (c : C), size (append x c) = (size c).+1
 app_in : forall (x : T) (c : C), contains x (append x c)
}.
```

The class `container_t` takes two type parameters, where `T` specifies the `Type` of the container's elements and `C` the type of the data structure holding those elements. The identifier `mkContainer` is the class constructor—a function used to build instances. The member `append` is a function, taking an element of type `T` and a container of type `C` as parameters, while returning a new container. The member `contains` is a function with an element and a container as parameter that returns a boolean (`bool`), while `size` expects a container as input and returns a natural number (`nat`).

Intuitively it is clear what the append function is supposed to do. Still, in Java or C++ the *meaning* of the function is usually explained explicitly in comments or an additional document. Coq, however, allows to make the expected behaviour explicit—as illustrated by the members `app_inc` and `app_in`. These members are *proof obligations* (**POs**), i.e., properties that every instantiation of the container class has to respect. They indicate that the size of a container should increment by one when an element is appended and that a newly appended element always has to be present in the container. The usage of type classes is similar to classes in

[8] In Fig. 2, the arbitration, though valid, does not correspond to any specific algorithm, as it simply illustrates a situation where all constraints are respected. Moreover, the timing constraints in the standard represent **minimum** intervals. In the figure, the depicted constraints assume values greater or equal to these minimum values.

[9] https://en.cppreference.com/w/cpp/named_req/Container.

[10] https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html.

conventional programming languages—they can be used as parameters in functions and other type classes, where its members can be used.

Furthermore, the Coq command `Context` allows us to introduce variables in the local context—which behave like *abstract* instances, i.e., their members (functions and logical propositions/POs) can be used in the local context, even if concrete instances do not yet exist. As an example, as shown in Listing 1, we introduce the types `Q` and `R` in the local context, followed by `X`, which is an abstract instance of `container_t` using the previously defined types `Q` and `R`.

Listing 2: Using the `container_t` class.

```
(* introduces variables Q and R in the local context *)
Context {Q R: Type}.
(* introduces variable X of type container_t in the local context *)
Context {X : container_t (T:=Q) (C:=R)}.
```

We can now use the member functions of X—the abstract instance of `container_t`—to write other functions, such as `concatenate_list` (Listing 2). It takes a list of type `seq Q`[11] and recursively appends its elements into a variable of type `R` (previously defined in Listing 1). Note how the `append` function is used within the function.

Listing 3: Writing a function using members from `container_t`

```
Fixpoint concatenate_list (a : R) (b : seq Q) :=
  match b with
  | [::] ⇒ a
  | hd::tl ⇒ append hd (concatenate_list a tl)
  end.
```

Moreover, we can prove properties about `concatenate_list`, as shown by the `Theorem size_concatenate` in Listing 3—a theorem stating that the size of a container resulting from concatenating a list to a container is the sum of the sizes of each. The code between commands `Proof` and `Qed` is a *proof script*. A proof script relies on the use of `tactics`, which interactively allow the user to perform steps in proving a theorem. In this specific proof, we use the tactics `induction`, which performs mathematical induction on a specified variable; `simpl`, which simplifies terms by reduction; `rewrite`, which rewrites terms manually; and `reflexivity`, which is used to trivially prove goals in the equality form. Note how the proof applies the property `app_inc`, a member of the `container_t` type class.

[11] The `seq` type is an alias of Coq's standard list type with several useful pre-defined functions and lemmas. It is part of *Mathematical Components* (`mathcomp`)—a widely-used Coq library for formalised mathematics.

Listing 4: Using `container_t` to prove a property

```
Theorem size_concatenate (la : C) (lb : seq T) :
  size (concatenate_list la lb) = size la + seq.size lb.
Proof.
  induction lb; simpl; try rewrite addn0; try reflexivity.
  by rewrite app_inc IHlb - addnS.
Qed.
```

It is important to keep in mind that having only introduced abstract instances, the proof `size_append` will hold for any concrete instance of `container_t`. This feature is widely used in our framework to build implementations and proofs that are agnostic to the arrival model of requests, for instance.

In addition, type classes can also be instantiated, i.e., associated with *concrete* implementations of the member functions. Listing 4 shows an example of a concrete instance of `container_t` where we define some functions using the `seq` type.

Listing 5: Functions implementing the collection interface.

```
(* Using mathcomp.ssreflect.seq functions *)
Definition my_append (n : nat) (c : seq nat) : seq nat :=
 seq.append c n.

Definition my_contains (n : nat) (c : seq nat) : bool :=
 n \in c. (* returns true if n is a member of c *)

Definition my_size (c : seq nat) : nat :=
 seq.size c. (* returns the length of c *)
```

In Listing 6, we write proofs for the members `app_inc` and `app_in` using our defined functions and instantiate the container through its constructor—`mkContainer`. Notice that aside from types and functions, proofs are also passed as arguments to the constructor.

Listing 6: Proofs and instantiation of the collection interface.

```
Theorem my_appn_inc (x : nat) (c : seq nat) :
 my_size (my_append x c) = (my_size c).+1.
Proof.
 unfold my_size, my_append; by rewrite cats1 size_rcons.
Qed.

Theorem my_app_in (x : nat) (c : seq nat) :
 my_contains x (my_append x c).
Proof.
 unfold my_contains, my_append; by rewrite mem_cat mem_seq1 eq_refl orbT.
Qed.

(* Container instance *)
Instance my_list : container_t := mkContainer
        nat (seq nat) (* Types *)
        my_append my_contains my_size (* Functions *)
        my_appn_inc my_app_in. (* Proofs *)
```

Furthermore, we occasionally use Coq *Records* instead of type classes. Records are simple structures with fields—similar to C *structs* and record types used in other programming languages. In fact, each type class definition gives rise to a corresponding record declaration (apart from type classes with a single method).

## 5 Specification & proof obligations

The framework we propose is organised in two parts: 1) A generic and reusable formal specification containing all correctness criteria; and 2) Concrete Real-Time memory controller implementations. The formal specification is composed by the following elements: a DRAM model, a memory controller (or memory interface) model, a request arrival model, and an implementation interface—each model being a PO-carrying type class. These elements are presented, respectively, in Sects. 4.1, 4.2, 4.3, and 4.4. Implementations are discussed in Sect. 5. A high-level representation of the framework's architecture can be seen in Fig. 3.

In the figure, one can identify three main parts: the JEDEC standard at the top, the formal specification (the largest rectangle), and the implementations, at the bottom. The framework is designed in such a way that *any new Real-Time memory controller can be introduced as an implementation*. In practical terms, the design flow consists of using the implementation interface (bottom-most arrow in the figure) to implement the Memory Controller class, which consists of writing a transition system to produce DRAM command traces. Writing the formal specification itself and capturing properties described in the standard as proof obligations is a task only

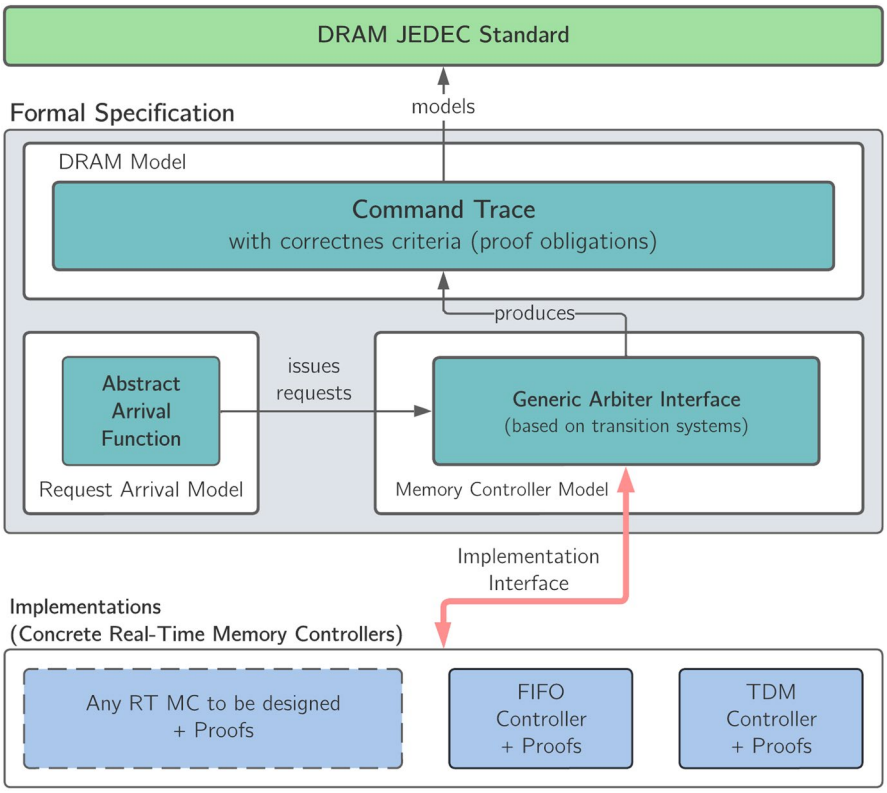


**Fig. 3** Framework Architecture

performed once (by the framework development team). Naturally, these properties can eventually be extended.[12]

Listing 7: System Configuration.

```
Class System_configuration := {
  BANKGROUPS : nat;   (* number of bankgroups *)
  BANKS      : nat;   (* number of banks *)
  ...
  T_BURST    : nat;
  T_WL       : nat;
  T_RRD      : nat;
  ...
  (* Proof obligations *)
  BANKS_pos : BANKS != 0; (* number of banks has to be non-zero *)
  T_RRD_pos :  T_RRD != 0; (* T_RRD constraint has to be non-zero *)
  ...
}.
```

[12] As it will be discussed in Sect. 9, it is in our plans to extend the properties captured by the formal specification.

### 5.1 DRAM model

Similar to the JEDEC standard itself, we do not model the internal state of actual DRAM devices nor the content of the memory. Instead we only model the device's main characteristics and its command bus. This is sufficient since the device's responses and internal state, as well as the usage of the data bus is entirely determined by the commands.

The first building block is a type class called `System_configuration`—shown in Listing 7. It provides symbolic names for all device parameters, which can then be used in algorithms and proofs. The latter are thus valid for all possible valuations of these parameters, i.e. all devices. For instance, it defines the symbol `T_RRD`—which specifies the intra-bank ACT to ACT delay—and all the other constraints from Table 1. The class also carries proof obligations that bound these symbols to positive values.

Moreover, we define requests as a record with six fields, as shown in Listing 8. The listing also shows the definition of `Request_kind_t`, an enumeration of possible kinds of requests. The remaining definitions are not shown for brevity: `Requestor_t` describes the type of requestors, and the remaining definitions of `Bank_group_t`, `Bank_t`, and `Row_t` are numeric types derived from the parameters of the `System_configuration` class.

Listing 8: Definitions of `Request_t` and `Request_kind_t`.

```
Inductive Request_kind_t : Set :=
    RD | WR.

Record Request_t := mkReq {
    Requestor : Requestor_t; (* The type of requestors *)
    Date : nat; (* The arrival date of requests *)
    Kind : Request_kind_t; (* Either if it is a RD or WR *)
    Bank_group : Bank_group_t; (* Type of bankgroups : bounded natural *)
    Bank : Bank_t; (* Type of banks : bounded natural *)
    Row  : Row_t (* Type of rows : bounded natural *)
}.
```

Listing 9: Definitions of `Command_t` and `Commands_t`.

```
Inductive Command_kind_t : Set :=
    ACT | PRE | PREA | CRD | CWR | NOP.
    (* CRD and CWR are CAS commands *)

Record Command_t := mkCmd {
    CDate   : nat; (* The issue date of commands *)
    CKind   : Command_kind_t; (* The type of commands *)
    Request : Request_t (* The request that has generated the command *)
}.

Definition Commands_t := seq Command_t.
```

We define DRAM commands as a record with three fields, as shown in Listing 9. The first field, `CDate`, a `nat` type, is the date (time stamp) that the controller has issued the command; the second, `CKind`, of type `Command_kind_t`, represents the command type, and `Request`, of type `Request_t`, is the corresponding request that lead to the command being generated. With the `Command_t` type, the `Commands_t` type can also be defined, which is simply a list of commands.

Based on the types presented in Listings 8 and 9, we model the DRAM command bus. This is done through the record `Trace_t`, which models the commands sent to the DRAM device over time—with one command being issued per bus clock cycle—as shown in Listing 10.

Listing 10: Trace of commands with proof obligations.

```
Record Trace_t := mkTrace {
 Commands : Commands_t;
 Time     : nat;
 ...
 (* PO: related to JEDEC timing constraint *)
 Cmds_T_RRD_ok : forall a b, a \in Commands → b \in Commands → isACT a → isACT b →
      (Same_Bank a b = false) → Before a b → a.(CDate) + T_RRD <= b.(CDate);
 ...
 (* PO: related to the correctness of the protocol *)
 Cmds_row_ok : forall b c, b \in Commands → c \in Commands → isCAS b → isPRE c →
      Same_Bank b c → Before c b →
      exists a, (a \in Commands) && (isACT a) && (Same_Row a b) && (After a c) && (
           Before a b)

 (* PO: related to the correctness of the protocol *)
 Cmds_ACT_ok: forall b c, c \in Commands → b \in Commands → isACT_or_CAS c →
      isACT b → Same_Bank c b → Before c b
    → exists a, (a \in Commands) && (isPRE a) && (Same_Bank c a) && (Before a b)
        && (After a c).

 (* implies the init state of the memory *)
 Cmds_initial : forall b, b \in Commands → isCAS b →
    exists a, (a \in Commands) && (isACT a) && (Same_Row a b) && (Before a b)
}.
```

A trace contains a command list (`Commands`), the last time stamp (`Time`), and a series of POs. Bear in mind that Listing 10 omits several POs and some function definitions for conciseness. The terms `isACT`, `isCAS`, `isPRE`, `After`, `Same_Bank` (implies same bank group), `Same_Row` (implies same bank), `Before`, and `After` are functions that evaluate to a `bool` type. Though not shown here, their definition can be taken literally by their name, e.g., `Before a b` evaluates to true if command `a` was issued before command `b`.

The first PO appearing in Listing 10, `Cmds_T_RRD_ok`, is read as: for two given commands `a` and `b`, knowing that `a` and `b` are elements of the list `Commands`, that both are ACT commands targeting different banks, that `a` was issued before `b`, then `a.(CDate)` $+T_{RRD} \leq$ `b.(CDate)` must hold. This ensures that every controller implementation has to respect the $t_{RRD}$ constraint. It follows that any implementation when calling the constructor `mkTrace` must provide a proof that this obligation is met. We model all JEDEC timing constraints in a similar way as `Cmds_T_RRD_ok`.

Table 2 establishes a correspondence between PO names and the underlying property described in the JEDEC DDR4 standard JEDEC (2021).

The second (`Cmds_row_ok`), third (`Cmds_ACT_ok`), and fourth (`Cmds_initial`) POs in `Trace_t` relate to what we call *protocol correctness*. These conditions—taken from the JEDEC standards (c.f Table 2)—ensure the correct functioning of controllers, i.e., the fact that only valid commands should be issued.

More specifically, the PO `Cmds_row_ok` ensures that there is always an ACT command to a given bank and row between a PRE to that bank and a CAS to that bank and row. In natural language, the PO is read as: for two given commands `b` and `c`, knowing that `b` and `c` are elements of the list `Commands`, `b` is a CAS command, `c` is an ACT or a PRE, `b` and `c` target the same bank, and `c` is issued before `b`, then command `a` must exist, where `a` is also in `Commands`, is an ACT command to the same bank and row as `b`, and is issued before `b` and after `c`. This situation is depicted in Fig. 4. In the figure (and in Fig. 5), the notation $CAS_{(bg,bk,r,cl)}$ represents a CAS to bank-group *bg*, bank *bk*, row *r* and column *cl*; $PRE_{(bg,bk)}$ represents a precharge to bank-group *bg*, bank *bk*; and $ACT_{(bg,bk,r)}$ represents an activate to bank-group *bg*, bank *bk*, and row *r*. The symbol "_" means that the corresponding field does not play a role in the PO, e.g., a $CAS_{(0,0,0,\_)}$ represents a CAS to bank group 0, bank 0, row 0, and any column.

The PO `Cmds_ACT_ok` states that an ACT to a given bank and row is always preceded by a matching PRE to the same bank, without any ACT or CAS to the same bank in-between. The PO is depicted in Fig. 5, where the PO is met by the conjunction of scenarios $S_1$ and $S_2$.

Furthermore, we model the memory's initial state through the `Cmds_initial` PO. It states that any CAS should be preceded by an ACT command, which implies that a CAS cannot be the first command sent to the device; only PRE and ACT commands are accepted. In other words, this is an assumption that at initialisation, every bank is closed, i.e., no row is loaded in any of the row-buffers, thus an ACT to a certain bank is needed before any CAS to that bank can be issued.

Together, these three POs guarantee that implemented controllers generate valid commands and respect the protocol described in the standard. The conditions established by the POs can also be visualised in Table 3. We use the notation $\rightarrow$ to denote an immediate sequence between commands, e.g, $PRE_{(bg,bg)} \rightarrow CAS_{(bg,bk,r,c)}$ represents a CAS issued directly after a PRE.

Each coloured cell in the table corresponds to a sequence and the fact if either it is allowed or not, e.g., the sequence $PRE_{(bg,bk)} \rightarrow CAS_{(bg,bk,r_0,\_)}$ is forbidden by `Cmds_row_ok`. Moreover, it can be seen from the table that consecutive ACT commands to the same bank, either to the same or different rows ($ACT_{(bg,bk,r_0)}$ or $ACT_{(bg,bk,r_1)}$), are forbidden by `Cmds_ACT_ok`. Additionally, ACT commands following a CAS command are also forbidden by `Cmds_ACT_ok`. Finally, a CAS following an ACT to a different row ($ACT_{(bg,bk,r_1)} \rightarrow CAS_{(bg,bk,r_0,\_)}$) is forbidden as well. The latter condition is met by applying the three POs: `Cmds_initial` states that an ACT to row $r_0$ must exist before $CAS_{(bg,bk,r_0,c)}$; then, `Cmds_ACT_ok` ensures that a $PRE_{(bg,bk)}$ is

**Table 2** Correspondence between proof obligations in our code and the DDR4 JEDEC Standard No. 79-4 JEDEC (2021)

| PO name | Property | Location in the standard |
|---|---|---|
| Timing constraints | | |
| Cmds_T_RCD_ok | $t_{RCD}$ is respected | Pgs. 163-166 |
| Cmds_T_RP_ok | $t_{RP}$ is respected | Pgs. 163-166 |
| Cmds_T_RC_ok | $t_{RC}$ is respected | Pgs. 163-166 |
| Cmds_T_RAS_ok | $t_{RAS}$ is respected | Pgs. 163-166 |
| Cmds_T_RTP_ok | $t_{RTP}$ is respected | Pgs. 101-103, 189-193 |
| Cmds_T_WTP_ok | Constraint between WR and PRE. Unnamed in the standard. ($t_{WR}$ is respected)[1] | Pgs. 115 (Figure 122), 188-193 |
| Cmds_T_RtoW_ok | Constraint between RD and WR. Unnamed in the standard | Pg.104 (Figure 100) |
| Cmds_T_WtoR_SBG_ok<br>Cmds_T_WtoR_DBG_ok[3] | $t_{WTR}$ is respected[2] | Pg. 112 (Figure 116) |
| Cmds_T_CCD_WR_SBG_ok<br>Cmds_T_CCD_RD_SBG_ok<br>Cmds_T_CCD_WR_DBG_ok<br>Cmds_T_CCD_RD_DBG_ok[3] | $t_{CCD}$ is respected | Pgs. 188-193 |
| Cmds_T_RRD_SBG_ok<br>Cmds_T_RRD_DBG_ok[3] | $t_{RRD}$ is respected | Pgs. 188-193 |
| Cmds_T_FAW_ok | $t_{FAW}$ is respected | Pgs. 188-193 |
| Command protocol correctness | | |
| Cmds_ACT_ok | Basic functionality is assured | Pgs. 8 and 9 |
| Cmds_row_ok | Basic functionality is assured | Pgs. 8 and 9 |
| Cmds_initial | Basic functionality is assured | Pgs. 8 and 9 |

[1] The PO `Cmds_T_WTP_ok` models the minimal distance between a WR and a PRE command, while the constraint $t_{WR}$ in the standard represents the distance between the end of the WR's **data bus utilisation** and a PRE command. Since our specification only considers the command bus trace, we capture timing constraints on the data bus only indirectly via the WR and PRE commands.

[2] Similar to `Cmds_T_WTP_ok`/$t_{WR}$,[1] the constraint $t_{WTR}$ in the standard represents the minimum distance between the end of the data bus utilisation of a WR and a RD command. We model $t_{WTR}$ indirectly through `Cmds_T_WtoR_SBG_ok` and `Cmds_T_WtoR_DBG_ok`, which represent the distance between a WR and a RD command to the same or a different bank group, respectively.

[3] Some constraints are represented through multiple POs—this is the case for $t_{WTR}$, $t_{CCD}$, and $t_{RRD}$. In DDR4 devices, these three constraints can admit different values depending on whether commands target the same bank group—which are modelled using distinct POs (`SBG` and `DBG`). Note that DDR3 device can be modelled as having a single bank group. The POs that require different bank groups (`DBG`) then trivially hold, since all commands always target the same (unique) bank group

between the two ACTs, and finally, `Cmds_row_ok` ensures that $ACT_{(bg,bk,r_0)}$ will be between the $PRE_{(bg,bk)}$ and the $CAS_{(bg,bk,r_0,\_)}$.

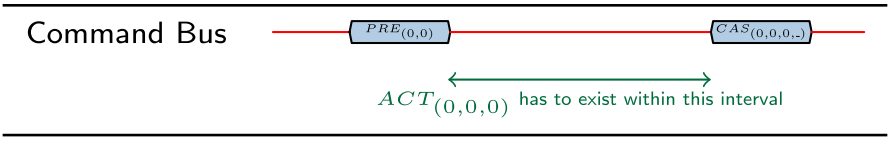


**Fig. 4** Illustration of the `Cmds_row_ok` PO

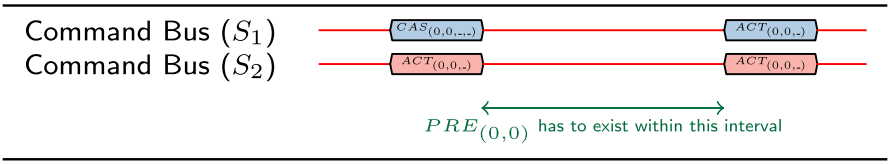


**Fig. 5** Illustration of the `Cmds_ACT_ok` PO

**Table 3** Protocol correctness for a given bank group and a given bank

| 1st | 2nd | | | |
|---|---|---|---|---|
| | $PRE_{bg,bk}$ | $ACT_{bg,bk,r_0}$ | $ACT_{bg,bk,r_1}$ | $CAS_{bg,bk,r_0,\_}$ |
| $PRE_{bg,bk}$ | OK | OK | OK | FORBIDDEN (Covered by `Cmds_row_ok`) |
| $ACT_{bg,bk,r_0}$ | OK | FORBIDDEN (Covered by `Cmds_ACT_ok`) | FORBIDDEN (Covered by `Cmds_ACT_ok`) | OK |
| $ACT_{bg,bk,r_1}$ | OK | FORBIDDEN (Covered by `Cmds_ACT_ok`) | FORBIDDEN (Covered by `Cmds_ACT_ok`) | FORBIDDEN (Covered by the combination of `Cmds_row_ok` and `Cmds_ACT_ok`) |
| $CAS_{bg,bk,r_0,\_}$ | OK | FORBIDDEN (Covered by `Cmds_ACT_ok`) | FORBIDDEN (Covered by `Cmds_ACT_ok`) | OK |

## 5.2 The request arrival model

The previous definitions specify what a *correct* DRAM command trace is. We now specify an interface with cores/requestors. The arrival of requests over time is modelled through the type class `Arrival_function_t` and its member function `Arrival_at` (Listing 11). It yields the set of requests (`Requests_t`) that arrive in the system at a certain time `t`. It also imposes two POs: `Arrival_date` ensures that if $r \in$ `Arrival_at ta`, $r.(Date) \equiv t_a$, i.e., the arrival

function needs to yield the arrival date of requests; `Arrival_uniq` guarantees that requests arriving at a given time `t` are unique.

Listing 11: Arrival Function.

```
Class Arrival_function_t := mkArrivalFunction {
  Arrival_at : nat → Requests_t;

  Arrival_date : forall ta x, (x \in (Arrival_at ta)) → x.(Date) = ta;

  Arrival_uniq : forall t, uniq (Arrival_at t);
}.
```

When writing concrete controller implementations, we do not create any instance of the `Arrival_function_t` class, which means that proofs should hold for any instance of the arrival function. This is an important feature that allows us to build implementations and proofs that are not constrained by any assumption on cores/requestors.

### 5.3 Memory controller model

As it can be seen in Fig. 3, in order to establish the link between the arrival of requests and the DRAM device, i.e., the processing of requests, we specify a memory controller model (Listing 12). The `Controller_t` class is made of the `Arbitrate` function and a PO. The `Arbitrate` function takes a (`nat`) parameter representing the number of commands to be produced and generates a corresponding `Trace_t`, i.e., the list of DRAM commands that have been generated by the controller up to that point.

Listing 12: Memory Controller

```
Class Controller_t {AF : Arrival_function_t} := mkController {
 Arbitrate : nat → Trace_t;

 (* Proof obligation: all requests must be handled *)
 Requests_handled : forall ta req, req \in (Arrival_at ta)
   → exists tc, (CAS_of_req req tc) \in (Arbitrate tc).(Commands);
}.
```

The PO `Requests_handled` ensures that each request that has arrived will eventually have a corresponding CAS command in the trace produced by the controller. It is read as: for a given time instant `ta` and a request `req`, knowing that `req` is an element of the set generated by `Arrival_at ta`, there must exist a timing instant `tc` such that a CAS command belonging to request `req` (`CAS_of_req`) and issued at `tc` is element of the trace generated by the `Arbitrate` function up to `tc`. Along with the `Cmds_ACT_ok` and `Cmds_row_ok` POs, this ensures that every request is **eventually** and **properly** handled.

In Sect. 5, we will furthermore show that for both our implementations, not only `Request_handled` holds, but also that the respective proofs yield closed formulas characterising the instant when a given request completes.

### 5.4 Implementation interface

Finally, in order to establish a systematic and reproducible way to design controllers, we provide what we call an *implementation interface*. It is made of a set of classes and functions used to describe transition systems that are capable of creating traces. The implementation interface is represented in Fig. 3 by the bottom-most arrow.

Designing an implementation pivots around providing an instance for the `Implementation_t` type class (Listing 13), which is made of the functions `Init` and `Next`. These functions operate on a set of currently-arriving requests (`Requests_t`) and implementation-specific controller states (`State_t`). Together, these functions form a state-machine that defines the controllers's request scheduling policy.

Listing 13: Interfaces for controller implementations.

```
Class Implementation_t := mkImplementation {
 (* Init produces an initial state *)
 Init : Requests_t → State_t;
 (* Next produces a new state and a command for a request *)
 Next : Requests_t → State_t → State_t * Command_kind_t * Request_t;
}.

Class Controller_state_t := mkControllerState {
 (* Common to all implementations *)
 Controller_Commands : Commands_t;
 Controller_Time   : nat;
 (* Implementation-specific *)
 Implementation_State  : State_t;
 ...
}.
```

More specifically, the `Init` function is responsible for creating the system's initial state: it takes the first set of currently-arriving requests and generates the first controller state. The `Next` function is responsible for creating every subsequent state, taking a set of currently-arriving requests and a controller state as arguments and producing a triple as result. The triple is made of the new controller state, a `Command_kind_t` (`ACT`, `PRE`, ...), and a `Request_t`. This means that an implementation outputs a new state and a new command at every time stamp, and that a command may belong to a request—though some commands, such as *No-Operations* (NOP)s, do not belong to any request.

The controller state can be described in two abstraction layers. The first one, `State_t`, is operated by the `Init` and `Next` functions and can be used to hold anything needed for implementing the controller's algorithm, such as counters, request queues, et cetera. If we were to think about `Next` as a *Finite State Machine* (FSM) implemented in hardware, the `State_t` would be the set of all signals that

are fed back to the sequential circuit, i.e., the circuit's internal state. Secondly, the `Controller_state_t` type serves as an overlay, containing a `State_t`, the current time instant `Controller_Time`, and `Controller_Commands` – a list with the history of all generated commands up to `Controller_Time`.

Note that `Controller_Commands` and `Controller_Time` have the same types as the members of `Trace_t`. This is because the members of `Trace_t` are constructed exactly from the members of an `Controller_state_t`; more specifically, the trace is built from the last state out of a sequence of generated states.

As a link between the DRAM model and its memory interface, we define the function `Default_arbitrate` (Listing 14). It allows us to reason about the behaviour of the DRAM model and implementation over time, and thus build our proofs—while cleanly separating the formal specification and proofs from the implementation. In practical terms, implemented instances call the `Default_arbitrate` function to generate the traces and to prove POs.

Listing 14: Default behaviour of controllers.

```
Program Fixpoint Default_arbitrate {AF : Arrival_function_t}
       {IM : Implementation_t} t {struct t} : Controller_state_t :=
 let R := Arrival_at t in
 match t with
  | 0     ⇒ mkControllerState [::] t (Init R)
  | S(t') ⇒ (* get controller state at instant t' *)
    let ast := Default_arbitrate t' in
    (* obtain impl. state and next command to issue *)
    let (ist, kind, req) := Next R ast.(Implementation_State) in
    (* append new command to command list *)
    let new_cmd := mkCmd t ckind creq in
    let cmd_list := (new_cmd ::ast.(Controller_Commands)) in
    (* build next controller state*)
    mkControllerState cmd_list t ist
 end.
```

In more detail, the `Default_arbitrate` function produces `t` states by recursion, where t is provided as argument. If $t = 0$, the function builds the first `Controller_state_t` with an empty command list, time stamp equal to 0, and internal state built by the function Init – which itself takes as argument the list of currently-arriving requests, `R` (`Arrival_at` cf. Listing 12). If, however, $t > 0$, the `Next` function is used to build a new state, taking the old state—obtained through a recursive call to `Default_arbitrate`—as an argument. Following that, we build the new command, append it into the command list, and finally build a new `Controller_state_t`. In addition, bear in mind that since `AF` is kept as an implicit parameter of `Default_arbitrate`, it is never instantiated and our proofs hold for all possible arrival functions, as long as they respect their specification and the POs in it. A diagram illustrating the production of states is shown in Fig. 6. The situation depicted in the figure results from a call to `Default_arbitrate t`.

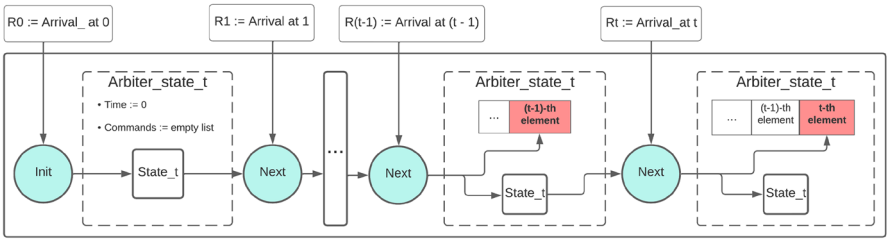


**Fig. 6** Production of states through Default_arbitrate

## 6 Implementations & proofs

We refine our specification with two implementations: one based on the *First-In First-Out* (FIFO) arbitration policy and the other on *Time-Division Multiplexing* (TDM). Both implementations differ in the way they schedule requests, i.e., they implement different *front-end* scheduling policies. As for the back-end and command generation, we implement both controllers with **closed-page** policies, i.e., for any request, we issue the same PRE-ACT-CAS sequence. The generated commands do not go through a new arbitration policy, they are sent to the command bus in the order that they are generated. Neither controller can preempt requests. Moreover, the two implementations are intended to serve as templates for future Real-Time memory controller implementations within the framework, which means that the high-level strategies used for discharging proof obligations are largely re-usable.

In the next subsection, we give implementation details about one of the two, TDM.[13] Then, in Sect. 5.2, as an example of how to solve proof obligations, we present step-by-step the high-level proof strategy used to prove the `Request_handled` PO and bound the worst-case latency for the TDM controller.

### 6.1 TDM implementation

We start by introducing our TDM controller main parameters, characteristics, and hypotheses, all represented in the `TDM_configuration` class, shown in Listing 15: `SN` is the number of slots within a period; we assume that each requestor occupies a slot, in conformance with the *temporal isolation* principle. As an example, if `SN := 3`, the TDM controller periodically serves three requestors in a defined order. `SL` is the length of each individual slot, which has to be big enough to fit the necessary commands to service each request while respecting the timing constraints, i.e., it has to be sufficiently large to fit a PRE, an ACT and a CAS.

[13] We only present details of TDM, since the methodology for FIFO is similar.

Listing 15: `TDM_configuration`.

```
Class TDM_configuration := {
  SN : nat; (* number of requestors/slots in the system *)
  SL : nat; (* TDM slot length in DRAM bus cycles *)

  (* Proof obligations *)
  SN_one : 1 < SN; (* there has to be at least two requestors in TDM *)
  SL_pos : 0 < SL;

  (* The slot length has to be big enough to fit all commands *)
  SL_ACT : ACT_date < SL;
  SL_CASS : CAS_date.+1 < SL;
  T_RAS_SL : T_RP.+1 + T_RAS < SL;
  T_RC_SL  : T_RC < SL;
  T_RRD_SL : T_RRD < SL;
  T_RTW_SL : T_RTW < SL;
  T_CCD_SL : T_CCD < SL;
  WTR_SL   : T_WTR + T_WL + T_BURST < SL;
  T_FAW_3SL : T_FAW < SL + SL + SL
}.
```

While it is easy to see from Listing 15 that the principle of temporal isolation is respected, the spatial isolation is imposed through the Axiom `Private_Mapping`, show in Listing 16. The bank mapping policy does not affect the performance of the FIFO arbitration, but it is important for TDM. We use it to prove the $T_{RTP}$ constraint: since consecutive TDM slots always target different banks, a PRE command can be issued in the cycle following a CAS command—an optimisation w.r.t FIFO.

Listing 16: Private Bank Mapping

```
Axiom Private_Mapping : forall a b, Same_Bank a b →
    TDM_slot (Default_arbitrate b.(CDate)).(Implementation_State) =
    TDM_slot (Default_arbitrate a.(CDate)).(Implementation_State).
```

It is important to remember that, when writing implementations, we do not create instances of the `TDM_configuration` class. This allows us to build the implementations and proofs with arbitrary parameter values. The same is true for the `Arrival_function_t` class and a few others. In a later stage, if one wants to run a simulation, concrete instances have to be provided for all classes, and proof obligations have to be resolved.

Note also that the `TDM_configuration` class contains POs that ensure that the `SL` and `SN` values are consistent with the timing parameters of the DRAM device. Differently than paper-and-pencil proofs, note that this approach allows us to effectively manage assumptions that a given controller relies upon by concisely writing them *all* at the same place.

Concerning the controller's functioning, at each time step it can find itself in one of two possible states: `IDLE` or `RUNNING`.

1. The controller is said to be `IDLE` during a TDM slot if no request is chosen to be serviced in that slot – which can only take place if the requestor that owns the slot does not have any pending request at the beginning of that slot.
2. The controller is in its `RUNNING` state whenever a request is serviced within a TDM slot.

Furthermore, at all times, the TDM controller carries within itself three state variables:

1. A counter of type `Slot_t`, keeps track of slots within a period—its maximum value being SN (`Slot_t` $\in [0, SN[$).
2. A counter of type `Counter_t`, keeps track of cycles elapsed within a slot—its maximum value being SL (`Counter_t` $\in [0, SL[$).
3. A list of pending requests.

If the controller is in the `RUNNING` state, it carries an additional variable used to identify the request currently being serviced.

Figure 7 illustrates a TDM arbitration between three requestors/cores: A, B, and C (`SN:= 3`). The run-time status of the state variables described above is shown in the figure as well. Four requests are present in the system: $r_{0a}$, from requestor A; $r_{0b}$, from requestor B; and $r_{0c}$ and $r_{1c}$, from requestor C. Bear in mind that requests from different requestors target different banks. The controller cannot initially serve $r_{0a}$ because it has arrived too late into the slot to be taken into account. Then, on cycle 9, request $r_{0b}$ is served, followed by request $r_{0c}$. In the next TDM period (starting at cycle 25), $r_{0a}$ is served. The request $r_{1c}$ is finally served in the slot starting at cycle 41.

The controller issues the PRE command on the first cycle of a slot and then issues the ACT and CAS commands at offsets within slots designated by `ACT_date` and `CAS_date` respectively. Those are functions that derive the right offset from the DRAM devices timing parameters. Note that CAS and PRE commands of consecutive slots are issued back-to-back, since, assuming at least two requestors (cf. Listing 15, PO `SN_one`), neighbouring TDM slots always target different banks (private bank mapping). Recall that $t_{WR}$ and $t_{RTP}$ are intra-bank constraints. Moreover, assume that, in the figure, the request queue is empty at the start. The annotations $t_{a_{r1c}}$, $t_{c_{r1c}}$, $t_{x_{r1c}}$, and $t_{k_{r1c}}$ will be introduced and used in Sect. 5.2.

### 6.2 Proving the requests_handled PO for TDM

The proof of the theorem that satisfies the PO `Requests_handled` is organised in steps, written in Coq as `Lemmas`. The key idea is to advance in time step-by-step, starting from the instant $t_a$ when a request arrives in the system, through intermediate steps ($t_c$, $t_x$, and $t_k$), up until the point where a matching CAS is

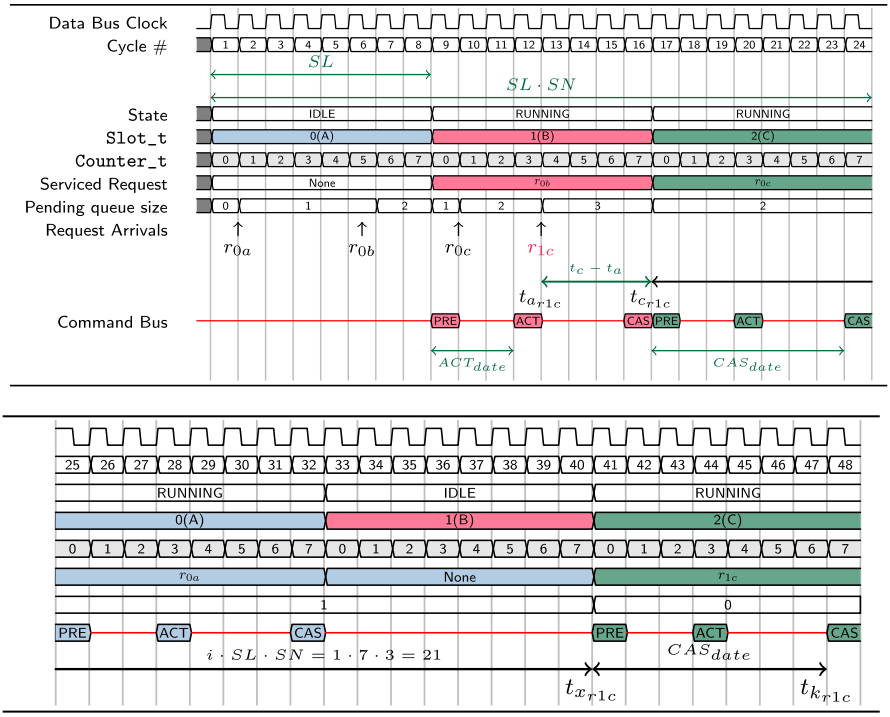


**Fig. 7** TDM arbitration

issued. Figure 7 provides a graphic illustration of these logical steps for request $r_{1c}$.

Concerning the low-level proof strategy, i.e., the strategy for individual steps, we rely heavily on induction (over time or queues), often accompanied by tactics that perform case analysis on the state variables described in the previous section. Although going into details about the low-level proof strategies is out of the scope of this work; we describe, in the following text, most of the high-level proof strategy—thus providing the reader a logical sequence of steps, which can be reproduced in future implementations.

**Definition 1** Let *ra* be an arbitrary request issued by requestor $A$ at instant $t_a$.

*Step 1:* `Pending_on_arrival` *We prove that ra is instantly inserted into the request queue.* (Proven by case analysis on state variables and function unfolding).

**Definition 2** Let $t_c$ be the next instant after $t_a$ when the controller can again decide to service a request from requestor $A$.

*Step 2:* `Pending_requestor_slot_start` *We prove that, once ra is in the pending queue, it stays there until at least* $t_c$. (Induction over time, case analysis on state variables)

**Definition 3** Let $P(t, R)$ be a function that returns an ordered sub-set of the pending queue at time stamp $t$ consisting only of elements issued by a given requestor $R$. Let $i$ be the position of $ra$ in $P(t_c, A)$. Let $t_x$ be equal to $t_c + i \cdot SN \cdot SL$.

*Step 3:* `Request_index_zero` *We prove that if ra is in the pending queue at* $t_c$*, then it will get to the head of* $P(t_x, A)$ *at* $t_x$*, i.e., exactly* $i \cdot SN \cdot SL$ *cycles after* $t_c$. (Induction over $i$)

*Step 4:* `Request_processing_starts` *We use steps 1, 2, and 3 to prove that if ra arrived at* $t_a$*, then it will get to the head of* $P(t_x, A)$ *at* $t_x$ *(Listing* 17*). Note that the proof script consists of using the Coq tactic* `apply`*, which simply makes use of existing lemmas.* (Follows directly from steps 1,2, and 3)

Listing 17: Step 4 – Proof outline of `Request_processing_starts`.

```
Lemma Request_processing_starts ta ra:
   (* Requestor_slot_start calculates tc for ta and ra *)
   let tc := Requestor_slot_start ta ra.(Requestor) in
   (* The controller state at tc *)
   let S  := (Default_arbitrate tc).(Implementation_State) in
   (* P(tc,A): pending queue of requestor A at instant tc *)
   let P  := Pending_of ra.(Requestor) S in
   (* i: position of ra in the pending queue *)
   let i  := index ra P in
   (* tx: instant when processing of ra starts *)
   let tx := tc + i * SN * SL in
   (* The controller state at tx *)
   let S' := (Default_arbitrate tx).(Implementation_State) in
   ra \in Arrival_at ta
 → ra \in (Pending_of ra.(Requestor) S') &&
   (index ra (Pending_of ra.(Requestor) S') = 0).
Proof.
 intros HA. (* HA := ra \in Arrival_at ta *)
 (* Use Step 1 *)
 apply Pending_on_arrival in HA as HP.
 ...
 (* Use Step 2 : Any pending request at least remains
    pending until its requestors slot is reached *)
 apply Pending_requestor_slot_start in HP.
 ...
 (* Use Step 3: Any pending request ultimately gets to the
    head of the pending queue (index zero) *)
 apply Request_index_zero in HP; simpl in HP; exact HP.
Qed.
```

*Step 5.1:* `Request_slot_start_aligned` *We prove that the internal counter (represented by* `TDM_counter` *of type* `Counter_t`*) is equal to 0 at* $t_x$, i.e., $t_x$ *marks the beginning of a slot.* (Follows from modulo arithmetic on `TDM_counter` from Coq's arithmetic libraries)

*Step 5.2:* `Request_starts` *We prove that, for any given t, if ra is the head of P(t, A) and* `TDM_counter` *equals 0, then the request starts to be processed the very next cycle.* (Case analysis on state variables)

*Step 6:* `Request_running_in_slot` *We prove that, for any given t, if a request is being serviced at t and* `TDM_counter` *is equal to 1 at t, then for all d such that* $d < SL - 1$*, the request will remain being served until at least* $t + d$*.* (Induction over *d*)

*Step 7:* `Request_processing` *We use steps 4, 5.1, 5.2, and 6 to prove, for all d smaller than* $SL - 1$*, that ra will be the request currently being processed (represented by* `TDM_request`*) at* $t_x + d$ *(Listing* 18*).* (Follows from previous steps)

Listing 18: Step 7 – Proof outline of `Request_processing`.

```
Lemma Request_processing ta ra:
   let tc := Requestor_slot_Start ta ra.(Requestor) in
   let S  := (Default_arbitrate tc).(Implementation_State) in
   let P  := Pending_of ra.(Requestor) S in
   let i  := index ra P in
   let tx := tc + i * SN * SL in
   ra \in (Arrival_at ta) → forall d, d < SL.-1
 →
   (* The controller state at tx + d *)
   let S':=(Default_arbitrate (tx + d)).(Implementation_State) in
   (* The conclusion *)
   (TDM_counter S' = d.+1) && (TDM_request S' = ra)
Proof.
 ...  (* Applies lemmas described in steps 4,5.1,5.2 and 6 *)
Qed.
```

**Definition 4** Let $t_k$ be $t_x + CAS\_date$, where *CAS_date* is the CAS command offset within a TDM slot (cf. Figure 7).

*Step 8.1:* `Request_CAS` *We use step 7 with d equal to CAS_date to prove that ra will have its CAS issued at* $t_x + CAS\_date$ *(Listing* 19*).* (Application of step 7, unfolding, and arithmetic simplification).

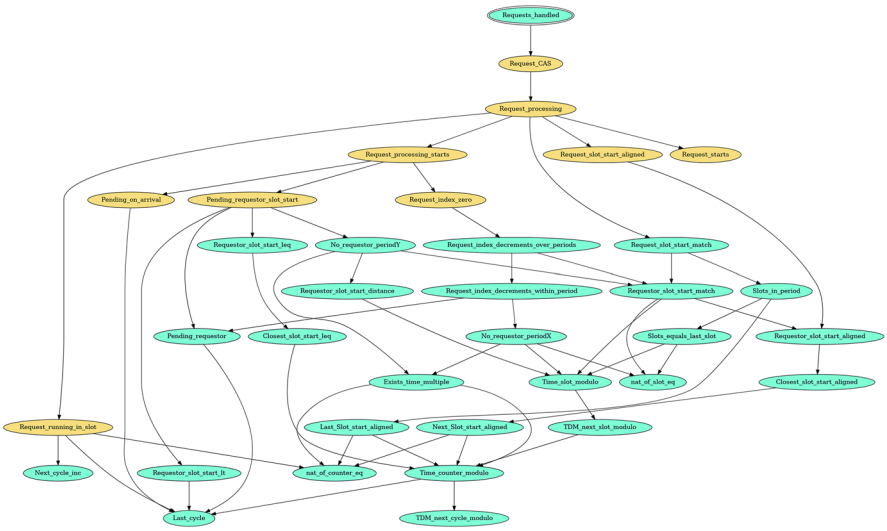


**Fig. 8** Proof structure: `Requests_handled`

Listing 19: Step 8.1 – Proof outline of `Request_CAS`.

```
Lemma Request_CAS ta ra:
 let tc := Requestor_slot_Start ta ra.(Requestor) in
 let S  := (Default_arbitrate tc).(Implementation_State) in
 let i  := index ra (Pending_of ra.(Requestor) S) in
 let tk := tc + i * SN * SL + CAS_date in
 ra \in (Arrival_at ta) →
 (CAS_of_req ra tk) \in (Default_arbitrate tk).(Controller_Commands)
Proof.
 intros HA. (* HA := ra \in Arrival_at ta *)
 (* Use Step 7 :
 any request is processed until the CAS date is reached *)
 apply Request_processing with (d := CAS_date) in HA as HR.
 ...
Qed.
```

*Step 8.2:* `Requests_handled` *We prove the theorem that satisfies the final PO* (Follows from step 8.1).

Figure 8 shows an automatically-generated graphical representation of the proof. For the sake of conciseness, we only show the high-level lemmas relevant for the proof strategy discussed above. The steps discussed above are highlighted in the graph.

We would like to emphasise that, as it can be seen in Listing 19, $t_k$ is the typical closed-form expression one would expect for TDM. The formula is made of three parts:

1. `tc`, which is bounded by $t_a + SL - 1$;
2. $i \cdot SN \cdot SL$, which is bounded by the number of pending requests of the respective requestor;
3. $CAS\_date$, which is constant.

A timing analysis tool may now easily derive correct latency bounds for our TDM implementation by evaluating this formula. For this it would need to supply the correct TDM configuration (`SL` and `SN`) and establish bounds on $i$ through an additional model of the requestor (i.e., by modelling the arrival function).

For example: considering a DDR4-2400U device (c.f Table 1), the minimum possible value of $SL$ is 40, as it can be seen from Eq. 1 (as determined by PO `SL_CASS` in Listing 15).

$$\begin{aligned} ACT_{date} &:= T_{RP} + 1 \\ CAS_{date} &:= ACT_{date} + (T_{RCD} + 1) = (T_{RP} + 1) + (T_{RCD} + 1) = 38 \\ SL &> CAS_{date} + 1 \\ SL &> 39 \end{aligned} \tag{1}$$

Listing 20: From implementations to instantiating the controller.

```
(* Creates a proved trace from the implementation *)
Definition TDM_arbitrate t :=
    mkTrace (Default_arbitrate t).(Controller_Commands)
            (Default_arbitrate t).(Controller_Time)
            (Cmds_T_RRD_ok t) (* proof *)
            (Cmds_T_FAW_ok t) (* proof *)
            ...  (* all the other proofs *)

(* Instantiate the TDM controller *)
Instance TDM_controller : Controller_t :=
    mkController AF TDM_arbitrate Requests_handled.
```

Consider now four cores ($SN = 4$) that can only issue two outstanding memory requests at a time (which implies that the maximum value of $i$ is 1). The worst-case latency $WCL$ is then given by Eq. 2:

$$\begin{aligned} WCL &= (SL - 1) + i \cdot SL \cdot SN + CAS_{date} \\ &= 39 + 1 \cdot 40 \cdot 4 + 38 = 237 \end{aligned} \tag{2}$$

Even more, analyses that are themselves formalised in Coq could simply reuse our proofs in order certify memory latency bounds.

### 6.3 Putting it all together

Finally, in Listing 20, we show the code that instantiates the trace through the constructor `mkTrace`, defines the proven arbitration function `TDM_controller`, and finally instantiates the controller (using `mkController`, cf. Listing 12). The

command trace instance is only accepted by Coq if all proofs are provided as arguments to the constructor, which means that the implementation is correct. In the listing, we show that `Cmds_T_RRD_ok t` and `Cmds_T_FAW_ok t` are provided as arguments. These are the actual proofs that satisfy the $t_{RRD}$ and $t_{FAW}$ constraints, respectively. The command trace itself is obtained from the last state produced by the `Default_arbitrate` function. Similarly, the `Controller_t` instantiation is only accepted when a proof for the `Requests_handled` PO is provided. Even for someone not familiar with Coq, it is thus trivial to verify the correctness of the controller implementation.

## 7 Modelling semantics

The expressivity provided by Coq allows us to model other interesting properties about DRAM controllers. Notably, we can write classes that define certain semantics, e.g., guarantees on the possible order in which requests are handled. To showcase this, we model two flavors of Lamport's definition of *Sequential Consistency* Lamport (1979) (SC). According to Lamport, sequential consistency in a multiprocessor system is achieved when two requirements are met:

- *Requirement* **R1**: Each processor issues memory requests in the order specified by its program.
- *Requirement* **R2**: Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of entering the request into this queue.

Requirement R1, on the one hand, is an assumption on the behaviour of processors, and is therefore not modelled from the memory controller's point of view. This makes sense, considering that a memory consistency model can be seen as a contract between software/programs and the hardware, and is conceptually implemented by both. Requirement R2, on the other hand, should be implemented by the memory controller.

Moreover, Lamport defines a relaxed version of R2 that still guarantees SC: *"We need only require that all requests to the same memory cell be serviced in the order that they appear in the queue."* This relaxed version of R2 comes from the observation that actually **only memory accesses to the same address** can introduce incoherence w.r.t the order of execution between cores, and therefore, a FIFO order of execution should be guaranteed only between accesses to the same addresses. We emphasise that Lamport's definitions are seen today as sufficient conditions, and a more formal definition of SC was introduced by Sezgin (2004).

In practical terms, we define two classes in our framework that model requirement R2 as proof obligation `R2` and its relaxed version as proof obligation `R2_relaxed`, respectively, as shown in Listing 21.

Listing 21: Modelling Controller Semantics.

```
Class SequentialConsistent_Controller {AF : Arrival_function_t} {AR : Arbiter_t}
     := mkSeqController {

    R2 : forall ta reqa tb reqb,
    reqa \in (Arrival_at ta) → (* reqa arrives at ta *)
    reqb \in (Arrival_at tb) → (* reqb arrives at tb *)
    (* either reqa arrived before reqb OR
       they arrived at the same instant, but there
       is an arbitrary order between reqa and reqb,
       and reqa is to be serviced before *)
    (ta < tb) ∨
    (ta = tb ∧ index reqa (Arrival_at ta) < index reqb (Arrival_at ta))
    (* txa, the completion date of reqa must happen before txb,
       the completion date of reqb *)
    → exists txa txb, (CAS_of_req reqa txa \in (Arbitrate txa).(Commands))
    && (CAS_of_req reqb txb \in (Arbitrate txb).(Commands)) && (txa < txb)
}.

Class W_SequentialConsistent_Controller {AF : Arrival_function_t} {AR :
     Controller_t} := mkWSeqController {

    R2_relaxed : forall ta reqa tb reqb,
    reqa \in (Arrival_at ta) → (* reqa arrives at ta *)
    reqb \in (Arrival_at tb) → (* reqb arrives at tb *)
    (ta < tb) ∨
    (ta = tb ∧ index reqa (Arrival_at ta) < index reqb (Arrival_at ta))
    (* Here, an additional pre-condition: reqa and reqb target the same row *)
        → reqa.(Row) = reqb.(Row) →
    (* txa, the completion date of reqa must happen before txb,
       the completion date of reqb *)
    exists txa txb,
    (CAS_of_req reqa txa \in (Arbitrate txa).(Commands)) &&
    (CAS_of_req reqb txb \in (Arbitrate txb).(Commands)) && (txa < txb)
}.
```

Note that, in both `R2` and `R2_relaxed`, two situations are possible: either `ta = tb`, which means that the two requests have arrived at the same time; or `ta < tb`, which means that `reqa` has arrived before `reqb`. This arrival order of requests assumes that the condition R1 has already been satisfied, i.e., if `reqa` and `reqb` are requests issued by the same requestor, then `reqa` was issued before `reqb`. In the case where `ta = tb` (which could happen in a multi-core system), the arrival order to be considered is given by the positions of both `reqa` and `reqb` in the ordered set `Arrival_at ta`, as a real implementation would indeed impose some order on requests that have arrived on the same cycle. In the listing, the position in the ordered set is given by the function `index`.

The PO `R2_relaxed` is a relaxed version of `R2`, in which the condition "requests targeting the same cell" is described through the logical proposition `reqa.(Row) = reqb.(Row)`. It follows that only controllers respecting the conditions set by the POs in the classes can instantiate them. Considering the two controllers implemented in this work, while FIFO respects both constraints, TDM can only instantiate the weakened version, `W_SequentialConsistent_Controller`. This is

**Table 4** Size of the code & Compilation time

| | Lines of code | Compilation time (s)[a] |
|---|---|---|
| Specification | 554 | 2.34 |
| FIFO implementation | 107 | 0.32 |
| FIFO proofs | 2515 | 9.63 |
| TDM implementation | 224 | 0.45 |
| TDM proofs | 2221 | 15.44 |

[a]Results obtained on a system with the following configuration: CPU—Intel(R) Core(TM)i5-10210U CPU 1.60GHz; Memory – 8GiB; Operating System—Ubuntu 20.04.3 LTS; Coq Version—8.13.2

because TDM (under private bank mapping), has the ability to prioritise a request that has arrived later than another request issued by another processor.

Furthermore, other memory consistency models, such as TSO and linearizability, could be modelled likewise. This feature distinguishes our contributions, since state-of-the-art RT memory controllers have virtually ignored the semantics aspects. Other work that aim at using formal methods to model DRAM properties (c.f Sect. 2.2) focus mainly on timing properties. The fact that we can straightforwardly extend our framework to model (which does not necessarily imply that proofs will follow straightforwardly) this kind of property is a statement of how powerful this methodology can be.

## 8 Evaluation & simulation

In the following, Sect. 7.1 evaluates our approach regarding size and compilation time and gives an additional insight on using our framework. Then, Sect. 7.2 explains how we extracted code from our framework and used it to integrate our approach into an external DRAM software simulator.

### 8.1 Evaluation

Table 4 presents the code size and the compilation times for the specification files, both implementations, and the proof scripts. As it can be seen, the implementations themselves are small compared to the specification and the proofs. The proofs are in the order of 10 to 25 times longer than the implementations. [14] As proofs are checked by Coq's kernel, compiling the files containing the proofs also takes more time than files containing simply code.

[14] As stated by Boldo et al. (2017), Coq proofs are usually as long as paper and pencil proofs, which means that automatising the proof process with Coq is comparable to manual proofs (concerning length).

Furthermore, the set of hypothesis used for the proofs form an exploitable mathematical model by itself. This set of hypothesis is made of formulas establishing linear inequalities between the controller parameters and the timing constraints, which can be fed to a *Linear Programming* (LP) solver in order to obtain optimal values for the controller parameters. Then, when used to concretely instantiate the controller, these optimal values are checked by Coq's engine—serving as a validity check for parameters values.

### 8.2 Code generation & simulation

External environments/frameworks can interact with our model through calls to the `Init` and `Next` functions. Because these functions together define a transition system, which is essential to hardware design, they can be embedded into foreign code, such as software simulators and hardware. To showcase this feature, we choose MCsim Mirosanlou et al. (2020), a cycle-accurate DRAM simulator written in C++ to host our proved code. To establish an interface between both, we use Coq's extraction feature, which can output code in different programming languages. We opt for Haskell, since it has a well-documented *Foreign Function Interface* (FFI)[15] library that allows Haskell programs to cooperate with C/C++ programs[16].

We use MCsim's trace reading function to model request arrivals, which fetches a new request after completing the preceding one. Moreover, the simulator provides a class to implement controllers, which contains a method `update`, called each clock cycle to update the internal state of the controller. This method makes a call to the `requestSchedule` method followed by a call to `commandSchedule`. According to pre-defined arbitration policies, the former fetches requests from the pending queues, generates commands and puts them into a buffer. The latter chooses one of the commands in this buffer and sends it to the device. Since our model performs all of these steps, we replace the calls to `requestSchedule` and `commandSchedule` with calls to `Init` (for the first cycle) and `Next`. In addition, we encapsulate the generated code inside Haskell wrapper functions to properly handle data on both sides: states are passed back and forth between C++ and Haskell through opaque pointers, and other data structures, such as requests and commands, are converted with the aid of *hsc2hs*,[17] a preprocessor that helps with writing Haskell bindings to C.

In order to validate our framework, we run the two traces that come with MCsim: one made of requests accessing sequential addresses and the other made of random ones. Every requestor executes the same trace—but are nevertheless forced to be mapped to different banks. The value of the `SL` TDM parameter is chosen to be the minimum possible through the methodology described in the end of Sect. 7.1. For the DDR3-2133N device, for instance, this value is equal to $t_{RP} + t_{RCD} = 28$. We

[15] https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/ffi.html.

[16] The generated Haskell code is compiled with *ghc* (version 8.10.4). Then, MCsim is compiled with g++ (version 9.4.0) and linked with ghc.

[17] https://hackage.haskell.org/package/hsc2hs.

**Table 5** Simulation results for sequential and random traces

| | FIFO | TDM | AMC | ROC | FRFCFS |
|---|---|---|---|---|---|
| Sequential trace | | | | | |
| Bandwidth (MB/s) | 433.89 | 799.97 | 609.536 | 5296.08 | 5931.93 |
| Requests completed (per requestor, on average) | 4237 | 7812 | 5952 | 51720 | 57929 |
| Maximum observed latency (cycles) | 236 | 136 | 168 | 41 | 23 |
| Random trace | | | | | |
| Bandwidth (MB/s) | 433.89 | 799.97 | 609.536 | 4995.1 | 4995.1 |
| Requests completed (per requestor) | 4237 | 7812 | 5952 | 48780 | 48780 |
| Maximum observed latency (cycles) | 236 | 136 | 168 | 25 | 25 |
| Average simulation time[a] (s) | | | | | |
| | 11.13 | 21.18 | 0.41 | 0.85 | 0.96 |

MCsim setup: 4 Requestors, 1 Channel, 1 Rank, DDR3 2133N 2Gb_x8 device, Private Banks, In-order cores, 1000000 cycles

[a]Values were obtained with the same computer setup described in the previous section

choose the value of FIFO's `WAIT`—a fixed delay to process requests, like TDM's `SL`—similarly.

As a first observed result, *the simulations follow through for both TDM and FIFO controllers*. This validates our specification, since MCsim's simulation stalls if timing constraints are not respected or incoming requests are not served at some point.

Table 5 compares the simulation output with other known DRAM controllers (AMC Paolieri et al. (2009), ROC Xin et al. (2019), and FR-FCFS Rixner et al. (2000))—our results are highlighted. Note that, for the described setup, our controllers provide competitive bandwidth compared to the real-time controller AMC. As expected, the bandwidth is smaller than that of high-performance controllers (ROC and FRFCFS). It can be seen that these high-performance controllers exhibit fluctuations on the maximum observed latency depending on the trace format (41 to 25 and 23 to 25 respectively), while the real-time controllers offer constant values, no matter the input. Moreover, the maximum observed latency values are consistent with the ones presented in related work (c.f Figure 14 on Ecco and Ernst (2015)). As a setback, since our integration with MCsim relies on a complex Haskell-C++ interaction, the simulation is considerably slower.

Bear in mind that only two things impact the simulation time and scalability: 1) The size of the input trace, i.e., how many requests arrive in the system. 2) The timing parameters of the device to be simulated. As faster devices (in terms of frequency) have larger timing constraints (in terms of clock cycles), more calls to MCsim's simulation function are needed, thus resulting in longer simulations.

We emphasise that for this paper, our goal is not to propose high-performance and/or high-bandwidth implementations, as the closed-page strategy together with FIFO and TDM arbitration policies results in rather simple and relatively slow controllers. Our integration with MCsim serves as a proof of concept for our specification, and using our framework to model competitive real-time controllers is listed as a part of our future work plans.

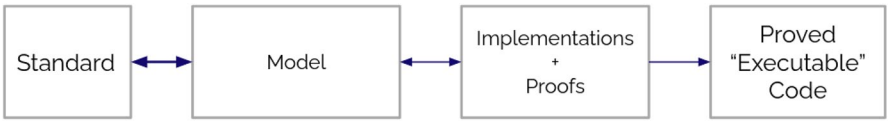


**Fig. 9** Design Flow with the Coq Framework

## 9 Why is it more trustworthy?

The framework provides a *trustworthy design path* from end to end. The design flow proposed by the framework is presented through a diagram in Fig. 9. In other words, we claim that the trust in every link in that diagram is increased, compared to other real-time hardware design methodologies.

First, consider the link between the JEDEC standards and the Coq-written model (leftmost arrow). The language used to write our specification is very close to the language used in the standard itself, which is a mixture of graphical and natural language. Take, for instance, the listings shown throughout the paper: *proof obligations* are formulated through propositions written in a mixture of first and higher-order logic and do resemble natural language. It is relatively easy to be convinced that our *proof obligations* do capture the actual behaviour described in the JEDEC standards. Other state-of-the-art work that implement some formal approach w.r.t DRAM problem rely on complex mathematical representations to represent the same properties. As comparison, consider the UPPALL finite state-machines proposed by Li et al. (2016), used to model the very same properties. While we do not claim that the representation is inherently harder, we do state that there is a larger gap between the languages used in the model and the standard; and this gap can lead to poor modelling.

Second, consider the link between specification (model) and implementation/proofs. From the fact that proofs and programs are essentially the same, many benefits follow:

1. Proofs are machine-checked by Coq's kernel and therefore more trustworthy.
2. Every single hypothesis used for proving obligations must be made explicit and grouped together in classes—which allows better management, since hypotheses are no longer scattered throughout the paper.
3. Proofs are complete, since hand-waving and oversimplifying will not work in Coq.
4. There is no trust gap coming from different representations of model and proofs. In other words, paper-and-pencil mathematical abstractions about models can introduce another language gap and lead to poor modelling.
5. Hardware designs can hide the underlying mathematical formalism of timing analyses from the reader. Differently put, if a new hardware component is introduced within such a framework, the designer can present it without giving details about timing analyses, by just saying that all the proof obligations have been successfully satisfied. Hence, the job of reviewers and readers then switches

from checking if paper-and-pencil proofs are correct to merely checking if the properties captured by the specification are correct (which leads us back to the first point).

Lastly, concerning the last and right-most blue arrow in Fig. 9, Coq provides extraction to many target languages. This feature allows proved executable code to be produced as output and serve as sources of trust in external software/hardware.

## 10 Conclusion & future work

We propose a new way of designing Real-Time Memory Controllers. We write a formal specification that defines the properties of interest as *proof obligations*. We focus on properties related to the respect of timing constraints, correctness of the command protocol, and assertiveness that every request is handled in bounded time. We refine our specification with two implementations (in which we write proofs for each proof obligation in the specification). We validate our specification through execution on a simulator and compare simulation results with other known controllers.

The follow-up to this work consists in using Cava (c.f Sect. 2.3) to produce HDL code directly from the framework. The generated HDL code will then be used inside of a DDR4 hardware simulation environment. After this step is finished, many directions are actually possible, the most promising being: modelling the remaining properties described in the JEDEC standards (such as DRAM refresh operations, power-down modes, et. cetera) with the goal of achieving a more complete specification; implement state-of-the-art RT MCs within the framework; capturing other types of properties, e.g., imposing security-related counter-measures as proof obligations; and writing tactics to achieve higher proof automation.

**Acknowledgements** This research was supported by Labex DigiCosme (Project ANR11L-ABEX-0045DIGICOSME) operated by ANR as part of the program "Investissement d'Avenir" Idex ParisSaclay (ANR11IDEX000302).



## References

Akesson B, Nasri M, Nelissen G, Altmeyer S, Davis RI (2020) An empirical survey-based study into industry practice in real-time systems. In: 2020 IEEE Real-Time Systems Symposium (RTSS). IEEE, pp 3–11

Bhat B, Mueller F (2011) Making DRAM refresh predictable. Real-Time Syst 47(5):430–453. https://doi.org/10.1007/s11241-011-9129-6

Bjesse P, Claessen K, Sheeran M, Singh S (1998) Lava: hardware design in haskell. ACM SIGPLAN Notices 34(1):174–184

Boldo S, Clément F, Faissole F, Martin V, Mayero M (2017) A Coq formal proof of the lax-milgram theorem. In: Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, pp 79–89

Bourgeat T, Pit-Claudel C, Chlipala A (2020) The essence of bluespec: a core language for rule-based hardware design. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, pp 243–257

Bozhko S, Brandenburg BB (2020) Abstract response-time analysis: A formal foundation for the busy-window principle

Cerqueira F, Stutz F, Brandenburg BB (2016) Prosa: A case for readable mechanized schedulability analysis. In: Euromicro Conference on Real-Time Systems, ECRTS'16. IEEE, pp 273–284

Chlipala A (2022) Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant. MIT Press, Cambridge

Choi J, Chlipala A, et al (2022) Hemiola: A DSL and verification tools to guide design and proof of hierarchical cache-coherence protocols. In: International Conference on Computer Aided Verification. Springer, pp 317–339

Choi J, Vijayaraghavan M, Sherman B, Chlipala A, Arvind (2017) Kami: aplatform for high-level parametric hardware specification and its modular verification. Proc ACM Prog Language 1:1–30

Ecco L, Ernst R (2015) Improved DRAM timing bounds for real-time DRAM controllers with read/write bundling. In: Real-Time Systems Symposium, RTSS'15. IEEE, pp 53–64

Guo D, Hassan M, Pellizzoni R, Patel H (2018) A comparative study of predictable DRAM controllers. ACM Trans Embed Comput Syst (TECS) 17(2):1–23

Guo D, Pellizzoni R (2017) A requests bundling DRAM controller for mixed-criticality systems. In: Real-Time and Embedded Technology and Applications Symposium, RTAS'17. IEEE, pp 247–258

Guo X, Lesourd M, Liu M, Rieg L, Shao Z (2019) Integrating formal schedulability analysis into a verified os kernel. In: Computer Aided Verification, CAV'19. Springer, pp 496–514

Hassan M, Patel H (2017) Mcxplore: Automating the validation process of DRAM memory controller designs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 37(5):1050–1063

Hassan M, Patel H, Pellizzoni R (2015) A framework for scheduling DRAM memory accesses for multi-core mixed-time critical systems. In: Real-Time and Embedded Technology and Applications Sym. IEEE, pp 307–316

Jalle J, Quinones E, Abella J, Fossati L, Zulianello M, Cazorla FJ (2014) A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study. In: Real-Time Systems Symposium, RTSS'14. IEEE, pp 207–217

(JEDEC), J.E.D.E.C.: DDR4 SDRAM standard (2021)

Jung M, Kraft K, Soliman T, Sudarshan C, Weis C, Wehn, N (2019) Fast validation of DRAM protocols with timed petri nets. In: International Symposium on Memory Systems, MEMSYS'19. ACM, pp 133–147

Lamport L (1979) How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans Comput C 9:690–691

Larsen KG, Pettersson P, Yi W (1997) Uppaal in a nutshell. Int J Softw Tools Technol Transf 1(1):134–152

Li Y, Akesson B, Goossens K (2014) Dynamic command scheduling for real-time memory controllers. In: Euromicro Conference on Real-Time Systems, ECRTS'14. IEEE, pp 3–14

Li Y, Akesson B, Lampka K, Goossens K (2016) Modeling and verification of dynamic command scheduling for real-time memory controllers. In: Real-Time and Embedded Technology and Applications Symposium. IEEE, pp 1–12

Lisboa Malaquias F, Asavoae M, Brandner F (2022) A Coq framework for more trustworthy DRAM controllers. In: Proceedings of the 30th International Conference on Real-Time Networks and Systems, pp 140–150

Maida M, Bozhko S, Brandenburg BB (2022) Foundational response-time analysis as explainable evidence of timeliness. In: 34th Euromicro Conference on Real-Time Systems (ECRTS 2022). Schloss Dagstuhl-Leibniz-Zentrum für Informatik

Mirosanlou R, Guo D, Hassan M, Pellizzoni R (2020) Mcsim: An extensible DRAM memory controller simulator. IEEE Comput Archit Lett 19(2):105–109
Mirosanlou R, Hassan M, Pellizzoni R (2020) Drambulism: Balancing performance and predictability through dynamic pipelining. In: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). IEEE, pp 82–94
Mirosanlou R, Hassan M, Pellizzoni R (2021) Duomc: Tight DRAM latency bounds with shared banks and near-cots performance. In: The International Symposium on Memory Systems, pp 1–16
Nipkow T, Paulson LC, Wenzel M (2002) Isabelle/HOL: a proof assistant for higher-order logic, vol 2283. Springer, Berlin
Paolieri M, Quinones E, Cazorla FJ, Valero M (2009) An analyzable memory controller for hard real-time cmps. IEEE Embed Syst Lett 1(4):86–90
Reineke J, Liu I, Patel HD, Kim S, Lee EA (2011) PRET DRAM controller: Bank privatization for predictability and temporal isolation. In: Conference on Hardware/Software Codesign and System Synthesis. IEEE, pp 99–108
Rixner S, Dally WJ, Kapasi UJ, Mattson P, Owens JD (2000) Memory access scheduling. ACM SIGARCH Comput Archit News 28(2):128–138
Schranzhofer A, Pellizzoni R, Chen JJ, Thiele L, Caccamo M (2011) Timing analysis for resource access interference on adaptive resource arbiters. In: Real-Time and Embedded Technology and Applications Symposium, RTAS'11. IEEE, pp 213–222
Sezgin A (2004) Formalization and verification of shared memory. The University of Utah
Shankar N (2018) Combining model checking and deduction. In: Handbook of Model Checking. Springer, pp 651–684
Sørensen MH, Urzyczyn P (2006) Lectures on the Curry-Howard isomorphism. Elsevier, Amsterdam
Valsan PK, Yun H (2015) Medusa: a predictable and high-performance DRAM controller for multicore based embedded systems. In: 2015 IEEE 3rd international conference on cyber-physical systems, networks, and applications. IEEE, pp 86–93
Wu ZP, Krish Y, Pellizzoni R (2013) Worst case analysis of DRAM latency in multi-requestor systems. In: Real-Time Systems Symposium, RTSS'13. IEEE, pp 372–383
Xin X, Zhang Y, Yang J (2019) Roc: DRAM-based processing with reduced operation cycles. In: Proceedings of the 56th Annual Design Automation Conference 2019, pp 1–6
Yun H, Pellizzon R, Valsan PK (2015) Parallelism-aware memory interference delay analysis for cots multicore systems. In: Euromicro Conference on Real-Time Systems, ECRTS'15. IEEE, pp 184–195



**Felipe Lisboa Malaquias** is a PhD candidate at Télécom Paris from late 2020. Before starting the preparation of his PhD thesis, he completed his Master's degree in Computer Science in Télécom Paris and Université Paris-Saclay and his Bachelor's degree in Electrical Engineering at Universidade Federal de Minas Gerais, in Brazil. Since the late stride of his Masters, Felipe has gained interest in the problems of Real-Time Systems, culminating in his thesis, which looks at memory controllers design for mixed-criticality systems and formal reasoning. After starting his PhD, he published in three different venues: RTNS, in June 2022; Nasa Formal Methods, in May 2023; and Ada Europe, in June 2023. Besides research, Felipe has also performed teaching activities, providing lab support for Master's courses. In addition, Felipe has also had the chance to do some research in industry: once in 2020, when he worked with OS development in a secure environment; and again in 2023, when he worked with formal methods to design trustworthy RISC-V architectures. Moreover, Felipe is also fluent in Portuguese, French, English, and Spanish. He also has a good level of German and Italian.

**Mihail Asavoae** is a R&D engineer in CEA LIST, Paris since Dec 2017. He obtained his PhD from UAIC, Iasi in 2012 and worked pre- and post-thesis in several international universities and reasearch centers: NUS, in Singapore from 2002-2008, Verimag Lab, in Grenoble from 2013-2015 and INRIA Paris from 2015-2017. He also had short-term scientific stays in UCM, Madrid and in Saarland University, Saarbruecken. Mihail's research interests are on the design and analysis of safety-critical systems using formal verification, in particular on timing predictability-related problems (e.g. worst-case execution time analysis, detection of timing anomalies, interference analysis etc.). He also works on formal verification of security properties in HW/SW systems. Mihail is fluent in Romanian, English and French and has basic notions of German.

**Florian Brandner** has been an Associate Professor at Télécom Paris (Institut Polytechnique de Paris) since 2015. He received his Diploma and PhD in 2004 and 2009 from the Vienna University of Technology. He then worked as a post-doctoral researcher at ENS de Lyon and the Technical University of Denmark, before joining ENSTA ParisTech as an Assistant Professor in 2013. His research focuses on compilation and analysis techniques as well as time-predictable computer architecture design in the context of embedded real-time systems.