INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2024IPPAT020

Thèse de doctorat

CoqDRAM – A Foundation for Designing Formally Proven Memory Controllers

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom Paris

École doctorale n° 626
École doctorale de l'Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat : Information, communications, électronique

Thèse présentée et soutenue à Palaiseau, le 21 juin 2024, par

MR. FELIPE LISBOA MALAQUIAS

Composition du Jury :

Sylvie Putot Professeure, Laboratoire d'informatique de l'École polytechnique (LIX), École Polytechnique	Président Examinatrice
Rolf Ernst Professeur, Institute of Computer and Network Engineering, Technische Universität Braunschweig	Rapporteur
Claire Pagetti Directrice de recherche, Office national d'études et de recherches aérospatiales (ONERA)	Rapporteur
Clément Pit-Claudel Professeur assistant, EPFL	Examinateur
Lirida Naviner Professeure, LTCI, Télécom Paris	Directrice de thèse
Florian Brandner Maître de conférences, LTCI, Télécom Paris	Co-directeur de thèse
Mihail Asavoae Chercheur, CEA List	Invité
Sumantha Chaudhuri Maître de conférences, LTCI, Télécom Paris	Invité

Dedicated to Romildo and Neli, my first teachers.

Acknowledgments

There are several people without whom this thesis would never have seen the light of day and I am beyond grateful to all of them.

First of all, I would like to thank my advisors: Florian Brandner and Mihail Asavoae. Whenever master students enquire me about doing a PhD, I tend to say something in the lines of: "it depends a lot on who will advise you". Indeed it does, and I certainly hit jackpot. Since day one, Florian has given me nothing but great advice. He has a brilliant and creative mind, and the amount of knowledge I gained from our discussion over the last years is immeasurable. Our discussions went beyond very niche technical topics, often touching subjects such as philosophy and science ethics. Besides being a bright scientist, he is an even better human being, who treats everyone with the uttermost respect and empathy. As as a PhD candidate, you cannot ask for more than that. So, Florian, I thank you for teaching me so much about technical subjects but also about being a great person, and of course, for your great jokes and *interesting* musical recommendations. Second, but certainly not least, Mihail's presence never wavered. He always found a way, even through hard times, to make himself available, whether it is to give me insightful technical feedback or to be a friend and talk about life. Mihail, thanks for teaching me so much about formal methods, how to have such a positive outlook on life, and how to lighten the mood with a timely joke.

Next, Lirida Naviner also deserves a big amount of gratitude. She has accompanied me throughout all times, always ready to help and do whatever is necessary to make sure that things went smoothly. I also thank Sumanta Chaudhuri, who never ceased to give me sharp feedback and helped me set up the hardware simulation environment for one of the validation experiments. Laurent Pautet and Tarik Graba were also always ready to help – I am grateful for their support throughout these years. Philip Harris, Stylianos Basagiannis, and Raúl de la Cruz were also great mentors to me, teaching me much about the applications of formal methods and real-time systems in the aviation sector.

Moreover, I profoundly believe that being able to have fun and let yourself get distracted is paramount to achieve success. Often times, when the cognitive load of writing proofs are

developing new algorithms gets overwhelming, few things help to get over the hump as much as a nice pause. I do not know how many times I have been able to solve a difficult problem only after allowing myself one such pause. For that, I have to thank my friends and fellow PhD candidates Franco, João, Luciano, and Andrey for the table tennis matches; Maxime, Thomas, and Petr for the *jam* sessions at *La Scene*; and Dominik for the pleasant coffee breaks and language *tandems*. Thanks to you, I started describing my days at Télécom not only as being productive, but also *fun*.

I could not go forward without expressing sincere gratitude to all of my dear friends. Thank you for celebrating with me when I accomplished goals, supporting me when times were rough, and just being there when times were *ordinary*.

I would also like to thank my dear partner, Carolin. Your smile and joy make tough days much brighter and happy days even more cheerful. Thanks for your friendship and patience, for listening to my presentations, for giving me all sorts of feedback and words of encouragement, and more than anything, for always being there for me. This would have been incredibly harder without you. I am also grateful to Ingrid, Andreas, and Sophia for warmly embracing me and inspiring me to not take things so seriously.

Of course, none of this would have been possible without the support of my family. To my parents, José Romildo and Neli, there are hardly enough words. Thank you for raising me the way you did and for teaching me to ask questions. You are my highest role models. To my sisters, Ana Carolina and Luíza, thank you for making me a better person. To Lucas, my brother-in-law, and to all of my uncles, aunts and cousins, thank you for your constant support.

CoqDRAM – A Foundation for Designing Formally Proven Memory Controllers

by

Felipe Lisboa Malaquias

Submitted to the EDIPP Paris
on April 21, 2024 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY
IN
COMPUTER ENGINEERING

ABSTRACT

Recently proposed real-time memory controllers tackle the performance-predictability trade-off by trying to offer the best of both worlds. The common underlying idea is to optimise bandwidth and average-case latency while still being able to precisely characterise the system's performance and derive worst-case latency upper bounds for memory requests. However, as a consequence, designs have become complex and often present mathematical developments that are lengthy, hard to read and review, incomplete, and rely on unclear assumptions. Given that such components are often designed as part micro-architectures that are used in safety-critical real-time systems, a high degree of confidence that systems behave *correctly* is required in order to meet certification goals.

To address that problem, we propose a new framework written in the Coq theorem prover named *CoqDRAM*, in which we model DRAM devices and controllers and their expected behaviour as a formal specification. The framework is intended to aid the design of correct-by-construction, *trustworthy* DRAM scheduling algorithms. The CoqDRAM specification captures correctness criteria according to the JEDEC standards and states other high-level properties, such as fairness and sequential consistency.

Following such approach, paper-and-pencil mathematical developments are replaced by machine-checked proofs, which increase confidence that designs are indeed correct. Beyond that, the approach reduces the burden of peer-reviewing: instead of thoroughly checking complex paper-and-pencil mathematical analyses, reviewers are left with the task of verifying that proofs compile and that correctness criteria have been correctly stated. This is an important result, since recent findings in literature report that most errors on modern safety-critical real-time systems come from poorly-written formal specifications/requirements rather

than faulty implementations. Furthermore, using Coq allows designers to leverage powerful abstractions to design correct algorithms that are generic, scalable, and re-usable. The fact that proofs are direct linked to the algorithms' implementation also bridges the semantic gaps found in works that perform only a paper-and-pencil mathematical analysis.

In this dissertation, we present two iterations of CoqDRAM: a first, more rudimentary version, formalises the set of correctness criteria and proposes a skeleton for writing scheduling algorithms. Then, a second version includes the modelling of refresh commands and introduces the necessary machinery to make standard compliance proofs re-usable for future algorithm implementations, thus enhancing the framework aspect of CoqDRAM.

We showcase CoqDRAM's usability by modelling and proving two proof of concept scheduling algorithms: one based on the *First-in First-Out* (FIFO) arbitration policy and the other on *Time-Division Multiplexing* (TDM). For these two implementations, proof obligations are met and certified by Coq's kernel. Moreover, using CoqDRAM, we propose a new DRAM scheduling algorithm called *TDMShelve*, which extends and improves previous work on *work-conserving* dynamic TDM arbitration. More specifically, TDMShelve exploits information about the internal state of the memory at request scheduling level, thus providing a good balance between predictability and average-case latency for mixed-criticality real-time systems. The fact that an algorithm as complex as TDMShelve can be designed and proved in CoqDRAM shows that the approach can be realistically used to design complex DRAM scheduling algorithms in a trustworthy manner.

We connect the algorithms written in CoqDRAM to an external simulation environment. The fact that the experiment runs without triggering any errors related to timing or protocol correctness further validates the CoqDRAM model and helps to build confidence that the correctness criteria have been correctly stated.

Thesis supervisor: Florian Brandner
Title: Maître de Conférences, LTCI, Télécom Paris

Thesis supervisor: Mihail Asavoae
Title: Researcher, CEA List

Thesis supervisor: Lirida Naviner
Title: Professor, LTCI, Télécom Paris

Résumé

Les contrôleurs mémoire temps réel récemment proposés s'attaquent au compromis fondamental entre performance et prédictibilité, en cherchant à offrir le meilleur des deux aspects. L'idée sous-jacente commune consiste à optimiser à la fois la bande passante et la latence moyenne, tout en garantissant une caractérisation précise des performances du système. Ce faisant, il est possible de dériver des bornes supérieures pour la latence maximale des requêtes mémoire. Cependant, cette approche a entraîné des conceptions complexes, souvent accompagnées de développements mathématiques longs, difficiles à lire et à évaluer dans leur intégralité, et parfois incomplets ou fondés sur des hypothèses peu claires. Compte tenu du fait que ces composants sont fréquemment intégrés dans des micro-architectures de systèmes critiques pour la sécurité en temps réel, il est indispensable d'avoir un haut niveau de confiance dans le comportement correct de ces systèmes pour satisfaire aux exigences de certification.

Pour remédier à ces problèmes, nous introduisons un nouveau cadre formel baptisé CoqDRAM, développé au sein du prouveur de théorèmes Coq. Dans ce cadre, nous modélisons les dispositifs DRAM ainsi que les contrôleurs, et nous exprimons leur comportement attendu sous forme de spécifications formelles. L'objectif de ce cadre est d'assister dans la conception d'algorithmes d'ordonnancement DRAM corrects par construction et de garantir leur fiabilité. La spécification CoqDRAM capture les critères de correction en accord avec les normes JEDEC et formalise également d'autres propriétés de haut niveau telles que l'équité et la cohérence séquentielle. En suivant cette approche, les développements mathématiques manuscrits sont remplacés par des preuves vérifiées par machine, augmentant ainsi la confiance dans le fait que la conception est effectivement correcte. Par ailleurs, cette démarche allège le fardeau de la vérification par les pairs : au lieu d'examiner minutieusement des analyses mathématiques complexes sur papier, les évaluateurs n'ont plus qu'à vérifier la compilation des preuves et la correcte spécification des critères de correction. Cela constitue un résultat crucial, car des études récentes ont révélé que la plupart des erreurs dans les systèmes temps réel critiques pour la sécurité modernes proviennent de spécifications ou formulations formelles mal rédigées, plutôt que d'implémentations incorrectes.

En outre, l'utilisation de Coq permet aux concepteurs de tirer parti d'abstractions puissantes pour concevoir des algorithmes corrects, génériques, évolutifs et réutilisables. De plus, le lien direct entre les preuves et l'implémentation des algorithmes comble les écarts sémantiques observés dans les travaux basés uniquement sur des analyses mathématiques sur papier.

Dans cette thèse, nous présentons deux itérations du cadre CoqDRAM : la première version, plus rudimentaire, formalise un ensemble de critères de correction et propose une structure pour la rédaction d'algorithmes d'ordonnancement. La deuxième version, plus avancée, modélise les commandes de rafraîchissement et introduit des mécanismes destinés à rendre les preuves de conformité standard réutilisables pour des implémentations futures d'algorithmes, renforçant ainsi l'aspect cadre de CoqDRAM.

Nous démontrons l'utilité de CoqDRAM en modélisant et en prouvant deux algorithmes d'ordonnancement de preuve de concept : l'un basé sur la politique d'arbitrage First-in First-Out (FIFO), et l'autre sur le multiplexage temporel (TDM). Pour ces deux implémentations, les obligations de preuve sont remplies et certifiées par le noyau de Coq. De plus, grâce à CoqDRAM, nous proposons un nouvel algorithme d'ordonnancement DRAM, appelé TDMShelve, qui améliore et étend les travaux existants sur l'arbitrage TDM dynamique à conservation de travail. TDMShelve exploite les informations relatives à l'état interne de la mémoire lors de l'ordonnancement des requêtes, offrant un bon compromis entre prédictibilité et latence moyenne pour les systèmes temps réel à criticité mixte. Le fait qu'un algorithme aussi complexe que TDMShelve ait pu être conçu et prouvé dans CoqDRAM montre que cette approche peut être utilisée de manière réaliste pour concevoir des algorithmes d'ordonnancement DRAM complexes de façon fiable et rigoureuse.

Enfin, nous avons relié les algorithmes écrits dans CoqDRAM à un environnement de simulation externe. Le fait que les simulations se déroulent sans déclencher d'erreurs liées au timing ou à la correction des protocoles valide davantage notre modèle CoqDRAM et renforce la confiance dans la formulation correcte des critères de correction.

Contents

Chapter 1

Introduction

Real-time systems are computing systems that are sensitive to time, i.e., not only computations need to produce the correct results (a property also known as *functional correctness*), but also need to happen at the right time (i.e., *timing correctness*).

As an example, consider an airplane performing an *automatic landing*, a common procedure for airplanes to land in low-visibility conditions, i.e., when pilots cannot rely on their eyes to perform a manual landing (i.e., a *visual* landing), usually due to the presence of intense fog over the runway. Such a procedure relies on a system called *Instrument Landing System* (ILS), which includes several radio transmitters placed near the runway, as shown in Figure 1.1. In that situation, the computers in the airplane have to process the radio signals transmitted by the ground equipment and calculate which actions the plane must take in order to follow the right directions, which could include both lateral and vertical corrections. Moreover, not only the computers have to be able to calculate the adequate direction inputs, but they have to do that precisely at the right instant. If the computer takes too long, for instance, than the airplane will already be far from the position where it was when the data was initially received, and the correction will happen too late. Potential

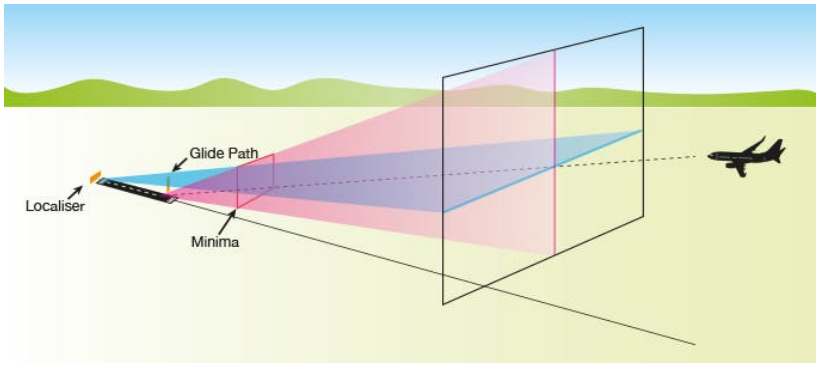


Figure 1.1: *Instrument Landing System* (ILS) for automatic landing [1].

consequences of this include the plane missing the descent line - also called *glide path* – and having to perform another landing attempt (which is expensive manoeuvre for airlines and may delay arrival, thus disturbing passengers). In the worst-case scenario, the plane could not be able to recover and be forced to perform a dangerous landing or even hit the ground or an object when flying too low.

In such systems, in order to ensure timing correctness, computations – often referred to as workloads or tasks – have associated *deadlines*, which indicate the latest acceptable time that the workload should finish executing. Furthermore, a *safety-critical* (or just *critical*) real-time system is a system in which a wrong computation or the non-respect of a deadline could result in potentially catastrophic outcomes, such as loss of life or severe environmental damage (e.g., airplanes, railway signalling systems, automotive cruise control, pacemakers). Safety-critical systems are thus subject to strict safety certification procedures, which aim at providing functional and timing correctness guarantees. Software embedded into airplanes, for example, have to comply with the DO-178C standard [2], which establishes multiple pre-requisites to determine if the code can be deemed *safe* before it is deployed.

Computing systems such as the ones found in cars and airplanes (and in modern real-time systems, more generally) are evolving in a way such that increasingly more functions are delegated to computers instead of humans. Specifically, the recent advances in machine learning and computer vision (among other technologies) allow tasks such as driving and landing an airplane to be performed fully automatically. In more detail, functionalities can be classified according to different criticality levels in such systems: in a modern car, for instance, sub-systems such as entertainment, power management, and navigation are considered to be of *low-criticality*, while functionalities such as steering-assistance [3]; *Collision avoidance systems* (CAS) [4]; *Advanced emergency breaking system* (AEBS) [5]; and *Adaptive cruise control* (ACC) [6] are considered to be of *high-criticality*. Figure 1.2 depicts a number of computer systems present in an average modern vehicle. Computing system that include tasks with different criticality levels are also referred to as *mixed-criticality* (or *multi*-criticality) systems [7].

In airplanes, critical functions include sensing and processing data such as airspeed, altitude, and temperature; calculating the necessary corrections to follow a given route; and applying such corrections to the engines and control surfaces (i.e., the *rudder*, the *elevators*, and the *ailerons*). Non-critical functions include some secondary cockpit displays, lighting, and passenger *infotainment* systems.

Furthermore, in the past, each major functionality was deployed on a dedicated computing node, each with its own memory system, conforming to a *federated approach*, or *federated architecture*. Such strategy provided strong *isolation* guarantees to a given system,

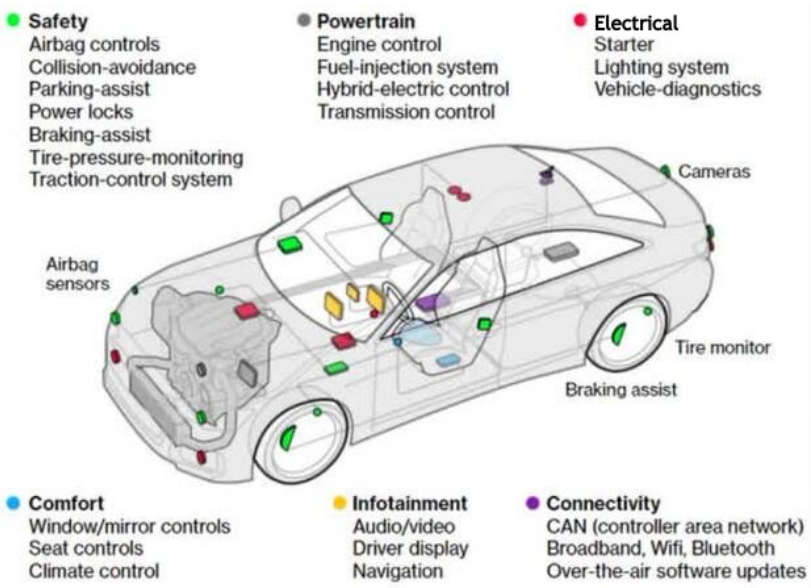


Figure 1.2: Computer-performed functions in an average modern car [8].

i.e., tasks could interfere very little with each other's data (thus ensuring *spatial* isolation) or execution time (*temporal* isolation). This strategy, however, has inherent limitations, due to the unreasonable increase in the number of wires, connectors, energy consumption, and overall cost. These factors are sometimes put together and summarised by the term *Size, Weight, Power and Cost*, or simply *SWaP-C* [9]–[11]. In the automotive domain, for example, modern cars have more than 20 million lines of code deployed on around 100 *Electronic Computing Units* (ECUs), and the total value of electronic components ranges from 40% for traditional vehicles up to 75% for electric vehicles [9], [12].

The most often adopted solution to the SWaP-C problem is to use *multi-core* devices, which can act as power-full centralised computing units capable of simultaneously hosting tasks of different criticality levels.[1] Besides that, multi-core processors offer very good *performance* by performing different computations simultaneously. Given the intense computational load brought by the recent technology trends in modern real-time systems, multi-core systems are vital to achieve good performance while maintaining a reasonable SWaP-C.

Furthermore, although multi-core devices proposes a solution to the performance and scalability problems [14], it comes with an inconvenience: contention to shared resources, i.e., elements that can be accessed by all cores. Examples of such elements are: parts of the memory hierarchy (e.g. *Last-Level-Cache*, interconnects, buses, and the main memory),[2]

[1] In the aviation industry, however, although existent, the use of multi-core is not yet widespread. Since the certification procedure is stricter than that of the automotive industry, the adoption of multi-core processors has advanced at a much slower pace [13].

[2] From a bottom-up perspective of the system architecture, the utilisation that individual tasks make of

and peripherals, such as user input devices (e.g., steering wheels, control sticks, and touch-sensitive screens), sensors, actuators, and displays. In other words, tasks may *interfere* with each other and thus break the isolation principle.

Therefore, a multi-core device – when used in the context of real-time systems – should still provide the necessary means to achieve *predictable* timing behaviour. Predictability, in the context of real-time systems, means "to be able to satisfy the timing requirements of critical tasks with 100% guarantee over the life of the system, be able to assess overall system performance over various time frames, and be able to assess individual task and task group performance at different times and as a function of the current system state" [15]. Or, more informally, it means that, at design time (of the respective computing platforms), all possible scenarios are accounted for, including the worst-case. Since all processing delays are accounted for beforehand, a predictable system enjoys the guarantee that the time it takes to process/complete tasks will never exceed its calculated bounds. Moreover, besides worst-case metrics, it is also important to be able to study and characterise the system's performance over time and identify different classes of latency.

Regarding predictability in multi-core devices, a typical solution to the previously mentioned spatial isolation problem is to organise the cache system (the first layer of the memory hierarchy) into levels and assign individual lower-level caches for each core. Therefore, cores only have to access the shared upper levels when they cannot find the address they are looking for in their own individual lower level, which makes it less likely that contention occurs. Besides cache *organisation*, memories (from shared caches to the main memory) can also be *partitioned*, in a way such that different cores can only access a sub-region of the respective resource – thus (nearly) eliminating interference due to contention.

Regarding time, a *scheduler* is responsible for granting individual cores periods of exclusive access to the shared resources. The instructions – or series of steps – performed by a scheduler can be referred to as a *scheduling algorithm*. In order to guarantee predictable timing behaviour in a real-time system, conservative scheduling algorithms are often needed, which sacrifice *average-case performance* and *bandwidth* [16]–[19] (i.e., how much data the algorithm can transfer back and forth each second), for properties such as *fairness* (i.e., the guarantee that every core eventually gets a chance to access the resource) and *non-starvation* (the guarantee that every request that has entered the system will eventually be serviced). This dissertation, specifically, discusses the scheduling of memory accesses to the main memory.

Predictability is often (indirectly) quantified in literature through a metric called *Worst-Case Latency* (WCL). Or, more precisely, predictability refers to the underlying complexity

many of those shared resources at the processor level is too low to justify a dedicated use of them [14].

behind the WCL formula and the proof that the formula is indeed correct. For a scheduling algorithm used to grant access to the main memory, for instance, the WCL is a theoretical upper bound on the time that a memory request waits to be serviced, i.e., to get its data retrieved from the memory (if it is a *read* request), or to write its data to the memory (if it is a *write* request). That metric is calculated by considering the most pessimistic outcome in every scenario (e.g., assuming that memory requests always result in cache misses). Algorithms that perform aggressive optimisations in order to get good average-case metrics typically carry two consequences: 1) The WCL expression may be difficult to analyse, i.e., the algorithm takes so many decisions that it becomes difficult to derive a correct expression for the WCL (thus making it *unbounded*); and 2) The WCL could be too pessimistic, which might be a problem for systems that require rapid processing (as it is the case of airplanes and cars). Literature typically refers to this compromise as the "*performance-predictability trade-off*" [16]–[19]. In other words, the more complex and performance-oriented a scheduling algorithm is, the more burdensome it is to derive a correct and tight WCL.[3]

Specifically, this dissertation focuses on scheduling algorithms used to control access to the main memory, typically a *Dynamic Random-Access Memory* (DRAM) device, and the shared system bus. The hardware components executing such algorithms are *memory controllers* and *bus arbiters*, respectively. Recently proposed DRAM controllers have been tackling the performance-predictability trade-off by trying to offer the best of both worlds: algorithms that offer good average-case performance while at the same time offering (reasonably) tight and well defined worst-case latency bounds [17]–[37].

Culminating to the main topic of this dissertation, one very important aftereffect following this recent trend in real-time memory controllers is the increase in complexity of the underlying WCL analysis. Lengthy paper-and-pencil mathematical developments are hard to read and review, may rely on unclear assumptions and omit important parts of the proof for the sake of conciseness and simplicity. Moreover, the discussion of WCL in recent works proposing real-time memory controllers occupy 30% to 50% of the total space of each article [38]. 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,

[3]If the reader is unfamiliar with these terms, the following analogy might be helpful: Imagine that someone is folding clothes to put into a drawer. If that person is prioritising speed, most pieces are successfully folded quite rapidly, but the technique could get sloppy, and in the worst-case, the person would need to repeat the move a couple of times to get the folding right, which would take a significant amount of time. Now imagine that, instead, the person folds each piece slowly, but carefully, to the point that it is impossible to get it wrong. On the average case, folding might be slower than in the previous case, but since the technique is precise, a costly mistake never happens, i.e., the *worst-case latency* of the folding process is smaller.

results, and other engineering aspects, for the sake of reproducibility.

1.1 What Is This Dissertation Exactly About ?

This dissertation investigates and proposes solutions to two problems:

1. How to improve an specific existing State-of-the-Art memory request scheduling algorithm [39], [40], designed for real-time mixed-criticality systems?

2. How to address the development of memory controllers' scheduling algorithms in a *trustworthy* manner?

While the thesis proposal emphasised the first point, it did not take long for the second problem to become apparent and thus become our main focus. In more detail, our interest for trustworthiness and verification problems arose from the observation that mathematical developments in the field were burdensome, sometimes incomplete, and most importantly, lacked readability. To address these issues, we propose a framework to guide the design of *formally proven* DRAM controllers. To achieve that, we rely on a *formal verification* tool – Coq.[4] More specifically, Coq is a *theorem prover* – a *formal system* comprised of a programming language and features to interactively prove logical properties about programs.

We use Coq to formalise the meaning of *correctness* for memory controllers, which concretely, means that we write an interface for memory controllers that specifies *correctness criteria*, which in Coq are called *proof obligations*. In other words, any memory controller implementation developed within the framework must present mathematical proofs – which are also developed and checked in Coq – that these correctness criteria are met. Moreover, in the framework, we model different sorts of logical propositions about memory controllers: correctness accordingly to the *Joint Electron Device Engineering Council* (JEDEC) standards [41], [42], which defines the correct behaviour of a memory device; and other sorts of properties relevant for real-time systems or concurrent systems, e.g., non-starvation (or, differently put, that the WCL is a closed expression), fairness, and sequential consistency.

According to that approach, memory controllers developed within the framework no longer have to explicitly present dense mathematical developments in order to support their predictability claims. Instead, proofs about the WCL (or other properties) can be presented as Coq artefacts and checked by its kernel.[5] Hence, two direct consequences follow: 1) The peer-review process becomes more trustworthy, in the sense that reviewers no longer have

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

[5] The Coq kernel is highly trustworthy, a point that will be revisited in Chapter 2.

to perform a thorough (and highly error-prone) check of paper-and-pencil mathematical developments; and 2) The work of checking a proof is replaced by the work of checking if the property has been correctly stated.

Finally, coming back to the first point of interest of this dissertation, we use our novel framework, which will be from now on referred to as *CoqDRAM*, to propose a new DRAM scheduling algorithm – which we call *TDMShelve*. The algorithm builds on previous results by Hebacche et al. [39], [40]. Hebbache's algorithm finds itself in the class of algorithms that try to offer a good compromise between performance and predictability for multi-criticality systems: while high-criticality tasks are guaranteed to finish execution before a determined upper bound, low-criticality tasks are executed in the time that the memory finds itself idle.

In summary, we improve that algorithm, which previously considered the main memory device as a black box, to take the internal state of the DRAM device into consideration when taking scheduling decisions. Thus, we ensure better performance for non-critical workloads while simultaneously guaranteeing the same upper bounds for the execution time of critical workloads.

1.2 Contributions

Below, we list the scientific outcomes of this PhD thesis:

- *CoqDRAM* [43], a novel framework to design and explore DRAM scheduling algorithms. The framework contains a generic and reusable model of DRAM devices in the form of a formal specification that can be refined through actual implementations. The specification carries correctness criteria (as proof obligations in Coq) of different sorts: timing and functional correctness according to the JEDEC standards [42], and higher-level properties, such as fairness and sequential consistency;

- Two *Proof-of-Concept* (PoC) DRAM scheduling algorithms implemented in CoqDRAM: one based on the *First-In First-Out* (FIFO) arbitration policy and the other on *Time division multiplexing* (TDM). For both, proof obligations are met and certified by Coq's kernel;

- *TDMShelve* – a novel DRAM scheduling algorithm developed with CoqDRAM. TDMShelve builds on previous work by Hebbache et al. [39], [40];

- A methodology to validate CoqDRAM implementations. We use the code generated by Coq's extraction mechanism as a plug-in replacement in an existing cycle accurate DRAM simulator – MCsim, developed by Mirosanlou et al. [44].

1.3 Organisation

The remainder of this dissertation is organised as follows:

Chapter 2 introduces ubiquitous technical concepts. It is divided into 3 sections: Section 2.1 introduces key concepts about real-time systems, including definitions, notions about multi-core systems, and contention to shared resources; Section 2.2 takes a deeper look into an specific element of the memory hierarchy – DRAM devices, which is the main focus of this dissertation; and Section 2.3 introduces the fundamentals of formal verification and Coq.

Next, Chapter 3 presents and discusses related work and motivates our approaches by discussing the problems found in the development of State-of-the-Art real-time memory controllers.

The presentation of our novel technical contributions begins in Chapter 4, with a high-level overview of our framework's architecture and rationale. Next, Chapters 5 and 6 detail how the framework is designed and used, respectively. Chapter 6, more specifically, details the two PoC implementations and briefly discusses proof strategies. Furthermore, Chapter 7 describes a validation experiment in which we run proved Coq-generated code in an external simulation environment in order to further build confidence that the specification is correct.

Chapter 8 presents the features most recently added to CoqDRAM, which aim to improve the re-usability aspect of the framework. Chapter 9 follows by presenting our novel algorithm, *TDMShelve*, which leverages these new features.

Chapter 10 concludes this dissertation by presenting "big picture" point-of-view of the research, reflects on what has been accomplished, and concludes by suggesting promising future work directions.

Appendix A details a research effort in which we explored an equivalence "link" between CoqDRAM algorithms and their "hardware versions" written in Cava – a Coq DSL for hardware design.

Chapter 2

Background

2.1 Real-Time Systems

Several definitions for the term "*Real-time System*" (RTS) exist. Stankovic [45] defines it as follows: "*Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced.*" Arguably, the most important requirement for system of such type is the respect of strict *deadlines* – defined informally as the latest date (time) that a processing unit should receive an answer from the computing system (i.e. have its calculations finished, and/or its requests serviced).

Real-time systems might be further classified by the severity of a deadline miss: while *hard* real-time systems consider a deadline miss as a *total failure*; *soft* real-time systems consider that the usefulness of a result degrades after its deadline, thereby degrading the system's quality of service (QoS) [46] (but not characterising a total failure).

More specifically, in this work, we are interested in the challenge of meeting such real-time requirements with multi-core processors. As briefly mentioned in Chapter 1, multi-core architectures rose to prominence due to their significant benefits w.r.t performance, lower clock rates, and energy efficiency; compared to single-core processors [47]. A basic requirement for using multi-core architectures for real-time applications, however, is that the timing behaviour of the processor should be predictable, i.e., it must be possible to determine an upper bound of the maximal execution time of a task, or *Worst-Case Execution Time* (WCET), which is guaranteed to hold. The main obstacle to achieve predictable behaviour in a multi-core architecture is contention due to *shared resources* – of which the *memory hierarchy* of a processor is a key part.

2.1.1 The Memory Hierarchy & Contention

To go deeper into how the memory system works, we start with registers. Registers are used to hold values during the execution of all sorts of instructions inside a processor. Registers provide the fastest access to data in the memory hierarchy (typically just 1 clock cycle, in the case of a pipelined processor). Each core possesses an individual set of registers, which are thus not subject to contention by other cores.

Next, since there are only about a few dozens to a couple of hundred[1] 32-bit or 64-bit registers in a core, whenever a core has to manipulate larger chunks of data or instructions, it reads and writes from memories called *caches*. Modern caches are organised in levels: each processor typically has *two* exclusive L1 caches, which are not shared with other cores. While one of the two L1 caches is exclusively used to store instructions, the other is used exclusively to store data (in some special cases there might be a single L1 cache used for both data and instructions). L1 caches are small and typically range between 16KB to 64KB and provides the fastest access of all caches. The following level, L2, is larger (in size) than L1, ranging from 256KB to 2MB, and it may or may not be shared between cores. Today's processors often have an L3 level and rarely an L4 level [49]. The highest cache level in a processor is called the *Last Level Cache* (LLC), and is typically shared among all processors. If some data or instruction is not found in the L1 cache, then cores look for it in the L2 cache, and so forth.

Both registers and caches are built with *static random-access memory* (SRAM) technology, a type of volatile memory, i.e., it holds its data as long as it is powered on. Thanks to how SRAMs are built, accesses to registers and caches are fast: register access takes only one clock cycle on pipe-lined processors, and access to data present in the caches takes from two to ten clock cycles (depending on the level). Speed, naturally, comes with a price: SRAMs are significantly more expensive than other technologies used in lower levels of the hierarchy.

Whenever a processor tries to access a given memory address, either the address it wants to read from or write to is already in the cache, or not (since the cache is limited in size, it will not always be possible to store every data or instruction needed by the program). If the address is in the cache, it is said that the access was a *cache hit*; if not, then it is a *cache miss*. In the case of a cache miss (i.e., the address to be read from or written to was not present in any of the levels, from L1 to LLC), the data must be retrieved from (or written to) the main memory. Figure 2.1 [50] illustrates how the memory hierarchy is organised. In the figure, note the terms "primary" and "secondary storage". Primary storage consists

[1]The actual number of registers on a core varies greatly. While simple processors such as the ARM Cortex-M3 [48] – often found in micro-controllers – only has 17 registers 32-bit registers, a modern x86_64 processor have around a hundred.

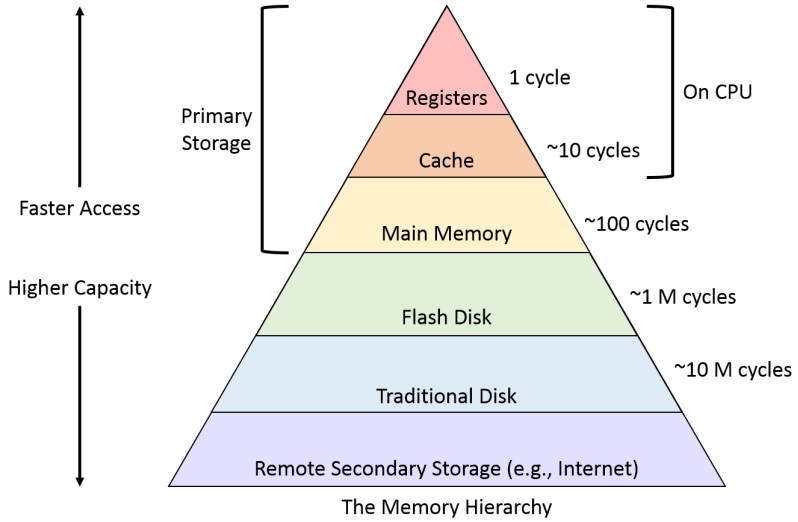


Figure 2.1: The Memory Hierarchy [50].

of the upper layers of the memory hierarchy, closer to the processor. Elements of primary storage are (almost exclusively) made of volatile memory, used by the processor to handle data during the execution of programs.

The main memory – the last layer of primary storage in a computer – is built with *Dynamic Random Access Memory* (DRAM) technology, which is also a type of volatile memory that is slower and cheaper than SRAM. Typically, DRAM memories are some orders of magnitude larger than SRAMs, with its capacity ranging from 8GB to 64GB (for the DDR4 generation). Section 2.2 is dedicated to DRAMs, further detailing their internal structure and functioning.

Finally, below the main memory, we find secondary storage units (or *external* memory), which are non-volatile memories, i.e., data that does not get erased when the device is powered down (which is why it is also called *permanent* storage). Following the trend seen before, because it is cheap, modern computers typically have two orders of magnitude more secondary storage than all of the primary storage. However, it is significantly slower: while the access time per byte for primary storage is measured in nanoseconds, accesses to the secondary storage are measured in milliseconds. Note however, that real-time systems may not include permanent storage at all. In fact, following the recent technological trend, real-time embedded computers prefer to store elements in cloud-based services rather than carrying additional hardware for local storage.

The main obstacle to achieve predictable timing behaviour in a multi-core architecture is contention to *shared resources*. Examples of shared resources on commercial multi-core processors are: the shared levels of the memory hierarchy (e.g., system bus, memory bus

and controller, main memories, caches), *Direct Memory Access* (DMA) controllers, *Graphics Processing Units* (GPUs), interrupt controllers, support on-chip logic (e.g., coherency mechanism, *Transaction Look-aside Buffers*), I/O devices, et. cetera [51].[2] Different applications running simultaneously on different cores may "compete" for exclusive access to such shared resources, conforming *inter-task* interference, which have to be carefully examined for a multi-core architecture to be considered real-time compatible.

Shared resource	**Mechanism**
System bus	Contention by multiple cores Contention by other device - IO, DMA, etc. Contention by coherency mechanism traffic
Bridges	Contention by other connected buses
Memory bus and controller	Concurrent access
Memory (DRAM)	Interleaved access by multiple causes address set-up delay Delay by memory refresh
Shared cache	Cache line eviction Contention due to concurrent access Coherency: Read delayed due to invalidated entry Coherency: Delay due to contention by coherency mechanism read request by lower level cache Coherency: Contention by coherency mechanism on this level
Local cache	Coherency: Read delayed due to invalidated entry Coherency: Contention by coherency mechanism read
TLBs	Coherency overhead
Addressable devices	Overhead of locking mechanism accessing the memory I/O Device state altered by other thread/application Interrupt routing overhead Contention on the addressable device - e.g. DMA, interrupt controller, etc. Synchronous access of other bus by the addressable device (e.g. DMA)
Pipeline stages	Contention by parallel hyperthreads
Logical units	Contention by parallel applications
	Other platform-specific effects, e.g. BIOS Handlers, Automated task migration, Cache stashing, etc.

Table 2.1: List (non-exhaustive) of possible sources of interference due to contention in a multi-core systems [51]. Highlighted cells represent contention phenomena relevant to this dissertation.

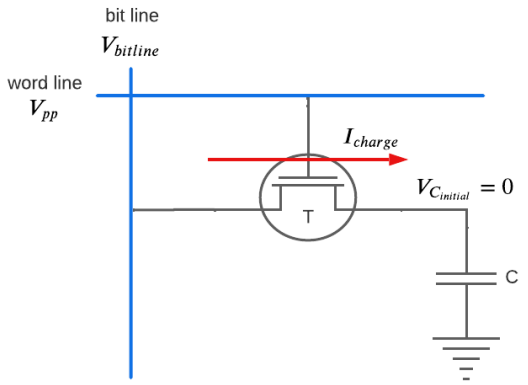
Table 2.1, reproduced from a survey by Kotaba et al. [51], provides a comprehensive overview of undesired shared-resource contention phenomena affecting temporal predictability in multi-core processors. In this dissertation, we focus on the problem of contention on

[2]Even in single-core processors, accesses to shared resources by parallel tasks can cause interference, since the ordering of accesses can vary significantly. With speculative hardware micro-architecture mechanisms like caches, out-of-order execution, or branch prediction, predictability degrades [47].

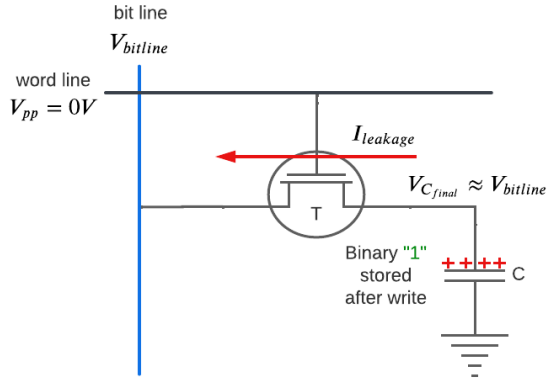
the main memory – DRAM devices – but also the system bus, memory bus, and memory controller; all highlighted in green in Table 2.1.

2.2 DRAM

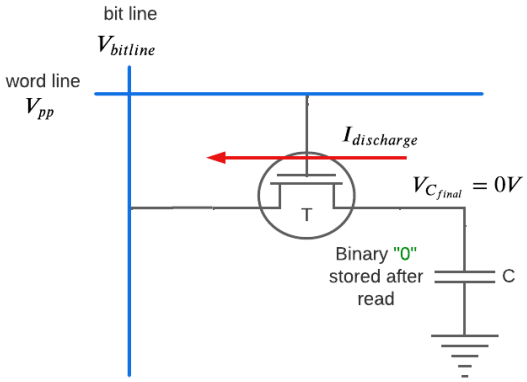
DRAM devices are multi-dimensional structures, where each element capable of storing a bit is a called a *cell*. A single DRAM cell is depicted in Figure 2.2: it is made of a CMOS transistor and a capacitor (**T** and **C** in the figure, respectively).



(a) Write "1" operation: word line is powered up, allowing current to flow through the transistor. If $V_{bitline} > V_{C_{initial}}$, the capacitor will charge.



(b) After writing, the cell will retain the written bit, but not indefinitely, since the capacitor slowly loses charge due to the leakage current in the transistor.



(c) Reading operation: word line is powered up, allowing current to flow from the charged capacitor to the bit line. After reading, the capacitor is fully discharged and now stores bit "0", meaning that the read operation on DRAM cells is a destructive process.

Figure 2.2: Reading and writing in DRAM cells.

The basic operation of a DRAM cell can be described as follows: to write a "1" bit into the cell (a situation shown in Figure 2.2a), one must provide power to *word line*, which consequently closes the transistors and allows current to flow from the *bit line* to the capacitor, or vice-versa. If $V_{bitline} > V_{C_{initial}}$ (which will always be true during a write operation, for reasons discussed further), current flows from the bit line to the capacitor, charging it to a value close to $V_{bitline}$, thus representing binary "1".

Next, if the word line is powered down, current stops flowing through the transistor and the capacitor remains charged (with a binary "1", in the case from Figure 2.2b). It is important to note, however, that the capacitor does not remain charged indefinitely, since it slowly loses charge due to leakage currents through the transistor.

Finally, if a read operation is to be performed on the cell, word line needs to be powered up again, which allows current to freely flow through the transistor, as shown in Figure 2.2c. If $V_C > V_{bitline}$, then current flows from the capacitor to the bit line, thus representing a binary "1" being read. Note that during the read process, the capacitor discharges, which means that reading from cells is a destructive process (i.e., at the end of the transaction, the charge over the capacitor, $V_{C_{final}}$, is zero).

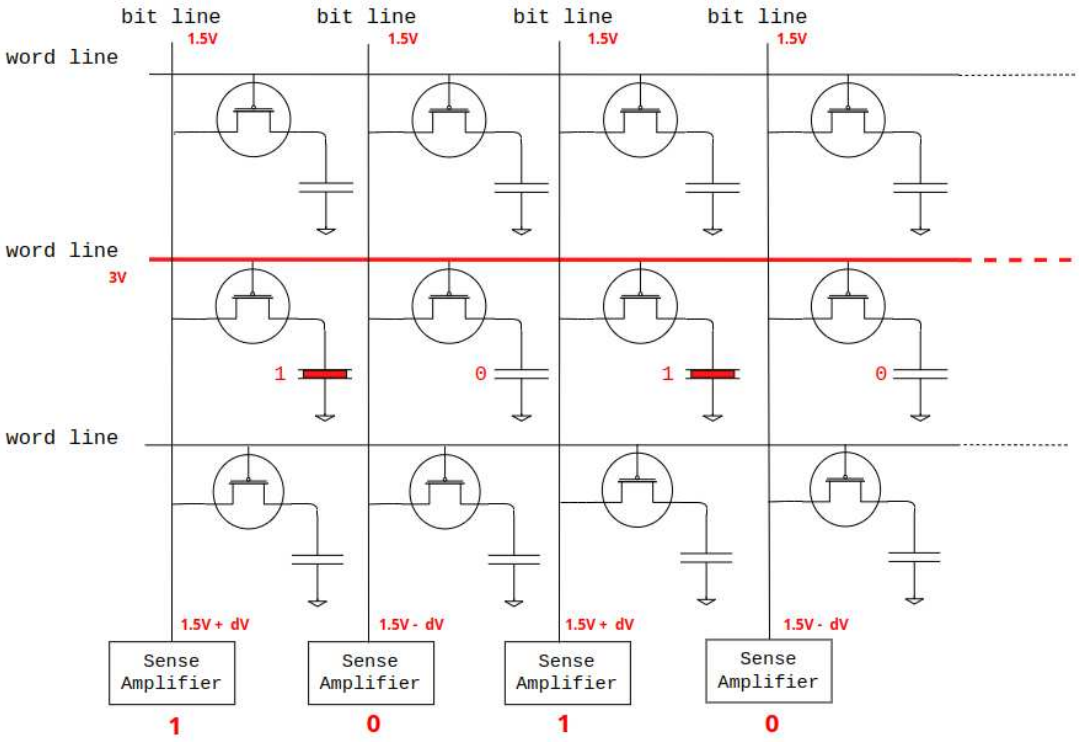


Figure 2.3: Reading DRAM cells. ■ Powered-up line/Charged capacitor/Voltage values.

Furthermore, as it can be seen in Figure 2.3, multiple cells are organised in rows and

columns, and read or write requests operate on entire rows at once. In order to read a line, all bit-lines are precharged at half of the supply voltage of the DRAM ($1.5V$ in the example in the figure). Then, the word-line of the requested row is connected to the supply voltage ($3V$ in the example), which make the transistors' gate close, allowing current to flow. Consequently, all cells whose capacitors are charged (representing bit "1") will transfer charge to the bit-line, and all cells with empty capacitors (representing bit "0"), will receive charge from the bit-line. The (small) voltage fluctuations in the bit-line are then detected by differential amplifiers, or *sense amplifiers* – which have the circuitry to store the information if the cell contains a "1" or a "0". Again, note that the read operation is destructive, i.e., the data stored in the sense amplifier needs to be written back into the cells before another row is read. The write operation is very similar: the word-line (row) is activated, but the charge to the bit-lines comes from a memory request – which will then be stored in the cell capacitors.

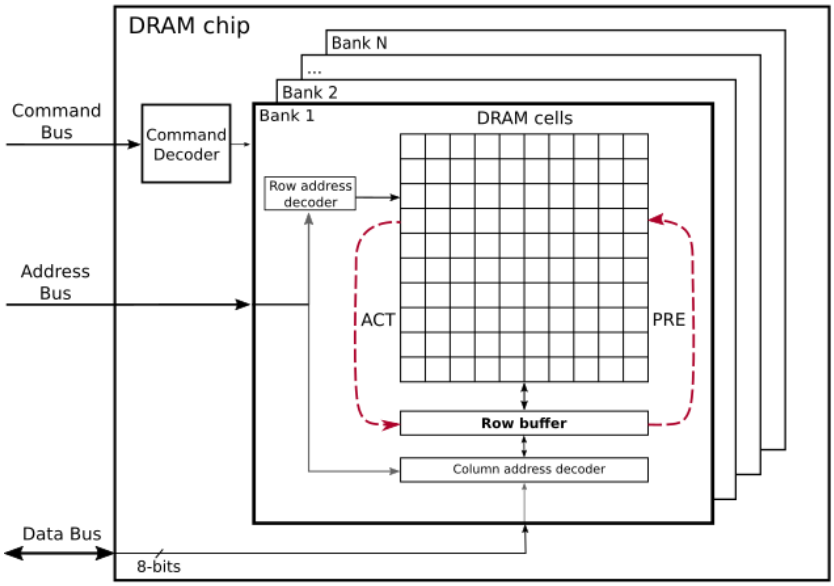


Figure 2.4: DRAM chip structure.

We call a two-dimensional grid of memory cells a *bank*. A single DRAM device has multiple banks (8 in DDR3 and 4 to 16 in DDR4).[3] The full structure of a DRAM chip, abstracting from individual cells, can be seen in Figure 2.4. In the figure, notice the *Row-buffer*, a per-bank cache-like mechanism: it stores the last accessed row of the bank, which can then be accessed with smaller latency (i.e., memory requests that target the same row in the DRAM will enjoy faster access). Hence, a memory request can either be a *row-hit* or

[3]DDR4 banks are further grouped into bank-groups, which makes DDR4 devices' logical structure four-dimensional. More on that later.

row-miss, much like a cache, with variable latencies. Furthermore, each bank includes row and column decoders, which take a physical address coming from the memory controller and decodes it into row and column.

Moreover, the top-most level in the logical structure of the DRAM is a *rank*. Each rank has its own command, address and data buses pins. Independently from its logical structure, ranks are also divided in chips (the rectangular structures that can be typically seen in commercial DRAM modules) – which are 8-bit wide. Each rank has 8 chips, which makes the ranks' data-bus 64-bit wide – the size of a single transaction.[4]

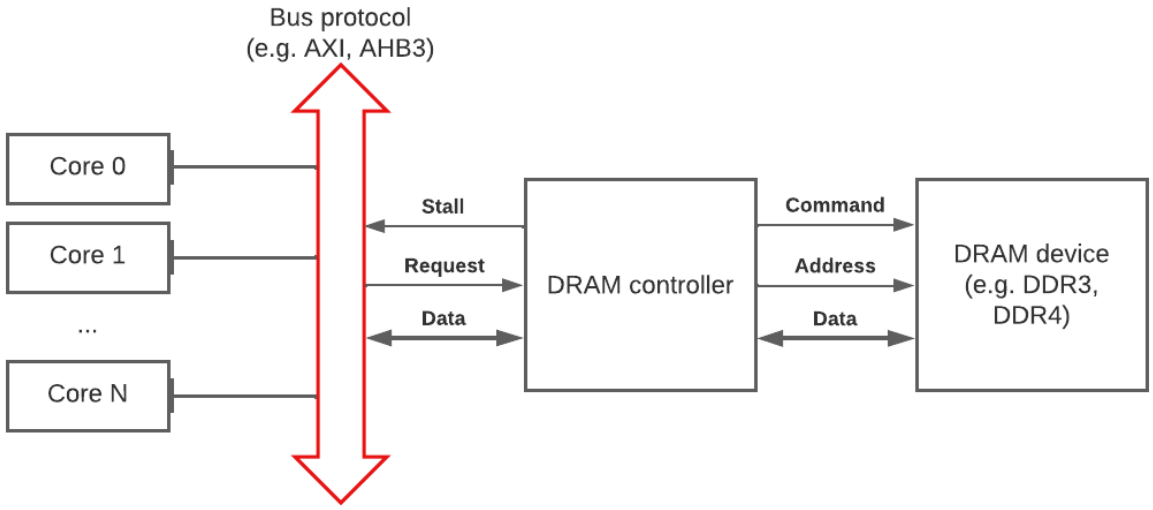


Figure 2.5: The connection between cores and the memory.

The memory is effectively controlled through *commands* sent by the memory controller, as shown in Figure 2.5. More generally, Figure 2.5 shows how the memory controller connects processors to the memory device. Processors issue their requests to a common bus, usually controlled by an arbiter implementing some kind of handshake protocol, such as the AMBA-AXI[5] or AMBA3-AHB protocols.[6] Then, if accepted by the protocol, the bus arbiter redirects requests to their target destination – the DRAM device, in this case. The DRAM controller (which has its role explained in more detail in the following section), connects to the memory device via three buses: 1) the address bus, which contains encoded information specifying which bank-group, bank, row, and column are to be accessed; 2) the command bus, which contains information about which operation is to be performed on the device; and 3) the bi-

[4]Bear in mind that chips are not relevant for addressing though, i.e., a single DRAM address is simply divided between chips and all chips are accessed simultaneously to compose the 64-bit data.

[5]https://developer.arm.com/documentation/ihi0022/latest/

[6]https://developer.arm.com/documentation/ihi0033/latest/

directional data bus, which transports data back and forth. Moreover, the DRAM controller can also communicate with the bus protocol through additional signals. For example, if the memory controller decides to stall the arrival of requests (because its request queue might get full, for example), then it might use a "stall" signal to communicate the stall condition to the bus arbiter.

The functionality of DRAM devices is described in the JEDEC standards [42]. Specifically, the state of a DRAM device is specified in a per-bank manner: each bank is a state-machine, with transitions being triggered by DRAM commands sent by the memory controller. Figure 2.6 – a close reproduction of the state-machine specified in the JEDEC DDR4 standard [42] – shows a simplified bank state machine.[7]

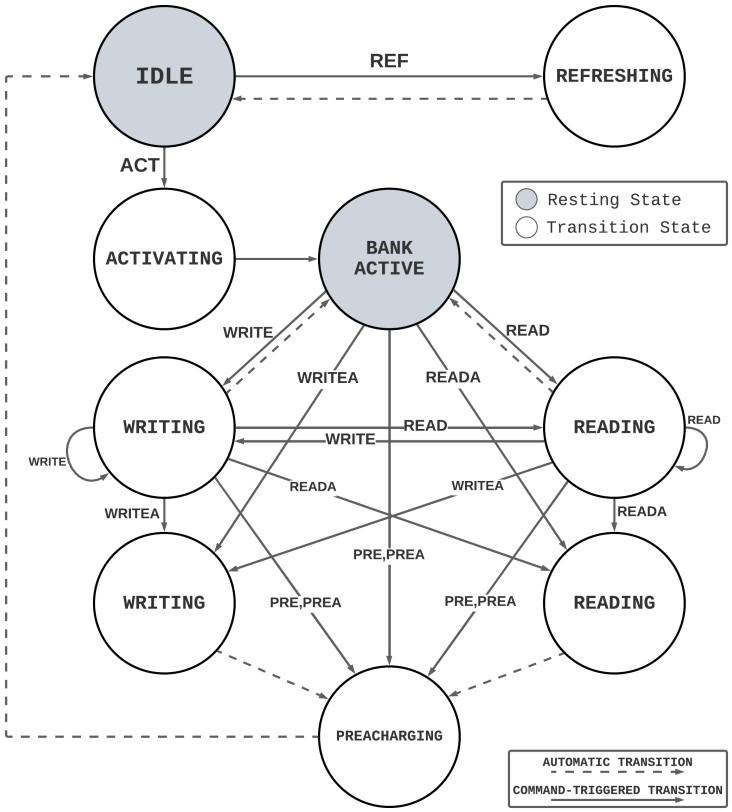


Figure 2.6: DRAM State Diagram.

As it can be seen, there are two "resting states" in the state diagram, in the sense that

[7]For simplicity, Figure 2.6 does not include details about the initialisation procedure and the following mechanisms: calibration, write levelling, self refresh, *Per DRAM Addressability* (PDA) mode, *Power Down* mode, and configuration of registers. Consult the JEDEC DDR4 standards for detail about these procedures [42]. In our work, we only consider the DRAM in a normal operating state, i.e., after initialisation, calibration, and mode configuration. Since we are mainly interested in timing properties, we consider that the critical operations, which may affect predictability and performance when actually running software, take place on the running state of the device.

the memory can stay in such states as long as it remains "undisturbed": *IDLE* and *ACTIVE*. A bank is said to be IDLE when there is no row loaded into the row buffer, and it is said to be ACTIVE when there is a row in the row buffer. As one can see in the figure, there are a handful of different commands types:

- `ACT` commands, also called *Row Address Strobe* (RAS), are used to load a row into the row-buffer;
- `READ` (shortened as `CRD`) and `WRITE` (shortened as `CWR`), also called *Column Address Strobe* (`CAS`) commands, read or write to a column address from an active row (the row already loaded into the row-buffer), respectively. Note that `CRD` and `CWR` commands can happen either from a `BANK ACTIVE` state or directly after another `CWR` or `CWR` command (which is useful for read- or write-burst operations);
- Since loading a row into the row-buffer is a destructive process, the `PRE` command writes back the content of the row-buffer into the cell matrix. A `PRE` is necessary whenever a row-miss happens, i.e., when the controller tries to access a different row than the one already loaded into the row-buffer of a given bank;
- `PREA` commands – in a per-bank sense – are equivalent to `PRE` commands, but they precharges all banks at once;
- `WRITEA` (shortened as `CWRA`) and `READA` (shortened as `CRDA`) commands behave as standard `CWR` and `CRD` commands, but also automatically trigger a precharge procedure (the "A" at the end stands for *Auto-Precharge*);
- `REF` commands are periodically needed to recharge cell capacitors in a given row of the bank. This is due to leakage currents in both the capacitor and transistor in the cell. The memory itself keeps a counter of which row has to be recharged, and the controller only has to issue the `REF` command at the right instant.

Furthermore, for the memory to function correctly, the JEDEC standard establishes several constraints on how far apart in time different commands must be (i.e., lower bounds on the time elapsed between commands), which are summarised in Table 2.2, for two specific devices – a DDR3 and a DDR4 device. These constraints come from physical characteristics of transistors and capacitors inside cells and the data bus. Note that while some constraints apply only for commands sent to the same bank (*intra-bank*), some apply for all commands, regardless of bank, and a couple apply just to commands sent to different banks (*inter-bank*). In addition, for DDR4 devices, some constraints can have two values, with identifiers

Symbol	Description	DDR3-1600K (in data bus clock cycles)	DDR4-2400U (in data bus clock cycles)
	Exclusively intra-bank		
t_{RCD}	ACT to CWR/CRD	11	18
t_{RP}	PRE to ACT	11	18
t_{RC}	ACT to ACT	39	57
t_{RAS}	ACT to PRE	28 (min), $9 \times T_{REFI}$ (max)	39 (min), $9 \times T_{REFI}$ (max)
t_{WL}	CWR to data bus transfer	8	12
t_{RL}	CRD to data bus transfer	11	18
t_{RTP}	CRD to PRE	6	9
t_{WR}	WR data to PRE	12	15
	Intra and inter-bank		
$t_{RD-to-WR}$	CRD to CWR	9	12
t_{WTR}	End of WR transaction to CRD	6	s=3, l=9
$t_{WR-to-RD}$	CWR to CRD	18	s=19,l=25
t_{BURST}	Data bus transfer	4	4
t_{CCD}	CWR-to-CWR or CRD-to-CRD	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

Table 2.2: JEDEC timing constraints for a DDR3 and a DDR4 device. In the device name, the number represents the device's speed in MHz and the letter is the speed grade.

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.

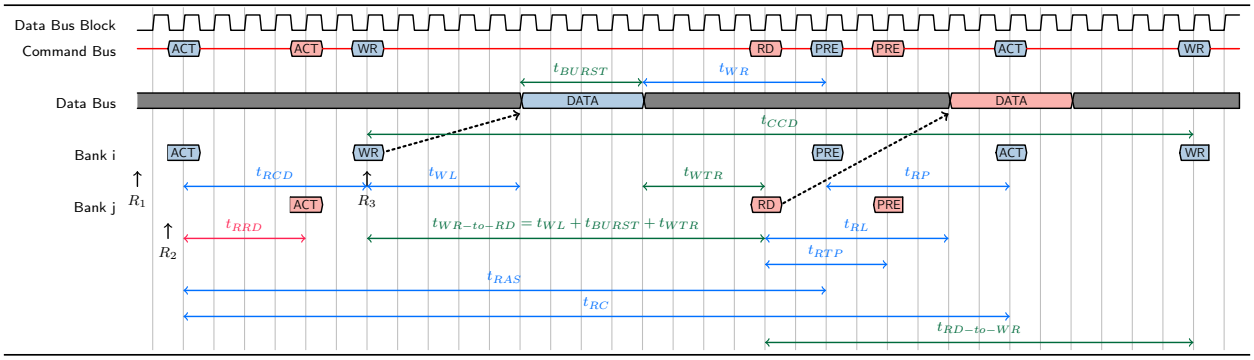


Figure 2.7: Example of timing constraints for a DDR3-800E device.

Commands and data: Bank i, Bank j
Constraints: Exclusively intra-bank, Inter- and intra-bank, Exclusively inter-bank

Figure 2.7 illustrates a valid sequence of commands sent to a DDR3-800E device (in the sense that it operates according to the state machine described in Figure 2.6 and all timing constraints in Table 2.2 are respected). To interpret the figure, consider two arbitrary banks i and 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 (more on how exactly the controller services memory requests later): 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 all requests concurrently (in the sense the processing of requests is *interleaved*), always respecting the timing constraints between commands. All requests in the example are *row-misses*, since they are serviced with the sequence `PRE-ACT-CAS`, altough the `PRE` command for requests R_1 and R_2 are not shown in the figure. Note that R_3 targets a different row than R_1 in Bank i, since it needs to issue a `PRE` request. Bear in mind that in Figure 2.7, the arbitration, though valid, does not correspond to any specific algorithm, as it simply illustrates a situation where all constraints are respected. In the figure, the depicted constraints assume values greater or equal to these minimum values.

2.2.1 Memory Controllers

Memory controllers are hardware components connecting processing units (e.g. CPUs, GPUs, DMA-mapped devices, etc.) – also called *requestors* throughout this dissertation – to the DRAM. The controller receives memory requests from different sources as inputs, and generates memory commands to service these requests. More generally, a memory controller is usually responsible for the following tasks: **1)** *Address Mapping* – it maps physical addresses received from processors to actual elements of the DRAM device, i.e., bank-group, bank, row, and column; **2)** *Row-Buffer Policy* – it decides on how the banks' row-buffer will be managed; **3)** *Request Scheduling* – it implements a scheduling algorithm to decide in which order incoming requests will be serviced; **4)** *Command Generation* – it generates the commands to correctly service incoming requests; **5)** *Command Scheduling* – generated commands can be further subjected to a scheduling algorithm, typically aiming at minimising the average waiting time due to timing constraints; **6)** *Refresh Management* – it guarantees that cells are periodically refreshed; and **7)** It guarantees that the memory does not reach a faulty state (which may occur if invalid commands are issued) and that timing constraints are respected.

Figure 2.8 [28] shows an example of a memory controller – a design proposed by Ecco et al. [28]: note how incoming request, issued by processors, go through bank address mapping, per-bank request queues, request scheduling, command generation (*Bank Schedulers* in the figure), and command scheduling (*Channel Scheduler* in the figure). The controller in Figure 2.8 implements a specific algorithm [28], but its inclusion here is rather focused at giving the reader a structural view of what components *usually* constitute a memory controller.

Outside of scheduling policies, the two most relevant features of the controller are arguably *Address Mapping* and the *Row-Buffer Policy*. We discuss these two features in greater depth. There are 3 strategies for the mapping: *interleaved*, *shared*, and *private*.

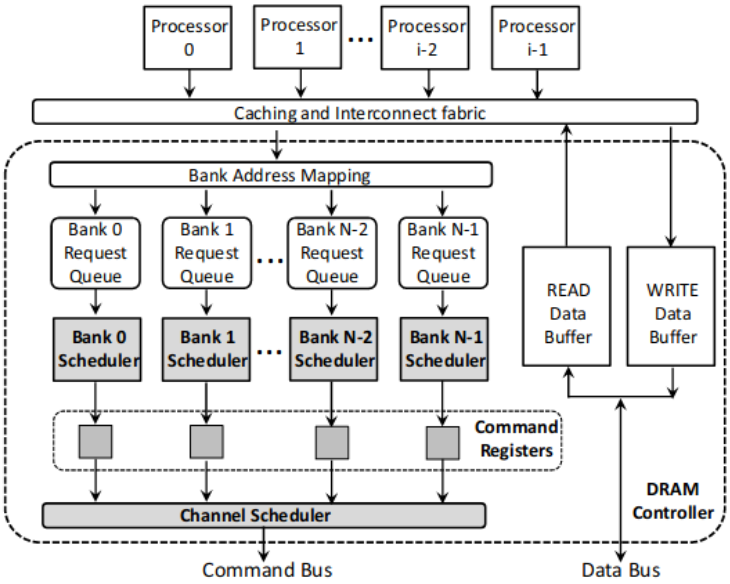


Figure 2.8: A memory controller proposed by Ecco et al [28].

In interleaved bank mapping [52], each request accesses all banks on a single transaction. In both shared [24], [26] and private [18], each request accesses a single bank on a transaction. The difference is that in the latter strategy, requestors are assigned exclusive access to one bank [28].

Regarding row-buffers, there are two main management strategies: the first is the *closed-row* policy [20], [21], [52], [53]. Under closed-row, the memory controller appends all `CAS` commands with the Auto-Precharge flag (`CRDA` and `CWRA`), which closes the utilised row right after the `CAS` is executed. This simplifies the timing analysis, because every request is served with a simple `ACT-CRDA`, `ACT-CWRA`, `ACT-CAS-PRE`, or `PRE-ACT-CAS` command sequence. However, this strategy is not optimised for bandwidth, since a request targeting an already open row would need only a `CAS`, thus reducing the total latency needed for it to be serviced. Conversely, the *open-row* policy [18], [24], [26] leaves the row buffer open for as long as possible. The row buffer is only precharged if a `REF` must be issued, or if a row miss happens. This requires the memory controller to keep track of the row currently open for each bank. Incoming requests are then translated to either a `CAS` command (if the required row is already in the row buffer) or to a `PRE-ACT-CAS` command sequence (if the required row is not in the row buffer) [28], for example.

2.3 Formal Verification

As mentioned in Section 1.1, the main problem we investigate and attempt to solve in this dissertation is *how to effectively design trustworthy real-time memory controllers?* Since critical (or mixed-criticality) real-time systems require a high level of assurance, we propose to tackle the development of such components with the aid of *formal methods*.

Formal methods, or *formal verification*, are a set of techniques used to perform mathematical/logical reasoning about programs or systems. In order to perform such reasoning, programs must be represented as well-known mathematical objects, such as transition systems or expressions in a programming language with formal semantics, which can, for instance, be an implementation of some (evolved) variant of Alonzo Church's λ-*calculus* [54] or Floyd-Hoare's logic [55]. In general, these two examples might be presented, more generically, as two approaches to formal verification: *Model Checking* and *Deductive Verification*. Keep in mind, though, that although model checking and deductive verification are two important branches of formal methods, there are other approaches as well, such as symbolic execution, abstract interpretation, and type systems. Here, we constrain the discussion to model checking and deductive verification due to their relevance and adoption in the problems discussed in this dissertation.

Model Checking consists, first, in representing programs, systems, or hardware designs as transition systems. There are several options regarding which exact mathematical model to use, which are all different flavours of transition systems, but have different definitions and applications, e.g., Büchi automata, *Petri Nets* [56], *Finite State Machines* (FSMs), *Timed Automata* [57], *Kripke Structures* [58]. Next, the user specifies properties to be checked using a formal *specification* language (or a *logic*) – which the program *shall* respect. Again, there are different options for specifying properties about transition systems, such as *Linear Temporal Logic* (LTL) [59] (which is very useful to state timing properties, such as fairness, discussed in Section 1); *Computational Tree Logic* (CTL) [60]; TCTL [61] – an extension to the syntax of CTL to include quantitative temporal operators; and *Property Specification Language* (PSL) [62].

The advantage of model-checking based verification is that tools typically work in a "push-button" manner, i.e., fully automatically. In other words, given a model and a set of properties described in a certain logic, the verification tools will either: 1) successfully verify that the property holds; 2) inform the user that the property does not hold, providing a counter-example; or 3) not converge to a solution (which can happen because the problem to be solved is computationally too large, a problem known as the *state-space explosion*). The fact that the verification effort is fully automated made the model checking approach

a popular choice for verifying systems in industrial settings – it is highly used in hardware design, for instance, where design mistakes discovered post silicon tape-out can be extremely costly, and should thus be avoided at all costs.

In summary, besides the model and property specification languages, a model checker includes a *verification procedure*, i.e., an intelligent algorithm that performs an exhaustive search of the model state that determines whether the specification is satisfied or not [63]. Formally, the model checking problem can be stated as:

▷ Let M be a state-transition system and let f be a temporal logic formula. The model checking problem is to find all the states $s \in S$ such that $M, s \models f$.

The origins of model checking date back to the early 1980s, when Clarke and Emerson [64] introduced CTL and a corresponding verification algorithm.[8] The algorithm's core works by computing for each stat s the set $label(s)$ of sub-formulas of f that are true in s. When the computation of such sets is finished, we will have that $M, s \models f$ iff $f \in label(s)$.

Because the algorithm explicitly accesses all the states of the transition system, this approach is also known as explicit-state model checking [63]. Since the number of states of a model can be enormous, this approach might not be practical for large systems. Research in both academia and industry performed throughout the years that followed have come up with several major advances regarding the state explosion problem. These techniques include *symbolic model checking* with *binary decision diagrams* (BDDs); *partial order reduction*; *counterexample-guided abstraction refinement* (CEGAR); and *bounded model checking* (which we refrain from discussing in detail, for brevity). These techniques allowed model checking to be a practical technique widely deployed in both research and industry.

Some examples of model-checking based verification tools are: the UPPAAL model checker [66] – in which models are modelled as timed automata (graphically) and in a C-subset (textually), and properties are written in TCTL; NuSMV [67] – in which models are defined in its own input language (based on writing transition relations of finite Kripke structures as expressions in propositional calculus) and properties are written in CTL, LTL, and PSL; SPIN [68] – in which models are written in a language called *Promela* and properties are written in LTL; and TLC [69], TLA+'s model checker – in which models and specifications are written in TLA+'s own language.

The second approach to formal methods discussed here is deductive verification. Deductive verification aims at formally verifying that all possible behaviours of a given program satisfy formally defined, possibly complex properties, where the verification process is based on some form of logical inference, i.e., "deduction" [70]. Several frameworks within this

[8] Model checking is also attributed to Queille and Sifakis, who independently developed a similar logic and verification procedure in 1982 [65].

paradigm exist: for instance, to verify imperative and object-oriented programs, formal systems derived from Floyd-Hoare [55] logic have been developed. For functional programs, formal verification systems are based on (highly evolved) variants of the typed λ-calculus. In this dissertation, we focus on the latter.

Typed λ-calculi are closed related to logic and proof theory via the *Curry-Howard Isomorphism* [71], which establishes a direct relationship between computer programs and mathematical proofs. This isomorphism, or correspondence, can be summarised by the following statement: "*a proof is a program, and the formula it proves is the type for the program*" [72]. The correspondence has been the starting point of a large spectrum of formal systems, designed to act both as *proof systems* and as *typed functional programming language*. Such formal systems are also called *proof assistants*, or *theorem provers*.

The verification workflow with theorem provers typically goes as follows: 1) write a program using a typed functional programming language; 2) write a set of logical statements (or predicates) about the program; 3) write a proof script to show that the program obeys the stated properties (in the same language or using a more convenient language which uses *tactics* to manipulate proof states). Each predicate about the program is also called a *proof obligation*, and the whole set of proof obligations is a part of the *formal specification*.

In modern proof assistants, the degree of automation w.r.t proving statements varies. While some tools offer none at all, some other can automatically prove a statement by connecting to SMT solvers – engines specialised in automatically solving problems stated as logical formulae – here, we enter the realm of *Automated Theorem Provers* (ATPs) [73]. This kind of automation, however, is limited to statements involving well known decidable theories, such as linear integer arithmetic, and *Boolean satisfiability*. Complex non-decidable statements (which is the case of most interesting verification problems) will very rarely directly benefit from that sort of automation without at least some degree of human intervention.

While theorem provers do not offer the same level of automation as model checkers, they have several advantages:

1. Programs and specifications are typically written in a full fledged functional programming language. In its core, specification languages for (some) proof assistants typically rely on calculi capable of encoding *higher-order logic* (HOL) (e.g., the *Calculus of Inductive Construction* [74]), which makes them very expressive and useful for reasoning about undecidable problems;

2. Since formal proofs are usually not computationally intensive, deductive verification typically scales better than model checking, i.e., using quantified statements, and other

techniques such as structural induction, one can prove properties about arbitrarily large/parametrized systems;

3. Certified (proved) programs can be *extracted* to another language, which can then be used as trustworthy components in larger developments;

4. Some proof assistants, specifically the one we use in this work, comply with the *de Bruijn criterion* [75] – which is said to be satisfied when the generation of the proof (i.e. the part of the software that helps us construct the proof) and the checking of the proof (i.e. the part of the software that checks whether a proof is correct) are independent. In other words, one needs to trust just a small proof-checking kernel (of around 5000 lines of code – for the Coq proof assistant) in order to trust mechanised proofs, which makes them very reliable. This is not the case for many automatic SMT-solvers, which have a "trusted code-base" of around half a million lines of code (e.g., Z3 [76] and CVC4 [77]).

In this work, particularly, we use the Coq proof assistant, which is based on the *Calculus of Inductive Constructions* [74]. Furthermore, a compelling set of arguments on why to choose Coq over other proof assistants has been presented by Adam Chlipala in *Certified Programming with Dependent Types* [78], which is summarised here (presenting only the most relevant arguments and without getting in too much detail):

1. Coq is based on a Higher-Order Functional Programming Language – which allows users to work with functions over functions;

2. Coq has full support for *Dependent Types* – which allows users to include references to programs inside of types and pass proofs as arguments to programs. For instance, the type of an array might include a program expression giving the size of the array, making it possible to verify the absence of out-of-bounds accesses statically [78];

3. Coq satisfies the previously mentioned *de Bruijn* criterion – which means that only a small kernel has to be inherently trusted;

4. Coq provides convenient proof automation – which simplifies the "engineering" challenges of writing large certified software developments.

Next, we include a brief introduction to Coq, which attempts to convey an intuitive notion of how the development of certified programs works in practice. Bear in mind that the following text is more closely related to the *program verification* approach to Coq rather than a pure mathematical use of the tool.

2.3.1 Coq

One of the key Coq features we rely upon in Coq is *Type Classes*. A type class in Coq is, in some regards, similar to abstract classes or interfaces from object-oriented programming languages. It allows to specify a set of members (i.e., attributes and methods) that are grouped together. As a running example, we will model a container interface – similar to the Container[9] and Collection[10], in C++ and Java, respectively. Listing 2.1 shows the container interface in Coq.

Listing 2.1: 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 (e.g., natural numbers, booleans, real numbers, other data structures, et cetera) and `C` the type of the data structure holding those elements (e.g., linked lists, hash maps, et cetera). The identifier `mkContainer` is the class constructor – a function used to build instances of the type class. 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 these functions are 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.[11] 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 conventional programming languages – they can be used as

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

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

[11] The POs in Listing 2.1 are not exhaustive though, i.e., the correctness of `container_t` could be stated through many other properties. `app_in` and `app_inc` are just examples of such properties.

parameters in functions and other type classes, where its members can be used.[12]

Listing 2.2: Inductive list definition.

```
Inductive list (T : Type) : Type :=
  | nil : list T
  | cons : T → list T → list T.
```

We may now begin to write concrete container implementations. To hold elements, we will use Lisp-style, polymorphic, inductive single-linked lists, defined in Listing 2.2. The inductive list is parameterised over type `T`, i.e., it can hold elements of any arbitrary type T. Note that `list` is defined by a *recursive inductive type*. In Coq, an inductive type, briefly, is a data type for which one or more constructors are defined. `list`, for instance, has two constructors: `nil` – used to build an empty list; and `cons` – used to build a new list from a single element followed by another list. The constructors of an inductive type can be seen as functions used to build new terms of the inductive type. Moreover, inductive types can be recursive, meaning that the type definition uses the definition of the type itself and parametric over other types (categorising *polymorphic* inductive types). The *list* example is both recursive and polymorphic.

Listing 2.3: A first attempt at implement an append function.

```
Fixpoint my_append T (x : T) (l : list T) : list T :=
  match l with
  | nil ⇒ cons T x (nil T)
  | cons hd tl ⇒ cons T hd (my_append T x tl)
  end.
```

Next, an append function can be implemented as shown in Listing 2.3. The recursive function `my_append` takes an element `x` of the type parameter `T` and appends it at the end of a list `l` – which holds elements of the same type `T`. The core concept used in the function definition is *pattern matching*: since `list` is a (recursive) inductive type, we can use a `match` to reason by cases. When the list is `nil` (empty), then the resulting list will simply be the

[12]Furthermore, sometimes, instead of type classes, we use Coq *Records*. Records are simple structures with fields – similar to C *structs* and record types used in other programming languages. In fact, apart from type classes with a single method – so called *singleton classes* –, each type class definition gives rise to a corresponding record declaration and each instance is a regular definition, i.e., a type class is a special type of record. In short, the two important differences between both are: 1) Records support field syntax, i.e., accessing individual fields directly from an object of the record type; and 2) Type classes employ more advanced inference mechanisms. Overall, as a general rule of thumb, thanks to its advanced inference mechanisms, type classes are often used to model interfaces that can be used abstractly, while records are used more as a structure in the typical sense. Concretely, Chapters 4 and 5 explain, in our work, when and why records have been used instead of type classes.

new element `x` followed by an empty list (`nil T`); when the list is already made of an element `hd` and the list `tl` (i.e., a `cons`), then the resulting list will be the first element (`hd`) followed by a recursive call to `my_append`. The function will make successive recursive calls until an empty list is passed as argument to `my_append`.

Listing 2.3 is not very readable, however. It is obviously burdensome to work with lists using `nil` and `cons`. We would rather prefer to have a more intuitive notation system that allows us to handle lists more visually. In addition, it is inconvenient to have to specify the type of the constructor's arguments each time – it would be better if Coq already knew that, within a function, whenever we refer to `nil` and `cons`, the type of elements is implicit. Solutions to these problems are shown in Listing 2.4, in which we use two Coq features that will ease the manipulation of lists.

Listing 2.4: Changing arguments and defining notation.

```
(* Makes T an implicit argument of nil and cons *)
Arguments nil {T%type} : rename.
Arguments cons {T%type} : rename.

Infix "::" := cons. (* "cons a b" can be written as "a ::b" *)
Notation "[::]" := nil. (* "[::]" can replace "nil" *)
Notation "[ ::x1 ]" := (x1 ::[::]).
Notation "[ ::x & s ]" := (x ::s).
Notation "[ ::x1 , x2 , .. , xn & s ]" := (x1 ::x2 ::.. (xn ::s) ..).
Notation "[ ::x1 ; x2 ; .. ; xn ]" := (x1 ::x2 ::.. [:: xn] ..).
```

There are (mainly) two things happening in Listing 2.4. First, the Coq command `Arguments` allows us to change some aspects of arguments of previous definitions. Here, we say that `T` will now be an implicit argument of `nil` and `cons`, i.e., it should be inferred automatically from the type of elements held by the list. Second, `Infix` and `Notation` allow us to define symbols that can be used instead of `nil` and `cons`. The "::" infix, for instance, will allow us to write `a :: b` instead of `cons a b`.

The append function can now be rewritten as shown in Listing 2.5. Note that Listing 2.5 also uses the `Section` mechanism. Briefly, `Section` is a shortcut for writing multiple definitions that share the same parameters. Since we will define many functions using the same type parameter `T`, we define a new `Section`, in which `T` is a `Variable`. When referring to terms that were defined inside of a `Section` outside of that same `Section`, every `Variable` becomes a parameter.

Next, we implement functions `contains` and `size` (Listing 2.6). There is a twist, however: `my_contains`, the implementation of `contains`, needs to compare elements of the parametric

Listing 2.5: Rewriting the append function.

```
Section Implementation.
    Variable T : Type. (* define a type to be used as parameter in
       the following definitions *)
    Fixpoint my_append_ (x : T) (l : list T) : list T :=
       match l with
       | [::] ⇒ [:: x] (* produce a list of single element x *)
       | hd :: tl ⇒ hd ::(my_append_ x tl) (* hd followed by recursive call to
          my_append_ *)
       end.
```

type `T` (previously defined through the command `Variable`, in Listing 2.5), i.e., it needs to decide if two elements of type `T` are equal.

Listing 2.6: Implementing `contains` and `size`.

```
Context {EqT : ·EqDec T eq eq_equivalence}.

Fixpoint my_contains (x : T) (l : list T) : bool :=
    match l with
    | [::] ⇒ false
    | hd :: tl ⇒ if (hd = x) then true else my_contains x tl
    end.

Fixpoint my_size (l : list T) : nat :=
    match l with
    | [::] ⇒ 0
    | hd :: tl ⇒ (my_size tl) + 1
    end.
```

The problem is that `T` might be a type with no *decidable equality*, i.e., no algorithm is known to determine if two terms of such type are equal. We must, therefore, tell Coq that such decidable equality does exist. To achieve that, we use the command `Context` to introduce `EqT`, an instance of the `EqDec` class. In other words, we affirm that within the current `Section`, the decidability of elements of type `T` is true, as an axiom. Having `EqT` in the context allows us to use a comparison between elements of type `T` in an `if` clause, i.e., since there is an algorithm to decide if elements of type `T` are equal, we can write decision procedures based on such comparisons. Later, if one wants to execute such functions, the actual type `T` must be instantiated, and of course, a concrete instance of the `EqDec` class must also be provided for the chosen instance of type `T`. For Coq's basic types, these instances already exist and Coq

Listing 2.7: Proving `my_app_inc`.

```
Lemma my_app_inc :
    forall (x : A) (l : list A), my_size (my_append_ x l) = (my_size l).+1.
Proof.
    intros x l.
    induction l.
    { cbn; reflexivity. }
    { simpl. rewrite IHl; rewrite !addn1; reflexivity. }
Qed.
```

can find them automatically. The actual reasoning behind the definitions of `my_contains` and `my_size` being quite straightforward, we let the task of analysing their implementation to the reader.

Having implemented all of the three functions specified in Listing 2.1, the only thing left to do is prove that `app_inc` and `app_in` are true for our implementations (see Listing 2.1). Listing 2.7 shows the proof of Lemma `my_app_inc`, a statement of the same type as `app_inc`. The code between the keywords `Proof` and `Qed` is a proof script, written in a *tactic* language called `Ltac`, which is implemented in OCaml and can be used to prove statements interactively. After `Proof`, Coq goes into *proof mode*, where `Ltac` tactics can be used to manipulate the proof state.

The proof of `my_app_inc`, specifically, is by induction on the structure of `l`, the inductive list. In more detail, first, we use the tactic `intros` to give names to the variables bounded by the `forall` quantifier – thus getting rid of the `forall` and continuing the proof for arbitrary `x` and `l`. Next, `induction l` will create two sub-cases – one for the base case (when the list is `nil`), and another for the recursive case (when the list was constructed using `cons`). Figure 2.9 shows the state of the proof after using the `induction` tactic. To the left of the figure is the editor (in which it is possible to see that the file has been compiled up until the highlighted line); and to the right of the figure is the *proof state*.

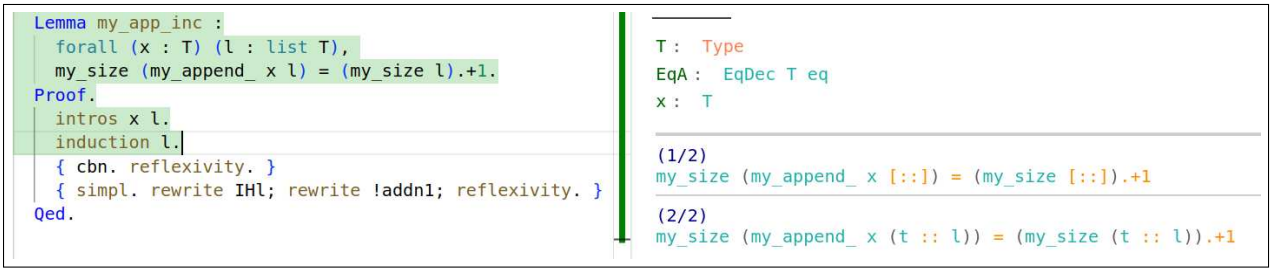


Figure 2.9: Proof state after performing induction.

On the right side of Figure 2.9, above the grey horizontal line, are all of the *known* facts,

i..e, everything in the *local context*. After executing `induction l`, there are three things in the context: the type parameter `T`; `EqA` – the instance of the `EqDec` class – which allows Coq to know that equality between terms of type `T` is decidable; and `x`, an arbitrary bound variable of type `T` (introduced by the `intros` tactic). Below the horizontal line, there are two *goals* to be proved: for the base case of the induction, one can see that `l` has been replaced by the empty list ([::]), and for the recursive case, by `t :: l`, where `t` is an element of type `T` and `l` is of type `list T`. We solve one case at a time by isolating them with curly braces "{}" in the proof script. For the base case, we use a smart reduction tactic, `cbn`, which can compute both sides of the equality up to $1 = 1$, which can be solved by `reflexivity`, a tactic capable of finishing goals that involve equalities where both sides are syntactically identical.

For the recursive case, we first use another reduction tactic `simpl` to simplify the goal. In this case, just by applying arguments (i.e., *Beta reduction*) on the left-hand side (LHS) of the equality, Coq knows that:

▷ `my_append_ x (t ::l) = t :: my_append x l`,

And hence, that

▷ `my_size (t ::my_append x l) = my_size (my_append_ x l)+ 1`.

The right-hand side (RHS) can be simplified likewise. The state of the proof after simplifying the goal can be seen in Figure 2.10 (note the induction hypothesis above the horizontal line on the right hand side of the figure). After that, it suffices to use the induction hypothesis and apply a known arithmetic lemma, `addn1`, to get to the end of the proof. `addn1`, in mode detail, states that `(x + 1) = x.+1`; it comes from Coq's `mathcomp` library and establishes an equality between addition with an unit `(x + 1)` and the successor function `(x.+1)`. The proof of `my_app_in` is very similar, and is therefore, omitted for the sake of conciseness.

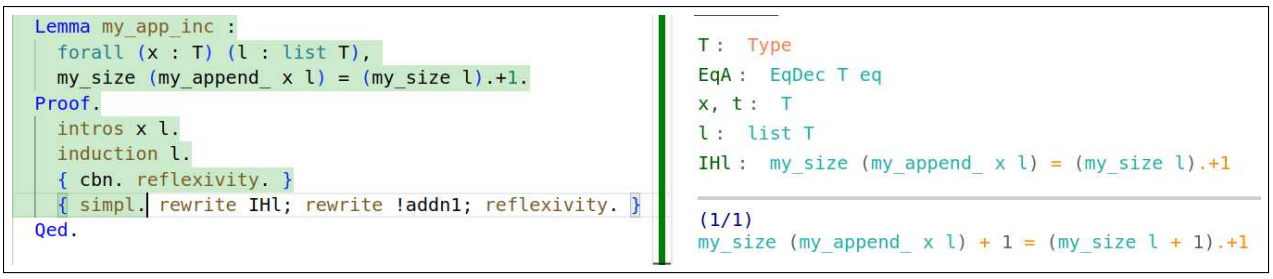


Figure 2.10: Proof state after simplifying the recursive case of the induction.

Finally, having implemented the functions specified by the `container_t` interface and proved that the proof obligations hold for the implementations, we can call victory and instantiate the whole class `container_t`. Here, the advantage of having written functions

with parametrized types becomes evident: we can instantiate the class for any type we want, under the condition that a decidable equality exists for that type!

Listing 2.8: Creating instances of `container_t`.

```
Section CreatingInstances.

    Instance my_container_nat : container_t := mkContainer
        nat (* C : nat, type of elements *)
        (list nat) (* T : list, type of data structure holding nats *)
        (my_append_ nat) (* first function *)
        (my_contains nat) (* second function *)
        (my_size nat) (* third function *)
        (my_app_inc nat) (* first proof *)
        (my_app_in nat). (* second proof *)

    Instance my_container_bool : container_t := mkContainer
        bool
        (list bool)
        (my_append_ bool)
        (my_contains bool)
        (my_size bool)
        (my_app_inc bool)
        (my_app_in bool).

    Instance my_container_natlists : container_t := mkContainer
        (Datatypes.list nat)
        (list (Datatypes.list nat))
        (my_append_ (Datatypes.list nat))
        (my_contains (Datatypes.list nat))
        (my_size (Datatypes.list nat))
        (my_app_inc (Datatypes.list nat))
        (my_app_in (Datatypes.list nat))

End CreatingInstances.
```

In Listing 2.8, we instantiate `container_t` for three basic Coq types: `nat`, `bool`, and lists of natural numbers, which is actually a `list` type identical to the one we have defined in this text, with type parameter instantiated with `nat` (such a type had actually already been defined in Coq's standard library; we refer to the standard library implementation

of the list as `Datatypes.list`). In the latter case, the type of elements of our inductive list is a `Datatypes.list nat`, which makes it a two dimensional structure. Note that, even though types have changed, we pass the same functions and proofs as arguments to create all instances. Moreover, for all of these types, Coq can automatically find an instance of the `EqDec` class, which defines the decidability of equality between terms.

Chapter 3

Related Work & Motivation

First, in Section 3.1, we discuss the advantages and drawbacks of other approaches that aim at verifying the correct behaviour of DRAM systems – and compare these approaches with our own. In other words, we discuss *how the problem has been solved by other formal-verification-based methodologies*. Next, in Section 3.2, we rather look at *how our methodology (or similar) has been used to deal with related problems*. Finally, before diving into the technical contributions of this dissertation, Section 3.3 analyses what we describe as a general *lack of formalism* in recent literature, a point that highly motivates the work described in this dissertation.

3.1 Verification of DRAM Systems

The idea of applying formal property checking to DRAM controllers was first proposed by Datta and Shinghal, back in 2008 [79]. They wrote a set of *SystemVerilog Assertions* (SVA) to formally verify that the Sun OpenSPARC DDR2 controller[1] is compliant with the DDR2 JEDEC standard. The problem is that the SVA properties are (arguably) stated in a less natural way, i.e., there is a larger semantic gap between the temporal logics of SVA and the natural language (English) used in the standard, compared to what can be accomplished with a language such as Coq. Moreover, model checking SVA properties suffers from the well-known scalability problem – if a memory controller design is large, the model checker may fail to converge.

More recently, Jung et al. [80] addressed the problem of *automatically* verifying whether DRAM controllers or simulators *conform to any given JEDEC memory standard*. The work is motivated by the recent increase in the number of different main memory standards – by

[1] https://www.oracle.com/servers/technologies/opensparc-overview.html

automatically verifying that a controller implementation conforms to a given standard, one avoids the work of tediously checking if output traces conform to the standard. To achieve that, they developed a formal mathematical model based on *Timed Petri Nets* [81], [82], which contains DRAM states, transitions and timings. Moreover, they present a DSL – *DRAMml* – for describing the memory functionality and timing dependencies of a JEDEC standard. From a description written in DRAMml, an executable Petri Net is generated automatically (through a *correct-by-construction* process), which can be used to perform fast simulation-based validation of memory controllers and DRAM simulation models.

In 2022, Jung et al.'s work was extended by Steiner et al. [83], who proposed the generation of SVA from DRAMml – which can be used to perform the formal property checking of a memory controller – for instance (instead of the previously used simulation-based validation). In addition, like Datte and Shingal, Jung's and Steiner's work do not cover the validation of scheduling algorithms (in terms of high-level properties), it is rather an approach to verify that a memory controller conforms to the JEDEC standards.

While Jung's and Steiner's approach presents an effective way to automatically verify controllers' conformance to the JEDEC standards, it is limited to that. In other words, the DRAMml description cannot be used to express properties of different nature. In our approach, thanks to the expressiveness of Coq, we can use our model to not only verify conformance to the JEDEC standards, but also to derive worst-case latency bounds and prove high-level properties – such as non-starvation, sequential consistency, handling of data dependencies, and execution of atomic operations. Furthermore, a promising way to leverage results from Steiner's and Jung's work would be to automatically generate Coq POs from DRAMml. This would make it possible to extend CoqDRAM to cover several different JEDEC standards. This is further discussed in Chapter 10.

Furthermore, the core of Steiner's idea [83], i.e., capturing design specifications from JEDEC standards and automatically generating SVA, had been first introduced by Kayed et al. [84]. Kayed et al. describe the properties from the JEDEC standards using *Timing Diagram Markup Language* (TDML). Kayed's methodology, however, requires modifications to the RTL implementation to insert the generated SVA and was not complete w.r.t the transitions and timing constraints defined in the JEDEC standards.

Li et al. [85] model a DDR3 device and a memory controller using *Timed Automata* (TA) in the UPPAAL model checker [66]. The DRAM model captures timing constraints and the controller model captures the underlying scheduling and mapping algorithms. Moreover, the UPPAAL models are used to derive the *Worst-case response time* (WCRT) and *Worst-case bandwidth* (WCBW), which are thoroughly checked against the output of a cycle-accurate simulation tool under the same inputs, thus building confidence that the TA models are

correct. The UPPAAL analysis reduces the WCRT bound by up to 20% and improve the WCBW by up to 25% compared to the (then) state-of-the-art [86].

While Li et al.'s work successfully analyses worst-case metrics with a high level of confidence, it has drawbacks: 1) The formalisation in UPPAAL is specific to one controller [27], and modelling another controller with the same approach would require substantial work; 2) Due to state space explosion, UPPAAL fails to analyse the WCRT and WCBW of an arbitrary mix of transactions, and the authors were forced to limit requestors to issue at most one outstanding transaction (thus excluding the possibility of out-of-order cores in the system) in order to keep the state space manageable; 3) Although the authors do validate their models by comparing traces and worst-case metrics with the ones outputted by a cycle accurate simulator, the models lack readability. Hence, it remains (arguably) hard to get convinced that the TA models reflect the actual DRAM states specified in the JEDEC standards. Our work addresses all of the drawbacks mentioned above.

In an entirely different approach, Hassan et al. [87], [88] propose an automated framework for simulation-based validation of memory controllers, called *MCXplore*. Differently than the previous two approaches, MCXplore uses model checking (NuSMV [67], specifically) to generate test plans/suites for memory controllers. MCXplore's main advantage is the fact that it is agnostic to the actual design of the memory controller.

While Hassan's approach is quite valuable for generating meaningful test-benches, the approach still relies on the effectiveness of validation engineers at translating test plans (written in natural language) to temporal logic formulas, which serve as input to NuSMV for the generation of the test templates. Moreover, nothing guarantees that the models used to generate the test templates are correct against some sort of specification (such as the JEDEC standards). Conversely, the logical properties that define correctness in our framework are plain and simple, and easy to check against the JEDEC standards. If the model of a memory controller designed using our framework is incorrect in any way, then the correctness formulas will simply not hold; but the correctness of the formulas themselves are easy to check. In other words, our approach defines the meaning of *correctness* more straightforwardly.

Furthermore, Sahoo et al. [89] have performed interesting research in formalising *DRAM cache controllers* (DCCs) (DRAM modules can be stacked, conforming die-stacking technology, to build high bandwidth and low-latency memory, which can be used as last level cache in a system). DCCs, however, different than typical memory controllers used to handle main-memory DRAM, carry additional complexity, since not only the controller has to manage DRAM commands and related timings, but also *manage* the content of caches (the *metadata*). Given such complexity, the authors use bounded and symbolic model checking

to verify safety, liveness, and timing properties about DCCs, expressed in LTL.

The authors succeed in modelling properties of different nature with LTL and verify that multiple instances of DCCs comply (or not) with the stated properties. However, the approach suffers from a scalability issue (as it is typical of model-checking-based verification projects). Beyond 64 rows per DRAM bank, the authors report that the main-memory of the machine used to run the verification was saturated (with 32 GB RAM). Moreover, the authors do not propose a method to generate executable code or synthesizable RTL from their models – which makes its practical usability limited.

It is important to state the limitations of our approach as well – the most important one being arguably the process of proving properties. Although we do make an effort in the design of our framework to implement some proof automation, it is still burdensome at times to prove even the most straightforward properties. Moreover, programming in Coq requires patience, as it has quite a steep learning curve.

3.2 How theorem provers have been used to deal with related problems

Here, we look at how recent research has treated the problem of applying deductive verification to related problems.

Coq, and other similar theorem provers, such as Isabelle/HOL [90] and F* [91] have been used successfully to model and prove the correctness of complex systems. PROSA [92], 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. [93] 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 et al. [94] 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 re-developed 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 coherent with the arguments we present in this work, which aims at providing

a formal foundation for memory controller design. In 2022, Maida et al. [95] extended the results from Bozhko et al. [94] 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[2] *silicon root of trust*. The hardware is implemented in Cava (described in more detail in Appendix A). Kami, a similar system, was first developed by Choi et al. [96] and later adopted by SiFive [97]. It has its roots in Bluespec,[3] 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 dissertation.

Following the same rule-based design approach, two recently proposed DSLs Koîka [98] and Hemiola [99] extend the foundations of 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 pipelined system. The latter proposes a method to prove the *serializability* property of cache-coherence protocols.

Furthermore, ReWire [100] is also a DSL implemented in Haskell, which has a back-end compiling to VHDL. The provability aspect in ReWire is quite flexible: logical properties exported from Haskell can be proved in any theorem prover supporting infinite streams, such as Isabelle, Coq, and Agda [101]. In addition, the "fidelity" of ReWire models, differently than the previously mentioned Coq DSLs, can also be automatically checked against an existing Verilog design with Yosys [102] by model-checking. CλaSH [103] is yet another functional hardware description language that borrows both its syntax and semantics from Haskell. Although CλaSH is an interesting project that has been used to develop several complex designs, it overlooks the verification problem. In this work, we explored using Cava to propose a link between CoqDRAM developments and their hardware counterparts. This exploration is further discussed in Appendix A.

[2] https://opentitan.org/
[3] https://bluespec.com/

3.3 How proofs are typically written, and why we should do better

Latency-analysis (or timing analysis, more generally) 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, which can be hard to deal with, in more than one aspect.

We analysed the most-often cited (at the time of writing) real-time memory controllers in the literature regarding the length of latency analysis/proofs [17], [18], [20]–[22], [24], [27]–[29], [31], [32], [34], [104]. 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 developments from the reader, i.e., the formalisation can be presented with the appropriate level of detail.

Moreover, these analyses are often presented for the simplest of cases, leaving out essential details. As an example, Mirosanlou et al. [104] 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 et al. [28] proceeds in the same way, omitting the proof of a lemma based on the similarity argument.

In other work, the timing analysis is based on assumptions that reduce the set of valid scenarios. For instance, work by Guo et al. [34] describes their timing analysis as being only valid for a subset of DDR3 devices.[4] Work by Wu et al. [24] 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

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

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. [16], 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 et al. [28], 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.

Although *we do not deem wrong* nor contest the authors' 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. Oppositely, Coq proofs, besides being inherently more trustworthy, are also reusable to some extent, meaning that new controller developments should benefit from a library containing components and proofs.

- 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. Conversely, Coq forces one to mall assumptions explicit, and even allows to find them automatically.

- The fact that there is no formal link between system implementation and the mathematical abstractions used to perform timing analysis may also introduce untrustworthiness. In other words, there is no guarantee that the paper and pencil analysis of a system, or the implementation of a system, correctly captures the real behaviour of the system. Using Coq, proofs and programs/models are tightly coupled.

Chapter 4

An Overview of the Framework

The objective of this chapter is to discuss the top-level functionality of the framework proposed in this dissertation – CoqDRAM – and the rationale behind each of its constituting parts. Then, in the following chapters, we dive into the Coq implementation, design decisions, nuances, and detailed functionality of each part.

Figure 4.1 presents an architectural view of the framework. In the figure, each coloured rectangle is a "logical" element of the framework, being encoded in Coq as either a type class, a record, or merely a set of definitions grouped together. Moreover, names in parentheses/typewriter indicate Coq classes/records modelling the respective concept. Note that the only elements that are not type classes or records are *Interface Sub-Layer* and *Bank Machine*, which are rather just a set of definitions (types and functions) grouped together to form a logical component.

We start by discussing the specification part of the framework, followed by the implementation part. Moreover, as a clarification about naming elements, since we already refer to the class `Implementation_t` as being the "Implementation Interface", we refer to everything related to writing implementations, which refine the specification (i.e., proved scheduling algorithms), as the "*implementation world*" and everything in the formal specification as the "*specification world*".

Furthermore, throughout the discussion in the present and following chapters, we use the term *implementation* to describe an actual algorithm that has been implemented and proved. In the figure, these are represented as circles. More generally, we use the verb "*to implement*" interchangeably with "*to refine*" and "*to instantiate*" to describe the Coq mechanism of instantiating a type class and discharging its proof obligations. Sometimes, we use the term "*abstract instance*" referring to an implicit argument introduced in a `Section` (see Listing 2.6).

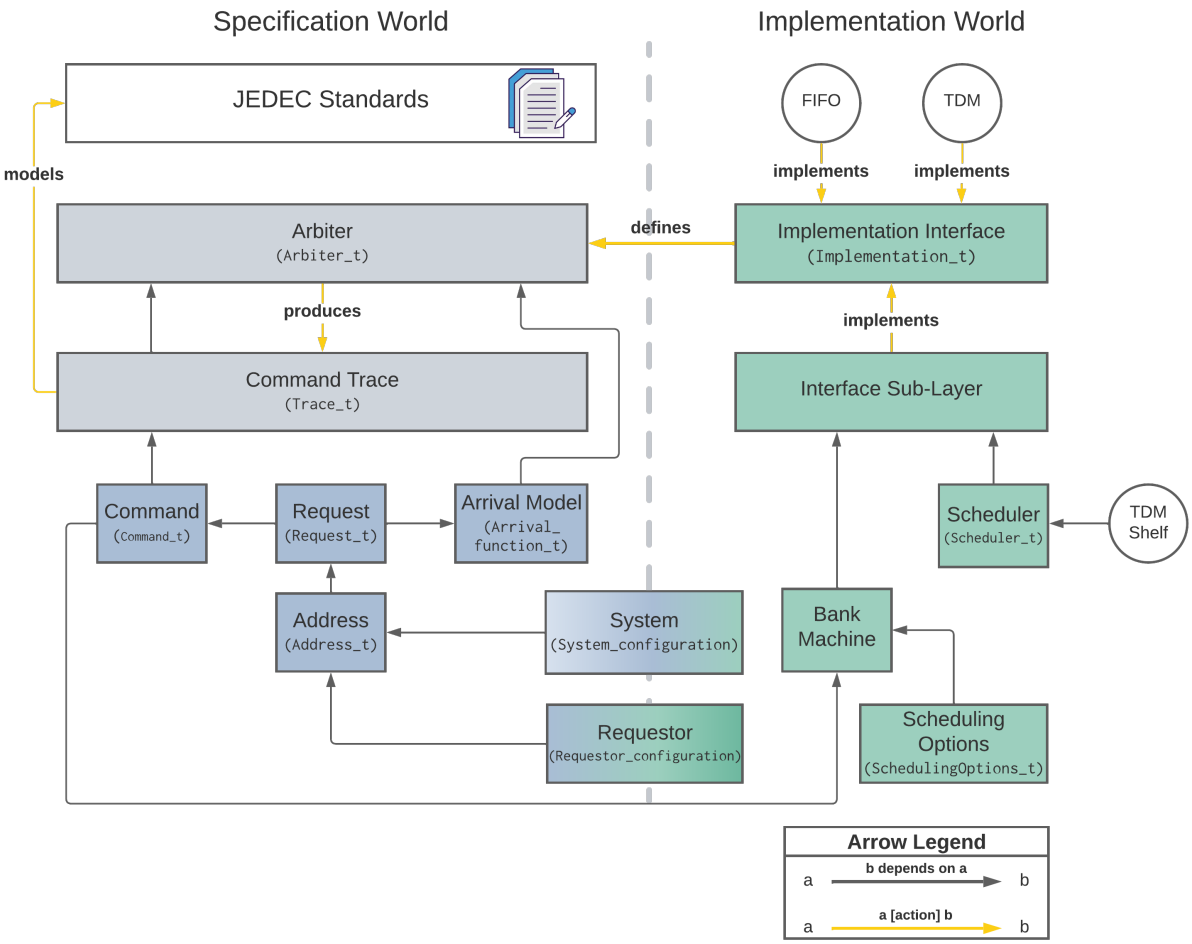


Figure 4.1: CoqDRAM Architecture.
"Building Blocks"/Data structures,
Main Logical Elements, Implementation-related Elements

4.1 Specification

Command. `Command_t` is a record representing the type of memory commands (sent to the DRAM device). Each command (as it will be shown in detail later in the text) contains an issue date and a kind (i.e., whether it is an `ACT`, `CAS`, et cetera). Moreover, some commands may also carry information concerning the underlying request from which that command originated.

Command Trace. The framework is centred around the `Trace_t` class, which represents the arbitrarily large sequence of commands sent to the DRAM device. `Trace_t` captures correctness criteria coming from the JEDEC standards as proof obligations over such sequence of commands. Intuitively, it can be said that the "objective" of the entire framework is to build a *correct trace* of DRAM commands.

Arbiter.[1] The `Arbiter_t` class is used to build correct traces, i.e., concretely *produce* traces. Note that, in the figure, the black arrows denotes only a module dependency, i.e., the module containing the trace definition must be compiled before the module containing the arbiter definition. That makes sense, since an arbiter only makes sense if the definition of a trace already exists. The `Arbiter_t` class is centred around a function that takes a non-zero natural number `n` as parameter and produces a trace of commands of length `n`.

It is at this point that higher-level properties can be stated as proof obligations. For instance, we *impose* a proof obligation on `Arbiter_t` saying that every request that has arrived in the system must be eventually serviced (completed). In the most up-to-date version of the framework (in development at time of writing), however, we adopt a different approach: more generally, we define different classes of arbiters, all of which produce a trace of commands, but impose different proof obligations on the traces they produce. For example – while a strict real-time controller might require completion for every request, a mixed-criticality controller might abandon already pending low-criticality requests on a mode change, or only accept requests above a certain criticality depending on its current mode. Moreover, for controllers used in multi-processor systems, one might want to write a controller that implements sequential consistency (or other memory consistency models) between memory accesses.

Writing the infrastructure allowing the user to implement these different classes of arbiters stating different *high-level* proof-obligation is ongoing work (at time of writing). The frozen version of the framework only includes one mandatory proof obligation about non-starvation,

[1] In this dissertation, we use the term *arbiter*, interchangeably with *scheduler*, to refer to the actual scheduling algorithm inside of a memory controller.

i.e., every implementation has to present evidence (a proof) that requests cannot starve. For now, we have also modelled an additional arbiter class, the one of sequential consistent arbiters, in which the completion proof obligation is complemented by a proof obligation stating that arbiters implement sequential consistency. Finally, `Arbiter_t` also defines some handy proof obligations aimed at facilitating later proofs.

The bottom-most blocks in Figure 4.1 are also labelled as *building blocks* – these are type classes used to represent real life entities/data-structures (such as commands, requests, and addresses) and useful abstractions (such as the arrival model).

System. The `System_configuration` class is a building block that is part of both specification and implementation worlds. It defines the system parameters, such as number of the banks and bank-groups in the memory device, timing constraints values, and axioms guaranteeing that the system is feasible (e.g., that timing constraints are non-zero, that there is at least one bank and one bank-group in the system, et cetera)

Requestor. The `Requestor_configuration` class only contains a type declaration – the type of requestors (processing units capable of issuing memory requests). The rationale is that designers should be able to represent requestors according to their needs. For instance, for some classes of systems, requestors might have different criticality levels, while for others, that is not the case. Like `System_configuration`, `Requestor_configuration` is part of both the specification and implementation worlds. For the specification world, although the concept of a *requestor* needs to exist, the actual requestor type is not relevant. In practice, an abstract instance of `Requestor_t` (i.e., an implicit parameter inside a `Section`) is introduced throughout the specification. In the implementation world, however, concrete implementations need to refine the `Requestor_t` type class, since algorithms *may* rely on concrete information about the nature of requestors (such as a numeric "ID", criticality level, et cetera).

Arrival Model. `Arrival_function_t` is a class that models the arrival of memory requests in the system. It is a function that takes an arbitrary non-zero natural number `n` as argument and returns the set of requests that have arrived at that instant. The class also imposes a few proof obligations that are helpful on later proofs. The assumption in `Arrival_function_t` is that arriving requests have already been accepted by some arbitrary bus/interconnect protocol.

Address. `Address_t` is a record containing fields of custom defined types, such as the type of banks, the type of bank-groups, and the type of rows. Bear in mind that we do not model columns as part of the address, since the current framework (at time of writing) does not model data content (i.e., transaction/requests payloads). Although the delays related

to data transfers over the data bus are well taken into account, the data payload (either write or read data) is not relevant for timing purposes. Therefore, it suffices to analyse a request's row to determine if it is a row hit or miss, and thus, calculate which commands are needed to service the request. Modelling data, however, is something that could be done in the future, as it would allow the framework users to write memory controllers capable of reasoning about data integrity and error correction codes, for example.

Request. `Request_t` is a record representing the type of memory requests. Each request contains an address, a kind (read or write), and an arrival date.

4.2 Implementation

Implementation Interface. The class `Implementation_t` defines a standard way to write new controllers. In other words, it defines a sort of "skeleton" to write algorithms in the form of finite state machines. It includes the type definitions of states, set of initial states, the transition function, and a recursive function that calls the transition function to build a sequence of states. Each state of the state machine contains a list of commands, and at each new state, a new command is appended into the previous list of commands. This is the same as `Trace_t`, but without the proof obligations. Each call to this recursive function represents a bus clock transition on the real arbiter implementation. Differently put, `Implementation_t` can be seen as the real engine that creates the trace of commands, while `Arbiter_t` is its interface (specification) containing correctness criteria (POs). Chapter 6 presents and details FIFO and TDM, two PoC command scheduling algorithms written using the `Implementation_t` interface.

Next, the framework user developing a new memory controller can choose between two paths: either instantiate `Implementation_t` directly by defining a concrete transition function, set of initial states, and internal state structure; or use the Interface Sub-Layer abstraction – to be presented next.

Interface Sub-Layer. Because writing complex scheduling algorithms with such low-level abstraction is hard (i.e., via the state machine interface specified by `Implementation_t`), we provide a second way of implementing scheduling algorithms. Interface Sub-Layer defines a "*standard implementation*" for `Implementation_t`, i.e., it defines a standard transition function, a standard set of initial states, and a standard structure of internal states to mimic the evolution of the memory device. This standard transition functions uses a *bank machine* to keep track of the internal states of each DRAM bank, including row-buffer status, timing constraints between commands, and refresh timing.

This represents an important change of paradigm: instead of delegating to the user the work of proving that timing constraints between commands are respected, Interface Sub-Layer relies on the bank machine to only allow *valid* commands to be available for scheduling in the first place. That way, proofs about timing constraints and functional correctness are written only once, and new scheduling algorithms do not have to reason at the granularity of timing constraints and protocol correctness. Moreover, the only proofs left to do for new developments are related to high-level properties, such as non-starvation, defined in `Arbiter_t`.

Bank Machine. As mentioned above, Bank Machine is a set of definitions that are useful for keeping track of (mimicking) the state of the memory device. It contains type definitions, including the type of timers to account for timing constraints and refresh operations, and other state variables to keep track of things such as the row-buffer status and the direction of the data bus. In summary, the two goals of a bank machine are: 1) to provide a function to test if a given command can be issued to its respective bank at a given time `t`; and 2) to provide a function that receives a command to be issued to the device (relying on the hypothesis that such command is valid) and updates the state of the system accordingly (i.e., increments or resets counters, changes bus direction, changes the status of the bank, changes the information of which row is loaded into the row-buffer, et cetera).

Scheduling Options. `SchedulingOptions_t` is a type class used to *customise* the behaviour of the bank machine. It defines, for example, the row-buffer policy to be used (closed or open-page policy), which itself defines which commands are generated for each request.

Scheduler. Finally, `Scheduler_t` is a type class used in combination with the bank machines in the interface sub-layer. While the bank machine manages the DRAM's internal state, `Scheduler_t` actually allows the user to write command scheduling algorithms. Different than algorithms written directly by instantiating `Implementation_t`, instantiating the `Scheduler_t` class provides a higher-level of abstraction. To do so, the class specifies a function that can only choose from a list of *ready* (valid) commands – according to the bank machine. In Chapter 9 we present *TDMShelve*, a new DRAM scheduling algorithm which implements the interface described by the `Scheduler_t` class.

Chapter 5

A Deeper Look into the Framework

In this chapter, we discuss the implementation of the framework following a bottom-up approach (contrarily to the top-bottom view from the previous chapter). The reader is encouraged to have read the background section on Coq (c.f. Chapter 2).

System

`System_configuration` is intended to capture the parameters of the memory device. When presenting `System_configuration`, we take a step back and use the opportunity to discuss the scope of the framework, more broadly. We clarify exactly which parts of the standards we are modelling and which are excluded. We also add a brief discussion about other JEDEC standards. To begin, Listing 5.1 shows the definition of the `System_configuration` type class.

`System_configuration` starts by defining the number of bank-groups (`BANKGROUPS`) and banks (`BANKS`) in the system quite straightforwardly. Note the two proof obligations (POs) stating that there must be at least one bank-group and at least one bank in the system (in Lines 3 and 6, respectively). Moreover, this representation allows the framework to model controllers for both DDR3 and DDR4 devices, as DDR3 devices can be seen as a special case of DDR4 devices with a single bank-group, regarding most aspects.[1]

`System_configuration` also defines *command-related* timing constraints that can be found in the JEDEC DDR3 [41] and DDR4 [42] standards. We require all these timing constraints

[1]More precisely, compared to DDR3, DDR4 offers higher module density and lower voltage requirements, which allows smaller timing constraints. Furthermore, unlike previous generations of DDR memory, pre-fetch has not been increased above the 8n used in DDR3; the basic burst size is eight 64-bit words, and higher bandwidths are achieved by sending more read/write commands per second. To allow this, the standard divides the DRAM banks into two or four selectable bank groups, where transfers to different bank groups may be done more rapidly [42]. Although in the physical world these differences are non-trivial, in terms of a software model, these differences can be seen as only distinct values of timing constraints, and a DDR3 device is therefore a special case of a DDR4 device with only one bank-group and different timing constraints.

Listing 5.1: Definition of `System_configuration` (see Table 2.2).

```
Class System_configuration := {
  BANKGROUPS    : nat;
  BANKGROUPS_pos : BANKGROUPS > 0;

  BANKS         : nat;
  BANKS_pos     : BANKS > 0;

  T_BURST : nat;   (* delay of a burst transfer RD/WR *)
  T_WL    : nat;   (* delay between a WR and its bus transfer *)
  T_RRD_s : nat;   (* ACT to ACT delay inter-bank : different bank groups *)
  T_RRD_l : nat;   (* ACT to ACT delay inter-bank : same bank groups*)
  T_FAW   : nat;   (* Four ACT window inter-bank *)
  T_RC    : nat;   (* ACT to ACT delay intra-bank *)
  T_RP    : nat;   (* PRE to ACT delay intra-bank *)
  T_RCD   : nat;   (* ACT to CAS delay intra-bank *)
  T_RAS   : nat;   (* ACT to PRE delay intra-bank *)
  T_RTP   : nat;   (* RD to PRE delay intra-bank *)
  T_WR    : nat;   (* WR end to PRE delay intra-bank *)
  T_RTW   : nat;   (* RD to WR delay intra- + inter-bank *)
  T_WTR_s : nat;   (* WR to RD delay intra- + inter-bank : different bank groups *)
  T_WTR_l : nat;   (* WR to RD delay intra- + inter-bank : same bank groups *)
  T_CCD_s : nat;   (* RD/WR to RD/WR delay intra- + inter-bank : different BGs *)
  T_CCD_l : nat;   (* RD/WR to RD/WR delay intra- + inter-bank : same BGs *)

  T_RRD_s_pos : T_RRD_s > 0;
  T_RRD_l_pos : T_RRD_l > 0;
  T_FAW_pos  : T_FAW > 0;
  (* ... more POs of the same kind ... *)

  T_RRD_bgs : T_RRD_s < T_RRD_l;
  T_WTR_bgs : T_WTR_s < T_WTR_l;
  T_CCD_bgs : T_CCD_s < T_CCD_l;
}.
```

to be positive. Moreover, timing constraints that depend on whether commands target the same or different bank-groups have two versions: constraints ending with a "`_s`" (for "*small*") describe intervals between commands targeting different bank groups, and constraints ending with a "`_l`" (for "*long*") describe intervals between commands targeting the same bank group. For coherence, we also require the "`_l`" version of constraint to be strictly greater than their `_s` versions (Lines 30 to 32).

Modelling refresh commands

Note that refresh commands are not part of the specification (at this point). At first, as it is typical in the literature, we considered refresh commands merely as additional time interference and disregarded them in CoqDRAM. However, in a later stage, while deriving hardware versions of our implementations (part of an exploration described in Appendix A), we realised that refreshes were crucial for the correct functioning of real hardware components, and thus they should be modelled in CoqDRAM. Moreover, *refresh management* is an important algorithmic problem [105] for memory controllers, as the JEDEC standards allow a certain degree of flexibility regarding **when** to issue refresh commands. This flexibility can be used to optimise the average-case latency of memory requests, for example.

Hence, the most up-to-date version of the framework, described in Chapters 8 and 9, contains a formalisation of refresh commands, including timing constraints, associated POs, and a refresh management mechanism. The current chapter describes the framework without accounting for refreshes. The following chapter, which describes the two first PoC scheduling algorithms developed using the abstractions presented in this chapter, also does not mention refresh commands. In Chapter 8, we show how refreshes are modelled in the most up-to-date version of CoqDRAM. We also detail the refresh management mechanism part of the new implementation interface – Interface Sub-Layer.

Scope of the framework: exactly which features from the JEDEC standards are modelled?

It is important to note that we do not model *all* timing constraints in the JEDEC standards. For instance, the DDR4 standards defines a series of clock timing parameters, such as $t_{CK}(avg)$ (average clock period), $t_{CH}(avg)$ (average high pulse width), $t_{JIT}(duty)$ (duty cycle jitter), et cetera. Although these timing constraints are relevant for the correct behaviour of the memory, they are not usually compromised by algorithmic decisions taken at the request/command scheduling level. In other words, the correctness of these constraints/-parameters are in the realm of electronic design of the memory chip rather than that of

scheduling (algorithmic) decisions taken by a memory controller.

Other timing constraints/parameters are not included. For instance, the DDR4 standards include a Mode Register Set (MRS) command, which allows users to configure the DDR4 device by writing to seven registers, providing application flexibility (to dynamically change the rate in which refresh commands are needed, or the size of data bursts, for example). These MRS commands are also subject to timing constraints other then the ones described in Table 2.2 (namely t_{MOD}, t_{MRD}, et cetera), which are also not included in `System_configuration`. The reason for that is that the current version of the framework only models controllers that do not change their configurations dynamically, i.e., we assume that the controller is operating under a given configuration and that it will never change. Although such abstraction excludes an important feature, it still allows designers to reason about the correctness of scheduling algorithms under a given configuration. Furthermore, most configuration changes translate as different values of timing constraints. In the approach proposed by our framework, algorithms are defined and proved for any timing constraint under a small set of axioms, which means that if an algorithm is proven correct for a specific configuration, it should still be correct if the configuration changes, as long as the small set of axioms still holds for the new set of timing constraints. Of course, that is not to say that modelling configuration change is not of interest, as the moment of change could compromise correctness and send the memory into a faulty state. Modelling the MRS command (responsible for entailing configuration changes) can be seen as future work.

Moreover, still on the scope of timing constraints that appear in Listing 5.1, besides clock and MRS timing, we do not model timing constraints related to DRAM data (e.g. t_{DQSQ}, t_{QH}), data strobe (e.g. t_{QSH}, t_{QSL}), Maximum Power Saving Mode (MPSM) (e.g. t_{MPED}, t_{tCKMPE}), calibration (e.g. t_{ZQCS}, t_{ZQInit}), reset/self-refresh (e.g. t_{XPR}, t_{XS}), power down (e.g. t_{XP}, t_{CKE}, t_{PD}), Per-DRAM Addressability (PDA) (e.g. t_{MRD_PDA}, t_{MOD_PDA}), On-die Termination (ODT) (e.g. t_{AONAS}), write levelling (e.g. t_{WLS}), Command Address (CA) parity (e.g. PL), and Cyclic Redundancy Check Codes (CRC) error reporting (e.g. $t_{CRC_{ALERT}}$). These timing constraints are related to DDR4 features that are not (yet) covered by our framework.

More generally, the list below exposes all of the features in the JEDEC DDR4 standard that are not covered in our framework and explains why that is. Moreover, Table 5.1 highlights which *commands* (and related timings) – of all commands defined by the DDR4 standard – are covered/modelled by CoqDRAM. In summary, *we concentrate on features that are most likely to impact the worst and average latency of memory requests when the DRAM device is operating under normal – steady-state conditions* (i.e., after the initialisation procedure has been completed).

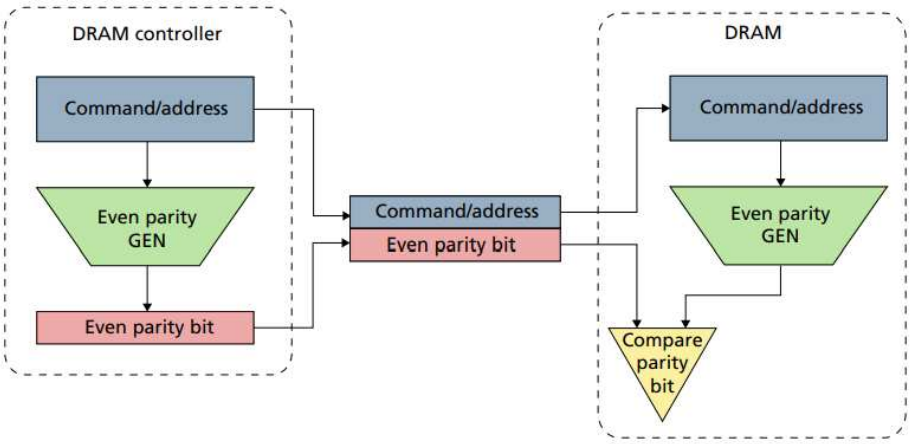


Figure 5.1: CA Parity Operation [106].

- **Data**: we do not model the actual payload of memory transactions. Although it might be interesting from some perspectives – such as ensuring data integrity and correctness of error correction codes – the payload is not essential for the functionality of the device and its correct timing behaviour. Therefore, it was not included in the current version of the framework.

- **CA Parity & CRC Error Detection**: on DDR4, Command/Address (CA) Parity is a mechanism for error detection on the command and address buses. Specifically, CA parity takes the CA parity signal (PAR) input carrying the parity bit for the generated address and command signals, and matches it to the internally (in the DRAM device) generated parity from the captured address and command signals, as shown in Figure 5.1. For error detection over the data bus, the devices dispose of a CRC Error detection mechanism, which provides real-time error detection on the DDR4 data bus, improving system reliability during write operations [106]. Both features are important for safety-critical systems, which means that modelling them can be seen as a potential improvement to the framework.

- **On-the-fly burst length**: we do not explicitly model the commands necessary to implement "on-the-fly" burst lengths, i.e., controllers modelled in the framework can only send `CRD` and `CWR` commands with fixed burst-length. The latest version of the framework does not yet capture such functionality.

- **PDA**: Per-DRAM addressability allows different DRAMs in the same module to be programmed (configured) differently. This feature is not modelled by the framework for the same reasons as MRS commands are not modelled (we are interested in encoding algorithms operating under a static configuration).

- **Reset, initialisation procedure, ZQ Calibration & Read/Write Training**: the JEDEC standards establishes a 15-step "power-up initialisation sequence" and a 3-step "reset initialisation with stable power sequence." In the framework, we consider the DRAM device to be already on steady-state, after initialisation. The reason for that is the same as with clock timing – the reset and initialisation procedures (and their related timings) are not relevant for the correctness and high-level properties of scheduling algorithms, which is what we are ultimately interested in. Moreover, ZQ Calibration is related to tuning resistors at the data pins (DQ) to the exact value of 240Ω, since the value of these might fluctuate due to voltage and temperature changes [107]. ZQ calibration is part of the initialisation routine, and as explained previously, we are only interested on the behaviour of the device after initialisation. After having performed ZQ calibration, the initialisation procedure is complete and the DRAMs are in IDLE state, but the memory is *still* not operational [107]. The controller still has to perform a few more important steps before data can be reliably written-to or read-from the DRAM. This important phase is called Read/Write Training (or Memory Training or Initial Calibration). This stage is required to adjust the latency of data transfers coming from different chips in the DIMM [107]. Again, not surprisingly, this is part of the initialisation procedure and not covered by our framework.

- **Self-refresh, MPSM & Power Down Mode**: the self-refresh command can be used to retain data in the DDR4 SDRAM, even if the rest of the system is powered down [42]. Furthermore, MPSM provides the lowest power consuming, which is similar to the self-refresh status with no internal refresh activity [42]. Again, as before, during both the self-refresh operation and the MPSM mode, the device finds itself in a power-saving state rather than the steady-state that we are interested in. The *Power Down* mode is not modelled for similar reasons.

- **Other**: other features that we do not cover on the framework are: *On-Die Termination* (ODT) mode, DLL-off mode, input clock frequency change, control gear down mode, programmable preamble and postamble. These features mostly involve configuration changes (which is out of the current scope of the framework, as explained previously). While modelling some of these features can be seen as potential future work, some of them are just not interesting in terms of verification in a high level model like the one we built with Coq, which is aimed at verifying the correctness of algorithmic scheduling decisions.

Function	Command	Supported by CoqDRAM
Mode Register Set	MRS	No
Refresh	REF	Yes
Self Refresh Entry	SRE	No
Self Refresh Exit	SRX	No
Precharge all Banks	PREA	Yes
Bank Activate	ACT	Yes
Write (Fixed BL8 or BC4)	WR	Yes
Write (BC4, on the fly)	WRS4	No
Write (BL8, on the fly)	WRS8	No
Write with Auto-Precharge (Fixed BL8 or BC4)	WRA	Yes
Write with Auto-Precharge (BL8, on the fly)	WRAS4	No
Write with Auto-Precharge (BL8, on the fly)	WRAS8	No
Read (Fixed BL8 or BC4)	RD	Yes
Read (BC4, on the fly)	RDS4	No
Read (BL8, on the fly)	RDS8	No
RDA with Auto-Precharge	RDA	Yes
Read with Auto-Precharge (BL8, on the fly)	RDAS4	No
Read with Auto-Precharge (BL8, on the fly)	RDAS8	No
No Operation	NOP	Yes
Device Deselected	DES	No
Power Down Entry	PDE	No
Power down Exit	PDX	No
ZQ Calibration Long	ZQCL	No
ZQ Calibration Short	ZQCS	No

Table 5.1: Extensive list DRAM commands from the DDR4 JEDEC standard [42]. The commands modelled in CoqDRAM are highlighted in green.

What about other memory standards?

This is a good point to discuss the relationship between CoqDRAM and other DRAM memory standards/technologies. Figure 5.2, reproduced from a recent research paper from Jung et. al [80], shows the evolution of JEDEC standards over the last two decades.

As it can be seen, the bottom-most dots in the figure are different generations of "*general purpose*" memories – DDRs. Of those, our framework covers the DDR3 and DDR4 standards. The older generations, DDR and DDR2 are obsolete and therefore not covered.

In summary, DDR5, unlike DDR4, modules have two independent command/address channels, and each control a 32-bit data bus, without *Error Correction Code* (ECC). The reduced bus width is compensated by a doubled minimum burst length of 16, which preservers the minimum access size of 64 bytes, which matches the cache line size uses by many modern processors. In summary, DDR5 devices have better bandwidth/speed (up to $6400MHz$ as opposed to DDR4's max of $3200MHz$), require less power ($1.1V$ instead of $1.2V$ required by DDR4), and have higher capacity ($128GB$ per module against DDR4's max of $32GB$ per module) [108]. Two other important types of DDR memory are *Low Power* DDR (LPDDR) and *Graphic* DDR (GDDR). LPDDRs are mainly used in mobile devices and tablets, and differ significantly from DDR is terms of performance, battery life, power consumption, and data transfer rates; and GDDRs are used in graphic cards, game consoles, and high-performance computing.

We did not thoroughly investigate to which extent CoqDRAM is effective for modelling DDR5, LPDDR, GDDR, HBM, and other technologies/standards. At first sight, it seems like *most* of the (new) features from DDR5 and LPDDR devices are translatable as *new values* for the already modelled timing constraints – captured by `System_configuration`. Not *all* features might be covered so easily, however. An example is the introduction of variable

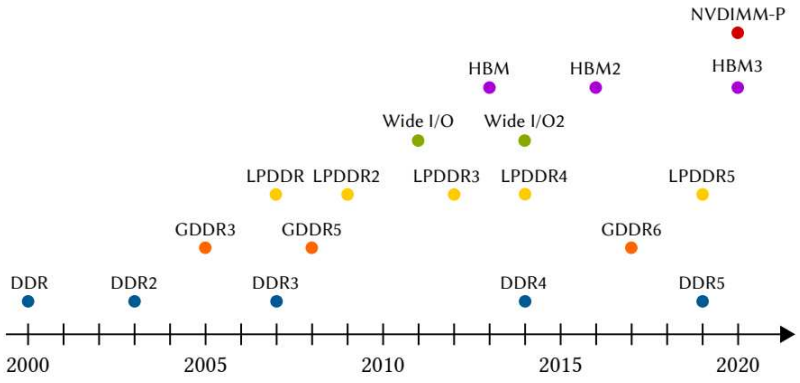


Figure 5.2: Releases of JEDEC standards [80].

command lengths in LPDDR4 [109], which was an important change, since the command length was always fixed to a single clock cycle in previous standards. This would require the introduction of at least another timing constraint, t_{CK}, the duration of a command, which would be 1 for most devices, but could admit other values for LPDDR4 devices. Another example are GDDRs, which have the ability to open two memory pages at once in a bank. Such novelty drastically changes the algorithmic of memory controllers (in the sense that a scheduling algorithm written for a DDR4 device would probably no longer work for such device) – thus requiring changes in CoqDRAM as well.

Bear in mind this is only a superficial analysis. Overall, these standards are still unexplored territory considering the scope of this work, and analysing how much adapting our framework would need to fit all of these standards can be seen as future work. Specifically, in Chapter 10, we discuss one promising future work direction that suggests using results from previous research by Jung et al. [80] to automatically generate Coq specifications from a DSL tailored to model different JEDEC standards.

Requestor

Listing 5.2 shows the definition of the type class `Requestor_configuration`.

Listing 5.2: Definition of `Requestor_configuration`.

```
Class Requestor_configuration := {
  Requestor_t : eqType;
}.
```

Quite straightforwardly, `Requestor_configuration` defines the type of requestors, i.e., processing units capable of issuing memory requests. Requestors are represented as a type class with a single element, `Requestor_t`, of type `eqType`. `eqType` in Coq is the type of types with a *decidable equality*, i.e., types for which there is a known deterministic algorithm to determine whether two terms of such a type are equal or not. It is defined in the ubiquitous *Mathematical Components*[2] (`mathcomp`) library of Coq – an extensive library of formalised mathematics. In the case of requestors, it is important that requestors are distinguishable, i.e., one must be able to compare requestors and determine if they are equal, in an algorithmic way (i.e., without depending on axioms).

One obvious choice for `Requestor_t`, is `nat`, representing a numeric ID, for instance. There exists already a decision procedure (function/algorithm) in mathcomp (also in Coq's standard library), which determines whether two members of the type `nat` are equal, and

[2] https://math-comp.github.io/

produces a boolean `true` or `false`. Then, in order to show that `nat` (or any other type with such an equality-defining decision procedure) is indeed an `eqType`, one must prove the following lemma, where `e` represents such decision procedure.

▷ `Equality.axiom e` ↔ `e : rel T` is a valid comparison decision procedure for type `T`, i.e., `reflect (x = y)(e x y)` for all `x y : T`.

Or, in natural language, relating `x` and `y` by the binary relation `e` "is the same" as the rewritable equality `x = y`. The "*is the same*" part, however, is a bit more tricky than it looks, as it relies on a concept called *reflection* – relating propositions (`Prop` in Coq) and booleans. While this discussion lies outside of the scope of this dissertation, it is important to know that in Coq, logical properties about programs are of type `Prop`. Coq, however, is a tool based on constructive mathematics instead of classical, which means that a term of type `Prop` is not trivially `True` or `False`. Or, in other words, the law of *excluded middle* does *not* apply:

▷ For every statement `P`, either `P` or `not P` holds.

Roughly speaking, contrarily to the classical point of view, in which statements are assumed to be either true or false; the constructive point of view says that we are justified in asserting that a statement is true only when we have verified that it is true, and we can only correctly assert that it is false only when we have verified that it is false [110]. Back to `Requestor_t`, `reflect` allows us to go from the procedural world of booleans back to the constructive world of Coq logical propositions.

Address

Next, after having defined the number of banks in the system (`BANKS`) and the number of bank-groups (`BANKGROUPS`), we can define how a memory address is represented. We begin with bank-groups, whose type definition is shown in Listing 5.3.

Start by noting how we use the `Section` mechanism to introduce `SYS` as an ubiquitous implicit argument (which we sometimes refer to as an "*abstract instance*"). This tells Coq that every following definition inside of the Section expects an instance of `System_configuration` as an implicit argument and thus allows us to refer to members of that class, such as `BANKGROUPS` and `BANKS`.

Next, `Bankgroup_t` is defined as a *sigma type*. In Coq, sigma types are *dependently typed*, which means that they carry "built-in" logical properties. `Bankgroup_t`, specifically, is the sub-set of natural numbers which satisfies the predicate `forall x : nat, x < BANKGROUPS`. Importantly, this is already a way to ensure correctness, i.e., we ensure that bank-groups in

Listing 5.3: Definition of `Bankgroup_t`.

```
1  Section Address.
2
3    Context {SYS : System_configuration}.
4
5    Definition Bankgroup_t := { bg : nat | bg < BANKGROUPS }.
6
7    Definition Bankgroup_to_nat (a : Bankgroup_t) : nat := proj1_sig a.
8
9    Program Definition Nat_to_bankgroup a : Bankgroup_t :=
10     match a < BANKGROUPS with
11       | true ⇒ (exist _ a _)
12       | false ⇒ (exist _ (BANKGROUPS − 1) _)
13     end.
14   Next Obligation.
15     rewrite subn1 ltn_predL ltOn; by specialize BANKGROUPS_pos.
16   Defined.
```

addresses cannot be higher than the available number of bank-groups in the system. This comes with advantages and disadvantages: the advantage is that whenever we are dealing with a bank-group in a proof, we have that property at disposal. The "disadvantage" is that whenever we need to "generate" a new bank-group from a natural number, we must provide a proof that this new number is also bounded by `BANKGROUPS`.

The next two definitions, `Bankgroup_to_nat` and `Nat_to_bankgroup` are handy functions used to convert between simple natural numbers and `Bankgroup_t`. The first part is quite straightforward, as `proj1_sig` will return the numeric part of a sigma type, ignoring the proof part. The inverse way (from a natural number to a sigma type) is more complicated. We pattern-match over the boolean comparison `a < BANKGROUPS` to test if `a` is already smaller than `BANKGROUPS`. If it is, then we are done, and we must use the constructor `exist` to build the sigma type (at Line 11). The two underscores, in Lines 11 and 12, are *holes*, which can be useful for two things: 1) It can occupy the place of an implicit argument, which Coq can automatically deduce, or 2) It can be a missing proof. The second option is only possible because we are using the Coq `Program` library, which allows this kind of construct.

Back to the case where `a < BANKGROUPS = true`, the second hole of `exist` is a missing proof, but it is trivial, so Coq can automatically fill that hole. The second case (at Line 12), where the natural number `a` that we're trying to convert to a bank-group is superior to the actual number of bank-groups in the system, requires a choice: we ignore `a` and instead build a sigma type of numeric value `BANKGROUPS − 1`. Then, the second hole actually requires a proof that `BANKGROUPS − 1 < BANKGROUPS`, which is presented in Line 15 through a

Listing 5.4: Definition of Bank_t.

```
Definition Bank_t := { a : nat | a < BANKS }.
    ...  (* conversion functions omitted *) ...
Definition Banks_t := seq.seq Bank_t.

Definition All_banks : Banks_t := map Nat_to_bank (iota 0 BANKS).
```

proof script using tactics (note that it requires using the axiom BANKGROUPS_pos, defined in System_configuration).

We define Bank_t similarly, as shown in Listing 5.4, with the conversion functions omitted. We also include the definitions of Banks_t, which is a list of elements of type Bank_t, and All_banks, which is the specific list of type Banks_t and length BANKS, i.e., a list of all banks in the system. Row_t is defined identically, and thus omitted here.

Finally, we can define the type of addresses as shown in Listing 5.5.

Listing 5.5: Definition of Address_t.

```
Record Address_t := mkAddress {
    Bankgroup : Bankgroup_t;
    Bank : Bank_t;
    Row : Row_t;
}.
```

Since we must be able to distinguish between different elements of the types Bankgroup_t, Bank_t, Row_t, and Address_t, we must also prove that they are instances of eqType, as it has been done for Requestor_t. While for Requestor_t we discussed abstractly the already-existing decision procedure to determine the equality of elements of nat, here, we must write such decision procedures ourselves.

Listing 5.6: Showing that Address_t is also an eqType.

```
(* Procedure to determine the equality of two terms of type Address_t *)
Local Definition Address_eqdef (a b : Address_t) :=
    (a.(Bankgroup) = b.(Bankgroup)) &&
    (a.(Bank) = b.(Bank)) &&
    (a.(Row) = b.(Row)).

Lemma Address_eqn : Equality.axiom Address_eqdef.
Proof. (* omitted proof *) Qed.
```

Listing 5.6 shows the definition of `Address_eqdef` and the Lemma `Address_eqn`, which is name given to `Equality.axiom` with the binary relation `Address_eqdef`. We omit the proof here for brevity, as it does not reveal any interesting insights at this point. Note how `Address_eqdef` just tests if all fields of addresses `a` and `b` are equal (assuming that the functions determining the decidable equalities of types `Bankgroup_t`, `Bank_t` and `Row_t` have already been defined at this point). Moreover, the "&&" symbol represents a boolean AND operation.

Request

We start by defining a datatype `Request_kind_t` representing the *nature* of a request (i.e., whether it is a read or write request). Again, for this inductive type, `Equality.axiom` must be proved, but that part is omitted from Listing 5.7 for conciseness. From now on, the discussion about types requiring a decision procedure for equality and the proof of `Equality.axiom` will be skipped, since the same techniques previously discussed apply.

Listing 5.7: Definition of `Request_kind_t`.

```
Section Requests.
    Context {SYS_CFG : System_configuration}.
    Context {REQ_CFG : Requestor_configuration}.
    Inductive Request_kind_t : Set := RD | WR.
```

Start by noticing the implicit arguments `SYS_CFG` and `REQ_CFG` introduced within the `Section`. For requests, not only the parameters defined within `System_configuration` need to exist, but `Requestor_t` must also exist, which means that `Requestor_configuration` is an implicit parameter of everything defined inside of `Section Requests`.

Listing 5.8: `Request_t` and `Requests_t`

```
Record Request_t := mkReq {
    Requestor : Requestor_t;
    Date : nat;
    Kind : Request_kind_t;
    Address : Address_t;
}.

Definition Requests_t := seq.seq Request_t.
```

We define `Request_t` and `Requests_t` as shown in Listing 5.8. `Request_t` is a record with four fields: `Requestor` is of type `Requestor_t` and represents the processing unit that has

issued the request; `Date` is of type `nat` and represents the arrival date the request (after having been accepted by the bus arbiter); `Kind` is of type `Request_kind_t` and represents the nature of requests (read or write); and finally, `Address` is of type `Address_t` and represents the address which the memory request reads from or writes to. Moreover, we also define a type to represent an unbounded sequence of requests, `Requests_t`.

Commands

We start by defining an inductive datatype `Command_kind_t` representing the different sorts of DRAM command presented in Chapter 2 as shown in Listing 5.9.

Listing 5.9: `Command_kind_t`

```
Inductive Command_kind_t :=
    | CRD  : Request_t → Command_kind_t
    | CRDA : Request_t → Command_kind_t
    | CWR  : Request_t → Command_kind_t
    | CWRA : Request_t → Command_kind_t
    | ACT  : Request_t → Command_kind_t
    | PRE  : Request_t → Command_kind_t
    | NOP  : Command_kind_t.
```

As previously discussed, we only model commands relevant for servicing memory requests (after the initialisation phase of the memory). Note that we also model the auto pre-charge version of `CAS` commands (`READA` and `WRITEA` from Section 2.2). Moreover, except from `NOP`, commands take a `Request_t` as an argument. The reason is that commands that do take a request as argument can only be originated from an incoming memory request, and therefore, we want to be able to track from which request a given DRAM command originated. The `NOP` command is not linked to any request, since they do nothing. Finally, we can define commands as a record, as shown in Listing 5.10.

Listing 5.10: `Command_t`

```
Record Command_t := mkCmd {
    CDate : nat;
    CKind : Command_kind_t;
}.
```

Straightforwardly, commands are defined by their kind (`CKind`, of type `Command_kind_t`), and their issue date (`CDate`, of type `nat`).

Furthermore, we define a series of helper functions to improve code readability. Here, we show a few of these function definitions, which will appear throughout the remainder of this dissertation. In Listing 5.11, we show the definitions of `isCWR`, `isCAS`, `get_bank`, and `Same_Bank`. `isCWR`, as its name suggests, will pattern match on `cmd`, an argument of type `Command_t`, and return a `bool`: if `cmd` is a `CWR` or a `CWRA`, then it returns `true`, otherwise it returns `false`. We define other functions such as `isCRD`, `isACT` and `isPRE` likewise. At Line 7, `isCAS` evaluates to true if a command passed as argument is either a `CRD`, `CRDA`, `CWR`, or `CWRA`.

Listing 5.11: Helper functions for commands.

```
1  Definition isCWR (cmd : Command_t) :=
2      match cmd.(CKind) with
3      | CWR _ | CWRA _ ⇒ true
4      | _ ⇒ false
5      end.
6
7  Definition isCAS (cmd : Command_t) := isCRD cmd || isCWR cmd.
8
9  (* Partial function to access the bank of the request that originated commands *)
10 Definition get_bank (cmd : Command_t) : option Bank_t :=
11     match cmd.(CKind) with
12     | ACT r | CRD r | CRDA r | CWR r | CWRA r | PRE r ⇒
13         Some r.(Address).(Bank)
14     | _ ⇒ None
15     end.
16
17 Definition Same_Bank a b := (get_bank a = get_bank b).
```

At Line 10, we define the partial function `get_bank`. Partial functions in Coq are (usually) implemented with the type `option`, a basic datatype in Coq. `get_bank`, specifically, takes a command as argument and *possibly* returns a `Bank_t` (i.e., it returns an `option Bank_t`). In other words, if the given command does not have an associated request, then it returns `None`, meaning that the function is undefined for such value of `cmd`. In case the command does have an associated request, we simply use the record field notation to access its bank. Similar to `get_bank`, we also define other access functions, such as `get_req`, `get_bankgroup`, and `get_row`. At Line 17, we also show how we use `get_bank` to define the function `Same_Bank`, which returns a boolean `true` if two given commands `a` and `b` target the same bank.

Command Trace

In the following discussion, rather than showing a large code snippet with the entire definition of `Trace_t`, we will break down its definition into smaller code snippets – thus keeping the textual explanation close to the code snippets. But bear in mind that these are all part of the same record definition – `Trace_t`.

Listing 5.12: Definition of `Trace_t`.

```
1  Record Trace_t := mkTrace {
2   Commands : Commands_t;
3   Time     : nat;
4
5   (* All commands must be uniq, i.e., no duplicate commands *)
6   Cmds_uniq   : uniq Commands;
7   (* All commands have to occur before the current time instant *)
8   Cmds_time_ok : forall cmd, cmd \in Commands → cmd.(CDate) <= Time;
9
10  ...  (* POs about timing and functional correctness -- discussed below *)
11 }
```

Start by noticing in Listing 5.12 that `Trace_t` is a record with only two non-propositional (i.e., not of type `Prop`) members: `Commands`, of type `Commands_t`, and `Time`, of type `nat`. `Commands` is a list modelling all commands that have been issued to the memory device up until (and including) clock tick `Time`. Or, differently put, it is a list of `Command_t` of length `Time`. Then, all of the following elements of the trace are *proof obligations* (POs), which together constitute one of the main pieces of this framework. These POs establish correctness according to the JEDEC standards, both in terms of functional and timing correctness. We go through some of these POs in the following paragraphs (we do not discuss every single PO in detail because, as it will soon become evident, many of these are stated in a very similar form).

The first proof obligations, `Cmds_uniq` and `Cmds_time_ok`, are exceptionally not related to the JEDEC standards – they are instead used to ensure that the trace is coherent (i.e. feasible in real life). `Cmds_uniq` uses the `mathcomp` function `uniq`, which states that all elements in a sequence (list) are "*pairwise different*", i.e., there are no duplicate commands. This is an important property, as having identical commands (with the same kind and issue date) in the trace would be a clear violation (the command bus can only transfer one command at a given clock tick). Next, `Cmds_time_ok` ensure that, for any given command in the trace, the issue date of such command must be less or equal to `Time`, the trace length. Note that

`Cmds_time_ok` use the operator `\in`, defined in `mathcomp` to test list membership. Moreover, if the trace is empty, than the `\in` clauses evaluate to `false`, which would result in POs containing expressions of the form `false → _`, which are trivially true.

Proof obligations ensuring timing correctness in `Trace_t`

Listing 5.13: POs ensuring timing correctness in `Trace_t`.

```
(* ------------------- Intra-bank constraints -------------------------- *)
(* Ensure that the time between an ACT and a CAS commands respects T_RCD *)
Cmds_T_RCD_ok : forall a b, a \in Commands → b \in Commands →
   isACT a → isCAS b → (Same_Bank a b) →
   Before a b → Apart_at_least a b T_RCD;

(* Ensure that the time between a PRE and an ACT commands respects T_RP *)
Cmds_T_RP_ok : forall a b, a \in Commands → b \in Commands →
   isPRE a → isACT b → (Same_Bank a b) →
   Before a b → Apart_at_least a b T_RP;

(* Ensure that the time between two ACT commands respects T_RC *)
Cmds_T_RC_ok : forall a b, a \in Commands → b \in Commands →
   isACT a → isACT b → (Same_Bank a b) →
   Before a b → Apart_at_least a b T_RC;

(* Ensure that the time between an ACT and a PRE commands respects T_RAS *)
Cmds_T_RAS_ok : forall a b, a \in Commands → b \in Commands → isACT a →
   isPRE b → (Same_Bank a b) → Before a b → Apart_at_least a b T_RAS;

(* Ensure that the time between a CRD and a PRE commands respects T_RTP *)
Cmds_T_RTP_ok : forall a b, a \in Commands → b \in Commands →
   isCRD a → isPRE b → (Same_Bank a b) →
   Before a b → Apart_at_least a b T_RTP;
```

As it can be seen in Listing 5.13, the proof obligation `Cmds_T_RCD_ok` is the first of a series of POs dedicated to ensuring the correct timing behaviour of the DRAM device – it ensures that the t_{RCD} constraint is always respected in the trace (the reader can review the definition of t_{RCD} in Table 2.2 and Figure 2.7). The PO can be read as: for any two commands `a` and `b`, given that `a` is a member of `Commands`, `b` is also a member of `Commands`, `a` is an `ACT` command, `b` is a `CAS` command, `a` and `b` target the same bank, and `a` has been issued before

b, then it must follow that the issue date of b is greater then or equal to the issue date of a plus t_{RCD}.

Besides the helper functions isACT, isCAS, and Same_Bank, note that we also use Before and Apart_at_least: Before a b returns true if a.(CDate) < b.(CDate) and Apart_at_least a b x returns true if a.(CDate) + x ≤ b.(CDate). We use these functions throughout the other proof obligations in the trace. Proof obligations Cmds_T_RP_ok, Cmds_T_RC_ok, Cmds_T_RAS_ok, and Cmds_T_RTP are stated in a nearly identical format.

Proof obligation Cmds_T_WTP_ok, shown in Listing 5.14 is slightly different: it does not explicitly model a constraint from the JEDEC standards (if one looks for it, the constraint t_{WTP} is not included in the standards. However, even if the name t_{WTP} does not exist in the standards, there exists a timing constraint between CWR commands and PRE commands to the same bank. That timing constraint is the sum of t_{WL} (latency between the CWR command issue date and the start of the data bus transfer), t_{BURST} (duration of the data bus transfer), and t_{WR} ("*write recovery*" time – a constraint between the end of a write transaction on the data bus and pre-charging the respective bank). For a visual understanding, see Figure 2.7. Interestingly, the JEDEC standards define explicitly a t_{RTP} constraint, which constraints the delay between CRD and PRE to the same bank.

Listing 5.14: POs ensuring timing correctness in Trace_t (continuation).

```
1 (* Ensure that the time between a CWR and a PRE commands
2  respects T_WR + T_WL + T_BURST *)
3 Cmds_T_WTP_ok : forall a b, a \in Commands → b \in Commands →
4    isCWR a → isPRE b → (Same_Bank a b) →
5    Before a b → Apart_at_least a b (T_WL + T_BURST + T_WR);
```

Two other interesting POs are Cmds_T_WtoR_SBG_ok and Cmds_T_WtoR_DBG_ok, shown in Listing 5.15. Similar to t_{WTP} and t_{RTP}, the JEDEC standards define a constraint to account for the minimum delay between a CRD followed by a CWR, namely, t_{RTW}. However, the minimum delay between a CWR followed by a CRD (to any bank) does not have an explicit name and is given by $t_{WL}+t_{BURST}+t_{WTR}$. Note that t_{WL} and t_{BURST} are the same parameters used in Cmds_T_WTP_ok, but t_{WTR} is yet another timing parameter defined in System_configuration representing the minimum delay between the end of a write transaction on the data bus and a CRD. Usually, as it can be seen in Table 2.2, the sum $t_{WL} + t_{BURST} + t_{WTR}$ results in a value greater than t_{RTW}, which means that write-to-reads have higher latency than read-to-writes.

Moreover, the write-to-read PO comes in two flavours: one for commands targeting the same bank-group (Cmds_T_WtoR_SBG_ok), and another for commands targeting different bank-groups (Cmds_T_WtoR_DBG_ok). Note that the only difference between the two is the

Listing 5.15: POs ensuring timing correctness in `Trace_t` (continuation).

```
(* ------------------ Intra and Inter-bank constraints ---------------------- *)
Cmds_T_WtoR_SBG_ok : forall a b, a \in Commands → b \in Commands →
   isCWR a → isCRD b → Before a b → (Same_Bankgroup a b) →
   Apart_at_least a b (T_WL + T_BURST + T_WTR_l);

Cmds_T_WtoR_DBG_ok : forall a b, a \in Commands → b \in Commands →
   isCWR a → isCRD b → Before a b → not (Same_Bankgroup a b) →
   Apart_at_least a b (T_WL + T_BURST + T_WTR_s);

Cmds_T_RtoW_ok : forall a b, a \in Commands → b \in Commands →
   isCRD a → isCWR b → Before a b → Apart_at_least a b T_RTW;

Cmds_T_CCD_SBG_ok : forall a b, a \in Commands → b \in Commands →
   (isCRD a ∧ isCRD b) ∨ (isCWR a ∧ isCWR b) → Before a b →
   (Same_Bankgroup a b) → Apart_at_least a b T_CCD_l;

Cmds_T_CCD_DBG_ok : forall a b, a \in Commands → b \in Commands →
   (isCRD a ∧ isCRD b) ∨ (isCWR a ∧ isCWR b)→  Before a b →
   not (Same_Bankgroup a b) → Apart_at_least a b T_CCD_s;
```

use of t_{WTR_l} and t_{WTR_s}, where $t_{WTR_l} > t_{WTR_s}$. In summary, if the commands target the same bank-group, then they must wait for longer than they would have to if they targeted the same bank-group. Note that this model allows us to cover both DDR3 and DDR4 devices. For DDR3, specifically, since there is only a single bank-group, the hypothesis `not (Same_Bankgroup a b)` is always false. A property containing a false hypothesis in Coq is trivially true (by the $\bot\ I$ inference rule of the natural deduction implemented in Coq). Therefore, a controller written for a DDR3 device specifically would have to present a proof only for `Cmds_T_WtoR_SBG_ok`, and t_{WTR_l} would need only to be instantiated with the correct value of t_{WTR} defined in the JEDED DDR3 standards [41]. The exact same discussion about bank-groups applies for the t_{CCD} constraint and the two versions of the PO modelling it.

Finally, POs `Cmds_T_FAW_ok`, `Cmds_T_RRD_SBG_ok`, and `Cmds_T_RRD_DBG_ok`, shown in Listing 5.16, cover the exclusively inter-bank set of timing constraints (see Table 2.2). In `Cmds_T_FAW_ok`, `Diff_bank` [::a;b;c;d] returns true if commands `a`, `b`, `c`, and `d` all target different banks. Furthermore, like for t_{CCD} and t_{WTR}, timing constraint t_{RRD} also admits different values depending on whether commands `a` and `b` target the same or different bank-groups, respectively t_{RRD_l} and t_{RRD_s}.

Listing 5.16: POs ensuring timing correctness in `Trace_t` (continuation).

```
(* ------------------ Exclusively inter-bank constraints ------------------ *)
Cmds_T_FAW_ok : forall a b c d, a \in Commands → b \in Commands →
   c \in Commands → d \in Commands →
   isACT a → isACT b → isACT c → isACT d → Diff_Bank [::a;b;c;d] →
   Before a b → Before b c → Before c d → Apart_at_least a d T_FAW;

Cmds_T_RRD_SBG_ok : forall a b, a \in Commands → b \in Commands →
   isACT a → isACT b → not (Same_Bank a b) →
   (Same_Bankgroup a b) →
   Before a b → Apart_at_least a b T_RRD_l;

Cmds_T_RRD_DBG_ok : forall a b, a \in Commands → b \in Commands →
   isACT a → isACT b → not (Same_Bankgroup a b) →
   Before a b → Apart_at_least a b T_RRD_s;
```

Proof obligations ensuring functional correctness in `Trace_t`

As shown in Listing 5.17, there are four POs dedicated to asserting functional correctness: `Cmds_ACT_ok`, `Cmds_row_ok`, `Cmds_initial`. First, 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.

`Cmds_ACT_ok` is further depicted in Figure 5.3, where the PO is met by the conjunction of scenarios S_1 and S_2. In the figure (and in Figure 5.4), 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.

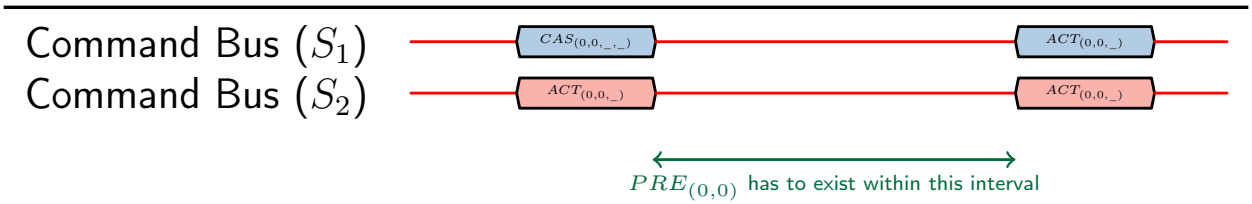


Figure 5.3: Illustration of the `Cmds_ACT_ok` PO.

Moreover, 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

Listing 5.17: POs ensuring functional correctness in `Trace_t`

```
1  Cmds_ACT_ok: forall a b, a \in Commands → b \in Commands →
2     isACT a ∨ isCAS a → isACT b → (Same_Bank a b) → Before a b →
3     exists c, (c \in Commands) && (isPRE c) && Same_Bank b c
4     && Same_Bank a c && (After c a) && (Before c b);
5
6  (* Every CAS command is preceded by a matching ACT
7     without another ACT or PRE in between *)
8  Cmds_row_ok : forall b c, b \in Commands → c \in Commands →
9     isCAS b → isPRE c → Same_Bank b c → Before c b →
10    exists a, (a \in Commands) && (isACT a) && (get_row a = get_row b)
11    && (After a c) && (Before a b);
12
13 (* Implies the initial state of the memory *)
14 Cmds_initial : forall b, b \in Commands → isCAS b →
15    exists a, (a \in Commands) && (isACT a) && (get_row a = get_row b) && (Before a b);
```

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 Figure 5.4.

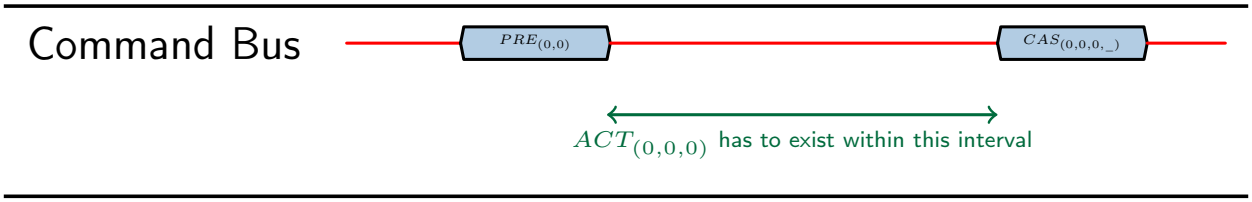


Figure 5.4: Illustration of the `Cmds_row_ok` PO.

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 POs guarantee that implemented controllers generate valid commands and respect the protocol described in the DDR3 and DDR4 JEDEC standard. The conditions established by the POs can also be visualised in Table 5.2. We use the notation → to denote an immediate sequence between commands, e.g, $\texttt{PRE}_{(bg,bg)} \rightarrow CAS_{(bg,bk,r,c)}$ 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 $\texttt{PRE}_{(bg,bk)} \rightarrow \texttt{CAS}_{(bg,bk,r_0,\_)}$ 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 ($\texttt{ACT}_{(bg,bk,r_0)}$ or $\texttt{ACT}_{(bg,bk,r_1)}$), 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 ($\texttt{ACT}_{(bg,bk,r_1)} \rightarrow \texttt{CAS}_{(bg,bk,r_0,\_)}$) 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 $\texttt{CAS}_{(bg,bk,r_0,c)}$; then, `Cmds_ACT_ok` ensures that a $\texttt{PRE}_{(bg,bk)}$ is between the two `ACT`s, and finally, `Cmds_row_ok` ensures that $\texttt{ACT}_{(bg,bk,r_0)}$ will be between the $\texttt{PRE}_{(bg,bk)}$ and the $\texttt{CAS}_{(bg,bk,r_0,\_)}$.

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

Table 5.2: Protocol correctness for a given bank group and a given bank.

Arrival Model

Listing 5.18: Definition of `Arrival_function_t`.

```
1 Class Arrival_function_t := mkArrivalFunction
2 {
3   Arrival_at : nat → Requests_t;
4
5   Arrival_date : forall ta x, (x \in (Arrival_at ta)) → x.(Date) = ta;
6
7   Arrival_uniq : forall t, uniq (Arrival_at t);
8 }.
```

Before finally discussing the `Arbiter_t` class, we must define the arrival model for memory requests. Listing 5.18 shows the definition of the `Arrival_function_t` class. Most importantly, `Arrival_function_t` defines the function `Arrival_at`, which takes a parameter of type `nat` and returns the set of requests that arrive into the system at that instant. The class

members `Arrival_date` and `Arrival_uniq` are POs that guarantee that `Arrival_function_t` is a realistic model: `Arrival_date` states that if a request `x` is in `Arrival_at ta`, then the arrival date of `x` is `ta`. Next, `Arrival_uniq` states that, for a given instant `t`, arriving requests are unique, i.e., no duplicate requests can arrive at a single instant `t`.

This is a rather relaxed model, in the sense that it allows an unbounded number of requests to arrive at any instant. As we describe in Appendix A, when we attempt to prove equivalence between a `CoqDRAM` controller and a hardware model, this arrival function needs to be made more strict, since such abstraction is obviously not possible on real silicon.

Arbiter

Listing 5.19: Definition fo `Arbiter_t`.

```
1  Class Arbiter_t {AF : Arrival_function_t} := mkArbiter {
2      Arbitrate : nat → Trace_t;
3
4      (* Time has to match *)
5      Time_match : forall t, (Arbitrate t).(Time) = t
6
7      (* All requests must handled *)
8      Requests_handled : forall ta req, req \in (Arrival_at ta) →
9          exists tc, (CAS_of_req req tc) \in ((Arbitrate tc).(Commands));
10 }.
```

Finally, we can define the class `Arbiter_t` as shown in Listing 5.19. Quite straightforwardly, the main element in `Arbiter_t` is the `Arbitrate` function, builds a `Trace_t` of length given by a `nat` parameter. Or, in other words, an arbiter is an entity that issues commands to the memory devices, i.e., creates a command trace. The fact that the trace should be of length given by the first parameter of `Arbitrate` is expressed by the PO `Time_match`, which states that, for an arbitrary `t`, the `Time` field of the trace obtained by `Arbitrate t` must be equal to `t`. Recall that, from `Cmds_time_ok` (see Listing 5.12), every command in the trace must have its date less or equal to `Time`.

`Requests_handled` is an important PO: it states that, for any given instant `ta` and for any given request `req`, if `req` has arrived at `ta`, then there must exist an instant `tc` when a `CAS` command belonging to `req` is present in the trace. Or, in other words, every request that has arrived is eventually completed. This PO ensures non-starvation, or *fairness*. The current version of the framework requires this to be true for all concrete arbiter instances, although

it is future work to relax this PO, making it dependent on an user-defined predicate.

Sequential Consistency

In fact, the idea of having different types of arbiters, with different properties, is a key aspect of the framework. To showcase this, we model two flavours of Lamport's definition of *Sequential Consistency* (SC) [111] – a guarantee on the possible order in which requests are handled. 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 [112].

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 5.20. The code in the listing involves two universally quantified requests, `reqa` and `reqb`, which have respectively arrived at `ta` and `tb`. We assume that `reqa` has an earlier arrival order than `reqb`. Such arrival order is expressed by a disjunction: *either* `ta` is strictly less then `tb` *or* they are equal but the position of `reqa` in the queue is strictly less then `reqb`'s position. In `R2_relax`, note the pre-condition `reqa.(Row)` = `reqb.(Row)`, modelling Lamport's condition of "targeting the same memory address."

Listing 5.20: Modeling Sequentially Consistent Arbiters.

```
Class SequentialConsistent_Arbiter {AF : Arrival_function_t} {AR : Arbiter_t} :=
    mkSeqArbiter {

    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_Arbiter {AF : Arrival_function_t} {AR : Arbiter_t} :=
    mkWSeqARbiter {

    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)
}.
```

While a concrete implementation *must* implement the `Arbiter_t` specification, it *may* also (optionally) implement the `SequentialConsistent_Arbiter` and `W_SequentialConsistent_Ar−biter` specifications. Again, the idea of having a framework with different arbiter specifications, with different properties, is a key aspect of the framework. In the future, we aim to create a wider variety of possible specifications (including arbiters with relaxed fairness POs for mixed-criticality systems, different shared memory models, security properties, et cetera).

Implementation Interface

Last, we describe `Implementation_t`, which defines a transition system that serves as a skeleton for writing scheduling algorithms. As it can be seen in Listing 5.21, `Implementation_t` is class defining two functions: `Init` and `Next`. The former defines the initial state of the transition system, and the latter is the transition function implementing the scheduling algorithm. They both operate on a datatype `State_t`, which is an arbitrary `Type`, defined in the `Arbiter_configuration` class. The idea is that states of the transition system should be customisable, due to the fact that different algorithms might need different state variables to implement algorithms, such as different counters, queues, et cetera. In more detail, as their types suggest, `Init` takes as argument the initial set of outstanding requests and produces a state; and `Next` takes as arguments a set of arriving requests, a state, and produces a new state and a `Command_kind_t` – the command to be issued to the memory device.

Listing 5.21: Definition of `Implementation_t`.

```
1 Class Arbiter_configuration := {
2     State_t : Type;
3 }.
4
5 Class Implementation_t := mkImplementation {
6     Init : Requests_t → State_t;
7     Next : Requests_t → State_t → State_t * Command_kind_t;
8 }.
```

Next, we define `Arbiter_state_t` – a "second layer" on the top of `State_t` – as shown in Listing 5.22. The goal of `Arbiter_state_t` is to represent/hold intermediate states of the trace. It includes the internal state of the algorithm, (`Implementation_State`), a list of commands (`Arbiter_Commands`), and a time stamp (`Arbiter_Time`). The relation between these two elements is described by the proof obligation `Arbiter_Commands_date`, which states

Listing 5.22: Definition of `Default_arbitrate`.

```
1  Record Arbiter_state_t := mkArbiterState {
2      Arbiter_Commands     : Commands_t;
3      Arbiter_Time         : nat;
4      Implementation_State : State_t;
5
6      Arbiter_Commands_date : forall c,
7          c \in Arbiter_Commands → c.(CDate) <= Arbiter_Time
8  }.
9
10 (* Computes the t_th state *)
11 Fixpoint Default_arbitrate {AF : Arrival_function_t} {IM : Implementation_t}
12     (t : nat) : Arbiter_state_t :=
13     let R := Arrival_at t in
14     match t with
15     | 0     ⇒ mkArbiterState [::] t (Init R)
16     | S(t') ⇒ let old_state := Default_arbitrate t' in
17               let (new_state,new_cmd_kind) := Next R old_state.(Implementation_State) in
18               let new_cmd := mkCmd t new_cmd_kind in
19               let cmd_list := (new_cmd ::old_state.(Arbiter_Commands)) in
20               mkArbiterState cmd_list t new_state
21     end.
```

that `Arbiter_Time` is the upper bound for the issue date of every command member of `Arbiter_Commands`. From now on, since there are two "*state*" abstractions, we will refer to `State_t` – the abstraction used to hold algorithmic states manipulated by `Init` and `Next` – as *internal states*, and `Arbiter_state_t` will be merely referred to as *states* (or sometimes *arbiter states*).

In order to build a command trace, we define the function `Default_arbitrate`. The function takes a natural number `t` as parameter, which represents the trace length. At Line 13, `R` is set to be the set of arriving requests at `t`. Moreover, the function assumes the existence of an arrival function (`AF`) and a scheduling algorithm (`IM`). The function works by recursively building new states (of type `Arbiter_state_t`). As it will be seen later, the trace of length `t` will then be constructed by a call to `Default_arbirate t`.

Inside the function, we pattern match on `t`. If `t` is zero, it means that we are building the first state, which is accomplished by the term `mkArbiterState [::] t (Init R)` (Line 15). We call `mkArbiterState` to build the initial state with an empty command list `[::]`, time stamp 0 (`t`), and internal state given by the `Init` function, defined in `IM` (of type `Implementation_t`). We pass `R` – the set of pending requests at initialisation – as a parameter to `Init`.

If `t` is not zero (i.e., if it is a successor of some number `t'`), then we call `Default_arbitrate`

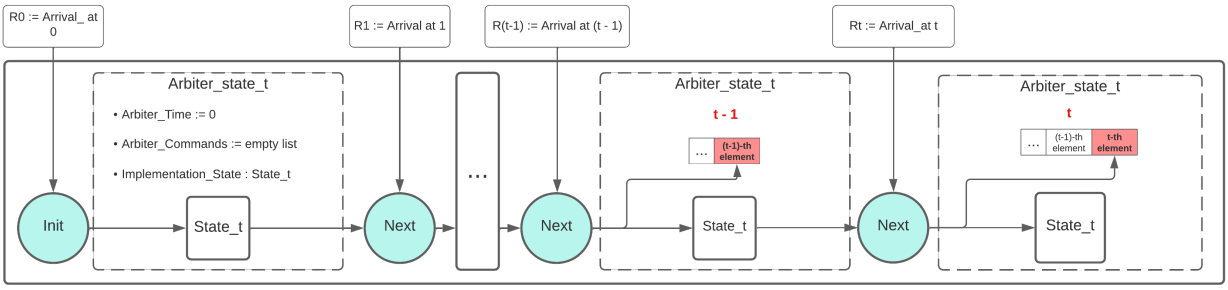


Figure 5.5: State generation through `Default_arbitrate`.

on `t`' to build the previous state (Line 16), which is given the name `old_state`. Then, we call the transition function `Next` to build a new internal state, passing as arguments `R` (the set of arriving requests at `t`), and the old state's internal state. Then, a new command is built with the command constructor `mkCmd`, with issue date `t` and kind given by the output of the `Next` function. Finally, on Line 19, the new command is appended to the previous state's command list (`new_cmd` ::`old_state`.(`Arbiter_Commands`)) and the current state is built using the constructor `mkArbiterState`, with arguments `cmd_list` (the command list appended with the newly generated command), `t` (the current time stamp), and `new_state` (the current scheduling internal state). Figure 5.5 depicts the behaviour of `Default_arbitrate`.

The goal of `Implementation_Interface` is to provide an interface for implementing arbiters. 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. The next chapter is aimed at showing how this interface can be used to encode and prove real algorithms.

Chapter 6

Writing and Proving Scheduling Algorithms

In this chapter, we present one of two approaches for writing scheduling algorithms. It consists in directly implementing the `Implementation_t` interface presented in the previous chapter. Following that methodology, algorithms have to schedule memory requests, generate commands, and issue commands to the device in a way such that timing constraints are respected.

Conversely, the (newer) second approach, described in Chapters 8 and 9, consists in using a set of functions to keep track of the state of the memory. Hence, scheduling algorithms are written in a way that *only valid commands are allowed to be sent in the first place*. The second approach has the benefit of being implementation-agnostic, i.e., any algorithm written on the top of such abstraction inherits the correctness proofs about timing constraints.

Following the first methodology, algorithms are written as instances of `Implementation_t`, i.e., algorithms have to provide definitions for the functions `Init` and `Next` (c.f. Listing 5.21).

6.1 *First-In-First-Out* (FIFO)

The FIFO algorithms works as follows: when a request arrives in the system, it immediately enters a (unbounded) waiting queue. If the scheduler is idle (i.e., no request is currently being processed), the scheduler starts processing the request sitting at the head of the queue. Processing a request means issuing a sequence `PRE-ACT-CAS`, conforming to a closed-page policy (see Chapter 2). Once one request finishes processing, the scheduler checks if there is another request in the queue: if yes, then processing of the next request starts immediately, if not, the scheduler goes back to idle and stays idle until another request arrives in the system.

To decide when to issue commands, the arbiter keeps a *counter*: when the processing of a request starts, a `PRE` command is issued and the counter is set to 0. Then, when the counter reaches the value $t_{RP} - 1$, the scheduler issues an `ACT`. Next, when the counter reaches the value $t_{RP} - 1 + t_{RCD}$, the scheduler issues the appropriate `CAS` command (a `CRD` if the request is a read transaction or a `CWR` if the request is a write transaction). Finally, another request cannot start being processed immediately after the `CAS` from the previous request has been issued, since that could violate some timing constraints. The length of the processing slot/window is defined by the scheduler parameter `WAIT`, as shown in Figure 6.1.

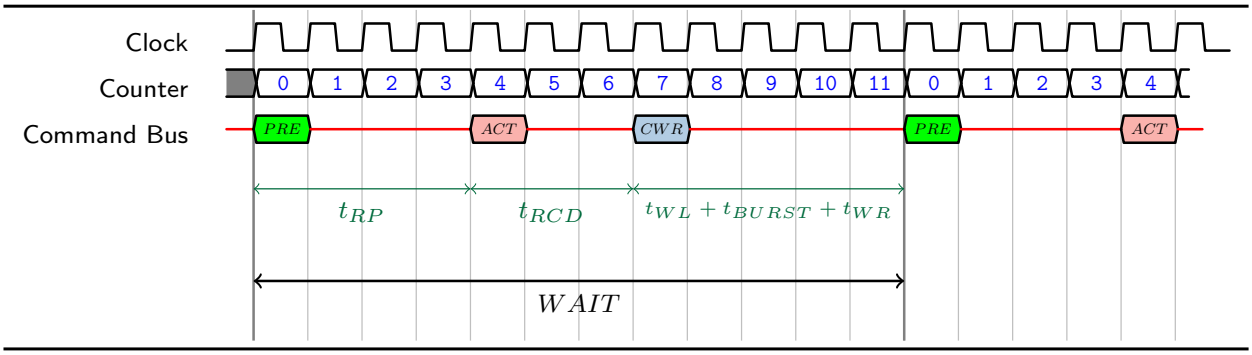


Figure 6.1: FIFO command scheduling during request processing.

For the scheduler to be correct, `WAIT` has to be chosen in a way such that a list of axioms are respected. For *existing* DDR3 and DDR4 devices, the most constraining axiom is $t_{RP} + t_{RCD} + t_{WL} + t_{BURST} + t_{WR} \leq$ `WAIT`. That statement comes from the following observation: in a FIFO scheduling scheme, in the worst-case, two requests being processed one after the other could always target the same bank. For that reason, we must assume that a `PRE` cannot occur in the clock cycle directly after the `CAS` (which would have been possible if the requests targeted different banks – see Table 2.2). Moreover, the latency between a `CWR` command to a `PRE` is generally larger than the latency between a `CRD` and a `PRE`, hence why the worst-case is calculated using the `CWR` command, as seen in Figure 6.1.

The full set of axioms constraining `WAIT` are defined in the class `FIFO_configuration`, shown in Listing 6.1. The condition described above is stated as `WAIT_END_WRITE` in the code. Overall, because `WAIT_END_WRITE` is the most constraining inequality on all existing DDR3 and DDR4 devices, `WAIT` is typically chosen in a way such that `WAIT_END_WRITE` is minimally respected, i.e. `WAIT` $= t_{RP} + t_{RCD} + t_{WL} + t_{BURST} + t_{WR}$.

Moreover, these axioms "emerge" from the proof obligation stated in `Trace_t`, i.e., they are needed at some point during a proof and cannot be derived (in general) from other axioms. Beyond that, `FIFO_configuration` is a complete list of all axioms needed for the proofs to go through. This is much more compact way to formalise and group all assumptions that an

algorithm relies upon, compared to the paper-and-pencil approach, where assumptions and axioms are typically scattered throughout papers and sometimes stated only informally. Second, `FIFO_configuration` works as a checker: Coq will only consider an instance of such class to be valid when all proof obligations have been discharged (these simple proof obligations are almost always discharged automatically when the inputed values are valid).

Listing 6.1: `FIFO_configuration` – axioms about `WAIT`.

```
1  Definition ACT_date := T_RP.−1.
2  Definition CAS_date := ACT_date + T_RCD.
3
4  Class FIFO_configuration :={
5      WAIT : nat;
6      WAIT_pos : WAIT > 0;
7
8      (* Length of the minimum slot (write): T_RP + T_RCD + T_WL + T_BURST + T_WR *)
9      WAIT_END_WRITE : CAS_date + T_WR + T_WL + T_BURST < WAIT;
10
11     (* Length of the minimum slot (read): T_RP + T_RCD + T_RTP *)
12     WAIT_END_READ : CAS_date + T_RTP < WAIT;
13
14     (* Other axioms *)
15     RC_WAIT       : T_RC < WAIT;
16     CCD_WAIT      : T_CCD_l < WAIT;
17     RRD_WAIT      : T_RRD_l < WAIT;
18     RTW_WAIT      : T_RTW < WAIT;
19     RAS_WAIT      : T_RP + T_RAS < WAIT;
20     WTP_WAIT      : T_WR + T_WL + T_BURST < WAIT;
21     WTR_WAIT      : T_WTR_l + T_WL + T_BURST < WAIT;
22     FAW_WAIT      : T_FAW < WAIT + WAIT + WAIT;
23 }.
```

To implement the algorithm, as mentioned previously, the functions `Init` and `Next`, from `Implementation_t`, must be implemented. However, a requestor type and an internal state must be defined beforehand. A requestor type is the implementation-dependent information about the processing units issuing memory requests. For FIFO specifically, no information about requestors is used, therefore, we instantiate the requestor type as `unit_eqType`, an inductive Coq type with single inhabitant (`tt`) – a way of stating that all elements of this type are the same. The internal state must hold the necessary variables to implement the

algorithm. In the case of FIFO, we need the notion of "idle" and "running" states, a counter, a list of pending requests, and in the case the arbiter finds itself in the running state, an identifier for the request currently under processing. The requestor type and the internal state definitions are shown in Listing 6.2.

Listing 6.2: FIFO preliminearies – defining a requestor type and internal state.

```
1  Instance REQUESTOR_CFG : Requestor_configuration := {
2      Requestor_t := unit_eqType
3  }.
4
5  Definition Counter_t := ordinal WAIT.
6
7  Inductive FIFO_state_t :=
8      | IDLE    : Counter_t → Requests_t → FIFO_state_t
9      | RUNNING : Counter_t → Requests_t → Request_t → FIFO_state_t.
10
11 Instance ARBITER_CFG : Arbiter_configuration :={
12     State_t := FIFO_state_t;
13 }.
```

We define the counter to be of type `Counter_t`, which is defined at Line 5. `ordinal` is yet another dependent type defined in `mathcomp` (somehow similar to sigma types). The construct `ordinal n` (or `'I_n` in Coq notation) represents the finite sub set of integers `i` $<$ `n`, whose enumeration is $\{0, ..., n-1\}$. As for sigma types, an ordinal carries a numeric value and a proof that such value is smaller than its bound `n`. Here, since we want the counter to wrap around the value `WAIT` -1, defining it as a `ordinal WAIT` ensures that the counter can never assume a value greater than `WAIT`.

At Line 7, we define `FIFO_state_t`, the internal state manipulated by the scheduling algorithm. The arbiter can find itself in two states: `IDLE` and `RUNNING`. An `IDLE` state is "associated" to a counter and a list of pending requests. A `RUNNING` state, besides the counter and the list of pending requests, is also associated with a `Request_t`, representing the request under processing. Note that `IDLE` and `RUNNING` can also be seen as constructors, i.e., they can be used as functions to build a new `FIFO_state_t`. In that case, the terms appearing after ":" are the type of the arguments expected by the constructors. In addition, at Line 11, we indicate to Coq that `ARBITER_CFG` is an instance of `Arbiter_configuration` (see Listing 5.21). This ensures that we can pass terms of type `FIFO_state_t` where Coq expects a `State_t` (i.e., it creates a coercion between types).

Now, having defined `REQUESTOR_CFG` and `FIFO_state_t`, the algorithm can be implemented

by defining the body of the functions `Init` and `Next`. The definition of `Init_state` is shown in Listing 6.3. The function takes `R`, of type `Requests_t` as argument and returns a scheduler state, of type `FIFO_state_t`. The function creates an `IDLE` state, with a counter initialised to 0 (`OCycle0`) and a pending request queue made of the arriving requests at initialisation (`R`).

Listing 6.3: FIFO initial state.

```
Definition OCycle0 := Ordinal WAIT_pos.
Definition Init_state R := IDLE OCycle0 R.
```

Listing 6.4: FIFO next state.

```
1  Definition Next_state R (AS : FIFO_state_t) : (FIFO_state_t * Command_kind_t) :=
2  match AS with
3    | IDLE c P ⇒ (* current state is IDLE *)
4      let c' := Next_cycle c in (* calculates next value of the counter *)
5      let P' := Enqueue R P in (* enqueues arriving requests into pending queue *)
6      match P with
7        | [::]     ⇒ (IDLE c' P',NOP) (* build new IDLE state *)
8        | r :: PP ⇒ (RUNNING OCycle0 (Enqueue R (Dequeue r P)) r, PRE r)
9      end
10   | RUNNING c P r ⇒
11     let P' := Enqueue R P in
12     let c' := Next_cycle c in
13     if      c = OACT_date then (RUNNING c' P' r, ACT r) (* issue ACT *)
14     else if c = OCAS_date then (RUNNING c' P' r, (Kind_of_req r) r) (* issue CAS *)
15     else if c = WAIT.-1 then (* transition to next request or go back to IDLE *)
16       match P with
17         | [::] ⇒ (IDLE OCycle0 P', NOP)
18         | r :: PP ⇒ (RUNNING OCycle0 (Enqueue R (Dequeue r P)) r, PRE r)
19       end
20     else (RUNNING c' P' r,NOP)
21 end.
```

In the listing, `OCycle0` is defined as `Ordinal WAIT_pos`: the element of `'I_n` with (nat) value i, given a proof of `i < n`. In this case, `WAIT_pos` is a proof of $0 <$ `WAIT`, which means that `Ordinal WAIT_pos` is a pair with `nat` value 0 and a proof that $0 <$ `WAIT_pos` (one of the axioms in Listing 6.1).

The core of the algorithm is defined by `Next_state`, shown in Listing 6.4. The function,

as stated by its specification `Next`, takes a list of arriving requests `R`, a state `AS`, and produces a pair made of a new state and a command – the latter being issued to the memory device.

The algorithm begins by testing if the current state `AS` is `IDLE` or `RUNNING`. If it is `IDLE`, then we must look at the queue of pending requests, `P`. If the queue is empty (Line 7), then we build a new state `IDLE c' P'` and issue a NOP command, with `c'` standing for `Next_cycle c`, and `P'` standing for `Enqueue R P`. The definitions of `Next_cycle` and `Enqueue` are omitted for conciseness, but their behaviour is simple: the former increments the counter (wrapping around `WAIT` − 1) and the latter appends `R` (the arriving requests) to the tail of `P` (the currently pending requests). If, however, `P` is not empty (i.e., it has a head `r` and a tail `PP`), then the processing of its head, `r`, must begin (Line 8). The arbiter proceeds to build a state `RUNNING OCycle0 (Enqueue R (Dequeue r P))r` and to issue a `PRE r` command. This means that the algorithm goes into the `RUNNING` state, resets the counter to 0, removes `r` from the pending request queue (`Dequeue r P`), and enqueues `R` in the resulting queue. In addition, `r` is also set to be the request currently under processing. Note also that `r` is associated with the `PRE` command.

If `AS` is `RUNNING` (Line 10), then the defining factor for determining what happens next is the value of the counter, `c`. If `c` is equal to `OACT_date` (which is equal to $t_{RP} - 1$, with a proof that $t_{RP} - 1 <$ `WAIT`), then the arbiter stays in the `RUNNING` state, with the new counter given by `Next_cycle c` and the new pending queue given by `Enqueue R P`. The request currently under processing remains being `r`, and the arbiter issues an `ACT r`. The case when the counter is equal to `OCAS_date` is similar, except that we use `Kind_of_req` to test if `r` is either a read or a write request and decide whether to issue a `CRD` or a `CWR`. If the counter is equal to `WAIT` − 1, however, it means that the algorithm has reached the end of a processing window. In this case, we must again look at the pending queue (Line 16): if the pending queue is empty, then we go back to `IDLE` whilst issuing a NOP, and if it is not, we start processing a new request, issuing its associated `PRE` command. In all other cases (Line 20), we stay in the `RUNNING` state, increment the counter, enqueue arriving requests, and issue a NOP.

Listing 6.5: Instantiating `Implementation_t` using the FIFO functions.

```
Instance FIFO_implementation : Implementation_t :=
    mkImplementation Init_state Next_state.
```

Finally, after having defined both functions, an instance of `Implementation_t` can be created as shown in Listing 6.5. Bear in mind that this version of FIFO does not include refresh-related operations. As mentioned in the Chapter 5, refresh commands, the related protocol, and timing constraints were included only in a later development stage, which means that neither FIFO or TDM implement such functionality. We have implemented a

version of FIFO that does issue `REF` commands, namely FIFOREF, but do not detail it here, since it is still work-in-progress.

6.2 *Time Division Multiplexing* (TDM)

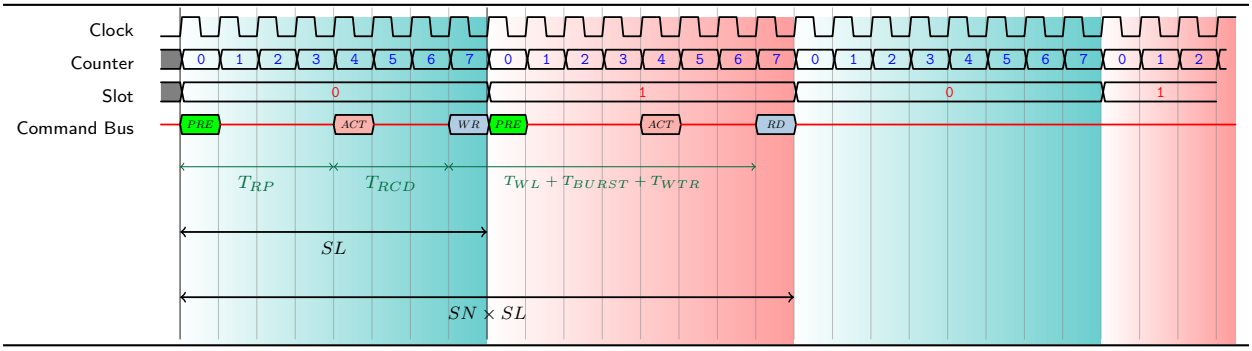


Figure 6.2: TDM command scheduling (`SN` := 2).

The TDM algorithm works as follows: we suppose that there are `SN` requestors in the system, and each requestor has its own (unbounded) waiting queue. The scheduler attributes slots (i.e. processing windows) of length `SL` to each requestor in a pre-defined order. Each requestor gets a slot, independent if it has an outstanding request in the queue. Within a slot, requests are always serviced with the same `PRE-ACT-CAS` sequence (according to a closed-page policy). Figure 6.2 show an example of a TDM scheduling scheme, where `SN` := 2 and `SL` := 8. In the scenario depicted in the figure, two requests are serviced: first, a read request belonging to requestor 0, and then a write request belonging to requestor 1. The scheduler does not service any other request after that.

In order to optimise the `SL` parameter, we make an assumption on the address mapping: we assume that requests belonging to different requestors will always target different banks (i.e., a private bank mapping policy [18]). Therefore, since each slot is associated to a different requestor, it is possible to conclude that different slots target different banks. This is formalised in Coq as shown in Listing 6.6. Note that first, as an axiom, we say that if there are two commands targeting the same bank in the trace, then the respective TDM slots at the time these commands were issued are the same. Note that we use the function `TDM_slot` to access the value of a slot at a given instant. Then, using a property from Coq's standard library: `forall A B : Prop, A ↔ B → not A ↔ not B`, we can prove `PrivateBankMapping`. Moreover, as we show soon, we also impose the fact that there must be *at least* two alternating TDM slots. From the latter assumption, it is possible to conclude that *neighbouring* slots always target different banks.

Listing 6.6: Formalising the private bank mapping assumption.

```
Axiom PrivateBankMapping_eq :
    forall t a b,
    a \in (Default_arbitrate t).(Arbiter_Commands) →
    b \in (Default_arbitrate t).(Arbiter_Commands) →
    Same_Bank a b ↔
    nat_of_ord (TDM_slot (Default_arbitrate b.(CDate)).(Implementation_State)) =
    nat_of_ord (TDM_slot (Default_arbitrate a.(CDate)).(Implementation_State)).

Lemma PrivateBankMapping :
    forall t a b,
    a \in (Default_arbitrate t).(Arbiter_Commands) →
    b \in (Default_arbitrate t).(Arbiter_Commands) →
    not Same_Bank a b ↔
    nat_of_ord (TDM_slot (Default_arbitrate b.(CDate)).(Implementation_State)) !=
    nat_of_ord (TDM_slot (Default_arbitrate a.(CDate)).(Implementation_State)).
Proof. (* ... omitted ... *) Qed.
```

Knowing that neighbouring slots will never have requests targeting the same bank allows us to always issue a `PRE` on the cycle immediately following a `CAS` (`CRD` or `CWR`), since no timing constraints apply between `CAS` and `PRE` commands to different banks. This assumption is reasonable, since the minimum number of banks to satisfy such assumption is three, if the number of requestors is odd (two, if the number of requests is even). Note, in addition, that `SL` must still be chosen in such a way that the inter-bank constraints between `CWR` and `CRD` commands are respected. Besides that, there are other constraints that must be respected when choosing a value for `SL`. As for FIFO, Listing 6.7 contains the TDM parameters and all necessary axioms/assumptions w.r.t the choice of `SN` and `SL`. We define `ACT_date` and `CAS_date` as constants marking the time when `ACT` and `CAS` commands have to be issued within a TDM slot.

Implementing the algorithm requires implementing the functions `Init` and `Next`, from `Implementation_t`. Similar to FIFO, however, we must define a requestor type and an internal state beforehand. We start by defining a counter `Slot_t` to account for slots. This counter is encoded to be of type `ordinal SN` – which ensures that all counter values will be strictly less than `SN`. The requestor type is instantiated with the same counter type, since requestors are now identified by a numeric value (which is the same as their slot number). Furthermore, we define another wrap-around counter, `Counter_t`, to account for cycles within a slot.

The internal state of a TDM is defined as an inductive datatype `TDM_state_t`. Again,

Listing 6.7: TDM configuration and axioms.

```
Definition ACT_date := T_RP.+1.
Definition CAS_date := ACT_date + T_RCD.+1.

Class TDM_configuration := {
    SN : nat;
    SN_one : 1 < SN; (* At least two slots *)

    SL : nat;
    SL_pos : 0 < SL; (* SL must be non-zero *)
    SL_ACT : ACT_date + 1 < SL; (* Window is large enough to issue an ACT *)
    SL_CAS : CAS_date + 1 < SL; (* Window is large enough to issue a CAS *)

    (* Axioms needed for proofs *)
    WTR_SL : T_WL + T_BURST + T_WTR_l < SL;
    T_RTP_SN_SL : T_RTP < SN * SL - CAS_date;
    T_WTP_SN_SL : (T_WR + T_WL + T_BURST) < SN * SL - CAS_date;
    T_RAS_SL : T_RP.+1 + T_RAS < SL;
    T_RC_SL : T_RC < SL;
    T_RRD_SL : T_RRD_l < SL;
    T_RTW_SL : T_RTW < SL;
    T_CCD_SL : T_CCD_l < SL;
    T_FAW_3SL : T_FAW < SL + SL + SL;
}.
```

Listing 6.8: TDM preliminearies: deifning a requestor type and internal state.

```
1  Definition Slot_t := ordinal SN. (* Account for TDM slots *)
2
3  Instance REQUESTOR_CFG : Requestor_configuration := {
4      Requestor_t := Slot_t
5  }.
6
7  Definition Counter_t := ordinal SL. (* Account for cycles elapsed within a slot *)
8
9  Inductive TDM_state_t :=
10     | IDLE    : Slot_t → Counter_t → Requests_t → TDM_state_t
11     | RUNNING : Slot_t → Counter_t → Requests_t → Request_t → TDM_state_t.
12
13 Instance ARBITER_CFG : Arbiter_configuration := {
14     State_t := TDM_state_t;
15 }.
```

Listing 6.9: TDM initial and next state functions.

```
1  Definition Init_state R := IDLE OZSlot OZCycle R.
2
3  Definition Next_state R (AS : TDM_state_t) : (TDM_state_t * Command_kind_t) :=
4      match AS with
5        | IDLE s c P ⇒ (* If scheduler is IDLE *)
6          let P' := Enqueue R P in (* enqueue arriving requests into request queue *)
7          let s' := Next_slot s c in (* calculate next slot value *)
8          let c' := Next_cycle c in (* calculate next cycle value *)
9          if (c = OZCycle) then (* if at beginning of slot *)
10             match Pending_of s P with (* filter pending queue according to slot *)
11             | [::] ⇒ (IDLE s' c' P',  NOP)
12             | r::_ ⇒ (RUNNING s' c' (Enqueue R (Dequeue r P)) r, PRE r)
13             end
14         else (IDLE s' c' P',  NOP)
15       | RUNNING s c P r ⇒ (* If scheduler is RUNNING *)
16         let P' := Enqueue R P in
17         let s' := Next_slot s c in
18         let c' := Next_cycle c in
19         if c = OACT_date then(RUNNING s' c' P' r, ACT r) (* Issue ACT *)
20         else if c = OCAS_date then (RUNNING s' c' P' r, (Kind_of_req r) r) (* Issue CAS *)
21         else if c = OLastCycle then (IDLE s' c' P', NOP) (* Finish window *)
22         else (RUNNING s' c' P' r,  NOP) (* Otherwise just stay at runing *)
23     end.
24
25 Instance TDM_implementation := mkImplementation Init_state Next_state.
```

similar to FIFO, the algorithm includes an `IDLE` and a `RUNNING` state. Each state carries "variables" (which are actually defining distinct states): both `IDLE` and `RUNNING` states include a counter of type `Slot_t`, a counter of type `Counter_t`, and a queue of pending requests, of type `Requests_t`. The `RUNNING` state also includes an additional term of type `Request_t`, used to store the request currently under processing. These definitions are all shown in Listing 6.8. Finally, at Line 13, we indicate to Coq that `ARBITER_CFG` is an instance of `Arbiter_configuration` (see Listing 5.21).

The algorithm is implemented as shown in Listing 6.9. We start by defining the initial state as being `IDLE`, with slot counter equal to 0 (`OZSlot`), cycle counter being equal to 0 (`OZCycle`), and pending queue made of the set of arriving requests at initialisation (`R`). The `Next_state` function implements the specification `Next`: it receives a set of arriving requests `R` and a state `AS`, and produces a pair consisting of a new state and a command to be issued to the device.

In Listing 6.9, `Pending_of s P` returns the sub-set of requests in P whose requestor is equal to `s`. `Next_slot s c` increments `s` by one when `c` reaches its bound, but also wrapping around `SN` − 1. `Next_cycle` increments `c` by one, wrapping around `SL` − 1. We leave the task of analysing the rest of the algorithm to the reader.

Finally, at Line 25, we create an instance of `Implementation_t` by calling the constructor `mkImplementation` (see Listing 5.21) with the functions `Init_state` and `Next_state` as arguments. Again, as for FIFO, this version of TDM does not manage refresh commands nor its related timing constraints. As mentioned earlier, refreshes were only added at a later development state, and a TDM version that does manage refreshes (called TDMREF) has been implemented, but it is still work-in-progress.

6.3 Proving Properties

So far, Sections 6.1 and 6.2 have detailed the process of writing algorithms using one of the abstractions available in CoqDRAM. Now, these algorithms need to be "turned into" arbiters confirming to the formal specification described in Chapter 5. In order to achieve that, we need to show that these implementations produce valid traces over time by proving the necessary proof obligations.

A single proof is presented in detail here: the proof of `Request_handled` for the TDM scheduler. We do not discuss details about the many other proofs in the framework, such as the proofs satisfying the properties stated in `Trace_t`. The reason for that is that most proofs are repetitive and quite long, and showing more proof scripts does not add much meaningful information. By showing the proof of `Request_handled`, we present to the reader

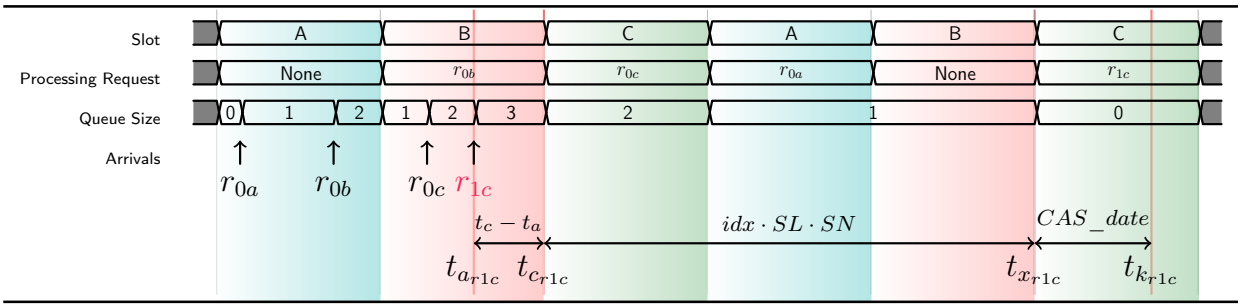


Figure 6.3: Time stamps in proof of `Request_handled`.

a comprehensive-enough overview of what the proofs in the CoqDRAM framework look like. Moreover, since the techniques employed for proving both FIFO and TDM are very similar, and TDM is more complex (hence more interesting), we discuss the proof of `Request_handled` only for TDM.

The proof of the theorem that satisfies the PO `Requests_handled` is organised in steps, written in Coq as `Lemmas`. The idea at the core of the proof is to advance in time step-by-step, starting from the instant t_a when a request arrives in the system, through intermediate steps up until the point where a matching `CAS` is issued – which represents the completion date of that request.

Figure 6.3 provides an illustration of these steps for request r_{1c} under TDM scheduling. In the depicted scenario, there are three requestors: A, B and C (`SN` := 3). The slot length is some arbitrary `SL`. The three requestors issue four requests: r_{0a}, r_{0b}, r_{0c}, and r_{1c}, with arrivals in that order. The notation r_{xp} represents the x-th request from requestor p. The time stamps that are meaningful for the proof (and shown in Figure 6.3) are first described informally: $t_{a_{r1c}}$ is the arrival date of request r_{1c}; $t_{c_{r1c}}$ is the earliest date after the arrival of r_{1c} when the scheduler can take a scheduling decision concerning requests of requestor c; $t_{x_{r1c}}$ is the date when r_{1c} gets to the head of its requestor's pending queue and starts being processed; and $t_{k_{r1c}}$ is the completion date, i.e., the date when the `CAS` command belonging to r_{1c} is finally issued. Furthermore, in the figure, idx represents the position (index) of r_{1c} in its requestor's pending queue. In this specific case, since requestor C had already issued r_{0c} before r_{1c}, idx at $t_{c_{r1c}}$ is equal to 1, since r_{0c} already occupied position 0.

Concerning the actual proofs, i.e., the proof scripts for individual steps, we rely heavily on induction over time or lists/queues; case analysis on the state variables described in the previous section; and reduction (i.e., simplification of terms). In the following text, we describe most of the high-level proof strategy – thus providing the reader a logical sequence of steps, which can be largely reproduced for other algorithms. The actual content of the proof scripts is often omitted, and in some cases, discussed only superficially through comments

in the code snippets or in the text.

For some basic understanding, the most commonly used tactics in the proof scripts are: `intros`, which introduces premises (i.e. antecedents) in logical implications to the *local context*; `destruct`, which performs case analysis on inductive types, creating a *sub-goal* for each constructor; `unfold`, which unfolds constants, i.e., replace their definitions by their bodies/values (a process also called δ-reduction); `induction`, which applies *the induction principle* of an inductive type; `rewrite`, which substitutes one or many occurrences of a term by an equivalent term;[1] and `simpl`, a smart reduction tactic, which reduces a term to "something still readable", instead of fully normalising it.

Moreover, we rely heavily on *some* proof constructs from `mathcomp`'s `SSReflect` proof language.[2] While the differences between `SSReflect` tactics and tactics from vanilla Coq lie out of the scope of this dissertation, it is worth mentioning that most proofs in the framework mix both proof styles.

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

Step 1: `Pending_on_arrival` **(Listing 6.10)** *We prove that ra is instantly inserted into the request queue.* Proven by case analysis on t_a, case analysis on state variables, and δ-reduction (i.e., function unfolding).

Definition 2 Let t_c be the next instant after t_a when the arbiter 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.* Proven by 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. Such function is an abstraction of `Pending_of`, from Listing 6.9. Let idx be the position of ra in $P(t_c, A)$. Let t_x be equal to $t_c + idx \cdot SN \cdot SL$.

Step 3: `Request_index_zero` **(Listing 6.11)** *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 $idx \cdot SN \cdot SL$ cycles after t_c.* Proven by induction on idx. In Listing 6.11, the function `Requestor_slot_start ta ra` calculates t_c from a given arrival date t_a and request ra; and `Pending_of` returns a sub-set

[1] Given that such equivalence is stated through a rewritable Leibiniz or setoid equality.
[2] https://coq.inria.fr/doc/V8.19.0/refman/proof-engine/ssreflect-proof-language.html

Listing 6.10: Step 1 – `Pending_on_arrival`.

```
(* Returns the waiting queue/list of outstanding requests for a given state AS *)
Definition TDM_pending (AS : State_t) :=
    match AS with
        | IDLE _ _ P      ⇒ P
        | RUNNING _ _ P _ ⇒ P
    end.

Lemma Pending_on_arrival : forall (ta : nat) (ra : Request_t),
    ra \in Arrival_at ta →
    ra \in TDM_pending ((Default_arbitrate ta).(Implementation_State)).
Proof.
    intros HA. (* HA : ra \in Arrival_at ta *)
    destruct ta. (* Case analysis on ta *)

    (* Case 1 -- Solved by the definition of TDM_pending:
       TDM_pending (Default_arbitrate 0).(Implementation_State) reduces to
       TDM_pending (IDLE OZSlot OZCycle (Arrival_at 0)), which reduces to
       (Arrival_at 0), by the definition ot TDM_pending. The resulting goal,
       (Arrival_at 0), is exactly the same as hypothesis HA. To perform all
       such steps, it suffices to unfold the definition of TDM_pending. *)
    – by unfold TDM_pending.

    (* Case 2 -- Case analysis on state variables and delta-reduction *)
    (* Unfold function definitions and reduces *)
    – unfold TDM_pending; simpl; unfold Next_state
      (* Analyse every case in Next_state *)
      destruct (Default_arbitrate ta).(Implementation_State),
      (c = OZCycle), (c = OACT_date), (c = OCAS_date), (c = OLastCycle);
      try (destruct (Pending_of s P));
      (* Reduces what can be reduced (applies to all sub-goals) *)
      simpl in *;
      (* Conclusions follows by the definition of Enqueue, two existing Lemmas
        (mem_cat and orbT), and the hypothesis HA *)
      by unfold Enqueue; rewrite mem_cat HA orbT.
Qed.
```

Listing 6.11: Step 3 – `Request_index_zero`.

```
Lemma Request_index_zero (ta : nat) (ra : Request_t)
    (* Instant tc *)
       let tc := Requestor_slot_start ta ra.(Requestor) in
    (* The state of the scheduler at tc *)
       let S := (Default_arbitrate tc).(Implementation_State) in
    (* The position of ra in its requestor's pending queue at tc *)
       let idx := index ra (Pending_of ra.(Requestor) (TDM_pending S)) in
    (* Antecedent of implication : if ra is pending at tc *)
       ra \in (TDM_pending S) →
    (* The state of the scheduler at tc + idx * SN * SL *)
       let S' := (Default_arbitrate (tc + idx * SN * SL)).(Implementation_State) in
    (* Consequent of the implication : ra must still be pending at tx *)
       ra \in (TDM_pending S') ∧
    (* AND ra must arrive in the head of its requestor's pending queue at tx *)
       index ra (Pending_of ra.(Requestor) (TDM_pending S')) = 0.
Proof. (* proof omitted *) Qed.
```

of requests in the request queue belonging to a given requestor. In this case, `Pending_of ra.(Requestor)(TDM_pending S)` returns the list of outstanding requests at `S` (the state at t_c) that were issued by the requestor of `ra`. This is implemented using the `filter` function from `mathcomp ssreflect.seq` library, and it serves as a way to mimic the hardware behaviour of having one queue per requestor. Although the proof script is omitted, the full Lemma is provided in Listing 6.11 and is thoroughly commented for easiness of understanding.

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.* This follows directly from Steps 1, 2, and 3.

Step 5.1: `Request_slot_start_aligned` **(Listing 6.12)** *We prove that the internal counter (`Counter_t` in Listing 6.8 and `c` in Listing 6.9) is equal to 0 at the beginning of every slot.* This follows from Lemmas concerning modulo arithmetic in Coq's standard library.

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 (Step 5.1), then the request starts to be processed the very next cycle.* Proven by 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*

Listing 6.12: Step 5.1 – `Request_slot_start_aligned`.

```
(* Returns the internal counter for a given state AS *)
Definition TDM_counter (AS : State_t) :=
    match AS with
        | IDLE _ c _      ⇒ c
        | RUNNING _ c _ _ ⇒ c
    end.

Lemma Requestor_slot_start_aligned (ta : nat) (s : Slot_t)
    (* S is the state at a given start of slot *)
    let S := (Default_arbitrate (Requestor_slot_start ta s)).(Implementation_State) in
    (* The counter at S must always be equal to OZCycle *)
    TDM_counter S = OZCycle.
Proof. (* proof omitted *) Qed.
```

Listing 6.13: Step 7 – `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 idx  := index ra P in
   let tx := tc + idx * SN * SL in
   ra \in (Arrival_at ta) → forall d, d < SL.−1 →
   (* The arbiter state at tx + d *)
   let S':=(Default_arbitrate (tx + d)).(Implementation_State) in
   (* The conclusion: counter at tx + d and processed request *)
   (TDM_counter S' = d + 1) && (TDM_request S' = ra)
Proof. ...  (* Applies lemmas described in Steps 4,5.1,5.2 and 6 *) Qed.
```

request will remain being served until at least $t + d$. Proven by induction over d.

Step 7: `Request_processing` **(Listing 6.13)** *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* $t_x + d$ *and that the counter at* $t_x + d$ *will be equal to* $d + 1$. This follows from the previous steps.

Definition 4 Let t_k be $t_x +$ `CAS_date`, where `CAS_date` is the `CAS` command offset within a TDM slot (cf. Figure 6.3).

Step 8.1: `Request_CAS` **(Listing 6.14)** *We use Step 7 with* d *equal to* ***`CAS_date`*** *to prove that ra will have its* ***`CAS`*** *issued at* $t_x +$ ***`CAS_date`***. It follows directly from Step 7 and δ-

Listing 6.14: Step 8.1 – `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 idx  := index ra (Pending_of ra.(Requestor) S) in
    let tk := tc + idx * SN * SL + CAS_date in
    ra \in (Arrival_at ta) →
    (CAS_of_req ra tk) \in (Default_arbitrate tk).(Arbiter_Commands)
Proof.
    (* HA : ra \in Arrival_at ta *)
        intros HA.
    (* Use Step 7 : any request is processed until the CAS date is reached *)
        apply Request_processing with (d := CAS_date) in HA as HR.
    ...  (* rest of the proof is omitted *)
Qed.
```

reduction.

Step 8.2: `Requests_handled` **(Listing 6.15)** *Finally, we prove the theorem that satisfies the* ***`Request_handled`*** *PO*, which follows from Step 8.1.

Listing 6.15: Step 8.2 – Proof of final Theorem.

```
Theorem Request_handled (ta : nat) (ra : Request_t)
    ra \in (Arrival_at ta) → exists tc,
    (CAS_of_req ra tc) \in (Default_arbitrate tc).(Arbiter_Commands)
Proof.
    intros HA.
    (* Use Step 8.1: get the CAS command *)
    apply Request_CAS in HA as H.
    (* Finish the proof: the tc we are looking for is the one coming from the
        conclusion of Request_CAS *)
    set tc := _.+1 in H; exists (tc); exact H.
Qed.
```

We would like to emphasise that, as it can be seen in Listing 6.14, t_k is the typical closed-form expression one would expect for TDM:

$$t_k = t_c + idx \cdot SL \cdot SN + \texttt{CAS\_date} \tag{6.1}$$

Equation 6.1 is made of three parts:

1. t_c, which is bounded by $t_a + SL - 1$;

2. $idx \cdot SN \cdot SL$, which for given values of SN and SL, is bounded by the maximum number of pending requests allowed by a requestor; and

3. `CAS_date`, which is constant.

A timing analysis tool may now 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 *idx* through an additional model of the requestor (i.e., by modelling the arrival function). Even more, analyses that are themselves formalised in Coq could simply reuse our proofs in order certify memory latency bounds.

6.4 Putting It All Together

Finally, in Listing 6.16, we show the code that instantiates the trace through the constructor `mkTrace`, defines the proven scheduling function, `TDM_arbiter`, and finally instantiates the arbiter through `mkArbiter`. Note how we pass the timing and protocol correctness proofs as arguments to `mkTrace` and a proof of `Requests_handled` as an argument to `mkArbiter`. *Note also the call to `mkTrace` does not include any refresh-related proof obligations.* As previously mentioned, the reason for that is that refresh commands and related timing constraints were added only at a later development stage. Other algorithms that use the bank machine abstraction do comply with the later-added `REF`-related POs, as described in Chapter 8. Finally, note how the command trace itself is obtained from the last state produced by the `Default_arbitrate` function.

Be mindful that algorithms developed and proved according to the presented methodology are valid for any DRAM device, as a long as a small set of assumptions hold (which should be trivial for any *existing* DDR3 and DDR4 device, given that these axioms only model the validity and coherence of timing constraints and the number of banks and bank-groups). As discussed in the previous chapter, all needed parameters and assumptions that algorithms rely upon are conveniently grouped in `System_configuration` and `FIFO_configuration` (or `TDM_configuration`). This is very advantageous, since formally verifying algorithms using model checking, for instance, would require concrete values for timing constraints and algorithm parameters.[3] In summary, CoqDRAM algorithms are proved correct against any set of timing constraints and parameters, as long as a compact set of assumptions hold.

[3]Although parameterised model checking does exist [113], it is still not very mature.

Listing 6.16: Generating proved traces and arbiters.

```
(* Creates a proved trace from the implementation *)
Program Definition TDM_arbitrate (t : nat) := mkTrace
    (* ------ The trace of commands ------ *)
    (Default_arbitrate t).(Arbiter_Commands)
    (Default_arbitrate t).(Arbiter_Time)
    (* ------ Proofs of feasibility ------ *)
    (Default_arbitrate_cmds_uniq t)
    (Default_arbitrate_cmds_date t)
    (* ------ Timing correctness ------ *)
    (Cmds_T_RCD_ok t)
    (Cmds_T_RP_ok t)
    (Cmds_T_RC_ok t)
    (Cmds_T_RAS_ok t)
    (Cmds_T_RTP_ok t)
    (Cmds_T_WTP_ok t)
    (Cmds_T_RtoW_ok t)
    (Cmds_T_WtoR_SBG_ok t)
    (Cmds_T_WtoR_DBG_ok t)
    (Cmds_T_CCD_SBG_ok t)
    (Cmds_T_CCD_DBG_ok t)
    (Cmds_T_FAW_ok t)
    (Cmds_T_RRD_SBG_ok t)
    (Cmds_T_RRD_DBG_ok t)
    (* ------ Protocol correctness ------ *)
    (Cmds_ACT_ok t)
    (Cmds_row_ok t)
    (Cmds_initial t)
Defined.

(* Instantiate the TDM arbiter *)
Instance TDM_arbiter := mkArbiter AF TDM_arbitrate Requests_handled.
```

6.4.1 Code Size & Compilation Times

We can also analyse the size and the compilation time of the entire specification, implementation interface, concrete implementations, and proofs. As it can be seen in Table 6.1, 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.[4] As proofs are checked by Coq's kernel, compiling the files containing the proofs also takes more time than files containing simply code.

Results were 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 – 8.17.1. Compilation times were obtained using the *time* program from command line.

	Lines of code	**Compilation time (s)**
Specification	750	5.24
Implementation Interface	899	10.91
FIFO implementation	133	1.1
FIFO proofs	2217	27.12
TDM implementation	228	0.97
TDM proofs	2431	54.46

Table 6.1: Size of the code and compilation time.

[4] As stated by Boldo et al. [114], 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).

Chapter 7

A Validation Experiment

CoqDRAM is a high-level framework, conceived to explore and design DRAM scheduling algorithms. Even if algorithms designed in CoqDRAM enjoy the trustworthiness coming from machine-checked proofs, the correctness criteria stated in the specification are still written by a human, and are thus subject to errors. In other words, even if implementations are proved against a formal specification, nothing guarantees that the specification itself really models and captures the correct behaviour of DRAM devices.

Bear in mind that although this is true, we still claim that manually checking that a set of correctness criteria have been correctly stated/encoded is much less burdensome than a thorough check of hand-written proofs.

However, bugs in the specification may still occur. To address that issue and *gain confidence* that the CoqDRAM formal specification is *correct*, we perform a validation experiment. The experiment consists in generating code from a Coq implementation and executing it on a third party DRAM simulation framework/environment.

As a pre-requisite, the "*host*" environment on which our *trusted* controllers run must have a way to detect faulty behaviours, such as exceptions, for a software-based simulation, or immediate *SystemVerilog Assertions* (SVA), for a hardware-based simulation. Moreover, we try to insert our trusted component in a way that modifies the host environment as little as possible – with the aim of showing that the methodology is modular and compatible with other DRAM simulation environments.

In addition, the reader should keep in mind that the objective of these experiments is *not* to evaluate simulation performance nor bandwidth and throughput of our implementations. As it will be presented in the following, the simulation methodologies rely on far-from-optimal techniques w.r.t performance. Furthermore, the algorithms we test, FIFO and TDM, are fairly basic and do not compete with state-of-the-art DRAM controllers w.r.t bandwidth. The objective is rather to make a proof-of-concept, i.e., show some evidence

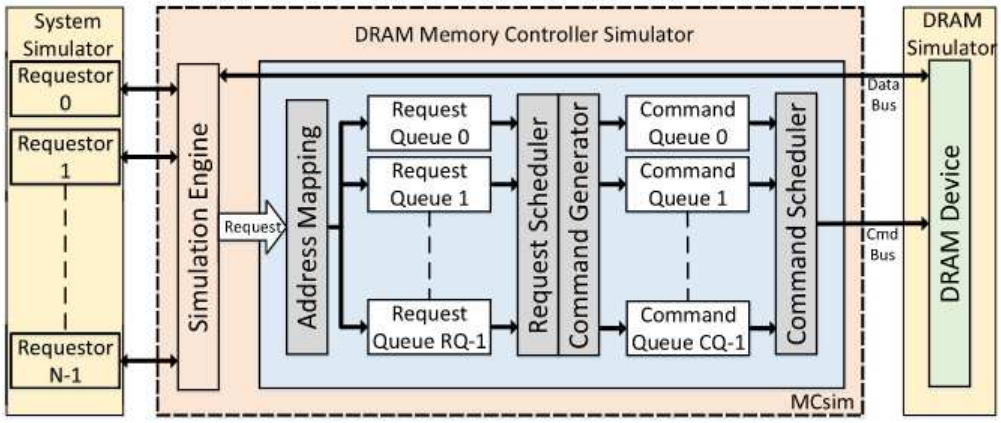


Figure 7.1: MCsim architecture [44].

that the controllers developed in CoqDRAM and are functional and run without triggering any errors.

In the experiment described in this chapter we use MCsim [44], a cycle-accurate object-oriented simulation framework for DRAM controllers. MCsim is written in C++ and can be built on any platform supporting C++11. The fact that MCsim's design focuses on modularity and extensibility makes it a great choice to host our algorithms. The original MCsim architecture is shown in Figure 7.1 [44].

MCsim's "*front-end*" (i.e., "System Simulator" in Figure 7.1) can run as a trace-based simulator. It also provides a way to connect to CPU simulators, such as gem5 [115] and MacSim [116] (although we did not explore that feature). As for the "*back-end*", i.e., the connection to DRAM device models, MCsim employs a generalised interface – so that the framework is not tied to any specific DRAM device model. MCsim can also connect to Ramulator [117] – an example of a validated and open-source device simulator which supports a wide variety of DRAM standards.

Moreover, as it can be seen in Figure 7.1, MCsim proposes a general structure for DRAM controllers, made of four parts: address mapping, request scheduling, command generation, and command scheduling. Along with the fact that the queuing mechanism is highly configurable, such structure allowed the authors of MCsim to model 14 highly-cited DRAM scheduling algorithms of different sorts [44] (i.e., algorithms focusing on predictability, performance-intensive algorithms, and "middle-ground" algorithms).

In order to integrate CoqDRAM algorithms, three parts of MCsim are replaced by generated code derived from CoqDRAM algorithms: request scheduling, command generation, and command scheduling. This comes from the fact that CoqDRAM algorithms perform exactly these three tasks. Everything else in MCsim is kept, including its front- and back-ends, its address mapping mechanism, exception/error tracking features, et cetera.

From an engineering perspective, integrating Coq generated code to such a system is not trivial. As it is based on a functional paradigm, Coq programs can be extracted to OCaml, Haskell, and Scheme. Since Haskell relies on a relatively well-document library for cooperating with C programs, we opt to extract CoqDRAM algorithms to Haskell. The Haskell library is called "*Foreign Function Interface*" (FFI[1]) and allows Haskell programs to call *foreign* functions and foreign functions to call Haskell code. In our case, we go in the latter direction, as we want MCsim to call Haskell code generated from Coq.

In the following, we first discuss the engineering aspects of making Coq-generated Haskell code cooperate with MCsim's code. Second, we discuss what are the exact changes made in MCsim and where the Haskell calls are made. Finally, we discuss the experimental results.

Representing data

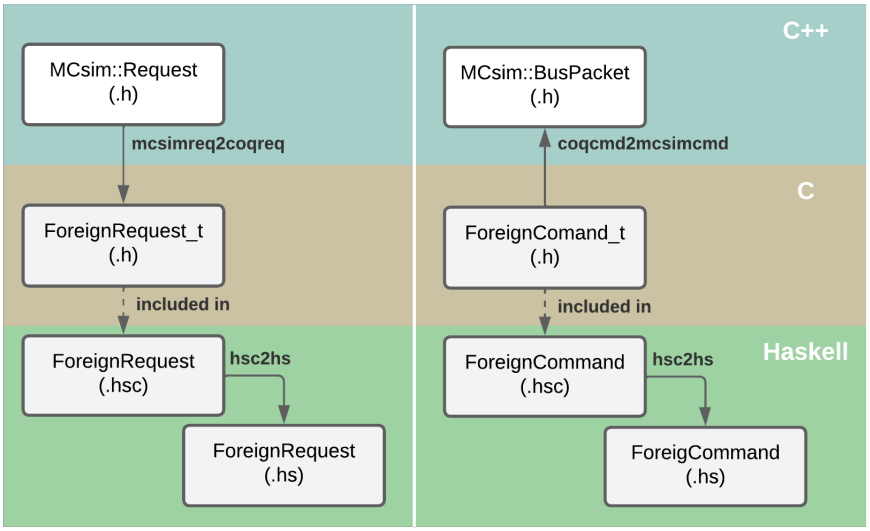


Figure 7.2: Data exchange between MCsim and Haskell algorithms.

Although foreign function calls only happen in one direction, we do need to exchange *data* in both directions: memory requests from MCsim need to be processed by our algorithm, which will output a command – which will then be interpreted by MCsim and sent to the device model. Figure 7.2 depicts how data structures are translated between MCsim and Haskell.

MCsim defines a class called `Request`, which models memory requests. Ideally, we would want to translate such class definition directly to Haskell. However, since Haskell's FFI does not support some of the C++ class syntax, we define an intermediate C `struct` in a separate header file, which mirrors MCsims' `Request`. We call this `struct ForeignRequest_t`. Similarly,

[1] https://wiki.haskell.org/Foreign_Function_Interface

MCsim models DRAM commands in the class `BusPacket` – and we likewise define the C `struct ForeignCommand_t`.

The function `mcsimreq2coqreq` translates the original C++ `Request` class to the C structure passed to Haskell. Commands need a translation in the opposite sense: since commands coming from Haskell must be handled in C++, we define the function `coqcmd2mcsimcmd` to translate a `ForeignCommand` to a `BusPacket`. Since the procedure for both requests and commands are quite similar, we concentrate most of the following discussion on `Request`.

To translate the C `struct ForeignRequest_t` to a Haskell data structure, we use *hsc2hs* – a program that can be used to automate some parts of the process of writing Haskell bindings for C code. In more detail, quoting the package's page,[2] "it [hsc2hs] reads an almost-Haskell file with embedded constructs and outputs a real Haskell file with these constructs processed, based on information taken from some C headers. The extra headers provide Haskell counterparts of C types, values of C constraints, including sizes of C types, and access to field of C structs". This can be visualised in Figure 7.2: hsc2hs processes a file *ForeignRequest.hsc*, which includes the C header *ForeignRequest_ t.h* to produce a pure Haskell file – *ForeignRequest.hs*.

In more detail, in Haskell, a garbage collector typically manages the memory. When using the FFI, however, we sometimes need to do some manual memory management to comply with the data representation of the foreign codes. Such process of manually managing the memory is called *marshalling* in the FFI documentation.

In order to correctly manipulate the C struct in Haskell, we use a type called `Ptr`, declared in the FFI. A value of type `Ptr` represents a pointer to an object, or an array of objects, which may be *marshalled* to or from Haskell values of the type pointed by `Ptr`. In our case, `Ptr` is used to define a type which represents a pointer to the `ForeignRequest_t` structure:

```
▷ type PtrRequest = Ptr ForeignRequest_t.
```

A pointer to the `ForeignCommand_t` structure can be defined likewise:

```
▷ type PtrCommand = Ptr ForeignCommand_t.
```

Furthermore, the *.hsc* file contains functions to help users handle this marshalling of data and find the correct field offsets in a `struct` referenced by a pointer. In other words, the *.hsc* makes it "easy" for users to specify how fields of a structure can be accessed in Haskell.

Coming back to the CoqDRAM implementation interface, defined by `Implementation_t` (c.f. Listing 5.21). Since the `Next` function interface also expects an arbiter state in order to generate the next state, it is also necessary to model CoqDRAM *arbiter states* in MCsim. However, differently than requests and commands, states are not manipulated by MCsim

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

itself (i.e., there is no corresponding `struct` that represents a CoqDRAM state). Therefore, a state can be modelled as an opaque pointer, since no information is needed on how to access information. On the Haskell side, we also require arbiter states to be protected from Haskell's garbage collection, and for that, we use the FFI type `StablePtr`:

```
▷ type ArbiterPtr = StablePtr Arbiter.Arbiter_state_t.
```

Note that marshalling is not necessary for `ArbiterPtr`, since a corresponding `struct` does not exist in the C++ side. The command issued to the memory device is always taken as the last command in the `Arbiter_Commands` list – part of the state produced by the `Next` function. For conciseness, we do not discuss marshalling in further detail.

Calling the Coq-generated transition function from C++

The underlying idea of integrating CoqDRAM to MCsim is fairly simple: since MCsim is cycle-accurate, it suffices to call CoqDRAM's `Init_state` at initialisation and `Next_state` at each subsequent cycle. Unfortunately, it is not possible to simply make a call to the Coq-generated `Next_state` function from C++. The Haskell code relies on algebraic data-types, i.e., self-defined types – coming from Coq inductive type definitions. This representation is not directly compatible with C types, which occupy a precise size in memory.

Hence, it is necessary to manually write *wrapper* functions in Haskell that take arguments compatible with C types (e.g. integers and pointers), translate them to the Coq-generated types, call the Coq-generated algorithm and translates the result of the algorithm to a C type again (a pointer). For each implementation, these wrapper functions need to be written only once.

Moreover, we add the "`coq_`" prefix to every Coq-generated Haskell definition, to avoid confusion with the actual Coq definitions. We also use qualified names to improve code readability. For each implementation, these wrappers and conversion functions are grouped in a file called "*App<X>.hs*", where *X* stands for the implementation name. For TDM, for instance, the wrapper file is called *AppTDM.hs*, and the TDM `Next_state` function can be referenced as `TDM.coq_Next_state`.

A Haskell-wrapper around `TDM.coq_Next_state` has its type signature shown below:

```
tdm_next_state_wrapper :: CInt -> ForeignRequest.PtrRequest -> ArbiterPtr
   -> IO ArbiterPtr
```

The function `tdm_next_state_wrapper` takes three arguments: a C integer (of type `CInt`), a pointer to a `ForeignRequest_t` structure, and a pointer to a `Arbiter.Arbiter_state_t`. Note that `PtrRequest` is actually a *list of requests* – representing the list of arriving requests expected

by the `Next_state` function (see Listing 5.21). The first argument, of type `CInt`, is thus used to precise the amount of memory read, starting from the pointer location. This means that the C++ code must make the Haskell call passing not only a pointer to `ForeignRequest_t` as an argument, but also the number of `ForeignRequest_t` objects to be read. The function also expects the current arbiter state (of type `ArbiterPtr`) as an argument and returns the next arbiter state wrapped around the Haskell `IO` monad[3]). The monad is responsible for indicating to the compiler that the function is impure, i.e., side effects can take place during the computation, which is the case when dealing with pointers.

The implementation of `tdm_next_state_wrapper` is omitted for conciseness, since it requires knownledge of the Haskell and FFI syntax – which is out of scope here. But briefly, the function works by analysing the first argument: if it is not zero, it turns the second argument (of type `PtrRequest`) into a proper *Haskell list*, in which elements are of type `ForeignRequest_t`. Then, it converts this Haskell list to a Coq list of requests, which is finally fed to the Coq-generated function, `TDM.coq_Next_state`. If the value of the first argument is 0, then the same procedure applies, only using an empty Haskell list instead – which is translated to an empty Coq list (cf. `nil` from Section 2.3.1).

Furthermore, as the current design communicates arbiter states to MCsim, a second function, `getcommand`, is necessary to retrieve the last command from the trace contained inside the opaque `ArbiterPtr`. Its type signature is shown below:

```
getcommand :: ArbiterPtr -> IO (ForeignCommand.PtrCommand)
```

Modifying MCsim

Our modifications to MCsim take place exclusively in the `MemoryController` class, which plays a central role in MCsim's design. For details on MCsim's design and its class diagram, see Figure 3 from the paper introducing MCsim [44].

`MemoryController` is responsible for orchestrating all other tasks that take place during a trace execution: from configuring the operating modes, address mapping, executing scheduling algorithms, to reporting results and tracking statistics. Specifically, the class provides the method `update`, which calls the request scheduler, the command generator, and the command scheduler, in that order. Since we want to replace these three tasks with generated code, our modifications take place in the method `update`.

As a disclaimer, this is undoubtedly not a great way of inserting a new scheduler in MCsim, which provides a cleaner modular interface to do that. In more detail, since CoqDRAM algorithms perform request scheduling, command generation, and command scheduling in a

[3] https://www.haskell.org/tutorial/io.html

combined fashion that cannot be split automatically, it is not possible to reuse the interface classes intended for these purpose in MCSim.

As it can be seen in Listing 7.1, we insert four new attributes in the declaration of the `MemoryController` class:

Listing 7.1: Modifications in the `MemoryController` class.

```
class MemoryController {
    public:
        // ...
    private:
        // ...
        vector<ForeignRequest_t *> coq_requestQueue;
        vector<ForeignRequest_t *> coq_pending;
        void * coq_state = nullptr;
        ForeignCommand_t * coq_cmd;
        // ...
}
```

- `coq_requestQueue` holds arriving requests on a single cycle – it is clared every cycle after executing the scheduling function and replenished again in the beginning of the next cycle with requests arriving that cycle;
- `coq_pending` holds the requests that have been sent to the Coq scheduling function. When the scheduler issues a `CWR` or `CRD` command for a request in that list, the request is considered completed and is thus removed from the pending queue;
- `coq_state` holds the current Coq arbiter state passed to the scheduling function;
- `coq_cmd` is used to store the command emitted by the CoqDRAM algorithm, resulting from a call to the `getcommand` function.

Finally, Listing 7.2 shows how the calls to the Coq-generated Haskell functions in the MCsim code are made. From Lines 2 to 6, we create an array of `ForeignRequest_t` from `coq_requestQueue`. At Line 8, we check the current clock cycle: if it is 0, then a call to `tdm_init_state_wrapper` is made, with arguments `num_req` and `reqlist`, where `num_req` is the number of requests to be read from `reqlist`. In every other cycle, `tdm_next_state_wrapper` is called instead, where `coq_state` is the current arbiter state. Note that `coq_state` is overwritten by the function call. At Line 13, we use the `getcomand` function to retrieve the last issued command from `coq_state` and finally, at Line 15, the function `coqcmd2mcsimcmd` is used to

Listing 7.2: Call to the Coq-generated Haskell scheduling function.

```
void MemoryController::update() {
    unsigned int num_req = coq_requestQueue.size();
    ForeignRequest_t * reqlist = (ForeignRequest_t *) malloc(num_req *
       sizeof(ForeignRequest_t));

    for (size_t i = 0; i < num_req; i++)
        reqlist[i] = *(coq_requestQueue[i]);

    if(clockCycle == 0)
        coq_state = tdm_init_state_wrapper(num_req,reqlist);
    else
        coq_state = tdm_next_state_wrapper(num_req,reqlist,coq_state);

    coq_cmd = (ForeignCommand_t *) getcommand(coq_state);
    // Creating a BusPacket
    outgoingCmd = coqcmd2mcsimcmd(coq_cmd);
    // ... continues with unmodified MCsim code
}
```

convert a `ForeignCommand_t` to a `BusPacket`, which is processed according to the normal MCsim flow from there on.

Experiment Results

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 mapped to different banks.

As a first observed result, *the simulations follow through for both, the TDM and the FIFO arbiters*. This validates our specification, since MCsim's simulation stalls if timing constraints are not respected or incoming requests are not served at some point. The fact that we get MCsim to run with our algorithm without triggering any errors builds confidence that the CoqDRAM specification is correct. It is worth stating that we also modified our algorithms to be "wrong" to confirm that they *can* trigger errors if timing constraints or the protocol are not respect.

Table 7.1 compares the simulation output with other known DRAM controllers (AMC [20], ROC [118], and FR-FCFS [119]) – our results are highlighted in green. Note that, for the described setup, our arbiters provide a bandwidth that is comparable to that of the other real-time controller, AMC. As expected, the bandwidth is smaller than that of high-performance controllers (ROC and FRFCFS). Moreover, as one would expect, it can be seen that the high-performance controllers exhibit fluctuations on the maximum observed latency depend-

	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 (s)					
	11.13	21.18	0.41	0.85	0.96

Table 7.1: Simulation results for sequential and random traces.
MCsim setup: 4 Requestors, 1 Channel, 1 Rank, DDR3 2133N 2Gb_x8 device, Private Banks, In-order cores, 1000000 cycles.

ing on the trace format (41 to 25 and 23 to 25, respectively), while the real-time controllers offer constant latency, no matter the format of the input trace. Moreover, the maximum observed latency values are consistent with the ones presented in related work (c.f Figure 14 on [28]). On the downside, since our integration with MCsim relies on a complex and sub-optimal Haskell-C++ interaction, the simulation is considerably slower.

The FIFO `WAIT` parameter and the TDM `SL` and `SN` parameters used to obtain the results from Table 7.1 were calculated by solving the linear problem stipulated by the axioms in Listing 6.1 and 6.7. In other words, we choose the smallest `WAIT` and `SL` such that all axiomatic constraints are respected. Furthermore, the values in Table 7.1 are obtained with the same computer setup as the one described in Table 6.1, and the average simulation time is calculated taken the average of 100 executions.

Note that only two things impact the simulation time: 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.

Chapter 8

Improving Re-usability – A New Iteration of CoqDRAM

This chapter is aimed at presenting the features most recently added to CoqDRAM. The first version of CoqDRAM expected users to write algorithms based on the `Implementation_t` interface, described in Chapter 5. Although that was already a good start towards establishing a foundation for formalising DRAM scheduling algorithms, it did not emphasise the "framework" aspect, i.e., parts were not really re-usable.

In other words, it was expected that each new algorithm written by users managed all operations, from address mapping, refresh management, row-buffer policy management, command generation, request scheduling, and command scheduling. Then, for each new algorithm, the user was expected to prove every proof obligation described in Chapter 5. Clearly, the usability of such model is very limited. With that in mind, we improved the framework, focusing on the re-usability aspect.

In more detail, we propose a new iteration of CoqDRAM. The new approach consists in using *Bank Machines* to track the state of the DRAM device in order to know which commands *are allowed* to be sent beforehand. This includes keeping track of timing constraints, bus direction (read or write), and status of the row-buffer in each bank. This makes it possible to write schedulers as algorithms that manipulate a list of *ready* commands, which, as a consequence, makes it *impossible* to violate timing or functional correctness criteria. This design paradigm brings two additional advantages: 1) the proofs in `Trace_t`, which describe timing and functional correctness, do not have to be written again for each new implementation, and 2) algorithms optimising performance become easier to write, since algorithms can easily opt for a first-ready type of procedures, something that would have been hard to do without tracking the state of the DRAM device.

Furthermore, in this new iteration of CoqDRAM, we include refresh commands. As

briefly mentioned in Chapter 5, while exploring a different research path (described in Appendix A), we realised that refresh management was a crucial part of real *hardware* memory controller designs. Moreover, *refresh management* is an important algorithmic problem [105] for memory controllers, as the JEDEC standards allow a certain degree of flexibility regarding **when** to issue refresh commands. This flexibility can be used to optimise the average-case latency of memory requests, for example.

Finally, be mindful that this new version of the framework is compatible with the first, which means that it is built on top of what had been already done. This means that the algorithms described in Chapter 6 are still correct under the new framework – except for refresh compliance.

8.1 Modelling Refresh

According to the JEDEC standards, all banks must be idle for at least t_{RP} cycles before a refresh command is issued to the device. Typically, the memory controller achieves this by issuing a `PREA` command t_{RP} cycles before the `REF` is due, which pre-charges all banks simultaneously. The first step is thus to include these two commands in `Command_kind_t`, as shown in Listing 8.1.

Listing 8.1: Modelling the `PREA` and `REF` commands.

```
Inductive Command_kind_t :=
    | CRD  : Request_t → Command_kind_t
    | CRDA : Request_t → Command_kind_t
    | CWR  : Request_t → Command_kind_t
    | CWRA : Request_t → Command_kind_t
    | ACT  : Request_t → Command_kind_t
    | PRE  : Request_t → Command_kind_t
    | PREA : Command_kind_t (* new *)
    | REF  : Command_kind_t (* new *)
    | NOP  : Command_kind_t.
```

Next, we insert two timing constraints in `System_configuration`: t_{REFI} and t_{RFC}, as shown in Listing 8.2. Naturally, we also state two proof obligations restricting these constraints to be positive numbers.

The meaning of the constraint t_{RFC} is quite straightforward: it is the minimum time between a `REF` command and the next valid command issue to the device. In other words, the controller must wait *at least* t_{RFC} cycles after a `REF` before issuing another valid command.

Listing 8.2: Introducing two timing constraints related to refresh commands.

```
Class System_configuration := {
    ...
    (* Refresh related constraints *)
    T_RFC : nat;
    T_RFC_pos : T_RFC > 0;
    T_REFI : nat;
    T_REFI_pos : T_REFI > 0;
}.
```

Moreover, in general, according to the standard, "a `REF` command needs to be issued to the DDR4 device regularly every t_{REFI} interval." To, allow for improved efficiency in scheduling, some flexibility is allowed, which means that refresh commands can be postponed and issued in advance. However, the DDR4 JEDEC standards includes only a short section explaining exactly how this flexibility works and what are the timing constraints between different refresh commands. In addition, the text presenting is not perfectly clear. The timing diagrams are confusing as well and do not help much in resolving the ambiguities from the text.

In detail, the standard states that "a maximum of 8 refresh commands" can be postponed, meaning that **at not point in time more than a total of 8 refresh commands are allowed to be postponed**. Therefore, the upper-bound on the time elapsed between two consecutive refresh commands is $9 \times t_{REFI}$, which happens after having postponed 8 refresh commands in a row. In addition, a maximum of 8 refresh commands can be issued in advance (i.e., "pulled-in"). The advancing of refresh commands is further constrained by the statement that "at any given time, a maximum of 16 refresh commands can be issued within $2 \times t_{REFI}$".

From this description, taken directly from the DDR4 JEDEC standard, we can derive three rules that dictate the timing of refresh commands:

1. The maximum spacing between two consecutive refresh commands is $9 \times t_{REFI}$.

2. The amount of refresh commands issued during a time window of length $2 \times t_{REFI}$ cannot be superior to 16, i.e.,
   $\forall t \in \mathbb{N}, \mathbf{card}\,\{cmd \in Commands \mid isREF\, cmd \ \wedge\ (t \leq cmd.(CDate) \leq t + 2 \times t_{REFI}))\}$
   ≤ 16.

3. At any point in time, the amount of refresh commands issued during a time window of length $\forall x \in \mathbb{N}, x \times t_{REFI}$ cannot be inferior to $(x + 1) - 8$.

These three conditions, plus the t_{RFC} constraint, can be formalised in Coq as shown in Listing 8.3.

Listing 8.3: POs ensuring timing correctness (specific to refresh) in `Trace_t`.

```
(* ------------------ Refresh-related constraints ------------------------- *)
Cmds_T_RFC_ok : forall a b, a \in Commands → b \in Commands → isREF a → not isNOP b →
    Before a b → Apart_at_least a b T_RFC;

Cmds_T_REFI_max_dist_ok : forall a, a \in Commands → isREF a →
    exists b, b \in Commands ∧ isREF b ∧ b.(CDate) <= a.(CDate) + 9 * T_REFI;

Cmds_T_REFI_density1_ok : forall t,
    size ([seq cmd ← Commands |
    isREF cmd && (t <= cmd.(CDate)) && (cmd.(CDate) <= t + 2 * T_REFI)]) <= 16;

Cmds_T_REFI_density2_ok : forall x, (x+1) − 8 <
    size ([seq cmd ← Commands | isREF cmd && (cmd.(CDate) <= x * T_REFI)]);
```

Since t_{RFC} is a lower-bound, the `Cmds_T_RFC_ok` PO is state very similarly to the other timing POs from `Trace_t` from Chapter 5. Moreover, `Cmds_T_REFI_max_dist_ok` states that any `REF` in the trace must be followed by another refresh issued not more than $9 \times t_{REFI}$ cycles later. The following 2 proof obligations user the `filter` and `size` functions from `mathcomp.seq`.[1] The notation `[seq x ← s | C]` means that the sequence `s` is filtered in a way that only elements respecting the boolean predicate `C` are kept. The boolean predicate `C` is stated as an anonymous function of `x` – a name given to an arbitrary element of `s`. To the best of our knowledge, these POs constitute the first deduction-based formalisation/mechanisation of DRAM refresh timing constraints.

Listing 8.4: PO ensuring functional correctness (specific to refresh) in `Trace_t`.

```
(* Banks must be idle for at least T_RP cycles before a REF is issued *)
Cmds_REF_ok : forall ref, ref \in Commands → isREF ref →
    (exists prea, (prea \in Commands) &&
    (isPREA prea) && (Before prea ref) && (prea.(CDate) + T_RP <= ref.(CDate))) ∨
    (forall bank, bank \in All_banks → exists pre, (pre \in Commands) &&
    (Before pre ref) && (pre.(CDate) + T_RP <= ref.(CDate))) }.
```

Finally, it is also necessary to include the *protocol* constraint about refresh commands. Simply put, all banks must be idle for at least t_{RP} cycles before a `REF` is issued. We formalise

[1] https://math-comp.github.io/htmldoc/mathcomp.ssreflect.seq.html

this through an extra PO in `Trace_t`, as shown in Listing 8.4. In natural language, we say that for any arbitrary refresh command (`ref`) in the trace, there must exist a `PREA` command issued at least t_{RP} cycles before, or, for every bank, there must exist a corresponding `PRE` command issued at least t_{RP} cycles before the `REF` command.

8.2 The Interface Sub-Layer & the Bank Machines

In this section, we describe the framework features that allow users to write scheduling algorithms that re-use proofs about timing and protocol correctness. In other words, algorithms written using the functions described in the following do not have to manage timing or protocol constraints nor refresh operations. In Chapter 9, we describe a new scheduling algorithm developed using the proposed approach.

We begin by writing a set of definitions, which together form the *Interface Sub-Layer* – an implementation of `Implementation_t` (recall the framework architecture from Chapter 4). Like FIFO and TDM, described in Chapter 6, the sub-layer must define the functions `Init` and `Next`. Therefore, as it has been done for FIFO and TDM, we must also define the schedulers internal state beforehand (i.e. we must instantiate `Arbiter_configuration` by defining/instantiating `State_t`). Listing 8.5 shows the definition of `ARBITER_CFG`, an instance of `Arbiter_configuration` in which `State_t` is defined as `option ImplSubLayerState_t`. The reason why we use `option` is explained later in the text.

In addition, Listing 8.5 shows the definition of `ImplSubLayerState_t` and `Scheduler_InternalState`. Together, these definitions form a 3-layer state structure, in which the upper-most layer, `ARBITER_CFG`, is the one manipulated by `Default_arbitrate`. The second layer, defined by `ImplSubLayerState_t`, plays the same role as FIFO's `FIFO_state_t` and TDM's `TDM_state_t` – it defines the scheduler's internal state. Here, however, the internal state is yet another intermediate state abstraction containing three members:

1. `SystemState`, of type `SystemState_t` (whose definition is shown later), represents the state of the memory device. It contains the status of each bank, the bus direction, and counters used to keep track of timing constraints.

2. `CMap`, of type `ReqCmdMap_t` represents the "*command map*", a data structure holding the *commands* needed to service the requests present in the system. Since command also hold information about their originating requests, `CMap` can also be seen as a list of pending requests/commands.

3. `SchState`, of type `SchState_t` – which is an arbitrary `Type`. `SchState` is the third layer

Listing 8.5: Sub-Layer State & State Hierarchy.

```
(* Layer 3: specific to scheduling algorithms *)
Class SchedulerInternalState := mkSIS {
    SchState_t : Type
}.

(* Layer 2: the internal state - common for all scheduling algorithms *)
Record ImplSubLayerState_t := mkImplSubLayerState {
    (* The state of the DRAM *)
    SystemState : SystemState_t;
    (* The data structure holding requests *)
    CMap    : ReqCmdMap_t;
    (* The scheduler algorithm internal structure *)
    SchState   : SchState_t;
}.

(* Layer 1 : the states used to build the trace *)
Instance ARBITER_CFG : Arbiter_configuration := {
    State_t := option ImplSubLayerState_t;
}.
```

and is used to hold internal information for scheduling algorithms, such as additional counters and queues, and whatever may be needed to implement a given algorithm.

Figure 8.1 provides a visual scheme for understanding the hierarchy of states using this new interface. The figure shows the first two states created following a call to `Default arbitrate n`, where $n > 1$. Note how the functions `Init_SL_state` and `Next_SL_state` (whose definition is yet to be discussed) implement `Init` and `Next`, respectively. The next state function, `Next_SL_state`, produces a new command each cycle and a new internal state (as it was with FIFO and TDM). Remember that, in the figure, the internal state is said to be of type `State_t`, but that type is instantiated to be `option ImplSubLayerState_t` by `ARBITER_CFG` (in Listing 8.5).

Furthermore, the reason why `State_t` is instantiated using an `option` type is that, as we will see later in the text, the functions in the bank machines are written in a functional ML-programming style: we write partial functions that *may fail* if hypotheses do not hold (i.e., if something goes wrong). For example, if we try to update the state of the system with a command that is invalid (in the sense that it violates a timing constraint, for instance), then an *update* function may produce an invalid state. We prefer this kind of design pattern rather than using hypotheses inside of programs to invalidate forbidden behaviour. Although the technical details about such decision lies outside of the scope of this dissertation, it is simpler

to manipulate programs without proofs, both in terms of proofs and performing computation (which is useful for testing). The design choice is thus to keep a clean separation between programs and proofs.

Before explaining the behaviour of `Init_SL_state` and `Next_SL_state` in detail, we must first introduce `Scheduler_t` – the interface for specifying new scheduling algorithms according to this new approach. Be mindful that `Scheduler_t` is ultimately what is exposed to the framework user. The definition of `Scheduler_t` can be seen in Listing 8.6. The `Schedule` function takes three arguments: 1) the list of ready commands (of type `ReqCmdMap_t`), 2) the state of the DRAM device (of type `SystemState_t`), and 3) the scheduler's internal state (of type `SchState_t`). It produces a new command to be issued to the DRAM device (of type `Command_kind_t`). `Scheduled_Empty` and `Schedule_In` are proof obligations which together guarantee that the scheduled command comes from the list of ready commands, or is a `NOP`, if the list is empty or the algorithm decides to issue a `NOP` (this could happen, since an algorithm might choose to not issue a ready-command instantaneously). Moreover, `InitSchState` represents the initial `SchState_t`; and `UpdateSchState` creates a new state (of type `SchState_t`), taking as arguments the list of *all* pending commands (and not just ready

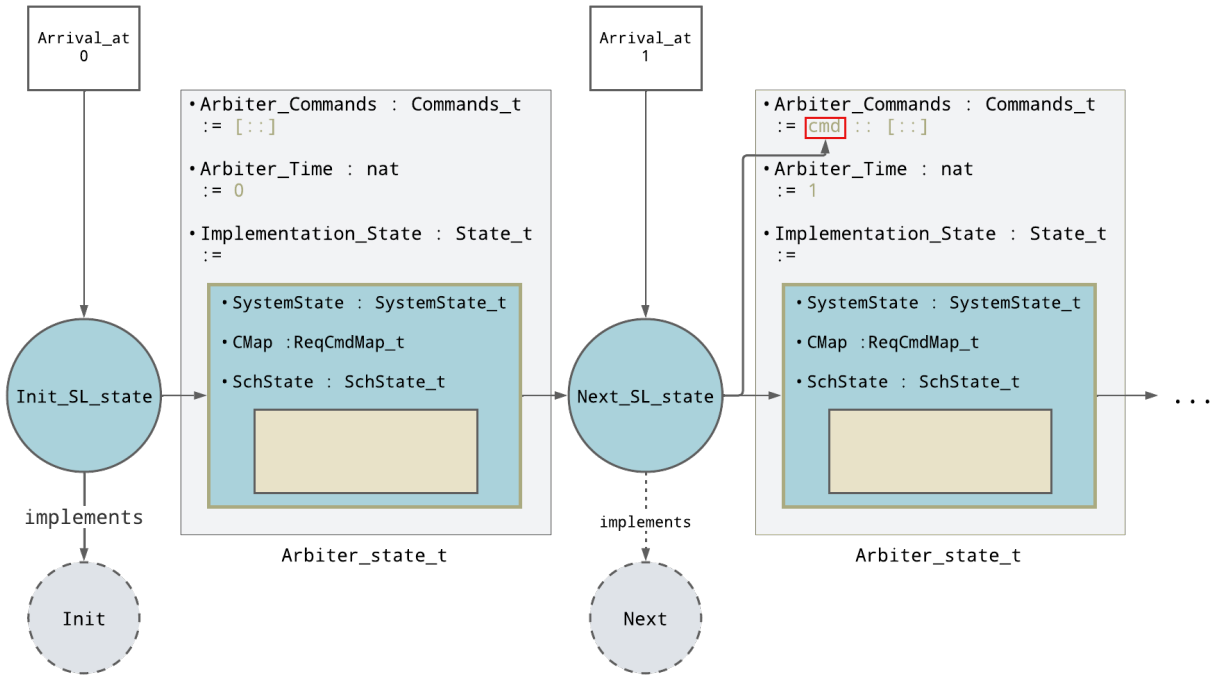


Figure 8.1: State hierarchy using the Interface Sub-Layer.
Legend: Layer 1 – states manipulated by `Default_arbitrate` and used to build the command trace. Layer 2 – states manipulated by the interface sub-layer, agnostic to the implemented algorithm. Layer 3 – states manipulated by the scheduling algorithm.

Listing 8.6: Definition of `Scheduler_t`.

```
Class Scheduler_t := mkScheduler {

    (* Scheduling function *)
    Schedule : ReqCmdMap_t → SystemState_t → SchState_t → Command_kind_t;

    (* If there is no commands in the system (which can only happen if there is no
       request in the system), produce a NOP *)
    Schedule_Empty : forall m SYS_ST SCH_ST, m = [] → Schedule m SYS_ST SCH_ST = NOP;

    (* Scheduled command must come from the list of commands *)
    Schedule_In : forall m SYS_ST SCH_ST, let sch_cmd := Schedule m SYS_ST SCH_ST in
  (sch_cmd = NOP) ∨ (sch_cmd \in m);

    InitSchState : SchState_t;

    UpdateSchState : Command_kind_t → ReqCmdMap_t → SystemState_t → SchState_t →
       SchState_t;
}.
```

commands), the state of the DRAM device, and the scheduler's current internal state. The fact that `UpdateSchState` takes as argument the list of all pending commands is important, since it needs to have visibility over all requests in the system, and not just those that ready commands associated with them.

Having introduced the necessary abstractions, we can discuss `Next_SL_state`, presented in Algorithm 1 as a generic procedure. The core idea is to "*map*" arriving *and* outstanding (i.e., requests already in the system) to the next DRAM command needed to service such request at each clock cycle, given a certain page policy (Lines 7 and 8). Commands related to refresh management might also be included in the map when they are due (Line 9). The precise refresh management mechanism is explained in more detail later in the text. Moreover, this "map" is actually implemented as a list – also called *command map* throughout this text.[2] Then, commands that do not satisfy *all* timing constraints are filtered-out of the list (Line 11), being replaced by `NOP` commands, and the resulting list is handed to the scheduler (Line 12).

Next, the scheduler, following its own policy, chooses a command from the list of ready/-

[2]The ideal data structure to hold such command mapping would be a hash map, in which keys are requests and elements are commands. The implementation of hash maps in Coq's standard library, however, is dependently typed and overall hard to work with. For that reason, we work with lists instead. Since commands (of type `Command_kind_t`) "carry" their originating request with them, we can still access information about requests over a list of commands.

Algorithm 1 `Next_SL_state`.

Inputs: `R : Requests_t`, `ST : option ImplSubLayerState_t`
Outputs: `ST_new : option ImplSubLayerState_t`, `cmd : Command_kind_t`

```
1: if ST is valid then
2:
3:     SS ← ST.(SystemState)                    ▷ Access the fields of ImplSubLayerState_t
4:     SCH ← ST.(SchState)
5:     cmd_map ← ST.(CMap)
6:
7:     cmd_map ← map_running(cmd_map)           ▷ Map existing requests to commands
8:     cmd_map ← map_arriving(cmd_map)          ▷ Map arriving requests to commands
9:     cmd_map ← map_REF(cmd_map)                                  ▷ Refresh management
10:
11:    ready_cmds ← filter_non_ready(cmd_map)   ▷ Filter out non-ready commands
12:    sch_cmd ← Schedule(ready_cmds,SS,SCH)    ▷ Choose a command to be issued
13:
14:    if isCAS sch_cmd then
15:        cmd_map ← remove(sch_cmd,cmd_map)    ▷ Request is done, remove from list
16:
17:    SS_new ← SystemUpdate(sch_cmd,SS)                ▷ Update the state of the DRAM
18:
19:    if SS_new is valid then
20:        SCH_new ← UpdateSchState(sch_cmd,cmd_map,SS,SCH)
21:        ST_new ← (Some SS_new,cmd_map,SCH_new)   ▷ Create a new ImplSubLayerState_t
22:    else(None,NOP)
23: else(None,NOP)
```

valid commands. If the chosen command is a `CAS`, then it means that its associated request has completed, and the entry is removed from the list (Line 15). Finally, the system is updated. Updating the system means, first, updating the status of the DRAM device (`SystemState_t`) considering the effects of the command that has been chosen to be issued (Line 17), and second, updating the internal state of the scheduler (Line 19). The new state, of type `ImplSubLayerState_t`, is a 3-tuple (Line 20) made of the new DRAM state (`SS_new`), the updated command map (`cmd_map`), and the updated scheduler state, `SCH_new`.

Moreover, note that the algorithm tests if `ST` is valid (Line 1). If that is not the case, it produces another invalid state and issue a `NOP` (Line 22). It also tests if the new DRAM state generated after updating is valid (Line 18). If that is not the case, again, it generates an invalid state and issue a `NOP` (Line 21). This means that the algorithm can never recover from an invalid state. This makes sense, since we design a system that can never reach an invalid state. In fact, most proofs using this machinery rely on the fact that an invalid state cannot be reached (more on that soon).

As an example, Figure 8.2 depicts a scenario consisting of two subsequent executions of `Next_SL_State`. The system depicted in the figure includes requests targeting three banks: B_I, B_{II}, and B_{III}. Consider, in the first execution (at cycle n), that the last command issued to the DRAM device (at cycle $n-1$) was a `PRE` to bank B_{II}. Consider also that two requests are currently in the system: $R_1(WR_M)$ – a write request targeting bank B_I, and $R_2(RD_H)$ – a read request targeting bank B_{II}. R_1 is a row-buffer miss, meaning that, when it arrived, the row loaded in B_I's row-buffer was not R_1's target row. R_2 is a row-buffer hit. At instant n, an additional request arrives, $R_3(RD_H)$, a read request targeting bank B_I. At the time it arrives, R_3 is a row-buffer hit, meaning that B_I contais R_I's targeted row loaded in its row-buffer.

Note, in the figure, that `ST.(CMap)` initially contains two commands: since R_1 was a row-buffer miss when it arrived, *assuming that no other request interfered with its bank*, `ST.(CMap)` maps R_1 to a `PRE`, i.e., `PRE` is the next command needed to service R_1. Similarly, since R_2 was a row-buffer hit when it arrived, and again, assuming that no other request interfered with its bank in the meanwhile, `ST.(CMap)` maps a R_2 to a `CRD`, i.e., `CRD` is the next command needed to service R_2.

The first function used in the algorithm is `map_running`, which calls the command generation function on the elements of `ST.(CMap)`. Here, we assume an open-page command generation policy. Since B_{II} has just been precharged at $n-1$, R_2's `CRD` is no longer valid, which means that R2's must now issue an `ACT`. In other words, it can be said that another request (not present in the figure) interfered with R_2, making it regress. Next, `map_arriving` creates a new entry for R_3 – a `CRD`, since R_3's row is open at n. At this point, `map_REF` does

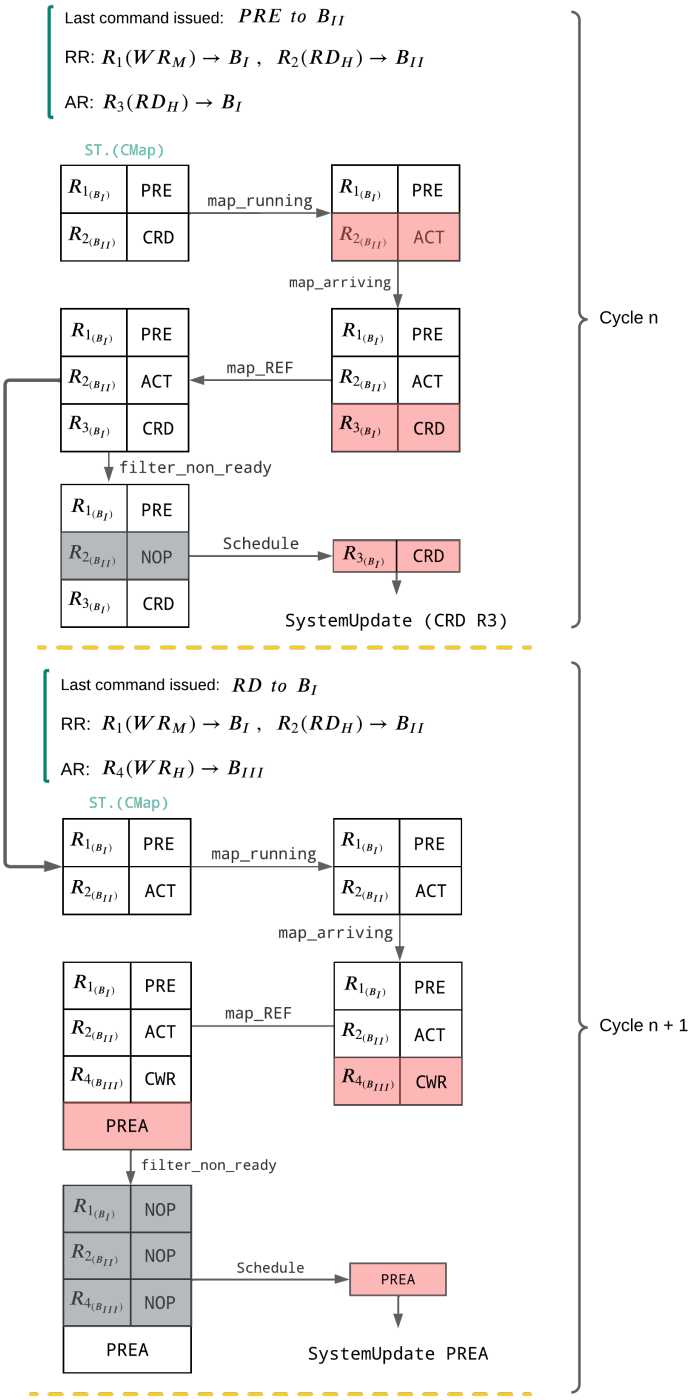


Figure 8.2: Example of execution of `Next_SL_state`. Arbitrary/unknown scheduling algorithm, open-page policy. `RR` is the list of pending requests and `AR` is the list of arriving requests at a given instant.

not insert any refresh-related commands. Afterwards, `filter_non_ready` creates a copy of the map, replacing R_2's `ACT` by a `NOP` – the reason being that since a `PRE` to B_{II} has just been issued at $n-1$, an `ACT` to B_{II} has to wait at least t_{RC} cycles.

The scheduler must then choose a command in the list [`PRE R1`; `CRD R3`]. Here, the scheduler employs its scheduling policy, which is unknown in this example. Consider that the scheduler chooses to issue R_3's `CRD` (certain scheduling policies indeed work by prioritising `CAS` commands over any other type of command). Since the scheduled command is a `CAS`, it is immediately removed from the list, and the system is updated considering the effect of that command on the DRAM device.

At cycle $n+1$, the same two requests R_1 and R_2 are still pending, and yet another request R_4 arrives – a write-hit request to B_{III}. The algorithm goes through the same series of steps described above: first, `map_running` does not change `ST`.(`CMap`), since neither R_1's nor R_2's bank state has been affected by the last issued command, R_3's `CRD`. Next, `map_arriving` creates a new entry for R_4, a `CWR` – since R_4 is a row-buffer hit.

This time, however, `map_REF` inserts a `PREA` in the map. Here, it is important to emphasise that the algorithm cannot violate the refresh timing constraints, i.e., `PREA` and `REF` commands are only inserted in the map when they can be immediately scheduled, and other commands take the urgency of refresh operations into consideration. In other words, if scheduling a given command would violate the timing constraints of a `PREA` or a `REF`, then such a command is also considered not ready. Note that, in the example from Figure 8.2, according to this rule, R_3's `CRD` would theoretically not have been allowed in the first place, since a `PREA` was due at $n+1$. For the sake of simplicity in this example, we ignore the timing constraint between a `CRD` and a `PREA` and consider that both commands are valid.

Finally, `filter_non_ready` invalidates all commands except the `PREA`: R_1's `PRE` is not ready, since a `CRD` to the same bank has been issued at n; $R_2's$ `ACT` is not ready, since a `PRE` to B_{II} was issued at $n-1$ and t_{RC} has not yet elapsed; and R_4's `CWR` is also not ready; since thre must be at least t_{RTW} cycles between any `CRD` and `CWR`, and a `CRD` has been issued at n. The scheduler only has one option then: `PREA`, which is scheduled and used to update the system.

While exploring the definition of every function used in Algorithm 1 would take a huge amount of space, it is worth looking at the definition of `filter_non_ready`, the function which determines if commands are ready to be issued or not. Listing 8.7 shows the definition of `filter_non_ready`. Straightforwardly, we test the predicate "`iscmdOK`" in each element of a given list of commands, `cmdmap`. If `iscmdOK` is true, the command is mapped to itself, otherwise, it is replaced by a `NOP`.

Now, to understand what happens inside of `iscmdOK`, we must first look at the definition

Listing 8.7: Filtering non-ready commands.

```
Definition filter_non_ready (cmdmap : ReqCmdMap_t) (SS : SystemState_t) :=
    seq.map (fun cmd ⇒ if (iscmdOK cmd SS) then cmd else NOP) cmdmap.
```

of the type `SystemState_t`, shown in Listing 8.8. `SystemState_t`, used to keep track of the state of the memory device, is a record with three fields:

Listing 8.8: `SystemState_t`, used to keep track of the state of the memory device.

```
Inductive BankState_t :=
    | IDLE : LocalCounters → BankState_t
    | ACTIVE : Row_t → LocalCounters → BankState_t.

Definition BankStates_t : Set := {l : seq.seq BankState_t | seq.size l = BANKS }.

Record SystemState_t := mkSystemState {
    Banks : BankStates_t;
    SysCounters : GlobalCounters;
    Busdir : Busdir_t
}.
```

1. `Banks`, of type `BankStates_t`, is a list containing the state of all banks in the system. The list is modelled as a *sigma type*, meaning that it must necessarily have the length `BANKS` (the parameter defined in `System_configuration`). The type of elements in the list is `BankState_t` – a type modelling the state of a single bank. It is defined as an inductive datatype with two constructors, `IDLE` and `ACTIVE`, according to Figure 2.6. Moreover, both constructors also carry a set of counters to keep track of **intra-bank** timing constraints, `LocalCounters`. In more detail, `LocalCounters` is a record containing six counters, which are all of type `nat`. These counters work according to the same principle – they get reset to 0 when their related command is issued to their respective bank, and in all other cases, they are incremented by one:

    - `cACTsb` accounts for `ACT` commands issued to a given bank;
    - `cPRE` accounts for `PRE` commands issued to a given bank;
    - `cRDsb` accounts for `CRD` commands issued to a given bank;
    - `cRDA` accounts for `CRDA` commands issued to a given bank;
    - `cWRsb` accounts for `CWR` commands issued to a given bank;

- `cWRA` accounts for `CWRA` commands issued to a given bank.

Finally, if the bank is `ACTIVE`, there is also an associated `Row_t`, which is the row currently loaded in the row-buffer.

2. `SysCounters`, of type `GlobalCounters`, is a record containing counters used to keep track of **inter-bank** timing constraints. Although the precise fields of the record are not shown here, their working principle is very similar to `LocalCounters`, except that there is only one instance of `SysCounters` in the entire system.

3. `Busdir`, of type `Busdir_t`, is used to keep track of the direction of the data bus (read or write). `Busdir_t` is an inductive type with two constructors, and its definition is omitted for brevity.

We can now analyse the (incomplete) definition of `iscmdOK`, shown in Listing 8.9. The function takes a command (`cmd`) and a system state (`SS`) as arguments, and produces a `bool` as result. The goal of the function is to determine if, given the state of the memory, `SS`, `cmd` can be immediately issued, i.e., `cmd` is valid.

Lets focus on read commands for now, other commands are handled in a similar way and briefly discussed later. Start by noting how, at Line 3, `CRD` and `CRDA` are always associated with a request, `req`. Then, from Lines 4 to 7, we give names to `req`'s bank and bank-group: `bk` and `bg`, respectively. The state of `bk` is labelled `BS`. Then, at Line 10, a pattern matching is performed on `BS`. `BS` being `IDLE` consists of a violation, since `CAS` commands are not allowed on an IDLE bank (Line 8). More precisely, an `ACT` is required to load a row in the row-buffer before a `CRD` or a `CRDA` can be issued.

If `BS` is `ACTIVE` (Line 9), we must verify whether all protocol and timing constraints are satisfied. First we test to see if `req` is a row-buffer hit or miss. In the latter case, another violation occurs, since we must read from the correct row (Line 10 and 35). If it is a hit, we can proceed with the checks, which will depend on the direction of the data bus. If the direction of the bus is read (indicated by `BRD`), then, the following constraints must be satisfied:

- At least t_{CCD_l} cycles must have elapsed since the last `CRD` command to the same bankgroup, to either the same of different banks, was issued (Lines 15 and 16);

- At least t_{CCD_s} cycles must have elapsed since the last `CRD` command to a different bankgroup was issued (Line 17);

- At least t_{RCD} cycles must have elapsed since the last `ACT` command to `bk` was issued (Line 18);

Listing 8.9: Definition of `iscmdOK`.

```
Definition iscmdOK (cmd : Command_kind_t) (SS : SystemState_t) : bool :=
    match cmd with
    | CRD req | CRDA req ⇒
          let bk := (Bank_to_nat req.(Address).(Bank)) in (* req's bank *)
          let bg := Bankgroup_to_nat req.(Address).(Bankgroup) in (* req's BG *)
          let BS := seq.nth def_BKS (proj1_sig SS.(Banks)) bk in (* the state of req's
              bank *)
        match BS with
        | IDLE _ ⇒ false (* Violation: CRD not ok on an IDLE bank *)
        | ACTIVE row lc ⇒
            if (req.(Address).(Row) = row) then ( (* row hit *)
            match SS.(Busdir) with
            | BRD ⇒
                let lastRD_sbg := seq.nth 0 (proj1_sig SS.(SysCounters).(cRDbgs)) bg in
                let lastRD := SS.(SysCounters).(cRDdb) in
                (T_CCD_l <= lc.(cRDsb)) && (* RD-to-RD same bank and bankgroup *)
                (T_CCD_l <= lastRD_sbg) && (* RD-to-RD dif. bks (but same BGs) *)
                (T_CCD_s <= lastRD) && (* RD-to_RD different BG *)
                (T_RCD <= lc.(cACTsb)) && (* ACT to RD *)
                 (* Can't issue a CRD if there's an incoming REFRESH cycle. *)
                (SS.(SysCounters).(cREF) <= T_REFI − T_RP − T_RTP) &&
                (* Can't issue a CRD after a REF for T_RFC cycles *)
                (T_RFC <= SS.(SysCounters).(cREF))
            | BWR ⇒
                let lastRD_sbg := seq.nth 0 (proj1_sig SS.(SysCounters).(cRDbgs)) bg in
                let lastRD := SS.(SysCounters).(cRDdb) in
                (T_WL + T_BURST + T_WTR_l <= lc.(cWRsb)) && (* RD-to-WR same bank (
                    also same BG) *)
                (T_WL + T_BURST + T_WTR_l <= lastRD_sbg) && (* RD-to-WR different
                    banks (but same BG) *)
                (T_WL + T_BURST + T_WTR_s <= lastRD) && (* RD-to-WR different
                    bankgroups *)
                (T_RCD <= lc.(cACTsb)) && (* ACT to WR *)
                (* Can't issue a CWR if there's an incoming REFRESH cycle. *)
                (SS.(SysCounters).(cREF) <= T_REFI − T_RP − T_RTP) &&
                (* Can't issue a CWR after a REF for T_RFC cycles *)
                (T_RFC <= SS.(SysCounters).(cREF))
            end)
            else false (* CRD not ok when the active row is not the request's row *)
        end
    | CWR req | CWRA req ⇒ (* continues ... *)
```

- The issue date of the *next* `REF` command must be at least $t_{REFI} - t_{RP} - t_{RTP}$ cycles away (Line 20);
- At least t_{RFC} cycles must have elapsed since the last `REF` command was issued (Line 22).

Anecdotally, the code from Lines 13 and 14 (as well as 24 and 25) are responsible for assigning names to counters, using functions to access global/system counters in `SS`. Moreover, if the direction of the bus is `BWR` (meaning that the last successful transaction was a write), a different set of constraints apply, which are not shown here.

Note how, for all timing constraints, we use counters that are local to banks and counters that are relevant for all banks. Looking back at Algorithm 1, these counters are incremented whenever the system is updated after issuing a new command. This process takes place in the function `SystemUpdate` – whose type signature is shown in Listing 8.10. `SystemUpdate` expects two arguments: 1) `cmd`, of type `Command_kind_t`, represents the command issued to the memory device; and 2) `SS`, of type `option SystemState_t`, represents the current state of the system. The function produces another `option SystemState_t` as output.

As discussed previously, the fact that we use an `option` type means that the current system state can be valid or not. If it is not valid (i.e., when `SS` is equal to `None`, one of the constructors of `option`), then `SystemUpdate` produces another invalid state. If `SS` is valid, then the `SystemUpdate` may or may not produce a valid state, depending on `cmd` and `SS`.

Listing 8.10: Type of `SystemUpdate`, a function to update the state of the DRAM device based on the effects of issuing a given command.

```
Definition SystemUpdate
    (* The issued command chosen by the scheduler function *)
    (cmd : Command_kind_t)
    (* The current system state *)
    (SS  : option SystemState_t) : option SystemState_t.
```

For conciseness, the implementation of `SystemUpdate` is not discussed, but in summary, it updates everything in a given `SystemState_t` when a command is issued. If a `CRD` is issued, for instance, then two counters are reset to zero: `cRDsb`, which keeps track of `CRD` commands to a given bank, and `cRDbgs`, which keeps track of `CRD` commands to a given bank-group. In addition, the bus direction might also be updated: if it was a `BWR`, then it changes to a `BRD` and remains unchanged otherwise.

As another example, if `cmd` is `CRD r` in a call to `SystemUpdate`, where `r` is the request associated with the `CRD`, then we look at the status of `r`'s bank: if `r`'s bank is `IDLE`, that is a violation, since a `CRD` cannot be issued to a bank without any active row. As a result,

`SystemUpdate` produces a `None`, signalling that a violation has occurred. Again, this style of programming allows faulty behaviours by construction, as opposed to programming with dependent types. On a later stage, we prove that a faulty behaviour cannot occur and the system never falls into a faulty state – from which it could never recover.

8.2.1 Unified Proofs for Timing Constraints

One of the advantages of the bank machine approach is that proofs about timing constraints, such as t_{RCD}, are done just once and are agnostic to the actual scheduling algorithm. In the following, as an example, the proof strategy for the `Cmds_T_RCD_ok` PO is discussed. The discussion involves a more "high-level" proof strategy rather than a low-level proof script analysis, although the main insights of proof scripts are sometimes indicated. For convenience, the top-level theorem is reproduced below.

```
Theorem Cmds_T_RCD_ok t a b:
    a \in (Default_arbitrate t).(Arbiter_Commands) →
    b \in (Default_arbitrate t).(Arbiter_Commands) →
    isACT a → isCAS b → (get_bank a = get_bank b) → Before a b →
    a.(CDate) + T_RCD <= b.(CDate).
```

The proof is carried out in 4 steps. At a high-level, these steps are described below:

1. Given that commands `a` and `b` are in the trace, `iscmdOK` *must have been true* the cycle before these commands were issued, i.e., at `a.(CDate)` $-$ 1 and `b.(CDate)` $-$ 1;

2. Knowing that `iscmdOK` is true at `a.(CDate)` $-$ 1 and that `a` is issued at `a.(CDate)`, we can derive the information that the intra-bank `ACT` counter, `cACTsb`, is reset to 0 the following cycle by `SystemUpdate`;

3. Similarly, knowing that `iscmdOK` is true at `b.(CDate)` $-$ 1 and that commnad `b` is issued at `b.(CDate)`, we can derive the information that the intra-bank `ACT` counter, `cACTsb`, is at least grater then (or equal to) t_{RCD} at `b.(CDate)` $-$ 1, from the definition of `iscmdOK`;

4. Finally, knowing that `a` and `b` target the same bank, that `a` happens before `b` in the trace, that the counter `cACTsb` was 0 at `a.(CDate)` (Step 2), and that it was greater than or equal to t_{RCD} at `b.(CDate)` (Step 3), we use an auxiliary lemma stating that counters increases monotonically (unless a reset happens) to conclude that `a.(CDate)` $+t_{RCD} \leq$ `b.(CDate)`.

These steps are presented in greater detail below:

Step 1. If a non-NOP command, cmd, exists in the trace at an arbitrary instant t, then iscmdOK is true at instant cmd.(CDate) − 1. Intuitively, this makes sense, since according to Algorithm 1, a non-NOP command can only be issued when iscmdOK is satisfied for such command in the previous cycle. The proof of Lemma 1 is further organised into multiple sub-steps – each stated in Coq as a Lemma.

Lemma 1.1 (valid_state_when_cmd_in_trace cmd). First, we prove that, if cmd is in the trace at t, the arbiter state at cmd.(CDate) must be valid. The proof follows from the fact that an invalid state would have generated a NOP command, and since we know that cmd is not a NOP, the state of the arbiter when the command was issued could only have been valid. This is described in Coq as the Lemma valid_state_when_cmd_in_trace, shown in the snippet below. The proof proceeds by induction on cmd.(CDate). In the induction step, a case analysis on the arbiter state is required, along with smart reduction tactics. Note that the validity of states is tested with the Coq function isSome, which, for a given term of an option type, returns true if the term was built by the constructor Some, and false if it was built by None.

```
Lemma valid_state_when_cmd_in_trace cmd t :
    cmd \in (Default_arbitrate t).(Arbiter_Commands) → not (isNOP cmd) →
    isSome (Default_arbitrate cmd.(CDate)).(Implementation_State).
```

The proof of valid_state_when_cmd_in_trace depends on an auxiliary Lemma called SystemUpdate_valid, which states that the SystemUpdate function **must** produce a valid state when fed with a valid state and with a command coming from the filtered list filtered_map. Recall from Listing 8.7 the definition of filter_non_ready: either commands in the resulting list respect the iscmdOK predicate or are replaced by NOP commands. In the latter case, NOPs are trivially validated by iscmdOK, which means that SystemUpdate cannot produce an invalid state.

```
Lemma SystemUpdate_valid n IS map Sys_S Sch_S :
    (Default_arbitrate n).(Implementation_State) = Some IS →
    let filtered_map := filter_non_ready map Sys_S in
    isSome (SystemUpdate (SCH.(Schedule) filtered_map Sys_S Sch_S) (Some Sys_S)).
```

Lemma 1.2 (previous_state_is_valid). If the state at instant n + 1 is valid, then the state at instant n must have been valid. This follows from the fact that the system can never recover from an invalid state; therefore, a valid state can only be preceded by another valid state. This is stated in Coq as Lemma previous_state_is_valid.

```
Lemma previous_state_is_valid n IS:
    (Default_arbitrate n.+1).(Implementation_State) = Some IS →
    exists IS', (Default_arbitrate n).(Implementation_State) = Some IS'.
```

Lemma 1.3 (iscmdOK_true_between_valid_states). If both the state at instant n + 1 and n are valid, then the command issued by the scheduling function at instant n + 1 satisfies the predicate iscmdOK at n. The Coq formalisation in shown in the snippet below. In the code, the state at instant n + 1 is given the name IS_cur, and the state at instant n is given the name IS_prev – for "current" and "previous", respectively. Note that cmd is the command coming from SCH.(Schedule) – the arbitrary scheduling function, where SCH is an abstract instance of the Scheduler class (see Listing 8.6).

```
Lemma iscmdOK_true_between_valid_states n IS_prev IS_cur map cmd:
    (Default_arbitrate n).(Implementation_State) = Some IS_prev →
    (Default_arbitrate n.+1).(Implementation_State) = Some IS_cur →
    (* The system state at n *)
    let SS := IS_prev.(SystemState) in
    (* The internal scheduler state at n *)
    let SCH_ST := IS_prev.(SchState) in
    (* The command issued at t + 1 by the arbitrary scheduling function *)
    cmd = (mkCmd n.+1 (SCH.(Schedule) (replace_nonrdy_cmds map SS) SS SCH_ST)) →
    iscmdOK cmd.(CKind) IS_prev.(SystemState).
```

Step 1 follows from Lemmas 1.1, 1.2, 1.3. Together, theses lemmas allow us to go from the top level hypothesis:

▷ `H : forall t, a in (Default_arbitrate t).(Arbiter_Commands),`

to the following conclusion:

▷ `iscmdOK a.(CKind) IS_prev,`

where IS_prev is the state of the arbiter at a.(CDate) − 1.

Step 1 can thus be applied in commands a and b from Cmds_T_RCD_ok, knowing that both a and b are respectively ACT and CAS commands.

Step 2 (cACTsb_reset). If iscmdOK (ACT r) SS holds, where r is an arbitrary request and SS is an arbitrary system state (of type SytemState_t), then updating SS with ACT r results in another system state on which the *intra-bank ACT counter* value is 0.

To better understand this step, bear in mind that by "ACT counter", we mean the *intra-bank* counter which is reset whenever an ACT command is issued in a bank. In all other cycles

```
Lemma cACTsb_reset : forall (r : Request_t) (SS : SystemState_t),
    iscmdOK (ACT r) SS →
    (* SS' is the state produced by SystemUpdate *)
    let SS' := SystemUpdate (ACT r) (Some SS) in
    (* get_cACTsb_ retrieves the counter cACTsb of a given bank from SS' *)
    get_cACTsb_ SS' r.(Address).(Bank) = Some 0.
```

that an ACT is not issued, it is incremented. This counter is named cACTsb in the code and is part of the LocalCounters record, which can be seen in Listing 8.8 as part of BankState_t. The Coq formalisation of Step 2 can be seen in the code snippet above.

Note that syntactically, SS' – the state produced by SystemUpdate – is of type option SystemState_t. But logically, SS' cannot be invalid, since that would contradict the hypothesis iscmdOK. Moreover, get_cACT_sb is a function that retrieves the cACTsb counter from an arbitrary system state for a given bank. This function is written in the partial-function programming style, meaning that it expects an option SystemState_t as parameter. In the case it receives an invalid state, it returns None.

Step 2 allows us to go from the conclusion of Step 1, iscmdOK a.(CKind), to the conclusion that the value of cACTsb, for the bank associated with a's request, is zero at instant a.(CDate).

Step 3 (cACTsb_gt_TRCD). If iscmdOK cmd SS holds, where cmd is any CAS command (i.e., a CRD, CRDA, CWR, or CWRA) associated with a request r and SS is an arbitrary system state; then the intra-bank ACT counter for r's bank must be greater than or equal to t_{RCD} at state SS. The Coq formalisation of Step 3 can be seen below.

```
1  Lemma cACTsb_gt_TRCD : forall (t : nat) (IS : ImplSubLayerState_t) cmd,
2      (Default_arbitrate t).(Implementation_State) = Some IS →
3      isCAS cmd → let r := get_req cmd in
4      match r with
5      | None ⇒ False
6      | Some r ⇒ (* r is the request associated with cmd *)
7          (* The iscmdOK hypothesis *)
8          iscmdOK cmd.(CKind) (IS.(SystemState)) →
9          (* Accessing the state of r's bank *)
10         let bk := Bank_to_nat r.(Address).(Bank) in
11         let BS := seq.nth def_BKS (proj1_sig IS.(SystemState).(Banks)) bk in
12         (match BS with
13         | IDLE _ ⇒ False (* contradicts iscmdOK *)
14         | ACTIVE _ lc ⇒ is_true (T_RCD <= lc.(cACTsb))
15         end) end.
```

Unfortunately, different than `cACTsb_reset`, the formalisation of Step 3 requires some other syntactic constructs used to access record fields and deal with partial functions that make the statement less readable. Nonetheless, it should be clear that `T_RCD <= lc.(cACTsb)` is the conclusion (Line 14), where `lc` is the set of local counters for `r`'s bank, `r` being the request associated to `cmd`.

The fact that `cACTsb` is greater than t_{RCD} comes from the definition of `iscmdOK`: a `CAS` is only allowed to be issued whenever at least t_{RCD} cycles have elapsed since the last `ACT` to the same bank has been issued. The reader is encouraged to visit Listing 8.9 again, where this condition is stated at Line 18.

Similar to Step 2, Step 3 allows us to go from the conclusion of Step 1, `iscmdOK b.(CKind)`, to the conclusion that value of `cACTsb`, for the bank associated with `b`'s request, is greater than or equal to t_{RCD} at instant `b.(CDate)` − 1.

Step 4 (`counter_monotonic`). If, for a given bank `bk`, `cACTsb` is equal to an arbitrary `x` at an instant `t`, then `forall t'` such that `t' > t` and `cACTsb` is greater than or equal to some `x + d`, then `t'` is greater than or equal to `t + d`. The Coq formalisation of Step 4 is shown below.

```
Lemma counter_monotonic (bk : Bank_t) (t t' : nat) (x d : Counter_t) IS IS' :
    t < t' → d > 0 →
    (Default_arbitrate t).(Implementation_State) = Some IS →
    (Default_arbitrate t').(Implementation_State) = Some IS' →
    let SS := IS.(SystemState) in (* The system state at t *)
    let SS' := IS'.(SystemState) in (* The system state at t' *)
    (* First condition on the counter - satisfied by Step 2 *)
    (match get_cACTsb_ (Some SS) bk with
        | None ⇒ False
        | Some z ⇒ z = x
    end) →
    (* Second condition on the counter - satisfied by Step 3 *)
    (match get_cACTsb_ (Some SS') bk with
        | None ⇒ False
        | Some y ⇒ is_true (y >= (x + d))
    end) → t' >= t + d. (* Conclusion *)
```

In natural language, Step 4 says that, if the counter at `t` was incremented by at least an arbitrary non-zero value `d`, then at least `d` clock cycles must have elapsed since `t`. This is because, excluding the reset condition of the counter, its value increases monotonically, which allows us to conclude that a counter increment implies an equal time increment.

Step 4 allows us to connect the results from Steps 2 and 3, which contain information

about the value of `cACTsb`, to the issue date of commands `a` and `b`. Applying (through the Coq tactic `apply`) Step 4 on `Cmds_T_RCD_ok` solves the goal, where `t'` is `b.(CDate)` and `t` is `a.(CDate)`. The two conditions on the values of the counter are satisfied by Steps 2 and 3, which means that `counter_monotic` is used with `x` $= 0$ and `d` $=$ `T_RCD`. Note also that `bk` is the bank from the requests associated to `a` and `b` – which, from the hypothesis `SameBank a b` in `Cmds_T_RCD_ok`, happens to be the same.

Finally, using `counter_monotonic` with `t` $:=$ `a.(CDate)`, `t'` $:=$ `b.(CDate)` $- 1$, `d` $:=$ `T_RCD`, and `x` $:= 0$, we can prove that `a.(CDate)` $+ t_{RCD} \leq$ `b.(CDate)`.

8.3 Re-implementing FIFO

Using the features described in the previous section, it is easy to re-implement a FIFO algorithm similar to the one described in Chapter 6. The first step, as before, is to define an implementation-specific requestor model. Recall that requestor information is not relevant for a FIFO arbitration scheme, which means that we can define the requestor model as `unit_eqType` again.

Next, we must use the interface `Scheduler_t` (described in Listing 8.6) to implement the algorithm. Before implementing the function `Schedule`, the arbiter's internal state must be defined. In this case, differently than the FIFO implementation described in Chapter 6, no internal state is required, since there is no need to keep a counter to account for the passing of time and manage timing constraints. Hence, we define the algorithm's state type as shown below.

```
Definition FIFO_internal_state := unit.
*[global] Instance SCH_ST : SchedulerInternalState := mkSIS FIFO_internal_state.
```

Note that this version of FIFO also does not require a parameter to model processing windows (such as `WAIT`, from the previous FIFO implementation). This comes from the fact that timing constraints cannot be violated, hence, it suffices to pick commands from the list of ready commands whenever they appear.

Finally, the `Schedule` function can be implemented as shown in Listing 8.11. Straightforwardly, we use the `ohead` function to select the first command from the list of pending commands/requests. Since the `Next_SL_state` algorithm always inserts arriving requests in order and never changes the order of requests in the data structure, the head of the list (`map`) always contains the oldest command/request. If the list is empty, the `ohead` function returns a `None`, which in this case, results in `FIFO_schedule` issuing a `NOP` command.

Bear in mind that this FIFO implementation also issue `REF` commands, since it relies on

Listing 8.11: Re-implementing FIFO using the bank machines.

```
Definition FIFO_schedule (map : ReqCmdMap_t) (SS : SystemState_t)
    (FIFO_st : FIFO_internal_state) : Command_kind_t :=
    let cmd := seq.ohead map in
    match cmd with
    | Some cmd ⇒ cmd
    | None ⇒ NOP
    end.
```

the bank machine described in the previous section. More specifically, it issues a REF command at every t_{REFI} cycles, each preceded by a PREA command issued t_{RP} cycles before. As explained previously, other types of commands take the urgency of the refresh management operation into consideration, i.e., commands are considered not ready whenever a PREA or a REF is due. This also means that the (current) refresh management strategy implemented in the bank machine does not exploit the flexibility that comes from postponing and advancing the issuing of refresh commands. Future developments could further improve the bank machine by implementing other refresh management strategies, which ideally should be defined as parameters, as it is currently done for the row-buffer policy.

Furthermore, although we did not yet prove the refresh-related POs, intuitively, it is clear that issuing a REF command every t_{REFI} cycles clearly complies with the stated POs. Note, however, that the correctness of the refresh strategy is independent from the scheduling algorithm, i.e., once its correctness is proven, every algorithm written on top of Scheduler_t can benefit from refresh management and its underlying correctness.

More generally, this implementation of FIFO is not yet fully proved. The t_{RCD} proof, described in Section 8.2.1 is inherited by the implementation, but other timing constraints are yet to be proved. Proving the remaining timing constraints, along with the high-level properties stated in Arbiter_t is left as future work, as discussed in Chapter 10.

Of course, not all scheduling algorithms can be that easily implemented with the new interface. In the following chapter, we describe TDMShelve, a complex new algorithm built using the features described in this chapter.

Chapter 9

TDMShelve – A New DRAM Scheduling Algorithm

Chapter 6 introduced FIFO and TDM, two *Proof-of-Concept* DRAM scheduling algorithms. While these two served as valid first use-cases for CoqDRAM, they are too simple. In this chapter, we introduce `TDMShelve`, a *new* DRAM scheduling algorithm.

To better understand the motivation behind the design of `TDMShelve`, we look back at the two research problems stated in Section 1.1:

1. *How to address the limitations of a specific existing state-of-the-art memory request scheduling algorithm for real-time mixed-criticality systems?*

2. *How to address the development of real-time memory controller scheduling algorithms in a trustworthy manner?*

While the CoqDRAM framework itself provides an answer to the second question, a natural follow-up presents itself as "*can the CoqDRAM approach be used to design complex and competitive DRAM scheduling algorithms (w.r.t. other State-of-the-Art algorithms)?*" Moreover, as stated in the first point, we wish to improve Hebbache's algorithm [39], [40]. The `TDMShelve` algorithm attempts to answer/resolve both of these questions.

At time of writing, the algorithm has been successfully implemented and has gone through some sanity checks. Since the algorithm is implemented using the bank machine abstraction from Section 8, it *inherits* the proofs about timing constraints from the Interface Sub-Layer. Unfortunately, due to limited time, we did not yet prove the algorithm's high-level properties nor carry out an in-depth performance evaluation – these are some obvious starting points for future work (which are discussed in further detail in Chapter 10).

In the following, we briefly present Hebbache et al.'s algorithm before presenting `TDMShelve` and its CoqDRAM implementation.

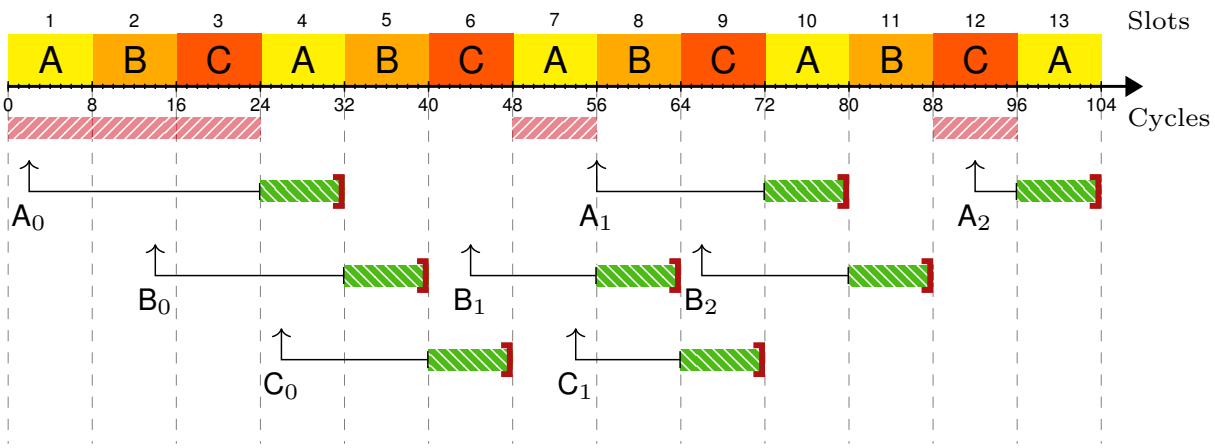


Figure 9.1: Regular TDM arbitration of three tasks A, B, and C, including periods of memory activity (green) and idling (red).

9.1 Work-Conserving Dynamic TDM

Hebbache et al.'s algorithm explores memory request arbitration on a multi-core architecture consisting of m cores with private caches and a single shared memory, i.e., cache misses result in memory requests to transfer cache blocks. In summary, the algorithm proposes an improved arbiter of memory requests that preserves the temporal isolation provided by TDM, but improves the bandwidth through better memory usage. To do so, the algorithm loosens the concept of *strict slot ownership*. In other words, requests can be serviced in arbitrary slots, and slot boundaries are merely used to keep track of deadlines. Each request is associated with a deadline that corresponds to the completion date of the request if it had executed under TDM arbitration. The algorithm is then able to relax the scheduling of memory requests, while ensuring that all deadlines are respected.

To start with a more detailed explanation, first consider a TDM arbitration scheme, which works only by considering slot boundaries – from now on called *strict TDM*. Such arbitration is depicted in Figure 9.1 (reproduced from Hebbache et al.'s paper [40]). Each task is assigned a dedicated TDM slot (vertical columns, labelled **A** through **C**) that alternate over time. As it can be seen, memory requests are only *processed* in their respective TDM slots – which results in the memory being idle (i.e., unused) for long periods.

Next, we present two variants of the algorithm proposed by Hebbache et al. To begin, the authors refine their task model by defining two criticality classes: critical and non-critical tasks. Citing the authors: "we simply assume that critical tasks are associated with a strict deadline that has to be met under all circumstances. The underlying computer platform and memory arbitration scheme thus have to provide a means to bound the WCET of these

tasks. Non-critical tasks, on the other hand, may miss their deadlines. In contrast to typical mixed-criticality systems, we do not demand strict worst-case execution time bounds for them in this work. The underlying hardware can thus execute these tasks in a best-effort manner."

The core idea of a first algorithm variant is to allow "non-critical tasks to share TDM slots, or, in the worst-case, simply recycle unused TDM slots leftover by the critical tasks" (i.e., the slots shown in red in Figure 9.1). Moreover, still according to the authors, "the strict separation of critical tasks would allow to easily establish worst-case execution time bounds, while non-critical tasks would improve the memory utilisation."

The first version of the algorithm implementing such ideas is dubbed `TDMds` (*dynamic TDM with slack counters*). According to the authors, for critical tasks, deadlines are derived for each request, which simply corresponds to the end of the task's next TDM slot after the request's issue date. The `TDMds` arbiter is then free to scheduler memory requests dynamically, as long as the request deadlines of critical tasks are respected [40]. Furthermore, a slack counter is associated with each critical task. This counter indicates how many cycles before a given request of a task completed before its deadline. When a new request is issued by a task that previously accumulated some slack, the request's issue date **occurs earlier than expected under a strict TDM scheme**. Consequently, also the corresponding deadline appears earlier than under strict TDM. The authors then proceed to compute a so-called *delayed issue date*, which simply consists of adding the previously accumulated slack back to the issue date of a new request. The delayed issue date may then potentially push the deadline farther into the future. During the execution of a task, the deadlines computed under `TDMds` considering the accumulated slack, exactly correspond to the deadlines/completion dates under strict TDM. Note, however, that the slack accumulated by a task is not preserved for subsequent jobs of a task, i.e., slack counters have to be reset to zero at task start.

Figure 9.2 shows an execution of the task set from Figure 9.1 under `TDMds`, considering tasks **A** and **B** as critical tasks and task **c** as non-critical. The deadline may well lie far after the request's actual completion date, and thus generate slack for the task issuing the request (e.g, requests A_0, A_2, and B_0). The value of the slack counter is displayed as a superscript for each request. For instance, request A_1 has accumulated 8 cycles of slack (superscript 8Δ in Figure 9.2). At the beginning of each TDM slot, the arbiter chooses one of the issued requests, independently from the actual owner of the slot. This is, for instance, the case for non-critical request c_0, which is granted access to the memory despite the fact that critical request B_1 has been issued. The slack accumulated by task **B** is here spent in favour of the non-critical task. In comparison to regular TDM (Figure 9.1), `TDMds` is more efficient in this

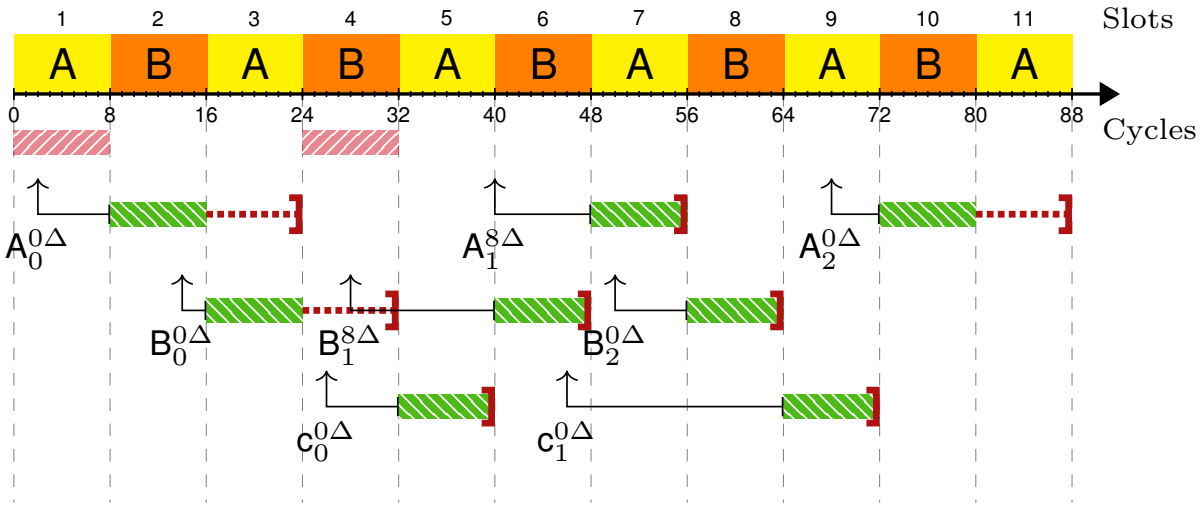


Figure 9.2: Improved arbitration using TDMds of two critical (**A** and **B**) and a non-critical task (**c**).

example. The last request A_2 completes 3 TDM slots earlier. Note, that the slack counters are all reset to zero for subsequent requests of the critical tasks **A** and **B**.

`TDMds`, however, still limits arbitration to be aligned with TDM slot boundaries, which results in the memory being idle in situations where there is at least one already-pending request. The authors call these periods of idling **issue delays**. For instance, `TDMds` generates an issue delay of 6 cycles for request c_0. This delay could have been avoided if requests were handled independently from TDM slots. For instance, task **A**, the owner of TDM slot 5, has accumulated some slack (8Δ), which ensures that the deadline of any request issued by task **A** after request c_0 will be at the end of slot 7 or later. It thus would be safe for request c_0 to immediately access the memory during the unused TDM slot 4. Moreover, the authors introduce yet another concept – **release delays**. Release delays denote the number of clock cycles during which at least one request is issued to the memory arbiter after the completion of the memory request of a used TDM slot. Here, for brevity, we only (briefly) describe the authors' solution to **issue delays**, an algorithm dubbed `TDMes` ("`es`" standing for "*early start*").

The core idea behind `TDMes` is to use, at any time, slack counters to take a peek into the near future and take arbitration decisions based on that information. In other words, at any time, the scheduler can use slack to analyse whether the *next* TDM slot can be used by another task without compromising the deadline of any potential memory request of the slot's owner. If the scheduler can conclude that indeed the next slot's owner will not have a request with a deadline marked by the end of the slot, then it is safe to schedule another memory request at any instant, thus decoupling arbitration from TDM slot boundaries.

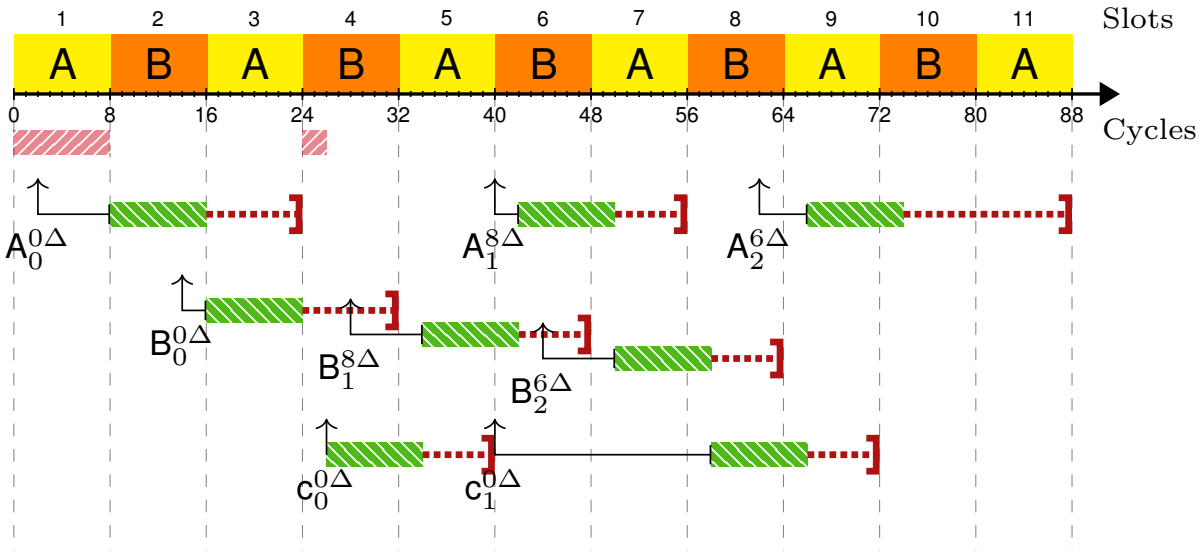


Figure 9.3: Reduced issue delays thanks to the `TDMes` arbiter, which operates independently from the actual TDM slot length.

Figure 9.3 illustrates the resulting arbitration under `TDMes` for the task set from before (**A**, **B**, and **c**). Note that, at cycle 26, request c_0 can start early, since at the time it arrives, **A** has enough slack for the scheduler to concluded that a possible incoming request from **A** would have its deadline pushed to the end of slot 7. Thus, it is safe to start processing c_0 at cycle 26. Here, we do not detail the precise rule that allows an early start to be triggered. Interested readers should refer to Hebbache et al. [40].

Moreover, although the rules for computing the deadlines and slack in `TDMShelve` are very similar to the ones proposed by Hebbache et al., the implementation adopted in `TDMShelve` differs greatly. For that reason, we only discuss these deadline and slack computation rules further in the text, when explaining the `TDMShelve` algorithm.

Limitations

Although the algorithm proposed by Hebbache et al. (in its different flavours) does succeed in providing good average-case performance[1] for low-criticality tasks while still guaranteeing strict TDM bounds for high-criticality tasks, there is still room for improvement. More generally, it can be said that both `TDMes` and `TDMds` do not exploit the slack information at other levels of the memory hierarchy – such as the DRAM. In more detail, the memory model considered by the authors is over-simplified, i.e., the servicing of memory requests is considered to be a contiguous time window, without any details about the underlying

[1] In simulation, considering a simple memory model.

memory technology and specific timings within such processing window. This is a somewhat incomplete interface, since considering the internal structure and features of the DRAM, for example, would allow better scheduling decisions.

As an example, take request A_0 from Figure 9.3: at the time it arrives in the system, task **B**, the owner of the subsequent slot has no slack, which means that it is impossible to conclude at that instant that slot 2 will go unused by B, thus resulting in slot 1 going fully unused (i.e., there is an incurred **issue delay**).

If, however, the arbiter disposed of information about the processing latency of A_0 and its *potential* impact on the latency of requests coming from B, then a more *informed* scheduling/arbitration decision could be made. More concretely, if, for example, the scheduler knew beforehand that requests coming from tasks **A** and **B** were *always* mapped to different DRAM banks (by imposing that bank mapping condition), then A_0 could start being processed at the time it arrives without compromising the guarantee that a potential request arrival from task **B** will miss its deadline. This comes from the fact that the timing constraints between DRAM commands targeting different banks are **generally** much shorter than those of commands targeting the same bank. Such idea is the core of `TDMShelve`, as explained in the following section. As a result, even more idle time can be allocated for the processing of memory requests coming from low-criticality tasks.

More generally, the objectives of `TDMShelve` are: 1) to extend Hebbache et al.'s idea by considering a realistic memory model – namely DDR, where requests are potentially processed through multiple commands that may be interleaved, and 2) to exploit the characteristic of DDR to improve scheduling decisions.

9.2 TDMShelve

First, we present the core ideas behind `TDMShelve`, with the goal of giving the reader an intuitive understanding of its basic functionality. Next, we present the algorithm more formally while going through an example. Last, we discuss its CoqDRAM implementation – which makes use of the interfaces described in Chapter 8.

There are two core ideas behind `TDMShelve`. The first one is to leverage information about address mapping and timing constraints between DRAM commands at *request scheduling* time. In more detail, since Hebbache et al.'s algorithm considered arbitration over the common bus rather than the DRAM, it did not exploit information about how memory requests are *mapped* to the memory (i.e., in terms of bank-groups, banks, rows, and columns) and how this mapping can influence the latency between DRAM commands.[2] Therefore, by

[2]As a reminder, the latency between *some* DRAM commands depends on whether they target the same

considering the address mapping and the variable command latency, safe assumptions can be made about the *timing interference* that a memory request can have over other requests.

The second idea is to *shelve* (i.e., in the sense of "interrupting" or "preempting"[3]) outstanding memory requests. This comes from the fact that memory requests are "translated" to a series of DRAM commands; thus, by keeping track of the state of the DRAM device, it is possible to "remember" the next command needed to service a given memory request at any given time. Knowing which command is required by any request at any given time allows scheduling techniques that are based on "picking" DRAM commands from a sort of "shelf" (hence the algorithm's name).

For instance, imagine that there are two pending requests sitting at the "shelf" (i.e., pending/outstanding requests) – R_A and R_B. We may, at first, pick R_A from the shelf and schedule the next command it requires. Next, we may decide to put R_A back in the shelf (assuming that it is not yet completed), pick R_B from the shelf, and schedule a command from R_B, which might be needed in order to make sure that its deadline is respected. In other words, it is possible to *interleave* the processing of different memory requests. `TDMShelve` makes use of this interleaving technique to shelve a request if scheduling another (critical) request is *necessary* for it to meet its deadline. As discussed later in the text, this ensures that critical requests always meet their deadlines (coming from strict TDM) and enhances overall performance, i.e., the memory stays less time idle (compared to Hebbache et al.'s algorithm), which, in consequence, frees more bandwidth for requests coming from non-critical requestors/tasks.

More formally, we start by presenting the algorithm's parameters and assumptions and then go through the scheduling rules while examining an example.

9.2.1 Preliminaries

The algorithm has three quantitative parameters. The first two come from strict TDM: SL is the TDM slot length, and SN is the number of slots in a TDM period. $SN \times SL$ is the number of clock cycles in a TDM period. The third parameter is the total number of *requestors* in the system, T_{req}.

Moreover, *the algorithm works with both open-page and closed-page row-buffer policies*. The row-buffer policy to use, along with the bank mapping scheme are also parameters to the algorithm (more on that later).

Requestors (and consequently memory requests) can have two criticality levels: *critical*

or different banks/bank-groups.

[3]The *shelving* mechanism employed in `TDMShelve` is not *quite* a preemption, as discussed further in the text.

requestors are always identified by uppercase letters and non-critical requestors by lowercase. The n-th request from a critical requestor A is thus written as A_n. The n-th request from a non-critical requestor c is written c_n. From this point onwards, a *critical requestor* will be shortened as "CR" and a *non-critical requestor* as "NCR". Requests issued by CRs and NCRs are referred to as "critical requests" and "non-critical requests", respectively.

Furthermore, there are as many TDM slots as there are critical requestors in the system, and each slot is *associated* to a CR. In fact, the concept of slot *ownership* is rather loose. A slot being associated to a requestor does not grant the requestor exclusive access to the memory, as in `TDMes`. Instead, as we will see, *slot bounds are used to keep track of deadlines* and slack – which are important information used by the scheduling algorithm.

Assumption 1: There are at least two CRs in the system. This also means that there are at least two slots ($SN > 1$).

Assumption 2: Critical requests that are mapped to adjacent TDM slots never target the same bank. In other words, CRs are mapped to sets of banks according to a private bank mapping scheme [18]. No assumptions are made on the mapping scheme for non-critical requests (i.e., NCRs can target any bank – as in a shared bank mapping scheme [24], [26]).

Note that Assumption 2 implies that there must be at least two disjoint sets of banks, if the number of CRs is even, or three, if the number of CRs is odd. This is reasonable, since DDR3 DRAM devices have 8 banks, and DDR4 DRAM devices have from 4 to 16 banks.

9.2.2 Determining the Slot Length (SL)

Moreover, and most importantly, Assumption 2 allows us to significantly reduce the slot length SL – in comparison to a SL that would have been necessary if no assumption was made. Bear in mind that a shorter slot length means a tighter WCL for memory requests issued by CRs [40]. To better understand why this is, consider the scenario depicted in Figure 9.4.[4]

The figure involves two critical requestors: A and B. This means that there are two alternating TDM slots ($SN = 2$). β_A and β_B are disjoint set of banks attributed to A and B, respectively ($\beta_A \cap \beta_B = \oslash$).[5]

[4]For simplicity, Figure 9.4 does not mention the requests that lead to the depicted commands. The scheduling algorithm is also irrelevant. The purpose of the figure is solely to show how the slot length is calculated.

[5]Note also that although the figure shows different command buses for β_A and β_B, there is in reality just a single command bus, given by the overlap of both buses appearing in the figure.

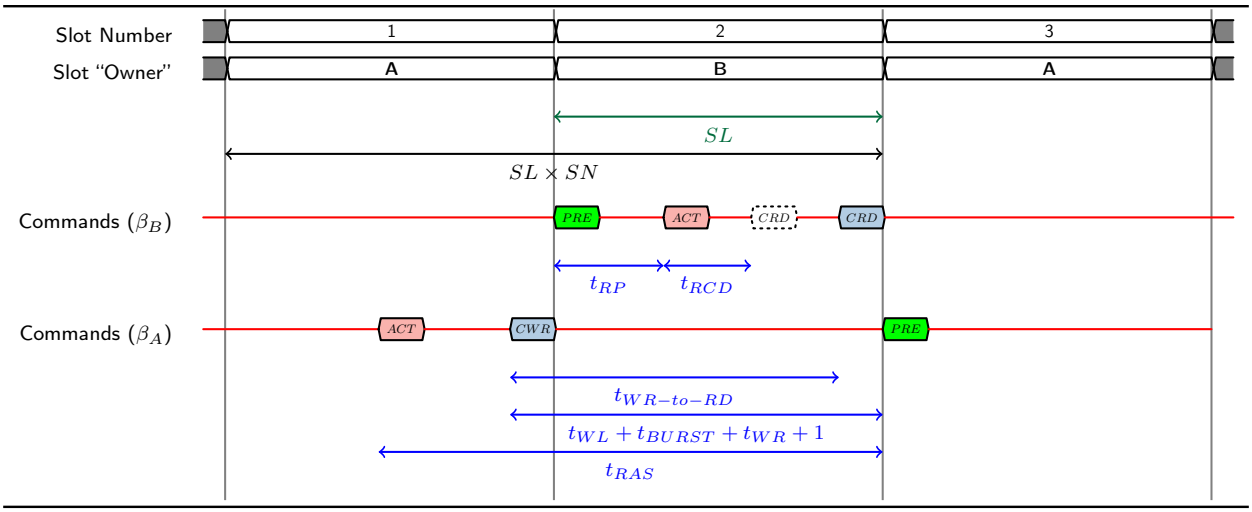


Figure 9.4: TDMShelve – choosing an appropriate slot length (SL).

In summary, **each slot is made long enough to accommodate a `PRE-ACT-CAS` (PAC) sequence, regardless of what happens in the neighbouring slots**. This comes from the observation that, in the worst-case, in order to meet its deadline (as we will see next, in more detail), a critical request needs to fit entirely in a slot. Or, more precisely, the *commands* needed to service a critical request need to fit entirely in a slot. Since `TDMShelve` does not exclude bank interference (recall from Assumption 2 that requests issued by NCRs may still interfere with CR's banks), we *must* assume that any critical request *may* be a row-buffer miss – which means that a slot must be at least big enough to issue a *valid* PAC sequence (i.e., all timing constraints must be respected between the commands of the sequence).

Moreover, because we know that neighbouring slots target different banks, no intra-bank timing constraints apply between commands associated with requests that target different banks. As an example, note that in Figure 9.4, A's first slot (Slot 1) ends with a CWR command. Since B targets a different set of banks, no timing constraint exists between A's CWR and B's PRE . If the possibility that requests of A and B targeted the same bank existed, then, in the worst-case, additional intra-bank constraints would apply – which would result in a larger SL.

In addition, inter-bank timing constraints between neighbouring slots still apply and must thus be considered in order to choose a correct SL. Notably, in Figure 9.4, even if t_{RCD} cycles have elapsed since B's ACT is issued (meaning that the `ACT-to-CAS` constraint has already been satisfied), B's CRD command is not yet *ready*, since the inter-bank $t_{WR-to-RD}$ constraint has not yet been satisfied.

Finally, intra-bank constraints between the two neighbouring slots of B must also be

considered – since these could be mapped to the same bank. In Figure 9.4, for instance, the slot must be large enough to "separate" A's ACT and CWR (in Slot 1) from a PRE happening at the beginning of A's subsequent slot (Slot 3).

Be mindful that Figure 9.4 does not depict *all* timing constraints involved in choosing an optimal SL. In fact, the optimal value of SL is the solution to a linear problem involving exactly *10* inequalities (which are shown in Section 9.2.5). For a DDR3-800E device, for example, solving the problem yields an SL value of 27.[6] Without the bank mapping assumption, the value of SL would be 45 (the duration of the PAC sequence, $t_{RP} + t_{RCD} + 1$, plus the CWR to PRE constraint: $t_{WL} + t_{BURST} + t_{WR}$).

9.2.3 The Algorithm

The scheduling rules from `TDMShelve` are based on deadlines and slack counters, similar to `TDMds` and `TDMes`.

The core idea of the algorithm is to pick *one* requestor among requestors that have pending requests to hold an exclusive *grant*. While a requestor holds the grant, *only* (ready) commands originating from requests issued by that requestor can be scheduled. If the requestor holding the grant does not have any ready commands, then it holds the grant nevertheless until the grant is given to some other requestor.

`TDMShelve` is based on the assumption that it picks from a list of ready commands (as a reminder, commands are said to be ready only when all timing constraints are satisfied). In the CoqDRAM `TDMShelve` implementation – this is managed by the bank machines described in Chapter 8. Therefore, the scheduling rule for commands is straightforward:

Command scheduling: The scheduled command (i.e., the command sent to the DRAM device) is the first *ready* command belonging to the requestor holding the grant. If the requestor holding the grant has no ready commands, or if no requestor holds the grant, than a `NOP` is issued instead.

The real innovations brought by `TDMShelve` come in the request scheduling part – or, in other words – *deciding who gets the grant*. There are only a few rules to decide who gets the grant. These rules are part of an update function that gets executed every clock cycle.

Rl_1 **Rule 1 (Shelving) – Request scheduling at the beginning of a new slot:** At the beginning of an arbitrary slot S – owned by a requestor A – if another arbitrary requestor B already holds the grant, then we test if A has a pending request. If it does, we call such

[6] For such device, the most constraining inequality is $t_{RAS} - 1 \leq SL$, which yields $SL = 27$.

request A_n. Then, we test if A_n's deadline is due at the end of S. If it is, this means that the processing of B's request must *halt* and the processing of A_n must immediately start for it to meet its deadline.

Rl_2 **Rule 2 (No Grant) – Request scheduling at the beginning of a new slot:** If, at the beginning of an arbitrary slot S, the conditions for **Rule 1** do not apply, *and* there are no pending requests or the command issued in the last cycle was a `CAS` (meaning that a request has just completed) – then no requestor gets the grant.

Rl_3 **Rule 3 (Pick) Request scheduling at the beginning of a new slot:** If, at the beginning of an arbitrary slot S, the conditions for **Rule 1** and **Rule 2** do not apply – then we choose a new grant holder based on `TDMShelve`'s scheduling function, which we call `BankFiltering` (described further down).

Rl_4 **Rule 4 (Middle) – Request scheduling at an arbitrary (non-zero) cycle of a slot:** If no requestor holds the grant at an arbitrary (non-zero) cycle/position of an arbitrary slot S, then a new grant holder is picked by `BankFiltering`. Otherwise, the grant holder keeps the grant, unless the command issued on the last cycle was a `CAS` – meaning that the request completed. In the latter case, no requestor gets the grant.

In order to understand how `BankFiltering` works precisely, it is necessary to detail how deadlines and slack are computed. These computations are similar to what has been proposed by Hebbache et al. [40], with only small modifications/adaptations.

Deadlines: Each requestor is associated to a decrementing counter to account for deadlines. We call such counter D_n – where n is an arbitrary requestor. The deadlines of CRs are initialised with a value equal to the number of cycles until the end of its respective TDM slot in the first TDM period. The deadlines of NCRs are initialised as 0. Then, each cycle, an update function is called to calculate a new deadline value for each requestor. For each CR, the deadline counter replenishment is triggered by three **disjunctive** conditions: 1) when the counter reaches 0; 2) when a request originating from the respective CR has just completed the previous cycle (which is indicated by the last issued command being a `CAS` belonging to the respective CR); or 3) when there is enough slack to conclude that the owner of the next slot will not have an associated deadline at its boundary. Replenishing a counter means adding a full TDM period to the counter, i.e., the current counter is incremented by another full TDM period. In all other cases, the deadline is decremented by one. For NCRs, whenever a request arrives, deadlines are set to the end of the next slot, independently of its

owner. The deadline counter is then decremented by one until it reaches 0. Once it reaches 0, it stays at 0 until another request from the respective NCR arrives.

Slack: Each critical requestor is associated to a counter to account for slack. We call such counter Δ_n – where n is an arbitrary critical requestor. At initialisation, all slack counters are set to 0. Then, at each clock cycle, an update function is called to calculate the new value of the counter for each CR. The updating rule is stated as follows: $\forall r \in CR$, if a request belonging to r has completed on the previous cycle, then the slack counter will be set to be the current deadline counter minus one, i.e., slack is the "remaining deadline" upon completion.

`BankFiltering`**:** The goal of `BankFiltering` is to *filter/pre-select* which requestors can be considered for scheduling at any given clock cycle. Whenever `TDMShelve` decides to pick a new grant holder (by rules Rl_3 and Rl_4), it calls `BankFiltering` to decide which requestors can be considered for scheduling. More precisely, `BankFiltering` filters out requestors that might interfere with the set of banks attributed to the next slot owner whenever a deadline miss is at risk. Then, after filtering out ineligible requestors, `BankFiltering` calls another algorithm P to finally decide who gets the grant. The actual choice of P is not relevant w.r.t the guarantee that deadlines are met, since this is assured by `BankFiltering` and the shelving rule of `TDMShelve` (Rl_1). An actual implementation of P is nevertheless discussed further in the text.

`BankFiltering` works precisely as follows: if the algorithm is called at an arbitrary cycle C of a given slot S – then it analyses the following slot, $S+1$. Assume that $S+1$ is associated with an arbitrary requestor A. The analyses consists in testing if A has any pending request, which results in a total of four cases:

1. BF_1 If A has no pending request, then `BankFiltering` checks A's slack counter, Δ_A: if, at C, there is enough slack to conclude that no request to A will miss its deadline, i.e.,

$$SL - C \leq \Delta_A,$$

   then `BankFiltering` does not filter out any requestors, i.e., *all* requestors are considered for scheduling by `P`. This comes from the fact that in this case, any request from A that arrives after C will have its deadline postponed by a whole TDM period ($SN \times SL$). Therefore, since A has no immediate upcoming deadline, any request can be issued, including requests that target A's bank.

2. BF_2 If, however, *there is no pending request but the slack condition is false*, then, at C, it is not possible to conclude that $S+1$ can be utilised without risking a deadline miss, which means that `BankFiltering` must keep only requests that *do not* target A's bank and requests belonging to A itself as inputs for `P`. Recall from the SL calculation that slots are designed in a way such that a PAC sequence can be issued regardless of what happens in the previous slot, as long as the previous slot does not contain commands to the same bank. Therefore, any request targeting a bank different than A's bank will not compromise A's deadline guarantee if an A request arrives between C and the start of $S+1$.

3. BF_3 If A *does* have an arbitrary pending request A_n at C, then, rather than looking at A's slack, `BankFiltering` looks at A_n's deadline, D_A: if D_A is due at the end of $S+1$, i.e.,

$$D_A \leq 2 \cdot SL - C,$$

then only A's requests or requests *not* targeting A's bank can be considered to hold the grant. This is because giving the grant to a requestor targeting A's bank could compromise the guarantee that A_n meets its deadline.

4. BF_4 If A has an arbitrary pending request A_n at C, but A_n's deadline is not due at the end of $S+1$, then `BankFiltering` does not filter out any requestor, since no choice can compromise the guarantee that D_A will eventually be met.

Now, we discuss a concrete implementation for P based on the *Earliest Deadline First* (EDF) policy adapted to mixed-criticality systems – which we name `mcEDF`.

`mcEDF:` The goal of `mcEDF` is to pick a new grant holder among an already-filtered list of requestors – passed as argument. The underlying algorithm loops over such a list of requestors, as shown in Algorithm 2.

As it can be seen, Algorithm 2 picks the requestor with the earliest deadline among the pre-selected requestor list. There are some special cases, however: if i (the iteration variable) is a requestor with an upcoming deadline (Line 6), then i is chosen. Moreover, on a deadline tie, i.e., when `min_deadline` and `cur_deadline` are the same (Line 8), the critical requestor i gets picked, since a tie can only happen between a CR and an NCR. If i is an NCR, then `mcEDF` can only pick i if the following condition is satisfied: the deadline associated to i (`cur_deadline`) must be strictly less then the saved deadline (`min_deadline`) **AND** the

Algorithm 2 `mcEDF`

Inputs: Requestors, State
Outputs: NewGrantHolder

```
1: s ← None
2: for i in Requestors do
3:     min_deadline ← getDeadline State s
4:     cur_deadline ← getDeadline State i
5:     if isCriticalRequestor i then                    ▷ i is a critical requestor (CR)
6:         if cur_deadline ≤ SL then s ← i
7:         else
8:             if cur_deadline ≤ min_deadline then s ← i
9:     else                                             ▷ i is a non-critical requestor (NCR)
10:        if (cur_deadline < min_deadline)
11:         AND (NOT ((isCriticalRequestor s) AND (min_deadline ≤ SL))) then s ← i
    return s
```

saved requestor must not be a critical requestor with an upcoming deadline (Lines 10 and 11). Furthermore, since Gallina (Coq's programming language) is a pure functional programming language, the CoqDRAM implementation of `TDMShelve` uses a *fold* rather than a `for` loop to implement `mcEDF`, but the logic is the same as described in Algorithm 2.

9.2.4 TDMShelve Execution Example

As an example of a `TDMShelve` execution, consider Figure 9.5. The scenario depicted in the figure involves two CRs, A and B, and two NCRs, c and d – which imply $SN = 2$ and $T_{req} = 4$.

The execution from the example is based on the open-page command generation policy, i.e., after accessing a column address from an open row (via a `CAS` command), no automatic precharge is triggered. Keep in mind that the algorithm would still work and the TDM guarantees would still hold if we had opted for a closed-page policy.

Moreover, the requestors are mapped to banks in the following way: A is mapped to bank β_A, B is mapped to bank β_B, c is mapped to banks β_A and β_C, and d is mapped to bank β_B. Note that, according to such mapping configuration, c can interfere with A and d can interfere with B. Furthermore, for simplicity, we only consider a single bank-group in the system, which is the default for DDR3 devices and a pessimistic assumption for DDR4 devices.

The example from Figure 9.5 comes from a real execution of `TDMShelve` on a CoqDRAM model of a DDR3-800E device. Using a real device model allows us to check the algorithm's functionality using real timing constraints. For such device, as mentioned previously, the

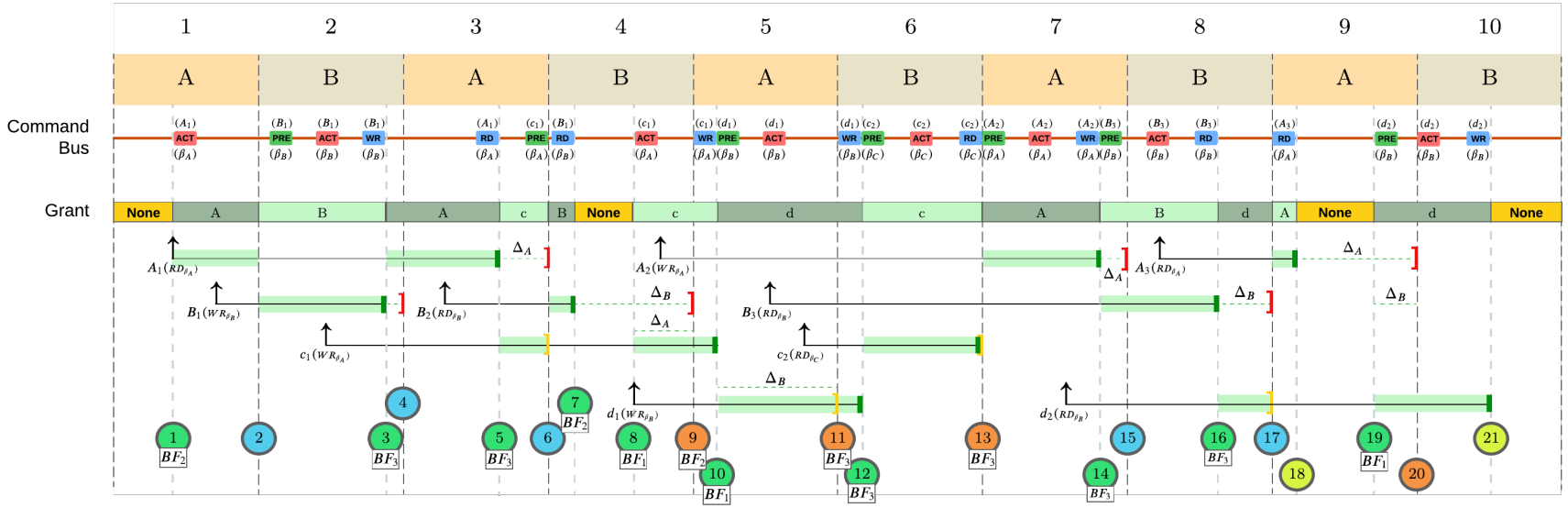


Figure 9.5: An execution of `TDMShelve` on a DDR3-800E device.
Parameters: $SL = 27$, $SN = 2$, $T_{req} = 4$.
Rules: Rl_1 (Shelve); Rl_2 (No Grant); Rl_3 (Pick); Rl_4 (Middle).
`BankFiltering` cases: BF_1 (No request, enough slack); BF_2 (No request, not enough slack);
BF_3 (Pending request, upcoming deadline); BF_4 (Pending request, deadline is not upcoming).

optimal value of SL (considering Assumption 2) is 27. We refrain from showing absolute dates in Figure 9.5, and instead only refer to specific instants – as indicated by the circles (n) on the bottom part of the figure. The circles are coloured according to scheduling the rule used at that instant (i.e., Rl_1, Rl_2, Rl_3, or Rl_4). Moreover, whenever `BankFiltering` is called to pick a new grant recipient (which can happen from either rules Rl_3 and Rl_4), we indicate which sub-case of `BankFiltering` is triggered (i.e., BF_1, BF_2, BF_3, or BF_4). In the figure, the latter is indicated by a rectangle below the circle indicating the instant/rule.

Keep in mind that the execution trace shown in the figure is only an *illustration* of the real execution trace, where the distance between commands in the commands bus have been exaggerated. Still, even with this exaggeration, the relative distance between commands and their placement within slots is coherent with the real trace.

Next, we go from time stamp to time stamp analysing which rules are used for scheduling the requests that arrive in the system. At initialisation, no requestor holds the grant – which is indicated by `None`. Moreover, consider that β_A is idle at initialisation, but not β_B and β_C (i.e., these banks already have a row loaded in their row-buffers). We consider that a request to be *finished* whenever a CAS command associated to that request is issued.

1 A_1 arrives, a `RD` request to β_A. Since we are somewhere in the middle of Slot 1, Rl_4 applies and `BankFiltering` is called to pick a new grant holder. At this point, `BankFiltering` looks at the next slot, which is "owned" by B: since B has no pending request and no slack **(case 2 –** BF_2**)**, it is not possible to conclude that B will not have a request whose deadline is the end of Slot 2. Therefore, `BankFiltering` only considers requests to a different bank than B's (besides requests from B itself – of which there are none at 1). Since requests from A always target a different bank than requests from B (from Assumption 2), A gets the grant and an ACT command is immediately issued, assumint that β_A was initially idle. Due to Rl_4, A holds the grant for the remainder of Slot 1, but no other command is issued – since timing constraints are not yet satisfied.

2 At the beginning of Slot 2, all pre-conditions for Rl_1 apply: A holds the grant but B now has an outstanding request with an upcoming deadline – B_1, a `WR` request to β_B. Hence, `TDMShelve` proceeds to take the grant from A and gives it to B, shelving A_1 in the process. In addition, consider that B_1 is a *row-miss*. Since β_B was not idle at initialisation and B_1 is a row-miss, a PAC sequence must be issued. Some time later, when the CWR command is issued, the request is considered completed, and, due to Rl_4, the grant goes back to `None` for a single cycle (although this idle phase/transition to `None` is not visible in the figure).

3 When B_1 finishes, Rl_4 applies again and `TDMShelve` picks a new grant holder by calling `BankFiltering`. Since the algorithm finds itself somewhere in Slot 2, it looks at Slot

3, which is "owned" by A: A, in this case, has a pending request with an upcoming deadline, A_1 **(case 3 –** BF_3**)**. Because of that, only requests targeting a different bank than A's are considered, which excludes c_1 – a `WR` request to bank β_A – from consideration. Since A_1 is the only option left, A gets picked. Note that, if c_1 targeted a different bank than A's, `BankFiltering` would not have filtered it out, but `mcEDF` *still* would have picked A, since on a deadline draw between a CR and a NCR, the CR is prioritised.

4 At the beginning of Slot 3, all conditions for Rl_1 are met, but since A is already the grant holder, A_1 does not get "shelved" and its processing continues. Strictly speaking, Rl_1 applies, A_1 gets shelved but picked up again. Note that, since no other requests closed A_1's row, it must only schedule a CRD , which is ready only some time after B_1's CWR was issued. When A_1 is completed, note that A accumulate some slack – which is indicated by Δ_A.

5 When A_1 finishes, a new grant holder is picked by `BankFiltering` according to Rl_4. Here, the function looks at Slot 4, which belongs to B: since B has an outstanding request with an upcoming deadline **(case 3 –** BF_3**)**, requests to the same bank as B's would be filtered out, if there were any. In this case, both pending requests (B_2 and c_1) are eligible to receive the grant. Then, `mcEDF` picks c over B to hold the grant, because its deadline happens earlier. c now holds the grant and a PRE is issued sometime after A_1's CRD , since c_1 targets a different row than the one loaded by A_1.[7]

6 At the beginning of Slot 4, according to Rl_1, c_1 gets shelved and B gets picked to be the grant holder instead. Note that this results in c_1 missing its deadline, which is allowed for NCRs. B_2 is a row-hit to β_B, which means that only a CRD command is needed to complete the request. Note that, after finishing, B accumulates some significant amount of slack –which is indicated by Δ_B.

7 When B_2 finishes, according to Rl_4, `BankFiltering` is called to pick new grant holder. The function analyses Slot 5, which belongs to A: at this point, A does not have a pending request and *there is not enough slack to conclude that Slot 5 will go unused* **(case 2 –** BF_2**)**. Therefore, `BankFiltering` filters out c_1, since "executing" it could compromise the guarantee that an A request meets its deadline (because c_1 targets A's bank). Since no other requests are pending, no requestor gets the grant.

8 At this point, Δ_A is enough to conclude that no request from A will have its deadline

[7]The requests that arrive in the system in Figure 9.5 obey the following rule of thumb: consecutive requests by the same requestor target the same row, which means that the second request would be a row-hit if no bank interference happened in the meanwhile. However, requests from a different requestor to the same bank always target a *different* row – as a way to generate bank interference between NCRs and CRs.

marked by the end of Slot 5 **(case 1 –** BF_1**)**. As a consequence, due to Rl_4, `BankFiltering` considers all requestors, c gets picked by `mcEDF` to be the grant holder, and the processing of c_1 continues. Note also that A_2 arrives some time after 8, but that does not trigger any grant changes, since the grant can only be transferred at slot boundaries or when a request completes.

9 Since A_2's deadline lies only at the end of Slot 7, no shelving occurs. Instead, according to Rl_3, the algorithm calls `BankFiltering` to pick a new grant holder. `BankFiltering` looks at Slot 6, which belongs to B: here, B has no pending request, but not enough slack to conclude that a potential request from B will have its deadline postponed **(case 2 –** BF_2**)**. Hence, only requests to different banks than B's can be considered – which rules out d_1. c_1 gets picked by `mcEDF` over A_2, since its deadline happens earlier than A_2's. After issuing c_1's CWR, c_1 completes.

10 A new grant holder must be picked following the completion of c_1. By Rl_4, `BankFiltering` looks at Slot 6 and sees that *now* there *is* enough slack to conclude that any potential request from B will have its deadline pushed further into the future (to the end of Slot 8, specifically). This allows d_1, a write request to β_B to be considered for scheduling. Next, `mcEDF` picks d over A, since the former's deadline occurs earlier. Note that, since d_1 is a row-miss, a PAC sequence is necessary.

11 According to Rl_3, no shelving occurs, since B_3's deadline is only at the end of Slot 8. `BankFiltering` gets executed again and filters out requests that target A's bank (since it has an upcoming deadline) **(case 3 –** BF_3**)**. In this case, there aren't any, which means that all four pending requests (A_2, B_3, c_2, and d_1) are considered. Note that, different than c_1, which targeted β_A, c_2 targets another bank – β_C. d gets picked by `mcEDF`, since its deadline is the earliest.

12 Following the completion of d_1, as at 10, Rl_4 is applied, none of pending requests gets filtered out by `BankFiltering` **(case 3 –** BF_3**)** and `mcEDF` picks c, since its deadline is the earliest. Note that a PAC sequence is needed to service c_2, since β_C bank was not initially idle.

13 Since c_2 finishes at the end of Slot 6, the algorithm finds itself in the beginning of Slot 7 with no active grant holder (although this is not visible in the figure), which means that Rl_3 applies and a new grant holder must be picked at by `BankFiltering` and `mcEDF`. Since B, the owner of Slot 8, has a pending request, B_3, with an upcoming deadline **(case 3 –** BF_3**)**, only requests to a different bank than B's (or belonging to B) can be considered (which is the case for both A_2 and B_3). Because its deadline occurs earlier, `mcEDF` chooses A, which finishes after a PAC sequence and accumulates some small amount of slack.

14 Rl_4 is applied, `BankFiltering` looks at B and makes d ineligible, since d_2 targets B's bank. Therefore, the only available option for `mcEDF` is B, which gets picked next. Note that, since d accessed B's bank since B's last access, B_3 is now a row-miss and thus requires a PAC sequence. After finishing, B accumulates some slack.

Since similar rules apply for the remainder of the trace execution, we refrain from detailing the remaining steps.

Be mindful that the description above does not detail the behaviour of the algorithm during refresh phases. In summary, during refresh phases, the algorithm *halts*, meaning that all counters are frozen, including cycle, slot, and deadline counters. After the refresh phase is done, the algorithm resumes its processing normally, picking up from the state where it left.

9.2.5 Implementation

The CoqDRAM implementation of `TDMShelve` is complex, and it is not the goal here to explain how the implementation works in depth. Instead, the key points and insights are discussed, with the objective of giving the reader an intuitive understanding of how the algorithm uses the new interfaces oferred by in CoqDRAM.

The implementation begins by specifying the algorithm's parameters and assumptions, as shown in Listing 9.1. It is at this point that the assumptions/axioms about the minimum SL discussed in Section 9.2.2 are specified. The command `Context` introduces an "abstract instance" of `TDMShelve_configuration`, i.e., inside of `Section`, all definitions that follow expect an instance of `TDMShelve_configuration` as an implicit parameter.

Next, a requestor model must be specified (recall from Chapter 5 that requestor models are implementation-dependent). In the case of `TDMShelve`, requestors must have a numeric *ID*, a criticality, and, in addition, critical requestors must have an associated slot. The requestor model is shown in Listing 9.2. The numeric ID is modelled as *sigma*-type, i.e., a bounded `nat` (discussed in Chapter 5). Then, requestors are set to be a `Record` with four fields:

1. `ReqSlot` of type `option Slot_t` *may* bound a requestor to a TDM slot. We use `option` to model the criticality of requestors: if `ReqSlot` is built with `Some`, then the requestor is critical; otherwise, it is non-critical;

2. `ID`, of type `RequestorID_t` represents the requestor's numeric ID;

3. `CR_Slot` is a PO stating the fact that the ID of CRs is equal to their respective slot number;

Listing 9.1: TDMShelve parameters and assumptions.

```
Section TDMShelve.

Class TDMShelve_configuration := mkTDMCFG {
    SN : nat;
    SN_gt_1 : SN > 1;

    SL : nat;
    SL_pos  : SL > 0;

    (* The total number of requestors, including critical and non-critical *)
    Treq : nat;
    Treq_gt_SN : SN <= Treq;

    (* ----------- Necessary for TDM guarantees ------------ *)
    (* SL is big enough to accomodate a PAC sequence *)
    SL_enough : T_RP + T_RCD + 1 <= SL;

    (* Inter bank constraints *)
    SL_WR_to_RD : T_WL + T_BURST + T_WTR_l <= SL;
    SL_RD_to_WR : T_RTW <= SL;
    SL_CAS_to_CAS : T_CCD_l <= SL;
    SL_ACT_to_ACT_SB : T_RRD_l <= SL;

    (* Intra-bank constraints *)
    SL_ACT_to_PRE : T_RAS - 1 <= SL;
    SL_ACT_to_ACT_DB : T_RC - T_RP - 1 <= SL;
    SL_RD_to_PRE : T_RTP - 1 <= SL;
    SL_WR_to_PRE : T_WL + T_BURST + T_WR - 1 <= SL;
}.

Context {TDMShelve_CFG : TDMShelve_configuration}.
```

Listing 9.2: TDMShelve requestor model.

```
Definition RequestorID_t := {id : nat | id < Treq}.

Record TDMShelve_requestor := mkRequestor {
    (* CRs have an associated slot, NCRs requestors do not *)
    ReqSlot : option Slot_t;

    (* ID is useful to differentiate between NC requestors. For CR, it could have
       been only slots *)
    ID : RequestorID_t;

    (* PO: Requestor ID matches the slot number for CRs *)
    CR_Slot : forall slt, ReqSlot = Some slt → ' ID = nat_of_ord slt;

    (* PO: NCRs do not have associated slots *)
    NCR_Slot : forall slt, ReqSlot = None → ' ID >= SN
}.
```

4. `NCR_Slot` is another PO stating the fact that the numeric ID of NCRs must be greater than or equal than SN.

In the example from Section 9.2.4, the numeric IDs of A, B, c, and d are 0, 1, 2, and 3, respectively (and $SN = 4$). A and B are further mapped to slots 0 and 1, respectively.

Moreover, and very importantly, the implementation is based on the `Scheduler_t` interface, described in Listings 8.5 and 8.6. This means that the proofs about timing and protocol correctness completed via the methodology (partially) described in Chapter 8 are all automatically inherited. Or, in other words, the implementation is *guaranteed* to respect every functional and timing correctness PO from `Trace_t` that has been proved.

After having defined parameters and a requestor model, the next step towards writing an implementation of the `Schedule` function is to define a concrete internal state, i.e., to instantiate the class `SchedulerInternalState` by defining the type `SchState_t`, which will be used to store algorithmic "state variables". The internal state of `TDMShelve` is defined as shown in Listing 9.3. It is a record with the following fields:

1. `Slot`, type `Slot_t` (defined in Listing 6.8, from Section 6.2), is a TDM-like counter that wraps around $SN - 1$, used to account for slots.

2. `Counter` is a wrap-around counter (wrapping around $SL - 1$), used to keep track of cycles elapsed within a slot;

Listing 9.3: `TDMShelve` internal state.

```
Inductive Grant_t :=
    | NoGrant : Grant_t
    | SomeGrant : RequestorID_t → Grant_t
    | RefreshGrant : Grant_t.

Record TDMShelve_internal_state := mkTDMShelve_internal_state {
    (* Typical TDM slot and cycle counters *)
    Slot     : Slot_t;
    Counter  : TDMShelve_counter_t;
    (* Who has the grant to issue commands *)
    Grant      : Grant_t;
    (* Both critical and non-critical have associated deadlines *)
    Deadlines  : { deadline_cnts : seq.seq nat  | seq.size deadline_cnts = Treq};
    (* A list used to track the arrival of requests *)
    Arrivals   : { nc_arrivals : seq.seq bool  | seq.size nc_arrivals = Treq};
    (* Slack associated with critical requestors *)
    Slack      : { slack_cnts  : seq.seq nat   | seq.size slack_cnts = SN };
}.
```

3. `Grant` is the grant used to control the "right" to issue commands, as in the previous section. It is of type `Grant_t`, a custom inductive type that models three situations: no requestor holding the grant (`NoGrant`), a requestor holding the grant (`SomeGrant`), and a special type of grant reserved for refresh phases (`RefreshGrant`);

4. `Deadlines` is a set of counters used to keep track of deadlines. Note that `Deadlines` is also a sigma-type enforcing that there must be exactly T_{req} deadlines;

5. `Arrivals` is a set of boolean *flags* indicating, for each requestor, if there is a pending request. Like `Deadlines`, it is also a sigma-type enforcing the size of the set/sequence;

6. `Slack` is a set of counters used to keep track of slack for each critical requestor. Again, like `Deadlines` and `Arrivals`, `Slack` is a sigma-type enforcing the fact that there are exactly SN slack counters.

Implementing the command scheduling function is straightforward – we must simply filter the list of ready commands to consider only commands originating from the grant holder, as shown in Listing 9.4. If the grant is specifically

Note that `TDMShelve_schedule` performs a single pattern matching over `TDM_st.(Grant)`, i.e., the value of the grant during the *current* state, `TDM_st`. If no requestors holds the grant, the scheduler issues a `NOP`. Otherwise, the function `Pick_cmd_from_requestorID` is called,

Listing 9.4: Picking a new command from the grant holder.

```
(* Pick the first command of map coming from a given requestor ID *)
Definition Pick_cmd_from_requestorID
    (map    : ReqCmdMap_t)
    (req_id : RequestorID_t) : Command_kind_t :=
    let f := filter (fun cmd ⇒
        match cmd with
        | CRD r | CRDA r | CWR r | CWRA r | ACT r | PRE r ⇒ r.(Requestor).(ID) = req_id
        | _ ⇒ false
        end) map in seq.head NOP f.

(* Command scheduling function : an implementation of the Schedule interface
    from Listing 6.17 *)
Definition TDMShelve_schedule
    (map    : ReqCmdMap_t)
    (SS     : SystemState_t)
    (TDM_st    : TDMShelve_internal_state) : Command_kind_t :=
    match TDM_st.(Grant) with
    | NoGrant ⇒ NOP
    | SomeGrant requestor_id ⇒ Pick_cmd_from_requestorID map requestor_id
    | RefreshGrant ⇒ Pick_refresh_cmd map
    end.
```

which filters the list of ready commands to include only commands from a given requestor ID. The implementation of `Pick_cmd_from_requestorID` is also shown in Listing 9.4: it uses the `filter` function from `mathcomp`. The `filter` function expects an anonymous predicate as first argument: in the case of `Pick_cmd_from_requestorID`, we only keep commands whose associated request comes from the requestor whose ID is `req_id`. When the grant is `RefreshGrant`, a similar function, `Pick_refresh_cmd`, is used to pick the `PREA` or `REF` command from the pending command list. Moreover, since the command scheduling function is reduced to a simple filter over the list of ready commands, it is not hard to prove the POs `Schedule_Empty` and `Schedule_Cons` from Listing 8.6. The proofs of these POs are omitted here for brevity.

The core of `TDMShelve` takes place during the process of updating the internal state, and more specifically, updating `Grant`. More generally, at each clock cycle, a new instance of `TDMShelve_internal_state` must be produced. The `Scheduler` type class specifies an interface for the state update function (recall Listing 8.6). We reproduce its type signature here for practicality:

```
UpdateSchState :
    Command_kind_t (* command sent to the memory *) →
```

```
ReqCmdMap_t (* unfiltered list of pending commands *) →
SystemState_t (* DRAM device state *) →
SchState_t (* current state *) →
SchState_t (* future state *)
```

As it can be seen, the update function expects four arguments: 1) the command sent to the memory device at the subsequent clock edge (i.e., the output of the `Schedule` function); 2) the *unfiltered* list of commands waiting to be issued; 3) the system state (i.e., the state of the memory device); and 4) the current state – which is about to be updated.

It is important that the list of commands waiting to be issued includes *all* commands, and not just the ones that are *ready*. This is because the `Next_SL_state` function *replaces* non-ready commands by `NOP` commands – which makes requests belonging to non-ready commands invisible to the state update function (see Algorithm 1). In summary, the state update function must be capable of "*seeing*" all requests in the system, including the ones belonging to non-ready commands.

To create a new instance of `TDMShelve_internal_state`, each field must be updated according to a specific function. `Slot` and `Counter` are updated straightforwardly – as they are similar to the counters used in strict TDM. As a reminder, `Counter` gets incremented by 1 at each clock cycle and wraps around $SL - 1$. `Slot` gets incremented by 1 whenever `Counter` reaches its maximum value, itself wrapping around $SN - 1$. `Deadlines` and `Slack` are updated according to the same rules proposed by Hebbache et al. [40].[8] This rules have been briefly discussed in Section 9.2.3. For brevity, we choose not to show the implementation of these functions/rules/ Moreover, `Arrivals` is updated in the following manner: whenever a command in the list of *all* pending commands is observed for the first time, its corresponding requestor's flag in `Arrivals` is set to `true`, and is only reset to `false` when the algorithm schedules a CAS belonging to such requestor. Furthermore, when the algorithm finds itself in the refresh phase, the state is no longer updated, except for the grant itself. The grant goes into refresh "mode" whenever a `PREA` command is seen on the bus and leaves it whenever a `REF` is seen on the bus.

Finally, the `Grant` update function is where the core of `TDMShelve` actually happens. Listing 9.5 shows the implementation of `UpdateGrant`. From Lines 6 to 9, we "access" individual elements of the current state. At Line 8, specifically, we use the function `getSlackFromSlot` to access the slack counter relative to the current TDM slot (keep in mind that each slot is associated to a requestor, therefore, it is also associated to a slot counter and a deadline).

Next, at Line 10, we test if the algorithm has been called at the beginning of a TDM

[8]Naturally, the CoqDRAM implementations of such algorithms/rules are different than the hardware design proposed by the authors.

Listing 9.5: Implementation of `UpdateGrant` – a function implementing TDMShelve's core.

```
Definition UpdateGrant
    (u_map : ReqCmdMap_t) (* The unfiltered list of pending commands *)
    (sch_cmd : Command_kind_t) (* The command chosen by TDMShelve_schedule *)
    (TDM_st : TDMShelve_internal_state) (* The current internal state *)
    : option RequestorID_t (* The result -- the grant holder *) :=
    let slt := TDM_st.(Slot) in (* The current slot *)
    let grant := TDM_st.(Grant) in (* The current grant holder *)
    let cur_slack := getSlackFromSlot TDM_st slt in (* The slack associated to slt *)
    let cnt := TDM_st.(Counter) in (* Current cycle *)
    if (cnt = OZCycle) then ( (* Beginning of slot *)
        match grant with
        | NoGrant ⇒ (* No requestor currently holds the grant *)
            if (u_map = [::]) (* No pending requests *)
                then NoGrant (* No one gets the grant → Rule 2 *)
            else (* Call BankFiltering to pick new grant holder → Rule 3 *)
                (BankFiltering u_map TDM_st)
        | SomeGrant requestor_id ⇒ (* Some requestor currently holds the grant *)
            (* checks if current slot has a pending request *)
            match checkIfPendingFromSlot u_map slt with
            | false ⇒ (* current slot has no pending request *)
                (* if a request is just completing, the grant gets reset → Rule 2 *)
                if (isCAS_cmdkind sch_cmd) then NoGrant
                else (* call BankFiltering to choose grant holder → Rule 3 *)
                    (BankFiltering u_map TDM_st)
            | true ⇒ (* current slot has a pending request *)
                    let preq := Slot_to_RequestorID slt in
                    (* the deadline of the pending request *)
                    let preq_deadl := getDeadlineFromID TDM_st (‘ preq) in
                    if (preq_deadl <= SL) (* deadline is upcoming *)
                        then (SomeGrant preq) (* have to shelve ! → Rule 1 *)
                        else (if (isCAS_cmdkind sch_cmd) then NoGrant (* Rule 2 *)
                            else (BankFiltering u_map TDM_st) (* Rule 3 *))
            end
        | RefreshGrant ⇒ RefreshGrant (* cannot happen *)
        end
    ) else ( (* Middle of the slot → Rule 4 *)
        match grant with
        | NoGrant ⇒ (* no requetor holds the grant *)
            if (u_map = [::]) then NoGrant
            else (BankFiltering u_map TDM_st) (* pick new grant holder *)
        | SomeGrant requestor_id ⇒ (* some requestor already holds the grant *)
            if (isCAS_cmdkind sch_cmd) then NoGrant
            (* keep processing commands from current grant holder *)
            else (SomeGrant requestor_id)
        | RefreshGrant ⇒ RefreshGrant (* cannot happen *)
        end).
```

slot (i.e., `cnt` = `OZCycle`). There are many nested sub-cases, and each sub-case fits into Rl_1, Rl_2, or Rl_3 – from Section 9.2.3. At Line 11, the algorithm tests if some requestor holds the grant. If not, then it proceeds to test if there is at least one pending request. To test such condition, we must check if `u_map` is empty. Remember that `u_map` is the **unfiltered** version of the command map – containing every pending command (and consequently every pending request). If there are no pending requests, no requestor will hold the grant (Line 14, (Rl_2)). If there is at least one pending request, then `UpdateGrant` calls `BankFiltering` to choose a new grant holder according to the rules discussing in Section 9.2.3 (Line 16).

If at the beginning of `slt` (the current TDM slot) a requestor (`requestor_id`) already holds the grant (Line 17), then the algorithm proceeds by checking if the **current** slot has an outstanding request (Line 19). To do so, it calls `checkIfPendingFromSlot`, a function that uses the `find` function from `mathcomp` to check if there is at least one command in the unfiltered pending command list whose associated request comes from the requestor associated to `slt`. In other words, the function checks if there is a pending request associated with the current TDM slot. If there is not (Line 20), then two cases are possible: if the scheduled command (`sch_cmd`) – chosen by the command scheduling function on that same clock cycle – is a `CAS`, this means that the processing of the current request has completed and the grant gets reset to `NoGrant` (Line 22). If, however, that is not the case, then the algorithm calls `BankFiltering` to pick a grant holder (Line 24). Note that it would make sense to simply continue to process the pending request from `requestor_id`, since it already holds the grant. The problem is that, since the algorithm finds itself at a new slot, the `BankFiltering` test must happen again, since blindly continuing to process `requestor_id` may compromise the TDM bounds for the next slot owner.

Consider now the case where the owner of `slt` does have a pending request (Line 25). In that scenario, we check if the deadline of such request is **upcoming**, i.e., is due to the end of `slt`. If that is indeed the case (Line 29), then `requestor_id` must be shelved and the algorithm starts processing `preq` – the priority requestor, for it to meet its deadline (Line 30, (Rl_1)). If the deadline of `preq` is not due at the end of `slt`, then the previous rules apply: if the processing of `requestor_id` has just finished, the grant gets reset (Line 31), otherwise, `BankFiltering` is called to pick a grant holder (Line 32).

Finally, at Line 37, the algorithm finds itself **not** at the beginning of a TDM slot. In that case, no shelving happens, i.e., the algorithm simply looks at the grant holder. If no one holds the grant and there is no pending request, then no requestor gets the grant (Line 39). If there is at least one pending request, then `BankFiltering` chooses a new grant holder according to its rules. If some requestor (`requestor_id`) does hold the grant, the algorithm resets it if the request currently under processing has finished (Line 42). Otherwise, the

processing of `requestor_id` simply continues (Line 44). For conciseness, we choose not to show the implementations of `BankFiltering`, `mcEDF`, and other auxiliary functions.

To conclude, `TDMShelve` shows that CoqDRAM can be successfully used to model complex scheduling algorithms. At time of writing, the implementation described above has been subjected to some basic sanity tests, which guaranteed basic functionality. However, since the second iteration of the framework is still work-in-progress, many properties are yet to be proved. As mentioned in Chapter 8, regarding timing proofs, only t_{RCD} has been proved. Protocol correctness proofs are also left undone, for now. The high-level properties, such as fairness are also unproven. Beyond that, it is an important objective to formalise (and prove) a theorem stating that the WCL latency of critical requests is bounded by the deadlines of strict TDM. Furthermore, the impact of refreshes on scheduling has yet to be taken into account in the algorithm's design.

Chapter 10

Discussion & Conclusion

In this chapter, we reflect on what has been accomplished, highlight the strengths and limitations of our approach, and conclude by analysing promising future work directions.

10.1 Discussion

CoqDRAM is a framework that can be *realistically* used to design *trustworthy* state-of-the-art memory controller scheduling algorithms. Although the high entry bar still represents a significant barrier for adoption, given the steep learning curve of Coq, we firmly believe that adopting the techniques proposed in this dissertation have the potential to increase the overall trust in memory controller designs.

The latter conclusion comes from several observations. First, developing algorithms in such a manner requires one to formally state every single assumption and hypothesis that a given design relies upon. In other words, while proving properties about algorithms in Coq, no hand-waving or oversimplification is allowed, which is often the case in paper-and-pencil proofs. Moreover, even when one cannot actually finish a proof, the mere exercise of attempting a proof can be seen as a sort of *bug hunting*. In some cases, parts of the proof can be axiomatic, and the process of writing such axioms already constitutes a kind of formal reasoning, forcing the developer to think thoroughly about his or hers design.

In addition, dependent types and type checking also enforce a great amount of correctness. In my personal experience, I found that often times, after successfully passing type checking, the design was indeed correct, or very close to be correct – which made proving an easier task. Furthermore, in CoqDRAM, we sometimes exploit dependent types, which further imposes correctness at type checking time. Although dependent types are doubled edged swords – in the sense that they effectively enforce correctness but are complex to work with – they do prevent an awful lot of problems that could be present otherwise. Typical implementation

bugs, such as out-of-bound array accesses or overflows can be ruled out. Invalid algorithmic behaviour can also be ruled out, but again, this is treacherous terrain – as proofs about programs including complex dependent types require more sophisticated techniques.

Another argument that greatly supports our *trustworthiness* claims is that the *semantic gaps* between standards, formal specification, and implementations are smaller than the ones found on standard paper-and-pencil approaches. Lets consider first the link between standards and a formal specification: while related work has tackled such problems using other techniques (see Chapter 3), the expressiveness of Coq's logic allows us to reproduce statements from the JEDEC standards written in natural language (English) in a very natural form, which is arguably easier to check then properties written in temporal logic, timed automata, or other.

Second, and most importantly, consider the link between formal specification and implementation. In the typical paper-and-pencil approach, the only thing that "links" an actual algorithm implementation to paper-and-pencil mathematical developments is simulation: usually, the algorithm is simulated a sufficient number of times, and observations from such experiments are expected to match the theoretical/mathematical model. While this is not invaluable, testing has its inherent limitations – nothing guarantees that corner cases that could invalidate the theoretical model have been exercised by the test-benches. Using the Coq approach, this is not a problem, since the algorithm implementation is directly linked to the theoretical model by the proof itself, i.e., there is no *gap* between implementation and formal specification.

Furthermore, we claim that adopting the CoqDRAM approach greatly increases readability, easiness of reviewing, and reproducibility. As discussed in Chapter 3, paper-and-pencil mathematical developments found in literature are hard to read and review. It certainly makes more sense to present proofs of correctness as machine-checkable artefacts, which moves the burden of checking a proof from humans to the machine. Reviewers (and readers, more generally) are thus left with a higher-level task: checking that properties have been correctly stated. This correlates with an important point recently raised by Leveson and Thomas [120]: most bugs found in modern safety-critical software or hardware come from poorly written specifications rather than faulty implementations. The greater reproducibility argument comes from the observation that spending less space on proofs in a paper leaves more space to discuss high-level explanations, engineering aspects, and experiment's results and methodologies.

Yet another advantage of CoqDRAM is that proved code can be extracted to other languages (e.g., Haskell and OCaml), which can further be used in external environments. As we have discuss in Chapter 7 (and also in Appendix A), we use this mechanism to validate

the CoqDRAM algorithms in both software and hardware simulation. While the software simulation experiment uses Haskell code generated from Coq, the hardware experiment relies on a translation script from Cava, which translates Cava/Coq-generated Haskell code to SystemVerilog (for more information regarding the latter, refer to Appendix A). Many formal method approaches still ignore this link between formal model and code that can be deployed in the real world.

As a final argument, Coq allows us to use powerful abstractions to prove properties about systems with virtually no size constraints. As discussed in Chapter 2, while the scalability of model checking techniques keeps getting better, the approach is still limited. In CoqDRAM, not only things such as timing constraints and number of banks can be parameterised, but also functions and types, such as an abstract arrival model and a representation of requestors. These powerful abstractions allow designers to design systems that are generic, scalable, and highly re-usable.

Limitations

It is also important to state the limitations of our approach. As mentioned previously, Coq is a large system with a great amount of features and non-trivial concepts. Its learning curve can be quite steep, which disencourages potential users. The good news on that regard is that this problem has been properly identified and partially addressed during the last few years. Currently, great resources allowing a gentler introduction are available, including Pierce's Software Foundations [121], Chlipala's CPDT [78] and *Formal Reasoning About Programs* (FRAP) [122].

It is also worth mentioning that proving properties can also be a time-consuming task, even when they initially appear to be straightforward. In CoqDRAM, we make an effort to provide some custom tactics for helping users to automatically solve *some* goals, but this is yet far from great. However, as mentioned previously, incomplete/axiomatic proofs may also carry value – if they are done in a modular/traceable way.

Furthermore, CoqDRAM only covers the DDR3 and DDR4 JEDEC standards. Although we do include a brief discussion about other memory standards in Chapter 5, the extent to which the framework could be easily adapted to model other standards is unexplored territory. This is an important problem, since many modern computing systems use other memories rather than general purpose DDRs. This includes *High-Bandwidth Memory* (HBM), LPDDR, GDDR, among a few others.

Closing the loop with the initial PhD project idea

Having designed TDMShelve as a new, state-of-the-art DRAM scheduling algorithm in CoqDRAM is satisfying – since we were able to close a loop w.r.t the initial research project (stated in the funding proposal) and address the research questions stated in Chapter 1.

More concretely, the proposal mentioned two main research questions/focus points, which come as limitations of Hebbache et al.'s original algorithm [39], [40]: 1) Address the information leakage that comes as a consequence of breaking the strict time isolation between requestors; and 2) Develop techniques that allow exploiting the slack information across the memory hierarchy and further improve the approach's efficiency.

While we did not have the time to cover the security aspects nor other parts of the memory hierarchy, it is safe to say that the design of `TDMShelve` successfully exploits the slack information at the DRAM command scheduling level. Beyond the proposal, our (initially unplanned) focus on formal verification allowed us to establish a solid foundation to develop algorithms that are *safe* and *reliable*.

Moreover, our most recent improvements emphasise the framework aspect, which means that new developments relying on the machinery described in Section 8 can inherit several proofs about standard compliance, for example.

10.2 Future Work

The work presented in this dissertation holds potential for exploration in numerous promising directions. Some of these directions are short-term goals – which are, essentially, objectives that could not be fully achieved due to time constraints. Other directions are more of a "long-shot", i.e., pointers towards promising new explorations building on this work.

Short-term

The most interesting pointer towards short-term future work is arguably to **work on the implementation and proof of state-of-the-art real-time memory controllers**. Notably, `TDMShelve`, presented in Chapter 9, is implemented but remains (mostly) unproved. A straightforward direction would thus be to prove high-level properties: most importantly, to formalise the already existing proof [40] that **TDM bounds are guaranteed for critical tasks**. Proving other high-level properties such as non-starvation and sequential-consistency for TDMShelve can also be seen as future work.

Moreover, it would be interesting to implement and prove other real-time memory controllers from the literature. Some good options to start with, presented from least to most

recent, would be: *Analysable Memory Controller* (AMC) [20]; Private Bank Open Row Policy (ORP) [24]; Dynamic Command Scheduling MC (RTMem) [27]; Rank Switching Open Row MC (ROC) [26]; Dual-Criticality MC (DCmc) [18]; Read/Write Bundling MC (ReOrder) [33]; Mixed Critical MC (MCMC) [25]; Programmable MC (PMC) [35]; DRAM-bulism [17]; and DuoMC [104]. Modelling these controllers in CoqDRAM would allow us to evaluate the effectiveness of the framework to formalise and prove more complex algorithms and their underlying mathematical developments.

Before tackling the modelling and proving of algorithms such as the ones mentioned above, it would be highly convenient to invest some time in the automation part of the framework, i.e., developing custom tactics or other sources of automation that would expedite and facilitate the proving part.

Furthermore, a couple of other short-term objectives are listed below. These correspond to tasks that could not be finished mainly due to time constraints and change of focus in the research.

- So far, only the T_{RCD} proof has been completed for the bank machine approach, described in Chapter 8. Proving every other timing constraint stated in `Trace_t` is left as future work. Fortunately, if someone sets to prove such properties, a lot can be taken from the already completed proof – since these proofs are rather repetitive. The ones that can be more burdensome are refresh related proofs, since none of these have been completed yet.

- As presented in Chapter 5, the most up-to-date version of the framework only includes one mandatory proof obligation for all arbiters: that requests must eventually be serviced. Moreover, we model two other arbiter interfaces corresponding to the two flavours of sequential consistency proposed by Lamport [111]. It is considered future work to define other classes of arbiters. An example would be an interface for mixed-criticality arbiters defining different requirements (proof obligations) for requests, depending on the criticality of their requestors. The latter would be the exact interface required by `TDMShelve`, for example.

Long-term

A first interesting long-term direction would be to keep building on the framework to model other important aspects of memory controllers. For instance, one could model features like handling of data dependencies between memory requests and implementation of atomic operations. Other things that could be modelled are parity bits over the command and address buses and error correction codes (ECCs) over the data bus. These latter features are

important parts of the JEDEC standards and are relevant for guaranteeing data integrity. It would thus be great if CoqDRAM could guarantee that the implementation of such features is correct against their specification in the JEDEC standards.

Another interesting pointer would be to mimic the CoqDRAM approach to model other parts of the memory hierarchy. The methodology described in this dissertation could be taken as a blueprint to model cache coherency protocols [99], protocols for shared buses and interconnects, and the logic of *Memory Management Units* (MMUs) likewise.

As mentioned previously, the thesis proposal also emphasised the *security* aspect. Given the increasing need for security in modern safety-critical real-time systems, CoqDRAM could be extended to model security properties at different levels. On one hand, one could model high-level properties, such as the absence of side-channel information leakage in scheduling algorithms. On the other hand, one could also use the framework to ensure that counter-measures are effectively implemented (i.e., are functionally correct). As one example of the latter case, CoqDRAM could enforce a defence mechanism against Rowhammer [123] – in which memory accesses are monitored and unusual patterns are detected and forbidden.

Moreover, another promising direction is to connect the algorithms written in CoqDRAM to equivalent hardware counterparts. In Appendix A, we describe an exploration in that direction. However, the methodology adopted in such exploration did not yield satisfactory results, mainly due to technical limitations of Cava – the Coq *Domain Specific Language* (DSL) used to model the hardware. The problems we encountered with Cava can possibly be solved by choosing another existing Coq DSL to model hardware, such as Kami [96] or Kôika [98]. Therefore, as future work, it would be interesting to see if an equivalence could be established between a CoqDRAM algorithm and a hardware controller written in these languages – in a way that is practical.

Furthermore, it would be interesting to explore the idea of going in the opposite direction, i.e., from a given memory controller written in an HDL (e.g. SystemVerilog) to Coq. More specifically, it would be interesting to see if both existing and new DRAM controllers written in Verilog (or other HDLs) could "straightforwardly" conform to the CoqDRAM specification. This direction has been explored by Bidmeshki & Makris in the VeriCoq project [124] – a Verilog to Coq converter. However, the VeriCoq conversion is not itself formally verified, which means that one cannot ensure that the Verilog module and the Coq representation are indeed equivalent. Morever, VeriCoq only supports a limited subset of Verilog, e.g., "generate" blocks and functions/tasks are not supported, which are employed a lot in practice [125]. The idea is nevertheless promising, since recent advances have been proposed by Choi [125] – who introduces formal *denotational semantics* of synthesizable Verilog in Coq. Yet another idea would be to use automated equivalence checking (with model-checking) to compare an

HDL version of a memory controller against a Coq version, in a way similar to what has been proposed by Harrison et al [100], [101]. Harrison's work, however, checks HDL designs against designs written in the ReWire DSL – embedded in Haskell. In order to implement a similar approach, we would require CoqDRAM designs to be somehow translated to a sub-set of Haskell that can be used in a model checker.

Finally, we could leverage DRAMml – a DSL used to model bank state transitions and timing constraints from different JEDEC standards – proposed by Jung, Steiner et al [80], [83] (see Chapter 3). In that work, DRAMml has been used to generate an executable SystemC model, which is used to perform simulation-based validation of an RTL memory controller [126] and a few DRAM simulators. As the authors mention, DRAMml models could also be converted to other languages. By converting DRAMml into Coq proof obligations, for example, one could automatically generate a CoqDRAM-like specification for many different memory standards – not just DDR3 and DDR4. Fortunately, the JEDEC standards from Figure 5.2 have already been described in DRAMml. Other (newer) standards would have to be described manually. This is arguably of the most interesting future-work directions – leveraging the results of two parallel research projects.

Appendix A

What About the Hardware? – An Exploration

While CoqDRAM is useful for modelling, exploring, and proving the correctness of DRAM scheduling algorithms, it remains a high-level, abstraction-rich model. Memory controllers, in reality, are hardware components, typically designed using lower-level *Register Transfer Level* (RTL) description languages. Therefore, for algorithms developed in CoqDRAM to be *relevant in real life applications*, it is necessary to somehow connect them to equivalent hardware models that can be synthesizable in an FPGA or an ASIC.

In this chapter, we present an *exploration*, i.e., a research effort that attempted solving the problem described above. While many options were possible, the approach described in this chapter consists in writing circuits that implement the scheduling algorithms in a DSL embedded in Coq, prove equivalence against the CoqDRAM model, and then "compile" the algorithm written in the DSL to SystemVerilog. The DSL we use is called **Cava**. More specifically, we develop a *design pattern* and a library in Cava – which we call *CavaDRAM*. Other approaches on how to connect hardware designs to higher-level models were discussed in Chapter 3.

Throughout this exploration, we were able to implement a couple of memory controllers, translate them to SystemVerilog, and validate their behaviour through assertion-based functional simulation on a third-party simulation environment. Then, we stated a theorem that establishes behavioural equivalence between these controllers and their counterpart CoqDRAM algorithms. Although the idea of linking CoqDRAM algorithms to an equivalent model that can be translated to RTL and synthesised into a netlist is an interesting and relevant problem, at some point, we decided to cease pursuing this research effort, due to several technical limitations of Cava. The exact limitations of Cava are discussed in detail in Section A.6. In summary, even if these experiments led to some achievements and

many lessons learned, I would not recommend using Cava for the kind of design/proof we attempted. As previously discussed in Chapter 10, using another Coq DSL for hardware design could *possibly* yield more satisfactory outcomes.

This appendix starts by presenting a comprehensive overview of Cava – since no paper, or even manual exists that describes Cava in detail. Then, we explain our methodology, present the outcomes from the exploration, and finishes by exposing what did not work and what were the lessons taken from the experiment.

A.1 Domain Specific Languages (DSLs)

In contrast with general-purpose languages, such as C, Python, Java, and Gallina (Coq's programming language), a *Domain Specific Languages* (DSL) is a computer language specialised to a particular application domain. While the line dividing the definition of a DSL and a general-purpose language is not always sharp, DSLs are usually smaller and have a reduced syntax that is just sufficient and expressive enough for the target application.

One interesting research direction that saw light in the late 2010's was to *embed* DSLs for hardware design in Coq. Coq, in this case, serves as the *host language*. One of the main advantages of using a system such as Coq to host a DSL is how convenient it is to define precise *syntax* (i.e., how terms of the language can be put together to form valid *expressions*) and *semantics* (i.e., what is the precise meaning of each element of the language). Specifically, semantics defined in a formal deduction system like Coq can be used to do proofs and to reason about correctness; and well-formed expressions (i.e., expressions that are syntactically correct) can be enforced by type-checking.

Actually, these concepts can be used in a broader context: we can talk about syntax and formal semantics of any source language **S** described in a host language **T**. Since we are interested in formal reasoning and proofs, we restrict the discussion to the case where **T** is a theorem prover (such as Coq, Lean, F*, or Isabelle). **S** can be a DSL, as mentioned previously, but also a full-fledged general-purpose programming languages, such as C [127]; or even other logic/calculi, such as a simply typed lambda calculus, separation logic [128], Lean [129][1], and HOL Light [130].

Before presenting some examples, it is important to discuss *how* the source language **S** can be described in the host-language – since there can be significant differences on how to reason about the source language based on the choices made. If terms of **S** are translated *directly* to the terms of **T**, then we say that the embedding of **S** in **T** is **shallow**. This is useful when one wants to simply *use* **S** and compute with it. Moreover, such technique only

[1] https://coq.discourse.group/t/alpha-announcement-coq-is-a-lean-typechecker/581

requires programmers to "translate" types and operations of **S** into types and operations of **T**, without explicitly specifying the syntactic rules for constructing terms of **S**.

Alternatively, one can represent the source language **S** as an Abstract Syntax Tree (AST). In Coq, for instance, these ASTs can be built using inductive data-types to represent terms of the source language according to its syntactic structure. We say that **S** is **deeply embedded** in **T**. A deep embedding allows **S** to be studied as a formal system and the users of such an embedding to do proofs about **S** itself, and not just *use* **S**. For example, if **S** is a programming language and **T** is a full-fledged proof assistant, we can define in **T** types that represent the syntax, the typing rules, and the operational semantics of **S** [131].

In summary, on the one hand, a deep embedding consists in representing expressions in a language as a data-structure, which can then be *interpreted* in different ways. A shallow embedding, on the other hand, represents a language directly as terms in the host language, bypassing the data-structure.

The CompCert compiler [127], for example, is a verified C compiler (and perhaps the most well known large-scale Coq development), in the sense that machine code generated by CompCert is guaranteed to preserve the semantics of the compiled C code. It is evident that such semantic preservation property depends on well-defined semantics for the C language. Therefore, in order to define such semantics, CompCert includes a **deep embedding of the (almost complete) C language** (ISO C 2011) [132]. The formalised semantics of C in CompCert can even be re-used by other tools, such as CertiCoq [133] – a verified compiler from Gallina to C light (a subset of the C language containing only *pure* functions, i.e., with no side-effects). Even Coq itself has been deeply embedded in Coq – an achievement part of a larger project called MetaCoq [134] in which researchers focus on formalising Coq's theory.

Other research projects have focused on the formal semantics of machine-code, or *Instruction Set Architectures* (ISAs). Sail [135], for instance, has been used to model the semantics of ARMv8, MIPS, CHERI, IBM-Power, and RISC-V. Sail ISA definitions are ASTs (i.e., deep embeddings), which can be translated to the language accepted by theorem provers and transformed into textual documentation, among other options. Researchers at MIT have also (manually) formalised the RISC-V semantics in Coq [136] – claiming that machine-generated theorem prover definitions (such as the ones generated by Sail, for example) are hard for humans to work with. Moreover, recent research at Microsoft developed a deep embedding of the Intel x64 instructions in F* [91] – whose formal semantics is used to prove the functional correctness of low-level implementations of cryptographic algorithms [137].

Some examples of shallow embeddings include Low* [138] – a sequential, well-behaved subset of C in F*; the project "Coq is a Lean Typechecker"[2] – which includes a direct

[2] https://coq.discourse.group/t/alpha-announcement-coq-is-a-lean-typechecker/581

translation of some Lean types to Coq types; and an encoding of HOL Light in Coq [130].

Coming back to the topic of DSLs and hardware design, recent research focused on developing embedded languages, which allow designers to *write circuits with formal semantics*. Circuits written in such a fashion can be proven correct: one could, for instance, prove that circuits behave as known mathematical functions or even prove that they *implement* instructions from an ISA with formalised semantics. In fact, one could go as far as writing a full-fledged processor implementation and proving its correctness against the underlying ISA – if the semantics for the latter have been formalised as well.

A.1.1 Cava

More specifically, in this work, we explore the use of *Cava* [139],[3] a DSL developed by Google Research back in 2020. Cava's design was greatly inspired by Lava [140]. Moreover, Cava was developed as part of the *Silver Oak project*, which focuses on the verification of high assurance components of the OpenTitan[4] silicon root of trust, i.e., a set of inherently trusted functions within a platform. Cava follows an approach inspired by the vision set out by Adam Chlipala in Certified Programming with Dependent Types [78]. Following that same approach, other Coq DSLs for designing circuits have been proposed in parallel or preceding research projects, such as Kami [96] and Kôika [98]. In summary, development with all these DSLs follows the same kind of procedure [96]:

1. Implement (write) the circuit in the DSL;
2. In a rich higher-order logic, state the most natural correctness theorem for the circuit;
3. Prove the theorem using scripts of tactics;
4. Use *extraction* to translate the program to a *Register Transfer Level* (RTL) language like SystemVerilog or VHDL automatically, and from here use standard development tools to synthesise and deploy it in an FPGA, for example.

In this work, we choose Cava over Kami [96] and Kôika [98] – the choice being motivated by the following arguments: 1) Cava circuit simulations generate a list of values, where each element represents the value of a wire at a given clock cycle – this emulates *time*, a key element relevant for this work; 2) Cava is relatively simpler and faster to get acquainted to; and 3) Cava designs resemble classic RTL design style in some sense, whereas other DSLs take an approach closer to the rule-based design of Bluespec SystemVerilog [141]. Section 3.2

[3] https://github.com/project-oak/silveroak
[4] https://opentitan.org/

discusses other hardware designs DSLs in more detail. Next, we present a brief introduction to some of the key concepts in Cava.

A. Multiple interpretations

Cava is a DSL embedded into Coq combining both shallow and deep embedding techniques. Cava works in a way such that circuit definitions are overloaded, i.e., circuits can be *interpreted* in different ways. More formally, we say that different *semantics* can be given to a single circuit definition. Actually, this is not something exclusive to Cava, the idea of having multiple interpretations is a natural follow-up to the concept of a *deeply-embedded* DSL [142].

The deep embedding aspect of Cava is (mainly) what allows circuits to accept different semantics and thus be interpreted both in the world of Coq, where proofs and testing can take place; and in the world of *netlists*. A netlist, simply put, is a *gate-level* description of an RTL design. It is a sort of graph connecting different blocks (i.e., elements of logic – ranging from simple gates to predefined FPGA blocks) through wires. A netlist is the result of *synthesising* an RTL description.

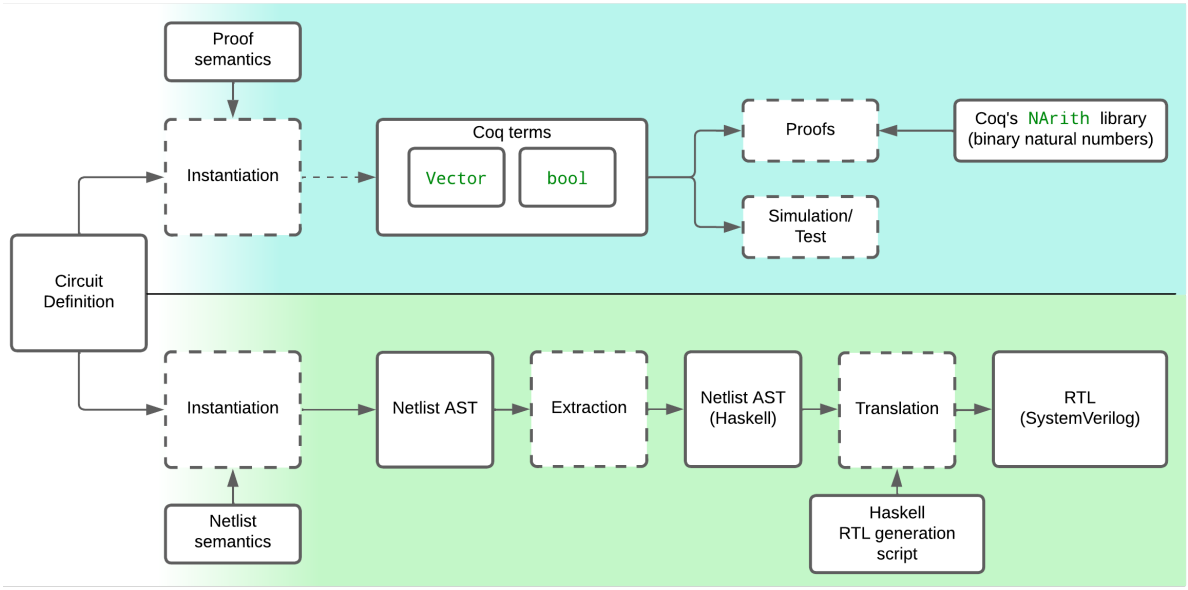


Figure A.1: The Cava design – circuits can have multiple interpretations.
Legend: ⬚ Actions, ☐ Artefacts.

Figure A.1 presents a visual scheme for better understanding the concept of *multiple interpretations* in Cava. A single circuit definition, when given semantics, can be transformed into Coq terms or into yet another data-structure – an AST representing a netlist.

The proof semantics (also called "*combinational semantics*" or "*simulation semantics*" in Cava) transforms native Cava types into Coq terms of type `bool` and `Vector`. These terms

can later be used in proofs or to perform testing/simulation within Coq. In the former case, one might, for instance, want to prove that the bit-vectors produced by circuits are equivalent to the arithmetic operations over binary natural numbers – described in Coq's `BinNat` and `NArith` libraries.

Regarding the flow from Cava to SystemVerilog, depicted in the bottom part of the figure, after translating the Cava circuit to a netlist AST, the Coq code is extracted to Haskell, and an additional algorithm can finally parse the AST and generate the corresponding SystemVerilog description. Both flows are better detailed at the end of this section.

B. Writing circuits in Cava

Concretely speaking, writing circuits in Cava evolves around three main components: 1) a shallow embedding allowing users to write *combinational*[5] circuits; 2) an inductive datatype implementing *sequential* circuits; and 3) a collection of higher-order and (sometimes) dependently typed constructs, which together allow users to make full use of Coq and functional programming to design complex circuits.

In the following, we start by discussing Cava's primitive types and then briefly go through the three main parts listed above. The code snippets shown in the following are part of Cava and can be found in the SilverOak repository [139]. Moreover, terms highlighted in `orange` are native Cava functions or types (or native Cava definitions, in general).

B.0. Signals. Cava relies on a deep embedding in the sense of *interpreting* signals. Concretely, an inductive type `SignalType` models the values that can "flow through wires" in a Cava circuit, shown in Listing A.1. Terms of type `SignalType` can be used to design circuits, regardless of semantics.

Listing A.1: Definition of `SignalType`.

```
Inductive SignalType :=
  | Void : SignalType                        (* An empty type *)
  | Bit : SignalType                         (* A single wire *)
  | Vec : SignalType → nat → SignalType      (* Vectors, possibly nested *)
  | ExternalType : string → SignalType.      (* An uninterpreted type *)
```

When writing concrete circuits, an assumption is made that there exists *some interpretation* of `SignalType` available – which can be abstract at the time circuits are defined. Or, in other words, there must be *some* function that translates Cava types into Coq types (i.e., a function of type `SignalType → Type`). At the time that circuits are written, such function

[5] We use the terms *combinational* and *combinatorial* circuits interchangeably to denote circuits that do not have *state*.

is merely an implicit argument, often called `signal`, and a concrete implementation to such function is only needed for proofs/simulation or code generation.

B.1. Combinational Circuits. The *operations* that can be used over signals/wires are defined in a way that resembles a shallow embedding. This means that these operations are not defined in an inductive type, but rather listed as member functions of a type class, and different semantics must eventually implement each function accordingly (recall the container interface presented in Section 2.3.1). Table A.1 lists some of the constructs provided by Cava to write combinational circuits.[6]

Cava function	Description
`constant`	A wire that is constantly evaluated to the same value
`constantV`	A wire of arbitrary width constantly evaluated to the same bit-vector
`inv`	Boolean NOT
`and2`	Boolean AND
`nand2`	Boolean NAND
`or2`	Boolean OR
`nor2`	Boolean NOR
`xor2`	Boolean XOR
`xnor2`	Boolean XNOR
`buf_gate`	Corresponds to the SystemVerilog primitive gate "buf"
`lut1`	1-input LUT
`lut2`	2-input LUT
`xorcy`	Xilinx fast-carry UNISIM
`muxcy`	Xilinx fast-carry UNISIM
`indexAt`	Dynamic indexing for bit-vectors

Table A.1: Some of the functions available to write combinational circuits.

As it can be seen, operators include constant values (1-bit or bit-vectors of arbitrary width), primitive logic gates, *look-up tables* (LUTs) (i.e., blocks that implement the logic of a certain boolean function, passed as argument), special operators from Xilinx' UNISIM library, and an indexing function – which takes two bit-vectors as arguments, the first one being used to index the second one. The existence of some of these operators is justified by the desire of having the flexibility to design circuits that can be synthesised using specific SystemVerilog constructs: `xorcy`, for example, is the same as `xor2` when applying proof semantics, but different when considering netlist semantics.

As an example of combinational circuit, consider a *ripple carry adder*, a circuit that performs the arithmetic sum of two positive binary numbers of arbitrary length. We can

[6]For brevity, Table A.1 does not list all available operators.

Inputs		Outputs	
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table A.2: Half adder truth table.

Listing A.2: Half adder circuit in Cava.

```
Definition halfAdder '(x,y : signal Bit) : cava (signal Bit * signal Bit) :=
    sum ← xor2 (x, y) ;;       (* calculates the Sum bit *)
    carry ← and2 (x, y) ;;     (* calculates the Carry bit *)
    ret (sum, carry).          (* circuits returns a pair *)
```

begin by first defining a half adder, which has its truth table shown in Table A.2. It should be clear that "Carry" works like an "overflow" bit: if both A and B are 1, for instance, then the result, in decimal, would be **2**, but since Sum is only 1-bit wide, 2 cannot be represent. Hence, the carry bit signals that an overflow has occurred. Note that Sum is simply a logical **XOR** between A and B, and Carry is simply a logical **AND** between A and B.

In Cava, a half adder circuit can be written as shown in Listing A.2. In the listing, `signal` is the "abstract" function that translates Cava types into Coq types (one can see it as a sort of "placeholder" for semantics later-to-be-defined). Moreover, `cava` is a *monad* – a concept whose explanation lies outside of the scope of this dissertation. Interested readers can refer to Lava [140].

Note that the circuit manipulates the type `Bit`, which is a native Cava type (defined in `SignalType`) and uses `xor2` and `and2`, from Table A.1. Moreover, the syntax `a ← b` can be seen as assignments, i.e., we "store" the value of `xor2(x,y)` in a variable called `sum` (the equivalent of defining a `wire` in SystemVerilog); and the same goes for `carry`. The circuit returns the pair (`sum`,`carry`).

Next, two half adders can be used to design a full adder, which has its truth table shown in Table A.3. The full adder sums three input bits instead of two and then stores the result in a two-bit output: a sum and a carry-out, just like the half adder. A full adder can be built with two half adders, as shown in Figure A.2. In Cava, the full adder can be defined as shown in Listing A.3. As it can be seen, two half adders are instantiated, i.e., the circuit is copied twice and the inputs are connect to wires `a`/ `b` and and `abl`/`cin`, respectively. The notation (`a`,`b`) ← `f` (`x`,`y`);; means, informally, that the result of `f` is "saved" into variables `a` and `b`, which can be used in the following code.

Inputs			Outputs	
A	B	Cin	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Table A.3: Full adder truth table.

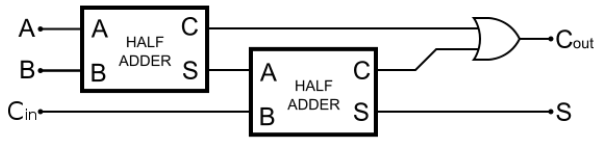


Figure A.2: A full adder built with two half adders.

Listing A.3: Full adder circuit in Cava.

```
Definition fullAdder '(cin, (a, b)) : cava (signal Bit * signal Bit) :=
    '( abl, abh) ← halfAdder (a, b) ;;
    '( s, abch) ← halfAdder (abl, cin) ;;
    cout ← or2 (abh, abch) ;;
    ret (s, cout).
```

B.2. Functional programming. The fact that Cava is implemented in Coq allows users to enjoy all of the functional programming features of Gallina. The Cava library, for example, defines a series of higher-order combinators that are useful for writing complex circuits in a generic way, such as binary trees, chains, et cetera. One such combinator is `col`, which replicates the logic of a single circuit an arbitrary N number of times, as shown in Figure A.3 (for simplicity, the figure shows the case where $N = 3$).

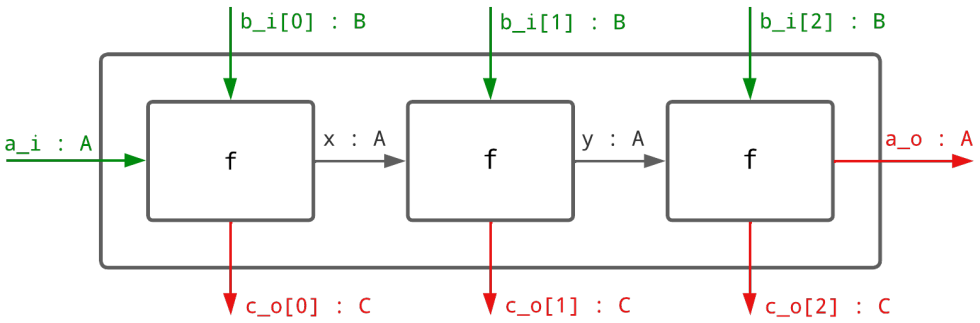


Figure A.3: The `col` combinator with $N = 3$. The circuit `f` is repeated 3 times in a chain-like fashion. Legend: ⟶ Inputs, ⟶ Outputs.

In the figure, `A`, `B`, and `C` are arbitrary signal types. A single circuit `f` is replicated in a

Listing A.4: Definition of the high-order circuit combinator `col`.

```
Fixpoint col {A B C} (f : A * B → cava (C * A)) (a_i : A) (b_i : list B) : cava (list C * A) :=
    match b_i with
    | [::] ⇒ ret ([::], a)
    | b0 :: b ⇒
        '(c0, a) ← f (a_i, b0) ;; (* Apply the circuit to a and b *)
        '(c, a) ← col f a b ;; (* Recursive call *)
        ret (c0 :: c, a) (* Append the result c0 to the list of results *)
    end.
```

way such that the output from a first instance of `f` is fed as input to a second instance of `f`, and so forth. Each instance of `f` takes an element of the input list `b_i` as argument and produces an element of the output list `c_o`. Terms of type `A` are propagated through different instances of `f`, starting by the input `a_i` being fed to the first instance of `f` and ending at the last instance of `f` generating the top-level output – `a_o`.

Listing A.4 shows how the `col` combinator is defined in Cava, where N is given by the length of the list `b_i`. The definition of `col` *pattern matches* over `b_i`. When `b_i` is `b0 :: b` (recall the definition of `list` from Section 2.3.1), `f` is applied to the pair (`a_i`,`b0`) and the result is "written" into a new pair (`c0`,`a`). Next, a recursive call to `col` is made, with arguments `a` and `b`, where `b` is the tail of the original list `b_i`. Each call to `col` finishes by appending the output `c0` to a list of all outputs that where recursively generated (`c`) and returning the pair ((`c0 :: c`), `a`).

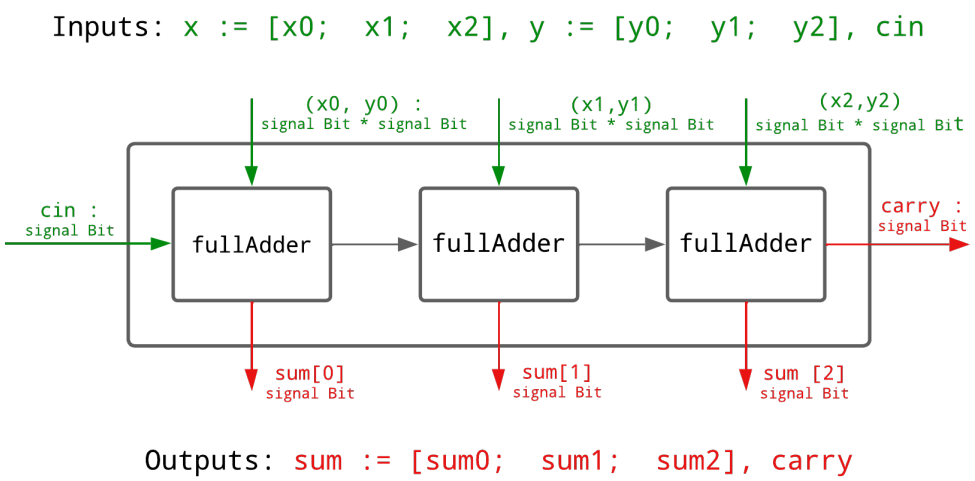


Figure A.4: Combining three `fullAdders` to make a 3-bit ripple carry adder.
Legend: ⟶ Inputs, ⟶ Outputs.

Cava already includes a handful of useful combinators, such as `col`, but this style of

programming and combining circuits can be largely put into use to define other complex circuits very concisely.

Further elaborating on the adder example, the `col` combinator can now be used to combine full adders in a chain and make a ripple-carry adder – as shown in Figure A.4. Again, for simplicity, the figure only shows the specific case where $N = 3$.

In the figure, `x` and `y` are 3-bit inputs, where `x0` and `y0` are the Least-Significant Bits (LSB), respectively. Note that, from the definition of `fullAdder` (Listing A.3), each instance of the circuit expects a pair (`cin`,(`a`,`b`)) as input. Hence, in Figure A.4, the first full adder is fed with the top-level input `cin` and the pair (`x0`,`y0`). The `cin` input of the subsequent circuit is the `cout` from the first one, and the input pair (`a`,`b`) is now (`x1`,`y1`). That pattern is repeated N times. The top-level outputs are: an N-bit output carrying the arithmetic sum of `x` and `y`; and a 1-bit `carry`, representing whether an overflow occurred. Listing A.5 shows the definition of `addC` – an implementation of the circuit described in Figure A.4.[7]

Listing A.5: Definition of `addC`.

```
Definition addC {N : nat}
    (* Three inputs *) (inputs :
        (* x from Figure 2.15 *) signal (Vec Bit N) *
        (* y from Figure 2.15 *) signal (Vec Bit N) *
        (* cin from Figure 2.15 *) signal Bit) :
    (* Two outputs *)
        (* sum from Figure 2.15 *) cava (signal (Vec Bit N) *
        (* carry from Figure 2.15 *) signal Bit) :=
    let '(x, y, cin) := inputs in
    x ← unpackV x ;; (* Transforms into Coq vector *)
    y ← unpackV y ;; (* Transforms into Coq vector *)
    col fullAdder cin (vcombine x y).
```

Note that, in the Listing, the circuit definition uses the function `unpackV` which converts Cava vectors (`Vec`) to Coq vectors (`Vector`). This is a necessary step to use the function `vcombine`, from the Coq Vector library. `vcombine` takes two vectors of equal length and merges them into a single one where each element is the pairwise combination of elements of the previous two.

Furthermore, a simplified interface for the generic adder can also be defined as shown in Listing A.6. `addN`, different than `addC`, forces `cin` to be `zero` and does not indicate an

[7]Note that the definition of `addC` uses vectors, while the definition of `col` uses lists. In fact, there is an extra circuit wrapper defined on the top of `col` which converts vector inputs to lists and list outputs back to vectors. We do not include such wrapper here for brevity.

Listing A.6: Definition of `addN`.

```
Definition addN {N : nat}
    (* Input *) (xy: signal (Vec Bit N) * signal (Vec Bit N)) :
    (* Output *) cava (signal (Vec Bit N)) :=
    '( sum, _) ← addC (xy, zero) ;; (* Ignores the carry out output *)
    ret sum.
```

overflow, i.e., it has no "carry out" bit. In other words, `addN` implements a sum in the finite field $GF(2^N)$, i.e., it wraps around 2^{N-1} when an overflow occurs.

B.3. Sequential circuits. Cava defines another layer called `Circuit` to represent sequential circuits. In fact, this upper layer is more generic than that – it is intended to describe *all circuits*, *including* purely combinatorial circuits. The definition of `Circuit` follows a deep-embedding pattern, with different circuits being defined as constructors of an inductive type.

Figure A.5 depicts the different `Circuit` constructors – which can be used to build complex combinatorial and sequential circuits. In the figure, the notation `Circuit A B` means "a circuit that takes as input a term of type `A` and produces as output a term of type `B`". In addition, types always start with capital letters.

As it can be seen in the figure, `Comb` is a wrapper around a combinatorial circuit `f`, written with the constructs described in the previous section. `Compose` is a circuit constructor that takes two other circuits as arguments and composes them, as in function composition. `First` and `Second` apply a circuit passed as argument to only the first or second element of a pair of signals, respectively. `DelayInitCE` represents the basic memory element – a register of arbitrary width holding a signal type with an enable signal.

Last, `LoopInitCE` is an interesting case. The idea behind it is to use a register ("Delay", in the figure) to store the *loop state*, and to use another `Circuit` passed as argument, i.e., the *loop body*, to "execute" some combinatorial or sequential logic that is a function of the input and the loop state. The loop body produces an output and a new state, which is stored in register "Delay" on the next clocking event. Moreover, the signal carrying the loop state is not visible from the top-level circuit point-of-view. This constructor constitutes the basic block to design *stateful* circuits. Note that the loop body can be any other circuit, including other circuits built with `LoopInitCE`. Nested loops represent circuits that use multiple registers to hold their state.

Furthermore, Cava includes another constructor `Loop`, derived from `LoopInitCE`. In `Loop`, the enable signal is set as `true` by default and the initial value of the loop state is set to be the default value of the type `S`. Other than `Loop`, Cava also provides a `LoopInit`, where enable is also set to `true` by default, but the user can specify an initial value for the register.

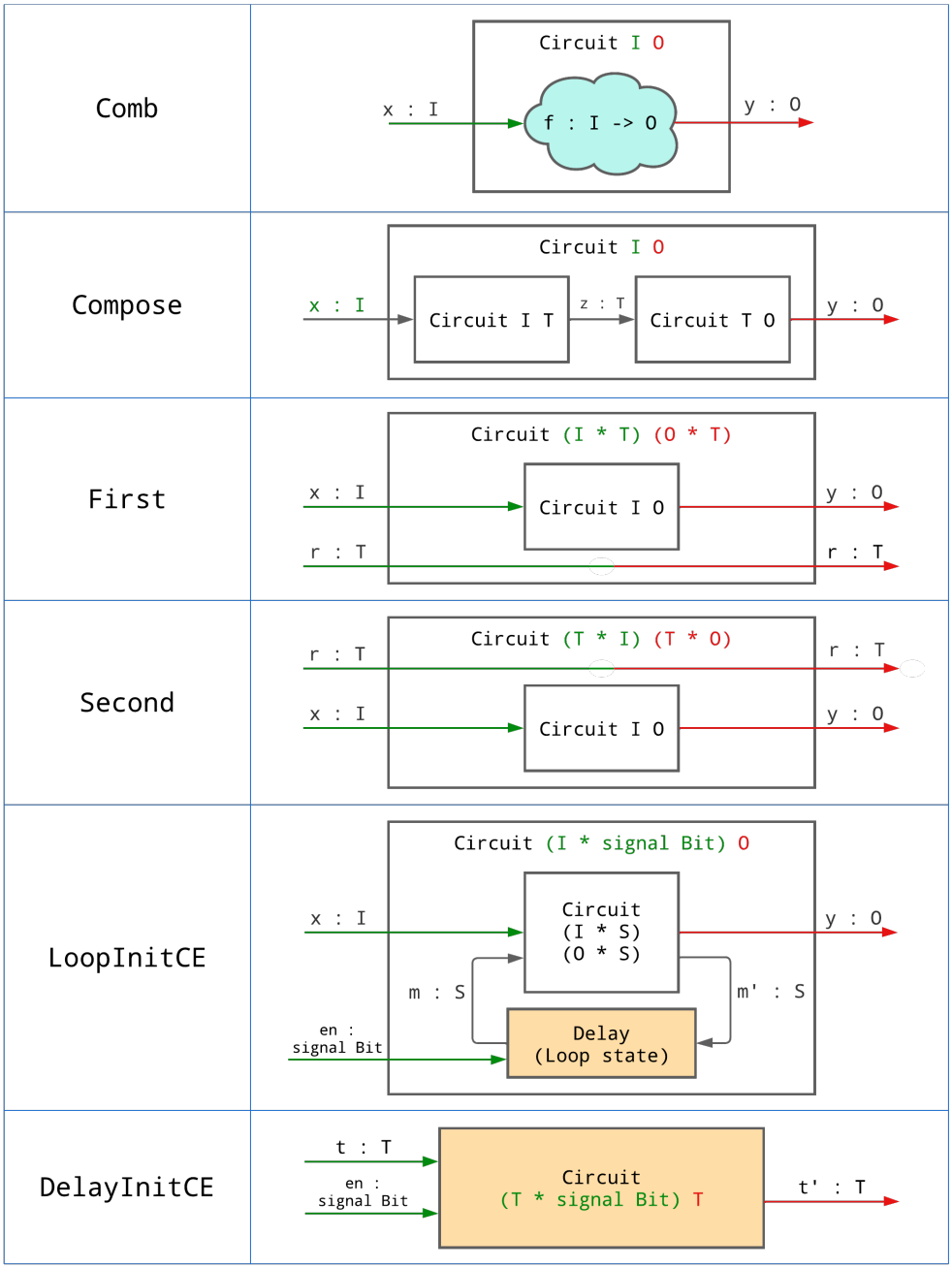


Figure A.5: `Circuit` constructors.
Legend: Combinational Logic, Registers,
→ Inputs, → Outputs.

Listing A.7: Defining a circuit that performs the rolling sum of an input stream.

```
Definition sum {N : nat} : Circuit (signal (Vec Bit N)) (signal (Vec Bit N)) :=
    Loop
        (Comb (* The combinational circuit that makes up the loop body *)
            (fun '(input, state) ⇒
                (* Use addN to sum the current input with the sum (circuit state) *)
                sum ← addN (input, state) ;;
                (* return output and new state (the same in our case) *)
                ret (sum, sum))).
```

To extend the adder example, we can now use `Loop` to define a circuit that performs the "*rolling sum*" of an input stream: at each clock cycle, the value at the input port is added to the value of a register storing the accumulated sum. This can be defined as shown in Listing A.7, where `N` is the width of the bit-vectors taken as input.

Note that `Loop` takes another circuit (the loop body) as argument: here, a purely combinational circuit defined with the constructor `Comb`. Recall from Figure A.5 that the body of a `Loop` constructor expects a pair `(I * S)`. In this case, `I` is the type `signal (Vec Bit N)` and `S` is the type `(Vec Bit N)`. The circuit calculates the sum of `input` and `state` using `addN`. The result, `sum`, is outputted by the circuit and stored into the feedback register.

C. Semantics

In the following, we briefly discuss both semantics presented in Figure A.1. These are dense topics and we opt for presenting only a brief intuitive understanding of how these semantics work and how they can be used. The part on proof semantics briefly discusses a proof about the correctness of `addN`, a purely combinational circuit. However, we do not discuss here, for instance, any proofs regarding sequential circuits, which often include *invariants* and more sophisticated proof strategies.

C.1. Proof/simulation semantics.

The first step of defining proof semantics is to translate Cava types (`SignalType`) to Coq types (`Type`). Listing A.8 shows the definition of `combType`, one possible interpretation for the `signal` function – discussed in parts A and B.

Specifically, `combType` translates Cava types into the Coq types `bool` and `Vector`. In Coq, vectors are a sort of polymorphic dependently-typed length-constrained list: "*polymorphic*" means that they are parameterised over the type of elements they hold, "*dependently-typed*" means that the definition of vectors contain built-in proofs; and "*length-constrained*" means that the number of elements in the vector is enforced by type checking. In addition, vectors

Listing A.8: Definition of combType, an interpretation of Cava types to Coq types.

```
Fixpoint combType (t: SignalType) : Type :=
  match t with
  | Void ⇒ unit
  | Bit ⇒ bool
  | Vec vt sz ⇒ Vector.t (combType vt) sz
  | ExternalType _ ⇒ unit (* No semantics for combinational interpretation. *)
  end.
```

which have elements of type `bool` can also be referred to by an alias – `Bvector`, which count on a large set of already proven properties, part of Coq's standard library.

Next, using the interpretation of Cava types provided by `combType`, implementations must be provided for all of the combinational operators (some of which were presented in Table A.1). For example, basic gates (`and2`, `xor2`, et cetera) are instantiated with the corresponding Coq functions over the type `bool` and something such as dynamically indexing a vector (`indexAt`) is instantiated as the function `nth_default` – an indexing function from the `Vector` library.

Furthermore, since combinational circuits often implement arithmetic operations, it is only natural that correctness proofs should compare circuits to operations on numbers – which already count on a large collection of proved facts. Specifically, bit vectors are very similar in structure to a Coq type called `N`, a representation of natural binary numbers defined in the library `BinNat` and developed in the libraries `NArith` and `Ndigits`.

Consider the unsigned 4-bit binary literal "1101" (decimal 13), for example. A bit-vector representation of this value (using `bool` and `Vector`) would look like the following (using vector notations – which are very similar to the `List` notation):

```
▷ [true;false;true;true]%vector.
```

Note that the LSB appears to the left of the vector, in a way such that the head of the vector always contains the LSB.

Coq defines `N` very similarly. The definition of `N` depends on `positive` – a recursive inductive type used to model *positive* natural numbers. Starting from digit 1, represented by the constructor `xH`, one can add a new LSB via the constructors `xO` (digit 0) or `xI` (digit 1). In Coq, this means that `x0` and `x1` are constructors that always expect another term of type `positive` as argument, while `xH` does not, thus representing the end of a series of digits. According to that representation, the same literal "1101" would be given by the term:

```
▷ xI (x0 (xI xH)).
```

The definition of `N` simply extends `positive` by adding 0 (`N0`).

Listing A.9: Correctness property for addN.

```
Lemma addN_correct n (a b : Bvector n) : (* a and b are bitvectors of length n *)
  let sum := N.add (Bv2N a) (Bv2N b) in (* add function from the N library *)
  addN (n:=n) (a,b) = N2Bv_sized n sum.
Proof. (* omitted *) Qed.
```

To conclude the adder example, we may now use that relation between Bvector and N to prove that addN performs the addition over binary natural numbers (N.add). The correctness Lemma can be stated as shown in Listing A.9. For the sake of conciseness, we refrain from discussing the proof of addN_correct. In the listing, note the use of the functions Bv2N, which converts Bvectors to N, and N2BV_sized – its dual.

While combinational circuits can be checked against mathematical functions, reasoning about sequential circuits requires the notion of passage of *time*. Therefore, the proof semantics for sequential circuits evolve around the step function – an evaluation of the Circuit datatype. The goal of step is to "run" the circuit for one clock cycle. It takes two inputs: a value representing the circuit's state and a value representing its inputs (which can be an arbitrary N-tuple). It produces two outputs: a new state and the circuit's output value.

To reference circuit states, Cava also defines a function circuit_state which, for a given Circuit, returns the type of its state. For example, when the circuit passed to circuit_state is composite (built with Compose), it returns a 2-tuple. Each element of the 2-tuple contains the state of the corresponding sub-circuit – obtained through recursive calls to circuit_state. If Circuit is purely combinational (built with Comb), then it has no state, and circuit_state returns unit (an inductive type with sole inhabitant). The type signature of the step function is shown below:

```
▷ Fixpoint step i o (c : Circuit i o)(s : circuit_state c)(in : i) : circuit_state c * o.
```

The step function takes a Circuit c as argument, a state of c, an input of c (of type i); and produces a pair made of a new state of c, and an output of c (of type o). For brevity, we do not discuss the actual implementation of step. The step function can be called an arbitrary number of times. This is exactly what is done by the simulate function – which simulates the circuit behaviour through an arbitrary number of clock edge transitions.

C.2. Netlist semantics. The semantics for creating a netlist AST is more complex than the proof/simulation semantics. It starts by defining an inductive type Signal – which aims at representing the various types of signals that can flow through a netlist.

Different than combType, which is a simple "mapping" from Cava types to Coq types, Signal is a kind of "wrapper" around Cava types. Signal is also syntactically richer then SignalType. In SignalType, every value that "flows" through the circuit is of type Bit (there

can also be bit-vectors and be multi-dimensional bit-vectors, but *values* are always of type `Bit`). In `Signal`, however, a `Bit` can be wrapped around `Signal` in different forms, such as: `Gnd`, representing a constant logic *low*; `Vcc`, representing a constant logic *high*; `Wire`, representing a numbered wire, or `NamedWire`, representing a named wire. This makes sense, since when defining a circuit, Cava users should not reason about wires, and only about bits and operations over bits. A netlist, however, requires reasoning about how circuits are wired, how wires are numbered and labelled, among other things.

A netlist is concretely represented by a list in which elements are of yet another inductive type: `NInstance` (the actual name in Cava is `Instance`, but we call it `NInstance` here, to avoid confusion with the Coq command `Instance`, which instantiates a type class). The definition of `NInstance` is partially shown in Listing A.10.

Listing A.10: `NInstance` – nodes and edges of the netlist AST.

```
Inductive NInstance : Type :=
  | Not:      Signal Bit → Signal Bit → NInstance
  | And:      Signal Bit → Signal Bit → Signal Bit → NInstance
  | (* ... *) .

Notation Netlist := (list NInstance).
```

`NInstance` represents the type of nodes of the AST. `And`, for example, can be thought of as a type of node with three directed edges: the first two edges lead to the node and thus can connect with other nodes which *produce* a `Signal Bit`, and the last edge goes from the `And` node to another node that *expects* a `Signal Bit` as input.

Moreover, a set of functions unfolds/explores a circuit definition, starting from the top-level definition, adding instances and connecting them through wires (AST edges). The implementation of the actual functions that parse the circuit and build the final AST are quite complex and rely on multiple concepts that lie outside of the scope of this dissertation.

Compiling a circuit definition into an AST is made quite simple for the Cava end-user. Besides the circuit itself, one needs only to write its interface (specifying inputs and outputs) and call the function `makeCircuitNetlist` to build the netlist AST. Finally, the file that contains the call to `makeCircutiNetlist` has to be extracted to Haskell. Then, the AST can finally be translated from Haskell to SystemVerilog by an additional script.

A.2 Overview

The connection we propose between CoqDRAM and CavaDRAM results in a design workflow – presented below as a series of steps:

1. Describe a DRAM scheduling algorithm in CoqDRAM and prove its correctness against the JEDEC standards and other high-level properties of interest (a process described in Chapter 6);
2. Describe the controller circuit in Cava;
3. Prove *equivalence* between the two representations;
4. Extract a SystemVerilog circuit from the Cava controller (automatically), which can then be used as a plug-in replacement in existing hardware designs.

On the one hand, CoqDRAM – written in plain Coq – has been conceived to design, explore, model, and finally prove the correctness of DRAM arbitration *algorithms*, abstracting from actual hardware implementations. It has little to no size constraints, a fact that allows users to use powerful abstractions to prove strong properties. On the other hand, CavaDRAM derives real memory controller hardware implementations. This means that hardware limitations become relevant, e.g., CoqDRAM uses queues that can grow to arbitrary sizes to store incoming requests, which is evidently not possible in a hardware model. Therefore, a logical equivalence proof will require the queue to be limited in size. This duality is formalised through a series of assumptions – which are presented further – that allow us to limit the scope of CoqDRAM algorithms.

Figure A.6 illustrates the coupling between CoqDRAM and CavaDRAM. For each scheduling algorithm implemented in CoqDRAM, we introduce a *provably equivalent* controller in CavaDRAM (the formal definition of *equivalence* is defined later). The RTL code produced from a controller implementation (using Cava's code extraction) can then be used in existing RTL designs. In our case, as a validation experiment, we use it as a plug-in replacement in an existing DDR4 controller implementation [126], as explained in further detail in Chapter 7. From a framework point of view, the additional workload introduced by the back-end coupling consists solely of writing the equivalent controllers in Cava and the equivalence proof with the representation in CoqDRAM.

[1]Figure A.6 omits several components and does not represent a complete architecture, as its goals are to ease comprehension and provide an overview of the system.

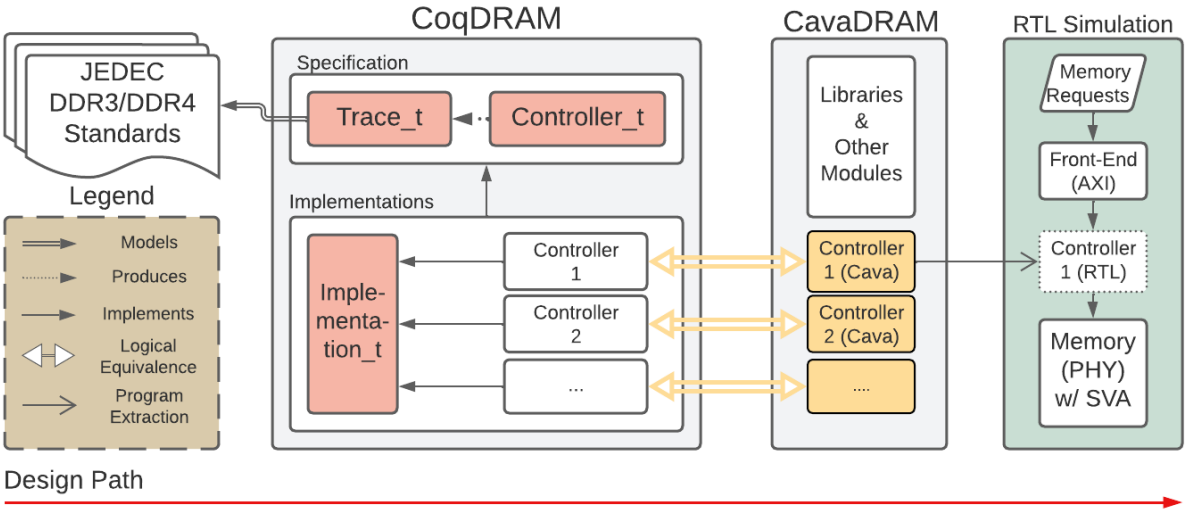


Figure A.6: System architecture.[1]

■ CoqDRAM classes, ■ Introduced workload, ■ DDR4 hardware simulation setup.

Hardware controller implementations impose some constraints that are not captured by CoqDRAM. The setup we use for simulation and synthesis, for instance, is equipped with an AXI bus interface, which expects an interface to communicate with the memory controller. From this point onwards, we focus on AXI-style buses. Other buses (and underlying protocols) require other types of interfaces. As a consequence, CavaDRAM controllers have to implement an interface containing the following input and output signals: a) the arrival of a new request is signalled by a 1-bit `pending_i` input signal; b) a *single* `request_i` is provided as a bit vector input; and c) the circuit has to produce a 1-bit `ack_o` signal as output. The `pending_i`/`ack_o` signals allow to perform a handshake (used in many bus implementations besides AXI).

Listing A.11: Controller interface.

```
Class ControllerInterface := mkCavaController {
  Controller : Circuit
   (combType Bit * combType (Vec Bit REQUEST_WIDTH))
   (combType Bit * combType (Vec Bit DRAM_CMD_WIDTH) * combType (Vec Bit REQUEST_WIDTH));
}.
```

Hence, it is necessary to define a generic interface for memory controllers in Cava – upon which axioms realising the signals described above can be written. The interface for memory controllers is shown in Listing A.11. Controllers take two inputs: a 1-bit signal representing

pending_i, and a REQUEST_WIDTH-bit signal representing the incoming request. Moreover, controllers produces three outputs: a 1-bit signal representing ack_o, a DRAM_CMD_WIDTH-bit signal representing the command issued to the memory device, and a REQUEST_WIDTH-bit signal representing the request associated to the issued command – which contains the physical address sent to the memory device over the address bus. REQUEST_WIDTH and DRAM_CMD_WIDTH are parameters to the interface.

Note that, in Listing A.11, we use combType instead of signal to "translate" Cava types into Coq types. This important difference reflects the fact that such interface is only used for proofs and tests in the Coq "*world*" (Section A.1). An actual circuit definition – shown later in the text – needs to be defined using the more generic signal, which allows the circuit to be interpreted according to different *semantics*.

Listing A.12: Assumption reflecting hardware-level implementation constraints.

```
Class HW_Arrival_function_t {AF : Arrival_function_t} {Cntrl : ControllerInterface}
    := mkHWArrivalFunction {
    HW_single : forall t, size (Arrival_at t) <= 1;

    (* Signals *)
    pending_i : nat → combType Bit; (* pending input for controller circuit *)
    request_i : nat → combType (Vec Bit REQUEST_WIDTH); (* request as input *)
    ack_o : nat → combType Bit; (* full signal sent back to bus protocol *)

    (* Connecting the signals to the circuit *)
    HW_output : forall t c,
        let inputs := (pending_i t, request_i t) in
        let '(_,((full,_),_)) := step Controller c inputs in full = ack_o (S t);

    (* If there is an arrival, it means that pending_i has been asserted and the
        memory controller is ready to accept a new incoming request *)
    HW_arrived : forall t, size (Arrival_at t) = 1 ↔
        (ack_o t = one) ∧ (pending_i t = one);

    (* If there is an arrival, then the arriving request is equivalent on both
        sides *)
    HW_request : forall t, size (Arrival_at t) = 1 →
        EqReq t (ohead (Arrival_at t)) (request_i t);
}.
```

Back to the link between CavaDRAM and CoqDRAM models. ControllerInterface is

a less generic interface than the arrival function defined in CoqDRAM, since it relies on these extra signals that communicate with the bus arbiter. An equivalence proof thus can only succeed by introducing two additional assumptions that constrain the arrival model of CoqDRAM. Recall that, in CoqDRAM, `Arrival_function_t` allows an arbitrary number of requests to arrive "in parallel" at any instant. In order to introduce such assumption-s/hypotheses, we define a class `HW_Arrival_function_t`, as shown in Listing A.12. Again, bear in mind that this is an interface specific for AXI-style busses, and other bus protocols might require a different interface. The class depends on an existing `Arrival_function_t` from CoqDRAM, and on an existing Cava memory controller – both passed as implicit arguments.

Assumption 1. The arrival function needs to be constrained to a *single incoming request per cycle*. In Coq, we model this constraint with a PO that limits the number of requests in the incoming arrival list; the PO is denoted by `HW_single` in Listing A.12. For one, this models the limitation of the AXI bus mentioned in the interface above. In addition, this reflects a fundamental limitation on memories, which are typically used to implement queues in hardware: the implementation cost of memories increases drastically with the number of read/write ports. In our case (Distributed/BRAM of FPGAs), is limited to a single read/write port each. Bear in mind that the proofs in CoqDRAM are valid for all possible arrival functions without any limitations, including `HW_Arrival_function`.

Assumption 2. Note that Assumption 1 does not constrain the number of outstanding requests from the requestors, it is just a constraint on the bus interfacing with requestors and memories of queues. The interface described before also comprises a handshake protocol, which allows a controller to accept (or not) newly in-coming requests. CoqDRAM only considers requests that are *accepted* by the controller, i.e., from the moment that the request is processed by the controller. Moreover, the interface to the bus arbiter – which implements the handshake protocol – consists of three signals: `pending_i`, `request_i`, and `ack_o`. All three signals represent *instant values*, i.e., booleans on a given time instant `nat`, passed as parameter. Finally, the assumption is realised through three proof obligations:

1. `HW_output` effectively links the signals to the circuit interface through the `step` function (described in more detail further in the text). In natural language, the proof obligation states that a controller that is fed with the pair `pending_i t`,`request_i t` shall produce a tuple in which one of its elements is `ack_o` (`S t`), i.e., the `ack_o` signal in the following cycle.

2. `HW_arrived` states that a request could only have arrived (indicated by the condition `size` (`Arrival_at`)= 1) **iff** `ack_o` at `t` is asserted to logical `one`, meaning that the memory

controller is ready to accept a new incoming request at at `t`, and if `pending_i` at `t` is also asserted to logical `one`, meaning that the handshake protocol is validating its transaction.

3. `HW_request` establishes a link between requests from both sides. In natural language, it states that if a request has arrived, then such request must be equal to the signal specified by `request_i`. The "equivalence" relation is established by the function `EqReq`, omitted from the text for conciseness. In summary, the function compares fields of the `Request_t` record to individual components of the `request_i` vector (remember that `combType` (`Vec Bit REQUEST_WIDTH`) is translated to `Vector.t bool REQUEST_WIDTH`).

A.3 From CoqDRAM to CavaDRAM Implementations

As a Proof-of-Concept, we implement a scheduling algorithm in Cava *based* on the *First-In-First-Out* (FIFO) scheduler, as originally proposed in CoqDRAM. However, a hardware implementation must also manage refresh operations. Actually, this was one important motivating factor to later consider modelling refresh commands in CoqDRAM (c.f. Chapter 8). In this chapter, we describe a Cava circuit that mimics the behaviour of a modified version of FIFO, namely `FIFOREF`.

The `FIFOREF` algorithm is a modified version of the FIFO algorithm presented in Chapter 6 that issues refresh commands. Similarly, we also propose a modified version of TDM that issues refresh commands, namely `TDMREF`. These implementations are work-in-progress, i.e., not all proof obligations have been discharged at time of writing, due to a change of research focus. Here, for brevity, we only include a brief description about the differences between `FIFOREF` and FIFO – given that the same techniques apply for `TDMREF`.

The `FIFOREF` modifications w.r.t FIFO can be summarised briefly as follows: the algorithm keeps a counter to track the date when a `REF` command is due. When that condition triggers, the scheduler stops the normal processing of requests and performs a refresh cycle, thus delaying any outstanding request that might be in the queue. Note that this preserves the correctness of the initial FIFO implementation, since the distance between other kinds of commands (`PRE`, `ACT`, and `CAS`) may only increase.

Listing A.13: Definition of `FIFO_state_t`.

```
Inductive FIFOREF_state_t :=
    | IDLE    : Counter_t → Counter_ref_t → Requests_t → FIFOREF_state_t
    | RUNNING : Counter_t → Counter_ref_t → Requests_t → Request_t → FIFOREF_state_t
    | REFRESHING : Counter_ref_t → Requests_t → FIFOREF_state_t.
```

Listing A.14: Circuit definition (CavaFIFOREF).

```
1  Definition CavaFIFOREF : Circuit
2      (* Inputs : pending_i and request_i *)
3      (signal Bit * signal (Vec Bit REQUEST_WIDTH))
4      (* Outputs *)
5      (signal Bit * signal (Vec Bit DRAM_CMD_WIDTH) * signal (Vec Bit REQUEST_WIDTH)) :=
6      (* Initial value of state *)
7          let s_init   : combType (state) := STATE_IDLE_VEC in
8      (* Initial value of cnt *)
9          let cnt_init : combType (counter) := CNT_NIL_VEC in
10     (* Initial value of cref *)
11         let cref_init : combType (counter_ref) := CNT_REF_NIL_VEC in
12     LoopInit s_init ( (* pending_i, request_i, state *)
13         LoopInit cnt_init ( (* pending_i, request_i, state, cnt *)
14             LoopInit cref_init ( (* pending_i, request_i, state, cnt, cref *)
15                 Second (ReadLogic) >==>Comb (...) >==>
16                 Second (Queue) >==>Comb (...) >==>
17                 Second (NextCR) >==>Comb (...) >==>
18                 Second (CmdGen) >==>Comb (...) >==>
19                 Second (Update_) >==>
20                 Comb (fun '(full,cmd,cr,(ns,nc,ncref)) ⇒ ret (full,cmd,cr,ns,nc,ncref))))).
```

Although it is out of scope to present the complete implementation of FIFOREF, we must discuss the definition of its internal state type, shown in Listing A.13. Note that the definition of FIFOREF_state_t is very similar to the FIFO state definition from Listing 6.2, with the addition of an extra state, REFRESHING and an additional counter, Counter_ref_t.

The core of the equivalence between the CoqDRAM and CavaDRAM algorithms lies in representing states in a *comparable* way. In other words, in order to prove that algorithms behave in a similar way (which is formally defined later in the text), we must first define what does it mean for states to be similar. It is thus necessary to analyse FIFOREF_state_t – the definition of a state in the FIFOREF implementation, shown in Listing A.13.

On the Cava side, we must write a circuit that manipulates the same "state variables", updating them according to a certain combinatorial logic at each cycle. As introduced in Chapter A.1, sequential circuits in Cava can be built with the constructor LoopInit, which expects an initial value and another circuit as arguments. Listing A.14 shows the top-level definition of the CavaFIFOREF circuit. In addition, Figure A.7 shows a high-level schematic of the resulting circuit.

The inputs to the circuit are defined at Line 3. Note that, contrarily to the more specific circuit interface in Listing A.11, the circuit definition uses the more generic signal, which means that the same circuit could be interpreted in different ways. Moreover, note that the

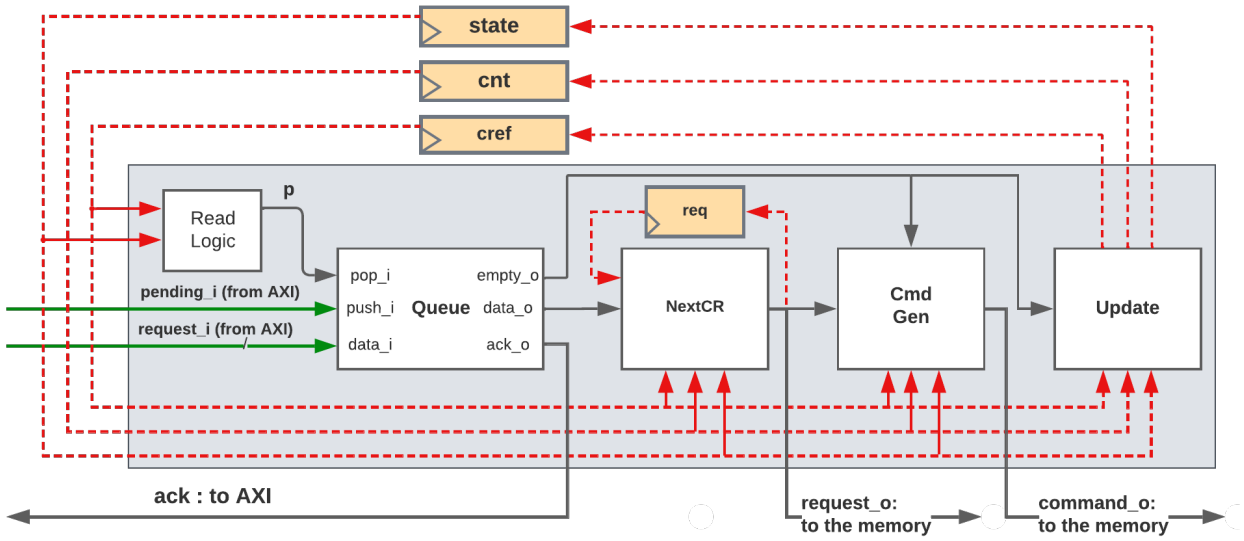


Figure A.7: `CavaFIFOREF` – top-level circuit schematic.

circuit expects the same two inputs and produces the same three outputs as specified by `ControllerInterface` considering the simulation semantics.

Next, note the three imbricated `LoopInit` constructors in Listing A.14. These constructors introduce three registers, indicated in Figure A.7 by `state`, `cnt`, and `cref`. These three registers define the state of the circuit, and a parallel with `FIFOREF_state_t` can be made: `state` is a 2-bit register used to identify if the controller is `IDLE`, `RUNNING`, or `REFRESHING`; `cnt` is of width `CNT_WIDTH`, and represents a counter equivalent to `Counter_t`, and `cref` is of width `CNT_REF_WIDTH`, and is yet another counter equivalent to `Counter_ref_t`.

Furthermore, the list of outstanding requests in `FIFOREF_state_t` (of type `Requests_t`) is represented by another stateful circuit defined separately, `Queue` – hence why its registers are not shown in Figure A.7. In addition, when the controller finds itself in the `RUNNING` state, the "variable" used to store the request currently under processing (of type `Request_t`) is also represented through a register in Figure A.7, labelled `req`. `req`, different than `state`, `cnt`, and `cref`, does not apper on the top level `Circuit` definition; it is rather a register belonging to the (stateful) `NextCR` circuit, and it is thus invisible from the point-of-view of `CavaFIFOREF`. Lines 7 to 11 attribute initial values to registers `state`, `cnt` and `cref`. For example, the type of `s_init`, the initial value of `state`, is `combType` (`state`), which translates to a Coq vector[2] in which elements are of type `bool`. The initial value, `STATE_IDLE_VEC`, is simply the 2-bit binary literal "00", an encoding for `IDLE`.

The `CavaFIFOREF` circuit starts manipulating a tuple made of the two inputs – `request_i` and `pending_i`, coming from `ControllerInterface`. After the first `LoopInit`, the circuit

[2] https://coq.inria.fr/doc/V8.17.1/stdlib/Coq.Vectors.VectorDef.html

manipulates the 3-tuple made of the inputs plus `state`. After the third `LoopInit`, the circuit manipulates the 5-tuple: ⟨`pending_i`, `request_i`, `state`, `cnt`, `cref`⟩.

The body of the circuit (Lines 15 to 19) applies some sequential or combinatorial logic to the inputs (`pending_i` and `request_i`) and the register values (`state`, `cnt`, `cref`). In the following, we discuss the basic functionality of individual sub-circuits from Figure A.7, with varying levels of detail.

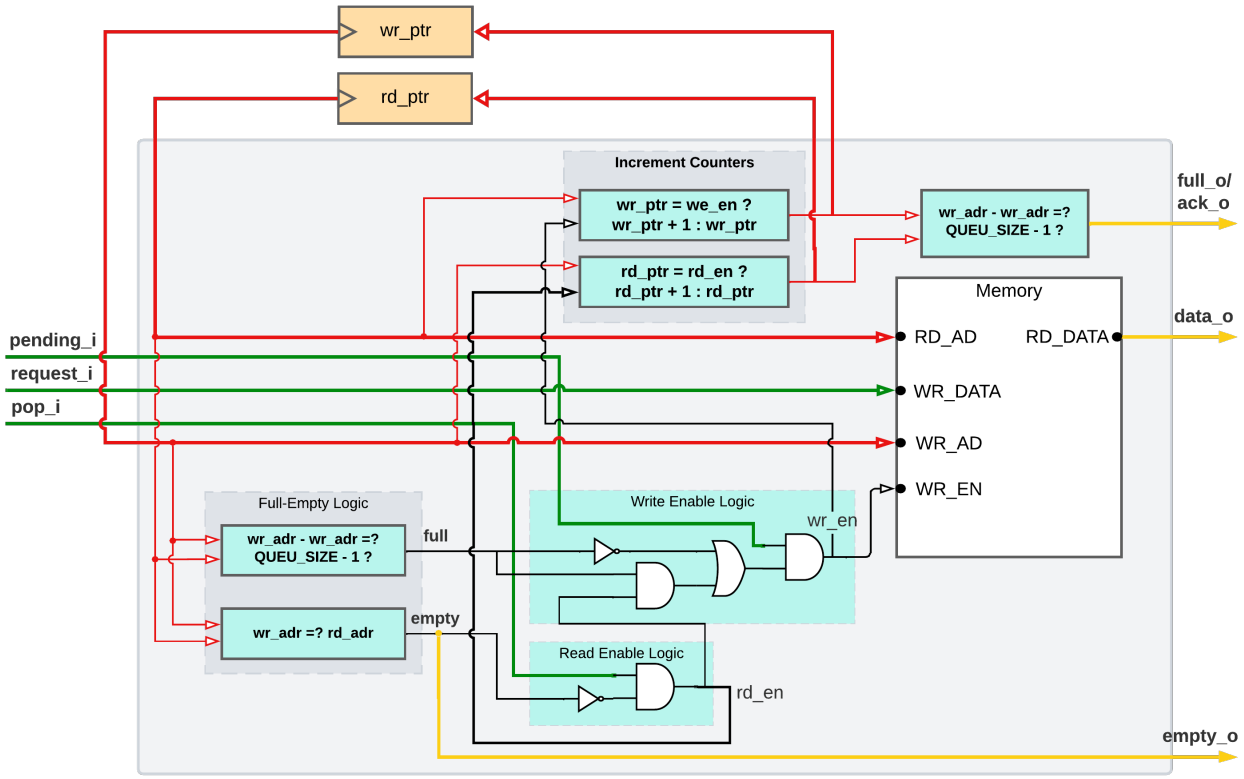


Figure A.8: `Queue` – circuit schematic.
Logic: Combinational Logic; Registers.
Signals: Register signals; Inputs; Outputs; Internal signals.

Queue. `Queue` consist of a multiple-port memory and combinational logic around it – which together implement a typical FIFO queue. In this case, since no simultaneous reads occur, only one read-port is used. While the Cava code with the definition of `Queue` is omitted for conciseness, its schematics can be seen in Figure A.8. Note that the circuit manipulates three inputs and: `push_i`, `request_i`, and `pop_i`; and two (internal) registers: `wr_ptr` and `rd_ptr`. The circuit is centred around an implementation of a multi-port memory.

In Figure A.8, the signals `wr_ptr` and `rd_ptr` are the write and read address fed to the memory, respectively. The incoming request, `request_i`, is the write data fed into the write

data port, WR_DATA. Moreover, the memory expects a "write enable" signal (WR_EN) – which comes from a combinatorial circuit. In summary, we write to the memory whenever push_i is asserted **AND** the queue is not full **OR** the queue is full, but a read will happen, that same cycle (the "Write Enable Logic" from Figure A.8). The task of analysing the rest of the logic in Figure A.8 is left to the reader.

The memory implementation in Cava is a central part of CavaDRAM. While its interface is simple, the same cannot be said about its implementation and proofs – which heavily rely on programming and proving with dependent types.

Read Logic. Straightforwardly, the circuit ReadLogic implements a logic to drive the pop_i signal to Queue. ReadLogic asserts pop_i whenever state is IDLE and a refresh operation is not about to start. The pop_i signal is then used in combination with the empty signal to drive rd_en, which is ultimately used to update the read pointer – rd_ptr.

NextCR. The goal of NextCR is to determine the value of the register req from Figure A.7, which holds the "request currently under processing" (RQUP). Analysing the algorithm in Listing 6.4, if state is IDLE and the queue is non-empty, the RQUP is the request read from the head of the request queue (i.e., data_o, from Queue). In all other cases, either the RQUP retains its value or is non-existent. Specifically, if state (AS in Listing 6.4) is RUNNING and the counter (c in Listing 6.4) is strictly less then WAIT, then the RQUP's value is kept. In all other cases, there is no associated RQUP, which is encoded in Cava as a vector of length REQUEST_WIDTH filled with 0s.

CmdGen. The purely combinational circuit CmdGen is responsible for issuing a new command to the memory device at each clock cycle. As it can be seen in Figure A.7, the circuit has five inputs: the register values state, cnt, cref, as well as the empty_o output signal from the request queue and the request currently under processing (req) coming from NextCR . Figure A.9 shows a schematic of CmdGen and Listing A.15 shows the corresponding Cava code. In the code, multiple auxiliary circuits are used, such as Sidle, Srun, Sref, CeqCAS, CeqACT, et cetera. These circuits are defined in a previous code location and perform simple functions, such as testing if the value of a bit vector is equal to or less equal than a literal. Moreover, in the circuit, we use the Cava primitives and2 and mux2, which – when given the right semantics for simulation – are already proved correct against Coq's andb (the boolean **AND** operation) and the if construct, respectively.

The behaviour of the circuit is simple: a PRE, for instance, is issued whenever the empty_o signal is false (which means that Queue is not empty), cref $\leq$ PREA_date, and the state of the

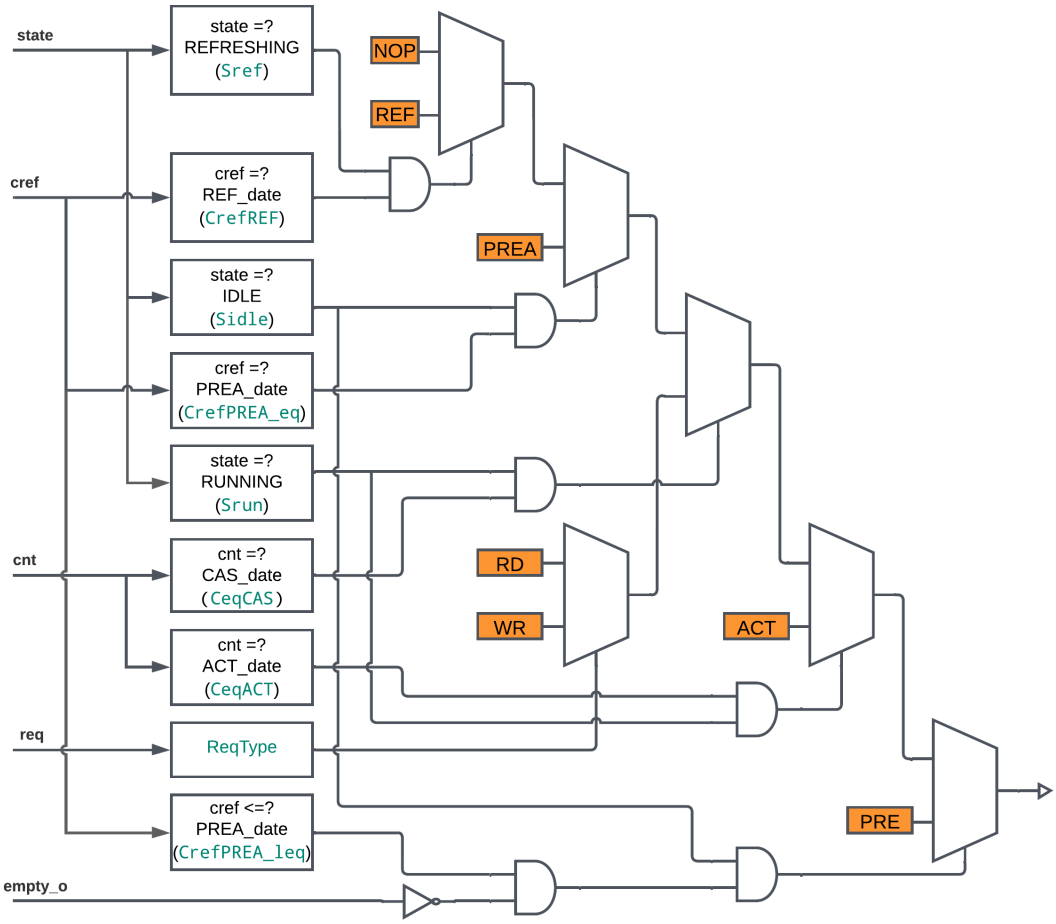


Figure A.9: `CmdGen` – circuit schematic.

Listing A.15: Definition of `CmdGen`.

```
Definition CmdGen : Circuit
    (* Inputs and Output *)
    (state_t * empty_t * counter_t * counter_ref_t * request_t) (command_t)
    := Comb (fun '(state,empty_o,cnt,cref,req) ⇒
    (* Intermediate signals about state *)
    s_idle ← Sidle state ;;
    s_run ← Srun state ;;
    s_ref ← Sref state ;;
    (* Intermediate signals about cnt *)
    c_cas ← CeqCAS cnt ;;
    c_act ← CeqACT cnt ;;
    (* Intermediate signals about cref *)
    c_prea ← CrefPREA_eq cref ;;
    c_prea' ← CrefPREA_lt cref ;;
    c_ref ← CrefREF cref ;;
    (* Intermediate signals about empty_o *)
    ne ← inv empty_o ;;
    (* ----------------- MUXES ----------------- *)
    (* REF mux *)
    ref_mux_sel ← and2 (s_ref,c_ref) ;;
    ref_mux_out ← mux2 ref_mux_sel (NOP,REF) ;;
    (* PREA mux *)
    prea_mux_sel ← and2 (s_idle,c_prea) ;;
    prea_mux_out ← mux2 prea_mux_sel (ref_mux_out,PREA);;
    (* CAS mux *)
    rd_wr_mux_sel ← RequestType (req) ;;
    rd_wr_mux_out ← mux2 rd_wr_mux_sel (RD,WR) ;;
    cas_mux_sel ← and2 (s_run,c_cas) ;;
    cas_mux_out ← mux2 cas_mux_sel (prea_mux_out,rd_wr_mux_out) ;;
    (* ACT mux *)
    act_mux_sel ← and2 (s_run,c_act) ;;
    act_mux_out ← mux2 act_mux_sel (cas_mux_out,ACT) ;;
    (* PRE mux *)
    t0 ← and2 (ne,c_prea') ;;
    pre_mux_sel ← and2 (s_idle,t0) ;;
    mux2 pre_mux_sel (act_mux_out,PRE)).
```

scheduler is `IDLE`. Compare this with the issuing of a `PRE` command in the FIFO algorithm from Listing 6.4 (bear in mind though that that algorithm did not include the condition on the refresh counter).

Update. Finally, `Update` applies the necessary logic to update each register, as seen in Figure A.7. In a way, it can be said that `Update` implements the transition function, since it updates the circuit state. For conciseness, the code of `Update` and its resulting circuit are not shown.

A.4 Defining an Equivalence Relation Between Models

Up until this point, the word *equivalence* has been used to describe memory controllers that behave similarly w.r.t their underlying scheduling algorithms. Here, we define equivalence formally, which is de facto a *bisimilarity* relation. Bisimilarity was introduced (formulated by Park [143], refining ideas from Milner [144]) as the notion of behavioural equality for processes [145].

In the following discussion, we consider the scheduling algorithms in both CoqDRAM and CavaDRAM as *Labelled Transition Systems* (LTS), defined below.

Definition A.4.1 (Labelled Transition System [146])**.** A *Labelled Transition System* (LTL) is a triple $\langle Pr, Act, \rightarrow \rangle$ where Pr is a non-empty set called the *domain* of the LTS, Act is the set of *labels* and $\rightarrow \subseteq \wp(Cons \times Act \times Pr)$ is the *transition relation*.

Which definition of equivalence to use?

Before diving into the definition of bisimulation, it is necessary to analyse more broadly the notion of "*equality of behaviours*". As Sangiorgi [146] states: "Intuitively, two processes should be equivalent if they cannot be distinguished by interacting with them".

A first idea would be to borrow the concept of isomorphism from graph theory, but it is too strong as a behavioural equivalence for LTS [146], since we are not interested on a strict structural equivalence between LTS' but rather on the information that transitions convey. Next, from automata theory, equality means that two automata accept the same language, i.e., the same set of strings, a property also called *trace equivalence*. However, whenever a LTS is non-deterministic, trace-equivalence is not a good option to state equivalent behaviour [146]. Since we do not want to exclude non-deterministic scheduling algorithms from our framework (although an algorithm of such nature would be unlikely to be modelled

in Coq), we opt for a broader concept, *bisimilarity*. In fact, for deterministic LTS', trace equivalence and bisimilarity are equivalent. The definition of bisimilarity is presented below:

Definition A.4.2 (Bisimulation and Bisimilarity [146])**.** A binary relation $\mathcal{R}$ on the states of the LTS is a bisimulation if whenever $P\,\mathcal{R}\,Q$:

1. *for all* P'*, with* $P \xrightarrow{\mu} P'$*, there is* Q' *such that* $Q \to Q'$ *and* $P'\,\mathcal{R}\,Q'$;
2. *the converse, on the transitions emanating from* Q*: for all* Q'*, with* $Q \to Q'$*, there is* P' *such that* $P \to P'$ *and* $P'\,\mathcal{R}\,Q'$;

Bisimilarity, *written* $\sim$*, is the union of all bisimulations; thus* $P \sim Q$ *holds if there is a bisimulation* $\mathcal{R}$ *with* $P\,\mathcal{R}\,Q$*.*

Remark [146]. Note that although bisimulation and bisimilarity are defined on a single LTS, it is also a valid definition for distinct LTS with the same alphabet of actions; as the union of two LTSs is again an LTS.

The definition of bisimilarity immediately suggests a proof technique: to demonstrate that P_1 and P_2 are bisimilar, find a bisimulation relation containing the pair $\langle P1, P2\rangle$.

In the case of CoqDRAM and CavaDRAM, we have to find a bisimulation relation $\mathcal{R}$. As a proof strategy, we start by placing an *arbitrary* pair of states $\langle S_{CoqDRAM}, S_{CavaDRAM}\rangle$ in the relation. Then, instead of trying to add other pairs arbitrarily (as a typical bisimulation proof often proceeds), we rather *suggest* an $\mathcal{R}$ and prove that it is a bisimulation. If our suggested $\mathcal{R}$ is a bisimulation, then we know that $S_{CoqDRAM} \sim S_{CavaDRAM}$ holds. Therefore, $\mathcal{R}$ is defined as the binary predicate `StateEq`, which links CoqDRAM states to CavaDRAM states – as shown in Listing A.16.

The binary relation `StateEq` takes two arguments: the state coming from the CoqDRAM side, of type `FIFOREF_state_t`; and the state coming from the Cava side, of type `State_t`. Moreover, as it can be seen at Line 1, the type of states of `CavaFIFOREF`, `State_t`, is defined through the Cava function `circuit_state` (see Section A.1).

From Lines 5 to 13, we use multiple functions to "*access*" different registers from the top-level circuit, `CavaFIFOREF`. `cava_state`, for instance, defined at Line 4, access the register `state` (depicted in Figure A.7); and `cava_RQ` accesses the request queue.

Next, at Line 14, a pattern matching on `CoqDRAM_state` defines the relation differently for each constructor of `FIFOREF_state_t`. For example, consider the case when `CoqDRAM_state` is `IDLE`, with variables named `cnt`, `cref`, and `Queue`. For the relation to hold it must be true that the Cava state (`cava_state`) is equal to the literal `STATE_IDLE_VEC` (a literal containing the encoding for `IDLE`); that the Cava counter (`cava_cnt`) is equal to the value of `cnt` converted to a bitvector, et cetera. In summary, it is required that every register in the circuit holds a

Listing A.16: Definition of `State_Eq`.

```
Definition State_t := circuit_state CavaFIFOREF.

Definition StateEq (CoqDRAM_state : FIFOREF_state_t) (CavaDRAM_state : State_t) :=
    (* Acessing registers *)
    let cava_state := get_st CavaDRAM_state in
    let cava_cnt   := get_cnt CavaDRAM_state in
    let cava_cref  := get_cref CavaDRAM_state in
    let cava_req   := get_cr CavaDRAM_state in
    let cava_RQ    := get_reqqueue CavaDRAM_state in (* The request queue *)
    (* The write and read addresses from the request queue *)
    let cava_wra := fst (get_addr_RequestQueue RQ) in
    let cava_rda := snd (get_addr_RequestQueue RQ) in
    match CoqDRAM_state with
    | IDLE cnt cref Queue ⇒
        (cava_state =? STATE_IDLE_VEC) &&
        (cava_cnt =? cnt2Bv cnt) &&
        (cava_cref =? cref2Bv cref) &&
        (EqMem Queue cava_rda cava_RQ) &&
        (EqQueue Queue cava_wrda cava_rda)
    | RUNNING c cref Queue r ⇒ (* ... Similar reasoning ... *)
    | REFRESHING cref Queue ⇒ (* ... Similar reasoning ... *)
    end.
```

Listing A.17: Definition of EqQueue.

```
Fixpoint EqQueue (P : Requests_t) (wra rda : Bvector ADDR_WIDTH) :=
    match P with
        | [::] ⇒ (wra =? rda)
        | x :: x0 ⇒ let S_rda := (N2Bv_sized ADDR_WIDTH (Bv2N rda + 1)) in
            negb (wra =? rda) && (EqQueue x0 wra)
    end.
```

value equivalent to the values in the inductive state coming from the CoqDRAM side. The clauses for the RUNNING and REFRESHING cases are omitted for conciseness – since the same logic applies, with only minor differences.

Note, specially, the use of the functions EqMem and EqQueue. The former tests if the entire content of the memory inside of Cava_RQ is mapped to an equivalent element in the Coq list Queue, and vice-versa. The latter, EqQueue, tests if the logic driving the write and read pointers from the Cava request queue is correct. The definition of EqQueue is shown in Listing A.17.

In summary, if P – the request queue coming from the CoqDRAM side – is empty, then the write and read pointers in the Cava FIFO must be equal. Otherwise, the pointers must be different and the function is called recursively for the remainder of the queue, meaning that there are as many elements in P as the absolute difference between wra and rda. The definition of EqMem is more complex and thus omitted for the sake of conciseness.

Finally, the bisimulation theorem is stated as shown in Listing A.18. Start by noticing the two states cava_state and coqdram_state bound by the forall quantifier in Lines 3 and 5, respectively. In addition, since t, the current time stamp, is also bound by a forall quantifier (Line 3), the inputs to the circuit CavaFIFOREF, pending_i and request_t, are also universally quantified by t in Lines 8 and 10, respectively.

Moreover, the theorem relies on one hypothesis: StateEq coqdram_state cava_state. Such hypothesis represents the antecedent in Definition A.4.2, i.e., the "*if whenever* $P \mathcal{R} Q$" part, with P being coqdram_state, $\mathcal{R}$ being StateEq, and Q being cava_state. The theorem's conclusion introduces four new names with a let binder: f_nextstate and f_cmd_o are, respectively, the derivative state of coqdram_state and the issued command, both resulting from a call to Next_state; and the names c_next_state and c_cmd_o are given to the derivative state of cava_state and the issue command, respectively – both resulting from a call to step CavaFIFOREF. We use Cava's step function to simulate the circuit for a single clock transaction.

Furthermore, note that from the axiom HW_arrived, since there is a single arriving request,

Listing A.18: The Bisimulation Theorem.

```
1  Theorem FIFOREF_bisim :
2      (* The state coming from the CavaFIFOREF circuit *)
3         forall (cava_state : State_t),
4      (* The t-th state coming from CoqDRAM *)
5         forall (t : nat), let coqdram_state :=
6             (Default_arbitrate t).(Implementation_State) in
7      (* The incoming request coming from the CavaDRAM side *)
8         let c_req := request_i t in
9      (* The push_i signal coming from the bus protocol interface *)
10        let c_pend := pending_i t in
11     (* ------------------- Hypotheses ------------------- *)
12     (* Hypothesis : coqdram_state and cava_state are equivalent, i.e *)
13        StateEq coqdram_state cava_state →
14     (* ------------------- Conclusion ------------------- *)
15     (* f_nextstate is the state obtained by using CoqDRAM's Next_state and f_cmd_o
          is the command issued at t *)
16        let '(f_nextstate,f_cmd_o) := Next_state (Arrival_at t) coqdram_state in
17     (* c_nextstate is the state obtained by applying a 1-cycle simulation step on
          the CavaFIFOREF circuit and c_cmd_o is the issued command *)
18        let '(c_nextstate,c_cmd_o) := step CavaFIFOREF cava_state (c_pend,c_req) in
19     (* Conclusion : derivative states and issued commands are equal *)
20        (StateEq f_nextstate c_nextstate) && (EqCmd f_cmd_o c_cmd_o).
21 Proof. (* ... omitted ... *) Qed.
```

i.e., `size (HArrival_at)`= 1, `pending_i` is necessarily true, and from axiom `HW_request`, `EqReq (Arrival_at t) c_req` is also true.

The theorem does not include an explicit existential quantifier, as Definition A.4.2 suggests. In fact, we prove a more general statement: since `cava_state` is universally quantified (at Line 3), the proof analyses every possible transition from `cava_state` and proves that there is a corresponding transition from the CoqDRAM side. In addition, besides proving bisimilarity, we also prove that the two LTS' produce equal commands in their outputs. The latter is stated via the `EqCmd` predicate, not shown here for conciseness.

A.5 Hardware Simulation & Synthesis

To gain confidence on the CavaDRAM model, we also perform an experiment on a *hardware setup*. In fact, being able to validate the functionality of the generated RTL code also builds confidence on CoqDRAM, since CavaDRAM and CoqDRAM are logically linked by the behavioural equivalence proof.

As a host environment, we pick a DDR4 controller implementation [126] for *transpecision computing* [147], which will be referred to as *DDR4cntrl* for brevity. In short, the transpecision computing paradigm "offers a dynamic precision reduction to the intermediate stages of a micro-architecture in order to achieve higher energy efficiently, without inheriting any errors in the final output" [126]. To accommodate the needs of such systems, the design of *DDR4cntrl* brings together some sophisticated techniques, such as fine granular refresh control [148] and exploitation of application knowledge [149].

Although *DDR4cntrl* offers an interesting set of features, we only use its simulation structure (i.e. front-end plus physical layer, or PHY), in a similar way to the MCsim experiment (c.f. Chapter 7). In other words, we do not use (most of) *DDR4cntrl*'s logic and command scheduling innovations.

Figure A.10 illustrates the *DDR4cntrl* architecture, with our modifications highlighted. In summary, we replace a module called *rank machine* by the CavaDRAM controller (like a "drop-in" replacement). Originally, the rank machine was responsible for scheduling requests, generating and issuing DRAM commands, as well as managing `REF` commands; the exact same tasks performed by the CavaDRAM controller.

Every other functionality is kept unchanged: the AXI logic and its interface to the controller, the logic to control the read and write data buffers, the PHY, and the memory model. The PHY, a Xilinx IP in this case, generates the signal timing and sequencing required to interface to the memory device, e.g. phase alignment between DQ and DQs signals, logic for initialising the DRAM after power-up, and conversion of slow clock to fast clock.[4]

[4]Inasmuch as the PHY runs at the system clock frequency (1/4 of the DRAM clock frequency), it expects four command/address per system clock and issues them serially on consecutive DRAM clock cycles on the DRAM bus. This means that the PHY interface provides four command slots: 0,1,2, and 3, which it accepts each system clock. To cope with the different clock domains, we insert CavaDRAM commands always in the first slot. The proofs in CoqDRAM do not lose validity, as lower-bounds still hold. The only proofs that

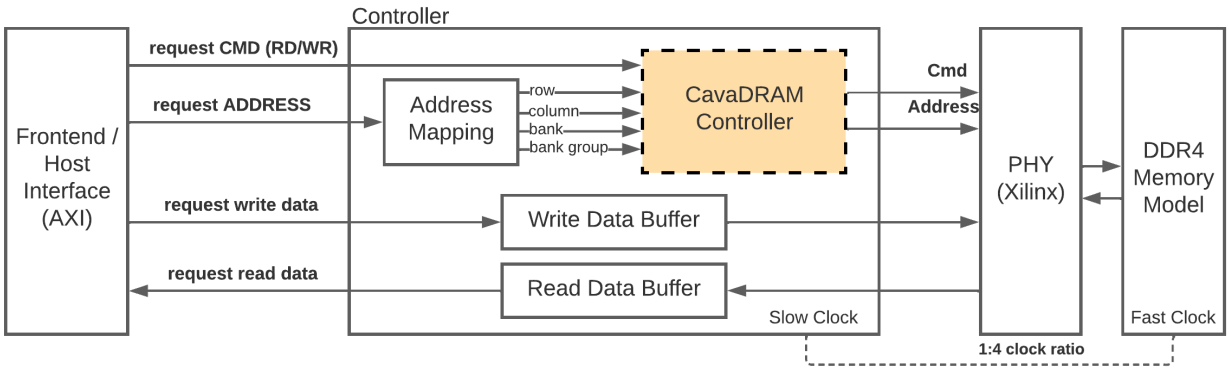


Figure A.10: An illustration of *DDR4cntrl* [126] with our modifications highlighted.

Simulation

We run the test-bench that comes with *DDR4cntrl*. Moreover, the test-bench (and the design) is filled with SystemVerilog Assertions (SVA), which will trigger if timing constraints are not respected, invalid commands are issued, transactions do not complete, or if data integrity is not guaranteed. The latter assertion, regarding data integrity, is a good *end-to-end* check: by checking if data written into the memory can be later accessed by a read to the same address, one can be highly confident that the design works correctly.

The setup was tested using two algorithms: `CavaFIFOREF` and `CavaTDMREF`, the latter being a Cava version of the TDMREF algorithm, which was briefly mentioned in Chapter 8. The parameters for both algorithms (`WAIT` for FIFO, and `SN` and `SL` for TDM) were obtained in a similar way to what has been described in the MCsim experiment description, in Chapter 7.

As an important result, *the CavaDRAM controllers run without triggering any assertions*. Anecdotally, as it has been done for MCsim, we changed the CavaDRAM logic to make sure that it *can* trigger assertions. Figure A.11 shows an example of the modified controller (running `CavaTDMREF` scheduling) succeeding a data integrity assertion.

```
# GOOD
# Written Data = 83e5ba338c4d7a1a37e55dfd222b3b438209291a323bc3c44d7e22219a5d9a122c2bb452127e212b229b33d5b3c7b48480c3c1d59b22d61c9a1e7fd9a162b22f
# Read Data    = 83e5ba338c4d7a1a37e55dfd222b3b438209291a323bc3c44d7e22219a5d9a122c2bb452127e212b229b33d5b3c7b48480c3c1d59b22d61c9a1e7fd9a162b22f
# GOOD
# Written Data = 41f2dd19c626bd0d1bf2aefe91159da1c104948d191de1e226bf1110cd2ecd091615da29093f1095914d99ead9e3da424061e0eacd916b0e4d0f3fecd0b15917
# Read Data    = 41f2dd19c626bd0d1bf2aefe91159da1c104948d191de1e226bf1110cd2ecd091615da29093f1095914d99ead9e3da424061e0eacd916b0e4d0f3fecd0b15917
```

Figure A.11: Data integrity assertions pass (`CavaTDMREF` simulation run).

Furthermore, Figure A.12 shows the `CavaTDMREF` input and output waves (the circuit is called `SM` in the simulation) resulting from one of the simulation runs. Note the port `cmd_o` issuing three commands to an associated request, `CR`. Moreover, in the figure, the TDM slot length (`SL`) is highlighted, as well as the `PRE`, `ACT`, and `CAS` commands being issued to the memory (1F is the encoding for the NOP commands, which do nothing to the memory). One can also see that the signals in `SM` correspond to the interface discussed in Section A.2.

We reiterate that achieving good performance metrics is not the goal of this work, but rather present the methodology and provide a few proof of concepts. With that being said; although RTL code generation from Cava performs as well as standard (modular) SystemVerilog designs, both simulation time and controller bandwidth worsen, for two reasons:

1. The Cava controller was not designed to exploit the 1:4 ratio between controller and memory clocks proposed by Sudarshen et al [126], which also implies lower bandwidth.

need adapting are `REF` related proofs, as they are upper bounds on the spacing between `REF` commands. We write modified version of such constraints considering the different clock domains.

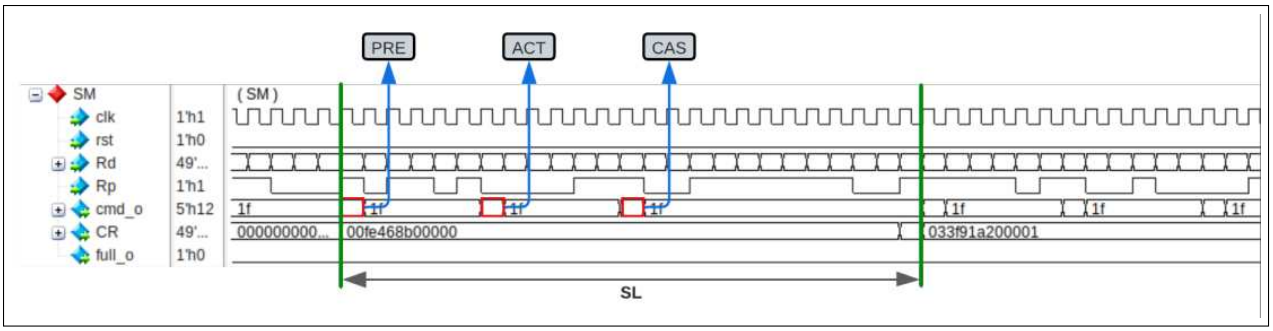


Figure A.12: `CavaTDMREF` simulation – commands being issued to the memory.

As a workaround, we place Cava generated commands in the first of four slots and leave the other slots blank. Note that this does not affect timing correctness, since every lower bound gets multiplied by four in the memory clock domain. Furthermore, Figure A.13 shows how three of the four available slots are not used, and the outputs generated by SM are inserted into slot number 0.

2. Both the TDM and FIFO algorithms employ basic scheduling algorithms and do not offer good bandwidth compared to more complex memory controllers.

Figure A.13: Clock ratio slots usage.

Synthesis

A comparison of synthesis utilisation metrics between the original *DDR4cntrl* and the modified *CavaDRAM* version can be seen in Table A.4. The target FPGA is a Xilinx Virtex UltraScale 095FFVB2104-2 board, with a request queue in CavaDRAM that can store up to 256 requests.

Unsurprisingly, the CavaDRAM version uses fewer LUTs, since the FIFO scheduling logic is simpler than what had been originally proposed in *DDR4cntrl*. The total number of Flip-Flops (FFs) in the design augments, since the Vivado synthesizer chooses to represent Cava queues with FFs rather than native FPGA FIFOs (accessible through the *bram_fifo_52x4*

		LUTs	Flip-Flops	bram_fifo_52x4
DDR4cntrl	`rank_machine`	8195	4644	16
	Full Design	9263	6129	16
CavaDRAM	`CavaDRAM`	5312	8605	0
	Full Design	6344	9832	0

Table A.4: Key metrics from the synthesis report showing the resource utilisation on a Xilinx Virtex UltraScale 095FFVB2104-2.

macro). Note also that *DDR4cntrl* took 88,46% and 75,77% of the total amount of LUTs and FFs in the design, respectively. For *CavaDRAM*, these percentages are 83,73% and 87,52%, respectively. These results show that the generated code introduces only a negligible imbalance w.r.t resource utilisation, compared to the replaced module.

A.6 Lessons Learned

Although the problem and methodology of establishing equivalence between CoqDRAM algorithms and models that can be translated to an RTL description is still relevant, the implementation we envisioned did not end up to be suitable for practical use – due to technical limitations of Cava. In summary, there are three main reasons that motivate our decision to abandon the exploration on CavaDRAM:

1. The most significant barrier is arguably the fact that proofs are carried out in an extremely *slow pace*. Proofs about large circuits (i.e., circuits built with composite and/or nested sub-circuits) involve terms that are lengthy and complex. Due to some poor design decisions in Cava (which have been acknowledge by researchers at Google[5]), simplifying and rewriting large circuits takes a long time (sometimes more than 15 minutes for a single step) – which makes proofs advance in an almost unbearable slow pace. As a side-effect it is virtually impossible to use tools for proof automation with Cava, which often try to apply different tactics in a trial and error fashion.

2. Another factor is that Cava lacks infrastructure/libraries with already-proved properties. This means that we had to manually define several basic circuits, of which the most important is the memory used to implement the request queue. At the time, we wrote an implementation that uses intricate dependent types, which makes the correctness proofs about it very hard (a factor that has been underestimated at the time). In addition, this correlates with choices made in the design of Cava itself. Notably, one

[5]Ben Blaxil, private communication, 2023.

of the main problems is using `Vectors` (from Coq's standard library) to represent bit vectors, which are inherently dependently typed and known to be hard to work with. Worse yet, the memory is implemented using registers on the Cava side, since there is no other primitive to describe storage. The synthesis tool is not able to recognise that this is in fact a memory – leading to costly and inefficient hardware. There is no primitive in Cava that allows to efficiently work with memories, neither on the proof, nor on the synthesis side.

3. The fact that the development of Cava has stopped since late 2022 also makes it a poor choice for carrying out this type of research. This means that there is no support for lacking features, no hope to get any of the aforementioned problems fixed, and no updates to make the DSL conform to more recent Coq versions.

One possible direction would be to change some things in Cava itself, such as replacing `Vectors` with length-constrained lists (using sigma types, for instance) and changing how states on sequential composite circuits are represented. On a side note, the SilverOak project had started developing second version of Cava by late 2021, which aimed at addressing some of the limitations of the first version of Cava, but unfortunately, the project has been stopped at a very early stage as well. For all these reasons I would not recommend to use Cava to derive HW for CoqDRAM algorithms. As outlined in Chapter 10, this remains an open problem and other options (Kami/Koika) should be explored to accomplish this instead.

Although this is a somewhat unsatisfying result, many lessons were taken from the experience. For instance, the difficulty of handling proofs about definitions containing dependent types influenced the design of Interface Sub-Layer, described in Chapter 8, and `TDMShelve`, described in Chapter 9. Choosing to keep a clean separation between programs and proofs allowed us to successfully implement the Interface Sub-Layer, which led to a clean, modular, implementation of `TDMShelve`. Moreover, deriving real hardware implementations also led to the realisation that managing refresh commands was required, which greatly influenced the design of the second iteration of CoqDRAM.

Finally, in Chapter 10, we proposed some possible future work directions that aim at solving the problem that CavaDRAM attempted to solve.

References

[1] A. Australia. "Instrument landing system for gold coast airport." (2020), [Online]. Available: https://engage.airservicesaustralia.com/2019-changes/news_feed/instrument-landing-system-for-gold-coast-airport.

[2] L. Rierson, *Developing safety-critical software: a practical guide for aviation software and DO-178C compliance*. CRC Press, 2017.

[3] N. M. Enache, M. Netto, S. Mammar, and B. Lusetti, "Driver steering assistance for lane departure avoidance," *Control engineering practice*, vol. 17, no. 6, pp. 642–651, 2009.

[4] P. Seiler, B. Song, and J. K. Hedrick, "Development of a collision avoidance system," *SAE transactions*, pp. 1334–1340, 1998.

[5] R. Zhang, K. Li, Z. He, H. Wang, and F. You, "Advanced emergency braking control based on a nonlinear model predictive algorithm for intelligent vehicles," *Applied sciences*, vol. 7, no. 5, p. 504, 2017.

[6] L. Xiao and F. Gao, "A comprehensive review of the development of adaptive cruise control systems," *Vehicle system dynamics*, vol. 48, no. 10, pp. 1167–1192, 2010.

[7] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *28th IEEE international real-time systems symposium (RTSS 2007)*, IEEE, 2007, pp. 239–243.

[8] S. Buchanan. "How many semiconductor chips are there in a car?" (2022), [Online]. Available: https://economistwritingeveryday.com/2022/01/04/how-many-semiconductor-chips-are-there-in-a-car/.

[9] J. P. Cerrolaza, R. Obermaisser, J. Abella, F. J. Cazorla, K. Grüttner, I. Agirre, H. Ahmadian, and I. Allende, "Multi-core devices for safety-critical systems: A survey," *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–38, 2020.

[10] J. Perez, D. Gonzalez, C. F. Nicolas, T. Trapman, and J. M. Garate, "A safety certification strategy for iec-61508 compliant industrial mixed-criticality systems based on multicore partitioning," in *2014 17th Euromicro Conference on Digital System Design*, IEEE, 2014, pp. 394–400.

[11] S. Chakraborty and S. Ramesh, "Guest editorial special section on automotive embedded systems and software," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 11, pp. 1701–1703, 2015.

[12] G. Macher, A. Höller, E. Armengaud, and C. Kreiner, "Automotive embedded software: Migration challenges to multi-core computing platforms," in *2015 IEEE 13th International Conference on Industrial Informatics (INDIN)*, IEEE, 2015, pp. 1386–1393.

[13] D. Radack, H. G. Tiedeman, and P. Parkinson, "Civil certification of multi-core processing systems in commercial avionics," in *27th Safety-critical Systems Symposium Proceedings*, 2018.

[14] G. Fernandez, J. Abella, E. Quiñones, C. Rochange, T. Vardanega, and F. J. Cazorla, "Contention in multicore hardware shared resources: Understanding of the state of the art," in *14th International Workshop on Worst-Case Execution Time Analysis (WCET 2014)*, 2014, pp. 31–42.

[15] J. A. Stankovic and K. Ramamritham, "What is predictability for real-time systems?" *Real-Time Systems*, vol. 2, no. 4, pp. 247–254, 1990.

[16] D. Guo, M. Hassan, R. Pellizzoni, and H. Patel, "A comparative study of predictable dram controllers," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 17, no. 2, pp. 1–23, 2018.

[17] R. Mirosanlou, M. Hassan, and R. Pellizzoni, "Drambulism: Balancing performance and predictability through dynamic pipelining," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, 2020, pp. 82–94.

[18] J. Jalle, E. Quinones, J. Abella, L. Fossati, M. Zulianello, and F. J. Cazorla, "A dual-criticality memory controller (dcmc): Proposal and evaluation of a space case study," in *2014 IEEE Real-Time Systems Symposium*, IEEE, 2014, pp. 207–217.

[19] R. Mirosanlou, M. Hassan, and R. Pellizzoni, "Duetto: Latency guarantees at minimal performance cost," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2021, pp. 1136–1141.

[20] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero, "An analyzable memory controller for hard real-time cmps," *IEEE Embedded Systems Letters*, vol. 1, no. 4, pp. 86–90, 2009.

[21] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "Pret dram controller: Bank privatization for predictability and temporal isolation," in *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, 2011, pp. 99–108.

[22] A. Schranzhofer, R. Pellizzoni, J.-J. Chen, L. Thiele, and M. Caccamo, "Timing analysis for resource access interference on adaptive resource arbiters," in *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, IEEE, 2011, pp. 213–222.

[23] S. Goossens, B. Akesson, and K. Goossens, "Conservative open-page policy for mixed time-criticality memory controllers," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2013, pp. 525–530.

[24] Z. P. Wu, Y. Krish, and R. Pellizzoni, "Worst case analysis of dram latency in multi-requestor systems," in *2013 IEEE 34th Real-Time Systems Symposium*, IEEE, 2013, pp. 372–383.

[25] L. Ecco, S. Tobuschat, S. Saidi, and R. Ernst, "A mixed critical memory controller using bank privatization and fixed priority scheduling," in *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*, IEEE, 2014, pp. 1–10.

[26] Y. Krishnapillai, Z. P. Wu, and R. Pellizzoni, "A rank-switching, open-row dram controller for time-predictable systems," in *2014 26th Euromicro Conference on Real-Time Systems*, IEEE, 2014, pp. 27–38.

[27] Y. Li, B. Akesson, and K. Goossens, "Dynamic command scheduling for real-time memory controllers," in *2014 26th Euromicro Conference on Real-Time Systems*, IEEE, 2014, pp. 3–14.

[28] L. Ecco and R. Ernst, "Improved dram timing bounds for real-time dram controllers with read/write bundling," in *2015 IEEE Real-Time Systems Symposium*, IEEE, 2015, pp. 53–64.

[29] M. Hassan, H. Patel, and R. Pellizzoni, "A framework for scheduling dram memory accesses for multi-core mixed-time critical systems," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, IEEE, 2015, pp. 307–316.

[30] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh, "A predictable and command-level priority-based dram controller for mixed-criticality systems," in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, IEEE, 2015, pp. 317–326.

[31] P. K. Valsan and H. Yun, "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, 2015, pp. 86–93.

[32] H. Yun, R. Pellizzon, and P. K. Valsan, "Parallelism-aware memory interference delay analysis for cots multicore systems," in *2015 27th Euromicro Conference on Real-Time Systems*, IEEE, 2015, pp. 184–195.

[33] L. Ecco, A. Kostrzewa, and R. Ernst, "Minimizing dram rank switching overhead for improved timing bounds and performance," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, IEEE, 2016, pp. 3–13.

[34] D. Guo and R. Pellizzoni, "A requests bundling dram controller for mixed-criticality systems," in *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, 2017, pp. 247–258.

[35] M. Hassan, H. Patel, and R. Pellizzoni, "Pmc: A requirement-aware dram controller for multicore mixed criticality systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, no. 4, pp. 1–28, 2017.

[36] M. Hassan and R. Pellizzoni, "Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2323–2336, 2018.

[37] K. Asifuzzaman, M. Abuelala, M. Hassan, and F. J. Cazorla, "Demystifying the characteristics of high bandwidth memory for real-time systems," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, IEEE, 2021, pp. 1–9.

[38] F. Lisboa Malaquias, M. Asavoae, and F. Brandner, "A formal framework to design and prove trustworthy memory controllers," *Real-Time Systems*, vol. 59, no. 4, pp. 664–704, 2023.

[39] F. Hebbache, M. Jan, F. Brandner, and L. Pautet, "Shedding the shackles of time-division multiplexing," in *2018 IEEE Real-Time Systems Symposium (RTSS)*, IEEE, 2018, pp. 456–468.

[40] F. Hebbache, F. Brandner, M. Jan, and L. Pautet, "Work-conserving dynamic time-division multiplexing for multi-criticality systems," *Real-Time Systems*, vol. 56, pp. 124–170, 2020.

[41] J. E. D. E. C. (JEDEC), "DDR3 SDRAM (JESD 79-3) standard," JEDEC, 2012.

[42] J. E. D. E. C. (JEDEC), "DDR4 SDRAM standard," JEDEC, 2021.

[43] F. Lisboa Malaquias, M. Asavoae, and F. Brandner, "A coq framework for more trustworthy dram controllers," in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, 2022, pp. 140–150.

[44] R. Mirosanlou, D. Guo, M. Hassan, and R. Pellizzoni, "Mcsim: An extensible dram memory controller simulator," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 105–109, 2020.

[45] J. A. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems," *Computer*, vol. 21, no. 10, pp. 10–19, 1988.

[46] H. Koptez, "Real-time systems: Design principles for distributed embedded applications," *Kluwer Academic Publisher*, 1997.

[47] D. Kästner, M. Schlickling, M. Pister, C. Cullmann, G. Gebhard, R. Heckmann, and C. Ferdinand, "Meeting real-time requirements with multi-core processors," in *Computer Safety, Reliability, and Security: SAFECOMP 2012 Workshops: Sassur, ASCoMS, DESEC4LCCI, ERCIM/EWICS, IWDE, Magdeburg, Germany, September 25-28, 2012. Proceedings 31*, Springer, 2012, pp. 117–131.

[48] ARM. "Cortex-m3 technical reference manual r2p0." (), [Online]. Available: https://developer.arm.com/documentation/ddi0337/h/programmers-model/processor-core-register-summary.

[49] C. Su and Q. Zeng, "Survey of cpu cache-based side-channel attacks: Systematic analysis, security models, and countermeasures," *Security and Communication Networks*, vol. 2021, pp. 1–15, 2021.

[50] K. Webb. "Cs31 : Introduction to computer systems, fall 2018." (2018), [Online]. Available: https://www.cs.swarthmore.edu/~kwebb/cs31/f18/memhierarchy/mem_hierarchy.html.

[51] O. Kotaba, J. Nowotsch, M. Paulitsch, S. M. Petters, and H. Theiling, "Multicore in real-time systems–temporal isolation challenges due to shared resources," in *16th Design, Automation & Test in Europe Conference and Exhibition*, 2013.

[52] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A predictable sdram memory controller," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, 2007, pp. 251–256.

[53] M. Paolieri, E. Quinones, and F. J. Cazorla, "Timing effects of ddr memory systems in hard real-time multicore architectures: Issues and solutions," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 1s, pp. 1–26, 2013.

[54] H. P. Barendregt *et al.*, *The lambda calculus*. North-Holland Amsterdam, 1984, vol. 3.

[55] R. W. Floyd, "Assigning meanings to programs," in *Program Verification: Fundamental Issues in Computer Science*, Springer, 1993, pp. 65–81.

[56] J. L. Peterson, "Petri nets," *ACM Computing Surveys (CSUR)*, vol. 9, no. 3, pp. 223–252, 1977.

[57] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

[58] M. C. Browne, E. M. Clarke, and O. Grümberg, "Characterizing finite kripke structures in propositional temporal logic," *Theoretical computer science*, vol. 59, no. 1-2, pp. 115–131, 1988.

[59] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, ieee, 1977, pp. 46–57.

[60] T. Hafer and W. Thomas, "Computation tree logic ctl* and path quantifiers in the monadic theory of the binary tree," in *International Colloquium on Automata, Languages, and Programming*, Springer, 1987, pp. 269–279.

[61] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking for real-time systems," in *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*, IEEE, 1990, pp. 414–425.

[62] "Ieee standard for property specification language (psl)," *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)*, pp. 1–182, 2010. DOI: 10.1109/IEEESTD.2010.5446004.

[63] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *LASER Summer School on Software Engineering*, Springer, 2011, pp. 1–30.

[64] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on logic of programs*, Springer, 1981, pp. 52–71.

[65] J.-P. Queille and J. Sifakis, "Specification and verification of concurrent systems in cesar," in *International Symposium on programming*, Springer, 1982, pp. 337–351.

[66] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks, "Uppaal 4.0," 2006.

[67] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings 14*, Springer, 2002, pp. 359–364.

[68] G. J. Holzmann, "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[69] Y. Yu, P. Manolios, and L. Lamport, "Model checking tla+ specifications," in *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, Springer, 1999, pp. 54–66.

[70] R. Hähnle and M. Huisman, "Deductive software verification: From pen-and-paper proofs to industrial tools," *Computing and Software Science: State of the Art and Perspectives*, pp. 345–373, 2019.

[71] M. H. Sørensen and P. Urzyczyn, *Lectures on the Curry-Howard isomorphism*. Elsevier, 2006.

[72] H. B. Curry, "Functionality in combinatory logic," *Proceedings of the National Academy of Sciences*, vol. 20, no. 11, pp. 584–590, 1934.

[73] D. W. Loveland, *Automated theorem proving: A logical basis*. Elsevier, 2016.

[74] C. Paulin-Mohring, *Introduction to the calculus of inductive constructions*, 2015.

[75] L. Paulson. "The de bruijn criterion vs the lcf architecturen." (2022), [Online]. Available: https://lawrencecpaulson.github.io/2022/01/05/LCF.html.

[76] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.

[77] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, Springer, 2011, pp. 171–177.

[78] A. Chlipala, *Certified programming with dependent types: a pragmatic introduction to the Coq proof assistant*. MIT Press, 2022.

[79] A. Datta and V. Singhal, "Formal verification of a public-domain ddr2 controller design," in *21st International Conference on VLSI Design (VLSID 2008)*, IEEE, 2008, pp. 475–480.

[80] M. Jung, K. Kraft, T. Soliman, C. Sudarshan, C. Weis, and N. Wehn, "Fast validation of dram protocols with timed petri nets," in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 133–147.

[81] L. Jacobsen, M. Jacobsen, M. H. Møller, and J. Srba, "Verification of timed-arc petri nets," in *SOFSEM 2011: Theory and Practice of Computer Science: 37th Conference on Current Trends in Theory and Practice of Computer Science, Novỳ Smokovec, Slovakia, January 22-28, 2011. Proceedings 37*, Springer, 2011, pp. 46–72.

[82] C. A. Petri, "Kommunikation mit automaten," 1962.

[83] L. Steiner, C. Sudarshan, M. Jung, D. Stoffel, and N. Wehn, "A framework for formal verification of dram controllers," *arXiv preprint arXiv:2209.14021*, 2022.

[84] M. O. Kayed, M. Abdelsalam, and R. Guindi, "A novel approach for sva generation of ddr memory protocols based on tdml," in *2014 15th International Microprocessor Test and Verification Workshop*, IEEE, 2014, pp. 61–66.

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

[86] Y. Li, B. Akesson, and K. Goossens, "Architecture and analysis of a dynamically-scheduled real-time memory controller," *Real-Time Systems*, vol. 52, pp. 675–729, 2016.

[87] M. Hassan and H. Patel, "Mcxplore: An automated framework for validating memory controller designs," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2016, pp. 1357–1362.

[88] M. Hassan and H. Patel, "Mcxplore: Automating the validation process of dram memory controller designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 5, pp. 1050–1063, 2017.

[89] D. Sahoo, S. Sha, M. Satpathy, M. Mutyam, S. Ramesh, and P. Roop, "Formal modeling and verification of controllers for a family of dram caches," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2485–2496, 2018.

[90] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002, vol. 2283.

[91] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, *et al.*, "Dependent types and multi-monadic effects in f," in *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 256–270.

[92] F. Cerqueira, F. Stutz, and B. B. Brandenburg, "Prosa: A case for readable mechanized schedulability analysis," in *Euromicro Conference on Real-Time Systems*, ser. ECRTS'16, IEEE, 2016, pp. 273–284.

[93] X. Guo, M. Lesourd, M. Liu, L. Rieg, and Z. Shao, "Integrating formal schedulability analysis into a verified os kernel," in *Computer Aided Verification*, ser. CAV'19, Springer, 2019, pp. 496–514, ISBN: 978-3-030-25543-5.

[94] S. Bozhko and B. B. Brandenburg, "Abstract response-time analysis: A formal foundation for the busy-window principle," 2020.

[95] M. Maida, S. Bozhko, and B. B. Brandenburg, "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, 2022.

[96] J. Choi, M. Vijayaraghavan, B. Sherman, A. Chlipala, and Arvind, "Kami: A platform for high-level parametric hardware specification and its modular verification," *Proceedings of the ACM on Programming Languages*, vol. 1, no. ICFP, pp. 1–30, 2017.

[97] SiFive. "Formal specification of risc-v isa in kami." (), [Online]. Available: https://github.com/sifive/RiscvSpecFormal.

[98] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, "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*, 2020, pp. 243–257.

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

[100] W. L. Harrison, C. Hathhorn, and G. Allwein, "A mechanized semantic metalanguage for high level synthesis," in *23rd International Symposium on Principles and Practice of Declarative Programming*, 2021, pp. 1–14.

[101] W. Harrison, I. Blumenfeld, E. Bond, C. Hathhorn, P. Li, M. Torrence, and J. Ziegler, "Formalized high level synthesis with applications to cryptographic hardware," in *NASA Formal Methods Symposium*, Springer, 2023, pp. 332–352.

[102] C. Wolf. "Yosys open synthesis suite." (), [Online]. Available: https://yosyshq.net/yosys/.

[103] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "C? ash: Structural descriptions of synchronous hardware using haskell," in *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, IEEE, 2010, pp. 714–721.

[104] R. Mirosanlou, M. Hassan, and R. Pellizzoni, "Duomc: Tight dram latency bounds with shared banks and near-cots performance," in *The International Symposium on Memory Systems*, 2021, pp. 1–16.

[105] I. Bhati, M.-T. Chang, Z. Chishti, S.-L. Lu, and B. Jacob, "Dram refresh mechanisms, penalties, and trade-offs," *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 108–121, 2015.

[106] Micron, "Tn-40-40: Ddr4 point-to-point design guide," Micron, 2020.

[107] SystemVerilog.io. "Ddr4 sdram - initialization, training and calibration." (), [Online]. Available: https://www.systemverilog.io/design/ddr4-initialization-and-calibration/.

[108] S. Schlachter and B. Drake, "Introducing micron® ddr5 sdram: More than a generational update," *XP055844818, May*, vol. 31, p. 6, 2019.

[109] J. E. D. E. C. (JEDEC), "LPDDR4 (JESD 209-4) standard," JEDEC, 2014.

[110] J. Avigad, "Classical and constructive logic," *Available from the Internet at the author's website*, pp. 1–23, 2000.

[111] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers c-28*, vol. 9, pp. 690–691, 1979.

[112] A. Sezgin, *Formalization and verification of shared memory*. The University of Utah, 2004.

[113] P. A. Abdulla, A. P. Sistla, and M. Talupur, "Model checking parameterized systems," *Handbook of model checking*, pp. 685–725, 2018.

[114] S. Boldo, F. Clément, F. Faissole, V. Martin, and M. Mayero, "A coq formal proof of the lax-milgram theorem," in *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, 2017, pp. 79–89.

[115] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[116] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "Macsim: A cpu-gpu heterogeneous simulation framework user guide," *Georgia Institute of Technology*, 2012.

[117] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer architecture letters*, vol. 15, no. 1, pp. 45–49, 2015.

[118] X. Xin, Y. Zhang, and J. Yang, "Roc: Dram-based processing with reduced operation cycles," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[119] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 128–138, 2000.

[120] N. G. Leveson and J. P. Thomas, "Certification of safety-critical systems," *Commun. ACM*, vol. 66, no. 10, pp. 22–26, Sep. 2023, ISSN: 0001-0782. DOI: 10.1145/3615860. [Online]. Available: https://doi.org/10.1145/3615860.

[121] B. C. Pierce, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey, "Software foundations," *Webpage: http://www. cis. upenn. edu/bcpierce/sf/current/index. html*, p. 16, 2010.

[122] A. Chlipala, "Formal reasoning about programs," *url: http://adam. chlipala. net/frap*, 2017.

[123] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[124] M.-M. Bidmeshki and Y. Makris, "Vericoq: A verilog-to-coq converter for proof-carrying hardware automation," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, IEEE, 2015, pp. 29–32.

[125] J. CHOI, "Formal semantics of verilog, revisited for deductive verification,"

[126] C. Sudarshan, J. Lappas, C. Weis, D. M. Mathew, M. Jung, and N. Wehn, "A lean, low power, low latency dram memory controller for transprecision computing," in *Embedded Computer Systems: Architectures, Modeling, and Simulation: 19th International Conference, SAMOS 2019, Samos, Greece, July 7–11, 2019, Proceedings 19*, Springer, 2019, pp. 429–441.

[127] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand, "Compcert-a formally verified optimizing compiler," in *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[128] P. O'Hearn, "Separation logic," *Communications of the ACM*, vol. 62, no. 2, pp. 86–95, 2019.

[129] L. de Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer, "The lean theorem prover (system description)," in *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, Springer, 2015, pp. 378–388.

[130] F. Wiedijk, "Encoding the hol light logic in coq," *Unpublished notes*, 2007.

[131] P. A. S. E. Andrej Bauer. "What is a deep embedding vs a shallow embedding? with examples?" (), [Online]. Available: https://proofassistants.stackexchange.com/questions/2499/what-is-a-deep-embedding-vs-a-shallow-embedding-with-examples.

[132] R. Krebbers, X. Leroy, and F. Wiedijk, "Formal c semantics: Compcert and the c standard," in *Interactive Theorem Proving: 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings 5*, Springer, 2014, pp. 543–548.

[133] A. Anand, A. Appel, G. Morrisett, Z. Paraskevopoulou, R. Pollack, O. S. Belanger, M. Sozeau, and M. Weaver, "Certicoq: A verified compiler for coq," in *The third international workshop on Coq for programming languages (CoqPL)*, 2017.

[134] M. Sozeau, A. Anand, S. Boulier, C. Cohen, Y. Forster, F. Kunze, G. Malecha, N. Tabareau, and T. Winterhalter, "The metacoq project," *Journal of automated reasoning*, vol. 64, no. 5, pp. 947–999, 2020.

[135] K. E. Gray, P. Sewell, C. Pulte, S. Flur, and R. Norton-Wright, "The sail instruction-set semantics specification language," Technical report published by Cambridge University, Tech. Rep., 2017.

[136] T. Bourgeat, I. Clester, A. Erbsen, S. Gruetter, A. Wright, and A. Chlipala, "A multipurpose formal risc-v specification," *arXiv preprint arXiv:2104.00762*, 2021.

[137] A. Fromherz, N. Giannarakis, C. Hawblitzel, B. Parno, A. Rastogi, and N. Swamy, "A verified, efficient embedding of a verifiable assembly language," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–30, 2019.

[138] J. Protzenko, J.-K. Zinzindohoué, A. Rastogi, T. Ramananandro, P. Wang, S. Zanella-Béguelin, A. Delignat-Lavaud, C. Hriţcu, K. Bhargavan, C. Fournet, *et al.*, "Verified low-level programming embedded in f," *Proceedings of the ACM on Programming Languages*, vol. 1, no. ICFP, pp. 1–29, 2017.

[139] G. Research. "Silver oak." (), [Online]. Available: https://github.com/project-oak/silveroak.

[140] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in haskell," *ACM SIGPLAN Notices*, vol. 34, no. 1, pp. 174–184, 1998.

[141] R. Nikhil, "Bluespec system verilog: Efficient, correct rtl from high level specifications," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.*, IEEE, 2004, pp. 69–70.

[142] J. Gibbons and N. Wu, "Folding domain-specific languages: Deep and shallow embeddings (functional pearl)," in *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, 2014, pp. 339–347.

[143] D. Park, "A new equivalence notion for communicating systems," *Bulletin EATCS*, vol. 14, pp. 78–80, 1981.

[144] R. Milner, *A calculus of communicating systems*. Springer, 1980.

[145] D. Pous and D. Sangiorgi, "Bisimulation and coinduction enhancements: A historical perspective," *Formal Aspects of Computing*, vol. 31, no. 6, pp. 733–749, 2019.

[146] D. Sangiorgi, *Introduction to bisimulation and coinduction*. Cambridge University Press, 2011.

[147] A. C. I. Malossi, M. Schaffner, A. Molnos, L. Gammaitoni, G. Tagliavini, A. Emerson, A. Tomás, D. S. Nikolopoulos, E. Flamand, and N. Wehn, "The transprecision computing paradigm: Concept, design, and applications," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2018, pp. 1105–1110.

[148] D. M. Mathew, É. F. Zulian, M. Jung, K. Kraft, C. Weis, B. Jacob, and N. Wehn, "Using run-time reverse-engineering to optimize dram refresh," in *Proceedings of the International Symposium on Memory Systems*, 2017, pp. 115–124.

[149] M. Jung, D. M. Mathew, C. Weis, N. Wehn, I. Heinrich, M. V. Natale, and S. O. Krumke, "Congen: An application specific dram memory controller generator," in *Proceedings of the Second International Symposium on Memory Systems*, 2016, pp. 257–267.

ECOLE DOCTORALE

Titre : CoqDRAM – Une Fondation pour la Conception de Contrôleurs de Mémoire Formellement Prouvés

Mots clés : Méthodes Formelles, Contrôleurs Mémoire, DRAM, Systèmes Temps-Réel, Ordonnancement

Résumé : Les contrôleurs mémoire temps-réel récemment proposés abordent le compromis entre performance et prédictibilité en cherchant à offrir le meilleur des deux mondes. Cependant, en conséquence, les conceptions deviennent complexes et présentent souvent des développements mathématiques qui sont longs, difficiles à lire et à examiner, incomplets, et reposent sur des hypothèses peu claires. Étant donné que de tels composants sont souvent conçus comme faisant partie de micro-architectures utilisées dans des systèmes temps-réel critiques, un degré élevé de confiance dans le comportement correct du système est nécessaire pour atteindre les objectifs de certification. Pour résoudre ce problème, nous proposons un nouveau *framework*, intitulé CoqDRAM, écrit dans l'assistant de preuves formelles Coq, dans lequel nous modélisons les dispositifs DRAM et les contrôleurs et leur comportement attendu en tant que spécification formelle. Le *framework* est destiné à aider à la conception d'algorithmes d'ordonnancement DRAM corrects par construction et dignes de confiance. La spécification CoqDRAM capture les critères de correction selon les normes JEDEC et énonce d'autres propriétés de haut niveau, telles que l'équité (*fairness*) et la cohérence séquentielle (*sequential consistency*). Suivant cette approche, les développements mathématiques sur papier-et-crayon sont remplacés par des preuves vérifiées par machine, ce qui accroît la confiance que la conception est effectivement correcte. Nous présentons l'utilisabilité de CoqDRAM en modélisant et en prouvant deux algorithmes d'ordonnancement de principe: l'un basé sur la politique d'arbitrage First-In First-Out (FIFO) et l'autre sur la multiplexage par répartition dans le temps (TDM). De plus, en utilisant CoqDRAM, nous proposons un nouvel algorithme d'ordonnancement DRAM appelé TDMShelf, qui étend et améliore les travaux précédents sur l'arbitrage *work-conserving* dynamique TDM. Plus précisément, TDMShelf exploite des informations sur l'état interne de la mémoire au niveau de l'ordonnancement des requêtes mémoire, fournissant ainsi un bon équilibre entre prédictibilité et latence moyenne pour les systèmes temps réel à criticité mixte.

Title : CoqDRAM – A Foundation for Designing Formally Proven Memory Controllers

Keywords : Formal Methods, Memory Controller, DRAM, Real-Time Systems, Scheduling

Abstract : Recently proposed real-time memory controllers tackle the performance-predictability trade-off by trying to offer the best of both worlds. However, as a consequence, designs have become complex and often present mathematical developments that are lengthy, hard to read and review, incomplete, and rely on unclear assumptions. Given that such components are often designed as part micro-architectures that are used in safety-critical real-time systems, a high degree of confidence that systems behave *correctly* is required in order to meet certification goals. To address that problem, we propose a new framework written in the Coq theorem prover named *CoqDRAM*, in which we model DRAM devices and controllers and their expected behaviour as a formal specification. The framework is intended to aid the design of correct-by-construction, *trustworthy* DRAM scheduling algorithms. The CoqDRAM specification captures correctness criteria according to the JEDEC standards and states other high-level properties, such as fairness and sequential consistency. Following such approach, paper-and-pencil mathematical developments are replaced by machine-checked proofs, which increase confidence that the design is indeed correct.We showcase CoqDRAM's usability by modelling and proving two proof of concept scheduling algorithms: one based on the *First-in First-Out* (FIFO) arbitration policy and the other on *Time-Division Multiplexing* (TDM). Moreover, using CoqDRAM, we propose a new DRAM scheduling algorithm called *TDMShelve*, which extends and improves previous work on *work-conserving* dynamic TDM arbitration. More specifically, TDMShelve exploits information about the internal state of the memory at request scheduling level, thus providing a good balance between predictability and average-case latency for mixed-criticality real-time systems. Finally, we connect the algorithms written in CoqDRAM to software and hardware simulation environments. These environments are used to perform simulation runs that further validate the correctness of the CoqDRAM model.

Institut Polytechnique de Paris
91120 Palaiseau, France