Fast Validation of DRAM Protocols with Timed Petri Nets

Matthias Jung
Fraunhofer Institute for Experimental Software Engineering (IESE)
Kaiserslautern, Germany
matthias.jung@iese.fraunhofer.de

Kira Kraft
Microelectronic Systems Design Research Group, TU Kaiserslautern
Kaiserslautern, Germany
kraft@eit.uni-kl.de

Taha Soliman
Microelectronic Systems Design Research Group, TU Kaiserslautern
Kaiserslautern, Germany
tsoliman@rhrk.uni-kl.de

Chirag Sudarshan
Microelectronic Systems Design Research Group, TU Kaiserslautern
Kaiserslautern, Germany
sudarshan@eit.uni-kl.de

Christian Weis
Microelectronic Systems Design Research Group, TU Kaiserslautern
Kaiserslautern, Germany
weis@eit.uni-kl.de

Norbert Wehn
Microelectronic Systems Design Research Group, TU Kaiserslautern
Kaiserslautern, Germany
wehn@eit.uni-kl.de

ABSTRACT

In recent years, an increasing number of different JEDEC memory standards, like DDR4/5, LPDDR4/5, GDDR6, Wide I/O2, HBM2, and NVDIMM-P have been specified, which differ significantly from the previous ones like DDR3 and LPDDR3. Since each new standard comes with significant changes in the DRAM protocol compared to the previous ones, the developers of memory controllers and memory simulation models regularly face challenges implementing and verifying these new standards. In order to keep pace with these frequent changes of the requirements and the large variety of variants a robust validation methodology must be established. The JEDEC standards describe the complex memory protocol, i.e., DRAM commands and their timing dependencies, by using a mixture of state machine diagrams, tables, and timing diagrams. However, there exists no unique formal description of the JEDEC standards which could be used for a fast simulation-based validation. In this paper, for the first time, we present a comprehensive and formal mathematical model based on Petri Nets that contains the DRAM states, transitions, and timings. Furthermore, we present a *Domain Specific Language* (DSL) for describing the memory functionality and timing dependencies of a JEDEC standard in just a few lines of code. From this DSL description an executable Petri Net is generated automatically, which is used for the fast simulation-based validation of memory controllers and DRAM simulation models.

CCS CONCEPTS

• **Computing methodologies → Modeling and simulation**; • **Hardware → Dynamic memory**; **Semi-formal verification**; *Simulation and emulation*; *Assertion checking*;

KEYWORDS

DRAM, Memory Controller, Validation, Petri Net

ACM Reference Format:
Matthias Jung, Kira Kraft, Taha Soliman, Chirag Sudarshan, Christian Weis, and Norbert Wehn. 2019. Fast Validation of DRAM Protocols with Timed Petri Nets. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19), September 30-October 3, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3357526.3357556

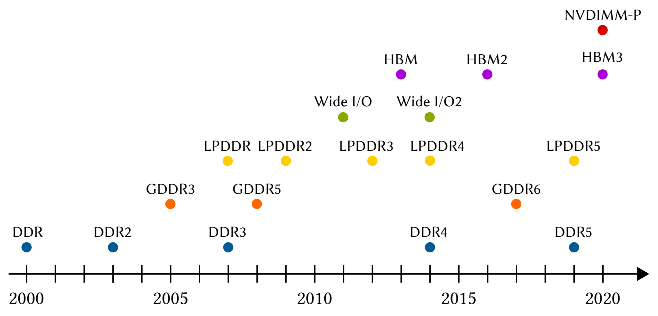

Figure 1: Releases of JEDEC Standards

1 INTRODUCTION

Over the last decades, the number of DRAM standards specified by the *Joint Electron Devices Engineering Councils* (JEDEC) is largely growing, as shown in Figure 1. With the advent of *Storage Class Memories* (SCM), the number of main memory standards will increase even further (e.g., RRAM, STT-MRAM, 3D XPoint), leading to the same amount of heterogeneity that we can already see on the compute side (e.g., CPUs, GPUs, TPUs, and custom accelerators).

Whenever a new standard is released, developers of memory controllers and memory simulation models must adopt to the changes of the new protocol in order to guarantee correct functionality of the DRAM controller according to the new standard. This adoption process usually requires tedious work adjusting the implementation of the memory controller and, even more importantly, verifying its standard conformity. An example is the introduction of variable command lengths in LPDDR4 [31], which was a significant change since the command length was always fixed to one clock cycle in previous standards. In order to keep pace with these frequent changes of the requirements and the large variety of variants, a quick validation is mandatory to enable fast time to market with new controllers. For the validation the developers apply either

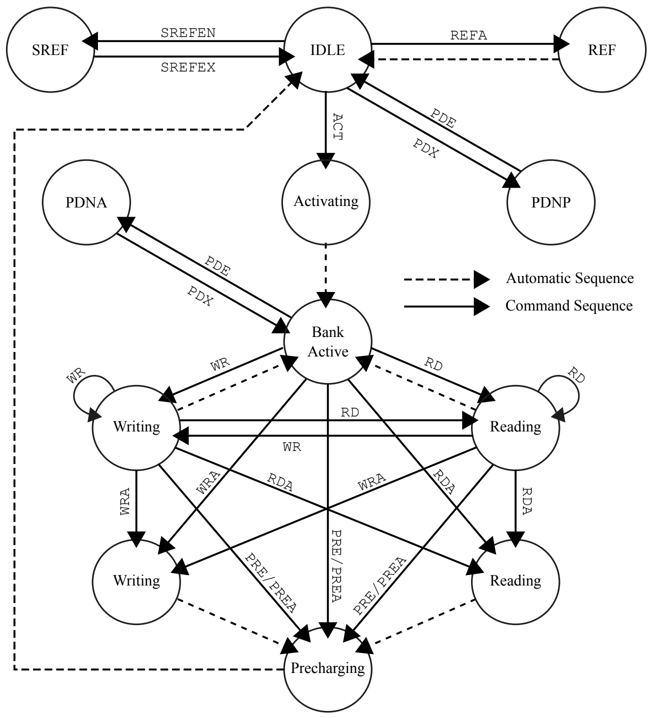

Figure 2: State Diagram of DRAM Commands According to JEDEC Standards (Initialization States are Omitted)

simulation-based or even formal verification approaches. The formal verification approach is complete but has the drawbacks that it has long run-times and works only for circuits with a moderate size. The simulation-based approach is not necessarily complete but fast. It requires input stimuli for the model under test, which generates outputs that are compared to expected values. The challenges for the simulation-based validation of memory controllers are first the generation of meaningful input stimuli and second to check if the produced outputs are compliant to the JEDEC specification. For the latter, it is important to automatically translate the JEDEC standards into formal models, which can validate the behavior of the simulation output and therefore help the developers to find issues quickly when, e.g., adopting the implementation to a new standard or implementing a new feature such as a new memory scheduling algorithm.

Unfortunately, there exist no unique formal descriptions of the JEDEC standards, which could be used for test case generation and therefore for the validation of a memory controller. Moreover, the JEDEC standards even provide inconsistencies in the description of the DRAM functionality, as pointed out in [18]. Figure 2 shows the *Finite State Machine* (FSM) provided by JEDEC's DDR3 standard [30] that is intended to provide an overview of the possible state transitions and commands. JEDEC even admits that this FSM is not fully correct [30], as it does not capture DRAM's inherent bank parallelism and therefore cannot model all possible events. In addition, the JEDEC state diagram lacks readability by mixing up DRAM states and DRAM commands [18]. This simplification is used because modeling the concurrent DRAM devices with an FSM will result in a state explosion[1].

Because of this state explosion, the authors of [18] used Petri Nets [34] in order to provide a comprehensive formal model of DRAM states and commands from a memory controller's perspective. In contrast to FSMs, Petri Nets are a model of computation, which was developed for describing highly concurrent and asynchronous systems. The model of [18] is intended to give an easy and interactive description of all possible DRAM states, including situations that involve more than one bank[2]. However, the authors only concentrate on the states and logical command dependencies, while the much more complex timing dependencies were not considered.

Based on the previous work, in this paper we use *Timed Petri Nets* [16] to provide a full model of the DRAM states, the logical command dependencies and the internal timing dependencies for the first time. The proposed model fulfills the requirements on a system model to be unique, precise, complete, and easy to modify. It can therefore be used as a formal reference specification for the simulation-based validation, e.g., of a DRAM controller [13] or a DRAM simulator [6, 12, 17, 21, 22, 26, 35]. Furthermore, we introduce a *Domain Specific Language* (DSL) in order to describe the semantics of this DRAM-specific Petri Net with a comprehensive and compact syntax. From the description in this DSL an executable SystemC model of the corresponding Petri Net can be automatically generated (extensions to other languages are certainly possible). A new JEDEC standard can therefore be adopted quickly with only minor changes in the DSL code. Furthermore, the generated executable specification with Petri Net semantic is correct by construction. In summary, our paper makes the following contributions:

- For the first time, we describe how the complex timing protocol of DRAMs can be modeled by timed Petri Nets. To the best of our knowledge, there exists no similar approach in the literature.
- We introduce a DSL called DRAMml for describing the commands, states and timings of DRAMs in a compact and extendable way.
- From this DSL we automatically generate an executable model which can be used for the simulation-based validation of memory controllers and simulation models.

The remainder of this paper is structured as follows: In Section 2, related and previous works combining DRAMs and Petri Nets are discussed. In addition, the state of the art for memory controller validation is reviewed. Section 3 introduces the basics of DRAMs and Petri Nets. The modeling of DRAMs with Petri Nets is presented in Section 4. First, the modeling of the DRAM states, commands, and their inter-dependencies [18] is shown. Next, Section 4.2 presents the modeling of the DRAM power consumption, while Section 4.3 extends the model by timing dependencies. In Section 5, the DSL DRAMml is introduced that is used to generate an executable model as described in Section 6. Section 7 showcases a possible application of this model. Finally, Section 8 concludes the paper and motivates future work.

[1]In order to model the complete state space of a DRAM, the number of states for this FSM is equal to $2^B + 4$ and the number of edges is $2 \cdot \binom{2B}{B-1} + 2^B + 2B + 5$, where B denotes the number of banks. For example, a DDR4 DRAM with $B = 16$ would feature 65,536 states connected by 1,131,478,243 edges.

[2]An executable model is uploaded on Github: https://github.com/tukl-msd/DRAMPetri

2 RELATED WORK

In this section we first discuss the related work with respect to modeling of DRAMs using Petri Nets and second show more general approaches for the verification of memory controllers.

2.1 Modeling of DRAMs using Petri Nets and Timed Automata

A similar approach for modeling DRAMs with Petri Nets has been presented by Gries [10]. However, this work captures all aspects in a detailed bottom-up approach from the DRAM cell, over the array, up to the memory controller and therefore features a high level of complexity that makes it infeasible to use this model for validation of modern DRAM systems. The authors of [8] use Petri Nets to model the DRAM power-down states in order to derive effective power-down strategies. However, their model focuses only on the power-down states. In comparison to these works, the authors of [18] give a formally correct, lean, and easily understandable description of the DRAM states and transitions, which can replace the JEDEC state diagrams. However, they do not model the complex timing dependencies. Li et al. [27] use timed automata models of a memory controller to derive the *Worst Case Response Time* (WCRT) and the *Worst Case Bandwidth* (WCBW) mathematically. The used timed automata models are complex, written by hand, and do not clearly reflect the actual DRAM states and command transitions.

2.2 Validation of Memory Controllers

The authors of [13, 14] focus on the automatic test case generation and the validation of memory controller simulation models. For the automatic test case generation they use *Linear Temporal Logic* (LTL) to describe properties which have to be fulfilled by the memory controller. These properties are negated and passed to a model checker, which works on an abstract state machine describing the DRAM behavior. Since the properties were negated, the model checker can find counterexamples (under the assumption that the provided state machine is correct!). From these counterexamples tests can be generated in order to perform a simulation-based validation of a memory controller or its simulation model. Furthermore, they show a methodology that uses high-level statistics such as bandwidth and latency for the verification of the memory controller's features. However, there can still be a situation where a bug exists, which does not manifest in a change of bandwidth and latency, and therefore cannot be detected. Furthermore, since the user has to provide the state machine as well as the properties, a full coverage of all possible cases cannot be guaranteed.

In [24], Kayed et al. show an approach where *System Verilog Assertions* (SVA) are generated from a *Timing Diagram Markup Language* (TDML). They redraw the timing diagrams which are shown in the JEDEC standards with a TDML tool and generate corresponding SVAs. However, this approach is not complete, as only those temporal dependencies are checked that have been drawn, and not all possible dependencies and interactions are covered.

The authors of [25] show a verification approach of a memory controller using the *Universal Verification Methodology* (UVM) with focus only on the DRAM power-down states. In [37] the authors present a formal state machine model and they derive an actual DRAM simulator from it. Unfortunately, both previous publications are short papers and lack details about the actual methodologies.

Table 1: DRAM Commands

Target	Symbol	Explanation
Row	ACT	*Activate*: A specific row in one bank is activated.
	PRE	*Precharge*: The currently activated row is closed and the bank is precharged.
Column	RD	*Read*: Read from an activated row.
	RDA	*Read with Auto-Precharge*: Read from a row and precharge the row afterwards.
	WR	*Write*: Write to an activated row.
	WRA	*Write with Auto-Precharge*: Write to a row and precharge the row afterwards.
Entire DRAM	PDE	*Power-Down Entry*: Enters the *PDNA* mode if in *Active* or *PDNP* if in *IDLE*.
	PDX	*Power-Down Exit*: Exits *PDNP* or *PDNA* mode.
	PREA	*Precharge All*: All active banks are precharged.
	REFA	*Auto-Refresh*: Refresh one or more rows in all banks.
	SREFEN	*Self-Refresh Entry*: Enters the *SREF* mode.
	SREFEX	*Self-Refresh Exit*: Exits the *SREF* mode.

The state-of-the-practice procedure to verify memory controller designs in industry is to perform an RTL simulation of the controller integrated with an appropriate DRAM model that ensures strict obedience of the DRAM protocol by the memory controller under test (i.e command sequence, timings, refresh, etc.). These DRAM models are provided by the DRAM manufacturers and are usually encrypted in order to protect the vendors' *Intellectual Property* (IP).

However, this method of validating the controller requires the DRAM model employed to be assumed functionally correct. In case of a bug in an encrypted DRAM model, it is impossible for a test engineer to find this bug and it might be propagated to the memory controller design. In contrast, our approach is fully transparent and does not reveal any IP since it contains only the public information provided by the JEDEC protocol. Therefore, the memory vendor and controller developer can easily agree on a common test scenario.

3 BACKGROUND

In this section we first introduce the basic terminology of DRAM controllers and internals of DRAM devices. Second, we define the original Petri Net and extensions with respect to Turing completeness and modeling of timing.

3.1 DRAM Devices & Controllers

A DRAM device is organized in a three-dimensional fashion of banks, rows and columns. It usually has eight (DDR3) or 16 (DDR4) banks, which can be used concurrently (*bank parallelism*). However, there are some constraints due to the shared data command/address bus. Each bank consist of, e.g., 2^{12} to 2^{18} rows, whereas each row can store, e.g., 512 B to 2 KB of data. A memory controller is usually composed of a *Frontend* and a *Backend*. The frontend performs arbitration and scheduling of incoming read and write requests, whereas the task of the backend is to translate these incoming requests to a sequence of DRAM commands, which have to be orchestrated with respect to the current state of the device and given timing dependencies. To access data in a row of a certain bank, the *activate* command (ACT) must be issued by the controller before any column access, i.e., *read* (RD) or *write* commands (WR), can be executed. The ACT command opens an entire row of the memory array, which is transferred into the bank's *row buffer*[3]. It acts like a small cache that stores the most recently accessed row of the bank. The latency of a memory access to a bank largely varies depending on the state of this row buffer. If a memory access targets the same row as the currently cached row in the buffer (called *row hit*), it results in a low latency and low energy memory access. Whereas, if a memory access targets a different row as the current row in the buffer (called *row miss*), it results in higher latency and energy consumption. If a certain row in a bank is active it must be precharged (PRE) before another row can be activated. In addition to the normal RD and WR commands, there exist read and write commands with an integrated auto-precharge (RDA, WRA). If auto-precharge is selected, the row being accessed will be precharged at the end of the read or write access.

A DRAM cell usually has to be refreshed every 64 ms to retain the data stored in it. Modern DRAMs are equipped with an *Auto-Refresh* (REFA) command to perform this operation. Besides the normal active mode operations presented above, a DRAM is capable to enter power-down modes to save energy by setting the clock-enable signal cke to low. There exist three major power-down modes called *Precharge Power-Down* (PDNP), *Active Power-Down* (PDNA) and *Self-Refresh* (SREF). Table 1 shows a list of all possible DRAM commands. During operation a DRAM device can be in five major states, as shown in Table 2. These states are used to calculate the background power of the DRAM by tools like, e.g., DRAMPower [5].

As shown in [18], the JEDEC FSM in Figure 2 has several drawbacks: First, it uses auxiliary states like *Activating* and *Precharging* that do not account for modeling the DRAM operations. Second, the state diagram uses doubled states (2× *Reading* and *Writing*) that are confusing for the reader and lead to logic inconsistencies when combined with an automatic sequence. For instance, if a RD command is scheduled, the automatic sequence will return the DRAM state to *Bank Active*, thus, all other transitions from the reading state become obsolete. Third, Figure 2 does not cover DRAM's inherent *bank parallelism*, which is crucial for an exact behavioral description.

So far, we only discussed the commands and states of the DRAM. However, the most complex part of DRAMs are timing dependencies between different commands. For example, in the DDR4 JEDEC standard more than 90 out of 260 pages are showing timing diagrams and explanations for the timings. Table 3 shows the timing

[3]The row buffer is a model, which abstracts the real physical DRAM architecture. It is basically a combination of *Primary Sense Amplifiers* (PSA) and *Secondary Sense Amplifiers* (SSA) of the memory arrays in one bank. This model is useful, e.g., for describing the functionality of a memory controller and its scheduling algorithms. Unfortunately, this model often leads to a misunderstanding of the real DRAM architecture. For further details on internal DRAM architecture we refer to [15].

Table 2: DRAM States

Type	Symbol	Explanation
Normal Operation	*Active*	At minimum one bank is active, no power-down (cke=1), no internal refresh (the DRAM controller has to schedule refresh commands).
	IDLE	All banks are closed and precharged, no power-down (cke=1), no internal refresh. The DRAM changes the state from *Active* to *IDLE* by issuing a precharge command (PRE).
Power-Down	*PDNP*	*Precharge Power-Down*: All banks are closed and precharged (in *IDLE* state, cke=0) and no internal refresh.
	PDNA	*Active Power-Down*: At minimum one bank is active (in *Active* state, cke=0) and no internal refresh.
	SREF	*Self-Refresh*: All banks are precharged and closed, the DRAM internal self-timed refresh is triggered (cke=0).

parameters for a DDR3 memory and Figure 3 shows a timing diagram with the relations to the commands for an artificial DRAM example[4]. In general, the different timings can be distinguished into four main categories:

- **Command-to-Command Timing Dependencies:** The execution of one command implies that other commands have to wait for a specific time interval. For example, if there is a row hit, a read command must wait at least t_{CCD} in order to avoid an overlap on the data bus, as shown in Figure 3. These command-to-command timing dependencies are the majority of timing dependencies in the JEDEC standards.
- **Command Bus Occupancy Dependency:** Only one command can be scheduled to the command bus during one cycle. For example, if there are two commands ready, the memory controller will resolve this issue by scheduling one command one cycle cycle later.
- ***N*-Activate Window:** In order to limit peak currents there exists a rolling time-frame in which a maximum of N banks can be activated, called t_{NAW}. For example in DDR3, where $N = 4$, it is called *Four Activate Window* (t_{FAW}), whereas in Wide I/O DRAMs, $N = 2$, called *Two Activate Window* (t_{TAW}).
- **Refresh Mechanism:** Each DRAM cell must be refreshed every $t_{REF} = 64$ ms. Therefore, a refresh command must be scheduled every $t_{REFI} = 7.8\,\mu s$ to avoid data corruption. However, the standards allow postponing the refresh command 8 times. In case that 8 refresh commands are postponed in a row, the resulting maximum interval between the surrounding refresh commands is limited to $t_{REFMAX} = 9 \cdot t_{REFI}$.

[4] Artificial Example: *Double Data Rate* (DDR) *Dynamic Random Access Memory* (DRAM) timings: $t_{RCD} = 3 \cdot t_{CK}$, $t_{CL} = 3 \cdot t_{CK}$, $t_{RP} = 2 \cdot t_{CK}$ $t_{RTP} = 2 \cdot t_{CK}$, $t_{RAS} = 7 \cdot t_{CK}$, and $t_{CCD} = 2 \cdot t_{CK}$ with *Burst Length* (BL) = 4

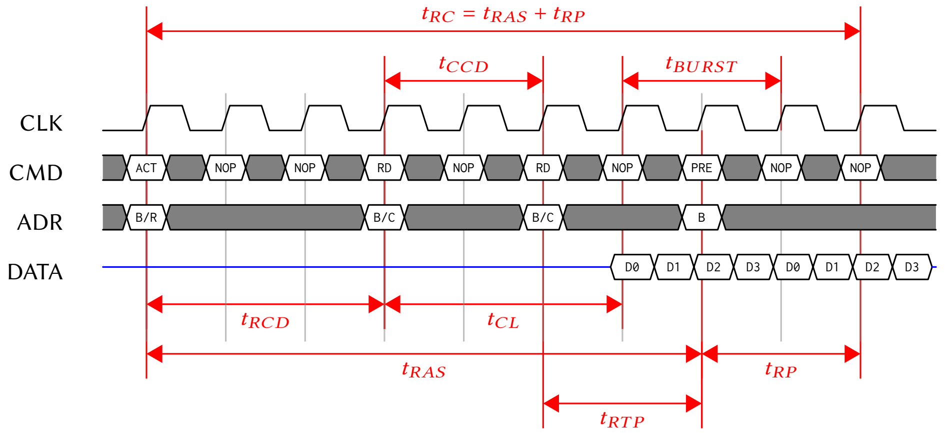

Figure 3: Basic DRAM protocol

Section 4.3 will explain the DRAM timings in further detail. In the next section the basics of Petri Nets are described.

3.2 Petri Nets

Petri Nets [34] are very general models for concurrent asynchronous systems. Thus they are widely used to describe system behavior on different levels [3, 7, 9, 40]. They consist of places holding tokens and transitions, which are connected to each other. Usually, places represent conditions or states and transitions represent events. In the following we define a *Petri Net* according to [32]:

Definition 3.1 (Petri Net). A Petri Net $N = (P, T, F, W, M_0)$ is a 5-tuple where $P = \{p_1, p_2, ..., p_m\}$ is a finite set of places, $T = \{\tau_1, \tau_2, ..., \tau_n\}$ is a finite set of transitions, $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs ($\longrightarrow$) connecting places and transitions, $W : F \longrightarrow \mathbb{N}$ is a weight function[5] (if not indicated the weight is set to 1) and $M_0 : P \longrightarrow \mathbb{N} \cup \{0\}$ is the initial marking. It is required that $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.

The simulation with Petri Nets is based on the following transition or firing rules [32]:

(1) A transition is said to be enabled if each input place p of τ is marked with at least $w(p, \tau)$ tokens, where $w(p, \tau)$ is the weight of the arc from p to τ.
(2) An enabled transition may or may not fire (depending on whether or not the event actually takes place).
(3) A firing of an enabled transition τ removes $w(p, \tau)$ tokens from each input place p of τ and adds $w(\tau, p')$ tokens to each output place p' of τ, where $w(\tau, p')$ is the weight of the arc from τ to p'.

Figure 4 shows an example of a simple Petri Net [32], using the well-known chemical reaction: $2H_2 + O_2 = 2H_2O$. Two tokens in each input place represent two available units of H_2 and O_2. The transition τ_1 is enabled by a chemical reaction (event). After firing τ_1, the marking will change according to the weights, and τ_1 is no longer enabled.

Several extensions to the original Petri Net in Definition 3.1 have been proposed in recent years. In order to model the behavior of

[5] $\mathbb{N}$ denotes the set of natural numbers without 0.

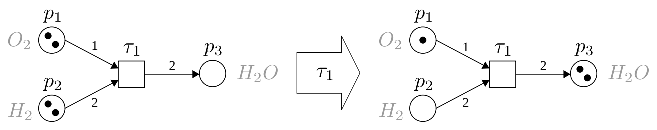

Figure 4: Petri Net Example According to [32]

DRAMs with Petri Nets, two extensions called *Inhibitor-* [1, 11] and *Reset-Arcs* [2] are required. A *Reset-Arc* is a type of arc that connects a place to a transition and its semantics is to remove all tokens from that place when the transition fires [39]. An *Inhibitor-Arc* is a type of arc that connects a place to a transition and its semantics is to prevent the transition from firing when the place contains more tokens than the arc weight indicates [39]. It is shown in [33] that Petri Nets with inhibitor-arcs have the modeling power of Turing machines.

Definition 3.2 (Reset Net). A *Reset Net* is a tuple, $N^R = (N, R)$, where N is a Petri Net and $R \subseteq (P \times T)$ denotes the set of reset-arcs ($\twoheadrightarrow$).

A reset-arc for a specific transaction τ empties all places p_i connected with reset-arcs when the transition fires. There is no precondition on firing imposed. As shown in the example in Figure 5 the place p_3 is cleared completely when τ_1 is fired.

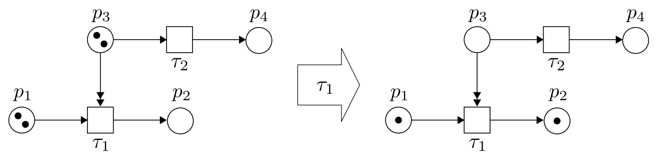

Figure 5: Reset Net Example [18]

Definition 3.3 (Inhibitor Net). An *Inhibitor Net* is a triple, $N^I = (N, I, W_I)$, where N is a Petri Net, $I \subseteq (P \times T)$ specifies the set of inhibitor-arcs ($\multimap$), and $W_I : I \longrightarrow \mathbb{N}$ is a weight function.

A transition τ connected with a place p by an inhibitor-arc of weight $w_I(\tau, p)$ is disabled as long as p holds at least $w_I(\tau, p)$ tokens.

Table 3: Timing Parameters for DDR3 [15]

Name	Explanation
t_{CL}, t_{RL}, t_{WL}	*CAS Latency*: The time needed to transfer the data from the PSA to the SSA and the interface, in other words: The delay from a RD command (t_{RL}) or a WR command (t_{WL}) until data can be read/written at the interface.
t_{CK}	*DRAM Clock*: The clock of the DRAM interface.
t_{CCD}	*Column-to-Column Delay*: The minimum column command timing, determined by internal burst (prefetch) length.
t_{FAW}	*Four Activate Window*: Only four ACT commands can be issued in this time window.
t_{RAS}	*Row Access Strobe*: The minimum active time for a row, in other words: The time interval between row access command and data restoration in a DRAM array.
t_{REFI}	*Refresh Interval*: The time interval between two refresh commands (e.g., $t_{REFI} = 7.8\mu s$).
t_{RCD}	*Row-to-Column Delay*: The time interval between row access and data ready at PSAs, in other words: The time interval between ACT and RD on the same bank.
t_{RFC}	*Refresh Cycle*: The duration of a refresh, in other words: The time interval between REF and ACT commands.
t_{RP}	*Row Precharge*: The time interval that it takes for a DRAM array to be precharged (PRE) and prepared for another row access.
t_{RRD}	*Row-to-Row Delay*: The minimum time interval between two ACT commands to different banks.
t_{RTP}	*Read to Precharge*: The time interval between RD and a PRE command.
t_{WR}	*Write Recovery*: The minimum time interval between the end of a WR burst and a PRE command.
t_{WTR}	*Write to Read*: The minimum time interval between the end of a WR burst and a RD command.
t_{XP}	*Power-Down Exit Delay*: The minimum time interval between PDX to any valid command.
t_{XS}, t_{XSDLL}	*Self-Refresh Exit Delay*: The minimum time interval (t_{XS}) between SREFEX to any valid command. For the commands like RD, RDA, etc. that require *Delay-Locked Loop* (DLL) to be enabled, the minimum time interval is t_{XSDLL}.
$t_{ACTPDEN}$	*ACT to Power-Down Delay*: The minimum timing of ACT command to Power-Down entry.
t_{PRPDEN}	*Precharge to Power-Down Delay*: The minimum timing of PRE command to Power-Down entry.
t_{RDPDEN}	*Read to Power-Down Delay*: The minimum timing of RD/RDA command to Power-Down entry.
t_{WRPDEN}	*Write to Power-Down Delay*: The minimum timing of WR command to Power-Down entry.
$t_{WRAPDEN}$	*Write with auto precharge to Power-Down Delay*: The minimum timing of WRA command to Power-Down entry.
$t_{REFPDEN}$	*Refresh to Power-Down Delay*: The minimum timing of REF command to Power-Down entry.
t_{PD}	*Power-Down Delay*: The minimum timing of Power-Down Entry to Exit.
t_{CKE}	*Clock Enable Delay*: Minimum interval between two consecutive CKE transitions.
t_{CKESR}	*Self-Refresh Delay*: Minimum CKE low width for Self-Refresh entry to exit timing.

Vice versa, it is enabled whenever p holds strictly less than $w_I(\tau, p)$ tokens[6]. As shown in the example in Figure 6 the transition τ_1 is inhibited by p_3. When τ_1 fires before τ_2 the tokens are moved to p_2 and p_4, respectively. However, if τ_2 fires first, p_4 inhibits the firing of τ_1. In this case p_2 will never get the token, because p_4 can never be cleared.

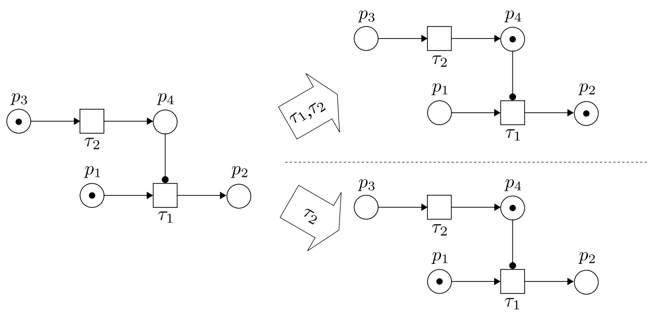

Figure 6: Inhibitor Net Example [18]

Up to now, the DRAM model is fully functional but does not respect the DRAM timings defined in the JEDEC standard. Therefore, the Petri Net has to be further extended to be able to model the correct timing constraints. The following definition is based on [16].

Definition 3.4 (Timed-Arc Net). A Timed-Arc Petri Net is a Petri Net where each token of the marking M gets an age $x_p \in \mathbb{R}_{\geq 0}$ assigned. The set of arcs is extended by age guards, i. e. $F \subseteq (P \times \mathcal{J} \times T) \cup (T \times P)$, where $\mathcal{J}$ is the set of all valid time intervals. Timed-arcs are denoted by $\xrightarrow{[t_1,\, t_2]}$ for $[t_1, t_2] \in \mathcal{J}$.

The firing rules for Timed-Arc Petri Nets are modified as follows:

(1) A transition is said to be enabled if each input place p of τ is marked with at least $w(p, \tau)$ tokens of age $x_p \in J$, where $w(p, \tau)$ is the weight of the arc from p to τ and $J \in \mathcal{J}$ is the age guard of the arc.
(2) An enabled transition may or may not fire.
(3) A firing of an enabled transition τ removes $w(p, \tau)$ tokens of age $x_p \in J$ from each input place p of τ and adds $w(\tau, p')$ tokens of age 0 to each output place p' of τ.

In the case of Timed-Inhibitor-Nets, a timed inhibitor-arc from p to τ inhibits the transition τ from firing as long as p holds at least $w_I(\tau, p)$ tokens of age $x_p \in J$, where J is the age guard of the timed inhibitor-arc. Timed inhibitor-arcs are depicted as $\overset{[t_1,\, t_2]}{-\!\!-\!\!-\!\!\bullet}$ for $[t_1, t_2] \in \mathcal{J}$.

[6]In the case where $w_I(p, \tau) = 1$, the transition τ may only fire when the connected place p is empty.

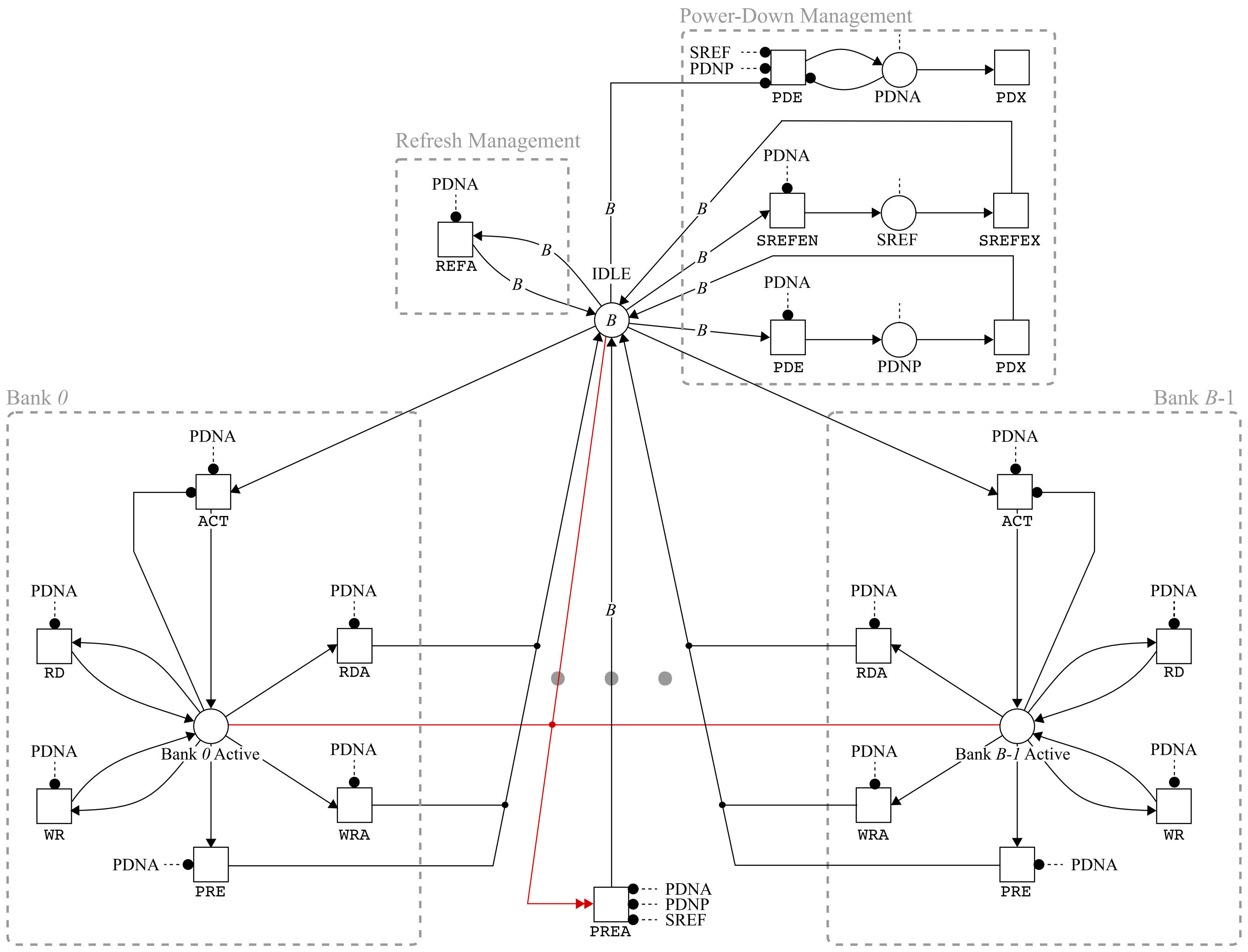

Figure 7: DRAM Petri Net Model from [18]

4 MODELING DRAMS WITH PETRI NETS

As shown in [18], the states and command transitions for DRAMs can be modeled by an *Inhibitor-Reset Petri Net*. In this paper, we extend the previous approach of [18] in order to model the timing dependencies between the DRAM commands, which can be modeled by *Timed-Arc Nets*. We introduce a new arc, which can be composed of a timed inhibitor-arc and a reset-arc, as shown in the following sections. For the sake of simplicity in the description we mostly concentrate on the DDR3 standard in a single-rank configuration, noting that all the other standards can also be modeled with this approach.

4.1 Modeling States and Commands

With the Definitions 3.1, 3.2 and 3.3, the states and command dependencies for DRAMs can be modeled by an *Inhibitor-Reset Petri Net* that eliminates the aforementioned drawbacks of the JEDEC state diagram by introducing the required bank parallelism without increasing the diagram's complexity and readability, as shown in [18]. Furthermore, we strictly distinguish between DRAM states (*IDLE*, *Active*, *PDNP*, *PDNA*, and *SREF*) and DRAM commands (ACT, PRE, RD, RDA, WR, WRA, PDE, PDX, PREA, REFA, SREFEN, and SREFEX). As shown in Figure 7, the transitions of the Petri Net represent the executed DRAM commands and the places denote the states of the DRAM, i.e., how many banks are active or if the DRAM is in power-down mode. It is assumed that the DRAM has *B* banks.
The DRAM Petri Net model is divided into several subnets:

(1) *B* Bank Subnets
(2) Refresh Management
(3) Power-down Management

In the beginning, the place *IDLE* is initialized with *B* tokens, whereas all the other places are cleared. If a row in bank *b* gets activated, the related ACT transition is fired and a token moves from the *IDLE* state to the *Bank-b-Active* state. The inhibitor-arc from the *Bank-b-Active* state to the ACT transition ensures that no other token can move into the *Bank-b* subnet. In the meantime, other banks can be activated. When a PREA command is issued, all places in the

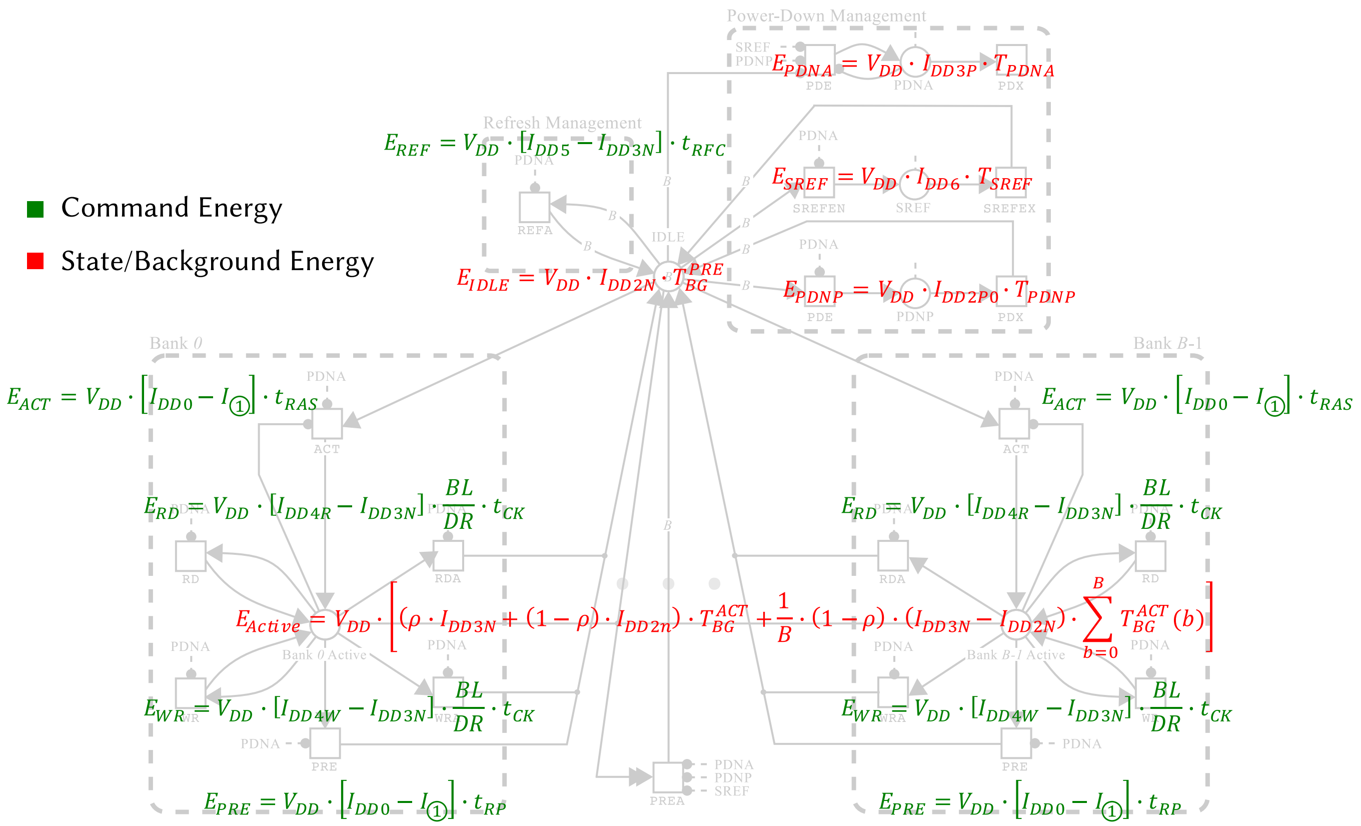
🟩 Command Energy
🟥 State/Background Energy
Power-Down Management
Refresh Management
Bank 0
Bank B-1
$E_{PDNA} = V_{DD} \cdot I_{DD3P} \cdot T_{PDNA}$
$E_{SREF} = V_{DD} \cdot I_{DD6} \cdot T_{SREF}$
$E_{PDNP} = V_{DD} \cdot I_{DD2P0} \cdot T_{PDNP}$
$E_{REF} = V_{DD} \cdot [I_{DD5} - I_{DD3N}] \cdot t_{RFC}$
$E_{IDLE} = V_{DD} \cdot I_{DD2N} \cdot T_{BG}^{PRE}$
$E_{ACT} = V_{DD} \cdot [I_{DD0} - I_{①}] \cdot t_{RAS}$
$E_{RD} = V_{DD} \cdot [I_{DD4R} - I_{DD3N}] \cdot \frac{BL}{DR} \cdot t_{CK}$
$E_{Active} = V_{DD} \cdot \left[(\rho \cdot I_{DD3N} + (1-\rho) \cdot I_{DD2n}) \cdot T_{BG}^{ACT} + \frac{1}{B} \cdot (1-\rho) \cdot (I_{DD3N} - I_{DD2N}) \cdot \sum_{b=0}^{B} T_{BG}^{ACT}(b)\right]$
$E_{WR} = V_{DD} \cdot [I_{DD4W} - I_{DD3N}] \cdot \frac{BL}{DR} \cdot t_{CK}$
$E_{PRE} = V_{DD} \cdot [I_{DD0} - I_{①}] \cdot t_{RP}$
PDE
PDNA
PDX
SREFEN
SREF
SREFEX
PDNP
REFA
IDLE
ACT
RD
RDA
WR
WRA
PRE
PREA
Bank 0 Active
Bank B-1 Active

Figure 8: Annotated Power Equations from DRAMPower [5, 20, 28]

bank subnets are cleared and the *IDLE* state is reinitialized with B tokens by the connected reset-arc. When the *IDLE* state hosts B tokens, a refresh (REFA) can be executed. Additionally, there is the possibility to enter precharge power-down or self-refresh. Several inhibitor-arcs exists in order to prevent prohibited state transitions. However, this modeling approach presented by [18] does not take the timing dependencies into account. Therefore, we present the modeling of timing in Section 4.3.

4.2 Modeling Power Consumption

As a side effect, we discovered that the equations of the DRAMPower simulation tool [5, 20, 28] can be projected onto the presented Petri Net model of [18] as shown in Figure 8. The power consumption of DRAMs can be divided into a background or static part and an active part which is needed for each command (ACT, PRE, RD, etc.). The background part is directly calculated from the current state of the DRAM (*IDLE*, *Active*, *PDNP*, etc.). Therefore, these equations are annotated to the states of the Petri Net. Accordingly, the active part is annotated to the transitions.

4.3 Modeling Timing Dependencies

With Definition 3.4, the timing dependencies for the DRAM protocol can be modeled by a *Timed-Inhibitor-Reset Petri Net*. As shown in Section 3.1 the different timings that have to be modeled can be distinguished into four categories. For each category we present a different solution based on timed-arcs. These approaches allow a clear separation from the state model in Section 4.1 such that the timing dependency network can be treated separately.

4.3.1 *Command-to-Command Timing Dependencies.*

In order to model command-to-command timing dependencies, for example t_{RRD} between two ACT commands on two different banks, we introduce a new custom arrow $\xrightarrow{t_x}$, which is composed of a place, a reset-arc, and a timed inhibitor-arc, as shown in Figure 9.

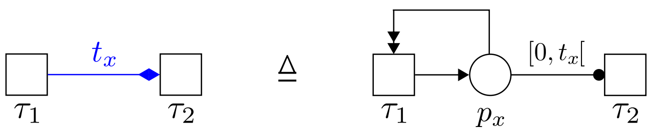

Figure 9: Custom Arc

When transition τ_1 is fired, all existing tokens on place p_x are cleared by the reset-arc and a new token of age 0 is generated. The timed inhibitor-arc attached to τ_2 inhibits its firing as long as the age of the token is smaller than the timing value t_x (i.e., as long as its age lies in the interval $[0, t_x[$). By the use of this arc all command-to-command timing dependencies can be modeled easily as shown in Figure 13 for an artificial DRAM with 2 banks.

For instance, between the ACT command of Bank 0 and the ACT command of Bank 1, there exists a special arc with t_{RRD} as timing constraint and vice versa.

Note that this custom timing arc does not alter the modeling power of the Petri Net, as it is just a short form for the specific combination of a place, a normal arc and a reset- and timed inhibitor-arc (see Figure 9).

4.3.2 Command Bus Occupancy.

The command bus occupancy could also be modeled with the special command-to-command arc by connecting each command transition to each other command transition with a t_{CK} timing dependency. However, in this case, the number of special timing arcs would grow exponentially and slow down the execution of the model. Therefore, we use a special place which represents the command bus. This place is connected to all command transitions in the Petri Net as shown in Figure 10.

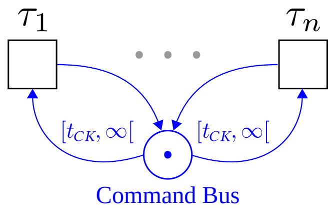

Figure 10: Modeling of the Command Bus Constraints

If the τ_i transition is fired (e.g., ACT command) it consumes the token on the command bus and generates a new token of age 0. The other transition τ_n (e.g., RD to a different bank) can only be fired if the token has at least age t_{CK}. With this constraint only one command can be sent to the DRAM at a time within one clock cycle. This small net also shows the flexibility of our approach. For example, in the case of LPDDR4, a variable command length was introduced (i.e., commands are longer than just one clock cycle). This new requirement can be modeled easily by changing the timing parameter from t_{CK} to the new timing parameter.

4.3.3 N-Activate Window.

As shown in Section 3.1 there exists the timing dependency t_{NAW}, the so-called *N-Activate Window*. In the case of DDR3 where $N = 4$, it is also called *Four Activate Window* or in short t_{FAW}. In this window only four ACT commands can be issued due to power constraints. Similar to the command bus dependency we use a special place called *NAW Pool* which holds exactly N tokens. Figure 11 shows an abstract DRAM model omitting non-relevant states in order to explain the functionality.

Whenever an ACT transition is fired, a token of age at least t_{NAW} is consumed and a new token of age 0 is generated. Therefore, the NAW Pool always holds N tokens of different ages. The tokens in the NAW Pool can be considered as "disabled" after their consumption for t_{NAW} time. Since there are N tokens in the pool, it is ensured that only N ACT transitions can happen in a t_{NAW} time window.

4.3.4 Refresh Mechanism.

To avoid data losses, refresh commands have to be scheduled regularly, i.e., refreshes have to be scheduled every $t_{REFI} = 7.8\mu s$. Since the JEDEC standard allows postponing refresh commands

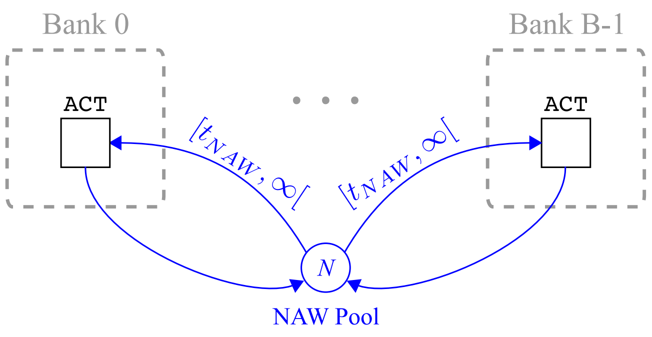

Figure 11: Modeling of t_{NAW} with B Banks and N-Activate Window

up to eight times, after at most $t_{REFMAX} = 9 \cdot t_{REFI}$ a refresh has to be performed. Figure 12 depicts the realization of this refresh mechanism in our Petri Net.

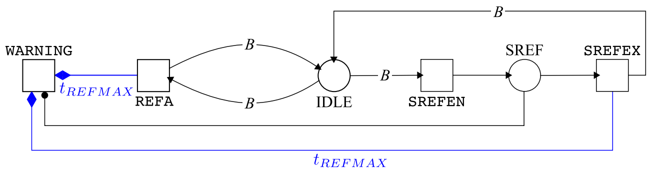

Figure 12: Modeling of Refresh Mechanism

If no REFA transition is fired within t_{REFMAX} time, the WARNING transition becomes enabled, which should trigger a warning output to the user. To avoid the warning during self-refresh, the *SREF* state additionally inhibits the WARNING and the SREFEX resets the t_{REFMAX} timer. Instead of using the WARNING transition, we could also use a specific *Halt* place in a similar way that would inhibit **all** transitions from firing whenever it holds a token of age larger than t_{REFMAX} to prevent the Petri Net from proceeding any further. However, especially in the context of new approaches like *Approximate DRAM* [19, 23], the refresh period can be enlarged or refresh can be completely turned off on purpose. Therefore, we decided to throw a warning instead of forcing a complete halt.

5 DRAMml: A DOMAIN SPECIFIC DRAM MODELING LANGUAGE

For the previously described Timed Petri Net semantic, we developed a DSL called *DRAMml*, which describes all the timing, state and command information of the JEDEC standards in a formal, short, comprehensive, readable and understandable format. From this DSL executable code can be generated (e.g., for SystemC, VHDL, Verilog, or timing verification languages) in order to interact directly with controller RTL and controller simulation models for a fast simulation-based validation approach. A key feature compared to the state of the art is that the generated formal executable model is correct by construction. For the design of the DSL we used the MPS Software from Jetbrains[7].

[7]https://www.jetbrains.com/mps/

Listing 1: DRAMml Describing DDR3

```
DRAM {
  Name = "DDR3-800D";
  B = 8;
  N = 4;

  Device {
    Places {
      IDLE (B);
      PDNP;
      SREF;
      PDNA;
    }

    Transitions {
      REFA;
      PREA;
      PDEP;
      PDXP;
      SREFEN;
      SREFEX;
      PDEA;
      PDXA;
    }

    Arcs {
      // Normal Arcs:
      IDLE   -> REFA   (Weight = B);
      REFA   -> IDLE   (Weight = B);
      PREA   -> IDLE   (Weight = B);
      IDLE   -> PDEP   (Weight = B);
      PDEP   -> PDNP;
      PDNP   -> PDXP;
      PDXP   -> IDLE   (Weight = B);
      IDLE   -> SREFEN (Weight = B);
      SREFEN -> SREF;
      SREF   -> SREFEX;
      SREFEX -> IDLE   (Weight = B);
      PDEA   -> PDNA;
      PDNA   -> PDXA;

      IDLE -> ACT;
      RDA  -> IDLE;
      WRA  -> IDLE;
      PRE  -> IDLE;

      // Inhibitor Arcs:
      PDNA -o REFA;
      PDNA -o PREA;
      PDNP -o PREA;
      SREF -o PREA;
      PDNA -o PDEP;
      PDNA -o SREFEN;
      PDNA -o PDEA;
      PDNP -o PDEA;
      SREF -o PDEA;
      IDLE -o PDEA (Weight = B);

      PDNA -o ACT;
      PDNA -o RD;
      PDNA -o WR;
      PDNA -o PRE;
      PDNA -o RDA;
      PDNA -o WRA;

      // Reset Arcs:
      ACTIVE ->> PREA;
      IDLE   ->> PREA;
    }

    // Define 8 Banks:
    B : Bank {
      Places {
        ACTIVE;
      }

      Transitions {
        ACT;
        RD;
        RDA;
        PRE;
        WR;
        WRA;
      }

      Arcs {
        // Normal Arcs:
        ACT    -> ACTIVE;
        ACTIVE -> RD;
        RD     -> ACTIVE;
        ACTIVE -> RDA;
        ACTIVE -> WR;
        WR     -> ACTIVE;
        ACTIVE -> WRA;
        ACTIVE -> PRE;

        // Inhibitor Arcs:
        ACTIVE -o ACT;
      }
    }
  }

  // Define Timing Constraints:
  TimingConstraints {
    Timings {
      tCK     = 2.5;
      tCCD    = 10;
      tRCD    = 12.5;
      tRP     = 12.5;
      tRAS    = 37.5;
      tRL     = 5*tCK;
      tWL     = 5*tCK;
      tRTP    = 4*tCK;
      tWTR    = 4*tCK;
      tRRD    = 4*tCK;
      tWR     = 15;
      tFAW    = 40;
      tRFC    = 110;
      tREFI   = 7800;
      tREFMAX = 9*tREFI;

      tRC     = tRAS + tRP;
      tRDWR   = tRL + tCCD + 2*tCK - tWL;
      tWRRD   = tWL + tCCD + tWTR;
      tRDAACT = tRTP + tRP;
      tWRPRE  = tWL + tCCD + tWR;
      tWRAACT = tWRPRE + tRP;

      tXP     = 3*tCK;
      tXS     = tRFC + 10;
      tXSDLL  = 512*tCK;
      tCKE    = 3*tCK;
      tCKESR  = tCKE + tCK;
      tPD     = tCKE;
      tRDPDEN = tRL + 5*tCK;
      tWRPDEN = tWL + 4*tCK + tWR;
      tWRAPDEN= tWL + 5*tCK + tWR;
      tREFPDEN= tCK;
      tACTPDEN= tCK;
      tPRPDEN = tCK;
    }

    Places {
      CMD_BUS;
      NAW_POOL (N);
    }

    Arcs {
      // First timing for intra-bank
      // Second timing for inter-bank
      // Timing constraints from ACT
      ACT -<> PRE  (tRAS,0);
      ACT -<> RD   (tRCD,0);
      ACT -<> WR   (tRCD,0);
      ACT -<> RDA  (tRCD,0);
      ACT -<> WRA  (tRCD,0);
      ACT -<> ACT  (tRC,tRRD);
      ACT -<> PDEA (0,tACTPDEN);
      ACT -<> REFA (0,tRC);
      ACT -<> PREA (0,tRAS);

      // Timing constraints from RD/RDA
      RD  -<> PRE    (tRTP,0);
      RD  -<> PREA   (0,tRTP);
      RD  -<> PDEA   (0,tRDPDEN);
      RD  -<> PDEP   (0,tRDPDEN);
      RDA -<> PDEA   (0,tRDPDEN);
      RDA -<> PDEP   (0,tRDPDEN);
      RD  -<> RD     (tCCD,tCCD);
      RD  -<> RDA    (tCCD,tCCD);
      RDA -<> RD     (0,tCCD);
      RDA -<> RDA    (0,tCCD);
      RD  -<> WR     (tRDWR,tRDWR);
      RD  -<> WRA    (tRDWR,tRDWR);
      RDA -<> WR     (0,tRDWR);
      RDA -<> WRA    (0,tRDWR);
      RDA -<> ACT    (tRDAACT,0);
      RDA -<> REFA   (0,tRTP+tRP);
      RDA -<> PREA   (0,tRTP);
      RDA -<> SREFEN (0,tRDPDEN);

      // Timing constraints from WR/WRA
      WR  -<> PRE    (tWRPRE,0);
      WR  -<> PREA   (0,tWRPRE);
      WR  -<> PDEA   (0,tWRPDEN);
      WRA -<> PDEA   (0,tWRAPDEN);
      WRA -<> PDEP   (0,tWRAPDEN);
      WR  -<> WR     (tCCD,tCCD);
      WR  -<> WRA    (tCCD,tCCD);
      WRA -<> WR     (0,tCCD);
      WRA -<> WRA    (0,tCCD);
      WR  -<> RD     (tWRRD,tWRRD);
      WR  -<> RDA    (tWRRD,tWRRD);
      WRA -<> RD     (0,tWRRD);
      WRA -<> RDA    (0,tWRRD);
      WRA -<> ACT    (tWRAACT,0);
      WRA -<> REFA   (0,tWRPRE+tRP);
      WRA -<> PREA   (0,tWRPRE);
      WRA -<> SREFEN (0,tWRPRE+tRP);

      // Timing constraints from PRE/PREA
      PRE  -<> ACT    (tRP,0);
      PRE  -<> REFA   (0,tRP);
      PRE  -<> PDEA   (0,tPRPDEN);
      PRE  -<> PDEP   (0,tPRPDEN);
      PRE  -<> SREFEN (0,tRP);
      PREA -<> ACT    (0,tRP);
      PREA -<> REFA   (0,tRP);
      PREA -<> PDEP   (0,tPRPDEN);
      PREA -<> SREFEN (0,tRP);

      // Timing constraints from PDN
      PDEP -<> PDXP   (0,tPD);
      PDEA -<> PDXA   (0,tPD);
      PDXA -<> PDEA   (0,tCKE);
      PDXP -<> PDEP   (0,tCKE);
      PDXP -<> REFA   (0,tXP);
      PDXP -<> SREFEN (0,tXP);
      PDXA -<> ACT    (0,tXP);
      PDXP -<> ACT    (0,tXP);
      PDXA -<> PRE    (0,tXP);
      PDXA -<> PREA   (0,tXP);
      PDXA -<> RD     (0,tXP);
      PDXA -<> RDA    (0,tXP);
      PDXA -<> WR     (0,tXP);
      PDXA -<> WRA    (0,tXP);

      // Timing constraints from REFA/SREF
      REFA   -<> ACT    (0,tRFC);
      REFA   -<> REFA   (0,tRFC);
      REFA   -<> PREA   (0,tRFC);
      REFA   -<> SREFEN (0,tRFC);
      REFA   -<> PDEP   (0,tREFPDEN);
      SREFEX -<> ACT    (0,tXS);
      SREFEX -<> REFA   (0,tXS);
      SREFEX -<> PDEP   (0,tXS);
      SREFEX -<> SREFEN (0,tXS);
      SREFEX -<> RD     (0,tXSDLL);
      SREFEX -<> RDA    (0,tXSDLL);
      SREFEX -<> WR     (0,tXSDLL);
      SREFEX -<> WRA    (0,tXSDLL);
      SREFEN -<> SREFEX (0,tCKESR);

      // Arcs for the CLK
      CMD_BUS -> * [tCK, inf];
      * -> CMD_BUS;

      // Arcs for the NAW
      NAW_POOL -> ACT [tFAW, inf];
      ACT      -> NAW_POOL;

      // Arcs for the Refresh
      REFA   -<> WARNING (0,tREFMAX);
      SREF   -o  WARNING;
      SREFEX -<> WARNING (0,tREFMAX);
    }
  }
}
```

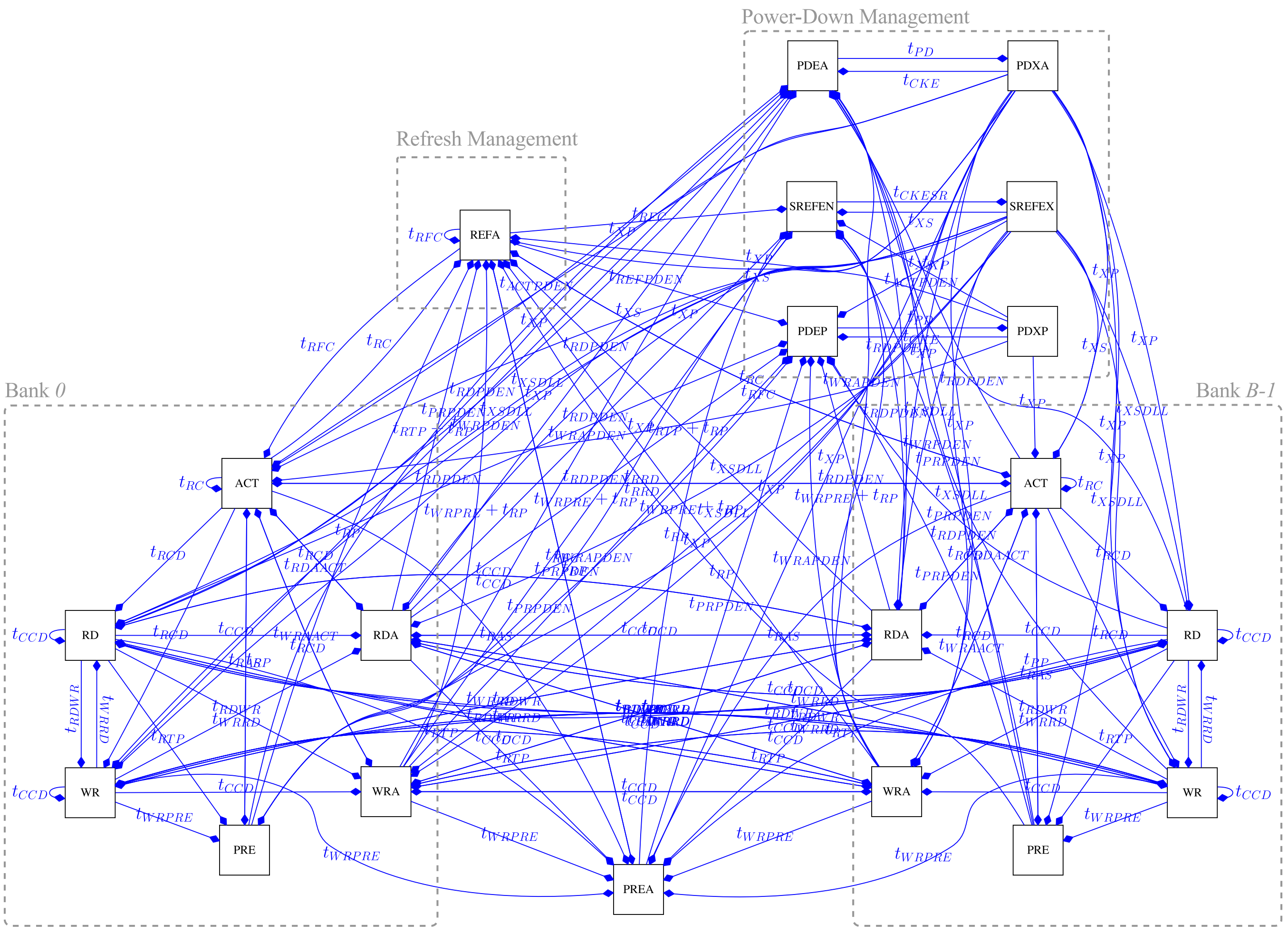

Figure 13: 2-Bank Petri Net with Command-to-Command JEDEC Timing Delays

An example for DRAMml describing a 1 Gb DDR3-800D DRAM with 1 KB page size and fast PDN exit mode is shown in Listing 1. The description starts with general definitions of the number of banks B and the number N needed for the NAW. Next, we describe the DRAM device hierarchically in a top-down approach. First, the DRAM device is described as consisting of places, transitions and arcs (everything outside the banks in Figure 7). In Line 71, B banks are inserted into the device that contain all the places, transitions and tokens which are inside one bank in Figure 7. Lastly, the timing constraints are inserted as described in Section 4.3. Lines 104–140 define the DRAM timings (in ns) for the given device. Next, places and transitions exclusively needed for the timing modeling of the whole device are added. Finally, the command-to-command timing arcs (Lines 151–241) and the timed-arcs for modeling the command bus occupancy (Lines 244–245), the NAW (Lines 248–249) and the refresh mechanism (Lines 252–254) are included.

The arcs used in DRAMml follow the symbols defined earlier:

- **`->`** denotes a normal arc.
- **`-> [t1, t2]`** denotes a timed-arc with age guard $[t_1, t_2]$.
- **`->>`** denotes a reset-arc.
- **`-o`** denotes an inhibitor-arc.
- **`-<> (t1, ... ,tn)`** denotes a command-to-command timing arc. The first timing holds for the lowest hierarchical level (i.e., inside the banks), while the last timing holds for the highest hierarchical level (i.e., outside the banks). If one of the timings is 0, no arc between the transitions exists on the corresponding hierarchical level.
- **`*`** denotes a wildcard for all commands, noting that $*$ in expressions has the semantic of an multiplication.

Note that an expression like `IDLE -> ACT` (Line 41 in Listing 1) denotes arcs from the `IDLE` place to **all** `ACT` transitions.

As an example for the used arcs, consider, e.g., the command-to-command timing arc `ACT -<> RD (tRCD,0)` from Line 152 in Listing 1. This line means that there are timed-arcs going from **all** `ACT` commands to **all** `RD` commands. The given timing `(tRCD,0)` means that the timing inside the banks is t_{RCD}, while outside the banks the timing is zero, i.e., there exists no timing constraint for inter-bank `ACT` and `RD` commands.

As stated before, this language allows an easy modification of timings (e.g., when changing timing parameters, or inserting new timings) as well as modifications of the whole DRAM structure. Consider the adjustment of the DSL given in Listing 1 to the concept of bankgroups in DDR4: A DDR4 DRAM has 16 banks in total, where each four of them are clustered in one bankgroup. Therefore, set `BG = 4` and `B = 4`. As the hierarchical level of bankgroups is between that of the whole device and that of single banks, we would adjust Listing 1 in Line 71 by the code shown in Listing 2.

Listing 2: Inserting Bankgroups

```
BG : Bankgroup {
  ...
  B : Bank {
    ...
  }
}
```

Since a third hierarchical level was included, the command-to-command timing arcs have to be changed from two to three parameters to include intra-bankgroup timings. Consider, e.g., the timing between two `RD` commands in DDR4, where short t_{CCD_S} timings capture inter-bankgroup timings, and longer t_{CCD_L} denote the timings for intra-bank and intra-bankgroup. Line 168 would then change to `RD -<> RD (tCCD_L,tCCD_L,tCCD_S)`. In a similar way, the model can be extended to include ranks. This shows that code written in DRAMml can easily be adopted to new standards without many changes.

6 CODE GENERATION

From the previously described DSL, an executable model can be generated, which can be used as a formal reference specification for simulation-based validation, e.g., of a DRAM controller or a DRAM simulator. Especially for testing a memory controller's backend this model has a large value since every timing violation will be reported.

As an example implementation we chose SystemC, noting that a conversion of the DSL in other languages is feasible as well. SystemC is an IEEE standard [29] for simulation and modeling of complex systems consisting of hardware and software. It is a library which is based on C++ and provides a simulation kernel with a discrete event semantic that enables the simulation of time.

In VHDL and Verilog, components use signals for the communication. In SystemC a signal is just a special case of a *Channel*. In SystemC channels are containers for communication protocols and synchronization events, i.e., they are separating the communication from the functionality. The different modules in SystemC can be connected to a channel via a port. Therefore, we built a SystemC model similar to [36] by using the idea of channels for modeling the places of Petri Nets, and moreover, also for modeling inhibitor-, reset- and timed-arcs. The transitions, which represent the DRAM commands, are modeled as SystemC modules (`SC_MODULE`). Listing 3 shows the basic interfaces for the custom channels and the module for the transitions.

These previously explained classes are the building blocks for implementing the timed Petri Net semantic in SystemC. By using SystemC's hierarchical module structure including the concept of ports, an executable SystemC model which describes the states, commands and timing dependencies can be automatically generated correct by construction from the description in DRAMml, shown in Section 5. Additionally, the code generator can be used to prune superfluous timing arcs, i.e., those that will never be a constraint. As an example, consider the timing t_{PRPDEN} which equals t_{CK} in the DDR3 configuration shown in Listing 1, Line 139. Since all transitions already have the command bus constraint, all arcs depending on t_{PRPDEN} can be pruned (for this specific configuration!).

Listing 3: Abstract SystemC Implementation of Timed Petri Nets Semantic

```
// Place Interface:
class placeIF : public sc_interface {
    virtual void addTokens(int TN) = 0;
    virtual void removeTokens(int TN) = 0;
    virtual bool testTokens(int TN) = 0;
    virtual int getTokens() = 0;
    virtual bool timeTestToken(sc_time MinTime) = 0;
};
// Place Channel:
template<unsigned int I=1, unsigned int O=1>
class place : public placeIF {
    std::queue<sc_time> tokens;
    place(int t) { ... }
    void addTokens( int TN) { ... }
    void removeTokens(int TN) { ... }
    bool testTokens(int TN) { ... }
    bool timeTestToken(sc_time MinTime) { ... }
    int getTokens() { ... }
};
// Timing Interface:
class timingInIF : public sc_interface {
    virtual bool testTiming() = 0;
};

class timingOutIF : public sc_interface {
    virtual void fired() = 0;
};

class timing : public timingInIF, public timingOutIF {
    sc_time delay;
    sc_time lastFired;

    timing(sc_time delay) : delay(delay),
        lastFired(SC_ZERO_TIME) {}
    timing(double d, sc_time_unit u) : delay(sc_time(d,u)),
        lastFired(SC_ZERO_TIME) {}
    void fired() { ... }
    bool testTiming() { ... }
};

// Transition:
template<unsigned int I  = 1,
         unsigned int O  = 1,
         unsigned int H  = 0,
         unsigned int TI = 0,
         unsigned int TO = 0,
         int TN = 0>
SC_MODULE(transition) {
    sc_port<placeIF, I, SC_ZERO_OR_MORE_BOUND> in;
    sc_port<placeIF, O, SC_ZERO_OR_MORE_BOUND> out;
    sc_port<placeIF, H, SC_ZERO_OR_MORE_BOUND> inhibitors[H];
    sc_port<timingInIF, TI, SC_ZERO_OR_MORE_BOUND> timingIn[TI];
    sc_port<timingOutIF, TO, SC_ZERO_OR_MORE_BOUND> timingOut[TO];
    int inhibitorsTokens[H];
    sc_time TokensAge[I];

    SC_HAS_PROCESS(transition);
    transition(sc_module_name name) : sc_module(name),
                                      in("in"), out("out") {}

    int fire() { ... }
    int TimedFire() { ... }
};
```

In the generated model, each DRAM command (transition) provides a `fire()` method, which can be called by the attached simulator or RTL model when a command is issued from the memory controller to the DRAM. In case of a timing violation, or the firing of a disabled transition, it produces an error message.

7 PRACTICAL USAGE

We connected the generated executable model to the most prominent DRAM simulators, namely DRAMSim2 [35], DRAMSys [21], Ramulator [26], as well as the DRAM controller in gem5 [4, 12]. Furthermore, we connected the executable model to an RTL memory controller [38]. The simulator DRAMSys is one of the most sophisticated DRAM simulators, which has support for auto-precharge, power-down, power and temperature simulation, as well as for temperature controlled refresh, etc. Furthermore, DRAMSys has a large testsuite for regression tests and an automated timing checking feature in its TraceAnalyzer tool [21]. For the timing checking feature, DRAMSys records all memory commands and their occurrence in time in an SQLite database. In the TraceAnalyzer the user can write queries to this database and check for potential timing violations. Even though this is a clever test strategy for checking the timing dependencies, we were able to reveal a bug in the DRAMSys simulator. In fact the timing t_{WRPDEN} was violated. This bug was not discovered earlier because the check for this timing did not exist in the TraceAnalyzer, although a test trace for this scenario was existing. This shows the importance of regression models being correct to discover potential DRAM protocol violations immediately. In contrast to the handwritten regression models of TraceAnalyzer our new approach based on a DSL will generate a regression model which is correct by construction. So far, no bugs were found in the other simulators by using the provided test-traces. We will perform further simulations in the future and report to the deveopers if any bugs were found.

8 CONCLUSION AND FUTURE WORK

In this paper we presented a new model for DRAMs using timed Petri Nets, which describes the DRAM states and commands and timings in a correct, compact and complete manner. This model can be used for the simulation-based validation of DRAM controllers and DRAM simulators. As a next step we will use this model for the generation of properties for formal verification of the RTL memory controller and test case generation similar to [14].

ACKNOWLEDGMENTS

This work was supported within the Fraunhofer and DFG cooperation programme (Grant no. WE2442/14-1) and supported by the *Fraunhofer High Performance Center* for *Simulation- and Software-based Innovation*. The authors thank Lukas Steiner, Felipe Prado and Tim Burchert for their support. Furthermore, we thank the anonymous reviewers for their valuable suggestions.

REFERENCES

[1] Tilak Agerwala and Mike Flynn. 1973. Comments on Capabilities, Limitations and Correctness of Petri Nets. In *Proceedings of the 1st Annual Symposium on Computer Architecture (ISCA '73)*. ACM, New York, NY, USA, 81–86. https://doi.org/10.1145/800123.803973

[2] Toshiro Araki and Tadao Kasami. 1976. Some decision problems related to the reachability problem for Petri nets. *Theoretical Computer Science* 3, 1 (1976), 85 – 104. https://doi.org/10.1016/0304-3975(76)90067-0

[3] B. Berthomieu and M. Diaz. 1991. Modeling and verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering* 17, 3 (Mar 1991), 259–273. https://doi.org/10.1109/32.75415

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[5] Karthik Chandrasekar, Christian Weis, Yonghui Li, Benny Akesson, Omar Naji, Matthias Jung, Norbert Wehn, and Kees Goossens. Last Access 15.08.2019. DRAMPower: Open-source DRAM power & energy estimation tool. http://www.drampower.info. (Last Access 15.08.2019).

[6] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth Pugsley, Aniruddha Udipi, Ali Shafiee, Manu Sudan, Kshitij amd Awasthi, and Zeshan Chishti. 2012. USIMM: the Utah SImulated Memory Module, A Simulation Infrastructure for the JWAC Memory Scheduling Championship. *Utah and Intel Corp.* (February 2012).

[7] Hartmut Ehrig, Wolfgang Reisig, Grzegorz Rozenberg, and Herbert Weber (Eds.). 2003. *Petri Net Technology for Communication-Based Systems - Advances in Petri Nets.* Lecture Notes in Computer Science, Vol. 2472. Springer.

[8] Xiaobo Fan, Carla S. Ellis, and Alvin R. Lebeck. 2003. Modeling of DRAM Power Control Policies Using Deterministic and Stochastic Petri Nets. In *Proceedings of the 2Nd International Conference on Power-aware Computer Systems (PACS'02)*. Springer-Verlag, Berlin, Heidelberg, 130–140. http://dl.acm.org/citation.cfm?id=1766991.1767003

[9] Claude Girault and Rüdiger Valk. 2013. *Petri nets for systems engineering: a guide to modeling, verification, and applications.* Springer Science & Business Media.

[10] Matthias Gries. 1998. Modeling a Memory Subsystem with Petri Nets: a Case Study. In *Workshop Hardware Design and Petri Nets HWPN98*. 186–201.

[11] M. Hack. 1976. *PETRI NET LANGUAGE.* Technical Report. Cambridge, MA, USA.

[12] A. Hansson, N. Agarwal, A. Kolli, T. Wenisch, and A.N. Udipi. 2014. Simulating DRAM controllers for future system architecture exploration. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on.* 201–210. https://doi.org/10.1109/ISPASS.2014.6844484

[13] M. Hassan and H. Patel. 2016. MCXplore: An automated framework for validating memory controller designs. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. 1357–1362.

[14] M. Hassan and H. Patel. 2018. MCXplore: Automating the Validation Process of DRAM Memory Controller Designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 5 (May 2018), 1050–1063. https://doi.org/10.1109/TCAD.2017.2705123

[15] Bruce Jacob, S. Ng, and D. Wang. 2010. *Memory Systems: Cache, DRAM, Disk.* Elsevier Science.

[16] Lasse Jacobsen, Morten Jacobsen, Mikael H. Møller, and Jiří Srba. 2011. Verification of Timed-Arc Petri Nets. In *SOFSEM 2011: Theory and Practice of Computer Science*, Ivana Černá, Tibor Gyimóthy, Juraj Hromkovič, Keith Jefferey, Rastislav Královič, Marko Vukolić, and Stefan Wolf (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 46–72.

[17] Min Kyu Jeong, Doe Hyun Yoon, and Mattan Erez. (Last Access: 15.08.2019). DrSim: A Platform for Flexible DRAM System Research. http://lph.ece.utexas.edu/public/DrSim. ((Last Access: 15.08.2019)).

[18] Matthias Jung, Kira Kraft, and Norbert Wehn. 2017. A New State Model for DRAMs Using Petri Nets. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*. 221–226. https://doi.org/10.1109/SAMOS.2017.8344631

[19] Matthias Jung, Deepak Mathew, Christian Weis, and Norbert Wehn. 2016. Approximate Computing with Partially Unreliable Dynamic Random Access Memory - Approximate DRAM. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. ACM, New York, NY, USA, Article 100, 4 pages. https://doi.org/10.1145/2897937.2905002

[20] Matthias Jung, Deepak M. Mathew, Éder F. Zulian, Christian Weis, and Norbert Wehn. 2016. A New Bank Sensitive DRAMPower Model for Efficient Design Space Exploration. In *International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS 2016)*.

[21] Matthias Jung, Christian Weis, and Norbert Wehn. 2015. DRAMSys: A flexible DRAM Subsystem Design Space Exploration Framework. *IPSJ Transactions on System LSI Design Methodology (T-SLDM)* (August 2015). https://doi.org/10.2197/ipsjtsldm.8.63

[22] Matthias Jung, Christian Weis, Norbert Wehn, and Karthik Chandrasekar. 2013. TLM modelling of 3D stacked wide I/O DRAM subsystems: a virtual platform for memory controller design space exploration. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '13)*. ACM, New York, NY, USA, Article 5, 6 pages. https://doi.org/10.1145/2432516.2432521

[23] Matthias Jung, Éder Zulian, Deepak Mathew, Matthias Herrmann, Christian Brugger, Christian Weis, and Norbert Wehn. 2015. Omitting Refresh - A Case Study for Commodity and Wide I/O DRAMs. In *1st International Symposium on Memory Systems (MEMSYS 2015)*. Washington, DC, USA.

[24] M. O. Kayed, M. Abdelsalam, and R. Guindi. 2014. A Novel Approach for SVA Generation of DDR Memory Protocols Based on TDML. In *2014 15th International Microprocessor Test and Verification Workshop*. 61–66. https://doi.org/10.1109/MTV.2014.15

[25] K. Khalifa and K. Salah. 2015. Implementation and verification of a generic universal memory controller based on UVM. In *2015 10th International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*. 1–2. https://doi.org/10.1109/DTIS.2015.7127364
[26] Y. Kim, W. Yang, and O. Mutlu. 2015. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Architecture Letters* PP, 99 (2015), 1–1. https://doi.org/10.1109/LCA.2015.2414456
[27] Y. Li, B. Akesson, K. Lampka, and K. Goossens. 2016. Modeling and Verification of Dynamic Command Scheduling for Real-Time Memory Controllers. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 1–12. https://doi.org/10.1109/RTAS.2016.7461341
[28] Deepak M. Mathew, Éder F. Zulian, Subash Kannoth, Matthias Jung, Christian Weis, and Norbert Wehn. 2017. A Bank-Wise DRAM Power Model for System Simulations. In *Proceedings of the 9th Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO '17)*. ACM, New York, NY, USA, Article 5, 7 pages. https://doi.org/10.1145/3023973.3023978
[29] IEEE Computer Society. 2012. IEEE Standard for Standard SystemC Language Reference Manual. IEEE Std 1666-2011 (2012).
[30] Jedec Solid State Technology Association. 2012. DDR3 SDRAM (JESD 79-3). (2012).
[31] Jedec Solid State Technology Association. 2014. LPDDR4 (JESD 209-4). (2014).
[32] T. Murata. 1989. Petri nets: Properties, analysis and applications. *Proc. IEEE* 77, 4 (Apr 1989), 541–580. https://doi.org/10.1109/5.24143
[33] James Lyle Peterson. 1981. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA.
[34] Carl Adam Petri. 1962. *Kommunikation mit Automaten*. Ph.D. Dissertation. Universität Hamburg.
[35] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *Computer Architecture Letters* 10, 1 (Jan 2011), 16–19. https://doi.org/10.1109/L-CA.2011.4
[36] C. Rust, A. Rettberg, and K. Gossens. 2003. From high-level Petri nets to SystemC. In *SMC'03 Conference Proceedings. 2003 IEEE International Conference on Systems, Man and Cybernetics. Conference Theme - System Security and Assurance (Cat. No.03CH37483)*, Vol. 2. 1032–1038 vol.2. https://doi.org/10.1109/ICSMC.2003.1244548
[37] D. Sahoo and M. Satpathy. 2016. MSimDRAM: Formal Model Driven Development of a DRAM Simulator. In *2016 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID)*. 597–598. https://doi.org/10.1109/VLSID.2016.88
[38] Chirag Sudarshan, Jan Lappas, Christian Weis, Deepak M. Mathew, Matthias Jung, and Norbert Wehn. 2019. A Lean, Low Power, Low Latency DRAM Memory Controller for Transprecision Computing. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Dionisios N. Pnevmatikatos, Maxime Pelcat, and Matthias Jung (Eds.). Springer International Publishing, Cham, 429–441.
[39] H. M. W. Verbeek, M. T. Wynn, W. M. P. van der Aalst, and A. H. M. ter Hofstede. 2010. Reduction Rules for Reset/Inhibitor Nets. *J. Comput. Syst. Sci.* 76, 2 (March 2010), 125–143. https://doi.org/10.1016/j.jcss.2009.06.003
[40] MengChu Zhou and Kurapati Venkatesh. 1999. *Modeling, simulation, and control of flexible manufacturing systems: a Petri net approach*. Vol. 6. World Scientific.