Memory Interface Analysis
Using the Real-Time Model Checker UPPAAL

Egle Sasnauskaite Marius Mikucionis
eglese@cs.auc.dk marius@cs.auc.dk
Department of Computer Science, Aalborg University
Frederik Bajers Vej 7E, 9220 Aalborg Øst, Denmark

July 14, 2002

Abstract

In this paper we present a model of a memory interface, which is a part of a radar system. The memory interface is modelled as a set of connected timed automata with UPPAAL extensions. The system is modeled and verified formally using the verification tool UPPAAL. The system safetiness, proper scheduling and the size of buffers are attempted to be verified and optimized. Partial-order reduction method was a key solution to avoid combinatorial state explosion. We used heuristic periodical approximation to predict the verification space and find close cases which we could verify having particular resources. We succeeded to model and to verify the memory interface with smaller buffer sizes and approximated memory refresh timing within a reasonable amount of time.

1 Introduction

Radar systems have a wide range of applications in air, naval or surface observations and require high precision and reliability. Modeling the memory interface (MI) and verifying the model in the early design phase is important to check if the system complies to the system specifications and avoid more expensive error corrections at a later stage. We define our problem domain in more details in Section 1.1, formulate our goals in Section 1.2 and present an overview of what has been done in Section 1.3.

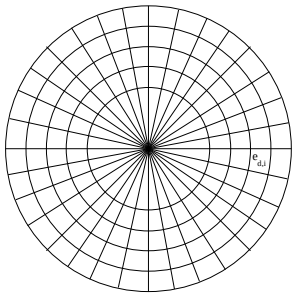
The rest of the article is organized as follows: Section 2 describes the methods we used in modeling and experimenting, Section 3 deals with the details of the constructed model, Section 4 discusses the results of the experiments and finally Section 5 summarizes what has been achieved.

1.1 Motivation

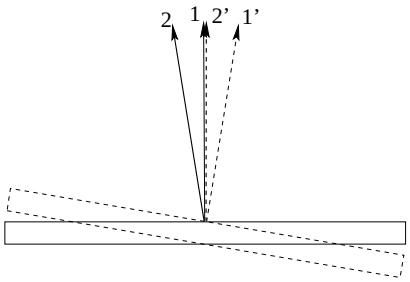
A radar system requires high reliability and the MI is directly responsible for the reliability of data stream manipulation. A verified and inspected MI model is useful to avoid design faults and assure realiability of the real time system.

The classical radar system consists of a rotating antenna, a single high frequency radio wave transceiver (transmitter and receiver unit) and a signal processing unit. The signal processing unit has discrete timing, thus the area around the antenna is divided into location cells (Figure 1(a)) with corresponding signal data $e_{d,i}$, where d is direction and i is a distance index.

Modern sensor systems [1] provide better rain penetration, resolution and accuracy of the radar sensor and are able to distinguish between various aircraft types and vehicles. These features are achieved by dual transmitter-receiver units configured for a frequency diversity. The frequency



(a) Signals from cells.



(b) Squint phenomena with frequency diversity.

Figure 1: Additional data processing in SCANTER 2001 Transceiver.

diversity is achieved by so called squint compensation (Figure 1(b)): an antenna transmits two pulses (1 and 2) of slightly different carrier frequencies in slightly different directions at time t; at the time $t + \delta t$ the antenna transmits other two pulses (1' and 2') and combines the signal (2') with the previous one (1). Here the memory interface needs to calculate the sliding window sum: $sum_{d,i} = e_{d,i} + sum_{d-1,i} - e_{d-m,i}$, where m is the number of sweeps to calculate sum over. Combination of the two signals is used to remove noise.

The MI synchronizes signals by storing data in the memory module. The MI consists of adders, FIFO buffers, registers and an arbiter. Signal data is transfered to the adders for the sliding window sum calculations. Buffers collect data from a narrow input or adders and pass to registers in larger packets. Other buffers are used to receive packets from registers, split into smaller words and transfer back to the adders. A register performs a role of a buffer with larger packets and exchanges data with the memory through a fast and broad data bus. The arbiter takes care of memory addressing and schedules registers for data transfer. The memory (SDRAM) is used for storing signals for a certain time in order to synchronize and combine them. Moreover the data transactions between the memory and registers are suspended while the memory is refreshing. The memory refresh cycle is considered in the model and is controlled by the arbiter.

1.2 Goals and Aims

Our aim is to model and analyze the memory interface between input, output and memory. For that purpose we focus on several issues:

- To model the memory interface as a combination of timed automatons with UPPAAL extensions.
- The model must be small enough (in terms of clocks and states) to be able to verify the system practically within a reasonable amount of time and memory space.
- To verify the system safety property (no deadlock, no underflow or overflow in buffers).
- To optimize the system in terms of the size of the buffers and the arbiter algorithm.
- To summarize the modeling methods for similar systems.

1.3 Contributions and Related Work

The article is a product of a case study, which was provided by the Danish radar manufacturer company Terma A/S. The case study covers modeling the memory interface for the radar system which

uses frequency diversity technology. The model was modeled using the UPPAAL tool developed in collaboration between BRICS at Aalborg University and Department of Computing systems at Uppsala University.

We used a form of partial-order reduction by introducing additional templates into our model.

There are many related projects (e.g. [2], [3]) done in this area since we are using relatively popular automated verification tool UPPAAL [10], but the verification of signal processing systems (such as in our case with buffered input-output and semi-synchronized buses controlling multiple components) with UPPAAL are still quite new.

We use concepts and diagrams [9] to understand, model and describe our case details. The ideas of the scheduler [3] and partial-order reduction [6], [8] are the key solutions widely used in our verification. Finally we propose methods for synthesis of the optimal infinite scheduling algorithm based on similar ideas for finite scheduling in [4].

Having components with periodical behaviour we developed method to predict the approximate amount of resources needed for verification. The concepts and acceleration methods of the periodical timed automata verification discussed in [5] did not help much in our case since the periods of our components are of the same rank and UPPAAL does not support those features yet.

2 Verification Methods

Full system verification establishes that a design or product possesses certain properties from the system's specification. Verification requires more time and effort than construction of a complex software or hardware system [8]. For that reason formal verification techniques are sought to reduce efforts and time, increase effectiveness and to design a system in a more reliable way.

Formal methods make it possible to obtain an early integration of verification in the design phase. Deductive and model based methods of formal verification can be distinguished. We will focus on the second one - the model based verification technique.

Model based techniques are based on models describing the system behaviour in a mathematical precise and unambiguous manner. After the accurate modeling of the system, verification takes place using simulation and model checking. We model the memory interface model as a timed automaton using UPPAAL environment and features.

2.1 Timed Automata

A timed automaton A is a tuple $(L, l_0, E, Label, C, clocks, guard, inv)$, where:

- L is a non-empty finite set of locations with the initial location $l_0 \in L$;
- $E \subseteq L \times L$ is a super set of directed edges.
- $Label$ is a function that assigns to each $l \in L$ a set of atomic propositions;
- C is a finite set of real-valued clocks that evolve at the same rate; a *clock valuation* v over C is a function $v : C \to \mathbb{R}^+$, assigning each clock $x \in C$ its current value $v(x)$; clock x valuation $reset$ in v is defined $(reset\, x\, in\, v)(y) = v(y)$ if $y \neq x$ or $(reset\, x\, in\, v)(y) = 0$ if $y = x$.
- $clocks$ is a function that assigns to each edge $e \in E$ a set of clocks $clocks(e)$;
- $guard$ is a function that labels each $e \in E$ with a clock constraint $guard(e)$ over C;
- inv is a function which assigns an invariant to each l. Invariants on clocks are used to limit the amount of time that may be spent in a location.

Enabling conditions and invariants are $clock\ constrains$. $\Phi(C)$ is a set of clock constraints over C. A clock constraint ϕ is defined as grammar $\phi := v(x) \sim c \mid v(x) - v(y) \sim c \mid \phi_1 \wedge \phi_2$, where $c \in \mathbb{N}$ and $\sim \in \{<, \leq, =, \geq, >\}$. Therefore, the addition of clock variables in clock constraints like $v(x) + v(y) < 3$ would make model checking undecidable.

The semantics of a timed automaton in defined by associating a transition system S_A. A $state$ of a timed automaton is a pair (l, v) where $l \in L$ is location and v is a valuation of a clock(s). Valuation v always satisfies the invariants in location l: $v \vDash inv(l)$.

There are two types of transitions in S_A:

- let $\delta \in \mathbb{R}^+$. We say $((l, v),(l, v + \delta))$ is a $\delta - delay$ transition, iff $v + \delta' \vDash inv(l)$ for all $0 \leq \delta' \leq \delta$.
- let $a \in \Sigma$. We say $((l, v), (l', v'))$ is a $a - action$ transition, iff an edge e exists such that $v \vDash guard(e)$, $v' = v[clocks(e) := 0]$ and $v' \vDash inv(l')$.

Timed automata are used to model finite state real time systems [8]. In our model the components are modeled as separate timed automata. Clocks and data variables [10] are used to define the behaviour of the model. The components form a timed automata network communicating through channels. We used UPPAAL to model, simulate and verify the MI as an extended timed automaton.

2.2 UPPAAL

UPPAAL[10] is a toolbox for symbolic simulation and automatic verification (via automatic model checking) of real time systems modeled as networks of extended timed automata.

UPPAAL uses templates to construct a compound system. A UPPAAL template resembles timed automata with additional features that we use: integer data variables and arrays of such variables, urgent channels and committed locations.

Data variables [11] do not change their values at the delay-transitions as the clock variables do; they can only be assigned to values from finite domains (bounded integers in our case), and therefore they do not cause infinite-stateness. Data variables form *non-clock constraints* similar to *clock constraints* (Section 2.1). $\Phi(D)$ is a set *non-clock constraints*, where D is a set of data variables. A *non-clock constraint* ϕ is defined as grammar $\phi := v(d) \sim k \mid v(d) * v(b) \sim k \mid \phi_1 \wedge \phi_2$, where v is a valuation of a data variable $d \in D$, $k \in \mathbb{Z}$, $\sim \in \{<, \leq, =, \neq, \geq, >\}$ and $* \in \{+, -, \times, /\}$. $\Phi(C, D)$ ranged over $guard$ is used to denote the set of formulas that are conjunctions of *clock constraints* and *non-clock constraints*. The elements of $\Phi(C, D)$ are called $constraints$ or $guards$.

The global arrays of bounded integers are used to share the state information among template instances. We exploit urgent channel synchronization priority among other channels to initialize the template instances before the actual simulation or verification. We use committed locations to resemble atomic ability to take several transitions in a row without other transition interruptions [11].

The tool provides support for automatic verification of safety and liveliness properties of real-time systems. It contains a number of additional features including graphical interfaces for designing and simulating system models.

The model checking with UPPAAL also experiences the problem of the state explosion. The state space of a system made up of several components is a product of the state spaces of the individual parts, and its size is therefore exponential in the number of components. The exploration of the state space is efficient only, when the reachable states are stored in the main memory and the problem occurs when the available amount of the main memory is highly insufficient. To overcome state-space explosion different techniques are used [8]. In our case we use a form of partial order reduction.

2.3 Partial-order Reduction

Partial-order reduction [6] is a verification method for concurrent finite state systems that avoid combinatorial explosion due to the modeling concurrency by interleaving. This method is typically applied to asynchronous systems consisting of several parallel components, which are described using an interleaving model of computation.

Partial-order reduction is a technique used to reduce the complexity of state space exploration. It explores a restricted number of independent concurrent transition interleavings, while preserving the verified property in the reduced model [7].

Concurrent events are modeled by allowing their execution in all possible orders relative to each other. This serialization creates a large number of possible states and paths. However, not all different interleavings can be generally distinguished by a specification. Partial-order reduction techniques take advantage of this by generating and exploring a model with only a reduced set of interleavings, and thus fewer states.

We use a form of static partial-order reduction method [8] at the model level and not as a technique for state space exploration during the actual searching. We obtain partial-order reduction in the model by introducing additional components: a timer and a bus. However forced event serialization order was not enough for our exact study case.

2.4 Heuristic Periodical Approximation

We exerienced that the complete memory interface verification is very time and space consuming even if the order totally determined. We found an approximate solution that seems to be feasible in practice. This technique lets us to predict how many system states have to be explored to fully verify it. These calculations can prevent us from running exhaustive queries and give us an idea of how to optimize the model.

A path [8] of timed automaton is an infinite sequence $s_0, a_0, s_1, a_1...$ of states alternated by transition labels such that $s_i \rightarrow^{a_i} s_{i+1}$ for all $i \geq 0$, where labels are either $a_i = *$ (action transition) or $a_i \in \mathbb{R}^+$ (delay transition).

A compressed path (trace) [5] of timed automata is a finite or infinite sequence $s_0, a_0, s_0, a'_0, s_1, a_1, s_1, a'_1...$ of states alternated by transition labels such that $s_i \rightarrow^{a_i} s_i$ $a_i \in \mathbb{R}^+$ is a delay transition in location s_i and $s_i \rightarrow^{a'_i} s_{i+1}$ $a'_i = *$ is an action transition.

A state cycle of a timed automaton is a finite sequence s_k, a_k, s_k, a'_k, s_{k+1}, a_{k+1}, s_{k+1}, a'_{k+1},...,a_{k+n-1}, s_{k+n-1}, a'_{k+n-1} which appears as a suffix in a compressed path, where $s_k = s_{k+n}$ and $i \neq j \Rightarrow s_i \neq s_j$ for all $k \leq i, j \leq k + n - 1$.

Obviously the timed automaton loops infinitely if it reaches the state cycle and has no other alternative enabled edge to escape the cycle. This property is experienced in MI model since only one edge at most is enabled at a time. UPPAAL stops the verification exactly when it encounters the first state cycle iteration end - this is why the verification space is approximately proportional to the period of the state cycle.

A period of a state cycle s_k, a_k, s_k, a'_k, s_{k+1}, a_{k+1}, s_{k+1}, a'_{k+1},...,a_{k+n-1}, s_{k+n-1}, a'_{k+n-1} is the time elapsed in one state cycle iteration: $p = \sum_{i=k}^{k+n-1} a_i$.

Note that a state cycle is different from a timed automaton location cycle defined in [5]. Acceleratable cycles are not explicitly expressed in our separate component models. Moreover the execution of the local component cycles highly depend on the initial variable values and components, which share the same global variables, thus making it impossible to compute an *acceleration [5]* of separate component models. The acceleration of MI cycles would require to transform the compound automata network into a single timed automaton consisting of $2^1 \cdot 5^1 \cdot 10^2 \cdot 7^9 \cdot 10^9 \cdot 6^1 \approx 2.4 \cdot 10^{20}$ states in our model.

Instead of accelerating the non-expressed cycles we explore how the compound system may be divided into several separate subsystems which interact with each other as little as possible to

preserve the property of a constant period. Then we try to find the periods for the smaller subsystems which is much less time and space consuming. Knowing the periods of the subsystems we calculate the period for the full system which is expected to be the *least common multiple* of the subsystems periods.

The *least common multiple* assumption is clearly true for unrelated periodical subsystems, but nothing can be said about closely interacting subsystems, which change each others periodical properties. So far, this case study shows that even closely interacting memory interface subsystems behave at least similarly to totally unrelated pseudo ones in terms of verification time and space.

3 Architecture of Memory Interface Model

The goal of the MI is to utilize the fast and broad memory bus to store and process the slow and narrow signal data streams. To ensure the continuous data-lossless transmission, the interface consists of a number of functional components shown in Figure 2[1]: adders, first-in-first-out (FIFO) buffers, registers, SDRAM memory and an arbiter which schedules the memory access. The notion "16b@10-100ns" means that a bus clock period lasts from 10 to 100 nano-seconds and transfers 16 bits during each cycle.

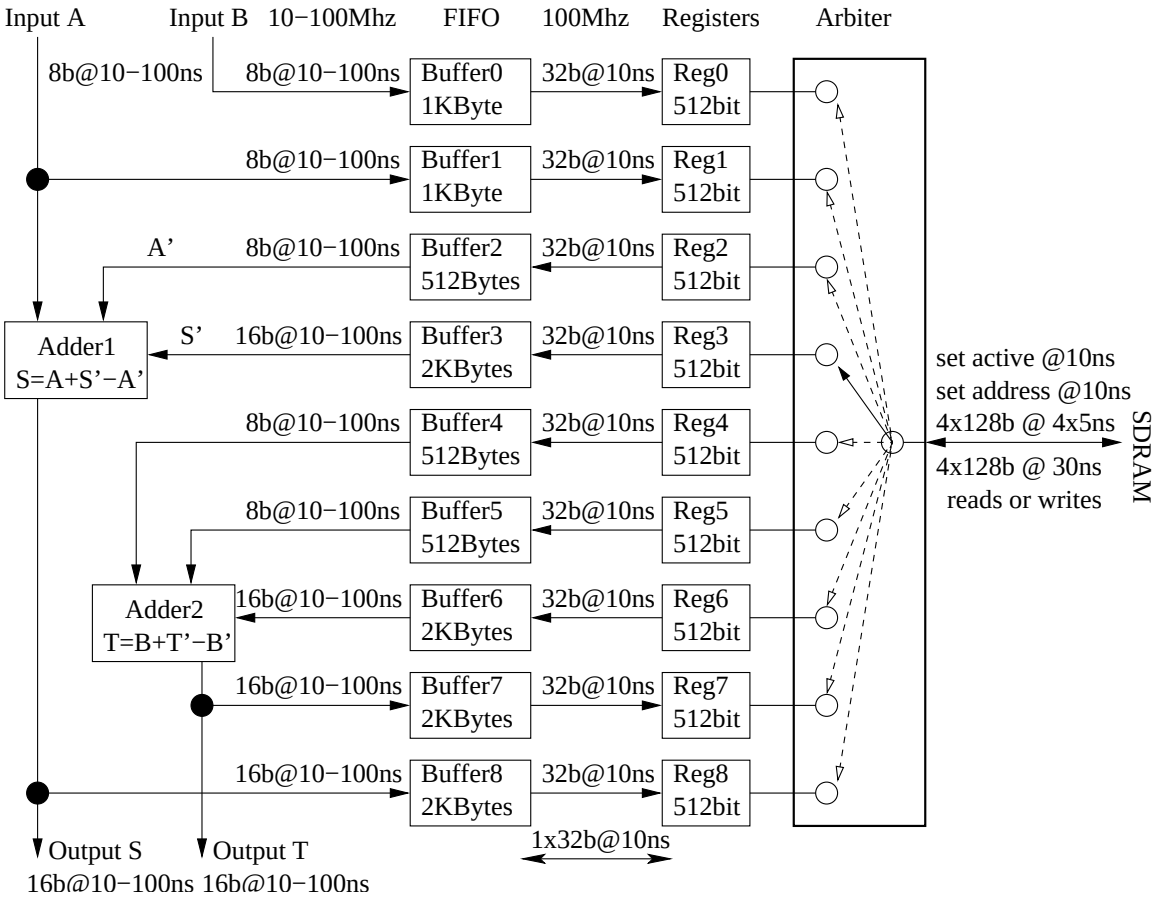


Figure 2: The memory interface implemented in the Scanter 2001 Transceiver[1].

All functional components are connected via wires which form several data busses. As noted in Figure 2 input, output, adders and buffer parts are working on the same logical bus which runs at

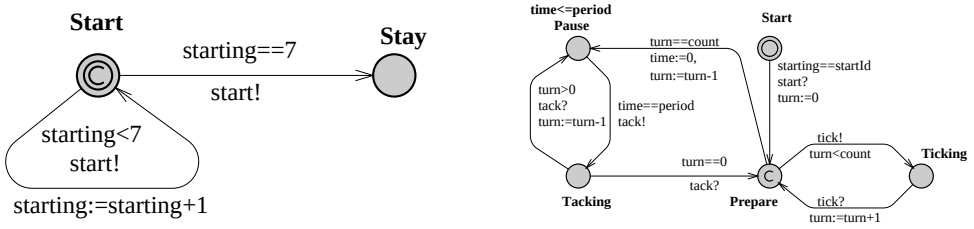
[1]We performed experiments on 8 times smaller buffers, treating bytes as bits.

some chosen frequency from 10MHz to 100MHz (depending on antenna rotation speed, resolution and locating distance). The second bus connects buffers and registers and runs at fixed 100MHz bus. The third bus is exclusive to only one register at a time, it runs at 100MHz through the arbiter, and transfers 128 bits in each half-period.

We describe all templates involved in the verification in descending order of abstraction level: at first we describe models responsible for the partial ordering technique (Section 3.1), then models connecting the functional components (Section 3.2) and finally the functional component details are in Section 3.3.

3.1 Order Control Templates

The MI model contains additional models that are responsible for the partial-order reduction. The atomic events are coordinated by series of channel synchronizations through committed locations. By default any output channel synchronization may occur with any input channel synchronization. To achieve a complete serialization of synchronizations we add order variables, which are shared and supported in both: order control and ordered templates instances such as buses (controlled by $timer$) and others (initialized by $starter$).



(a) The starter template.

(b) The timer template.

Figure 3: Templates for complete partial-order reduction.

3.1.1 Starter

The starter in Figure 3(a) initializes deterministicly all instances in order to avoid a combinatorical state explosion. To support serialized initialization all models must start by receiving $start$ channel synchronization while the global variable $starting$ is equal to its $startId$.

3.1.2 Timer

There is the only clock in the MI model and it is built into a separate template of a timer shown in Figure 3(b). The connectivity models are synchronized through the timer and the timer implements the partial order reduction among them.

After initialization the timer sends the cycle start signal and waits for a reply signal from every connectivity model ($tick$ channel synchronization). After they have started the timer waits for a period ($5ns$) to elapse and sends the end cycle signal (the $tack$ channel synchronization). The timer ends the cycle in a backward order - the first bus to start is the last to end (later this feature is exploited in the bus template).

3.2 Connectivity Models

Connectivity models control the order of actual event appearance, such as the start and the finish of sending or receiving data. We do not model the actual data flow, we consider only the timing and amount of data transfered, which is enough to detect data underflows and overflows. Alos the calculation of memory address for the data to be written or read is out of scope of the article; we consider only the time it takes for the arbiter to setup an address in the memory.

3.2.1 Bus

Our bus is a logical abstraction of all wires transferring data at the same timing and not the actual bus shared by components transfer the data just between two points. The bus provides a form of static partial-order reduction among events on the functional components (Section 3.3) level. There are three busses in the MI - one of them is built into the arbiter. The bus sends *begin* and *end* cycle signals to every functional component through the beg or end channel synchronization while the variable bus holds an ID of the component to receive an event (Figure 4). Every instance of

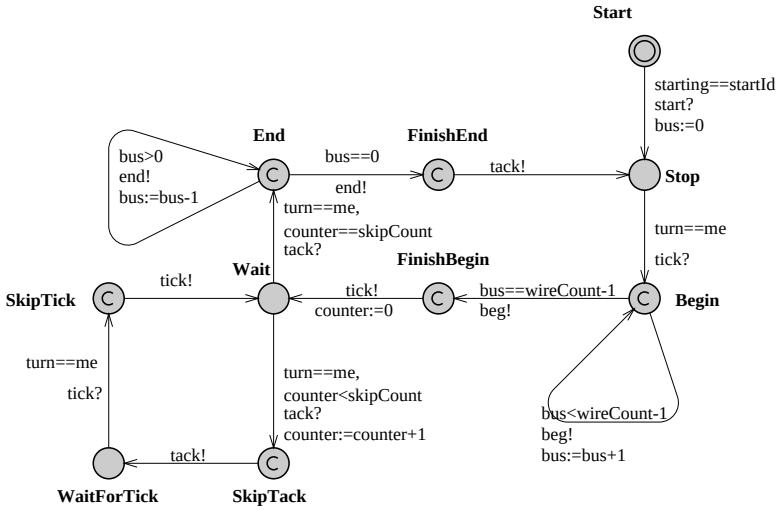


Figure 4: The template of the bus.

the bus has its own component ID to support flexible component configuration change. There are additional locations $SkipTack$, $WaitForTick$ and $SkipTick$ in the bus template to support longer bus periods than the period used in the timer.

3.2.2 Arbiter

The arbiter component has three purposes in the MI: simulates the memory (refresh), excludes the memory data bus to some register and provides partial bus functionality.

The arbiter template consists of three cycles (Figure 5): bus switching when a register is not ready, memory refresh when a register is ready and data transfer from a register to the memory. A bus switch cycle lasts for $10ns$ and takes place when a particular register is not ready. The arbiter chooses the next register from a queue of registers if the present register is not ready. After checking the last one in the queue the arbiter switches to the first one, i.e. the arbiter provides round-robin scheduling among the registers. That kind of scheduling is utilized in Terma's radar systems.

The data transfer cycle starts when a particular register is ready and has two phases: sets up the address for incoming data in the memory (lasts $10ns$) and transfers data in four chunks of 128 bytes

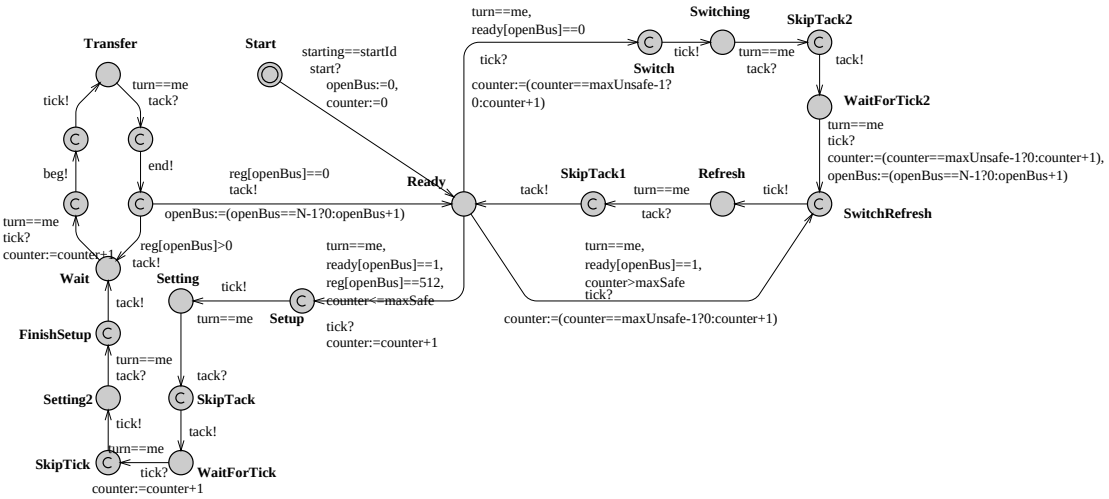


Figure 5: The template of the arbiter.

per $5ns$ each and delays $20ns$ in total.

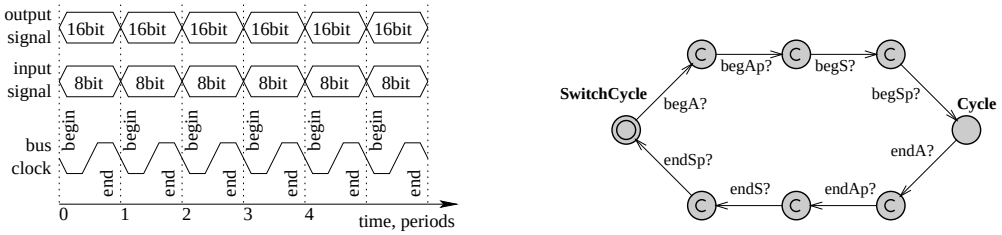
The arbiter simulates a memory refresh cycle (Section 3.3.4). The arbiter takes precautions against interruption of the register data transfer. There is a maximal safe period to start transferring data without interruption of data refresh. The arbiter suspends a data transfer when the maximal safe moment is reached ($counter <= maxSafe$). The arbiter can still switch registers during the memory refresh and if a register is ready for data transfer the arbiter just waits for the memory refresh to finish.

3.3 Component Models

In this section we describe the main components which were displayed in the transceiver memory interface in Figure 2. The memory interface contains input, output, adders. Their functionality is described in Section 3.3.1. A buffer is a region of memory reserved for use as an intermediate repository, which accumulates data signals into packets and then passes them on for further processing (Section 3.3.2). A register is a component closely connected to a certain buffer. It splits 256 bit words into 128 bit words and exploits broad memory bus to transfer them during each half-cycle. The details are described in Section 3.3.3.

3.3.1 Input, Output and Adders

After the bus concept introduction (Section 3.2.1) the input, output and even adders become trivial. The input sets up a new word (8 bits) transfer into the system by the beginning of a bus cycle, stops the transfer when the cycle ends, starts a new 8 bits transfer on the next cycle beginning and so on (see diagram in Figure 6(a)). The output component does the same thing with a different amount of data. The $adder$ component needs several data inputs/outputs to receive all needed data and to send a previous operation result. However these events happen at the same time: they start on the beginning of a cycle and finish at the end. The above implies the adder model shown in Figure 6(b). Having these arguments we do not put these components into the compound model since they just take up the verification memory resources and have no impact to actual verification.



(a) The bus events, the input and output signal timed diagram.

(b) The adder component model.

Figure 6: Input, output and adders are stateless components.

3.3.2 Buffer

A buffer is responsible for asynchronous data flow between the first (wires connecting input, output, adders and buffers) and the second (wires connecting buffers and registers) buses. There are two templates for the buffer component: *inBuffer* and *outBuffer*.

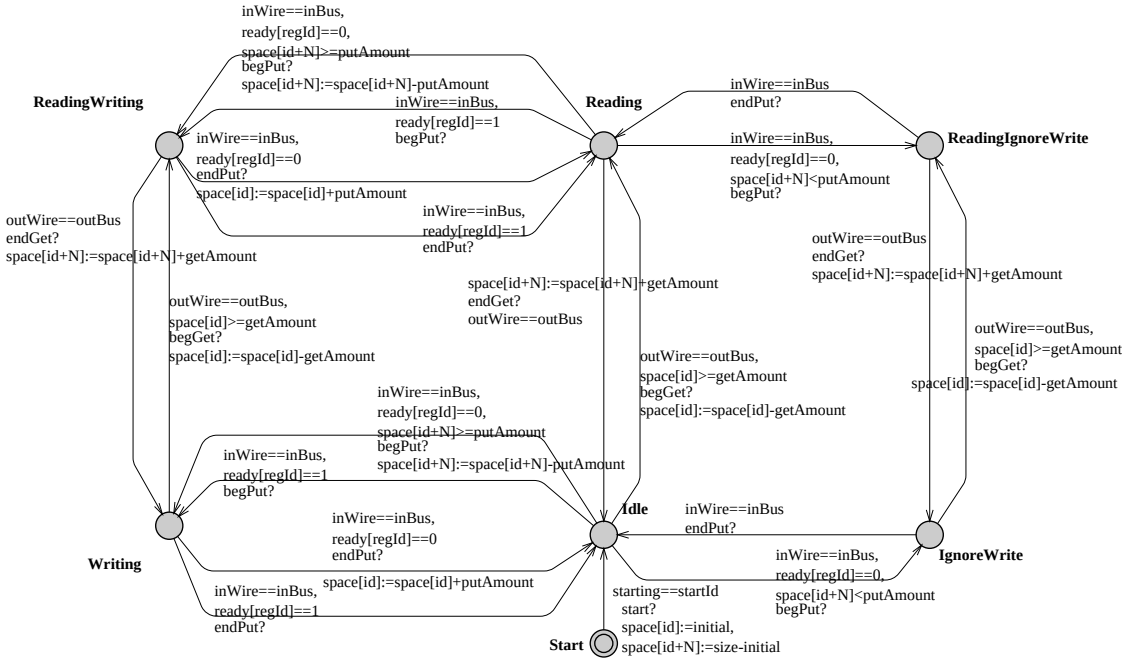
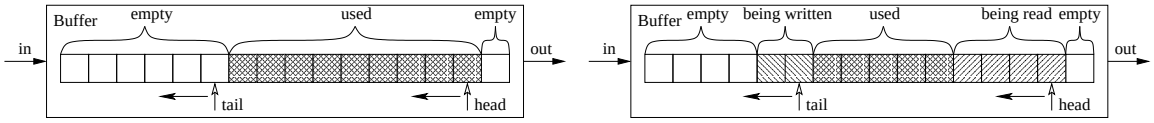


Figure 7: The UPPAAL template for the output buffer.

The $outbuffer$ template has parameters: *startId* for the initialization, *id* specifies the index used to access space array, *size* specifies the capacity of the buffer, *initial* initializes the buffer with requested amount of data, *regId* specifies the register the buffer is bounded to, *inWire* and *outWire* specifies number in the inputting and outputting bus queues, *inBus* and *outBus* provides the ordering in inputting ant outputting buses, *begPut* and emphendPut are synchronizations for beginning and ending the input, *putAmount* specifies the amount of data put at one bus cycle, *begGet*, *endGet*, *getAmount* mean the same for the outputting.

The output buffer (Figure 7) reads data from a register through a bus in $32b@10ns$ and outputs data to an adder through amother bus in $16b@10ns$ or $8b@10ns$. An important issue is the amount

of used and empty space in the buffer. The filled memory amount is stored in the space variable $space[id]$ where id is the indentificator of the buffer. The variable $space[id+N]$ stores the amount of the empty memory space, where N is the number of buffers. Figure 8 shows the possible buffer images which map to states *Idle* and *ReadingWriting* in Figure 7.



(a) Idle.

(b) Read and write.

Figure 8: Possible buffer implementation: various states.

The buffer writes, reads or does both operations at the same time if certain properties hold to perform a transaction: for writing the empty space is not smaller than the incoming amount of data $space[id+N] >= putAmount$; for reading the used space is not smaller that the outgoing amount of data $space[id] >= getAmount$; the register is ready for transaction $ready[regid] = 0$.

A peculiar property of the *outBuffer* is that it agrees with the register about getting data but it can not ignore constant out-going data flow.

The *inBuffer* performs the same operations, but it gets data from the input or the adder in $16b@10ns$ or $8b@10ns$ and transfers the data to a register in $32b@10ns$. Both templates are similar and have a "mirror" view of each other, just $ignoreWrite$ state is logically replaced by $ignoreRead$ in *inBuffer*. All buffers are connected to the same busses, but the variable for $inWire$ for one template becomes $outWire$ for another and vice versa.

Buffers and registers form tightly connected pairs which begin and end communication in a certain order described in Section 3.3.3.

3.3.3 Register

Every register is closely connected to a certain input or output buffer. The register template is designed to be used in both cases: to upload and download data to and from buffers through $begBuff$ and $endBuff$ channel synchronization. A register also exchanges data with the memory through $begMem$ and $endMem$ channel synchronization (Figure 9).

The register template consists of two main cycles: interaction with a buffer and data exchange with memory. The register interacts only with a certain buffer until the register gets full and is ready to communicate with the memory. Data transfer is suspended if the buffer is filled or contains no enough data.

A different situation is when the register gets full and ready to communicate with the memory. A register needs additional states and transitions to remember the buffer bus state during communication with the memory.

The buffer and the register are tightly connected and rely on the order of the bus events: the register needs to start a cycle before the buffer does since it needs to know the buffer state before cycle start; the buffer needs to end the cycle before the register does since the buffer needs to know the register state before the cycle ends. This ordering is implemented in the bus template: a bus cycles ends in an opposite ordering than it starts.

3.3.4 Memory

The memory used in MI is a Synchronous Dynamic Random Access Memory (SDRAM), which needs to be refreshed from time to time not to loose the data. The main argument to model the

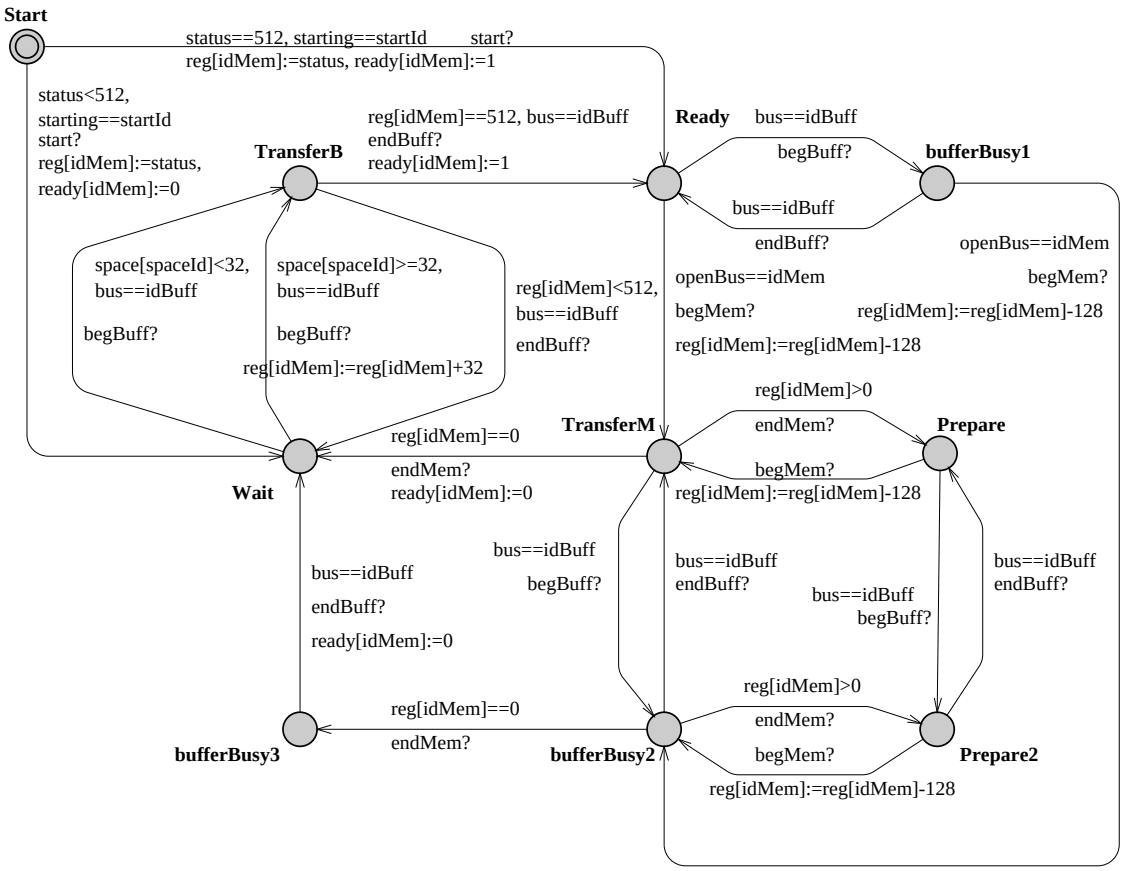


Figure 9: The template of a register.

memory is that memory can not participate in data transmitions during its refresh time.

The memory refresh period consists of the time before refreshing $15625ns$ and refresh time $100ns$. The whole memory refresh period lasts for $15725ns$. There can be other alternative memory refresh implementations, such that memory chip can remember only for $15625ns$ and thus must be refreshed before $100ns$ in advance. Such memory refresh would have the period of $15625ns$.

Memory refresh process isolates memory from other components, while they still have to sustain a data flow without interruptions. We model the behaviour of the memory inside the arbiter (Section 3.2.2) since the refresh process is simple enough. A separate template for the memory would take much efforts to simulate the synchronized actions with arbiter.

4 Verification Results

To develop and verify a complete compound MI at once is not an easy task. We have developed a much smaller similar system to experiment on first. The small system is connected the same way like the complete MI (Figure 2), except it consists of just two buffers, two registers, and the arbiter being the same. The input buffer receives 8 bits and the output buffer sends 16 bits.

Facing the periodical state explosion at first we search for the MI period length without memory refresh mechanism. However UPPAAL does not provide a straightforward mechanism to find the very long cycles or periods of the model (UPPAAL takes much memory and time to dump it all). The experiment procedure is a binary search for the first start and the first end of the system period. The search is formulated as a number of queries like $\exists\Diamond System.State == startState\, and\, systemtime >$

$startTime$ which is a shortcut for "all component states are equal to the cycle starting state after the period starting time". The results displayed in Table 1 are relevant to the model timing characteristics.

Period properties:	**Start (ns)**	**End (ns)**	**Length (ns)**
Small MI period:	80	720	$640 = 2^7 \cdot 5$
Complete MI period:	520	6280	$5760 = 2^7 \cdot 3^2 \cdot 5$
Memory refresh period:	0	15725	$15725 = 5^2 \cdot 17 \cdot 37$

Table 1: System periods without memory refresh.

After experimenting with different memory refresh timings we got the results summarized in Table 2. We were able to verify the small MI in any case of refresh timing, but had problems with

Model	**Case**	**Refresh period**	**LCM**	**Verification space**
Small MI	actual	$15725ns$	$2012800ns$	$14.0min(1.3G)$
Small MI	close	$15720ns$	$251520ns$	$200sec(346M)$
Small MI	fastest	$15360ns$	$15360ns$	$6sec(17.6M)$
Complete MI	actual	$15725ns$	$18115200ns$	not verified
Complete MI	close	$15720ns$	$754560ns$	$8min(777M)$
Complete MI	fastest	$15360ns$	$46080ns$	$82sec(136M)$
Complete MI	other	$15600ns$	$374400ns$	$7.5min(735M)$

Table 2: The small and complete memory interface verification results.

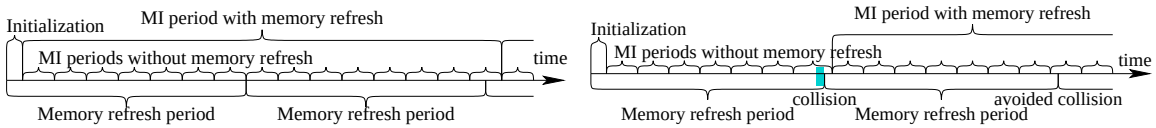
the actual case of the complete MI. Nevertheless we tried a bit different refresh timings which led to success. We successfully verified the very close case of the complete MI: the chosen $15720ns$ refresh period is reasonable since the memory interface could be programmed to refresh the memory a bit earlier without causing any damage, while the longer memory refresh timing seems to be more advantageous to the system. The "other" case in Table 2 is the closest to alternative memory refresh implementation we were able to verify.

After a number of experiments we discovered that the amount of memory used for small MI verification is proportional to *least common multiple* (LCM in the Table 2) discussed in Section2.4. This draws a conclusion that the small MI interacts with the memory refresh mechanism reasonably little. After an exhaustive search for verification space patterns in the complete MI we failed to find any tolerable correlation with LCM (Figure 13). We noticed clear correlations in cases with small LCM but not in larger ones. We observed that the ratio of the verification space and LCM varies $0.29 - 6.01KB/ns$. We can make an assumption that the actual case of the complete MI can be verified having $18 \cdot 10^6 \times (0.29 \div 6.01) \approx 5.2 \div 108.7G$ memory space. While the model is completely deterministic we need not to worry about the verification time: time in our verification is proportional to the memory space used and memory ended up being the critical resource, not time.

4.1 Experiment Data

Points above 4GB are not completely verified (because of the lack of memory). The hypothetical points in Figures 12 and 13 were computed with the following equation: $VerificationSpace = LCM(p_s, p_r) \cdot m_{ns}/1024$, where p_s is a period of the system without memory refresh, p_r is the memory refresh period, m_{ns} is an average memory amount per $1ns$ verification. $m_{ns} = 699bytes$ for the small MI and $m_{ns} = 2268bytes$ for the complete MI. m_{ns} for the small MI was adopted according to the most exhaustive verification, and m_{ns} for the complete MI came from the observation that the average memory space consumption is about 3 times larger than in the small system

according to the ratio of variables used in both systems. According to Figure 13 a complete MI may be verified in $\approx 38.26G$ space.



(a) Period lengths without collisions are predicted.

(b) Single collision shifts and prolongs the period.

Figure 10: Examples of period interactions.

A great difference between hypothetical and experimental data in $15360ns$ and $15680ns$ memory period cases in Figure 12 can be explained by collisions between MI periods and memory refresh period: if the memory refresh happens just before some register become ready to exchange data with the memory, the exchange is postponed and the MI period is prolonged (Figure 10(b)). Each such collision can cost a time which is no longer than the MI period in an ideal situation drawn in Figure 10(a). So far, the experimental data exceeded the hypothetical data no more than 2 times, i.e. there was one collision at most. Such collision, in some cases even helps to get to the situation where we have already been. This explains why some of the results verified faster than expected in Figure 13.

4.2 The Partial-Order Reduction Influence

Starter. We have 23 models in the system, so the $starter$ template needs to start the remaining 22 instances. Without the $starter$ template UPPAAL would initialize our models in a non-deterministic way having $22! \approx 10^{21}$ calculation branches instead of just one.

Timer. We have 3 buses running in parallel which would start $3! = 6$ non-deterministic calculation branches on each beginning of a timer period. On each ending of a timer period all branches meet at the same point, but until that point the verification memory has still wasted almost six times than actually needed.

Bus. We have 9 components on the input/output bus, 18 on the buffer-register bus and 9 on the arbiter bus. The bus elements save $9! \approx 3.6 \cdot 10^5$, $18! \approx 6.4 \cdot 10^{15}$ and $9!$ similar transition configurations respectively on the start of each bus cycle.

4.3 The Arbiter Algorithm

Verification showed that round-robin algorithm is sufficient for the specified buffer sizes. The arbiter algorithm is an important issue if we would like to optimize the buffer sizes. Inspired by [4] we propose to synthesize the more optimal arbiter algorithm by making it non-deterministic and searching for the proper schedule to satisfy liveliness property ($\exists\Box$ *not buffer.overflow and not buffer.underflow*). To implement this idea we need two additional states in the buffers to identify overflow and underflow, since they just deadlock the system in case of underflow or overflow in our present model.

As discussed in 4.2, the non-deterministic arbiter will explode the number of states in the system: if we assume at least two registers are ready at each timer cycle start and the system runs in periods of n timer cycles, we would have 2^n non-deterministic calculation branches. These non-deterministic branches may never end up, leading to the halting problem if such scheduling algorithm does not contain cycles or does not exist for particular buffer sizes. To avoid non-determinism we propose simple heuristics to choose particular register: choose the fullest buffer of the corresponding ready register. The reason is that the most filled/emptied buffer is most

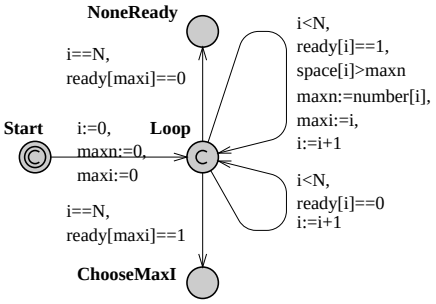


Figure 11: The UPPAAL template for finding the maximum.

likely to be overflowed/underflowed. Figure 11 shows a possible part of arbiter scheduling algorithm to choose ready register according to the maximum filled buffer. These heuristic solution is suitable only for the algorithm synthesis and not in practice, unless we could implement constant-time register decider in practice. In this case the verification path still could be long and generate a complex arbiter algorithm. The upper limit for this algorithm length would be $(1024/8)^2 \cdot (512/8)^3 \cdot (2048/16)^4 \cdot 9 = 9 \cdot 2^{60} \approx 10^{19}$ - the total state space of the system (multiplication of buffer sizes divided by granularity and the arbiter switch position count).

4.4 The Buffers Sizes

The size of the buffers used in the models was eight times smaller than proposed by Terma A/S. But the ratio among buffer sizes was kept the same. Because of much smaller buffer sizes we were able to verify the models in reasonable space and time.

Having a round-robin scheduling algorithm in the arbiter proved to be simple and sufficient solution for all verified memory refresh timing cases with existing buffer sizes. However any attempt to reduce the buffer sizes led into various deadlock situations where random buffers were overflowed or underflowed. We could not recognize the patterns which buffer sizes should be adjusted even to minimize the total buffer sizes.

5 Conclusion

We proposed a UPPAAL model of the memory interface which is small enough to verify the system practically within a reasonable amount of time and space with reasonably approximated memory refresh timing. According to our approximate calculations we would need about 38GB of memory to verify the model with exact memory refresh timing.

Our model is verified to be deadlock free in many cases but we experienced lack of the memory for the actual case verification.

So far all investigations led to negative results towards buffer size optimizations. However it may be possible with another arbiter algorithm.

The designed model of the memory interface is flexible to be adopted to systems with various component configurations. So far we noticed that:

- the bus template can be optimized to be aware of readiness of other components to avoid irrelevant cycles;
- the arbiter could have memory refresh in the middle of a register data transfer, which would minimize the memory refresh influence to MI periods.

We believe that the model can be extended to simulate actual data flow using additional arrays of bounded integers if needed.

Future UPPAAL versions will have arrays of channels. They could simplify the templates to avoid shared bus variables that control the order of other components.

References

[1] TERMA Elektronik AS. Product Specification SCANTER 2001 Transceiver, 2000.

[2] Klaus Havelund, Kim Guldstrand Larsen and Arne Skou. Formal Verification of a Power Controller Using the Real-Time Model Checker UPPAAL.

[3] Kim Guldstrand Larsen and others. Model-Checking Real-Time Control Programs Verifying LEGO©MINDSTORMS^{TM} Systems Using UPPAAL.

[4] Guided synthesis of control programs using UPPAAL for VHS case study 5. Thomas Hune, Kim G. Larsen, and Paul Pettersson. VHS deliverable in Workpackage CS.1.1, 1999. http://www.cs.auc.dk/research/FS/VHS/hlp99.ps.gz

[5] M. Hendriks and K. G. Larsen Exact Acceleration of Real-Time Model Checking, Technical Report 22. CSI University of Nijmegen, December 2001. ETAPS 2002. Theory and Practice of Time Systems (TPTS'02) April 6-7, Grenoble, France.

[6] P. Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In Proc. 2nd Workshop on Computer Aided Verification, volume 531 of Lecture Notes in Computer Science, pages 176-185, Rutgers, June 1990. Springer-Verlag. Extended version in ACM/AMS DIMACS Series, volume 3, pages 321-340, 1991.

[7] Marius Minea. Partial Order Reduction for Verfication of Timed Systems. December 1999, CMU-CS-00-102. School of Computer Science Carnegie Mellon University Pittsburgh, PA 15213.

[8] Joost-Pieter Katoen. Concepts, algorithms and tools for Model Checking. Lecture notes of the course "Mechanised validation of paralel systems". Course number 10359 1998/1999.

[9] Kenneth J. Breeding Digital design fundamentals. Englewood Cliffs, N.J. : Prentice Hall, cop. 1992. ISBN- 0132112779. P. 113

[10] Franck Cassez et al. Modeling and verification of parallel processes: 4th summer school, MOVEP 2000 Nantes, France, June 19 - 23, 2000: revised tutorial lectures. LNCS 2067. Berlin : Springer, 2001.

[11] Paul Pettersson. Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice. A dissertation in Computer Systems for the degree of Doctor of Philosophy. Publicly examined in room X, Uppsala University, 19 February 1999. Technical Report DoCS 99/101. ISSN 0283-0574.

Appended Experiment Data Graphs

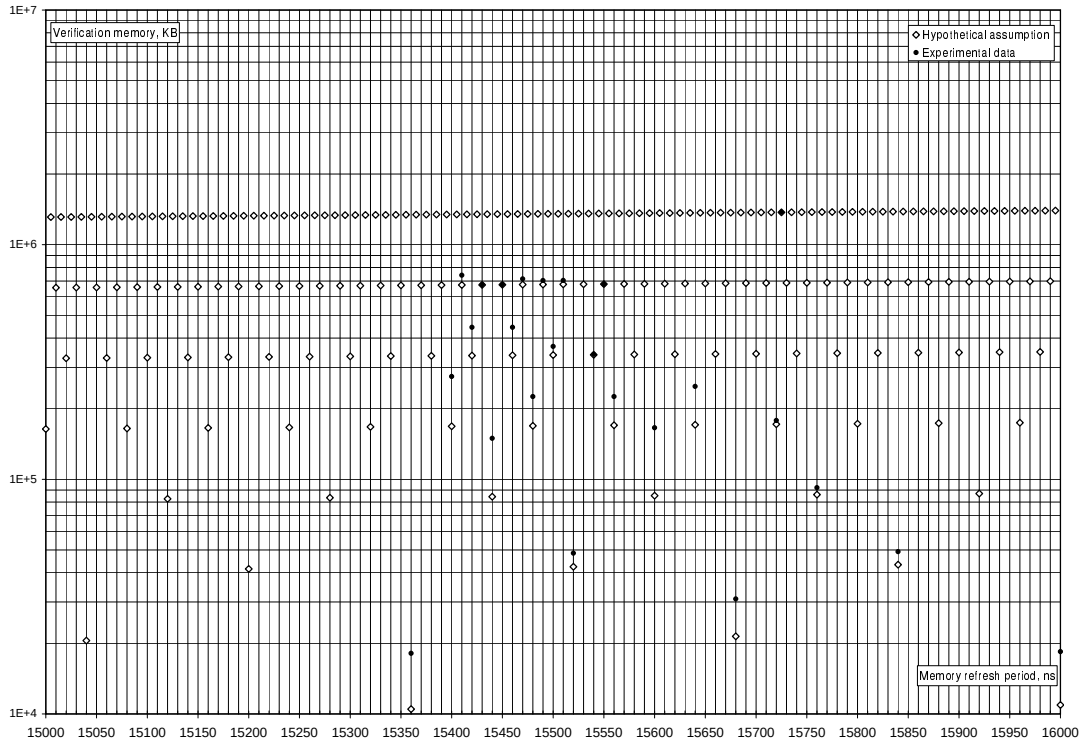


Figure 12: The small MI verification space on different refresh periods.

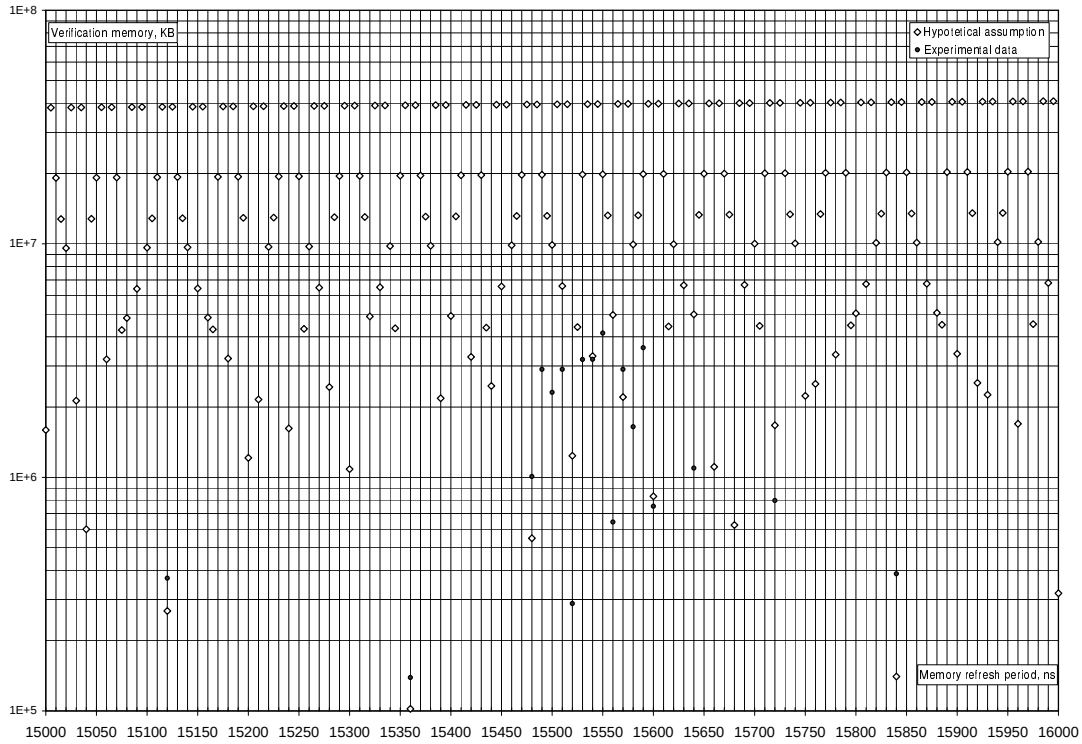


Figure 13: The complete MI verification space on different refresh periods.