Nordic Journal of Computing

MEMORY ARBITER SYNTHESIS AND VERIFICATION FOR A RADAR MEMORY INTERFACE CARD

JUHAN ERNITS
Institute of Cybernetics
Akadeemia tee 21, 12618 Tallinn, Estonia
juhan@cc.ioc.ee

Abstract. In this paper synthesis and verification are carried out by specifying two different problems of logic model checking which are solved by applying a concrete model checker — Uppaal and its extended counterpart Uppaal CORA. A great deal of care is taken in constructing the corresponding models as abstractions of the behaviour of the system from the point of view of memory communication. This results in a rather simple abstract model of the system which is one of the key factors to the success of the synthesis task. The other key factor is the application of a sound but incomplete method for space saving — bit-state hashing, where each visited state is represented by one bit in the hash table, in a location determined by the hash of the state.

We analyse a radar memory interface case study of the IST AMETIST project. The reasoning in this paper presents a way to synthesise a memory arbiter for the system, to minimise memory used for buffering the data streams and to verify that the resultant arbiter does not deadlock and never starves nor overflows any of the intermediate buffers.

ACM CCS Categories and Subject Descriptors: B.8.2 Performance Analysis and Design Aids; C.4 Modelling techniques.

1. Introduction

The main goal of this work is to act as a particle of catalyst in the process of tailoring formal methods and tools to assist system engineers in their day-to-day work. Formal methods make the resultant product better in some relevant sense. For example cheaper if it appears that the task can be solved using less components, and more reliable by discovering the overlooked behaviours of the system as counter-examples emerge during automatically checking whether the system conforms to certain desired properties.

Merging formal methods into industry practice is a long term process and there is no fear that the work presented here will put anybody out of work due to a sudden increase of productivity of system design tools. On the other hand, it is a contribution in the direction of integrating verification tools into the system design process.

Received February 23, 2005

In this paper we analyse a radar system memory interface card described in a case study from the IST Advanced Methods for Timed Systems project, AMETIST [2002–2005]. It was contributed to the project by Terma A/S [Behrmann *et al.* 2002]. The memory interface card performs signal processing calculations on two streams of input data and their delayed counterparts. The stream data is temporarily stored in synchronous dynamic RAM (SDRAM) that is shared by all streams. Dynamic memory is generally considerably less costly in larger amounts than static memory, which is used for intermediate buffers.

We present a way to *synthesise* a memory arbiter for the system, to minimise the amount of static RAM used for buffering the streams and to *verify* that the resulting arbiter does not deadlock and never starves nor overflows any of the intermediate buffers. Both the synthesis and the verification problem are solved by *model checking*.

The synthesis task is generally computationally harder than verification, thus we need to apply a number of abstractions to the system to make the task solvable by model checking.

1.1 Model checking

Logic model checking is a method to check, by exhaustive examination of the state space, whether a model (of, for example, software or hardware) exhibits some collection of properties that describe the desired behaviour of the system. The method was independently developed by Clarke and Emerson [1982] and by Queille and Sifakis [1982]. Since then model checking has been a target of wide and active research and has been applied successfully in several cases, as in [Clarke *et al.* 1993], [Bengtsson *et al.* 1996] and [Lindahl *et al.* 1998] to name a few. During the 1990s the method and several different implementations made their way into hardware design, domain specific software analysis, for example software of telecommunication systems, and real time system analysis. Several successful applications of model checking in more generic software contexts have emerged more recently, for example [Corbett *et al.* 2000], [Ball and Rajamani 2002], [Visser *et al.* 2000].

Given the model M of the system in some modelling language (timed automata, Promela, BIR, ...) and the design requirements ϕ in some logic (for example computation tree logic, CTL, or linear temporal logic, LTL or μ-calculus), it is possible to check, whether the model satisfies the requirements or not ($M \models \phi\,?$). If not, ($M \not\models \phi$), the method gives a trace leading to the error. Another benefit of the method is a high level of automation, meaning that once the model and requirements are there, it is possible to "push the button" and wait for the answer. No user intervention is required during the search.

There are several classifications of the desired properties that constitute the specification. The coarsest distinction is between safety and liveness properties. Safety properties generally characterise situations where something bad should never happen. Liveness properties on the other hand state

that something desirable should happen infinitely often. The latter property is hard to check in the general case, as the term "infinitely often" does not specify maximum delays between occurrences of desirable events. Therefore in practice the desired behaviour of the system is specified in terms bounded liveness properties, where the "boundedness" is expressed in terms of maximum delays between the occurrences of good events. This quite coarse classification of specification properties has been refined by Dwyer *et al.* [1999] by analysing a number of textual specifications of actual systems. The classification divides properties into occurrence (absence, universality, existence, bounded existence) and order (precedence, response, chain precedence and chain response) properties.

There is a broad class of properties (including safety and bounded liveness properties) which can be stated in terms of reachability [Aceto *et al.* 2003], for example, whether a particular configuration of variables in a program on some line can be reached or not. In this paper we apply the reachability approach for establishing the existence of a schedule for the system and for establishing the correctness of the resulting schedule. In this paper we apply model checking for two distinct purposes—scheduling and verification. In the case of scheduling we set up the problem in such a way that a positive answer to the reachability check (a path) gives us a schedule.

After we have found a recurring cycle in the system, we build a system that we verify to behave expectedly.

1.2 Choice of the Model Checking Tool

As was mentioned previously, there are several different model checkers available. We chose Uppaal [Amnell *et al.* 2001] because we found it convenient to model the current case study using Uppaal (timed) automata formalism and we could leverage the bit-state hashing (also known as hash compaction) symbolic state space representation feature and uniformly priced timed automata extension of the tool in the process. The latter extension is available in a recently released version of extended Uppaal — Uppaal CORA [Behrmann 2005].

1.2.1 Bit-State Hashing (Hash Compaction)

Hash compaction, first introduced by Wolper and Leroy [1993] and extended by Stern and Dill [1995] is also referred to as bit-state hashing by Holzmann [1998]. The idea of the method is reduce the memory consumption of reachability queries. The space reduction is achieved by storing one bit per state in a hash table which is addressed by a hash value calculated from the state itself. The method of symbolic state space representation is sound, meaning that when a state is reached symbolically, it is reachable in the current model. On the other hand, the method is incomplete, because it is possible that different states might have the same hash value or hash collision might occur and thus the state that is reached later is considered visited. Thus,

some parts of the state space might remain unexplored. The implementation details of bit-state hashing in Uppaal are described by Bengtsson [2002].

1.3 Outline

The rest of the paper is organised as follows. Section 2 gives an overview of related work. Section 3 introduces the radar memory interface board case study. Further, in Section 4 we describe a set of abstractions to customise the model to target the specific problem at hand. In Section 5 we give an overview about steps in synthesising the arbiter for the shared memory bus and about verification the resultant system containing the synthesised arbiter.

2. Related Work

This case study has previously been analysed by Weiss [2002] and by Sasnauskaite and Mikučionis [2002]. The solution described in the current paper differs from the former solution solution described in the following aspects:

- The solution presented by Weiss [2002] involves using SMV [McMillan 1999]. The current solution uses Uppaal [Pettersson and Larsen. 2000] as the model checker;
- In [Weiss 2002] the schedule for the example is reached using a parameterized model as model checking the full system was infeasible due to state space explosion. In the current approach, the Uppaal model of the component system contains abstractions that enable the schedule to be synthesized for the relevant aspect of the whole system.

Sasnauskaite and Mikučionis [2002] solve the verification part of the problem, but have trouble with optimising buffer sizes. They do not consider arbiter synthesis.

Uppaal has been previously applied for batch plant scheduling by Hune *et al.* [2001]. The approach presented therein is similar to the work presented here in the sense that they use model checking for finding a valid schedule. In addition, they leverage bit-state hashing to reduce memory consumtion of the model checking task. Our work differs from the latter in that we model the system as a synchronous system and the state is represented in terms of integers. We look for a suitable ordering of these states. The application domain is also different scheduling in the application domain (job shop scheduling versus hardware analysis) and in the modelling approach. We model the state of the target system in terms of integer variables which are updated on one transition by a nontrivial update.

Goel and Lee [2000] present a case study based on IBM CoreConnect$^{\text{TM}}$ processor local bus arbiter core. The case study presented therein is similar to current problem. The authors of the paper call for potential solutions for analysing the arbiter core.

Amnell *et al.* [2002] present a way to synthesise code for LEGO RCX bricks. The approach has more emphasis on the timed aspect of target systems.

3. Memory Arbiter Case Study

As was mentioned above, we use a case study from the IST AMETIST project by Behrmann *et al.* [2002]. The case study was provided to the project by Terma who is producing radar sensors mainly used for traffic control in ports and airports and for coastal surveillance. Some configurations of their radar sensor systems employ a technique known as *frequency diversity*. In this mode, two subsequent pulses that differ slightly in frequency are emitted right after each other from the antenna. As usually, the echo (in this case of two signals) is received, but due to the characteristics of the antenna, the signals got propagated in slightly different directions, and therefore the two simultaneously received signals do not correspond to exactly the same direction. To align the signals, one of the signals has to be delayed. This approach has two immediate benefits: it increases the output power of the radar for the purpose of better range and more reliable signal as two pulses are emitted and provides *time diversity*, i.e., makes it possible to compare the echoes of two signals from one direction at two slightly different subsequent moments for distinguishing, for example, a big wave from a small boat.

To remove noise, another technique, known as *sweep integration*, is used. The idea is to integrate the signal at the same direction from multiple sweeps. In the case of frequency diversity, sweep integration is performed on both return signals before the signals are combined.

In total, the signal processing board serves four purposes: sweep integration, frequency diversity combination, noise cancellation and a kind of differentiation (high pass filtering). The board uses dynamic synchronous RAM (SDRAM) for intermediate storage of the two input streams and their processed counterparts.

The signal processing board consists of signal processing units, SDRAM connected by a shared memory bus interfaced by 9 FIFO (First In First Out) buffers and governed by an arbiter which is reponsible for setting up communication between memory and one FIFO at a time so that none of the buffers is neither starved nor overflowed. A block diagram of the system is presented in Fig. 1 (Behrmann *et al.* [2002]).

Fig. 2 represents the internal structure of the 9 FIFO buffers. The size of the buffers range from 512 bytes to 2048 bytes (2KBytes) and they are implemented as ring buffers. The buffers that mediate data streams *to* SDRAM are called *input buffers* and, that mediate data streams *from* SDRAM are called *output buffers*.

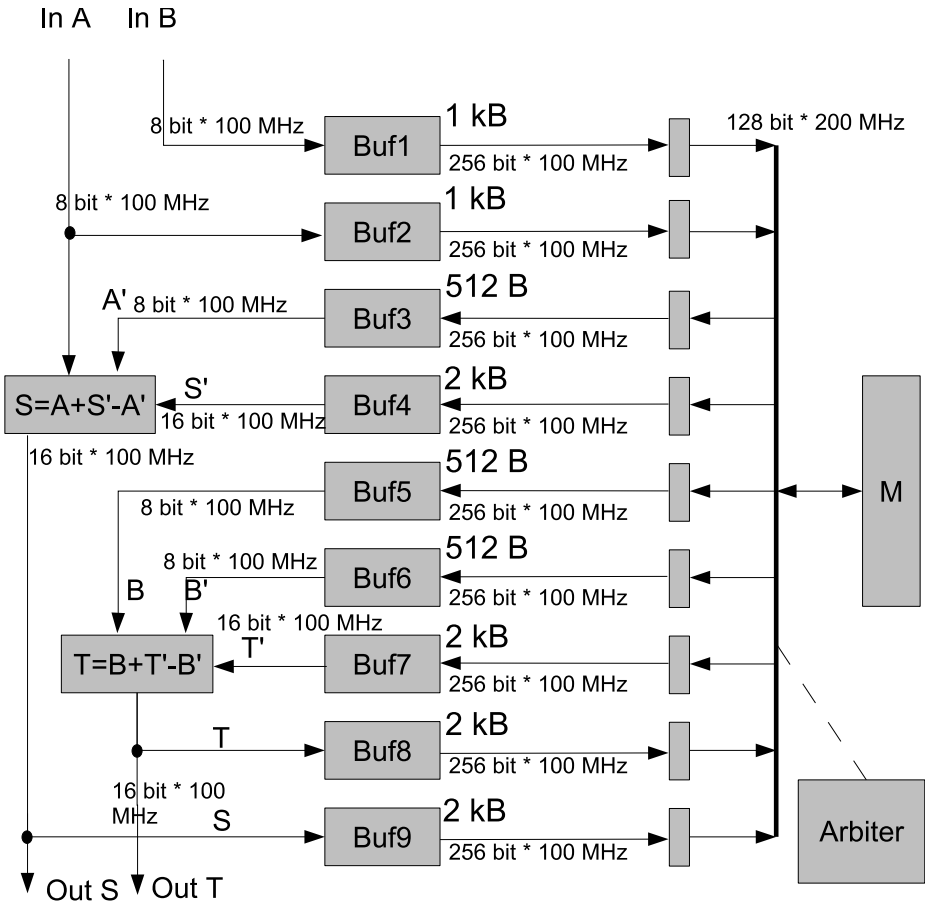

Fig. 1: Radar memory interface board

3.1 Observations of the system

Observation 1. The smallest quantity of data that can be moved in the system is 1 byte. For example inputs A and B in Fig. 1.

Observation 2. Data is written at fixed rate in 1 byte or 2 byte quantities (***a*** on Fig. 2). It is assumed that uninterrupted data flow of 1 byte at the frequency of 100 MHz is fed into the inputs A and B. Equivalently, it is assumed that outputs S and T can always be written to at the rate of 2 bytes at 100 MHz.

Observation 3. Data is read from the ring buffer into the register in 4 byte quantities (***b***, Fig. 2) whenever the two registers are not full and there are at least 4 bytes in the buffer at the beginning of the system clock cycle.

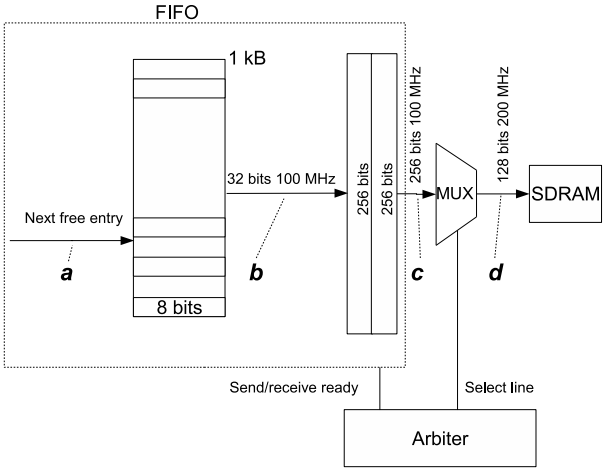

Fig. 2: The structure of a FIFO buffer

The duration of this transfer is one clock cycle. If there is less data in the buffer or the registers are full, no data is transferred. (This works vice versa in the case of output buffers. Data is written from the register to the buffer whenever the register is not empty and there is at least 4 bytes worth of space in the ring buffer).

Observation 4. On the memory bus (***c***, Fig. 2) data is always transferred when the register is full, i.e. in quantities of 512 bits = 64 bytes. (In the case of output buffers, data is transferred whenever the register is empty).

Observation 5. There is a multiplexer on the bus from the register to SDRAM as the memory bus runs in double data rate (DDR) mode and, is 128 bits wide but the register outputs 256 bits at 100 MHz. The DDR mode is achieved by transfering data on both the rising and the descending edge of a clock cycle and the multiplexer.

Observation 6. Whenever an output register is full, the *send* signal is set. Whenever an input register is empty, the *receive* signal is set.

3.2 Arbiter

The arbiter is responsible for setting up connections between SDRAM and registers that are ready for communication. The arbiter, that is presented in the case study, checks the *send/receive* signals of the buffers in a round-robin way. When a buffer is ready, a *select line* signal is issued by the arbiter.

Observation 7. Whenever an output register is full, the *send* signal is set (Fig. 2). Whenever an input register is empty, the *receive* signal is set.

Observation 8. In this case study calculation of the memory addresses for memory communication is not considered.

Observation 9. Transferring 64 bytes between the registers and SDRAM takes two additional system clock cycles, making it a total of 4 cycles.

Observation 10. SDRAM needs to be refreshed every 15625 ns. The refresh takes 100 ns (10 system clock cycles).

The aim of the original case study was to verify that the behaviour of the round-robin scheduling algorithm is correct. A further step would be to synthesise a scheduler for a set of buffers.

We modify the original goal slightly and seek solution to the following problems:

Problem 1. Check that no input buffers are full at the beginning of a clock cycle.

Problem 2. Check that no output buffers are empty at the beginning of a clock cycle.

Problem 3. Check that the system does not deadlock.

Problem 4. Synthesise an arbiter for the memory bus that guarantees that the above properties are always satisfied.

Problem 5. Find the smallest possible buffer values for the arbiter synthesised.

Problem 6. Check that the schedule guarantees that no data transfers are interrupted by a memory refresh.

4. Construction of the Abstract Model

In this section we summarise the decisions taken during the process of modelling the memory interface board. We are interested in one specific aspect of behaviour of the board, namely the *control behaviour of the memory arbiter*, and thus we disregard all detail that we manage to classify as not directly relevant. We model the system in terms of Uppaal modelling language [Bengtsson and Yi 2004], which corresponds to finite state automata extended with integer variables and clocks.

The most significant observation in approaching the case study is that *the system is fully synchronous meaning that all events are aligned to the system clock ticks*. Memory refresh, as specified by Observation 10, is the only exception to this rule. Keeping the synchronicity in mind enables us to reduce the number of intermediate states that should be distinguished in automatic analysis.

TABLE I: Data rates of the data streams. (A, ..., T' denote corresponding streams in Fig. 1. ***b*** denotes the corresponding stream in Fig. 2)

Data stream	A	A'	B	B'	S	S'	T	T'	***b***
Bus width (bytes)	1	1	1	1	2	2	2	2	4
Frequency (MHz)	100	100	100	100	100	100	100	100	100
Abstract bus width (bytes)	4	4	4	4	8	8	8	8	32
Abstrtract frequency (MHz)	25	25	25	25	25	25	25	25	25
Rate (MByte/s)	100	100	100	100	200	200	200	200	400

The first step in the current modelling approach is to choose the aspect of the system to focus on. We assume that the system is modelled in terms of some other (more or less formal) language, for example some block diagram language as in Fig. 1. If the system is sufficiently specified it is possible for the engineer to point to some part of the system and ask for assistance there. We assume that the engineer pointed to the shared memory bus and asked to remove nondeterminism from the model or, in other words, synthesise an arbiter.

4.1 Memory bus

As said, we pay special attention to the memory bus as the arbiter of the memory bus is what we focus on in this case study. We observe closely how the other components of the system interact with the memory bus. The properties of the memory bus determine the parameters of our model, such as, for example, time and data granularity.

Observation 9 refers to that data is transferred in bursts along the memory bus and that it takes 4 system clock cycles per burst. Thus we align our abstract system clock to the bursts on the memory bus.

Let us consider the main characteristics of the data streams, when dividing the system clock frequency by 4. The properties of the data streams are summarised in Table I. As the data rates should stay constant, we abstract the busses by widening them proportionally to the factor by which we reduced the clock frequency. The effect of this procedure will be discussed in detail below.

Now let us look at how the implementation details of the memory bus affect this approach. The data transfer from a register to memory and vice versa is performed on a 16 byte wide 200 MHz bus in the quantities of 64 bytes (Observation 4). As mentioned above, it takes 4 clock cycles to complete the transfer, so the memory-buffer data rate can be considered to be $64/4 = 16$ bytes/cycle $= 1600$ Mbytes/sec.

It is easy to estimate the solvability of the task in this case by the following simple sanity check:

$$throughput_{mb} - refresh \geq \sum_{streams} throughput_{stream}, \tag{1}$$

where $throughput_{mb}$ is the (abstract) data rate of the memory bus (1600 Mbytes/sec), *refresh* is bandwidth lost by staying in refresh state (100 ns / 15625 ns $\times throughput_{mb}$), and the right hand side is the sum of the throughputs of the data streams.

We assume that there is enough stream data stored in the SDRAM, so that we can always initiate a burst from some memory location to an output register when the latter is empty and vice versa for the input register. This assumption allows us to concentrate on the data levels of the buffers and registers and not to model the double data rate behaviour of the dynamic memory (bus ***d***, Fig. 2).

We will model the behaviour of the system in terms of data levels in the buffers. Under the above assumption we do not need to model levels in the memory. We model the data levels as values of integer variables in Uppaal automata.

4.2 Communication between Inputs, Outputs, and Buffers

We now turn to how the buffers and signal processing units behave from the point of view of the memory bus.

We observe that all of the input-adder, input-buffer, adder-output, adder-buffer, buffer-adder (Fig. 1) communication occurs in constant streams. We can model a constant flow by adding or subtracting a constant value to the integer variable representing a particular buffer at every (coarse) clock tick. Thus we can omit modelling the adders altogether.

Let us now have a look at how the interconnection of buffers and registers behaves. For example, take data transfer from buffer to (register Fig. 2). The width of the buffer to register bus is 4 bytes. In the abstract clock cycles (four concrete cycles) this amounts to 16 bytes. If the data level in the register is less than the allowed maximum of 64 bytes, data is written from the buffer to the register. Such analysis is repeated for all buffer-register pairs.

Fig. 4.2 gives an overview of the effect of the widening of the data paths that happens due to the coarsened clock. In the case of an input buffer, the buffer is considered empty until it is possible to read a whole coarse unit of data from it. In the case of output buffers, the buffer must have space for a whole coarse unit of data before data can be written to it.

We are not interested in the interleavings between filling/emptying independent buffers and registers. Therefore all the simultaneous interleavings are discarded by performing the updates of all registers and buffers as an update of a single transition of the automata model.

4.3 Assembling the Model

The model is divided into three automata:

- An automaton which is responsible for updating the states of buffers and registers. The automaton is called "Buffers".

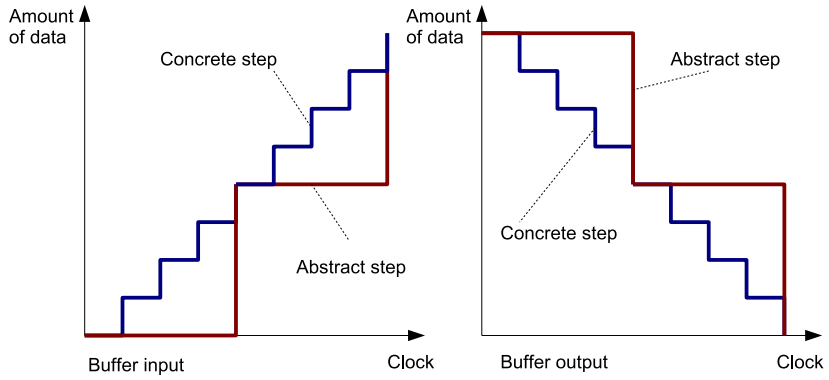

Fig. 3: Abstractions of the data flow

- An automaton that simulates the behaviour of memory (its need for refreshes). The automaton is called "Memory".
- An automaton for generating clock ticks. It is called "Clock".

In the next section we describe how the synthesis and verification problems are approached and what additional modifications are needed to the model described thus far.

5. Arbiter Synthesis and Verification

For the purpose of arbiter synthesis we create a conservative model of the system. By conservative we mean that we allow only valid behaviours of the system. We explicitly restrict the starvation and overflow of any of the buffers by introducing relevant guards.

The model in Fig. 5 has the following characteristics:

- The state of the buffers and registers is represented by integers.
- To model synchronicity, all variables representing buffers and registers are updated on one transition discarding a great deal of interleavings that are irrelevant in this context. There is a separate variable representing the behaviour of the arbiter that controls which register can access memory at a time.
- The granularity of time ticks is aligned with the duration of a transfer on the shared bus. The buffers and registers are updated by relevant multiples of bytes at the beginning of each such abstract tick.

For each buffer-to-memory and memory-to-buffer communication there is a transition in the Buffers automaton Fig. 5 that sets up relevant communication in the next clock cycle. In addition, there is a transition for memory update and idling.

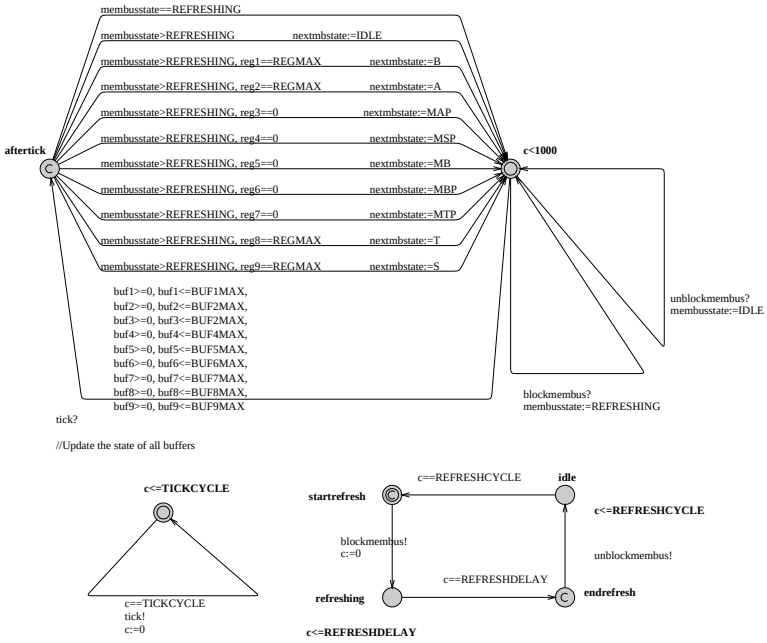

Fig. 4: The Uppaal model for schedule synthesis

The update of the states of the buffers and registers is performed by an update presented in Fig. 5. The semantics is that the update is taken at the end of the clock cycle, to make sure that no memory refresh would invalidate a data burst.

Memory refreshes are triggered by the Memory automaton.

The update of the buffers and registers is presented according to the syntax of Uppaal version 3.4.

5.1 Schedule synthesis

The current solution builds on the Uppaal model checker and its bit-state hashing implementation of symbolic representation of state space. Bit-state hashing is a memory consumption reduction technique is applied in finding schedules. A clock variable that is never reset and is checked in an invariant in one of the states of the Buffers automaton is used for bounding the depth of the search.

- The schedule is found using the $E\Diamond$ query given in Fig. 6. The amounts of data specified in the query are set to half the capacity of each buffer (we have reduced the buffer sizes by a factor of two)
- The scheduling task consumes 60 MB of RAM and takes 50 seconds with the hash table size of 10^7 bytes on a 2.4 GHz P4 processor (with

```
//Update the state of all buffers
//first we need to set up help variables:
// (for input buffers)
buf1help:=((reg1<=(REGMAX-BUF1OUT))&&(buf1>=BUF1OUT)?1:0),
// (for output buffers)
buf3help:=((reg3>=BUF3IN)?1:0),
// (for memory)
// next memory transaction:
// input buffers
reg1idle:=((nextmbstate==1)&&(reg1==REGMAX)&&(membusstate>REFRESHING)?0:1),
// output buffers
reg3idle:=((nextmbstate==3)&&(reg3==0)&&(membusstate>REFRESHING)?0:1),
//
//
//input buffers
//
buf1:=(buf1help?buf1+BUF1IN-(reg1idle*BUF1OUT):buf1+BUF1IN),
reg1:=(buf1help?reg1+(reg1idle*BUF1OUT):reg1),
//
//
// output buffers
//
buf3:=(buf3help?buf3-BUF3OUT+(reg3idle*BUF3IN):buf3-BUF3OUT),
reg3:=(buf3help?reg3-(reg3idle*BUF3IN):reg3),
//
// Memory
//
membusstate:=(membusstate>REFRESHING?0:membusstate),
//
membusstate:=(reg1idle==0&&membusstate>REFRESHING?1:membusstate),
membusstate:=(reg3idle==0&&membusstate>REFRESHING?3:membusstate),
memB:=(reg1idle==0&&membusstate==1?memB+REGMAX:memB),
memA:=(reg3idle==0&&membusstate==3?memA-REGMAX:memA),
//trasfer to and from registers
reg1:=(reg1idle==0&&membusstate==1?0:reg1),
reg3:=(reg3idle==0&&membusstate==3?REGMAX:reg3),
//
// clear help variables
reg1idle:=1,
reg3idle:=1,
//clean up:
buf1help:=0,
buf3help:=0,
initcomplete:=(membusstate>REFRESHING?1:initcomplete)
```

Fig. 5: The update on the transition that updates the states of registers and buffers (the actual updates are only shown for one input and one buffer)

512 kB of cache). It is possible to reduce the hash table size (a solution was found with the hashtable of size 10^5 bytes using 3.5 MB of memory and less than 1 second of cpu time), but it requires some experimentation to find a suitable size.

- A solution to the scheduling task with 1000 ns time bound and without the use of bit-state hashing consumed 3 GB of memory (the maximum that a process can be allocated on a 32 bit architecture in Linux).

$E\Box$ queries cannot be used in this approach because the specified reachability property does not hold along the path to the desired state. Instead, we have incorporated the path property into the guards of the synthesis model. This reduces the reachable state space.

The schedule is produced as a sequence of memory bus states.

5.2 Buffer Memory Minimization

Uppaal CORA [Behrmann 2005] is a tool that contains two different extensions to the Uppaal timed automata formalism. One extension is called

```
E<> Buffers.buf3+Buffers.reg3==256 and
    Buffers.buf5+Buffers.reg5==256 and
    Buffers.buf6+Buffers.reg6==256 and
    Buffers.buf1+Buffers.reg1==128 and
    Buffers.buf2+Buffers.reg2==128 and
    Buffers.buf4+Buffers.reg4==512 and
    Buffers.buf7+Buffers.reg7==512 and
    Buffers.buf8+Buffers.reg8==512 and
    Buffers.buf9+Buffers.reg9==512 and
    Buffers.initcomplete==1
```

Fig. 6: Specification of the model in terms of Uniformly Priced Timed Automata (modifications to the original model)

Linearly Priced Timed Automata (LPTA) and the other Uniformly Priced Timed Automata (UPTA) [Behrmann and Fehnker 2001]. In this example we make use of the UPTA extension by specifying a variable `cost` that is increased monotonously according to the increase in the range of buffer memory used. The cost value is the sum of differences of the maxima and minima of the amount of data in buffers. The appropriate modifications to the model that are shown in Fig. 7.

```
// Input buffers
//
buf1=(buf1help?buf1+BUF1IN-(reg1idle*BUF1OUT):buf1+BUF1IN),
cost+=(buf1>buf1max?1:0),
buf1max=(buf1>buf1max?buf1:buf1max),
cost+=(buf1<buf1min?1:0),
buf1min=(buf1<buf1min?buf1:buf1min),
reg1=(buf1help?reg1+(reg1idle*BUF1OUT):reg1),

// Output buffers
//
buf3=(buf3help?buf3-BUF3OUT+(reg3idle*BUF3IN):buf3-BUF3OUT),
cost+=(buf3>buf3max?1:0),
buf3max=(buf3>buf3max?buf3:buf3max),
cost+=(buf3<buf3min?1:0),
buf3min=(buf3<buf3min?buf3:buf3min),
reg3=(buf3help?reg3-(reg3idle*BUF3IN):reg3),
```

Fig. 7: Specification of the model in terms of Uniformly Priced Timed Automata (modifications to the original model)

The difference between the behaviours of resultant schedules in terms of buffer sizes is given on Fig. 5.2. This demonstrates that the buffer sizes can be reduced drastically even when looking at the problem from an abstract point of view. With the reduced buffer sizes the model can be analysed further.

5.3 System Verification

In this section we created a schedule for the memory arbiter that should satisfy the desired properties presented in the case study. To check that the schedule is valid we create a new model of the system where we embed the synthesised scheduler. The resultant model is in Figure 9. The update on the lower transition of the Buffers automaton is the same as in Figure 5. It appears that this is a deterministic automaton satisfying the properties

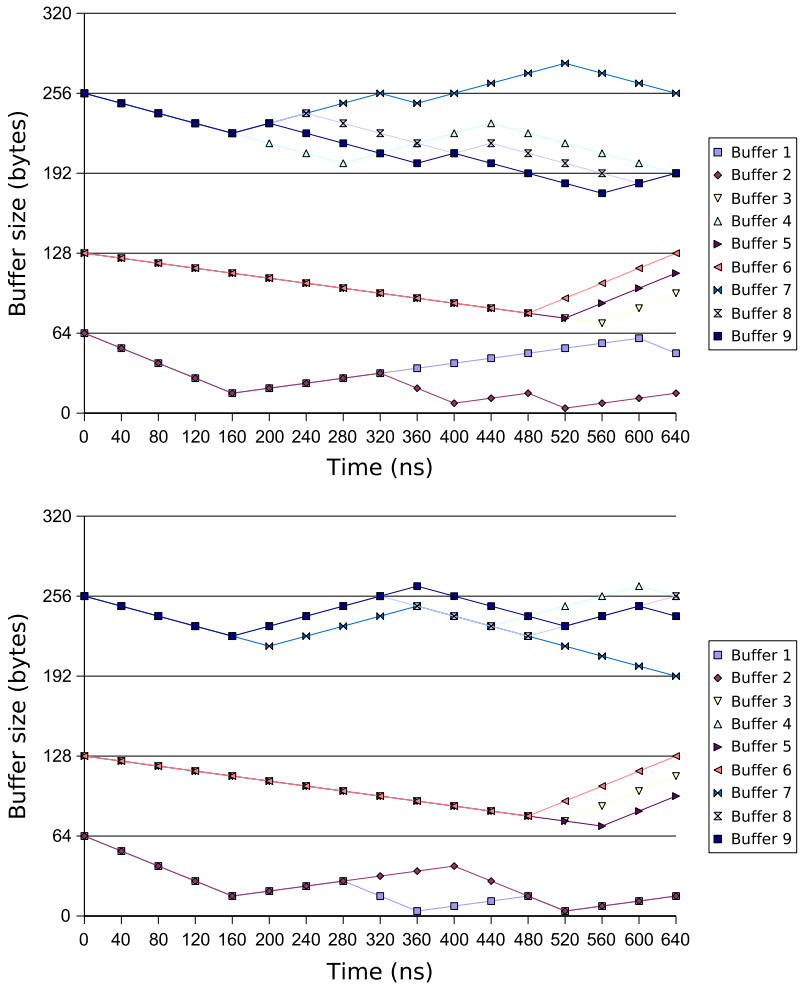

Fig. 8: Data fluctuations in buffers during a synthesised cycle. The above result is gained with bounded depth first search and uses a total buffer range of 528 bytes. The lower one is obtained by using uniformly priced timed automata extensions to the model. The total buffer range used is 452 bytes.

of never deadlocking and never starving nor overflowing any of the buffers under the assumption that memory refresh can be scheduled. The memory automaton is omitted from this verification model as refreshes are scheduled by the arbiter. It is necessary to construct a more complicated arbiter if memory refreshes are for some reason required to be allowed to occur periodically out of sync of the arbiter. In the current case such deterministic automaton is desirable because if the memory burst set-up needs to precede the actual burst by some short interval, it can be directy integrated into the scheduler.

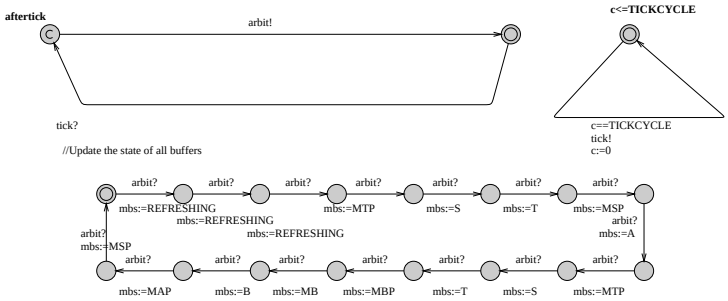

Fig. 9: The Uppaal model for verification of the system

5.4 Perspective

It is suggested by Lee [2003] that the integration of verification tools could be leveraged in modelling and simulation environments such as, for example, PtolemyII. Taking the underlying computational model of the target system into account could facilitate verification and synthesis as it would be easier to partition the system into smaller components or use the computational model as the basis of abstraction.

Edwards and Lee [2003] present an interesting semantics for synchronous block diagram language that enables to combine together components described in different synchronous formalisms. By combining such models with finite state machines, FSMs, would yield a heterogenous system. It seems to be appealing to research how (guided) synthesis by model checking and verification could be integrated into the setting. We believe, that procedures outlined in this paper are useful steps toward such integration.

6. Conclusion

In this paper we analysed the radar memory interface card case study of the IST AMETIST project. We presented a way to synthesise a memory arbiter for the system, to minimise the amount of static RAM used for buffering the streams and to verify that the the synthesised arbiter does not deadlock and never starves nor overflows any of the intermediate buffers. We presented the synthesis and verification tasks as two different problems of logic model checking. We designed the model for the synthesis task very carefully to reduce the potential state space that needs to be searched in the process. To achieve that, unwanted behaviour was restricted on the guard level of the model by disabling transitions which would starve or overflow the intermediate buffers. We used the assumption of synchronicity in modelling. This enabled us to make a number of simplifications in the model. As we used bounded (in terms of search depth) search for arbiter synthesis, it was necessary to verify whether it behaved expectedly under all circumstances.

For this purpose we constructed another which differs from the first model by containing an arbiter but not the restrictions mentioned in the synthesis model case. The model was verified against the buffer starvation/overflow invariant and was checked that it is deadlock free.

Synthesis tasks are generally computationally more challengeing than verification tasks. The success of the synthesis task is largely concealed in the rather simple abstract model of the system. The other key factor is the application of a sound but incomplete method for space saving—bit-state hashing, where each visited state is represented by one bit in the hash table in a location determined by the hash of the state. We showed that it is possible to synthesise a reasonable arbiter using this approach. Additionally, we augmented the model with cost variables (taking into account the range of buffers used) and applied the guided version of Uppaal—Uppaal-Cora to the synthesis task. This resulted in an optimal schedule in terms buffer memory on the abstraction level of the model.

7. Acknowledgements

This work was supported by the European Commission via a Marie Curie Fellowship, by the Estonian Information Technology Foundation via Tiigriülikool programme and by grant No 5775 of the Estonian Science Foundation.

References

Aceto, Luca, Bouyer, Patricia, Burgueño, Augusto, and Larsen, Kim G. 2003. The power of reachability testing for timed automata. *Theor. Comput. Sci. 300*, 1-3, 411–475.

AMETIST. 2002–2005. The EU Information Society Technologies project IST-2001-35304: "Advanced Methods for Timed Systems". http://ametist.cs.utwente.nl.

Amnell, Tobias, Behrmann, Gerd, Bengtsson, Johan, D'Argenio, Pedro R., David, Alexandre, Fehnker, Ansgar, Hune, Thomas, Jeannet, Bertrand, Larsen, Kim G., Möller, M. Oliver, Pettersson, Paul, Weise, Carsten, and Yi, Wang. 2001. Uppaal - Now, Next, and Future. In *Modelling and Verification of Parallel Processes*, Number 2067 in Lecture Notes in Computer Science Tutorial. Springer–Verlag, 100–125.

Amnell, Tobias, Fersman, Elena, Pettersson, Paul, Yi, Wang, and Sun, Hongyan. 2002. Code synthesis for timed automata. *Nordic J. of Computing 9*, 4, 269–300.

Ball, Thomas and Rajamani, Sriram K. 2002. The SLAM project: debugging system software via static analysis. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 1–3.

Behrmann, Gerd. 2005. Uppaal CORA. http://www.cs.aau.dk/~behrmann/cora/.

Behrmann, Gerd, Bernicot, Sébastien, Hune, Thomas, Larsen, Kim G., Lecamp, Sylvain, and Skou, Arne. 2002. Case study 2: Memory interface for radar system. Deliverable to the IST AMETIST project No. IST-2001-35304.

Behrmann, Gerd and Fehnker, Ansgar. 2001. Efficient Guiding Towards Cost-Optimality in UPPAAL. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 174–188.

Bengtsson, Johan. 2002. *Clocks, DBMs and States in Timed Systems*. Uppsala: Acta Universitatis Upsaliensis, Uppsala, Sweden.

Bengtsson, Johan, Griffioen, W.O. David, Kristoffersen, Kåre J., Larsen, Kim G., Larsson, Fredrik, Pettersson, Paul, and Yi, Wang. 1996. Verification of an Audio Protocol with Bus Collision Using Uppaal. Number 1102 in Lecture Notes in Computer Science. Springer–Verlag, 244–256.

Bengtsson, Johan and Yi, Wang. 2004. Timed Automata: Semantics, Algorithms and Tools. In *In Lecture Notes on Concurrency and Petri Nets*, Lecture Notes in Computer Science vol 3098. Springer–Verlag.

Clarke, Edmund M. and Emerson, E. Allen. 1982. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logic of Programs, Workshop*. Springer-Verlag, 52–71.

Clarke, Edmund M., Grumberg, Orna, Hiraishi, Hiromi, Jha, Somesh, Long, David E., McMillan, Kenneth L., and Ness, Linda A. 1993. Verification of the Futurebus+ Cache Coherence Protocol. In *CHDL '93: Proceedings of the 11th IFIP WG10.2 International Conference sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC on Computer Hardware Description Languages and their Applications*. North-Holland, 15–30.

Corbett, James C., Dwyer, Matthew B., Hatcliff, John, Laubach, Shawn, Pasareanu, Corina S., Robby, and Zheng, Hongjun. 2000. Bandera: extracting finite-state models from Java source code. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*. ACM Press, 439–448.

Dwyer, Matthew B., Avrunin, George S., and Corbett, James C. 1999. Patterns in property specifications for finite-state verification. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*. IEEE Computer Society Press, 411–420.

Edwards, Stephen A. and Lee, Edward A. 2003. The semantics and execution of a synchronous block-diagram language. *Sci. Comput. Program. 48*, 1, 21–42.

Goel, Amit and Lee, William R. 2000. Formal verification of an IBM CoreConnect processor local bus arbiter core. In *DAC '00: Proceedings of the 37th conference on Design automation*. ACM Press, 196–200.

Holzmann, Gerard J. 1998. An Analysis of Bitstate Hashing. *Form. Methods Syst. Des. 13*, 3, 289–307.

Hune, Thomas, Larsen, Kim G., and Pettersson, Paul. 2001. Guided Synthesis of Control Programs using Uppaal. *Nordic Journal of Computing 8*, 1, 43–64.

Lee, Edward A. 2003. Overview of the Ptolemy Project. Tech. Report UCB/ERL M03/25.

Lindahl, Magnus, Pettersson, Paul, and Yi, Wang. 1998. Formal Design and Analysis of a Gear-Box Controller. In *Proc. of the 4*th *Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Number 1384 in Lecture Notes in Computer Science. Springer–Verlag, 281–297.

McMillan, K. L. 1999. The SMV Language.

Pettersson, Paul and Larsen., Kim G. 2000. Uppaal2k. *Bulletin of the European Association for Theoretical Computer Science 70*, (Feb.), 40–44.

Queille, Jean-Pierre and Sifakis, Joseph. 1982. Specification and verification of concurrent systems in CESAR. In *Proceedings of the 5th Colloquium on International Symposium on Programming*. Springer-Verlag, 337–351.

Sasnauskaite, Egle and Mikučionis, Marius. 2002. *Memory Interface Analysis using the Real-Time Model Checker UPPAAL*. Master's thesis, University of Aalborg.

Stern, Ulrich and Dill, David L. 1995. Improved probabilistic verification by hash compaction. In *CHARME '95: Proceedings of the IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer-Verlag, 206–224.

Visser, Willem, Havelund, Klaus, Brat, Guillaume, and Park, SeungJoon. 2000. Model Checking Programs. In *ASE '00: Proceedings of the The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*. IEEE Computer

Society, 3.

Weiss, Gera. 2002. Optimal Scheduler for a Memory Card. Research report, Weizmann.

Wolper, Pierre and Leroy, Dennis. 1993. Reliable Hashing Without Collision Detection. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV-93)*, Volume 7. Springer, Berlin, 59–70.