

# Finding Extremal Models of Discrete Duration Calculus formulae using Symbolic Search

Paritosh K. Pandya<sup>1</sup>

*School of Technology and Computer Science  
Tata Institute of Fundamental Research  
Homi Bhabha Road, Colaba  
Mumbai 400005, India*

---

## Abstract

QDDC is a logic for specifying quantitative timing aspects of synchronous programs. Properties such as worst-case response time and latency (when known) can be specified elegantly in this logic and model checked. However, computing these values require finding by trial and error the least/greatest value of a parameter  $k$  making a formula  $D(k)$  valid for a program. In this paper, we discuss how an automata theoretic decision procedure for QDDC together with symbolic search for *shortest/longest* path can be used to *compute* the lengths of extremal (least/greatest length) models of a formula  $D$ . These techniques have been implemented into the DCVALID verifier for QDDC formulae. We illustrate the use of this technique by efficiently computing response and dead times of some synchronous bus arbiter circuits.

*Key words:* Symbolic model checking, Response time computation, Discrete duration calculus, Extremal models, SMV.

---

## 1 Introduction

For synchronous programs (e.g. clocked circuits), execution time is measured in terms of clock ticks, i.e. the notion of time is discrete. For many such programs, it is important to analyse quantitative timing properties such as response time and latency. Doing such quantitative analysis remains a challenging problem before the formal methods community.

Quantified Discrete-time Duration Calculus (QDDC) [12] is a highly expressive logic for specifying properties of finite sequences of states (behaviours). It is closely related to the Interval Temporal Logic of Moszkowski [11] and Duration Calculus of Zhou *et al* [16] (see [12,15,6] for their relationship.) It

---

<sup>1</sup> Email: [pandya@tifr.res.in](mailto:pandya@tifr.res.in)

provides novel interval based modalities for describing behaviours. For example, the following formula holds for a behaviour  $\sigma$  provided for all fragments  $\sigma'$  of  $\sigma$  which have (a)  $P$  true in the beginning, (b)  $Q$  true at the end, and (c) no occurrences of  $Q$  in between, the number of occurrences of states in  $\sigma'$  where  $R$  is true is at most 3.

$$\Box([P]^0 \frown [\lceil \neg Q \rceil \frown [Q]^0 \Rightarrow (\Sigma R \leq 3))$$

Here,  $\Box$  modality ranges over all fragments of a behaviour. Operator  $\frown$  is like concatenation (fusion) of behaviour fragments and  $\lceil \neg Q \rceil$  states invariance of  $\neg Q$  over the behaviour fragment. Finally,  $\Sigma R$  counts number of occurrences of  $R$  within a behaviour fragment. A precise definition of the syntax and semantics of QDDC is given in Section 2.

In spite of its high expressive power *QDDC* formulae can be model checked. An automata theoretic decision procedure allows converting a *QDDC* formula into a finite state automaton recognising precisely the models of the formula [12]. The automaton can be used as a synchronous observer for model checking the property of a synchronous program [8]. We have implemented this theory into a tool called DCVALID [12,13] which permits model checking *QDDC* properties of synchronous programs written in Esterel [2], Verilog and SMV [10] notations.

Quantified Discrete-time Duration Calculus, (*QDDC*), is a logic well suited to specifying quantitative timing properties of synchronous programs. It addresses a qualitatively different class of properties of synchronous programs from those considered earlier. Properties such as worst-case response time and latency (when known) can be specified elegantly in this logic and model checked. However, *computing* these values require finding by trial and error the least/greatest value of a parameter  $k$  making a formula  $D(k)$  valid for a program. Such a trial-error technique is inherently incomplete.

In the paper, we propose formulation of many interesting timing properties as lengths of extremal (shortest/longest) sub-execution of a system satisfying a property  $D$  written in the logic *QDDC*. By sub-execution we mean a finite (not necessarily initial) fragment of an execution. For example, response time can be formulated as the length of longest sub-execution during which *request*  $\wedge$   $\neg$ *acknowledgement* holds invariantly. Logic QDDC is well suited to specify complex timing requirements in this fashion. We call this approach *extremal model length* based specification.

In the paper, we show how an automata theoretic decision procedure for QDDC together with symbolic search for *shortest/longest* path can be used to *compute* the extremal model lengths. These techniques have been implemented into the DCVALID verifier for QDDC formulae. The implementation is built on top of the symbolic search routines for shortest/longest paths available in the NuSMV verifier.

We illustrate the use of our technique by computing response and dead times of some synchronous bus arbiter circuits using our tool DCVALID and

NuSMV, with some surprising results. It is our claim these properties are quite difficult to analyse by hand and a system designer’s intuition about them can be misleading. Hence, the availability of tools is crucial for the analysis such properties. In the paper, we also provide an experimental comparison of the efficiency of our extremal model length computation with traditional model checking.

The rest of the paper is organised as follows. A synchronous bus arbiter circuit model is presented in the next subsection. The logic QDDC and its model checking are briefly presented in section 2. The notion of extremal model lengths is defined in section 3. Section 4 presents the main technique used for symbolically computing extremal model lengths. Section 5 briefly describes the implementation of this technique into our tool DCVALID. It also gives the experimental results for the timing analysis of the bus arbiter circuits. The paper ends with a discussion.

### 1.1 Synchronous Bus Arbiter

**Example 1.1** A synchronous bus arbiter with  $n$  cells has request lines  $req_1, \dots, req_i, \dots, req_n$  and acknowledgement lines  $ack_1, \dots, ack_i, \dots, ack_n$ . At any clock cycle a subset of the request lines are high. It is the task of the arbiter to set at most one of the corresponding acknowledgement lines high. Preferably, the arbiter should be fair to all requests.

The bus arbiter circuit of Figure 1 (called MacArbV0) was analysed by McMillan [10] using the pioneering SMV verifier based on symbolic model checking<sup>2</sup>. A variant, MacArbV1, of McMillan’s arbiter is given in Figure 2. (The changes from the original arbiter are highlighted by dotted lines.) Both these arbiters have the property that at most one  $ack$  signal can occur at a time.  $\square$

**Example 1.2** We consider some quantitative timing properties of the arbiters.

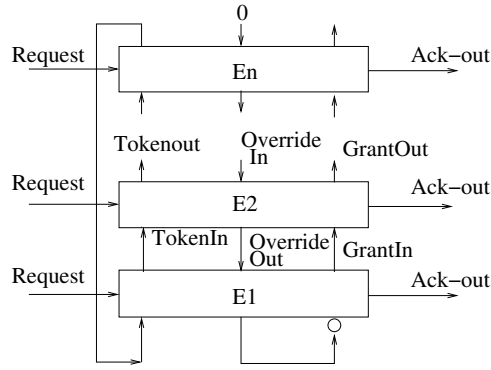
- *3-cycle response time*: The least number of cycles for which  $req_i$  must be held high continuously in the worst case to ensure 3 occurrences of  $ack_i$ .
- *Dead time*: The maximum possible number of consecutive *lost* cycles. A cycle is *lost* if at least one of the cells has its  $req$  high but all the cells have  $ack$  low, i.e.  $lostcycle \stackrel{\text{def}}{=} (\bigvee_i req_i) \wedge \neg(\bigvee_j ack_j)$ .  $\square$

## 2 Quantified Discrete-Time Duration Calculus (QDDC)

Let  $Pvar$  be a finite set of propositional variables representing some observable aspects of system state.  $VAL(Pvar) \stackrel{\text{def}}{=} Pvar \rightarrow \{0, 1\}$  be the set of

<sup>2</sup> The circuit elements are standard. The square box denotes a *D-latch* which delays the signal by one clock cycle.

### Cell Interconnection



### Cell Circuit

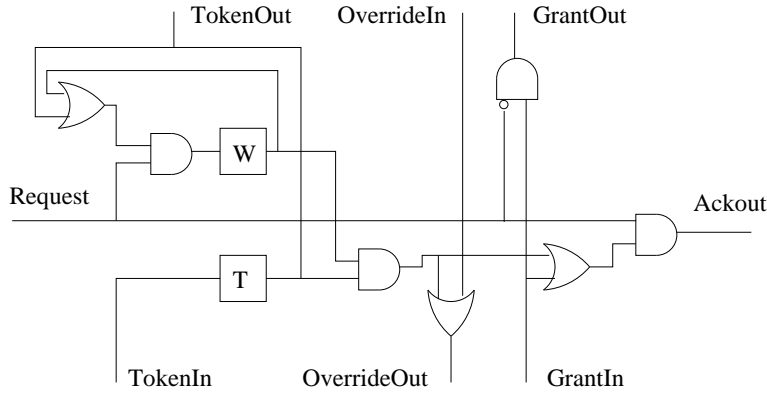


Fig. 1. McMillan's Arbiter: MacArbV0

### Modified Cell Circuit

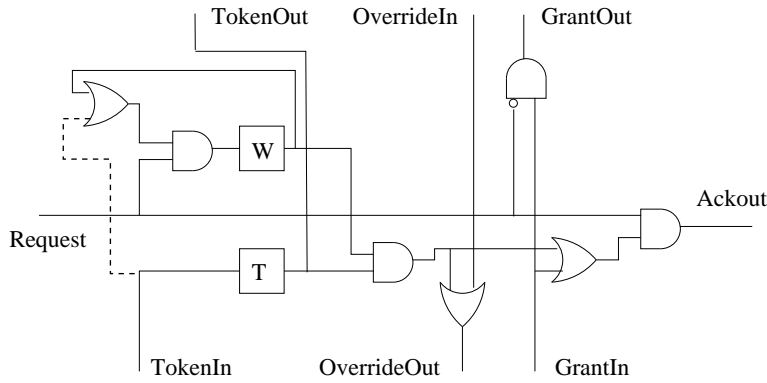


Fig. 2. A Variant of McMillan's Arbiter: MacArbV1

valuations assigning truth-value to each variable.

We shall identify behaviours with finite, nonempty sequences of valuations, i.e.  $VAL(Pvar)^+$ .

**Example 2.1** The following picture gives a behaviour over variables  $\{p, q\}$ . Each column vector gives a valuation, and the word is a sequence of such

column vectors.

p	1	0	1	1	0
q	0	0	0	0	1

The above word satisfies the property that  $p$  holds initially and  $q$  holds at the end but nowhere before that. QDDC is a logic for formalising such properties. Each formula specifies a set of such words.

Given a non-empty finite sequence of valuations  $\sigma \in VAL^+$ , we denote the satisfaction of a QDDC formula  $D$  over  $\sigma$  by  $\sigma \models D$

### Syntax of QDDC Formulae

Let  $Pvar$  be the set of propositional variables. Let  $p$  range over propositional variables,  $P, Q$  over propositions and  $D, D_1, D_2$  over QDDC formulae. Propositions are constructed from variables  $Pvar$  and constants 0, 1 (denote *true*, *false*) using boolean connectives  $\wedge, \neg$  etc. as usual.

The syntax of QDDC is as follows.

$$\begin{aligned} & [P]^0 \mid \llbracket P \rrbracket \mid D_1 \frown D_2 \mid D_1 \wedge D_2 \mid \neg D \mid \exists p.D \mid \\ & \eta \text{ op } c \mid \Sigma P \text{ op } c \quad \text{where } \text{op} \in \{<, \leq, =, \geq, >\} \end{aligned}$$

Let  $\sigma \in VAL(Pvar)^+$  be a behaviour. Let  $\#\sigma$  denote the length of  $\sigma$  and  $\sigma[i]$  the  $i$ 'th element. For example, if  $\sigma = \langle v_0, v_1, v_2 \rangle$  then  $\#\sigma = 3$  and  $\sigma[1] = v_1$ . Let  $dom(\sigma) = \{0, 1, \dots, \#\sigma - 1\}$  denote the set of positions within  $\sigma$ . The set of intervals in  $\sigma$  is given by  $Intv(\sigma) = \{[b, e] \in dom(\sigma)^2 \mid b \leq e\}$  where each interval  $[b, e]$  identifies the subsequence of  $\sigma$  between the positions  $b$  and  $e$ .

Let  $\sigma, i \models P$  denote that proposition  $P$  evaluates to true at position  $i$  in  $\sigma$ . We omit this obvious definition. We inductively define the satisfaction of QDDC formula  $D$  for behaviour  $\sigma$  and interval  $[b, e] \in Intv(\sigma)$  as follows.

$$\begin{aligned} \sigma, [b, e] \models [P]^0 & \quad \mathbf{iff} \quad b = e \text{ and } \sigma, b \models P \\ \sigma, [b, e] \models \llbracket P \rrbracket & \quad \mathbf{iff} \quad b < e \text{ and } \sigma, i \models P \text{ for all } i : b \leq i < e \\ \sigma, [b, e] \models \neg D & \quad \mathbf{iff} \quad \sigma, [b, e] \not\models D \\ \sigma, [b, e] \models D_1 \wedge D_2 & \quad \mathbf{iff} \quad \sigma, [b, e] \models D_1 \text{ and } \sigma, [b, e] \models D_2 \\ \sigma, [b, e] \models D_1 \frown D_2 & \quad \mathbf{iff} \quad \text{for some } m : b \leq m \leq e : \\ & \quad \sigma, [b, m] \models D_1 \text{ and } \sigma, [m, e] \models D_2 \end{aligned}$$

Entities  $\eta$  and  $\Sigma P$  are called *measurements*. Term  $\eta$  denotes the length of the interval whereas  $\Sigma P$  denotes the count of number of times  $P$  is true within the

interval  $[b, e]$  (we treat the interval as being left-closed right-open). Formally,

$$\begin{aligned} eval(\eta, \sigma, [b, e]) &\stackrel{\text{def}}{=} e - b \\ eval(\Sigma P, \sigma, [b, e]) &\stackrel{\text{def}}{=} \sum_{i=b}^{e-1} \left\{ \begin{array}{l} 1 \text{ if } \sigma, i \models P \\ 0 \text{ otherwise} \end{array} \right\} \end{aligned}$$

Let  $t$  range over measurements. Then,

$$\sigma, [b, e] \models t \text{ op } c \quad \text{iff} \quad eval(t, \sigma, [b, e]) \text{ op } c$$

Call a behaviour  $\sigma'$  to be  $p$ -variant of  $\sigma$  provided  $\#\sigma = \#\sigma'$  and for all  $i \in dom(\sigma)$  and for all  $q \neq p$ , we have  $\sigma(i)(q) = \sigma'(i)(q)$ . Then,

$$\sigma, [b, e] \models \exists p.D \quad \text{iff} \quad \sigma', [b, e] \models D \text{ for some } p\text{-variant } \sigma' \text{ of } \sigma$$

Finally,

$$\sigma \models D \quad \text{iff} \quad \sigma, [0, \#\sigma - 1] \models D$$

### Derived Constructs

We can also define some derived constructs. Boolean combinators  $\vee, \Rightarrow, \Leftrightarrow$  can be defined using  $\wedge, \neg$  as usual.

- $\lceil \rceil \stackrel{\text{def}}{=} \lceil 1 \rceil^0$  holds for point intervals of the form  $[b, b]$ .
- $\llbracket P \rrbracket \stackrel{\text{def}}{=} (\llbracket P \rrbracket \wedge \lceil P \rceil^0)$  states that proposition  $P$  holds invariantly over the extended closed interval  $[b, e]$  including the endpoint. Formula  $\llbracket P \rrbracket^+ \stackrel{\text{def}}{=} (\llbracket P \rrbracket \vee \lceil P \rceil^0)$  additionally also holds for point intervals where  $P$  is true.
- $\diamond D \stackrel{\text{def}}{=} true \wedge D \wedge true$  holds provided  $D$  holds for some subinterval.
- $\square D \stackrel{\text{def}}{=} \neg \diamond \neg D$  holds provided  $D$  holds for all subintervals.

### Decidability of QDDC

The following theorem characterises the sets of models of a QDDC formula. Let  $pvar(D)$  be the finite set of propositional variables occurring within a QDDC formula  $D$ . Let  $VAL(pvar) = pvar \rightarrow \{0, 1\}$  be the set of valuations over  $pvar$  as before.

**Theorem 2.2** *For every QDDC formula  $D$ , we can effectively construct a finite state automaton  $A(D)$  over the alphabet  $VAL(pvar(D))$  such that for all  $\sigma \in VAL(pvar(D))^*$ ,*

$$\sigma \models D \quad \text{iff} \quad \sigma \in L(A(D))$$

**Corollary 2.3** *Satisfiability (validity) of QDDC formulae is decidable.  $\square$*

### DCVALID

The reduction from formulae of QDDC to finite state automata as outlined in Theorem 2.2 has been implemented into a tool called DCVALID [12], which also checks for the validity of formulae as in Corollary 2.3. This tool is built

on top of MONA [9]. MONA is a sophisticated and efficient BDD-based implementation of the automata-theoretic decision procedure for monadic logic over finite words.

An associated tool, called CTLDC [13], translates the automaton into Esterel, SMV or Verilog module to give a synchronous observer [8] for the property. Using this, DCVALID can model check whether  $M \models D$  where  $M$  an Esterel, SMV or Verilog program and  $D$  is a *QDDC* formula [13].

**Example 2.4** [Arbiter Specification] We formalise the timing properties of the arbiters from Example 1.2 in QDDC.

- *3-cycle response time*: The minimum  $k$  such that following is *valid* for the arbiter.  $\square(\llbracket req_i \rrbracket \wedge \eta \geq k - 1 \Rightarrow \Sigma ack_i \geq 3)$
- *Dead time*: The minimum  $k$  such that the following is *valid* for the arbiter.  $\square(\llbracket lostcycle \rrbracket \Rightarrow \eta < k)$ . □

Note that traditional model checking can verify a property  $D(k)$  for a given constant  $k$ . In this paper, we propose some techniques which can *compute* the extremal values of  $k$ . Moreover, these techniques are experimentally shown to be effective in solving problems like the dead and response times of the arbiters. The techniques have been built into our model checking tool DCVALID.

### 3 Extremal Model Lengths

**Definition 3.1** A transition system  $M = (S, R, L, S_0)$  consists of a set of states  $S$ , a set of initial states  $S_0 \subseteq S$ , a transition relation  $R \subseteq S \times S$  and a labelling function  $L : S \rightarrow VAL(Pvar)$ . Here,  $Pvar$  is the set of observable propositions. Let  $M_1 \times M_2$  denote the *synchronous product* of transition systems  $M_1$  and  $M_2$ , as usual. It captures the parallel execution of the two transition system running in synchronous (lock-step) parallel fashion.

An execution of  $M$  is a (finite or infinite) sequence of states starting with an element of  $S_0$  where every (tuple consisting of) consecutive pair of states  $(s_i, s_{i+1}) \in R$ . A *behaviour* is a complete execution which is either infinite or ends in state which has no  $R$  successor. Let  $Beh(M)$  denote the set of behaviours of  $M$ . □

#### Notation

Let  $\alpha \in S^* \cup S^\omega$  be a finite or infinite sequence of states from  $S$ . Then,  $\alpha[i]$  denotes the  $i$ th element. Also,  $\alpha[i, j] = \alpha[i], \dots, \alpha[j]$  denotes the finite subsequence between positions  $i$  and  $j$ .

Let  $\omega$  denote set of natural numbers and  $\emptyset$  be the empty set. Let  $N \subseteq \omega$ . Then  $\max N$  denotes the least upper bound of  $N$  and  $\min N$  denotes the greatest lower bound of  $N$ . Some special cases of these functions are outlined below. Let  $\max \emptyset = 0$  and  $\max \omega = \infty$ . Let  $\min \emptyset = \infty$  and  $\min \omega = 0$ .

**Definition 3.2** [sub-executions] Let  $\alpha \in S^* \cup S^\omega$ . Then,

- $subseq(\alpha) \stackrel{\text{def}}{=} \{\alpha[b, e] \mid b, e \in dom(\alpha), b \leq e\}$ .
- $subexec(M) \stackrel{\text{def}}{=} \bigcup_{\alpha \in Beh(M)} subseq(\alpha)$   
The elements of  $subexec(M)$  will be called *sub-executions*. They denote finite fragments of the executions of  $M$ .
- Let  $\langle s_0, \dots, s_n \rangle \models_L D \stackrel{\text{def}}{=} \langle L(s_0), \dots, L(s_n) \rangle \models D$  denote that formula  $D$  holds for a sub-execution  $\langle s_0, \dots, s_n \rangle$ .

We now formalise the notion of lengths of extremal (i.e. longest/shortest) sub-executions of  $M$  which satisfy a QDDC formula  $D$ . Term  $MAXLEN(D, M)$  denotes the length (i.e. number of states) of longest sub-execution of  $M$  satisfying  $D$ . In case there are sub-executions of unbounded lengths satisfying  $D$ , then the term evaluates to  $\infty$ . If there are no sub-executions satisfying  $D$ , the term evaluates to 0. Term  $MINLEN(D, M)$  denotes the step length (i.e. number of edges) in the shortest sub-execution satisfying  $D$ . If there are no sub-executions satisfying  $D$ , the term evaluates to  $\infty$ .

**Definition 3.3** [Extremal sub-execution lengths]

- $MAXLEN(D, M) = \max \{(\#\sigma) \mid \sigma \in subexec(M) \text{ and } \sigma \models_L D\}$
- $MINLEN(D, M) = \min \{\#\sigma - 1 \mid \sigma \in subexec(M) \text{ and } \sigma \models_L D\}$

Many quantitative features of interest can be specified using the constructs  $MAXLEN$  and  $MINLEN$ .

**Example 3.4** For the arbiters of Example 1.1, we can elegantly formalise the response and dead-time (see Example 2.4) using  $MAXLEN$  as follows.

- **3-cycle response time** is given by  
 $MAXLEN(\llbracket req \rrbracket^+ \wedge (\Sigma ack = 2 \frown [ack]^0), \text{Arbiter})$   
 The 3-cycle response time is given by the length of longest sub-execution of *Arbiter* where *req* is invariantly true, where there are two occurrences of *ack* in between followed by *ack* at the end.
- **Dead-time** is given by  
 $MAXLEN(\llbracket lostcycle \rrbracket^+, \text{Arbiter})$   
 The dead-time is given by the length of the longest sub-execution of *Arbiter* with *lostcycle* invariantly true. □

## 4 Computing the Lengths of Extremal Models

In this section, we propose techniques for computing  $MAXLEN(D, M)$  and  $MINLEN(D, M)$  using symbolic search.

#### 4.1 Symbolic Search for Longest/Shortest Paths

Campos *et al* have investigated BDD based symbolic techniques for finding lengths of shortest/longest subsequences within the executions of  $M$  satisfying some simple conditions [3], [4]. We give a brief overview of their results. (Recall that fragments of executions of  $M$  are called sub-executions).

Consider a transition system  $M = (S, S_0, R, L)$ . Let

$$\begin{aligned} source \rightsquigarrow dest &= \{ \sigma \in subexec(M) \wedge \sigma[0] \in source \wedge \sigma[\#\sigma - 1] \in dest \} \\ source \leftrightarrow within &= \{ \sigma \in subexec(M) \wedge \sigma[0] \in source \wedge \\ &\quad \forall 0 \leq i \leq \#\sigma - 1. \sigma[i] \in within \} \end{aligned}$$

Then,  $source \rightsquigarrow dest$  denotes the set of all sub-executions of  $M$  which begin in a state from  $source$  and end in a state from  $dest$ . Also,  $source \leftrightarrow within$  denotes the set of all sub-executions of  $M$  which begin with a state from  $source$  and contain only the states from the set  $within$ .

Campos *et al* [3] have defined two algorithms (functions) called *MAXDELAY* and *MINDELAY* for computing maximum/minimum delay. These have been implemented in the model checking tool NuSMV<sup>3</sup> [5].

The function  $MINDELAY[source, dest, M]$  returns the step length (i.e. number of edges) in the shortest sub-execution within  $source \rightsquigarrow dest$ . If  $source \rightsquigarrow dest$  is empty then the algorithm returns the value  $\infty$ . Function  $MAXDELAY[source, within, M]$  in  $M$  returns the length (i.e. number of nodes) of the longest sub-execution in  $source \leftrightarrow within$ . If there are sub-executions of unboundedly many lengths then the algorithm returns the value  $\infty$ . In case the set  $source \leftrightarrow within$  is empty the algorithm returns the value 0. Formally, we have the following theorem.

**Theorem 4.1 (Campos, Clarke and Grumberg [4])**

$$\begin{aligned} MINDELAY[source, dest, M] &= \min \{ \#\sigma - 1 \mid \sigma \in source \rightsquigarrow dest \} \\ MAXDELAY[source, within, M] &= \max \{ \#\sigma \mid \sigma \in source \leftrightarrow within \} \end{aligned}$$

We now address the problem of finding the length of the longest sub-execution of  $M$  which starts with a state in  $source$  and ends with a state in  $dest$ . Consider a maximal (i.e. one which cannot be extended to another element) sub-execution  $\sigma$  in  $source \leftrightarrow \neg dest$ . Then, either  $\sigma$  ends in a state without any successor, or for all  $s \in S$ , if  $\sigma.s \in subexec(M)$  then  $s \in dest$ . Hence, computing  $MAXDELAY[source, \neg dest]$  does not give the correct answer.

Recall that CTL logic formula  $EFdest$  denotes the set of states of  $M$  from which there exists a path to a state in  $dest$ . Now, consider  $source \leftrightarrow EFdest$ . Let  $\sigma$  be a maximal sub-execution in it. Then,  $\sigma$  begins with a state from

---

<sup>3</sup> There are many variants of these algorithms in literature. We consider a version close to the NuSMV tool. Details can be found in the full paper.

```

Module AnyOnce
  m0st : { a,b,c } ;
  ASSIGN
    init(m0st) := {a,b} ;
    next(m0st) := case
      m0st = a : {a,b};
      m0st = b : c ;
      m0st = c : c;
    esac;
  DEFINE
    m0 := m0st = b;

```

Fig. 3.

*source* and ends with a state from *dest*. Hence, we have the following theorem.

**Theorem 4.2**  $MAXDELAY[source, EFdest, M] = \max \{ \#\sigma \mid \sigma \in source \rightsquigarrow dest \}$  □

Thus,  $MAXDELAY[source, EFdest, M]$  gives the length of the longest sub-execution of  $M$  which starts with a state in *source* and ends with a state in *dest*. If there are executions of unbounded lengths of this form, the algorithm gives the value  $\infty$ . If there is no such sub-execution (i.e the set  $start \leftarrow EF\ finish$  is empty) then the algorithm gives the value 0.

#### 4.2 Computing Extremal Model Lengths of QDDC Formulae

Let  $M = (S, S_0, R, L)$  be a transition system. Let  $m_0$  be a fresh propositional variable (not in image of  $L$ ) and let  $AnyOnce(m_0)$  be a transition system which nondeterministically sets  $m_0$  to true for at most one position in each of its execution. The SMV code for such a transition is given in Figure 3.

Let  $D'$  be a QDDC formula. By Theorem 2.2, we have a finite state automaton  $\mathcal{A}(D')$  which precisely accepts the models of  $D'$ . We can use  $\mathcal{A}(D')$  as a synchronous observer for  $D'$  and run it in synchronous (lock-step) parallel with  $M$  giving the transition system  $M \times \mathcal{A}(D')$ . Let  $end$  be a fresh propositional variable and (using the labelling function for  $\mathcal{A}(D')$ ) define  $end$  be true exactly when  $\mathcal{A}(D')$  is its final state.<sup>4</sup> In the following, we use  $D' = (true \wedge [m_0]^0 \wedge D)$ . Let,

$$(1) \quad M' = M \times AnyOnce(m_0) \times \mathcal{A}(true \wedge [m_0]^0 \wedge D)$$

Consider any sub-execution  $\sigma$  of  $M'$  which starts with  $m_0$  true and finishes with  $end$ . Then, it is obvious that  $\sigma \models_L D$ . Hence,  $MINDELAY[m_0, end, M']$  gives the length of the shortest sub-execution of  $M$  satisfying  $D$ . Formally,

---

<sup>4</sup> Our tool DCVALID is able to take a QDDC formula  $D'$  and construct an SMV module for  $\mathcal{A}(D')$  which defines such a proposition  $end$ .

**Theorem 4.3** *Let  $M'$  be as in Equation (1). Then,*

$$\text{MINLEN}(D, M) = \text{MINDELAY}[m_0, \text{end}, M']$$

*We omit the formal proof of this theorem.* □

Consider the product transition system  $M'$  in Equation (1). Let  $\text{endpref}$  be a new propositional letter which is *true* exactly when the observer automaton  $\mathcal{A}(D')$  is in a state from which it is possible for  $M'$  to reach a final state of  $\mathcal{A}$ . Thus, define  $\text{endpref} \Leftrightarrow (EF\text{end})$  where  $\text{end}$  is as before.

Consider a sub-execution of  $M'$  which begins with  $m_0$  true and which has  $\text{endpref}$  true throughout. Then, a maximal sub-execution of this form will be one where  $D$  holds. Hence, using Theorem 4.2 we have the following result.

**Theorem 4.4** *Let  $M'$  be as in Equation (1) and let  $\text{endpref} \Leftrightarrow EF\text{end}$ . Then,*

$$\text{MAXLEN}(D, M) = \text{MAXDELAY}[m_0, \text{endpref}, M']$$

*We omit the formal proof of this theorem.* □

## 5 Implementation and Experimental Results

Using Theorems 4.3 and 4.4, the computations of the values of  $\text{MAXLEN}(D, M)$  and  $\text{MINLEN}(D, M)$  can be reduced to simpler  $\text{MINDELAY}$  and  $\text{MAXDELAY}$  computations over the *transformed model*  $M'$ . Note that constructing  $M'$  requires taking synchronous product of  $M$  with the observer automaton for  $D'$  as in Equation (1). These reductions have been implemented into the tool DCVALID which takes as input an SMV module for system  $M$  as well the specifications consisting of  $\text{MINLEN}(D, M)$  and  $\text{MAXLEN}(D, M)$  computation commands. It produces a transformed SMV module corresponding to the transformed system  $M'$  as defined in Equation (1). It also produces the transformed  $\text{MINDELAY}$  and  $\text{MAXDELAY}$  computation specifications as in Theorems 4.3 and 4.4. We call this reduction as *observer generation*. Next, the generated SMV specification is given to the NuSMV tool to perform required  $\text{MINDELAY}$  and  $\text{MAXDELAY}$  computations by symbolic search. We shall call this step as *delay time computation*. The required answers are obtained at the end of this step. We now give some experimental results obtained using this tool.

### Experimental Results

We consider the  $n$ -cell synchronous bus arbiters MacArbV0 and MacArbV1 from Example 1.1 with their response time and dead time specifications as extremal model lengths as given in Example 3.4. The exact input to our tool DCVALID as well as the transformed SMV code produced by the tool can be found in the full version of the paper.

In Figure 4, we summarise the computation results obtained for the two 5-cell arbiters, namely MacArbV0 and MacArbV1. In particular, these show

3-cycle response time			Dead-time	
Arbiter	cell	cycles	Arbiter	cycles
MacArbV0	1	15 cycles	MacArbV0	5
	2 to 5	20 cycles	MacArbV1	inf
MacArbV1	1	15		
	2 to 5	16		

Fig. 4.

that the dead-time for the 5-cell arbiter MacArbV1 is  $\infty$ . *Thus, surprisingly, the arbiter MacArbV1 can loose unboundedly many consecutive cycles.* Note that this result is impossible to obtain using traditional model checking.

We compare the performances of the timing analysis of the arbiter circuits using (a) the extremal model length computation technique, and (b) traditional model checking of the specification given in Example 2.4. In both cases, first the observers (automata) are generated from QDDC formulae and then symbolic search is carried out. Hence, to measure performance, we give a pair of execution times in seconds. Let  $\uparrow n$  denote that the execution does not finish within  $n$  seconds, let  $\downarrow$  denote failed execution due to resource overrun (e.g. memory), and let  $*$  denote the absence of experiment.

The arbiter MacArbV0 from Figure 1 with different number of cells was used as the model and its two properties from Example 3.4 were checked. The results are given in the table below.

Problem	MAXLEN computation		Model Checking	
	Obs	Search	Obs	Search
20-cell arbiter Dead time	0.06	0.85	0.07	$\uparrow 600$
30-cell arbiter Dead time	0.06	19.95	0.07	*
40-cell arbiter Dead time	0.06	$\uparrow 600$	*	*
200 cell arbiter 1-cycle response time	0.07	13.78	4.47	105.43
20 cell arbiter 3-cycle response time	0.07	0.21	267	$\downarrow$
200 cell arbiter 3-cycle response time	0.07	18.38	$\uparrow 600$	*

## 6 Discussion

Many interesting properties of a discrete time systems such as response time and latency can be conveniently specified as finding *extremal* (least/greatest) value of a *parameter*  $k$  making a parameterised *QDDC* formula  $D(k)$  valid over the system  $M$ . Here, *QDDC* is a rich interval temporal logic which incorporates features (called durations) to count the number of occurrences of events within an interval.

To characterise such properties, in the paper we have proposed constructs (terms)  $MAXLEN(D, M)$  and  $MINLEN(D, M)$ . It is easy to see from their definitions that they capture extremal solutions of the following parameterised *QDDC* specifications:  $MAXLEN(D, M) = \min \{k \mid M \models \Box(D \Rightarrow \eta < k)\}$ , and  $MINLEN(D, M) = \max \{k \mid M \models \Box(D \Rightarrow \eta \geq k)\}$ .

In the paper, we have also proposed a technique to compute the values these terms  $MAXLEN$  and  $MINLEN$ . The technique makes use of the automata theoretic decision procedure for logic QDDC [12] and the symbolic search technique (called Delay computation) for shortest/longest path in  $M$  between specified sets of states *start* and *end* [3]. The technique has been implemented into the model checker DCVALID for checking QDDC properties of SMV programs.

Note that the traditional “yes/no” model checking can verify a property  $D(k)$  for a given constant  $k$ . However, finding extremal values of  $k$  requires trial-and-error with different values of  $k$ . Such a method is inherently incomplete and it can only be partially successful. If  $D(k)$  is unsatisfiable for all  $k$ , the trial-and-error method of finding minimum  $k$  will not terminate. Moreover, trial-and-error *cannot* determine the maximal  $k$  in general. At best, if the property is downward closed w.r.t  $k$  (i.e.  $D(k) \Rightarrow \bigwedge_{j < k} D(j)$ ) and we find least  $k$  s.t.  $D(k + 1)$  is violated then we can claim to have found maximal  $k$ . If there are unboundedly many  $k$  satisfying  $D(k)$ , again the trial-error method will not terminate. By contrast, our  $MAXLEN$  and  $MINLEN$  computation algorithms always give the answer. While the worst case theoretical complexity is high (non-elementary [12]), in practice the technique seems to be reasonably efficient as illustrated by our experiments.

In the paper, we have presented the results of experiments with our tool DCVALID to compute properties like 3-cycle response time and dead-time for circuits such as 20 to 200 cell bus-arbiters. *Using the MAXLEN computation method, we managed to establish for the first time that the arbiter MacArbV1 can loose unboundedly many cycles in sequence.*

Our experiments show that response time calculation using extremal model length computation can be orders-of-magnitude faster as compared with the traditional model checking of given response time value. In fact, the performance results tabulated in Section 5 show that traditional model checking is unable to handle problems like response and dead-time for circuits larger than 5-10 cells. By comparison the extremal model length computation technique

seems to work for circuits with over 200 cells. This trend is also borne out by some preliminary experiments with other problems like the job shop scheduling. Thus, we believe that the technique proposed in this paper represents a useful advance in our ability to carry out timing analysis. Even the dense-time systems can be analysed using these techniques by first digitizing them and then carrying out a discrete time analysis of their timing properties (see [6]).

### 6.1 Related Work and Comparison

Parameterised temporal logics have been studied by many researchers [7,1] and the question of finding optimal parameters has also been looked at [1]. These techniques rely on checking  $D(k)$  for some values of  $k$  up to some (typically large) theoretical bound  $m$  based on the model and the formula sizes. By contrast, the techniques based on symbolic search for the shortest/longest paths in  $M$  seem to be much more efficient in practice.

In their pioneering work, Campos *et al* first formulated the DELAY algorithms for symbolically computing the lengths of the shortest/longest paths within a transition system  $M$  between two specified sets of states, *source* and *dest* [3]. Campos, Clarke and Grumberg extended this by additionally specifying a LTL formula which must hold for the interval between *start* and *dest* [4]. The model checker NuSMV allows specification of sets *source* and *dest* by CTL formulae [5].

In this paper, we have generalised the method of Campos *et al* to compute the lengths of extremal sub-executions satisfying a formula of the logic QDDC. Note that the 3-cycle response time specification of our logic (Example 3.4) cannot be specified purely using the original MAXDELAY construct, and our extension is needed. Because of its ability to count events, logic QDDC can elegantly specify complex transactions and schedulability constraints. Hence we envisage that our techniques will be useful in analysing timing problems related to schedulability and planning.

## References

- [1] Alur, R., K. Etessami, S. La Torre and D. Peled, Parametric temporal logic for model measuring, in *Proc. 26th International Colloquium on Automata, Languages, and Programming*, LNCS 1644, Springer-Verlag, pp. 159–168, 1999.
- [2] Bouali, A., J.P. Marmorat, R. de Simone and H. Toma, Verifying Synchronous Reactive Systems Programmed in Esterel, in *Proc. FTRTFT'96*, LNCS 1135, Springer-Verlag, 1996.
- [3] Campos, S., E. Clarke, W. Marrero, M. Minea and H. Haraiishi, Computing Quantitative Characteristics of Finite-state Real-time Systems, in *Proc. IEEE Real-time systems symposium*, 1994.

- [4] Campos, S., E. Clarke and O. Grumberg, Selective Quantitative Analysis and Interval Model Checking, in *Proc. Eighth International Conference on Computer Aided Verification (CAV'1996)*, LNCS 1102, Springer-Verlag, 1996.
- [5] Cimatti, A, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani and A. Tacchella, NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking, in *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, LNCS 2404, Springer-Verlag, 2002.
- [6] Chakravorty, G. and P.K. Pandya, Digitizing Interval Duration Logic, in *Proc. International Conference on Computer Aided Verification (CAV 2003)*, Colorado, Boulder, July 2003 (Eds.) Warren A. Hunt, Jr. and Fabio Somenzi, LNCS 2725, Springer-Verlag, (2003) pp 167-179.
- [7] Emerson, E.A. and R.J. Treffler. Parametric quantitative temporal reasoning, in *Proc. 14th IEEE Symp. Logic in Computer Science (LICS'99)*, Trento, Italy, July 1999, pages 336–343, 1999.
- [8] Halbwachs, N., F. Lagnier and P. Raymond, Synchronous observers and the verification of reactive systems, in *Proc. Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, Springer-Verlag, 1993.
- [9] Henriksen, J.G., J. Jensen, M. Jorgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm, Mona: Monadic Second-Order Logic in Practice, in *First International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, LNCS 1019, Springer-Verlag, 1996.
- [10] McMillan, K., *Symbolic Model Checking*, Kluwer Academic Publisher, 1993.
- [11] Moszkowski, B., A Temporal Logic for Multi-Level Reasoning about Hardware, in *IEEE Computer*, **18**(2), 1985.
- [12] Pandya, P.K., Specifying and Deciding Quantified Discrete-time Duration Calculus Formulae using DCVALID: An Automata Theoretic Approach, in *Proc. Workshop on Real-time Tools (RTTOOLS'2001)*, Aalborg, Denmark, August 2001.
- [13] Pandya, P.K., Model checking CTL\*[DC], in *Proc. Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, Genova, Italy, LNCS 2031, Springer-Verlag, 2001.
- [14] Pandya, P.K., The saga of synchronous arbiter: On model checking quantitative timing properties of synchronous programs, in *Proc. Workshop on Synchronous languages, applications and programming (SLAP'2002)*, ENTCS 65.5, Elsevier Science B.V., April 2002.
- [15] Zhou Chaochen and M.R. Hansen, *Duration Calculus: A formal approach to real-time systems*, Springer, 2004.
- [16] Zhou Chaochen, C.A.R. Hoare and A.P. Ravn, A Calculus of Durations, *Info. Proc. Letters*, **40**(5), 1991.

## A Symbolic Delay Algorithms

We give algorithms which compute *MAXDELAY* and *MINDELAY*. These are originally due to Campos et al [3,4]. Tool NuSMV implements similar al-

```
function MINDELAY[source, dest]
  i := 0; W := source  $\wedge$  Reach;
  W' := Post(R, W)  $\cup$  W;
  while W  $\neq$  W'  $\wedge$  W'  $\cap$  dest =  $\emptyset$ 
  do
    i := i + 1; W := W';
    W' := Post(R, W)  $\cup$  W;
  od
  if W = W' then return  $\infty$ 
  else return i
fi

function MAXDELAY[start, within]
  i := 0; W := S;
  A := start  $\cap$  Reach
  W' := inside;
  while W  $\neq$  W'  $\wedge$  W'  $\cap$  A  $\neq$   $\emptyset$ 
  do
    i := i + 1; W := W';
    W' := Pre(R, W)  $\cap$  within ;
  od
  if W = W' then return  $\infty$ 
  else return i
fi
```

Fig. A.1.

gorithms where *MINDELAY*[start, within] is given by *MIN*[start, end] and *MAXDELAY*[start, within] is given by *MAX*[start,  $\neg$ within]. Note the negation of *within* in the *MAX* calculation.

## B Input to DCVALID for Response-time computation

We give the input file to tool DCVALID for specifying response time and dead-time computation. The file refers to module *system* of arbiter (given in next subsection) and its various variables such as *e2.Request*.

CTLFORM

```
    var lostcycle, req, ack, m0 ;
ctldc
-- Compute 3-cycle response
MAXDCLEN ( [[req]] && ((sdur ack = 2)^<ack>)) ;
```

```
-- Compute dead-time
MAXDCLEN ([[lostcycle]]) ;
```

```
SYSTEM smv system
aux1 m0;
```

OBSERVE

```
lostcycle :=
  (e1.Request || e2.Request || e3.Request || e4.Request || e5.Request )
```

```

    && !(e1.ackout || e2.ackout || e3.ackout || e4.ackout || e5.ackout ) ;
req := e4.Request ;
ack := e4.ackout ;
.

```

### B.1 Arbiter Module

This model of synchronous arbiter, originally due to MacMillan [10], is distributed with the NuSMV tool [5].

```

MODULE arbiter-element(above,below,init-token)

VAR
    Persistent : boolean;
    Token      : boolean;
    Request    : boolean;

ASSIGN
    init(Token) := init-token;
    next(Token) := token-in;
    init(Persistent) := 0;
    next(Persistent) := Request & (Persistent | Token);

DEFINE
    above.token-in := Token;
    override-out := above.override-out | (Persistent & Token);
    grant-out := !Request & below.grant-out;
    ackout := Request & (Persistent & Token | below.grant-out);

MODULE system

VAR
    e5 : arbiter-element(self,e4,0);
    e4 : arbiter-element(e5,e3,0);
    e3 : arbiter-element(e4,e2,0);
    e2 : arbiter-element(e3,e1,0);
    e1 : arbiter-element(e2,self,1);

DEFINE
    grant-in := 1;
    e1.token-in := token-in;
    override-out := 0;
    grant-out := grant-in & !e1.override-out;

```

## C Transformed NuSMV file Generated by DCVALID

```

MODULE main
VAR
  sys:system;
  m0st : { a,b,c } ;
  Newboundobs1 : Newboundobs1mod(lostcycle,req,ack,m0);
  Newboundobs2 : Newboundobs2mod(lostcycle,req,ack,m0);
  ASSIGN
    init(m0st) := {a,b} ;
    next(m0st) := case
      m0st = a : {a,b};
      m0st = b : c ;
      m0st = c : c;
    esac;
  DEFINE
    lostcycle:=((((sys.e1.Request | sys.e2.Request) | sys.e3.Request) |
      sys.e4.Request) | sys.e5.Request) & !((((sys.e1.ackout |
      sys.e2.ackout) | sys.e3.ackout) | sys.e4.ackout) | sys.e5.ackout));
    req:=sys.e4.Request;
    ack:=sys.e4.ackout;
    m0 := m0st = b;

  COMPUTE
    MAX [m0 , !(EF Newboundobs1.ENDST)]

  COMPUTE
    MAX [m0 , !(EF Newboundobs2.ENDST)]

MODULE Newboundobs1mod(lostcycle, req, ack, m0 )

VAR
  ODCST : { St1, St2, St3, St4, St5, St6, St7 };
  INIT
    ODCST = St1

  DEFINE
  DCST := case
    ODCST=St1 & !req : St1;
    ODCST=St1 & req & !ack & !m0 : St1;
    ODCST=St1 & req & !ack & m0 : St2;
    ODCST=St1 & req & ack & !m0 : St1;
    ODCST=St1 & req & ack & m0 : St3;

```

```

ODCST=St2 & !req : St1;
ODCST=St2 & req & !ack : St2;
ODCST=St2 & req & ack : St3;
ODCST=St3 & !req : St1;
ODCST=St3 & req & !ack & !m0 : St4;
ODCST=St3 & req & !ack & m0 : St2;
ODCST=St3 & req & ack & !m0 : St5;
ODCST=St3 & req & ack & m0 : St3;
ODCST=St4 & !req : St1;
ODCST=St4 & req & !ack & !m0 : St4;
ODCST=St4 & req & !ack & m0 : St2;
ODCST=St4 & req & ack & !m0 : St5;
ODCST=St4 & req & ack & m0 : St3;
ODCST=St5 & !req : St1;
ODCST=St5 & req & !ack & !m0 : St6;
ODCST=St5 & req & !ack & m0 : St2;
ODCST=St5 & req & ack & !m0 : St7;
ODCST=St5 & req & ack & m0 : St3;
ODCST=St6 & !req : St1;
ODCST=St6 & req & !ack & !m0 : St6;
ODCST=St6 & req & !ack & m0 : St2;
ODCST=St6 & req & ack & !m0 : St7;
ODCST=St6 & req & ack & m0 : St3;
ODCST=St7 & !req : St1;
ODCST=St7 & req & !ack & !m0 : St1;
ODCST=St7 & req & !ack & m0 : St2;
ODCST=St7 & req & ack & !m0 : St1;
ODCST=St7 & req & ack & m0 : St3;
esac ;

TRANS next(ODCST) = DCST

DEFINE
    ENDST := 0 | DCST=St3 | DCST=St5 | DCST=St7 ;

MODULE Newboundobs2mod(lostcycle, req, ack, m0 )

VAR
    ODCST : { St1, St2 };
INIT
    ODCST = St1

DEFINE
DCST := case

```

```

ODCST=St1 & !lostcycle : St1;
ODCST=St1 & lostcycle & !m0 : St1;
ODCST=St1 & lostcycle & m0 : St2;
ODCST=St2 & !lostcycle : St1;
ODCST=St2 & lostcycle : St2;
esac ;

```

```

TRANS next(ODCST) = DCST

```

```

DEFINE
  ENDST := 0 | DCST=St2 ;

```

```

MODULE arbiter-element(above,below,init-token)

```

```

-- as in Section A.1

```

```

MODULE system

```

```

-- as in Section A.1

```