Guided Reactive Synthesis with Soft Requirements

Amol Wakankar, Paritosh K. Pandya, and Raj Mohan Matteplackel

[1] Homi Bhabha National Institute, Mumbai, India.
Bhabha Atomic Research Centre, Mumbai, India.
Email: amolk@barc.gov.in
[2] Tata Institute of Fundamental Research, Mumbai 400005, India.
Email: {pandya, raj.matteplackel}@tifr.res.in

Abstract. This paper proposes a technique for synthesis of controllers from interval temporal logic QDDC based specification of hard and soft requirements. Hard requirements, which are past time temporal formulae, must hold invariantly. Soft requirements, which are also past time formulas may be satisfied by the controller "as much as possible". We propose a technique for computation of controller which guarantees hard requirements and meets as many soft requirements as possible at each step, in locally optimal fashion. We argue with examples and experiments that this past time soft reqirement guided synthesis provides a useful ability to specify and efficiently synthesize high quality controllers. We also investigate the use of a semi-symbolic automaton representation for efficient computation of controllers. The proposed method is implemented in a tool DCSYNTH. We present the case study of a minepump specification and its controller synthesis to illustrate our approach.

Keywords: Discrete Duration Calculus (QDDC), Reactive Controller Synthesis, Soft Requirements, Guided Synthesis, Latency Measurement.

1 Introduction

In reactive synthesis, the aim is to construct a controller (say a Mealy Machine) which explicitly computes the value of the output sequence for any given input sequence, in an online fashion such that the requirement is met. Considerable research has been carried out on the reactive synthesis problem and there are several tools which implement and experiment with reactive synthesis [11].

In reactive synthesis, systems are often under specified and several different controllers with distinct behaviors and qualities may all meet the specification. In this case "guidance" must be provided to the synthesizer to choose amongst them. A critical parameter in acceptance of automatic synthesis technique is the *quality* of the synthesized controller [1,3]. Thus, just correct-by-construction synthesis is not sufficient.

A temporal logic formula implicitly specifies the allowed sequence of inputs and outputs. This paper describes a method and a tool **DCSynth** which allows synthesis of controllers from safety and bounded liveness requirements given in interval temporal logic **QDDC** [16–18]. The paper mainly investigates the role of **soft requirements** (with priorities) in obtaining high quality controllers. In this context, a QDDC formula specifies past time properties, and it holds at a point in behavior if the past of the point satisfies the property (Reader may intuitively think of such a formula as specifying a Deterministic Finite Automaton (DFA) accepting the past). QDDC is a highly succinct and expressive logic for specifying DFA [16, 17]. Its bounded counting and regular expression like primitives allow complex quantitative properties to be specified elegantly. The minepump case study in Section 4 illustrates the kind of properties of interest to this work.

In DCSynth synthesis, *hard requirements* must be invariantly satisfied at all points in execution whereas *soft requirements* may be satisfied "as much as possible" in a best effort manner by the controller. The soft requirements (which are QDDC formulas) can be given weights. For each state of the controller, the synthesis algorithm selects from all permissible outputs meeting the hard requirements, the one which satisfies a (weighted) maximal subset of the soft requirements, in a "locally optimal fashion". Note that the soft requirements are met only if they do not contradict hard requirements and hence they hold intermittently. The use of *past time temporal properties* to guide the choice of output is a distinct feature of our method.

The soft requirements provide a powerful and practically useful ability to guide the controller synthesis to get high quality controllers. Quality can be measured as *worst case latency* achieved by the controller. An associated tool, CTLDC, allows measurement of worst case latencies using symbolic techniques for finding longest and shortest paths [19]. We present the case study of a minepump specification and show how past based soft requirements affect the worst case latency of the controller. Another quality parameter, *robustness*, pertains to the ability of the controller to meet commitments even when (some) environmental/plant assumptions are violated, and the ability of the controller to recover from transient environmental errors [3]. This can be achieved in our framework by specifying commitments as soft requirements, where as in hard requirements, commitments may be conditioned on assumptions being true. A separate paper [20] deals with the ability of DCSynth to specify and synthesize robust controllers [3] as well as efficient run-time enforcement shields [4].

As its second main contribution, this paper explores the implementation of guided synthesis using **semi-symbolic automata**. Such automata were introduced by the tool MONA [12], and used by the tool DCVALID for constructiong DFAs for QDDC formulas [16, 17]. We shall call such automata as **SSDFA**. Some optimization which directly allow computation of locally optimal controllers over SSDFA representation are presented. Eager minimization of automata/controllers is used at all stages of synthesis for efficiency. Experimental results evaluating the effect of these optimizations are presented in Tables 3 and 4.
We summarize the main contributions of this paper below:

- We propose a technique to automatically synthesize controllers from discrete duration calculus QDDC formulas. This extends the past work on model checking of QDDC with synthesis abilities.
- The paper explores guided synthesis of controllers based on past time soft requirements in a locally optimal fashion. A minepump case study shows how such soft requirements affect latencies in resulting controllers. Links to several other case studies are provided. To our knowledge, DCSynth is the first tool to explore past time soft requirement guided synthesis.
- We adapt semi-symbolic automaton (SSDFA) representation, originally proposed by tool MONA, for efficient implementation of the synthesis technique. Some optimizations for efficient synthesis over SSDFA are presented.
- We claim that past time soft requirements provide a very useful facility for efficient synthesis of high quality controllers.

DCSynth is available for download at [21], where the details of experiment reported here and full version of the paper are also available.

1.1 Related work

Reactive synthesis from LTL specification has been widely studied and considerable theory exists [1, 2]. Some of this theory has found its way into tools too. While several tools support safety synthesis over circuits (see [11], safety track), tools such as Acacia+ [5] and BoSy [7] support synthesis from temporal logics such as LTL and PSL (see [11], LTL track). Most tools focus on the future fragment of LTL. By contrast, this paper focuses on invariance of complex past time temporal properties. Logic QDDC is particularly suited for such specification as illustrated by the minepump example.

Duration Calculus(DC) was originally proposed by Zhou, Hoare and Ravn [6]. The use of DC in real time system design was explored in [22]. The minepump example was first encoded in DC by Liu [14], and then adapted to QDDC [16]. There is an extensive work on requirement modelling using DC as well as model checking DC and QDDC properties [16, 19]. However, algorithmic synthesis of controllers from QDDC specification as presented here seems new. Fraenzle [9, 10] presented an early analysis of this problem.

Most synthesis tools have focused on correct-by-construction synthesis from hard requirements. Criticizing this, Bloem et. al. [2,3] analyzes need for robust synthesis. The use of past time temporal soft requirements to guide synthesis, as proposed here, is new to our knowledge. For example, none of tools in recent SYNTCOMP17 [11] address the issue of guided synthesis.

Most synthesis tools use *fully symbolic BDD* representation for automata, game graphs and controllers [5]. Another prominent approach is using *bounded synthesis using SAT solving* [7, 8]. By contrast, the use of semi-symbolic automata (SSDFA) with eager minimization for synthesis, explored in this paper, is a novel alternative.

2 Logic QDDC

Let Σ be a finite non empty set of propositional variables. A *word* σ over Σ is a non-empty finite sequence of the form $P_0 \cdots P_n$ where $P_i \subseteq \Sigma$ for each $i \in \{0, \ldots, n\}$. Let $len(\sigma) = n + 1$, $dom(\sigma) = \{0, \ldots, n\}$ and $\forall i \in dom(\sigma) : \sigma(i) = P_i$. Let $\sigma[i : j]$ denote the subsequence $P_i \cdots P_j$ and $\sigma[i]$ denote P_i.

The syntax of a *propositional formula* over Σ is given by:

$$\varphi := 0 \mid 1 \mid p \in \Sigma \mid !\varphi \mid \varphi \;\&\&\; \varphi \mid \varphi \;||\; \varphi \mid -\varphi,$$

and operators such as $\Rightarrow$ and $\Leftrightarrow$ are defined as usual. Let Ω_Σ be the set of all propositional formulas over Σ. Let $i \in dom(\sigma)$. Then the satisfaction relation $\sigma, i \models \varphi$ is defined inductively as follows: $\sigma, i \models 1$ for all $i \in dom(\sigma)$. Also $\sigma, i \models p$ iff $p \in \sigma(i)$, $\sigma, i \models !\varphi$ iff $\sigma, i \not\models \varphi$, and $\sigma, i \models -\varphi$ iff $i > 0$ and $\sigma, i - 1 \models \varphi$. The satisfaction of boolean combinations defined in a natural way.

The syntax of a QDDC formula over Σ is given by:

$$D := \langle\varphi\rangle \mid [\varphi] \mid [[\varphi]] \mid \{\{\varphi\}\} \mid D \mathbin{\hat{}} D \mid !D \mid D \;||\; D \mid D \;\&\&\; D \mid D^* \mid \\ ex\ p.\ D \mid all\ p.\ D \mid slen \bowtie c \mid scount\ \varphi \bowtie c \mid sdur\ \varphi \bowtie c,$$

where $\varphi \in \Omega_\Sigma$, $p \in \Sigma$, $c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$.

An *interval* over a word σ is of the form $[b, e]$ where $b, e \in dom(\sigma)$ and $b \leq e$. Let $Intv(\sigma)$ be the set of all intervals over σ. Let σ be a word over Σ and let $[b, e] \in Intv(\sigma)$ be an interval. Then the satisfaction relation of a QDDC formula D over Σ, written $\sigma, [b, e] \models D$, is defined inductively as follows:

$$
\begin{array}{lll}
\sigma, [b,e] \models \langle\varphi\rangle & \text{iff} & b = e \text{ and } \sigma, b \models \varphi, \\
\sigma, [b,e] \models [\varphi] & \text{iff} & b < e \text{ and } \forall b \leq i < e : \sigma, i \models \varphi, \\
\sigma, [b,e] \models [[\varphi]] & \text{iff} & \forall b \leq i \leq e : \sigma, i \models \varphi, \\
\sigma, [b,e] \models \{\{\varphi\}\} & \text{iff} & e = b + 1 \text{ and } \sigma, b \models \varphi, \\
\sigma, [b,e] \models D_1 \hat{}\, D_2 & \text{iff} & \exists b \leq i \leq e : \sigma, [b,i] \models D_1 \text{ and } \sigma, [i,e] \models D_2,
\end{array}
$$

with Boolean combinations $!D$, $D_1 \mid\mid D_2$ and D_1 && D_2 defined in the expected way. We call word σ' a p-variant, $p \in \Sigma$, of a word σ if $\forall i \in dom(\sigma), \forall q \neq p : \sigma'(i)(q) = \sigma(i)(q)$. Then $\sigma, [b,e] \models ex\ p.\ D \Leftrightarrow \sigma', [b,e] \models D$ for some p-variant σ' of σ and, $\sigma, [b,e] \models all\ p.\ D \Leftrightarrow \sigma, [b,e] \not\models ex\ p.\ !D$.

Entities *slen* , *scount* , and *sdur* are called *terms*. The term *slen* gives the length of the interval in which it is measured, *scount* φ where $\varphi \in \Omega_\Sigma$, counts the number of positions including the last point in the interval under consideration where φ holds, and *sdur* φ gives the number of positions excluding the last point in the interval where φ holds. Formally, for $\varphi \in \Omega_\Sigma$ we have $slen(\sigma, [b,e]) = e - b$, $scount(\sigma, \varphi, [b,e]) = \sum_{i=b}^{i=e} \left\{ \begin{matrix} 1, \text{ if } \sigma, i \models \varphi, \\ 0, \text{ otherwise.} \end{matrix} \right\}$ and $sdur(\sigma, \varphi, [b,e]) = \sum_{i=b}^{i=e-1} \left\{ \begin{matrix} 1, \text{ if } \sigma, i \models \varphi, \\ 0, \text{ otherwise.} \end{matrix} \right\}$

In addition we also use the following derived constructs: $pt = < 1 >$ and $ext = !pt$. Also $<> D = true\hat{}D\hat{}true$ and $[]D = !<>!D$. Also $pref(D) = !((!D)\hat{}true)$. Thus, $\sigma, [b,e] \models []D$ iff $\sigma, [b', e'] \models D$ for all sub-intervals $b \leq b' \leq e' \leq e$. Also $\sigma, [b,e] \models pref(D)$ iff $\sigma, [b, e'] \models D$ for all prefix intervals $b \leq e' \leq e$.

Finally, $\sigma \models D$ iff $\sigma, [0, len(\sigma) - 1] \models D$, and the set of behaviours accepted by D is given by: $L(D) = \{\sigma \mid \sigma \models D\}$.

Theorem 1. *[17] For every formula D we can construct a DFA $\mathcal{A}(D)$ over alphabet 2^Σ such $L(\mathcal{A}(D)) = L(D)$. We call $\mathcal{A}(D)$ a* formula automaton *for D.*

The reader may refer to numerous papers [15–18] on QDDC for detailed description and examples of QDDC specifications and its model checking tool DCVALID. We now give some properties in QDDC which would be used for minepump requirement specification in Section 4.

- $lag(P,Q,n)$: `[]([[P]] && slen>=n => slen=n^[[Q]])`. Specifies that in any observation interval if P holds continuously for $n+1$ cycles and persists then Q holds from $(n+1)^{th}$ cycle onwards and persists till P persists.
- $tracks(P,Q,n)$: `[](<-P>^[[P]] && slen<n => [[Q]])`. In any observation interval that begins with rising edge of P, proposition Q remains true as long a P is true upto n cycles.
- $sep(P,n)$: `[](<-P>^[!P]^<P> => slen>n)`. Any interval which begins with a falling edge of P and ends with the next rising edge of P will have length of at least n cycles.
- $ubound(P,n)$: `[]([[P]] => slen<n)`. In any observation interval P can be continuously true for at most n cycles.

QDDC formulas can be used to specify past time properties in system behaviours (see [16]). For an infinite behaviour ρ we have $\rho, i \models D$ **iff** $\rho[0:i] \models D$ **iff** $\rho[0:i] \in L(D)$. Thus, property D holds invariantly over ρ iff $\forall i : \rho[0:i] \in L(D)$.

3 DCSynth Specification and Controller Synthesis

Our tool uses DCSynth specification as its input for controller synthesis. Formally, a *DCSynth specification* is a tuple $S = (I, O, W, D^h, \wedge_{i=1}^{i=k}(w_i \Leftrightarrow D_i^s), \langle P_1 : \theta_1, \cdots, P_l : \theta_l \rangle)$ where I and O are set of *input* and *output* variables respectively, of the controller. The QDDC formula D^h, which is over $I \cup O$, specifies *hard requirement* on the synthesized controller, i. e. every execution of a controller must satisfy D^h invariantly. We have a list of indicator definitions $D_1, \ldots, D_k$ where each D_i is associated with the *indicator variable* w_i which witnesses whether D_i^s holds for the execution so far, i. e. $\sigma, j \models w_i$ iff $\sigma, [0,j] \models D_i^s$. Let $W = \{w_i \mid 1 \leq i \leq k\}$ be the set of indicator variables. The *soft requirements* $\langle P_1 : \theta_1, \cdots, P_l : \theta_l \rangle$ is a list of propositional formulas where each P_i is a propositional formula over $I \cup O \cup W$ and $\theta_i \in \mathbb{N}$. Note that past time temporal requirements are included in soft requirements through the use of indicators w_i.

Soft Requirement Guided Quantitative Value of an Assignment For assignment $(i,o,w) \in 2^I \times 2^O \times 2^W$ let $(i,o,w) \models P_j$ denote that P_j evaluates to true for assignment (i,o,w). Let $val(P_j, (i,o,w)) = \theta_j$ if $(i,o,w) \models P_j$ and 0 otherwise. Then, *quantitative value of soft requirement* $\langle P_1 : \theta_1, \ldots, P_l : \theta_l \rangle$ over assignment (i,o,w) is given by $ValSoft(\langle P_1 : \theta_1, \ldots, P_l : \theta_l \rangle, (i,o,w)) = \Sigma_{j=1}^{j=l} val(P_j, (i,o,w))$. Note that $ValSoft$ function gives a quantitative value to every assignment for a given soft requirement.

Hence, it makes sense to talk of optimal assignments. Of course there may be several optimal assignments.

3.1 Controller Synthesis Method

Given $S = (I, O, W, D^h, \wedge_{i=1}^{i=k}(w_i \Leftrightarrow D_i^s), \langle P_1 : \theta_1, \cdots, P_l : \theta_l\rangle)$, a DC-Synth specification, we synthesize a controller as below (see pseudo code in Appendix A).

– The formula $D^{Ind} = pref(\wedge_{i=1}^{i=k}(true\hat{} < w_i ><=> D_i^s))$ states that at every point i in execution the value of w_i equals the truth-value of D_i^s over the past of i.
– A *monitor automaton* $A^{mon} = \mathcal{A}(D^h \wedge D^{Ind})$ is obtained using DC-VALID and MONA (see Theorem 1). This automaton has the alphabet $2^{I \cup O \cup W}$. The automaton is reduced to its minimal deterministic form. The following proposition relates infinite words with the language of finite words accepted by A^{mon}.

  Proposition 1. *For $\rho \in (2^{I \cup O \cup W})^\omega$ and for any position $i \in \mathbb{N}$, we have $\rho[0:i] \in L(A^{mon})$* **iff** *$(\rho[0:i] \models D^h$ and $\forall j \leq i : \rho[j] \models w_i$* **iff** *$\rho[0:j] \models D_i^s)$.* $\square$

  This automaton A^{mon} forms the arena on which further synthesis is carried out. Note that, unlike many other tools, the arena is not expanded into a game graph (which can be much larger for large alphabets).
– The *Maximally Permissive Non deterministic Controller* (**MPNC**) is computed from the automaton A^{mon} using standard safety synthesis algorithm. This algorithm iteratively removes those states from which there exists an input combination for which all output combinations lead to bad states. If the initial state gets pruned in construction, the specification is unrealizable. A counter-strategy in tree form is displayed as explanation of unrealizability. The implementation details are given in Section 5.1.
  Figure 1(a) gives an example of a MPNC automaton. Its alphabet Σ is 4-bit vectors giving value of propositions $(req_1, req_2, ack_1, ack_2)$.
  Remark 1: Given partition of variables into input, output and indicators as I, O and W, the MPNC is a DFA over $2^{(I \cup O \cup W)}$. This can also be treated as a *output-nondeterministic Mealy machine* with inputs 2^I and outputs $2^{(O \cup W)}$. For example, the DFA in Figure 1(a) actually denotes an output-nondeterministic Mealy machine with input alphabet (req_1, req_2) and output alphabet (ack_1, ack_2). Automaton is

nondeterministic in output as from state 1, on input $(1,1)$ it can move to state 2 with output $(1,0)$, or state 3 with output $(0,1)$. $\Box$

- As stated in Remark 1 above, MPNC as a Mealy Machine can be output nondeterministic. In constructing a deterministic controller, we must resolve (prune) this nondeterminism. We choose locally optimal output based on the quantitative value given by $ValSoft$. This maximizes the soft requirements. (If more than one choice of output gives the same maximal value we choose one arbitrarily.) This gives the *Locally Optimal Deterministic Controller* (**LODC**). This greedy strategy does not guarantee global optimality. The implementation details are given in Section 5.2.
- The LODC can then be encoded as controller in any target language. We provide the encoding of LODC to LUSTRE/SCADE or NuSMV, which also allows us to do simulation and model checking on the generated controller.

4 Case Study: Minepump Specification

In this section we illustrate the effect of *soft requirements* on the quality of the synthesized controllers with a case study of a minepump controller specification [14, 17]. The controller has two input sensors: high water level sensor $HH2O$ and methane leakage sensor $HCH4$; and two outputs, $Alarm$ to sound/persist the alarm, and $PumpOn$ to keep the pump on. The objective of the controller is to *safely* operate the pump and the alarm in such a way that the water level is never dangerously high, denoted by the auxiliary variable $DH2O$. We have following assumptions on the mine and the pump.

- Sensor reliability: $pref(DH2O \Rightarrow HH2O)$. If $HH2O$ is false then so is $DH2O$.
- Water seepage: $tracks(HH2O, !DH2O, w)$. Water level cannot become dangerous within first w cycles of it becoming high.
- Pump capacity: $lags(PumpOn, !HH2O, \epsilon)$. If pump is continuously on for at least $\epsilon + 1$ cycles then water level will not be high at the end (i. e. $\epsilon + 1$ cycles).
- Methane release: $sep(HCH4, \zeta)$ and $ubound(HCH4, \kappa)$. The minimum separation between the two leaks of methane is ζ cycles and the methane leak cannot persist for more than κ cycles.
- Initial condition: $init(<!HH2O> \&\& <!HCH4>, slen = 0)$. Initially neither the water level is high nor there is a methane leakage.

The commitments are:

- Alarm control: $lags(HH2O, Alarm, \delta)$ and $lags(HCH4, Alarm, \delta)$ and $lags(!HH2O\&\&!HCH4, !Alarm, \delta)$. If the water level is high or methane is present, then alarm will sound within δ cycles and persist as long as condition lasts. If neither the water level is dangerous nor there is a methane leakage then alarm turns off within δ cycles. The slack δ is to permit controller to respond.
- Safety condition: $[[!DH2O\ \&\&\ ((HCH4||!HH2O) \Rightarrow !PumpOn))]]$. The water level should never become dangerous and whenever there is a methane leakage or absence of high water, the pump should be off.

Let $ASSUME$ be the conjunction of assumptions and let $COMMIT$ be the conjunction of commitments. Then the hard requirement is

$$MinePump(w, \epsilon, \zeta, \kappa, \delta) = \texttt{all DH2O. (pref(ASSUME) => COMMIT)}$$

To improve the quality of pump controller we specify several soft requirements. Note that in hard requirement there is a slack of $\delta + 1$ cycles within which alarm should turn on/off. All the three soft requirements below resolve this slack in favour of keeping alarm off as much as possible. Additionally,

- *MPV1*: keep *pump on* whenever possible. $\langle PumpOn : 2, !Alarm : 1\rangle$.
- *MPV2*: keep *pump off* if there is a methane leak in the last 2 cycles otherwise keep it on. $\langle YHCH4 \Rightarrow !PumpOn : 4, PumpOn : 2, !Alarm : 1\rangle$. Indicator variable $YHCH4$ witnesses the formula `true^(slen=2 && <><HCH4>)` recording that there is a methane leakage in the last 2 cycles.
- *MPV3*: keep *pump off* as much as possible. $\langle !PumpOn : 2, !Alarm : 1\rangle$.

We have synthesized three different controllers by using each one of $MPV1$, $MPV2$ and $MPV3$ as the soft requirement, and $MinePump(8, 2, 10, 1, 1)$ as the hard requirement. Table 1 lists the performance of DCSynth tool on these three instances. Appendix B gives textual input to the tool and simulations carried out using synthesized controllers.

It can be argued that these controllers have different quality attributes. For example, $MPV1$ gives rise to a controller that aggressively gets rid of water by keeping pump on whenever possible. On other hand, $MPV3$ saves power by keeping pump off as much as possible. $MPV2$ also aggressively keeps pump on but it opts for a safer policy of not keeping pump on for two cycles even after methane is gone.

Table 1. DCSynth synthesis for the hard requirement $MinePump(8, 2, 10, 1, 1)$ with soft requirements $MPV1$, $MPV2$ and $MPV3$.

Soft Requirement	Controller Synthesis		
	states	time (Sec)	Memory (MB)
$MPV1$	31	0.07	9.1
$MPV2$	34	0.09	8.9
$MPV3$	83	0.04	9.1

Table 2. Worst Case Latency Analysis using CTLDC for MAXLEN computation

Soft Requirement	Response Formula (D^p)	MAXLEN value
$MPV1$	$[[AssumptionOk \&\& HH2O]]$	4
$MPV2$	$[[AssumptionOk \&\& HH2O]]$	7
$MPV3$	$[[AssumptionOk \&\& HH2O]]$	8

4.1 Quantitative Latency Measurement

A model checking technique, implemented in a tool CTLDC [19], can measure the worst case longest/shortest span of a QDDC formula D^p in a system M. This involves symbolic search for longest/shortest paths. Formally, $MAXLEN(D^p, M) = \sup\{e - b \mid \rho[b, e] \models D^p, \rho \in Exec(M)\}$ which computes the length of the longest interval satisfying D^p within the executions of M. Similarly, for $MINLEN(D^p, M)$. Thus, CTLDC allows measurement of *user defined latencies* in the worst case. This can be used to evaluate the performance (latency) of synthesized controllers. Table 2 gives worst case latency measurements carried out using tool CTLDC for the minepump controllers with soft requirements $MPV1$, $MPV2$ and $MPV3$. Clearly, $MPV1$ gets rid of water the fastest (in atmost 4 cycles) whereas $MPV3$ is the slowest in this sense (requiring upto 8 cycles).

More Case Studies Several case studies of controller synthesis have been carried out using DCSynth. These include variants of bus arbiters, small industrial case studies such as alarm annunciation system and valve control in a steam boiler. See Appendix C for details.

5 Synthesis with Semi-Symbolic DFA and Experiments

An interesting representation for total and deterministic finite state automata was introduced and implemented by Klarlund et. al. in the tool MONA [12]. It was used to efficiently compute formula automaton for MSO over finite words. We denote this representation as *Semi-Symbolic DFA* (SSDFA). In this representation, the transition function is encoded as *multi-terminal BDD* (MTBDD). The reader may refer to original papers [12, 13] for further details of MTBDD and the MONA DFA library.

Here, we briefly describe the SSDFA representation, and then consider controller synthesis on SSDFA. Figure 1(a) gives an explicit DFA. Its alphabet Σ is 4-bit vectors giving value of propositions $(req_1, req_2, ack_1, ack_2)$ and set of states $S = \{1, 2, 3, 4\}$. Being a safety automaton it has a unique reject state 4 and all the missing transitions are directed to it. (State 4 and transitions to it are omitted in Figure 1(a) for brevity.)

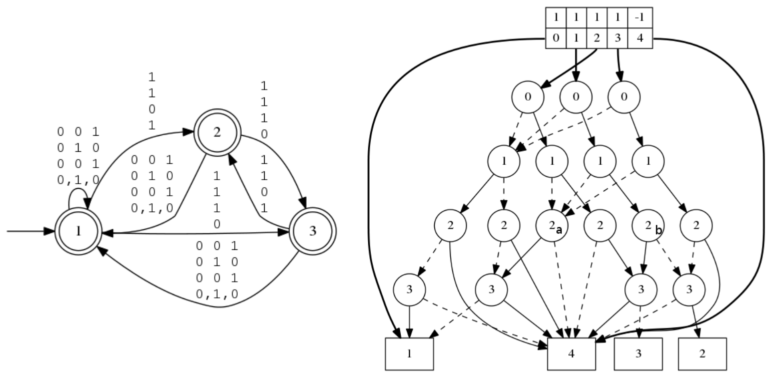

Fig. 1. (a): External format (b): SSDFA format

Figure 1(b) gives the SSDFA for the above automaton. Note that states are explicitly listed in the array at top and final states are marked as 1 with non-final states marked as -1. (For technical reasons there is an additional state 0 which may be ignored here and state 1 may be treated as the initial state.) Each state s points to shared MTBDD node encoding the transition function $\delta(s) : \Sigma \rightarrow S$ with each path ending in the next state. Each circular node of MTBDD represents a *decision node* with indices $0, 1, 2, 3$ denoting variables $req_1, req_2, ack_1, ack_2$. Solid edges lead to true cofactors and dotted edges to false cofactors.

MONA provides a DFA library implementing automata operations including product, complementation, projection and minimization on SSDFA. Moreover, automata may be constructed from scratch by giving list of states and adding transitions one at a time. A default transition must be given to make the automaton total. DCVALID computes a minimized, language-equivalent SSDFA, denoted by $\mathcal{A}(D)$, for a QDDC formula D [16, 17]. MONA and DCVALID use eager minimization while converting formula into SSDFA.

Remark 2: For A^{mon}, a transition has the form $\delta(s, (i, o, w))$ for $i \in 2^I, o \in 2^O, w \in 2^W$. However, the value of w in a given automaton is uniquely determined by $(s,(i,o))$. Hence we can abbreviate the transition as $\delta(s,(i,o))$. $\square$

We now show the synthesis of MPNC and LODC for the DCSynth specification $(I, O, W, D^h, \wedge_{i=1}^{i=k}(w_i \Leftrightarrow D_i^s), \langle P_1 : \theta_1, \cdots, P_l : \theta_l \rangle)$.

5.1 Computing MPNC

Let $A^{mon} = \langle S, 2^{I \cup O \cup W}, F, \delta \rangle$ be the automaton for $D^h \wedge D^{Ind}$ obtained by computing $\mathcal{A}(D^h \wedge D^{Ind})$ (cf. Section 3). Now to construct MPNC from A^{mon} we first compute the set of winning states $G \subseteq S$ with the following property: $s \in G$ iff $\forall i \exists o : \delta(s,(i,o)) \in G$. Let $Cstep(A^{mon}, X) = \{s \mid \forall i \exists o : \delta(s,(i,o)) \in X\}$. Then $ComputeWINNING(A^{mon}, I, O)$ iteratively computes G as follows:

Algorithm 1 ***ComputeWINNING***

```
G=F; do {G1=G; G=Cstep($A^{mon}$,G1)} while (G != G1); return(G)
```

This requires efficient implementation of $Cstep(A^{mon}, X)$ over SSDFA A^{mon}. The symbolic algorithm for Cstep marks (a) each leaf node having state s by truth value $s \in X$, (b) each decision node associated with an input variable with AND of its children's value, and (c) each decision node associated with output variable with OR of its children's value. The computation is carried out bottom up on MTBDD and take time $|MTBDD|$, where by $|MTBDD|$ we mean the number of nodes in it.

Next we compute the automaton $A^{mpnc} = \langle G \cup \{r\}, 2^{I \cup O \cup W}, G, \delta' \rangle$ by only retaining transitions between the winning states G. We consider the following two methods.

- *Enumerative method:* A^{mpnc} is constructed by adding a transition at a time as follows: for any $s \in G$ if $\delta(s,(i,o,w)) \in G$ then $(s,(i,o,w)), \delta(s,(i,o,w)) \in \delta'$. Note that w is fixed for given s and (i,o) by A^{mon} (see Remark 2). Clearly, this algorithm has time complexity $|S| \times 2^{|I \cup O|}$. Finally, we make A^{mpnc} total by adding all the unaccounted transitions to r.
- *Symbolic method:* in this method, the MTBDD of A^{mon} is modified so that each edge pointing to a state in $S - G$ is changed to go to r. Note that this makes states in $S - (G \cup \{r\})$ inaccessible. Now this modified SSDFA is minimized to get rid of inaccessible states and to get smaller MPNC. The time complexity of this computation is $O(|MTBDD|)$ for modifying the links and $Nt\log(N)$ for minimization where N is number of states and t is the size of alphabet in A^{mon}.

Table 3 gives experimental results comparing the computation of A^{mpnc} using the two algorithms. It can be seen that the symbolic algorithm can be faster by several orders of magnitude.

Table 3. Enumeration method vs symbolic method for MPNC synthesis.

Hard Requirement	Enumeration Method		Symbolic Method	
	time (Sec)	Memory (KB)	time (Sec)	Memory (KB)
$Arb^{hard}(4,4)$	0.011	3.9	0.0009	2.8
$Arb^{hard}(5,5)$	0.43	28.4	0.02	23.3
$Arb^{hard}(6,6)$	33.05	425.6	1.8	321.9
$Arb^{soft}(4,2)$	0.003	2.4	0.0002	2.0
$Arb^{soft}(5,3)$	0.13	10.7	0.004	7.1
$MPV1$	0.002	2.5	0.0005	2.3
$MPV2$	0.003	2.8	0.0009	2.4
$MPV3$	0.002	2.5	0.0005	2.3
$AMBA$	6284.5	268.6	0.22	82.8

5.2 Computing LODC

Recall that A^{mpnc} is an output-nondeterministic Mealy Machine (see Remark 1). To construct a deterministic controller we must resolve this nondeterminism by selecting one of the choices in output. Note that the choice of output can have great impact on the *quality* of the controller. We resolve the nondeterminism by choosing an output that maximizes the (weighted) number of soft requirements given by $ValSoft$ function.

To compute A^{lodc} we again have two methods:

– *Enumerative method:* For each state $s \in G$ and for each input $i \in 2^I$ compute $o_{max} = argmax_{o \in 2^O}\{ValSoft(\langle P_1 : \theta_1, \cdots, P_l : \theta_l\rangle, (i, o, w)) \mid \delta(s, (i, o, w)) \in G\}$. Add the transition $(s, (i, o_{max}, w), \delta(s, (i, o_{max}, w)))$ to A^{lodc}. This takes $|G| \times 2^{|I|}$ computations of o_{max}. For finding o_{max} for a given input, we enumerate each possible output maintaining the value of current optimal o_{max}. Finding o_{max} takes $2^{|O|} \times l$ steps where l is the number of soft goals. Hence, time complexity of entire algorithm is $|G| \times 2^{|I \cup O|} \times l$.

– *Symbolic method:* For this optimization to be applicable we assume that all the input variables occur before the output variables followed by the indicator variables. We also assume that all soft requirements are defined as propositional formulas over $(O \cup W)$. A node in MTBDD is called *frontier node* if it is labelled with an output or an indicator variable, and all its ancestors are labelled with input variables. (In Figure 1(b), these are nodes labelled 2. They happen to occur at same level in this example.) For each frontier node enumerate each path π within the MTBDD below the frontier node (this fixes $O \cup W$) and update o_{max} for paths seen so far. This takes time $O(d_f \times l)$ where d_f is the number of paths in MTBDD below the frontier node f. This optimal output $o_{max}(f)$ is stored in each frontier node f. The total time taken is $O(d_{output} \times l)$ where $d_{output} = \Sigma_{f \in Fr}\ d_f$, where Fr is the set of all frontier nodes.

In second step, for each state $s \in G$, enumerate each path from state s to a frontier node f. This fixes the valuation of input i. Insert a transition $(s, (i, o_{max}, w), \delta(s, (i, o_{max}, w)))$ to A^{lodc}. Let the total number of paths upto frontier nodes be d_{input}. Then the second step takes time $O(d_{input} + |G|)$ where time taken to insert a transition is assumed to be constant. Hence total time for computing A^{lodc} is $O(d * l + |G|)$ where d is total number of paths in MTBDD of A^{mpnc}.

Note that in worst case, the total number of MTBDD paths d is of size $O(2^{|I \cup O|})$ and two algorithms have the same complexity. But in most cases, the total number of MTBDD paths $d \ll 2^{|I \cup O|}$ and the symbolic algorithm is more efficient. In **Table 4** we present experimental evaluation of time taken for computing A^{lodc} using the two techniques.

Table 4. LODC synthesis: enumeration vs symbolic(time(seconds), memory(KB)).

Hard Requirement	Soft Requirement	Enumeration Method		Symbolic Method	
		time	Memory	time	Memory
$Arb^{hard}(4,4)$	$ack4 : 2^4, \ldots, ack1 : 2^1$	0.019	3.7	0.006	5.0
$Arb^{hard}(5,5)$	$ack1 : 2^1, \ldots, ack5 : 2^5$	0.72	39.5	0.22	56.4
$Arb^{hard}(6,6)$	$ack6 : 2^6, \ldots, ack1 : 2^1$	33.7	901.1	11.4	1157.0
$Arb^{soft}(4,2)$	$sr4 : 2^4, \ldots, sr1 : 2^1$	0.005	2.0	0.001	2.3
$Arb^{soft}(5,3)$	$sr5 : 2^5, \ldots, sr1 : 2^1$	0.3	10.9	0.06	15.7
$MinePump$	MPV1	0.003	2.5	0.001	3.0
$MinePump$	MPV2	0.004	2.6	0.001	3.1
$MinePump$	MPV3	0.003	2.6	0.001	3.1

6 Discussion

We have presented a technique and a tool for guided synthesis of controllers from hard and soft requirements. The requirements are written as QDDC formulas which specify complex past time properties. Hard requirements are formulas which must remain invariantly true and soft requirements must be kept true "as much as possible" by the controller. We have explored the use of such soft requirements for obtaining high quality controllers. Soft requirements allow robust controllers to be synthesized,

Table 5. Comparison of Synthesis in Acacia+, BoSy and DCSynth, interms of controller computation time and memory and number of states of the controller automaton.

Hard Requirement	Acacia+		BoSy		DCSynth	
	time(Sec)	Memory / States	time(Sec)	Memory / States	time(Sec)	Memory / States
$Arb^{hard}(4,4)$	0.4	29.8/ 55	0.75	-/4	0.08	9.1/ 50
$Arb^{hard}(5,5)$	11.4	71.9/ 293	14.5	-/8	5.03	28.1/ 432
$Arb^{hard}(6,6)$	TO[a]	-	TO	-	80	1053.0/ 4802
$Arb^{tok}(7)$	9.65	39.1/ 57	TO	-	0.3	7.3/ 7
$Arb^{tok}(8)$	46.44	77.9/ 73	TO	-	2.2	16.2/ 8
$Arb^{tok}(10)$	NC[b]	-	TO	-	152	82.0/ 10
MinePump	NC	-	TO	-	0.06	50/ 32

[a] TO=timeout(DCSynth and Acacia+ 3600secs, BoSy 600secs)
[b] NC=synthesis inconclusive

and also impact the latencies of the controller. The tool DCSynth implements the proposed technique.

We have also presented the use of SSDFA for efficient guided synthesis. Case studies and Experimental results show that the approach is useful and efficient. For example, a controller for the minepump case study presented here could be synthesized by DCSynth where as both Acacia+ (based on fully symbolic BDD) and BoSy (based on bounded synthesis technique) failed to compute a controller. Similar results exist for several other case studies and a comparison is presented in Table 5. The main reason for the efficiency of DCSynth is eager minimization of SSDFA at all stages of synthesis. Thus, our synthesis using SSDFA provides a novel addition to existing state of the art techniques. At the same time, fully symbolic DFAs and bounded synthesis are expected to scale up in a better way and bounded synthesis often computes controllers with fewer states. Perhaps, best results would require combining these techniques.

We have used past behaviour to guide the locally optimal choice of output in our guided synthesis. We have shown that this is useful. However, completely disregarding future is also restrictive. We are planning to extend DCSynth with bounded horizon planning and other game tree search techniques to take into account both past and future. We are also planning to extend synthesis from QDDC to more expressive GR1[DC] by combining our method with well known GR1 synthesis algorithm. Here GR1[DC] is a fragment of CTL*[DC] defined by Pandya [16] which can express all ω-regular properties.

References

1. Shaull Almagor, Udi Boker, and Orna Kupferman. Formally reasoning about quality. *J. ACM*, 63(3):24:1–24:56, 2016.
2. Roderick Bloem, Krishnendu Chatterjee, Karin Greimel, Thomas A. Henzinger, Georg Hofferek, Barbara Jobstmann, Bettina Könighofer, and Robert Könighofer. Synthesizing robust systems. *Acta Inf.*, 51(3-4):193–220, 2014.
3. Roderick Bloem, Rüdiger Ehlers, Swen Jacobs, and Robert Könighofer. How to handle assumptions in synthesis. In Krishnendu Chatterjee, Rüdiger Ehlers, and Susmit Jha, editors, *Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014.*, volume 157 of *EPTCS*, pages 34–50, 2014.
4. Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. Shield synthesis: - runtime enforcement for reactive systems. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 533–548. Springer, 2015.
5. Aaron Bohy, Véronique Bruyère, Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. Acacia+, a tool for LTL synthesis. In P. Madhusudan and Sanjit A. Seshia, editors, *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *Lecture Notes in Computer Science*, pages 652–657. Springer, 2012.
6. Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Inf. Process. Lett.*, 40(5):269–276, 1991.
7. Peter Faymonville, Bernd Finkbeiner, and Leander Tentrup. Bosy: An experimentation framework for bounded synthesis. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 325–332. Springer, 2017.
8. Bernd Finkbeiner and Sven Schewe. Bounded synthesis. *STTT*, 15(5-6):519–539, 2013.
9. Martin Fränzle. Synthesizing controllers from duration calculus. In Bengt Jonsson and Joachim Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 168–187, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.
10. Martin Fränzle1 and Markus Müller-Olm. *Compilation and Synthesis for Real-Time Embedded Controllers*, pages 256–287. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.
11. Swen Jacobs, Nicolas Basset, Roderick Bloem, Romain Brenguier, Maximilien Colange, Peter Faymonville, Bernd Finkbeiner, Ayrat Khalimov, Felix Klein, Thibaud Michaud, Guillermo A. Pérez, Jean-François Raskin, Ocan Sankur, and Leander Tentrup. The 4th reactive synthesis competition (SYNTCOMP 2017): Benchmarks, participants & results. *CoRR*, abs/1711.11439, 2017.
12. Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual.* BRICS, Department of Computer Science, Aarhus University, January 2001. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3.
13. Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. MONA implementation secrets. *International Journal of Foundations of Computer Science*, 13(4):571–586, 2002. World Scientific Publishing Company. Earlier version in Proc. 5th Inter-

national Conference on Implementation and Application of Automata, CIAA '00, Springer-Verlag LNCS vol. 2088.
14. Zhiming Liu. Specification and verification in the duration calculus. 1996.
15. Raj Mohan Matteplackel, Paritosh K. Pandya, and Amol Wakankar. Formalizing timing diagram requirements in discrete duration calulus.
16. Paritosh K. Pandya. Model checking CTL*[DC]. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2031 of *Lecture Notes in Computer Science*, pages 559–573. Springer, 2001.
17. Paritosh K Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using dcvalid. In *RTTOOLS 2001 Workshop (affiliated with CONCUR 2001)*. Aalborg University, 2001., 2001.
18. Paritosh K. Pandya. The saga of synchronous bus arbiter: On model checking quantitative timing properties of synchronous programs. *Electr. Notes Theor. Comput. Sci.*, 65(5):110–124, 2002.
19. Paritosh K. Pandya. Finding extremal models of discrete duration calculus formulae using symbolic search. *Electr. Notes Theor. Comput. Sci.*, 128(6):247–262, 2005.
20. Amol Wakankar, Paritosh K. Pandya, and Raj Mohan Matteplackel. DCSYNTH: guided reactive synthesis with soft requirements for robust controller and shield synthesis. *CoRR*, abs/1711.01823, 2017.
21. Amol Wakankar, Paritosh K. Pandya, and Raj Mohan Matteplackel. *DC-Synth Version 1.0*. STCS, TIFR, Mumbai, January 2018. Available from `http://www.tcs.tifr.res.in/~pandya/dcsynth/dcsynth.html`.
22. Chaochen Zhou and Michael Hansen. *Duration Calculus A Formal Approach to Real-Time Systems*. Springer, 2004.

A Pseudo Code for DCSynth

Internally, the monitor automaton, MPNC and LODC are all stored as SSDFA. The psuedocode of our synthesis algorithm, introduced in Section 3, is given below.

Algorithm 2 ***Synthesize:***

```
    **Input**: $S = (I, O, D^h, \wedge_{i=1}^{i=k}(w_i \Leftrightarrow D_i^s), \langle P_1 : \theta_1, \cdots, P_l : \theta_l\rangle)$
    **Output**: Controller for S.
    1. $A^{mon}$=DCVALID($D^{hard} \wedge D^{Ind}$)
            //Generates language equivalent safety automaton
    2. G = ComputeWINNING($A^{mon}$, I, O, W)
          IF initial state of $A^{mpnc}$ is NOT in G THEN
             S is unrealizable, generate a counter example tree
          ELSE
             Specification is realizable, GOTO step 3.
    3. $A^{mpnc}$ = ComputeMPNC($A^{mon}$, G)
    4. $A^{lodc}$ = ComputeLODC($A^{mpnc}$, $\langle P_1 : \theta_1, \cdots, P_l : \theta_l\rangle$)
            //Determinizes the MPNC with respect to Soft Requirements.
    5. Encode $A^{lodc}$ in an implementation language.
```

The procedure *Synthesize* starts by constructing the monitor automaton A^{mon} based on the procedure implemented in a the tool DCVALID [17]. The tool DCVALID translates the QDDC specification into MONA and generates the equivalent DFA using MONA library.

The procedure for ComputeWINNING is given in Algorithm 1. We implementation of ComputeMPNC and ComputeLODC are described in section 5, with a few implementation specific optimizations. This section also describes the optimization based on SSDFA to get an efficient algorithm.

Algorithm 3 ***ComputeWINNING:***

```
    **Input**: $A^{mon}$, I, O, W
    **Output**: $A^{mpnc}$.
    S = set of states in $A^{mon}$,
    F = set of accepting states in $A^{mon}$
    $\delta$: S $\times$ $2^{(I \cup O \cup W)}$ $\rightarrow$ S be the trasition function in $A^{mon}$.
    $\mathcal{V}$: S $\rightarrow$ {1, 0} be a value function over S
    SET G = F
    DO {
          SET Pre_G = G
        FOR each s $\in$ G DO{
```

```
        IF $C_{step}(s,\ G) = 0$ then
            $G = G - s$
    }
}WHILE ($Pre\_G \neq G$ )
RETURN G
```

Algorithm 4 ***GenLODC:***

```
Input: $A^{mpnc}$, I, O, $\langle P_1 : \theta_1, \cdots, P_l : \theta_l \rangle$,
Output: $A^{lodc}$.
S = set of states in $A^{mpnc}$
SET $A^{lodc}$ = NULL FOR every state $s \in S$ DO {
    FOR every valuation i of I DO {
        (o, t) = ValSoft(i, $A^{mpnc}$, $\langle P_1 : \theta_1, \cdots, P_l : \theta_l \rangle$)
        Add trasition from state ‘s‘ to ‘t‘ on valuation (i, o) in $A^{lodc}$
    }
        RETURN $A^{lodc}$ after minimization.
}
```

B More on Minepump Specification

B.1 Minepump Specification for DCSynth

```
BEGIN QDDCSYNTH MinePump
  INTERFACESPEC
    HH2Op: INPUT
    HCH4p: INPUT
    ALARMp: OUTPUT MONITOR x
    PUMPONp: OUTPUT MONITOR x
    YHCH4p: OUTPUT MONITOR x;
  SOFTREQS
    ((!YHCH4p)|(!PUMPONp))>>(PUMPONp) ;
  AUXVARS
    DH2O
  ;
  CONSTANTS
    – delta response time of PUMP and ALARMS after trigger
  delta = 1, w = 8, epsilon=2 , zeta=10, kappa=1
  ;
  DEFINE
  – Alarm control
  define alarm1(HH2O, DH2O, HCH4, ALARM, PUMPON) as
     HH2O =delta=> ALARM ;
  define alarm2(HH2O, DH2O, HCH4, ALARM, PUMPON) as
     HCH4 =delta=> ALARM ;
  define alarm3(HH2O, DH2O, HCH4, ALARM, PUMPON) as
      !HCH4 && !HH2O  =delta=> !ALARM ;
  – Water seepage Assumptions
  define water1(HH2O, DH2O, HCH4, ALARM, PUMPON) as
     [] ( [[ DH2O => HH2O ]] ) ;
  define water2(HH2O, DH2O, HCH4, ALARM, PUMPON) as
     HH2O <=w= ! DH2O ;
  – Pump capacity assumption
  define pumpcap1(HH2O,DH2O,HCH4,ALARM,PUMPON) as
     PUMPON =epsilon=> !HH2O ;
  – Methane Release assumptions
  define methane1(HH2O,DH2O,HCH4,ALARM,PUMPON) as
     [] ( [HCH4]∧[!HCH4]∧<HCH4> => slen > zeta ) ;
  define methane2(HH2O,DH2O,HCH4,ALARM,PUMPON) as
     [] ( [[HCH4]] => slen < kappa ) ;
  – Initial condition assumption
  define initdry(HH2O, DH2O, HCH4, ALARM, PUMPON) as
     <!HH2O> ∧ true ;
  – safety condition
  define safe(HH2O, DH2O, HCH4, ALARM, PUMPON) as
    [[!DH2O && ( (HCH4 || !HH2O) => !PUMPON)]];
  define plant(HH2O, DH2O, HCH4, ALARM, PUMPON) as
    initdry(HH2O, DH2O, HCH4, ALARM, PUMPON) &&
    water1(HH2O, DH2O, HCH4, ALARM, PUMPON) &&
    water2(HH2O, DH2O, HCH4, ALARM, PUMPON) &&
    pumpcap1(HH2O, DH2O, HCH4, ALARM, PUMPON) &&
    methane1(HH2O, DH2O, HCH4, ALARM, PUMPON) &&
    methane2(HH2O, DH2O, HCH4, ALARM, PUMPON);
  define req(HH2O, DH2O, HCH4, ALARM, PUMPON) as
  infer MPsyn as
  INDICATORS
     YHCH4p : (slen=2 && <><HCH4p>)
  ;
  ASSUME
    plant(HH2Op, DH2O, HCH4p, ALARMp, PUMPONp)
  ;
  REQUIRES
    req(HH2Op, DH2O, HCH4p, ALARMp, PUMPONp)
  ;
  SYNTHESIZE
  SynthG MPsyn
  END QDDCSYNTH
```

Fig. 2. DCSynth input file for minepump specification.

B.2 Simulation of Synthesized Minepump Controllers

The controllers are encoded as Lustre specification and Lustre V4 tools are used for simulation. The example simulation for these three variants of minepump with soft requirements MPV1, MPV2 and MPV3 are shown in figures 3, 4 and 5 respectively.

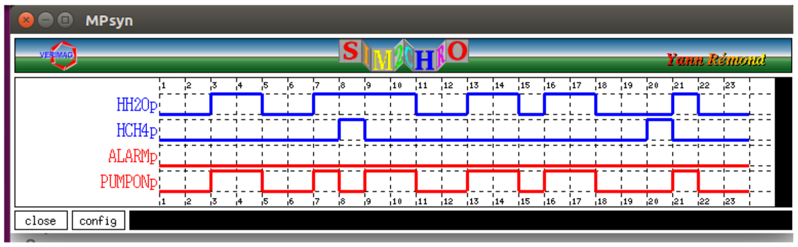

Fig. 3. Simulation of minepump controller with soft requirement MPV1

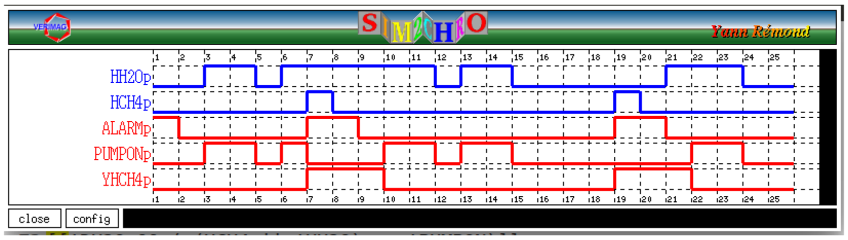

Fig. 4. Simulation of minepump controller with soft requirement MPV2

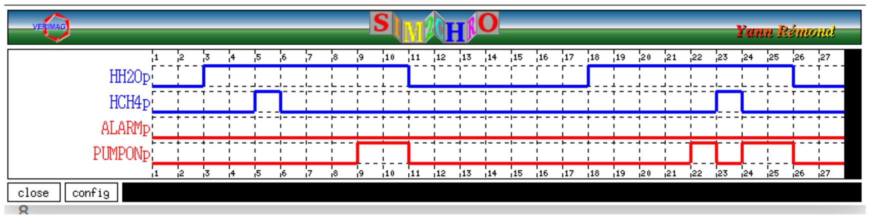

Fig. 5. Simulation of minepump controller with soft requirement MPV3

C More Case Studies

In this section we present 3 more case studies:

1. n-cell synchronous bus arbiter,
2. industrial pump/valve controller, and
3. alarm annunciation system.

C.1 Arbiter

An n-cell synchronous bus arbiter is a circuit with $req_1, \ldots, req_n$ as inputs and $ack_1, \ldots, ack_n$ as the corresponding outputs, and arbitrates among a subset of requests at each cycle by setting one of the acknowledgments (ack_i's) *true*. Hard requirements on the arbiter include the following three invariant properties.

$$\begin{array}{l} Mutex = [[\ \wedge_{i \neq j}\ \neg(ack_i \wedge ack_j)\ ]] \\ NoLoss = [[\ (\vee_i req_i) \Rightarrow (\vee_j ack_j)\ ]] \\ NoSpurious = [[\ \ \wedge_i\ \ (ack_i \Rightarrow req_i)\ \ ]] \\ ARBINV = Mutex \wedge NoLoss \wedge NoSpurious. \end{array} \tag{1}$$

Thus, $Mutex$ gives mutual exclusion of acknowledgments, $NoLoss$ states that if there is at least one request then there must be an acknowledgment and $Nospurious$ states that acknowledgment is only given to a requesting cell.

In the literature various arbitration schemes for the arbiter have been proposed, here we consider the following two schemes.

– k-cycle response time: let $Resp(req, ack, k)$ denote that if request has been high for last k cycles there must have been at least one acknowledgment in the last k cycles. Let $ArbResp(n, k)$ state that for each cell i and for all observation intervals the formula $Resp(req_i, ack_i, k)$ holds.

$$\begin{array}{l} Resp(req, ack, k) = (([[req]]\ \&\&\ (slen = k)) => true\hat{}\ < ack >\ \hat{}true) \\ ArbResp(n, k) = \wedge_{1 \leq i \leq n}\ [](Resp(req_i, ack_i, k)) \\ ARBHARD(n, k) = ARBINV \wedge ArbResp(n, k) \end{array} \tag{2}$$

Then $Arb^{hard}(n, k)$ is the following k-cycle response time DCSynth specification.

$$Arb^{hard}(n, k) = (\{req_1, \ldots, req_n\}, \{ack_1, \ldots, ack_n\}, ARBHARD(n, k), \langle\rangle, \langle\rangle) \tag{3}$$

- k-cycle response time as soft requirement: we specify the requirement of response in k cycles as a *soft requirement* as below.

$$\begin{aligned} Arb^{soft}(n,k) = \; & (\{req_1, \ldots, req_n\}, \{ack_1, \ldots, ack_n\}, ARBINV, \langle \; \rangle, \\ & \langle Resp(req_n, ack_n, k) : n, \ldots, Resp(req_1, ack_1, k) : 1 \rangle \;) \end{aligned} \tag{4}$$

- Token ring arbitration: a token is circulated among the masters in a round robin fashion. The token is modeled using the variables tok_i's $(1 \leq i \leq n)$. Exactly one of tok_i's will hold at any time and if tok_i is *true* then we mean that $master_i$ holds the token. The arbiter asserts ack_i whenever req_i and tok_i are *true*, i. e. priority is accorded to the request of the master which holds the token.

$$\begin{aligned} & TokInit(n) =< tok_1 \;\&\&\; (\wedge_{2 \leq i \leq n} !tok_i) > \;\hat{}\; true \\ & TokCirculate(n) = [](\wedge_{1 \leq i \leq n}(tok_i \;\hat{}\; (slen = 1) <=> \\ & \qquad\qquad\qquad\qquad\qquad (slen = 1) \;\hat{}\; tok_{i\%n+1})) \\ & TokResp(n) = \wedge_{1 \leq i \leq n}[[(req_i \;\&\&\; tok_i) => ack_i]] \\ & Token(n) = TokInit(n) \wedge TokCirculate(n) \wedge TokResp(n). \end{aligned} \tag{5}$$

  Let $ARBTOKEN(n) = ARBINV \wedge Token(n)$. Then $Arb^{tok}(n)$ is the following DCSynth specification.

$$Arb^{tok}(n) = (\{req_1, \ldots, req_n\}, \{ack_1, \ldots, ack_n\}, ARBTOKEN(n), \langle\rangle, \langle\rangle) \tag{6}$$

The following example of a 2 cell arbier shows the synthesis for a given arbiter specification.

Examples: Synthesis of 2 Cell Arbiter with Soft Requirements giving high priority to lower numbered request: Figure 6 gives the safety monitor automaton for 2-cell arbiter for following specification

$$S = (\langle req_1, req_2 \rangle, \langle ack_1, ack_2 \rangle, ARBINV \wedge Resp^{spec}(2,2), \langle\rangle, \langle ack_1, ack_2 \rangle).$$

In this example, there are no witness variables. Figure **??** show the symbolic MTBDD representation of this monitor automaton. Each transition is labeled by 4 bit vector giving values of $req_1, req_2, ack_1, ack_2$.

Fig. 7 gives the MPNC automaton for the 2-cell arbiter computed from the safety monitor automaton of Fig. 6.

In the example, the soft requirements are $\langle ack_1, ack_2 \rangle$ which give ack_1 priority over ack_2, we obtain the pruned LODC controller automaton of Fig. 8 from the MPNC of Fig. 7. Note that we minimize the automaton at each step.

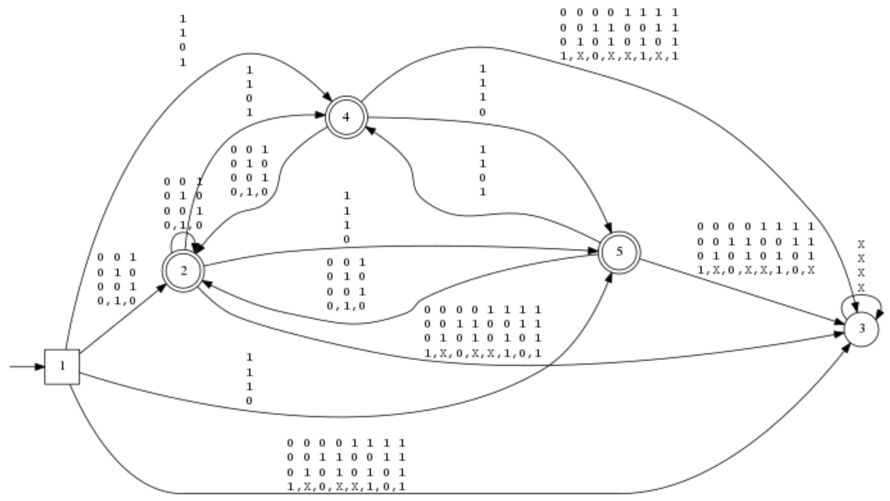
1
2
3
4
5
1 1 0 1
1 1 0 1
0 0 1 / 0 1 0 / 0 0 1 / 0,1,0
0 0 1 / 0 1 0 / 0 0 1 / 0,1,0
0 0 1 / 0 1 0 / 0 0 1 / 0,1,0
0 0 1 / 0 1 0 / 0 0 1 / 0,1,0
1 1 1 0
1 1 1 0
1 1 0 1
1 1 1 0
0 0 0 0 1 1 1 1 / 0 0 1 1 0 0 1 1 / 0 1 0 1 0 1 0 1 / 1,X,0,X,X,1,X,1
0 0 0 0 1 1 1 1 / 0 0 1 1 0 0 1 1 / 0 1 0 1 0 1 0 1 / 1,X,0,X,X,1,0,X
0 0 0 0 1 1 1 1 / 0 0 1 1 0 0 1 1 / 0 1 0 1 0 1 0 1 / 1,X,0,X,X,1,0,1
0 0 0 0 1 1 1 1 / 0 0 1 1 0 0 1 1 / 0 1 0 1 0 1 0 1 / 1,X,0,X,X,1,0,1
X X X X

Fig. 6. Safety Monitor Automaton: 2 Cell Arbiter

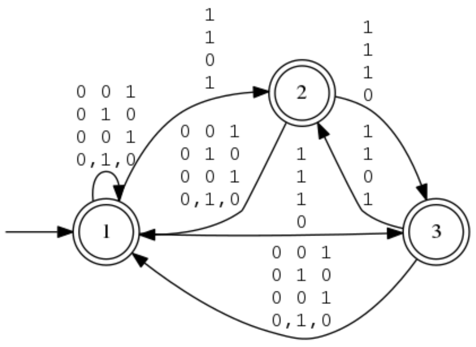

Fig. 7. MPNC : 2 Cell Arbiter

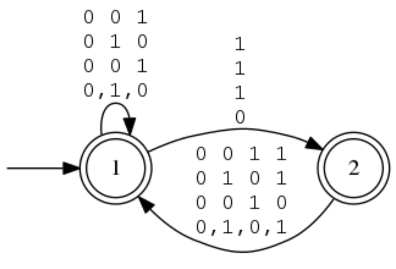

Fig. 8. LODC: 2 Cell Arbiter

C.2 Industrial Pump/Valve Controller

We now present the case study consisting of the control of equipments (e.g. Pump and Valve) related to a process control system in a nuclear reactor.

The texual specification of the requirements are as follows:

- Two types of equipments namely Valve and Pump are controlled:
  - Any Valve can have two positions: OPEN, CLOSE. Control Logic can specify two types of commands for valves: OPEN command to OPEN the valve and CLOSE command for closing.
  - Any Pump can have two states: RUNNING, STOPPED. Control Logic can specify two types of commands for Pumps: START command for running the pump and STOP command for stopping the Pump. A Pump can stop without any command (manual / automatic) from the control system. This can arise, when equipment fails. This change of state of the pump will be termed as SELF STOP.
- Equipment mentioned above can be in one of the following two modes of operation:
  - Auto : In this mode, equipment gets controlled through specified control logic.
  - Manual: In this mode, equipment is only controlled through operator command and automatic logic will have no affect on the equipment status, even if specified conditions for the logic are met. Any equipment can be individually set in one of the two modes.
- A discrete number of identified stable states of the process system are defined. In each of these states, equipment being controlled is taken to a specific state.
- Process system state can be changed either through manual command by the operator or through auto logic based on plant parameters. Each state has an entry action and the sustained action.
- The specification for entry and sustained action for each state is given in table 6. Initial state is NORMAL state and the following are the identifiers for the equipment being controlled:
  - Identifier of valve being controlled- V1, V3
  - Identifier of pump being controlled-P1
  - This process system has 5 states OFF, OFFH, ONH, START, NORMAL
  - Control logic uses two digital signals C16, C17
  - Control logic uses status of pump P2, which gets controlled by Process System-2

Result of Synthesis The industrial example gave a use case of soft requirements to generate a meaningful controller. Soft requirements allowed us to specify the default values of the output in concise manner. Without specification of soft requirements the sythesis algorithm was generating a trivial controller which was setting all the command for both the equipments which was not useful. Then we added the soft requirements to specify that the commands should not be generated if possible. Then the controller 10 state controller was produced to generated a command only when is it required.

The soft requirements also allowed us to uncover the source of inconsitency in the requirement (leading to unrealizability). Few of the issues resolved during the case study are as follows:

- Specification of default values: The requirements in state machine formalism were specified with entry action and sustained action, but the values of outputs when none of the conditions specified occurs, were not specified. Incomplete specification causes tool to generate a trivial controller that make all output commands true in every cycle. Synthesized controller meets all the requirements but it is not practically useful. This is due to the fact that default values of the outputs were not explicitly specified in the original requirements Soft requirements based specification was used to specify the default values to all the output variables, which produce useful controllers.
- Priority of transitions: State chart specification contains transitions from one state to another state based on specified conditions. Specified conditions are not always mutually exclusive, so behaviour of the state chart was undefined when multiple transitions are possible from one state. The resolution of this required the transition to be given priorities. The highest priority transition will be taken up whenever multiple transitions are possible. After discussion, it was clarified that priority assigned to interlock logic is in the same order as it appears in the state transition table provided in the requirements. Interlock logic, which appears first, is given highest priority.
- Persistence of state: The persistence condition for state was not mentioned explicitly in the requirements. It was assumed as per state chart semantics that the current state would persist, if there is no transition condition enabled.
- Strong Vs Weak transitions: The transition conditions were specified but whether it is a strong transition (transition taken immediately and actions of next state is executed in current cycle itself) or weak transition (in present cycle the action of the current state will be

executed and next cycle onward the actions of next cycle shall be executed) were not specified. The ambiguity was resolved by assuming the strong transition and Requirements were modelled with this assumption.

C.3 Alarm Annunciation System

The next case study is Alarm Annunciation System(AAS) used in a process control system for annunciation for various alarms in the control room. The Alarm Annunciation involves the standard *Automatic Ring-Back Sequence* for all the digital inputs meant for alarm annunciation and provide the necessary outputs. The specification of Automatic Ring-Back Sequence is given in Table 7. All digital inputs representing alarm conditions are scanned periodically.

As shown in the Table 7 that Automatic Ring-Back Sequence specification takes the alarm signal as input. The high value of signal represents the alarm state, otherwise the signal is said to be in normal state. Other inputs are Acknowlegment, Reset and Silence inputs, which are controller by the operator. There are three output elements: Lamp, Normal Hooter and Ringback Hooter. There is a Lamp corresponding to each alarm signal, whereas Hooters are comman to all alarm signals. Lamp can either be *Fast Flashing, Slow Flashing, Steady On or Off* states. We have encoded the requirements in DCSynth to synthesize the controller. The Silence input can be used by the operator to switch off Hooters.

Result of Synthesis : As discussed in the previous case study soft requirements helped in specification of requirements concisely. The controller synthesized has 8 states. We could simulate the controller and verified the correctness.

Table 6. Requirements for each state

State	**Entry Action**	**Sustained Action**
OFF	1) CLOSE command for V1, V3 2) STOP command for P1 3) V1, V3 and P1 are removed from AUTO mode	Nil
OFFH	1) CLOSE command for V1, V3 2) Pump P1 is set to AUTO mode, if transfer to this state was not made on self stopping of Pump P1	Change state to START if P2 stops **Constraints**: When value of C17 is TRUE, changing of state from OFFH to ONH shall not be permitted. Under this condition, if there is command for change of state to NORMAL, state shall be switched to START
ONH	1) OPEN command for V1, V3 2) Valves V1, V3 are set to AUTO mode 3) STOP command for P1 4) Pump P1 is set to AUTO mode, if transfer to this state was not made on self stopping of Pump P1	1) Change state to NORMAL if Pump P2 self stops. This change of state takes place only if Pump P1 is set in AUTO mode 2) Change state to OFFH if C17 is set to TRUE. This change of state shall be disabled, if C16 is also set to TRUE along with C17. Mentioned change of state shall be enabled again only, when both C16 and C17 becomes FALSE **Constraints:** Changing of state from ONH to START shall be disabled. When such command is issued, state shall be changed to NORMAL
START	1) START command for Pump P1	1) Change state to OFF if P1 self stops. 2) CLOSE V1 and V3, when value of signal C17 is set to TRUE. Above mentioned command shall be disabled, if C16 is also set to TRUE along with C17. Mentioned change of state shall be enabled again only, when both C16 and C17 becomes FALSE
NORMAL	1) START command for P1 2) OPEN command for V1, V3 3) Valves V1, V3 are set to AUTO mode (only when change to this state happens from OFF)	1) Change state to ONH if P1 self stops 2) Change state to OFFH, when signal C17 is set to TRUE. This change of state shall be disabled, if C16 is also set to TRUE along with C17. Mentioned change of state shall be enabled again only, when both C16 and C17 becomes FALSE

Table 7. Automatic Ring-Back Sequence for Alarm Annunciation

Input	**Lamp Output**	**Audio Output**
Normal to Alarm	Fast Flashing	Normal Alarm Hooter On
Acknowledged	Lamp On	Normal Alarm Hooter Off
Alarm to Normal	Slow Flashing	Ringback Hooter On
Reset	Lamp Off	Ringback Hooter Off