Specification and Optimal Reactive Synthesis of Run-time Enforcement Shields

Paritosh K. Pandya
Tata Institute of Fundamental Research
Mumbai 400005, India
pandya@tifr.res.in

Amol Wakankar
Homi Bhabha National Institute
Bhabha Atomic Research Centre
Mumbai 400085, India
amolk@barc.gov.in

A system with sporadic errors (SSE) is a controller which produces high quality output but it may occasionally violate a critical requirement $REQ(I,O)$. A run-time enforcement shield is a controller which takes (I,O) (coming from SSE) as its input, and it produces a corrected output O' which guarantees the invariance of requirement $REQ(I,O')$. Moreover, the output sequence O' must deviate from O "as little as possible" to maintain the quality. In this paper, we give a method for logical specification of shields using formulas of logic *Q*uantified Discrete Duration Calculus(QDDC). The specification consists of a correctness requirement REQ as well as a hard deviation constraint HDC which must both be mandatorily and invariantly satisfied by the shield. Moreover, we also use quantitative optimization to give a shield which minimizes the expected value of cumulative deviation in an H-optimal fashion. We show how tool DCSynth implementing soft requirement guided synthesis can be used for automatic synthesis of shields from a given specification. Next, we give logical formulas specifying several notions of shields including the k-Stabilizing shield of Bloem *et al.* [2, 9] as well as the Burst-error shield of Wu *et al.* [21], and a new e,d-shield. Shields can be automatically synthesized for all these specifications using the tool DCSynth. We give experimental results showing the performance of our shield synthesis tool in relation to previous work. We also compare the performance of the shields synthesized under diverse hard deviation constraints in terms of their expected deviation and the worst case burst-deviation latency.

1 Introduction

A system with sporadic errors (SSE) is a controller which produces high quality desirable output for any given input but it may sporadically violate a critical system requirement $REQ(I,O)$, where I and O are the set of input and output propositions. Many manually designed controllers have this character, as they embody designer's unspecified optimizations, however they may have obscure design errors. A run-time enforcement shield for a specified critical requirement $REQ(I,O)$ is a controller (Mealy machine) which receives both input and output (I,O) generated by SSE. The shield produces a modified output O' which is guaranteed to invariantly meet the critical requirement $REQ(I,O')$ (correct-by-construction). Moreover, in each run, the shield output O' must deviate from the SSE output O "as little as possible", to maintain the quality. This allows the shield to benefit from system designer's optimizations without having to formally specify these or to handle these in the synthesis. See Figure 2.

A central issue in designing run-time enforcement shields is the underlying notion of "deviating as little as possible" from the SSE output. There are several different notions explored in the literature [2, 9, 21, 20]. In their pioneering paper, Bloem *et al.* [2] proposed the notion of k-stabilizing shield which may deviate for at most k cycles continuously under suitable assumptions. If assumptions are not met the shield may deviate arbitrarily. This was proposed as a hard requirement which must be mandatorily satisfied by the shield in any behaviour. We call such constraints as **hard deviation constraints**.

J. Leroux and J.-F. Raskin (Eds.): Tenth International Symposium
on Games, Automata, Logics, and Formal Verification (GandALF'19).
EPTCS 305, 2019, pp. 91–106, doi:10.4204/EPTCS.305.7

Konighofer *et al* [9] have proposed some variants of the k-stabilizing shield requirement with and without fail safe state, which are also hard deviation constraints. Specific shield synthesis algorithms have been developed for each of these constraints.

As our first main contribution, we propose a logical specification notation for **hard deviation constraints** using the formulas of an interval temporal logic QDDC. This logic allows us to succinctly and modularly specify regular properties [13, 11, 12]. With its counting constructs and interval based modalities, it can be used to conveniently specify both the correctness requirement $REQ(I,O)$ as well as the hard deviation constraint HDC.

Criticizing the inability of k-stabilizing shields in handling burst errors, Wu *et al.* [21, 20] proposed a burst-error shield which enforces the invariance of the correctness requirement, and it locally minimizes the measure of deviation between SSE output O and the shield output O', at each step. An algorithm for the synthesis of such shields was given. We call such a shield as *locally deviation minimizing*.

In this paper, as our second main contribution, we generalize the Wu technique to minimize the cumulative deviation more globally. An H-optimal shield which minimizes at each point the expected value of cumulative deviation in next H-steps of shield execution is computed. The cumulative deviation is averaged over all possible H length inputs to arrive at the optimal estimate. A well known value iteration algorithm [1, 16] for optimal policy synthesis of Markov Decision Processes allows us to compute such a shield. We call such a shield as *H-optimally deviation minimizing*. This is a powerful optimization and in the paper we experimentally show its significant impact on performance of the shield. It may be noted that Wu's burst-error shield is obtained by selecting $H = 0$.

Finally, we propose a uniform method for synthesizing a run-time enforcement shield from given logical specification (REQ, HDC) and a horizon value (natural number) H. The resulting shield invariantly meets the correctness requirement REQ as well as the hard deviation constraint HDC. Moreover, the shield is H-optimally deviation minimizing. The shield synthesis is carried out by using the soft requirement guided controller synthesis tool DCSynth [18]. This tool allows synthesis of H-optimal controllers from specified hard and soft QDDC requirements.

Using the proposed formalism, in the paper, we formulate several diverse notions of shields. These include a logical specification of Bloem's k-stabilizing shield and Wu's burst-error shield, as well as a new notion of e,d-shield. A uniform synthesis method using the tool DCSynth can be applied to obtain the corresponding run-time enforcement shields. It is notable that tool DCSynth uses an efficient BDD-based semi-symbolic representation of automata/controllers with aggressive minimization. This allows the tool to scale better and to produce smaller sized shields. In the paper, we give an experimental evaluation of the performance of our DCSynth tool and compare it with some previously reported studies in the literature.

With the ability to formulate shields with diverse hard deviation constraints, it is natural to ask for a comparison of the **performance** of these shields. The performance must essentially measure the extent of deviation of the shield output from the SSE output. Towards this, we propose two measures of the shield performance.

- We compute the *probability of deviation* in long run. For this, we assume that the input to the shield is fully random, with each input variable value chosen independently of the past and each other. While simplistic, this does provide some indication of the shield's effectiveness in average.

- We measure the *worst case burst-deviation latency*. This gives the maximum number of consecutive deviations possible in the worst case. (If unbounded, we report ∞). A model checking technique implemented in a tool CTLDC [14] allows us to compute this worst case latency.

Tool DCSynth provides facilities for the computation of each of these performance measures for a synthesized shield. The reader may refer to the original papers on DCSynth [18, 15] for details of techniques by which such performance can be measured. In this paper, we synthesize shields with different hard deviation constraints and we provide a comparison of the performance of these shields. This allows us to draw some preliminary conclusions. Clearly, much wider experimentation is needed for firmer insight.

The rest of the paper is organized as follows. Section 2.1 describes the syntax and semantics of the logic QDDC. Section 2.3 gives the syntax of DCSynth specification and brief outline of the synthesis method. Section 3 describes the various logical notions of shield specification. Section 4 describes metrics to evaluate the shield performance and corresponding experimental results. In Section 5, we conclude the paper with discussion and related work.

2 Preliminaries

We provide a brief overview of logic QDDC as well as the soft requirement guided H-optimal controller synthesis method implemented in tool DCSynth. This method and tool is applied to the problem of run-time enforcement shield synthesis in this paper. The reader may refer to the original paper [18] for further details of these preliminaries.

2.1 Quantified Discrete Duration Calculus (QDDC) Logic

Let PV be a finite non-empty set of propositional variables. Let σ a non-empty finite word over the alphabet 2^{PV}. It has the form $\sigma = P_0 \cdots P_n$ where $P_i \subseteq PV$ for each $i \in \{0,\ldots,n\}$. Let $len(\sigma) = n+1$, $dom(\sigma) = \{0,\ldots,n\}$, $\sigma[i,j] = P_i \cdots P_j$ and $\sigma[i] = P_i$.

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

$$\varphi := false \mid true \mid p \in PV \mid !\varphi \mid \varphi \;\&\&\; \varphi \mid \varphi \;||\; \varphi$$

with $\&\&, ||, !$ denoting conjunction, dis-junction and negation, respectively. Operators such as $\Rightarrow$ and $\Leftrightarrow$ are defined as usual. Let $\Omega(PV)$ be the set of all propositional formulas over variables PV. Let $i \in dom(\sigma)$. Then the satisfaction of propositional formula φ at point i, denoted $\sigma, i \models \varphi$ is defined as usual and omitted here for brevity.

The syntax of a QDDC formula over variables PV is given by:

$$\begin{array}{rl} D := & \langle\varphi\rangle \mid [\varphi] \mid [[\varphi]] \mid D \,\hat{}\, D \mid !D \mid D \mid\mid D \mid D \;\&\&\; D \\ & ex\ p.\ D \mid all\ p.\ D \mid slen \bowtie c \mid scount\ \varphi \bowtie c \mid sdur\ \varphi \bowtie c \end{array}$$

where $\varphi \in \Omega(PV)$, $p \in PV$, $c \in \mathbb{N}$ 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 2^{PV}, let $[b,e] \in Intv(\sigma)$ be an interval. Then the satisfaction of a QDDC formula D written as $\sigma, [b,e] \models D$, is defined inductively as follows:

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

with Boolean combinations $!D$, $D_1 \;||\; D_2$ and $D_1 \;\&\&\; D_2$ defined in the expected way. We call word σ' a p-variant, $p \in PV$, of a word σ if $\forall i \in dom(\sigma), \forall q \neq p : q \in \sigma'[i] \Leftrightarrow q \in \sigma[i]$. Then $\sigma,[b,e] \models ex\ p.\ D$ iff $\sigma',[b,e] \models D$ for some p-variant σ' of σ; and $(all\ p.\ D) \Leftrightarrow (!ex\ p.\ !D)$.

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
r	0	0	1	0	0	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0
p	0	0	0	0	0	1	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1
q	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
$\phi_{until}(3)$	1	1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0

Figure 1: Example behaviour for $\phi_{until}(3)$

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

We also define the following derived constructs: $pt = \langle true \rangle$, $ext = !pt$, $\langle\rangle \boldsymbol{D} = true\hat{}D\hat{}true$, $[]\boldsymbol{D} = (!\langle\rangle!D)$ and $\boldsymbol{pref(D)} = !((!D)\hat{}true)$. Thus, $\sigma,[b,e] \models []D$ iff $\sigma,[b',e'] \models D$ for all sub-intervals $b \leq b' \leq e' \leq e$ and $\sigma,[b,e] \models pref(D)$ iff $\sigma,[b,e'] \models D$ for all prefix intervals $b \leq e' \leq e$.

Finally, we define $\sigma,i \models D$ iff $\sigma,[0,i] \models D$, and $\sigma \models D$ iff $\sigma,[0,\, len(\sigma)-1] \models D$. Let $L(D) = \{\sigma \mid \sigma \models D\}$, the set of behaviours accepted by D. Let D be valid, denoted $\models_{dc} D$, iff $L(D) = (2^{PV})^+$. Notice that $\sigma,i \models D$ denotes that the past of position i satisfies the formula D.

Example 1. *We give an example QDDC formula over propositions $\{p,q,r\}$ which specifies a typical recurrent reach-avoid behaviour required in many control systems. Intuitively, the formula $\varphi_{until}(n)$ holds at a position i in the behaviour if, since the previous occurrence of r, the proposition p persists till an occurrence of q. Moreover, q must occur within n time units from the last occurrence of r. For example, here r may denote entering of enemy air-space, p may denote that the UAV is invisible and q may denote that the target is reached. Let φ_3 abbreviate $\varphi_{until}(3)$. Figure 1 gives a possible behaviour σ where the last row gives the value of $\sigma,i \models \varphi_3$ for each position i.*

- *Until*(p,q,n): `((slen<(n)) && [[p]]) ||`
  `((((([p] || pt)^<q>) && slen<=n)^true).`
  The second disjunct holds for an interval $[b,e]$ provided q occurs at a position $b \leq j \leq e$ with $j \leq b+n$ and p persists from b to $j-1$. E.g. in Figure 1, $\sigma,[5,9] \models until(p,q,3)$ with $j=8$. The first disjunct holds for an interval $[b,e]$ provided $e-b<n$ and p holds throughout the interval. E.g. $\sigma,[11,12] \models until(p,q,3)$. Note that $\sigma,[2,4] \not\models until(p,q,3)$.

- *SinceLast*(p,D): `!(true^(<p>^((slen=1^[[!p]]) || pt) && !(D)))`
  This formula fails to hold at position i provided there is a previous (last) occurrence of p in the past of i, at say position $j \leq i$, and D does not hold for the interval $[j,i]$.

- *Let $\varphi_{until}(n)$ be the QDDC formula* `SinceLast(r,(Until(p,q,n)))`.
  Then, $\sigma,1 \models \varphi_3$ since there is no r at any position $j \leq 1$. Also, $\sigma,9 \models \varphi_3$ as, since the previous occurrence of r at position 5, the proposition p persists till 7 and q holds at 8 (with $8 \leq 5+3$). Note also that $\sigma,12 \models \varphi_3$ since the previous r occurs at 12 (with $12 < 12+3$) and $\sigma,[12,12] \models$ `[[ p ]]`. *Finally, $\sigma,4 \not\models \varphi_3$ as, since the previous r at position 4, neither does q occur in-between nor do we have $\sigma,[2,4] \models$* `[[ p ]]`. □

Theorem 2. *[13] For every formula D over variables PV we can construct a Deterministic Finite Automaton (DFA) $\mathscr{A}(D)$ over alphabet 2^{PV} such that $L(\mathscr{A}(D)) = L(D)$. We call $\mathscr{A}(D)$ a* formula automaton *for D or the monitor automaton for D.* □

A tool DCVALID implements this formula automaton construction in an efficient manner by internally using the tool MONA [8]. It gives *minimal, deterministic* automaton (DFA) for the formula D. We omit the details here. However, the reader may refer to several papers on QDDC for detailed description and examples of QDDC specifications as well as its model checking tool DCVALID [13, 11, 12].

In the rest of the paper we consider QDDC formulas and automata where variables $PV = I \cup O$ are partitioned into disjoint sets of input variables I and output variables O. Such a formula/automaton specifies a relation between inputs and outputs.

For technical convenience, we define a notion of *indicator variable* for a QDDC formula (regular property). The idea is that the indicator variable w witnesses the truth of a formula D at any point in execution. Thus, $Ind(D,w) = pref(EP(w) \Leftrightarrow D)$. Here, $\mathbf{EP}(\mathbf{w}) = (true\hat{}\langle w \rangle)$, i.e. in a behaviour σ and a position i, we have $\sigma, i \models EP(w)$ iff $w \in \sigma[i]$. If $\sigma \models Ind(D,w)$ then for for any i, we have $\sigma, i \models D$ iff $w \in \sigma[i]$. Thus variable w is true exactly at those positions where the past of the position satisfies D. These indicator variables can be used as auxiliary propositions in another formula using the notion of cascade composition $\ll$ defined below.

Definition 3 (Cascade Composition)**.** *Let $D_1, \ldots, D_k$ be QDDC formulas over input-output variables (I,O) and let $W = \{w_1, \ldots, w_k\}$ be the corresponding set of fresh indicator variables i.e. $(I \cup O) \cap W = \emptyset$. Let D be a formula over variables $(I \cup O \cup W)$. Then, the cascade composition $\ll$ and its equivalent QDDC formula are as follows:*

$$D \ll \langle Ind(D_1,w_1), \ldots, Ind(D_k,w_k) \rangle \quad = \quad D \wedge \bigwedge_{1 \leq i \leq k} pref(EP(w_i) \Leftrightarrow D_i)$$

This composition gives a formula over input-output variables $(I, O \cup W)$. □

Cascade composition provides a useful ability to modularize a formula using auxiliary propositions W which witness other regular properties given as QDDC formulas.

Example 4. *Consider a formula* `D` $=$ (`scount dev <= 3`) *which holds at a point provided the proposition* `dev` *is true at most 3 times in the entire past. Let formula* `D1` $=$ (`true^<o` $\neq$ `o'>`) *which holds at a point provided that the values of propositions* `o` *and* `o'` *differ at that position. Then,* `D` $\ll$ `Ind(D1,dev)` *is equivalent to the formula* (`scount dev <= 3`) `&&` `pref(EP(dev) <=> D1)`. *This formula holds at a position* `i`, *provided* `D1` *holds at most* 3 *time in the interval* `[0,i]`. *That is* `o`$\neq$`o'` *for at-most* 3 *positions in the interval* `[0,i]`. □

2.2 Supervisors and Controllers

Now we consider QDDC formulas and automata where variables $PV = I \cup O$ are partitioned into disjoint sets of input variables I and output variables O. We show how Mealy machines can be represented as special form of Deterministic finite automata (DFA). Supervisors and controllers are Mealy machines with special properties. This representation allows us to use the MONA DFA library [8] to efficiently compute supervisors and controllers in our tool DCSynth.

Definition 5 (Output-nondeterministic Mealy Machines)**.** *A total and Deterministic Finite Automaton (DFA) over input-output alphabet $\Sigma = 2^I \times 2^O$ is a tuple $A = (Q, \Sigma, s, \delta, F)$, as usual, with $\delta : Q \times 2^I \times 2^O \to Q$. An* **output-nondeterministic Mealy machine** *is a DFA with a unique reject (or non-final) state r which is a sink state i.e. $F = Q - \{r\}$ and $\delta(r,i,o) = r$ for all $i \in 2^I$, $o \in 2^O$.* □

Intuition is that the transitions from $q \in F$ to r are forbidden (and kept only for making the DFA total). Language of any such Mealy machine is prefix-closed. Recall that for a Mealy machine, $F = Q - \{r\}$. A Mealy machine is **deterministic** if $\forall s \in F, \forall i \in 2^I, \exists$ at most one $o \in 2^O$ s.t. $\delta(s,i,o) \neq r$. An output-nondeterministic Mealy machine is called **non-blocking** if $\forall s \in F, \forall i \in 2^I\ \exists o \in 2^I$ s.t. $\delta(s,i,o) \in F$. It follows that for all input sequences a non-blocking Mealy machine can produce one or more output sequence without ever getting into the reject state.

For a Mealy machine M over variables (I,O), its language $L(M) \subseteq (2^I \times 2^O)^*$. A word $\sigma \in L(M)$ can also be represented as pair $(ii,oo) \in ((2^I)^*,(2^O)^*)$ such that $\sigma[k] = ii[k] \cup oo[k], \forall k \in dom(\sigma)$. Here σ, ii, oo must have the same length. We will not distinguish between σ and (ii,oo) in the rest of the paper. Also, for any input sequence $ii \in (2^I)^*$, we will define $M[ii] = \{oo \mid (ii,oo) \in L(M)\}$.

Definition 6 (Controllers and Supervisors)**.** *An output-nondeterministic Mealy machine which is non-blocking is called a* **supervisor**. *A deterministic supervisor is called a* **controller**. □

The non-deterministic choice of outputs in a supervisor denotes unresolved decision. The determinism ordering below allows supervisors to be refined into controllers.

Definition 7 (Determinism Order and Sub-supervisor)**.** *Given two supervisors Sup_1, Sup_2 we say that Sup_2 is* more deterministic *than Sup_1, denoted $Sup_1 \leq_{det} Sup_2$, iff $L(Sup_2) \subseteq L(Sup_1)$. We call Sup_2 to be a* **sub-supervisor** *of Sup_1.* □

Note that being supervisors, they are both non-blocking, and hence $\emptyset \subset Sup_2[ii] \subseteq Sup_1[ii]$ for any $ii \in (2^I)^*$. The supervisor Sup_2 may make use of additional memory for resolving and pruning the non-determinism in Sup_1.

2.3 DCSynth Specification and Controller Synthesis

This section gives a brief overview of the soft requirement guided controller synthesis method from QDDC formulas. The method is implemented in a tool DCSynth. (See [18] for details). This method and the tool will be used for synthesis of run-time enforcement shields in the subsequent sections.

Definition 8. *A supervisor Sup* realizes invariance *of QDDC formula D over variables (I,O), denoted as* **Sup realizes AG (D)**, *provided $L(Sup) \subseteq L(D)$. Recall that, by the definition of supervisors, Sup must be non-blocking. The supervisor Sup is called* **maximally permissive** *provided for any supervisor Sup′ such that* **Sup′ realizes AG (D)**, *we have $Sup \leq_{det} Sup'$. Thus, no other supervisor with larger languages realizes the invariance of D. This Sup is unique up to language equivalence of automata, and the minimum state maximally permissive supervisor is denoted by* **MPS(D)**. □

A well-known greatest fixed point algorithm for safety synthesis over $\mathscr{A}(D)$ gives us $MPS(D)$ if it is realizable. We omit the details here (see [18]).

Proposition 9 (MPS Monotonicity)**.** *Given QDDC formulas D_1 and D_2 over variables (I,O) such that $\models (D_1 \Rightarrow D_2)$, we have:*

- *$MPS(D_2) \leq_{det} MPS(D_1)$, and*
- *If $MPS(D_1)$ is realizable then $MPS(D_2)$ is also realizable.* □

A **DCSynth specification** is a tuple (I,O,D^h,D^s) where I and O are the set of *input* and *output* variables, respectively. Formula D^h, called the *hard requirement*, and formula D^s, called the *soft requirement*, are QDDC formulas over the set of propositions $PV = I \cup O$. The objective in DCSynth is to synthesize a deterministic controller which (a) **invariantly** satisfies the hard requirement D^h, and (b) **optimally** satisfies D^s for as many inputs as possible.

The controller synthesis goes through following three stages.

1. The DCSynth specification (I, O, D^h, D^s) is said to be realizable iff $MPS(D^h)$ is realizable (i.e. it exist). The synthesis method first computes the maximally permissive supervisor $MPS(D^h)$ realizing invariance of D^h. When clear from context we will abbreviate this as MPS.

2. A sub-supervisor of $MPS(D^h)$ which satisfies D^s for "as many inputs as possible" is computed. This is formalized using a notion of **H-optimality** w.r.t. the soft requirement D^s. We explain this only intuitively. The reader may refer to the original paper [18] for a formal definition of H-optimality and the synthesis algorithm. Let H be a natural number called horizon. We construct the maximally permissive sub-supervisor of $MPS(D^s)$, called $MPHOS(D^h, D^s, H)$, by pruning the non-deterministic choice of outputs in MPS and retaining only the outputs which give *the highest expected count of (intermittent) occurrence of D^s over the next H steps of execution.* This count is averaged over all input sequences of length H. A well known value-iteration algorithm due to Bellman [1], adapted from optimal strategy synthesis for Markov Decision Processes [16], gives us the required H-optimal maximally permissive sub-supervisor. See the paper [18] for full details which are omitted here. Note that, by construction, $MPS(D^h) \leq_{det} MPHOS(D^h, D^s, H)$. By Definition 7, all the behaviours of $MPHOS$ will invariantly satisfy D^h and the $MPHOS$ will be H-optimal with respect to D^s. When clear from context, $MPHOS(D^h, D^s, H)$ will be abbreviated as $MPHOS$.

3. Both $MPS(D^h)$ and $MPHOS(D^h, D^s, H)$ are supervisors and they may be output-nondeterministic as there can be several optimal outputs possible. Any controller obtained by arbitrarily resolving the output non-determinism in $MPHOS(D^h, D^s, H)$ will also be H-optimal. In tool DCSynth, we allow users to specify a preference ordering Ord on the set of outputs 2^O. Any supervisor Sup can be determinized by retaining only the highest ordered output among those permitted by Sup. This is denoted by $Det_{Ord}(Sup)$. In tool DCSynth, the output ordering is specified by giving a lexicographically ordered list of output variable literals, as illustrated in Example 10 below. This facility is used to determinize supervisors $MPHOS(D^h, D^s, H)$ and $MPS(D^h)$ as required. These are denoted by $Det_{ord}(MPHOS(D^h, D^s, H))$ and $Det_{Ord}(MPS(D^h))$.

Example 10. *For a supervisor Sup over variables $(I, \{p, q\})$, an output ordering can be given as list $(!q > !p)$, Then, the determinization step will select the highest allowed output from the list $(p = false, q = false)$, $(p = true, q = false)$, $(p = false, q = true)$, $(p = true, q = true)$ in that order. This choice is made for each state and each input.* □

In summary, given a DCSynth specification (I, O, D^h, D^s), a horizon value H and a preference ordering ord on outputs 2^O, the tool DCSynth outputs maximally permissive supervisors $MPS(D^h)$ and $MPHOS(D^h, D^s, H)$ as well as controllers $Det_{Ord}(MPS(D^h))$ and $Det_{ord}(MPHOS(D^h, D^s, H))$.

Extended DCSynth specification: DCSynth supports the specification of soft requirements as an ordered list of formulas with user defined weights. This feature is used in the synthesis of run-time enforcement shields. The *extended DCSynth specification* is a tuple $S = (I, O, D^h, \langle D^s_1 : \theta_1, \cdots, D^s_k : \theta_k \rangle)$ where I and O are sets of *input* and *output* variables respectively. The QDDC formula D^h, which is over $I \cup O$, specifies the *hard requirement* on the controller to be synthesized. The *soft requirement* $\langle D^s_1 : \theta_1, \cdots, D^s_k : \theta_k \rangle$ is a list where each D^s_i is a QDDC formula over $I \cup O$. $\theta_i \in \mathbb{N}$ specifies the weight of the soft requirement D^s_i. The weight (reward) of a transition is sum of weights of each of the formula D^s_i which holds on taking the transition. The tool DCSynth produces a supervisor, which maximizes the cumulative expected value of this reward over next H-steps of execution. This cumulative reward is averaged over all input sequences of length H.

3 Specification and Synthesis of Run-time Enforcement Shields

Given a **correctness requirement** $REQ(I,O)$ as a QDDC formula over input-output propositions (I,O), a system with sporadic errors (SSE) may fail to meet the requirement at some of the points in a behaviour (ii,oo). (The reader may recall Definition 5 and its following two paragraphs for the notation.) A **run-time enforcement shield** is a Mealy machine with input variables $I \cup O$ and output variable O'. See Figure 2. For any input (ii,oo) the shield produces a modified output oo' such that (ii,oo') invariantly satisfies the correctness requirement $REQ(I,O')$. Moreover, the output oo' must deviate from the SSE output oo as little as possible to maintain quality. There are several distinct notions of "deviating as little as possible" leading to different shields.

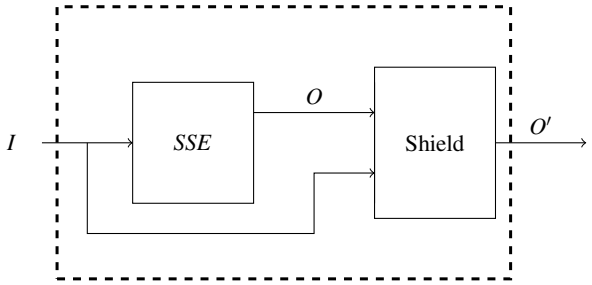

Figure 2: Run-time Enforcement Shield.

In this section, we give a logical framework for specifying various shields by using the logic QDDC. We then provide an automatic synthesis of a run-time enforcement shield from its logical specification using the tool DCSynth of the previous section. Thus, we achieve a logical specification and a uniform synthesis method for shields.

Deviation constraints specify the extent of allowed deviation in a shield's behaviour. Our specification has **hard deviation constraint** HDC which must be mandatorily and invariantly satisfied by the shield. (This is similar to the hard requirement in DCSynth.) We also define a canonical **soft deviation constraint** $Hamming(O,O')$ which will be useful in minimizing cumulative deviation during synthesis. Overall, a **shield specification** consists of a pair (REQ,HDC).

3.1 Hard Deviation Constraints

Two indicator propositions, $SSEOK$ and $Deviation$ play an important role in formulating hard deviation constraints. Proposition $SSEOK$ indicates whether the SSE is meeting the requirement $REQ(I,O)$ at the current position. Proposition $Deviation$ indicates whether at the current position, the shield output is different from the SSE output. Recall that in DCSynth specifications, the formula $Ind(D,w)$ defines a fresh output proposition w which is true at a position provided the past of the position satisfies formula D (see Definition 3). We use the following list of indicator definitions in formulating hard deviation constraints. Let, $O = \{o_1,\ldots,o_r\}$ and $O' = \{o'_1,\ldots,o'_r\}$.

$$INDDEF \;=\; \left\langle \begin{array}{l} Ind(\ REQ(I,O),\ SSEOK),\\ Ind(\ true\hat{}\langle \vee_i (o_i \neq o'_i)\rangle,\ Deviation) \end{array} \right\rangle$$

A **hard deviation constraint** HDC is a QDDC formula over propositions $SSEOK$ and $Deviation$. It specifies a constraint on $Deviation$ conditional upon the behaviour of $SSEOK$. In Subsection 3.4, we will give a list of several different hard deviation constraints.

For shield synthesis using DCSynth, we define the QDDC formula $HShield$ given in Equation 1) as the hard requirement over the input-output propositions $(I \cup O, O')$. Notice that in its formulation, we use the cascade composition from Definition 3. This allows us to modularize the specification into components REQ and HDC.

$$HShield \;=\; REQ(I,O') \wedge HDC(SSEOK, Deviation) \ll INDDEF \tag{1}$$

The constraint (QDDC formula) $HShield$ must be invariantly satisfied by the shield. Tool DCSynth gives us a maximally permissive supervisor $MPS(HShield)$ with this property (See definition 8). This supervisor can be termed as *shield-supervisor without deviation minimization* and it will be denoted by $MPS(REQ,HDC)$.

3.2 Soft Deviation Constraint

While HDC already places some constraints on the permitted deviation, we can further optimize the deviation in supervisor $MPS(REQ,HDC)$ of the previous section. Quantitative optimization techniques from Markov Decision Processes can be used. (Stocasticity comes from the distribution of inputs to the shield.) The tool DCSynth allows us to specify such optimization using a list of soft requirement formulas with weights. The tool optimizes a supervisor to a sub-supervisor which maximizes the expected value of cumulative weight of soft requirements over next H-steps. This cumulative weight is averaged over all input sequences of length H. See Section 2.3 and [18] for further details.

We make use of this H-optimal sub-supervisor computation to get a sub-supervisor which minimizes the expected cumulative deviation over next H-steps. Given the set of output propositions $O = \{o_1, \ldots, o_r\}$, consider the DCSynth soft-requirement

$$Hamming(O,O') = \langle (true^\frown \langle o_1 = o_1' \rangle) : 1,\ \ldots\ , (true^\frown \langle o_r = o_r' \rangle) : 1 \rangle \tag{2}$$

Thus, non-deviation of any output variable $o_i = o_i'$ at current position contributes a reward 1. This is summed over all output variables to give weight (reward) of the soft requirement. Thus, the weight of the soft requirement $Hamming(O,O')$ at any position k in a word (ii, oo, oo') is the value $(r-h)$ where h is the hamming distance between $oo[k]$ and $oo'[k]$. If oo and oo' perfectly match at position k then the weight at position k is r, whereas if oo and oo' differ in values of say p variables at position k then the weight at the position k is $r-p$.

By using $Hamming(O,O')$ as soft requirement and by selecting a horizon value H, we can apply the tool DCSynth to obtain a sub-supervisor

$$MPHOS(\ MPS(REQ,HDC),\ Hamming(O,O'),\ H\ )$$

of the supervisor $MPS(REQ,HDC)$. This sub-supervisor retains only the outputs which maximize the expected accumulated weight of $Hamming(O,O')$ over next H steps in future. This supervisor is called the *shield-supervisor with deviation minimization* and denoted by $MPHOS(REQ,HDC,H)$.

3.3 Determinization

The reader must note that both the shield-supervisors $MPS(REQ,HDC)$ and $MPHOS(REQ,HDC,H)$ are output non-deterministic. Multiple choice of outputs may satisfy the hard deviation constraints while being H-optimal for the soft deviation constraint. Any arbitrary resolution of the output non-determinism will preserve the invariance guarantees and H-optimality (see [18]).

In our method, we allow the user to specify a preference ordering *ord* on the shield outputs $2^{O'}$. A lexicographically ordered list of output literals is given as explained in Example 10. A deterministic controller is obtained by retaining only the highest ordered output from the non-deterministic choice of outputs offered by the supervisor. Thus, given a preference ordering *ord* we can obtain shields (**deterministic controllers**) $Det_{ord}(MPS(REQ,HDC))$ and $Det_{ord}(MPHOS(REQ,HDC,H))$.

In summary, given a correctness requirement $REQ(I,O)$ to be enforced by the shield, a hard deviation constraint $HDC(SSEOK,Deviation)$, a horizon value H (for globally minimizing the deviation over next H steps) and a preference ordering *ord* on shield outputs $2^{O'}$, we can synthesize shields $Det_{ord}(MPS(REQ,HDC))$ and $Det_{ord}(MPHOS(REQ,HDC,H))$. When ord,REQ,HDC,H are clear from context, these shields are referred to as *Shield_NoDM* (shield with no deviation minimization) and *Shield_DM* (shield with deviation minimization), respectively.

3.4 Variety of Hard Deviation Constraints and Shield-Types

In Table 1 below, we give a useful list of several different hard deviation constraints (*HDC*) as QDDC formulas. These include the specifications of the burst-error shield of Wu *et al.* and the k-stabilizing shield of Bloem *et al. as* well as a new notion of e,d-shield. Labels V_0 to V_3 are used to identify these specifications in the experiments. Each of these *HDC* can be used to synthesize shields with or without deviation minimization as explained in the previous subsection.

Table 1: Variety of Hard Deviation Constraints

	ShieldType	**HDC**
V_0	Burst-shield	*true*
V_1	k-shield	`[]([[Deviation]]=>slen<k)`
V_2	k-stabilizing shield	`[]([[SSEOK && Deviation]]=>slen<k) && ( []( (<!Deviation>^[[SSEOK]]) => [[!Deviation]] ) )`
V_3	e,d-shield	`[]((scount !SSEOK <= e) => (scount Deviation <=d) && ( []( (<!Deviation>^[[SSEOK]]) => [[!Deviation]] ) )`

We provide some explanation and comments on these specifications.

- The proposition *SSEOK* denotes that the *SSE* is not making correctness error where as proposition *Deviation* denotes that the shield is deviating from the *SSE* output. The QDDC formula `( []( (<!Deviation>^[[SSEOK]]) => [[!Deviation]] ) )` states that in any observation interval, if the interval begins with no deviation, and there is no error by SSE during the interval, then there is no deviation throughout the interval. This property can be called *NoSpuriousDeviation*. It is included as a conjunct in k-Shield V_2 as well as e,d-Shield V_3.
- Burst-shield (V_0) does not enforce any hard deviation constraint. Thus, only hard requirement on the synthesized shield is to meet $REQ(I \cup O, O')$ invariantly. However, we can use this together with deviation minimization using the soft deviation constraint $Hamming(O,O')$. By taking horizon $H = 0$, we obtain the burst sheild of Wu *et al.* [21] which locally optimizes deviation at each step without any look-ahead into the future. Larger horizon values give superior shields which improve the probability of non-deviation in long run, as shown by our experiments which are reported later in this paper.

- A k-shield (V_1) specifies (as its hard deviation constraint) that for any observation interval the deviation can invariantly happen for at most k cycles. Thus, a burst of deviation has length of at most k cycles. The k-shield (V_1) specifies that this property must hold unconditionally. *Such a specification is often unrealizable.* For example, if SSE makes consecutive errors for more than k cycles, the shield may be forced to deviate for all of these cycles. Hence, several variants of the V_1 shield have been considered.

- The k-stabilizing shield (V_2) specifies that the shield may deviate as long as *SSE* makes errors (even burst errors). Once *SSE* recovers from deviation (indicated by $SSEOK$ becoming and remaining true), the shield may deviate for at most k cycles. Thus, the shield must recover from deviation within k cycles once $SSEOK$ is established and maintained. Also, there must be no spurious deviation due to conjunct *NoSpuriousDeviation*. This specification precisely gives the k-stabilizing shield without fail-safe state, originally defined by Konighofer *et al.* [9]. By a variation of this, the k-stabilizing shield with fail-safe state [9] can also be specified but we omit this here.

- We define a new notion of shield called e,d-shield (V_3). This states that in any observation interval if the count of errors by SSE (given by the term `(scount !SSEOK)`) is at most e then the count of number of cycles with deviations (given by the term `(scount Deviation)`) is at most d. Thus e errors lead to at most d deviations. Also, there is no spurious deviation due to the conjunct *NoSpuriousDeviation*.

It may be noted that irrespective of the shield type the synthesized shield have to meet the requirement $REQ(I,O')$ invariantly as specified by the formula $HShield$ (See Equation 1).

4 Performance Measurement Metrics and Experiments

In this section we give the experimental results for shield synthesis carried out in our framework. We first benchmark the performance of our tool and compare it with some other tools for shield synthesis in Section 4.1. In Section 4.2 we define some performance measurement metrics for shields and we use these to compare various shield types.

4.1 Performance of Tool DCSynth in Shield Synthesis

We have synthesized Burst-shield V_0 with deviation minimization using DCSynth for all the benchmark examples given in [21]. The results are tabulated in Table 2. All our experiments were conducted on Linux (Ubuntu 18.04) system with Intel i5 64 bit, 2.5 GHz processor and 4 GB memory. The formula automata files of Wu *et al.*[19] were used in place of QDDC formulas for uniformity. For a comparision with other tools, the results for the k-stabilizing shield synthesis and the Burst-error shield synthesis for the same examples are reproduced directly from Wu *et al.* [21]. As these are for unknown hardware setup, a direct comparison of the synthesis times with the DCSynth synthesis times is only indicative.

As the table suggests, in most of the cases, the shield synthesized by DCSynth compares favorably with the results reported in literature [21], both in terms of the size of the shield and the time taken for the synthesis. Recall that DCSynth uses aggressive minimization to obtain smaller shields. As an example, for the specification AMBA G5+6+9e64+10, our tool synthesizes a shield significantly faster and with smaller number of states than the existing tools[2, 21].

Table 2: Synthesis of Burst shield-V_0 with Deviation Minimization optimization using DCSynth. For each specification, the number of states of the resulting shield and time (in seconds) for synthesizing it are reported. For comparision, results for k-stabilizing shield synthesis and Burst-error shield synthesis are reproduced directly from Wu *et al.* [21].

Specification	k-Stabilizing shield		Burst-error shield		Burst shield V_0 with DM			
					For H=0		For H=10	
	states	time	states	time	states	time	states	time
Toyota Powertrain	38	0.2	38	0.3	9	0.07	9	0.35
Traffic light	7	0.1	7	0.2	4	0.008	4	0.059
$F_{64}p$	67	0.7	67	0.5	67	0.009	67	0.029
$F_{256}p$	259	46.9	259	10.5	259	0.08	259	0.09
$F_{512}p$	515	509.1	515	54.4	515	0.24	515	0.26
$G(\neg q) \vee F_{64}(q \wedge F_{64}p)$	67	0.8	67	0.6	67	0.015	67	0.06
$G(\neg q) \vee F_{256}(q \wedge F_{256}p)$	259	46.2	259	10.7	259	0.16	259	0.27
$G(\neg q) \vee F_{512}(q \wedge F_{512}p)$	515	571.7	515	54.5	515	0.77	515	0.91
$G(q \wedge \neg r \rightarrow (\neg r \cup_4 (p \wedge \neg r)))$	15	0.1	145	0.1	6	0.002	6	0.013
$G(q \wedge \neg r \rightarrow (\neg r \cup_8 (p \wedge \neg r)))$	109	0.2	5519	4.5	10	0.003	10	0.023
$G(q \wedge \neg r \rightarrow (\neg r \cup_{12} (p \wedge \neg r)))$	753	6.3	27338	1414.5	14	0.009	14	0.03
AMBA G1+2+3	22	0.1	22	0.1	7	0.002	7	0.01
AMBA G1+2+4	61	6.3	78	2.2	8	0.2	8	1.69
AMBA G1+3+4	231	55.6	640	97.6	14	0.25	14	2.01
AMBA G1+2+3+5	370	191.8	1405	61.8	12	0.017	13	0.105
AMBA G1+2+4+5	101	3992.9	253	472.9	12	1.27	12	8.86
AMBA G4+5+6	252	117.9	205	26.4	18	0.86	18	7.99
AMBA G5+6+10	329	9.8	396	31.4	27	3.7	27	36.14
AMBA G5+6+9e4+10	455	17.6	804	42.1	46	5.58	46	52.96
AMBA G5+6+9e8+10	739	34.9	1349	86.8	64	7.44	64	70.73
AMBA G5+6+9e16+10	1293	74.7	2420	189.7	100	11.3	100	105.2
AMBA G5+6+9e64+10	4648	1080.8	9174	2182.5	316	37.17	316	202.52
AMBA G8+9e4+10	204	7.0	254	6.1	48	0.29	16	2.13
AMBA G8+9e8+10	422	22.5	685	33.7	84	0.55	20	3.49
AMBA G8+9e16+10	830	83.7	1736	103.1	156	1.02	28	6.32
AMBA G8+9e64+10	3278	2274.2	7859	2271.5	588	5.96	76	24.89

4.2 Comparison between various shield notions

For comparing the performance of shields synthesized with different shield types, we define the following performance metrics.

Expected Value of Non-deviation of a Shield in Long run: A shield is said to be in a non-deviating state if the shield output O' matches the SSE output O. A proposition `!Deviation` holds for such states. We measure the probability of shield being in such states over its long runs, as described below.

Given a shield S over input-output propositions $((I \cup O), O')$ and a QDDC formula (regular property) D over variables $I \cup O \cup O'$, we construct a *Discrete Time Markov Chain (DTMC)*, denoted as $M_{unif}(S, D)$, whose analysis allows us to measure the probability of D holding in long runs (steady state) of S under independent and identically distributed (iid) inputs. This value is called the expected value of D holding in a shield S and designated as $\mathbb{E}_{unif}(S, D)$.

The construction of the desired DTMC is as follows. The product $S \times \mathscr{A}(D)$ gives a finite state automaton with the same behaviours as S. Moreover, it is in accepting state exactly when D holds for the past behaviour. (Here $\mathscr{A}(D)$ works as a total deterministic monitor automaton for D without restricting S). By assigning uniform discrete probabilities to all the inputs from any state, we obtain the DTMC $M_{unif}(S, D)$ along with a designated set of accepting states. The DTMC is in accepting state precisely

when D holds. Standard techniques from Markov chain analysis allow us to compute the *probability* (Expected value) of being in the set of accepting states on long runs (steady state) of the DTMC. This gives us the desired value $\mathbb{E}_{unif}(S,D)$. A leading probabilistic model checking tool MRMC implements this computation [7]. In DCSynth, we provide a facility to compute $M_{unif}(S,D)$ in a format accepted by the tool MRMC. Hence, using DCSynth and MRMC, we are able to compute $\mathbb{E}_{unif}(S,D)$.

The expected value of a shield S being in a non-deviating state over long runs can be computed as $\mathbb{E}_{unif}(S,$`true^<!Deviation>`$)$.

Worst Case Burst-Deviation Latency: The *worst case burst-deviation latency* gives the maximum number of consecutive cycles for which the shield deviates even when the SSE is satisfying the requirement. Thus, it denotes the maximum length of an interval in the behaviour of the shield for which the formula "*SSEOK && Deviation*" holds invariantly.

Given a Shield S and a QDDC formula D, the latency goal $MAXLEN(D,S)$ computes

$$sup\{e-b \quad | \quad \rho,[b,e] \models D,\ \rho \in Exec(S)\}$$

i. e. it computes the length of the longest interval satisfying D across all the executions of S. Thus, it computes the worst case span of behaviour fragments matching D in S. Tool CTLDC [14] implements a model checking technique for computing MAXLEN(D,S). The worst case burst deviation latency of shield measures the maximum number of consecutive cycles having deviation in worst case. The worst case burst-deviation latency of a shield S can be computed as $MAXLEN([[SSEOK \ \&\& \ Deviation]],S)$.

4.2.1 Experiments and Findings

We can use the *expected value of deviation* and the *worst case burst-deviation latency*, defined above, for comparing the shields obtained using various shield-types defined in Section 3.4. We synthesized various shields for the correctness requirement $\varphi_{until}(n)$ given in Example 1 with $n = 5$ and the input-output propositions $(\{r\},\{p,q\})$. The output propositions of synthesized shield are $\{p',q'\}$. For each shield type V_i given in Table 1, the deterministic shields V_i_NoDM and V_i_DM were synthesized as outlined in the last paragraph of Section 3.3. Here V_i_NoDM denotes shield synthesized without deviation minimization where as V_i_DM denotes the shield obtained with deviation minimization optimization. The shield-supervisors were determinized with the preference ordering $(!q' > !p')$ on outputs.

Table 3 gives the results obtained. We report the number of states of the shield along with the time taken (in seconds) by the tool DCSynth to compute the shield. Moreover, for comparing the performance of the resulting shields, their **Expected Value of non-deviation** as well as the **worst case burst-deviation latency** are reported in the table under the columns titled Expected Value and Latency, respectively.

It is observed that with deviation minimization optimization, several different shield types resulted in identical shields, although the time to synthesize them differed. For example, shields in rows numbered 10 to 15 are identical. We indicate such a situation by merging the corresponding rows to a single cell. We give our findings below.

- The k-shield (V_1) is unrealizable as expected. See its description in Section 3.4 for an explanation. All the other shield types are found to be realizable.
- For shield synthesis without deviation minimization, we obtain distinct shields with distinct performance for each shield type. The Burst shield (V_0) has the poorest performance (expected non-deviation 0.25 and latency ∞) as it enforces trivial hard deviation requirement *true*. The best

Table 3: Shield Synthesis for the formula $\varphi_{until}(5)$ of Example 1 with various shield types defined in Table 1 and their Performance comparison. The *expected value of non-deviation in long run* and the *worst case burst-deviation latency* are reported.

Sr. No.	Shield Type	States	Time	Expected Value	Latency
Shield Synthesis of Requirement $\varphi_{until}(5)$ Without Deviation Minimization					
1.	V_0_NoDM	18	0.004	0.25	∞
2.	V_1_NoDM(k=1)	Unrealizable			
3.	V_2_NoDM(k=1)	14	0.004	0.7142793	1
4.	V_1_NoDM(k=3)	Unrealizable			
5.	V_2_NoDM(k=3)	18	0.009	0.5982051	3
6.	V_3_NoDM(e=1,d=1)	13	0.001	0.7499943	0
7.	V_3_NoDM(e=1,d=2)	26	0.005	0.7182475	1
8.	V_3_NoDM(e=1,d=3)	40	0.008	0.6614611	2
Shield Synthesis of Requirement $\varphi_{until}(5)$ With Deviation Minimization					
9.	V_1_DM(k=1)	Unrealizable			
10.	V_0_DM(H=0)	13	0.003	0.833252	0
11.	V_2_DM(k=1)(H=0)		0.005		
12.	V_2_DM(k=3)(H=0)		0.006		
13.	V_3_DM(e=1,d=1)(H=0)		0.004		
14.	V_3_DM(e=1,d=2)(H=0)		0.005		
15.	V_3_DM(e=1,d=3)(H=0)		0.004		
16.	V_0_DM(H=10)	8	0.016	0.8571396	0
17.	V_2_DM(k=1)(H=10)		0.01		
18.	V_2_DM(k=3)(H=10)		0.009		
19.	V_3_DM(e=1,d=1)(H=10)		0.008		
20.	V_3_DM(e=1,d=2)(H=10)		0.012		
21.	V_3_DM(e=1,d=3)(H=10)		0.013		

performance is obtained for the newly defined e,d-shield type V_3 by choosing $e = d$. This gives 0.74 as the expected value of non-deviation and worst case latency of 0 cycles. With increased difference $d - e$ the performance degrades. Similarly in k-stabilizing shield (V_2) the performance degrades with increase in the value of k, as expected.

- The performance of the shield considerably improves with the deviation minimization (DM) optimization. Expected value of 0.85 compares well against the best value of 0.74 without deviation minimization. Also burst-deviation latency drops to 0 with DM. We also notice that the performance improves with increase in the horizon value when using DM. This is intuitively clear as the tool performs global optimization across larger number of steps of look-ahead with increased horizon.

- For shield synthesis with deviation minimization optimization, all the different shield types V_0, V_2, V_3 resulted in identical shield for a given value of horizon H. Thus shields in rows 10-15 (synthesized with $H = 0$) and rows 16-21 (synthesized with $H = 10$) are found to be identical. This shows that deviation minimization effectively supersedes the different hard deviation guarantees provided by

the HDC. While this is not theoretically guaranteed, our experience with robust controller synthesis also indicates the overwhelming effectiveness of the DM-like optimization [15].

5 Discussion and Related Work

In this paper we have presented a logical framework for specifying error-correcting run-time enforcement shields using formulas of logic QDDC. The specification contains a correctness requirement REQ, specifying the desired input-output relation to be maintained, as well as a hard deviation constraint HDC which specifies a constraint on deviation between the system output and the shield output. Our shield synthesis gives a shield which invariantly satisfies both REQ and HDC. Moreover, a powerful optimization globally minimizes the cumulative deviation between the system and the shield output.

The idea of error-correcting run-time enforcement shield was proposed in the pioneering work of Bloem *et al.* [2], where the notion of k-stabilizing shield (with a synthesis algorithm) was proposed. This was further enhanced by Konighofer *et al.* [9]. Extension of shield synthesis to liveness properties has also been explored in this paper. Wu *et al.* [21, 20] defined the burst shield which is capable of handling burst errors. Moreover, they proposed optimizing the shield with the choice of output which locally minimizes the deviation at each stage. In this paper, we have enhanced this with global optimization of cumulative deviation across next H steps.

In our method, the shield is logically specified using QDDC formulas and a uniform method for the synthesis of the shield is proposed. A tool DCSynth implements the synthesis method. Logic QDDC [13, 12, 11] with its interval logic modalities, threshold counting constraints, regular expression like constructs and second-order quantification over temporal variables provides a very rich vocabulary to specify both the system requirements and the deviation constraints. Logic QDDC is a discrete time version of Duration Calculus proposed by Zhou, Hoare and Ravn [5, 4] with known automata theoretic decision and model checking procedures [13, 3, 17, 10]. Using the proposed technique, we have specified the k-stabilizing shield of Konighofer *et al.* [9], the burst shield of Wu *et al.* [21, 20], as well as a new e,d-shield. Moreover, we have measured the performance of the shields resulting from these different criteria in terms of the expected value of deviation in long runs, as well as the worst case burst deviation latency. Our experiments show an overwhelming impact of global deviation minimization on the quality of the shield. At the same time, hard deviation constraints provide a conditional hard guarantee on the worst case deviation. Hence, the combination of hard deviation constraint together with global minimization of deviation is useful.

Konighofer *et al.* [9] as well as Ehlers and Topku [6] propose controller/shield synthesis technique for optimal achievable value of parameter k in a regular specification. By contrast, our current method requires k to be specified. In our future work, we will address similar optimal parametric synthesis from parameterized QDDC specifications.

References

[1] R. E. Bellman (1957): *Dynamic Programming*. Princeton Univ. Press.

[2] Roderick Bloem, Bettina Könighofer, Robert Könighofer & Chao Wang (2015): *Shield Synthesis: - Runtime Enforcement for Reactive Systems*. In Christel Baier, editor: *TACAS*, *LNCS* 9035, Springer, pp. 533–548, doi:10.1007/978-3-662-46681-0_51.

[3] Gaurav Chakravorty & Paritosh K. Pandya (2003): *Digitizing Interval Duration Logic*. In Warren A. Hunt & Fabio Somenzi, editors: *CAV*, *LNCS* 2725, Springer, pp. 167–179, doi:10.1007/978-3-540-45069-6_17.

[4] Zhou Chaochen & Michael R. Hansen (2004): *Duration Calculus - A Formal Approach to Real-Time Systems*. Monographs in Theoretical Computer Science. An EATCS Series, Springer, doi:10.1007/978-3-662-06784-0.

[5] Zhou Chaochen, C. A. R. Hoare & A. P. Ravn (1991): *A Calculus of Durations*. *Inf. Process. Lett.* 40(5), pp. 269–276, doi:10.1016/0020-0190(91)90122-X.

[6] Rüdiger Ehlers & Ufuk Topcu (2014): *Resilience to Intermittent Assumption Violations in Reactive Synthesis*. In: *HSCC*, HSCC '14, ACM, New York, NY, USA, pp. 203–212, doi:10.1145/2562059.2562128.

[7] J. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns & D. N. Jansen (2011): *The ins and outs of the probabilistic model checker MRMC*. *Performance Evaluation* 68, pp. 89–220, doi:10.1016/j.peva.2010.04.001. Available at http://www.sciencedirect.com/science/article/pii/S0166531610000660.

[8] N. Klarlund, A. Møller & M. I. Schwartzbach (2001): *MONA Implementation Secrets* 2088, pp. 182–194. doi:10.1007/3-540-44674-5_15.

[9] Bettina Könighofer, Mohammed Alshiekh, Roderick Bloem, Laura Humphrey, Robert Könighofer, Ufuk Topcu & Chao Wang (2017): *Shield synthesis*. *FMSD* 51(2), pp. 332–361, doi:10.1007/s10703-017-0276-9.

[10] Shankara Narayanan Krishna & Paritosh K. Pandya (2005): *Modal Strength Reduction in Quantified Discrete Duration Calculus*. In: *FSTTCS*, *LNCS* 3821, Springer, pp. 444–456, doi:10.1007/11590156_36.

[11] Raj Mohan Matteplackel, Paritosh K. Pandya & Amol Wakankar (2017): *Formalizing Timing Diagram Requirements in Discrete Duration Calculus*. In: *SEFM 2017*, *LNCS* 10469, Springer International Publishing, pp. 253–268, doi:10.1007/978-3-319-66197-1_16.

[12] Paritosh K. Pandya (2001): *Model Checking CTL*[DC]*. In: *TACAS*, *LNCS* 2031, Springer, pp. 559–573, doi:10.1007/3-540-45319-9_38.

[13] Paritosh K. Pandya (2001): *Specifying and deciding quantified discrete-time duration calculus formulae using DCVALID*. In: *RTTOOLS (affiliated with CONCUR 2001)*, CiteSeer.

[14] Paritosh K. Pandya (2005): *Finding Extremal Models of Discrete Duration Calculus formulae using Symbolic Search*. *Electronic Notes in Theoretical Computer Science* 128(6), pp. 247 – 262, doi:10.1016/j.entcs.2005.04.015. Available at http://www.sciencedirect.com/science/article/pii/S1571066105002471. AVoCS 2004.

[15] Paritosh K. Pandya & Amol Wakankar (2019): *Specification and Reactive Synthesis of Robust Controllers*. *CoRR* abs/1905.11157. Available at http://arxiv.org/abs/1905.11157.

[16] Martin L. Puterman (1994): *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, 1st edition. John Wiley & Sons, Inc., New York, NY, USA, doi:10.1002/9780470316887.

[17] Babita Sharma, Paritosh K. Pandya & Supratik Chakraborty (2005): *Bounded Validity Checking of Interval Duration Logic*. In: *TACAS*, *LNCS* 3440, Springer, pp. 301–316, doi:10.1007/978-3-540-31980-1_20.

[18] Amol Wakankar, Paritosh K. Pandya & Raj Mohan Matteplackel (2019): *DCSYNTH: A Tool for Guided Reactive Synthesis with Soft Requirements, (To Appear in Proc. VSTTE 2019)*. *CoRR* abs/1903.03991. Available at http://arxiv.org/abs/1903.03991.

[19] Meng Wu (2016): *iShield2 Synthesizer*. *https://bitbucket.org/mengwu/shield-synthesis/*. Available at https://bitbucket.org/mengwu/shield-synthesis/.

[20] Meng Wu, H. Zeng, C. Wang & H. Yu (2017): *INVITED: Safety guard: Runtime enforcement for safety-critical cyber-physical systems*. In: *DAC*, ACM, pp. 1–6, doi:10.1145/3061639.3072957.

[21] Meng Wu, Haibo Zeng & Chao Wang (2016): *Synthesizing Runtime Enforcer of Safety Properties Under Burst Error*. In Sanjai Rayadurgam & Oksana Tkachuk, editors: *NFM*, *LNCS* 9690, Springer, pp. 65–81, doi:10.1007/978-3-319-40648-0_6.