Shield Synthesis for Cyber Physical Systems

A B.Tech Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Siddhartha Jain
(180101078)

Samay Varshney
(180101097)

under the guidance of

Dr. Purandar Bhaduri

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY GUWAHATI
GUWAHATI - 781039, ASSAM

CERTIFICATE

*This is to certify that the work contained in this thesis entitled "**Shield Synthesis for Cyber Physical Systems**" is a bonafide work of (**Roll No. 180101078 and 180101097**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Guwahati under my supervision and that it has not been submitted elsewhere for a degree.*

Supervisor: **Dr. Purandar Bhaduri**

Professor

Department of Computer Science & Engineering

Indian Institute of Technology Guwahati, Assam

Acknowledgements

We would like to express our gratitude to our supervisor **Dr. Purandar Bhaduri** for giving us the golden opportunity to explore the field of Shield Synthesis for Cyber Physical Systems and the much needed guidance and enthusiasm to delve into some of the state-of-the-art works.

Contents

Chapter 1

Abstract & Introduction

1.1 Abstract

Cyber-Physical Systems (CPS) are safety-critical real-time systems in which erroneous behaviour may inflict serious consequences. So, we require another controller which can correct the output of the system. Here, we explore the run-time enforcement shield as a controller which takes input and output from System SSE (SSE stands for system with sporadic errors which is a controller that takes inputs and produce correct outputs but it might sometimes violate a correctness requirement) and produces a corrected output $O^{'}$ which should deviate from system output as little as possible. In this report, we construct shield from specifications written using QDDC (Quantified Discrete Duration Calculus) formulas. The specification includes two requirements: correctness requirements REQ and HDC which stands for hard deviation constraints. Shield must be constructed in such a way both these requirements must be invariantly and mandatorily satisfied by it. We use tool DCSynth for automatic synthesis of shield from the given property/specification.

In later half, we explore the slugs tool which is a GR(1) (generalized reactivity(1)) synthesis tool with a powerful plugin architecture through which we can modify any aspect of the synthesis process to fit the application. We used the tool on various examples like

single robot moving, water reservoir.

1.2 Introduction

SSE stands for system with sporadic errors which is a controller that takes input I and produce correct output O but it may sometimes violate a correctness requirements. Here, output and input are set of atomic output and input propositions respectively. A run-time enforcement shield is a mealy machine which takes input and output produced by the system. The shield is constructed for a specific correctness requirement and it generates a corrected output $O^{'}$ which mandatorily and invariantly satisfy the correctness requirements REQ(I, O'). The shield also ensures that the shield output must deviate from that of system as little as possible to maintain the efficiency and quality.

A major issue in producing a shield is to make the shield output to deviate as little as possible from system's output. There are multiple approaches proposed regarding this issue. There is a *k-stabilizing* shield that says that shield's output can deviate for at most k cycles continuously from system's output under certain assumptions. If they are not fulfilled, then shield may deviate randomly. This assumptions are considered as hard requirement that must be compulsorily fulfilled in any behaviour by the shield. We call such constraints/assumptions as HDC which stands for hard deviation constraints.

We use interval temporal logic QDDC to specify correctness requirement REQ(I, O) and hard deviation constraints HDC. QDDC allows us to briefly specify regular properties using its interval based modalities and counting constructs.

k-stabilizing shields that we discussed earlier are incapable of handling burst errors. As a solution to this issue, burst-error shield was proposed, which invariantly satisfy the critical requirement, and also minimizes the deviation between shield and system output locally. Such shields are called as locally deviation minimizing shields.

Here, we generalize the above method to globally minimize the cumulative deviation. An H-optimal shield is computed which minimizes at each step the expected value of

cumulative deviation for the coming H-steps of execution of shield. Such shield are called as *H-optimally deviation minimizing*.

At last, we analyze a method for producing a shield from given correctness requirement, hard deviation constraints and a horizon integer value. The synthesized shield satisfy both REQ(I, O) and HDC and is also H-optimally deviation minimizing. The complete synthesis is done using the tool DCSynth. At last we formulate several types of shields such as k-stablizing shield, burst-error shield and a new notion of e,d-shield.

In later half, we explore reactive synthesis which automates the task of computing finite-state machines. We only need to write the specification and implementation is computed automatically. Out of multiple approaches, generalized reactivity(1) synthesis is commonly used. It is also known as GR(1) synthesis. The approach behind reactive synthesis is to take all the requirements of the desired implementation in the specification, and to then accept any implementation that fulfil the requirements. In many cases, without providing proper functions, properties such as few states or quick response cannot be captured correctly in the specification.

We explore tool **slugs** which provide a framework for generalized reactivity(1) synthesis and its modifications. The slugs tool has simple core implementation of generalized reactivity(1) synthesis algorithm that can be extended further using multiple user-written plugins. The main focus of the tool is on readability and conciseness of the code to make it simple and easier for the algorithms to be adapted for specific domains and to allow the users to use multiple plugins at the same time, where each one modifies only a small part of the synthesis process.

1.3 Organization of The Report

The remaining chapters are organized as follows: **Chapter 2** discuss all the preliminaries that are required in the algorithms and case studies. **Chapter 3** describes the idea and the algorithms that we worked on in this report. **Chapter 4** describes the implementation of

multiple case studies and examples. **Chapter 5** describes the results that we found when we run them on our machine. **Chapter 6**, contains the final remark of our work.

Chapter 2

Preliminaries

The following section contains a brief syntax and semantics [Pan00] [PW19] of QDDC logic as well as guided H-optimal controller synthesis with soft requirements that will be used in the later sections.

2.1 QDDC Logic & Syntax

Consider V as a finite set of propositional terms or variables. Over the alphabet 2^{V}, consider σ to be a non-empty word having the form $\sigma = \mathrm{V}_0 \cdots \mathrm{V}_\mathrm{m}$ where $\mathrm{V}_\mathrm{j} \subseteq \mathrm{V} \; \forall \; \mathrm{j} \in \{0,\cdots,\mathrm{m}\}$. Let $dom(\sigma) = \{0,\cdots,\mathrm{m}\}$, $len(\sigma) = \mathrm{m}+1$, $\mathrm{s[j]} = \mathrm{V_j}$ and $\sigma[\mathrm{i, j}] = \mathrm{V_i} \cdots \mathrm{V_j}$.

For a propositional formula, the syntax over variables V is given by:

$\phi :=$ true $\mid \phi \mid\mid \phi \mid \phi \mid \mathrm{p} \in \mathrm{V} \mid \phi \;\&\&\; \phi \mid$ false.

Let the propositional formulas set over variables V be $\Omega(V)$. Then the propositional formula j satisfaction at i, is given by $\sigma, i \vDash \phi$.

QDDC formula syntax over variables V is given by:

$\mathrm{Q} := \phi \mid [[\phi]] \mid [\phi] \mid \mathrm{Q} \wedge \mathrm{Q} \mid \mathrm{Q} \;\&\&\; \mathrm{Q} \mid \mathrm{Q} \mid\mid \mathrm{Q} \mid !\mathrm{Q}$

all p.Q | ex p.Q | sdur $\phi \bowtie$ q | slen $\bowtie$ q | scount $\phi \bowtie$ q

where $\phi \in \Omega(V)$, q $\in$ N, p $\in$ V and $\bowtie \in \{<, \leq, =, >, \geq\}$.

2.2 QDDC Semantics

Over σ, [a, b] is called an interval where a, b $\in dom(\sigma)$ and a $\leq$ b. Consider over σ, $Int(\sigma)$ to be the all intervals set. Consider $\sigma \in 2^{\mathrm{V}}$, [a, b] $\in Int(\sigma)$ be an interval. Then, for the QDDC formula Q, σ,[a, b] $\vDash$ Q, which is satisfaction of the formula is defined as follows:

σ,[a, b] $\vDash \langle\phi\rangle$ iff a = b and σ, a $\vDash \phi$,

σ,[a, b] $\vDash [\phi]$ iff a < b and $\forall$ a $\leq$ i < b : σ, i $\vDash \phi$,

σ,[a, b] $\vDash [[\phi]$ iff $\forall$ a $\leq$ i $\leq$ b : σ, i $\vDash \phi$,

σ,[a, b] $\vDash$ Q1 $\wedge$ Q2 iff $\exists$ a $\leq$ i $\leq$ b : σ,[a, i] $\vDash$ Q1 and σ,[i, b] $\vDash$ Q2.

slen, scount and sdur are some of the entities called as terms. The slen term gives the interval length in which it is measured. The scount term ϕ, where $\phi \in \Omega(V)$, gives the position count where ϕ holds including the endpoints of the interval.

Here are some of the derived terms:

ext = !pt, pt = $\langle true\rangle$, $\langle\rangle$Q = true $\wedge$ Q $\wedge$ true,

pref(Q) = !((!Q) $\wedge$ true) and []Q = (!$\langle\rangle$!Q).

Hence, σ, [a, b] $\vDash$ []Q iff σ, [a', b'] $\vDash$ Q $\forall$ sub-intervals a $\leq$ a' $\leq$ b' $\leq$ b.

σ, [a, b] $\vDash$ pref(Q) iff σ, [a, b'] $\vDash$ Q $\forall$ prefix intervals a $\leq$ b' $\leq$ b.

Consider $\sigma \vDash$ Q iff σ, [0, $len(\sigma)$ - 1] $\vDash$ Q and σ, i $\vDash$ Q iff σ, [0, i] $\vDash$ Q.

Now language of Q, L(Q) is the set of behaviours accepted by Q. L(Q) = $\{\sigma \mid \sigma \vDash Q\}$.

Here σ, i $\vDash$ Q denotes that the formula Q is satisfied by the past of position i.

Consider an indicator variable y for a QDDC formula. It witnesses, at any point in execution, the truth of a formula Q. Thus, Indicator(Q,y) = pref(EP(y) $\Leftrightarrow$ Q), where $Indicator$ is indicator symbol. Given σ and i, σ, i $\vDash$ EP(y) iff y $\in \sigma$[i], hence EP(y) = ($true \wedge \langle y\rangle$). If σ $\vDash$ Indicator(Q,y) then $\forall$ i, σ, i $\vDash$ Q iff y $\in \sigma$[i]. Hence, where past of the position satisfies Q, y exactly is true at those points. These are used with the cascade composition with some other formulas of Q which provides a measure to generate a formula which describe other regular properties using propositions W.

2.3 Cascade Composition

Consider QDDC formulas $Q_1,\cdots,Q_j$ over I/O variables (I, O) and let $W = \{w_1,\cdots,w_j\}$ to be the indicator variables list given $(I \cup O) \cap W = \varnothing$.

Over $(I \cup O \cup W)$, let Q be a formula. Now, the equivalent QDDC formula with its cascade composition $\ll$ is defined as:

$$E \ll \langle Indicator(Q_1, w_1),\cdots,Indicator(Q_j, w_j)\rangle = Q \wedge \bigwedge_{i=1}^{j} \text{pref}(EP(w_i) \Leftrightarrow Q_i).$$

This cascade composition gives an expression formula over I/O variables (I, $O \cup W$) where *Indicator* is indicator symbol.

2.4 Supervisors and Controllers

Supervisors along with controllers are simply mealy machines with additional properties for deciding the transitions between the states. Considering QDDC formulas and automata where variables $V = I \cup O$ are divided into disjoint sets of input and output variables as I and O respectively, allows us to accurately compute controllers and supervisors in the tool DCSynth.

2.4.1 Output Non-Deterministic Mealy Machines

A DFA over I/O alphabet Σ is a tuple D = (Q, Σ, a, δ, F), with $\delta : Q \times 2^I \times 2^O \to Q$, $\Sigma = 2^I \times 2^O$.

An output non-deterministic mealy machine is a DFA with F = Q - {z} and δ(z, j, o) = z for all $j \in 2^I$, $o \in 2^O$ where z is a unique reject (or non-final) state.

A Mealy machine is deterministic if $\forall$ a $\in$ F, $\forall$ j $\in 2^I$, $\exists$ at most one $o \in 2^O$ s.t. δ(a, j, o) $\neq$ z. An output non-deterministic mealy machine is non-blocking if $\forall$ a $\in$ F, $\forall$ j $\in 2^I$ $\exists$ $o \in 2^I$ s.t. δ(a, j, o) $\in$ F.

2.4.2 Controllers and Supervisors

A supervisor is an output non-blocking non-deterministic Mealy machine and a controller is a supervisor with deterministic nature.
In a supervisor, the non-deterministic output choices denotes unresolved decision. The deterministic ordering of supervisors leads to their refinement to controllers.
Given Sup1, Sup2 as two supervisors, Supervisor2 is more deterministic than Supervisor1 and is a sub-supervisor of Supervisor1, denoted Supervisor1 $\leq_{det}$ Supervisor2, iff L(Supervisor2) $\subseteq$ L(Supervisor1).

2.5 Slugs Specification

The slugs tool contains GR(1) synthesis framework, having GR(1) synthesis algorithm small core implementation, which can be extended further using user-written plugins. Multiple plugins can be used with the slugs architecture at the same time, wherein each plugin for the synthesis process, only modifies a part.

The synthesis tool of slugs computes implementations from their specifications. A specification is a text file in one of the two formats:

- The basic slugs-in format or
- The more easier to use structured-slugs format.

The structured-slugs format is an extension of the slugs-in format.

2.5.1 Slugs-in Syntax

Any line starting with a # symbol in a slugs-in file is a comment line. Empty lines are ignored. Before the first section header, there can only be empty or comment lines.
Constraints are denoted by the non-comment and non-empty lines in the assumption and guarantee sections and are always given on separate lines. Constraints are written as Boolean formulas, in which the operators are written before the operands and all non-unary

operators are assumed to be binary. There is also always a space between operators and operands with no next operator. If a constraint wants to refer to a variable value at the transition end, it is done by adding a ' to the variable name. The boolean operators used for specifying constraints are ! (not), | (or), (and), and ∧ (exclusive or). For the implication (→) and equivalence (↔) part, there are no designated operators in the slugs-in input language, but they can be simulated by the two-operator sequences "| !" and "! ∧".

2.5.2 Structured Slugs Syntax

The basic structure of a structured-slugs specification file is mostly same as for a basic slugs-in format. There are two extensions:

- Using infix notation in the constraints, and
- Availability of non-negative integer variables with domains constraints, with their additions and comparisons.

Hence, the structured-slugs format allows us to write expressions in a more human readable format. E.g. instead of writing | a' ! | x y, the user can simply write a' | !(x|y) which is more readable. Implications, bi-implication, and braces are also supported, hence (! a' → !(x|y)) is also a valid formula. In the infix and Polish prefix form, it is allowed to mix constraints, given that they are all on separate lines. Operator precedence: unary operations bind strongest, then *and* and then *or*.

Chapter 3

Methodology

Till now, we have looked at the basic terminologies that would be useful in understanding the report. We now turn our attention towards our main idea. We start by briefly discussing the algorithms used in the implementation. This includes algorithm for implementing the run-time enforcement shield using QDDC which is an interval temporal logic.

3.1 Algorithms Used

This section briefly elaborate the algorithms that were employed in our implementation.

3.1.1 Run-time enforcement shield

We discuss the algorithm [PW19] in this section which will be used to implement the run-time enforcement shield for the system in which errors occur irregularly.

Given a correctness/critical requirements and hard deviation constraints as a QDCC formulas over input/output proposition. A SSE takes input and produce correct output but it sometimes violate the correctness/critical requirement REQ.

Fig 3.1 shows a shield which takes input and output from SSE and produce correct output $O^{'}$. On top of this, it also ensures that shield output must deviate from that of system as little as possible. Along with correctness requirement, hard deviation constraints

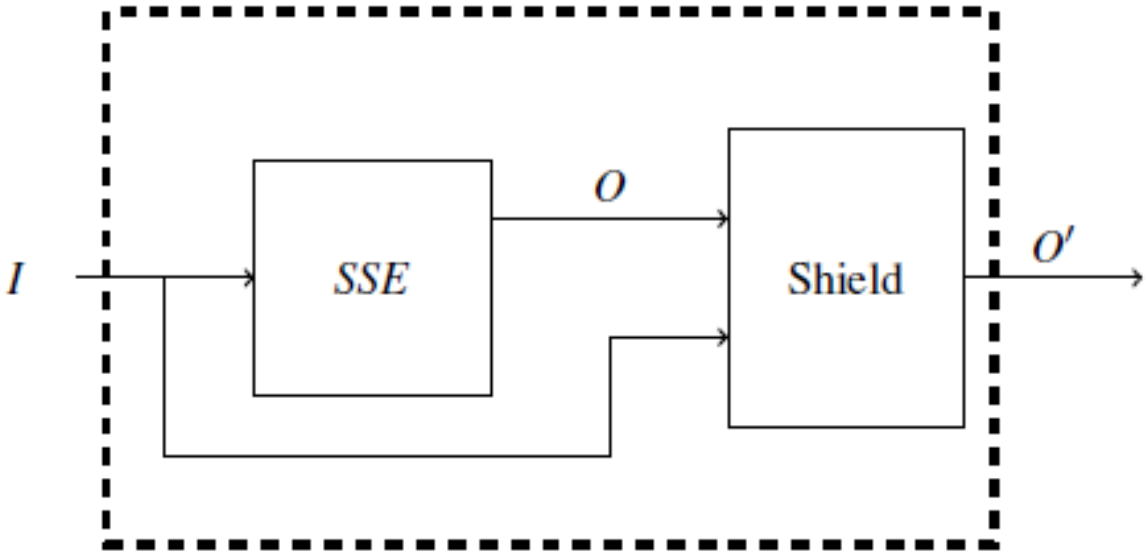

Fig. 3.1: Run-time enforcement shield

HDC are specified using QDCC formulas to specify the deviation allowed in a behaviour of shield. To minimize the cumulative deviation, $Hamming(O, O^{'})$, a soft deviation constraint is specified.

Hard Deviation Constraints HDC

It is formulated using two indicator propositions, SSE_OK and $Deviation\_Shield$. proposition $Deviation\_Shield$ is true if SSE output is different from shield output at the current position otherwise false. Proposition SSE_OK is true if correctness/critical requirement is satisfied by SSE at the current position. We use two indicator definitions in formulating HDC:

$$IND\_DEFINITION = \{Indicator(REQ(I,O),\ SSE\_OK),\ Indicator(true \wedge \langle \vee_i (o_i \neq o_i^{'}) \rangle, Deviation\_Shield)\}$$

A HDC is a QDCC logic consisting of $Deviation_Sshield$ and SSE_OK propositions. We define HDC_Shield as a QDCC formula using propositions $(I \bigcup O, O^{'})$ and cascade composition to modularize the specifications into two modules: HDC and REQ.

$$HDC\_Shield = REQ(I, O^{'}) \wedge HDC(SSE\_OK, Deviation\_Shield) \ll IND\_DEFINITION$$

DCSynth tool will give us a MPS(HDC_Shield), a maximally permissive supervisor satisfying the constraint HDC_Shield, it is denoted by MPS(REQ, HDC) and called as shield-supervisor without deviation minimization.

Soft Deviation Constraint

By using DCSynth tool, deviation in MPS supervisor can be optimized further by using soft requirement consisting of weights. It maximizes the soft requirement's cumulative weight expected value for the coming H-steps of execution and then averaged it over all the H length input sequences. DCSynth soft requirement formula to minimize the cumulative deviation for the coming H steps is:

$$Hamming\_Distance(O, O^{'}) = \langle (T \wedge \langle o_1 = o_{1'} \rangle) : 1, \cdots, (T \wedge \langle o_r = o_{r'} \rangle) \rangle : 1 \rangle$$

In above equation, T stands for true. If SSE output and shield output does not deviate at current position, then it contribute a reward a 1 and this summation gives a weight of the soft requirement Hamming(O,$O^{'}$). If SSE output sequence and shield output sequence are same at the step of execution then the weight of $Hamming(O, O^{'})$ will be r. If they differ is say q variables, then the weight at step n would be r-q. Sub-supervisor can be obtained by using the DCSynth tool as follows:

$$MPHOS(MPS(REQ, HDC),\ Hamming\_Distance(O, O^{'}),\ H)$$

of the MPS supervisor by using soft requirement Hamming_Distance(O,$O^{'}$) and horizon integer value which retains only the outputs that maximizes the soft requirement's expected weight for the coming H steps. This sub-supervisor is represented by MPHOS(REQ, HDC, H) and called as shield-supervisor with deviation minimization.

Determinization

Both MPHOS and MPS are non-deterministic supervisors i.e. multiple output choices are available that satisfy the HDCs while being H-optimal. To resolve this issue, we need to provide a preference order **ord** on the shield outputs. To obtain the deterministic controller, we maintain the output from the output choices we get from non-deterministic supervisors, which has the highest value according to preference order provided by the user.

So, given a REQ(I, O) (correctness requirements), HDC (hard deviation constraints) to be satisfied by the shield, a value H (horizon value) to globally minimizes the deviation for the coming H steps of execution, and a order of preference ord, the tool DCSynth synthesizes the shields $Det_{ord}(MPS(REQ, HDC))$ called as shield with no deviation minimization and $Det_{ord}(MPHOS(REQ, HDC, H))$ called as shield with deviation minimization.

Types of Shield and its Hard Deviation Constraints HDC

1. **Burst-shield:**

   True.

   It does not enforce any HDC.

2. **k-shield:**

   $[](([[Deviation\_Shield]] \Rightarrow slen < k)$.

   It says that deviation may occur for at most k cycles continuously.

3. **k-stabilizing shield:**

   $[](([[Deviation\_Shield]] \Rightarrow slen < k)\&\&([](((\langle !Deviation\_Shield\rangle \wedge [[SSE\_OK]] \Rightarrow [[!Deviation\_Shield]]))$.

   It specifies that the deviation occur as long as SSE system produce errors and for at most k cycles after SSE recovers from deviation.

4. ***e,d* shield:**

$$[](((scount!SSE\_OK \Leftarrow e) \Rightarrow (scountDeviation\_Shield \Leftarrow$$
$$d)\&\&([](((\langle !Deviation\_Shield\rangle[[SSE\_OK]] \Rightarrow [[!Deviation\_Shield]]))$$

It specifies that if the error count by system is at most e in any interval then the number of cycles having deviation is at most d.

3.1.2 Slugs Tool

The slugs tool [ER16] computes automaton from the specifications consists of multiple sections, begin by section headers. Following are section headers used in specifying specification:

- **[INPUT]:** It consists of lines having atomic input propositions.
- **[OUTPUT]:** It consists of lines having atomic output propositions.
- **[ENV_INIT]:** It consists of lines having environment initialization assumptions.
- **[SYSTEM_INIT]:** It consists of lines having system initialization guarantees.
- **[ENV_TRANS]:** It consists of lines having environment safety assumptions.
- **[SYSTEM_TRANS]:** It consists of lines having system safety guarantees.
- **[ENV_LIVENESS]:** It consists of lines having environment liveness assumptions.
- **[SYSTEM_LIVENESS]:** It consists of lines having system liveness guarantees.

Chapter 4

Implementation

In this chapter, we shall look at the examples which we convert from LTL to QDCC format and one case study of traffic light. Later, we shall look at the implementation of few real life examples that we implement using the slugs tool. We start by the example of water reservoir which maintain the level of water inside the chamber. Next, we discuss the example of single robot moving in a grid. Then we discuss the example in which the specification is unrealizable. Syntax and semantics of writing the specification in slugs format is giving in the chapter 3.

4.1 Run-time enforcement shield

As a first step to compute run-time enforcement shield we need specifications in the format of QDCC formulas. Then the specification as QDCC formulas is given to DCSynth tool to compute run-time enforcement shield.

We convert different specifications into QDCC formulas to be given input to DCSynth tool:

4.2 QDDC Examples

Here are some of the examples with their specification explanation and they are converted to QDDC language:

- If Load-R follows Store-R then Store-S should occur before Load-S.

$$\neg\Diamond(\lceil Store - R\rceil^0 \wedge \lceil\lceil\neg Store - S\rceil\rceil \wedge (true \wedge \lceil Load - R\rceil^0 \wedge true \wedge \lceil Load - S\rceil^0))$$

- If A is holding a shared variable and B is requesting, then it is not possible for A to release control and then get back the shared variable, before B gets control of the shared variable.

$$\Box(\lceil AHold \wedge BReq\rceil^0 \wedge \lceil\lceil\neg BHold\rceil \Rightarrow\rceil\Diamond(\lceil\lceil\neg AHold\rceil \Rightarrow ptpAHold))$$

- If A requests the shared variable and the shared variable is free, then A gets control of the shared variable within 10 clock cycles unless B or C request the shared variable in the mean-time.

$$\Box(\lceil AReq \wedge SharedVarFree\rceil^0 \wedge \lceil\lceil\neg(BReq \vee CReq)\rceil\rceil \wedge \eta = 9 \Rightarrow \Diamond\lceil AGrant\rceil^0)$$

- If speed sensors indicates impossibly high speed at least five times and no RESET event occurs, then ERROR will be generated.

$$\Box(((\Sigma ImpHighSpeed \leq 4) \wedge \lceil ImpHighSpeed\rceil^0 \wedge \lceil\lceil\neg RESET\rceil\rceil) \Rightarrow \Diamond\lceil ERROR\rceil^0)$$

4.3 LTL to QDDC: Traffic Light Example

LTL is linear temporal logic used to specify a variety of specifications. Let R, Y, G denote the red, yellow, green boolean signals of traffic light respectively.

- No two traffic lights can blink simultaneously.

LTL: $\square((R \rightarrow \neg(Y \vee G)) \;\&\&\; (Y \rightarrow \neg(R \vee G)) \;\&\&\; (G \rightarrow \neg(R \vee Y)))$

QDDC: $\square((R \rightarrow \neg(Y \vee G)) \;\&\&\; (Y \rightarrow \neg(R \vee G)) \;\&\&\; (G \rightarrow \neg(R \vee Y)))$

- Once red, the light cannot become green immediately.

LTL: $\square(R \rightarrow \neg \circ G)$

QDDC: $\square(\lceil R \rceil^0 \wedge \eta = 2 \rightarrow true \wedge \lceil \neg G \rceil^0)$

- Once red, the light always becomes green eventually after being yellow for some time.

LTL: $\square(R \rightarrow (\Diamond G \wedge (\neg G \cup Y)))$

QDDC: $\square((\lceil R \rceil \rightarrow true \wedge \lceil Y \rceil) \wedge (\lceil Y \rceil \rightarrow true \wedge \lceil G \rceil \wedge true) \;\&\&\; \eta \geq 30)$

4.4 Slugs Case Studies

We used a tool called slugs for implementation of controller using the specification for some of the real life examples.

4.4.1 Water Reservoir

```
[INPUT]
inflow1
inflow2

[OUTPUT]
level: 3...107
outflow
```

```
[SYSTEM_TRANS]
(inflow1 & inflow2 & outflow) -> (level' = level+1)
(inflow1 & inflow2 & !outflow) -> (level' = level+4)
(inflow1 & ! inflow2 & outflow) -> (level'+1 = level)
(inflow1 & ! inflow2 & !outflow) -> (level' = level+2)
(!inflow1 & inflow2 & outflow) -> (level'+1 = level)
(!inflow1 & inflow2 & !outflow) -> (level' = level+2)
(!inflow1 & ! inflow2 & outflow) -> (level'+3 = level)
(!inflow1 & ! inflow2 & !outflow) -> (level' = level)
(level'<=100)
(level'>=10)

[SYSTEM_INIT]
! outflow
level = 10

[ENV_INIT]
! inflow1
! inflow2

[ENV_TRANS]
| ! inflow1 ! inflow1'
| ! inflow2 ! inflow2'
```

Explanation: Input and output is controlled by environment and system respectively. In this example, water is coming into the chamber through two pipes and leaving the chamber through one pipe, so we have two inputs inflow1 and inflow2 for two inward pipes and one output outflow for outward pipe leaving the chamber and another output for the level of

water in the chamber which can have value between 3 and 107. We initialized the system with level of the water in the chamber with 10m and outflow as false (outward pipe is closed). Environment is initialized with both inward pipes as closed (inflow1 and inflow2 is false).

Here prime in front of variable indicates the value of that variable in the next step of execution. Inward pipe increase the level of water by 2 unit and outward pipe decrease the level of water by 3 unit in one step of execution. In first system transitions, both inward and outward pipes are open, so level of water will increase by 2+2-3 = 1 unit in that step of execution. In second transition, both inward pipes are open only, so level of water will increase by 2+2 = 4 unit. In third transition, one inward and outward pipes are open, so level of water will increase by 2-3 = -1 unit i.e. level will decrease by 1 unit as subtraction is not allowed in structured slugs syntax so we shift 1 in opposite side of the equation as addition. In fourth transition, only one inward pipe is open, so level of water will increase by 2 unit. Fifth and sixth transitions are same as third and fourth transitions respectively. In seventh transition, only outward pipe is open, so level of water will decrease by 3 unit. In eighth transition, all pipes are closed, so level of water remains unchanged.

4.4.2 Single Robot Scenario

```
[INPUT]
door1
door2

[OUTPUT]
mrx:0...7
mry:0...5

[SYSTEM_TRANS]
```

```
mrx+1>=mrx'

mrx'+1>=mrx

mry+1>=mry'

mry'+1>=mry


## no Collisions with the obstacles

mrx'!=1 | mry'>4 | mry'<2

mry'!=4 | mrx'<3 | mrx'>6

mry'!=3 | mrx'<6 | mrx'>6

mry'!=2 | mrx'<3 | mrx'>3

mry'!=1 | mrx'<3 | mrx'>5


## no Collisions with the doors

mry'!=5 | mrx'!=3 | door1'

mry'!=0 | mrx'!=4 | door2'


[ENV_TRANS]


[SYSTEM_LIVENESS]

mrx'=0 & mry'=0

mrx'=7 & mry'=0


[ENV_LIVENESS]

## Doors are infinitely often open

door1'

door2'
```

```
[SYSTEM_INIT]
mrx = 0
mry = 0

[ENV_INIT]
```

Explanation: Input and output is controlled by environment and system respectively. Here prime in front of variable indicates the value of that variable in the next step of execution. In this example, there are two doors controlled by the environment and two outputs storing the x and y coordinates of the robot. Robot can move in a grid of size 7 x 5 i.e. x coordinate can range from 0 to 7 units and y coordinate can range from 0 to 5 units.
System initializes x and y coordinate of the robot to (0,0). There are two doors at position (3,5) and (4,0) through which robot can move out of the grid. There are multiple obstacles at various positions: (3-6,4), (1,2-4), (3,2), (3-5,1) and (3,6). Robot has to move such that it does not collide with the obstacles and the door while it is closed. As doors are controlled by the environment, so environment liveness property states that door1 and door2 must open infinitely often so robot can move out of the grid finally. Above is the complete specification taking into account all the requirements.

4.4.3 Unrealizable Scenario

```
[INPUT]
c
d

[OUTPUT]
u
v
```

```
[SYSTEM_INIT]
! u
! v

[ENV_INIT]
! c
! d

[SYSTEM_TRANS]
| ! u ! v

[SYSTEM_LIVENESS]
& c d

[ENV_LIVENESS]
& c ! d
& d ! c
```

Explanation: Input and output is controlled by environment and system respectively. In this example, c and d are inputs and u and v are outputs. System initializes u and v to false and environment initializes c and d also to false. System transition states that u and v they cannot be true at the same step of execution. **Liveness property is the property that has to be true infinitely often**.

Why the specification is unrealizable?

System liveness property states the c and d need to be true together infinitely often but c and d are inputs controlled by the environment so its upto the environment whether it

makes c and d true together or not. Environment can generate a sequence of inputs in which c and d are not true together at any step of execution making the specification unrealizable as system liveness property does not hold true.

4.4.4 Simple Safety Example

```
[INPUT]
c
d

[OUTPUT]
e

[SYSTEM_INIT]
e

[ENV_INIT]
| ! c ! d

[SYSTEM_TRANS]
^ e' c'

[ENV_TRANS]
| c' d'
```

Explanation: Input and output is controlled by environment and system respectively. In this example, c and d are inputs and e is the output. System initializes the output e to true and environment initializes the input such that both are not true i.e. either True and False

or False and False or False and True. Environment transition states that the next values (the value in the next step of execution) of c and d cannot be both false. System transition states that in the next step of execution either e or c must be true.

Chapter 5

Results

In this chapter, we explore the results that was observed when we run the DCVALID tool on the traffic light case study discuss in the section 4.2 and later, we explore the results that was observed when we run the slugs tool on the examples discuss in the section 4.4.

5.1 QDDC Results: Traffic Light Example

Here are some of the results for traffic light example using DCVALID tool. It checks for the realizability for the specification and then generates a counter example and a satisfying example for the same.

5.1.1 1. No two traffic lights can blink simultaneously

The generated automaton for the specification 1 contains a total of 5 states, space taken 1 MB and 8-BDD nodes. A counter example of least length 2 is (0,1,1),(X,X,X) where (R,Y,G) denotes values of red, yellow and green respectively. X denotes value can be 0 or 1 both. A satisfying example of least length 1 is (0,0,X).

```
samay@samay-VM:~/Documents/dcvalid2/dccheck200$ ./dcvalid traffic_1
DCVALID 1.4

MONA v1.4 for WS1S/WS2S
Copyright (C) 1997-2000 BRICS

PARSING
Time: 00:00:00.00

CODE GENERATION
DAG hits: 328, nodes: 50
Time: 00:00:00.00

REDUCTION
Projections removed: 2 (of 6)
Products removed: 4 (of 29)
Other nodes removed: 1 (of 14)
DAG nodes after reduction: 42
Time: 00:00:00.00

AUTOMATON CONSTRUCTION
100% completed
Space: 1MB
Time: 00:00:00.00

Automaton has 5 states and 8 BDD-nodes

ANALYSIS
A counter-example of least length (2) is:
R               X 0X
Y               X 1X
G               X 1X

R = {}
Y = {0}
G = {0}

A satisfying example of least length (1) is:
R               X 0
Y               X 0
G               X X

R = {}
Y = {}
G = {}

Total time: 00:00:00.00
samay@samay-VM:~/Documents/dcvalid2/dccheck200$ ./dcautps traffic_1 traffic_1.ps
```

Fig. 5.1: Traffic Light: Spec 1

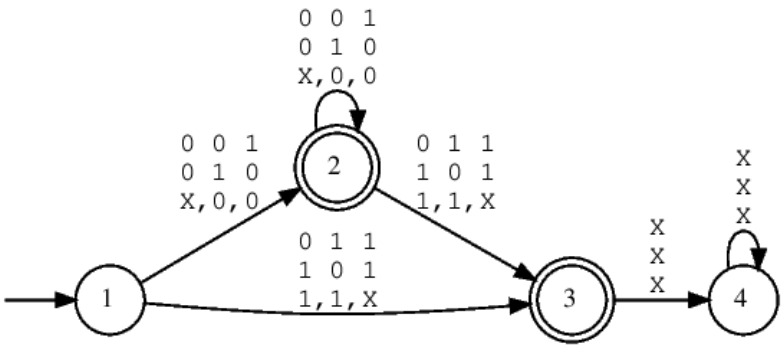

Fig. 5.2: Traffic Light: Automata Spec 1

5.1.2 2. Once red, the light cannot become green immediately.

The generated automaton for the specification 2 contains a total of 6 states, space taken 1 MB and 8-BDD nodes. A counter example of least length 3 is (1,X,X),(X,X,X),(X,1,X) where (R,Y,G) denotes values of red, yellow and green respectively. X denotes value can be 0 or 1 both. A satisfying example of least length 1 is (0,X,X).

```
samay@samay-VM:~/Documents/dcvalid2/dccheck200$ ./dcvalid traffic_2
DCVALID 1.4

MONA v1.4 for WS1S/WS2S
Copyright (C) 1997-2000 BRICS

PARSING
Time: 00:00:00.00

CODE GENERATION
DAG hits: 102, nodes: 75
Time: 00:00:00.00

REDUCTION
Projections removed: 3 (of 13)
Products removed: 6 (of 41)
Other nodes removed: 1 (of 20)
DAG nodes after reduction: 64
Time: 00:00:00.00

AUTOMATON CONSTRUCTION
100% completed
Space: 1MB
Time: 00:00:00.00

Automaton has 6 states and 8 BDD-nodes

ANALYSIS
A counter-example of least length (3) is:
R               X 11X
Y               X XXX
G               X X1X

R = {0,1}
Y = {}
G = {1}

A satisfying example of least length (1) is:
R               X 0
Y               X X
G               X X

R = {}
Y = {}
G = {}

Total time: 00:00:00.00
samay@samay-VM:~/Documents/dcvalid2/dccheck200$ ./dcautps traffic_2 traffic_2.ps
```

Fig. 5.3: Traffic Light: Spec 2

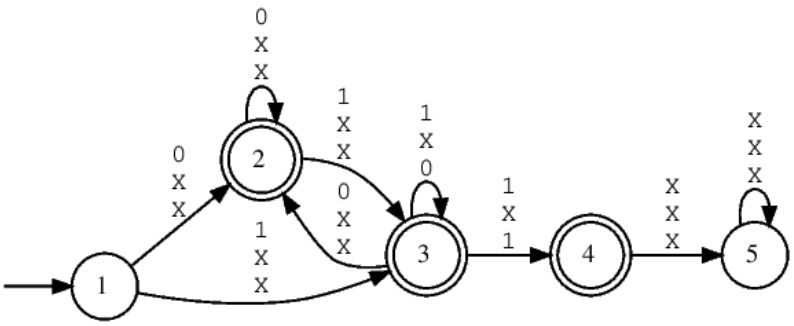

Fig. 5.4: Traffic Light: Automata Spec 2

5.2 Case Studies for Slugs Tool

5.2.1 Water Reservoir

In this example, since the specification is written in structured-slugs format due to the use of integer variables, the first command in fig 5.5 generates a slugs-in file from structured-slugs format. Then the explicit Strategy parameter is used for generating the automaton in json format as seen in the fig 5.5 and 5.6.

As seen in the fig 5.5, total number of variables used are 10. Out of them, inflow1 and inflow2 are input boolean variables, outflow is output boolean variable. As we have seen in the specification, there is one more output variable level which is a integer variable ranging from 3 to 107, It requires 7 bits to store maximum level 107 so there are 7 internal variables to store one bit each of the level of the water. Total number of automaton states are 190 and each state has valuation of each variable and its adjacency list. Thus an automaton can be generated using these.

5.2.2 Single Robot Scenario

In this example, the first command in fig 5.7 generates a slugs-in file from structured-slugs format. Then the explicit strategy parameter is used for generating the automaton using the json plugin as seen in the fig 5.7 and 5.8.

As seen in the fig 5.7, total number of variables used are 8. Out of them, door1 and door2

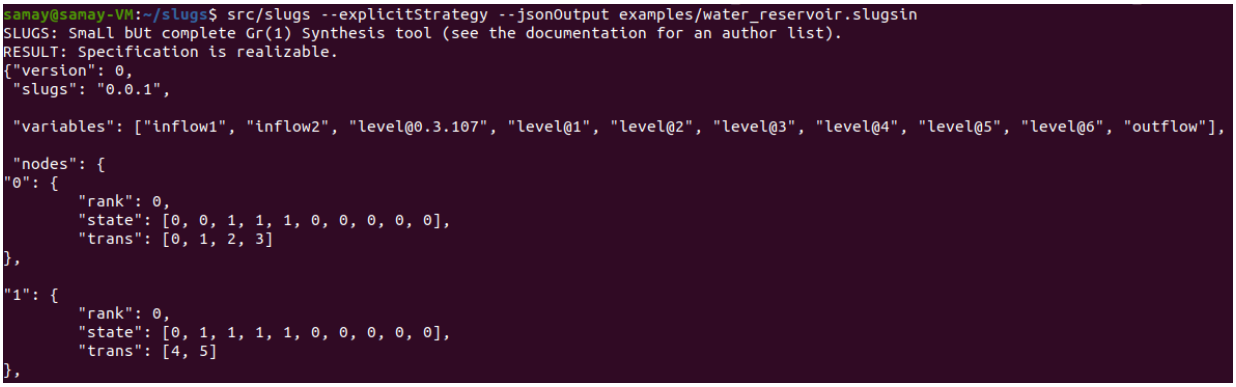

Fig. 5.5: Water Reservoir

Fig. 5.6: Water Reservoir

are boolean input variables, and x coordinate can vary from 0 to 7 i.e. it requires 3 bits to represent the x coordinate and y coordinate can vary from o to 5 i.e. it also requires 3 bits to represent the y coordinate so total 6 bits are required to represent the x and y coordinate of the robot. Total number of automaton states are 51 and each state has valuation of each variable and its adjacency list. Thus an automaton can be generated using these.

5.2.3 Unrealizable Scenario

In this example, the specification is unrealizable as seen in the fig 5.9. There are 4 variables, c and d are inputs and u and v are outputs. Here the environment liveness tries to make exactly either of c or d as $true$ and other as $false$, but the system liveness is to make both c and d as $true$. As c and d both are controlled by the environment so there can be a input sequence in which c and d both are not true at same time making the system liveness property false.

```
samay@samay-VM:~/slugs$ tools/StructuredSlugsParser/compiler.py examples/single_robot_scenario.structuredslugs > examples/single_robot_scenario.slugsin
samay@samay-VM:~/slugs$ src/slugs --explicitStrategy --jsonOutput examples/single_robot_scenario.slugsin
SLUGS: SmaLl bUt complete Gr(1) Synthesis tool (see the documentation for an author list).
RESULT: Specification is realizable.
{"version": 0,
 "slugs": "0.0.1",

 "variables": ["door1", "door2", "mrx@0.0.7", "mrx@1", "mrx@2", "mry@0.0.5", "mry@1", "mry@2"],

 "nodes": {
"0": {
        "rank": 0,
        "state": [0, 0, 0, 0, 0, 0, 0, 0],
        "trans": [4, 5, 6, 7]
},

"1": {
        "rank": 0,
        "state": [0, 1, 0, 0, 0, 0, 0, 0],
        "trans": [4, 5, 6, 7]
},

"2": {
        "rank": 0,
        "state": [1, 0, 0, 0, 0, 0, 0, 0],
        "trans": [4, 5, 6, 7]
},
```

Fig. 5.7: Single Robot Scenario

```
"50": {
        "rank": 0,
        "state": [1, 0, 1, 0, 0, 0, 0, 0],
        "trans": [4, 5, 6, 7]
},

"51": {
        "rank": 0,
        "state": [1, 1, 1, 0, 0, 0, 0, 0],
        "trans": [4, 5, 6, 7]
}

}}
```

Fig. 5.8: Single Robot Scenario

5.2.4 Simple Safety Example

In this example, as there is no integer input/output variable so the file is already is slugs-in format. Then the explicit strategy parameter is used for generating the automaton using the json plugin as seen in the fig 5.10 and corresponding automaton is shown in 5.11.
As seen in the figure 5.10, total number of variables used are 3: 2 for boolean input c and d and one for boolean output e. Total number of states are 5. As system transition states that in next step of execution, either e or c must be true (not both), we can see that every state has 1, 3, 4 as its neighbours and these states has either e as true and c as false or e as false and c as true. Environment transition states that in next step of execution, both c and d cannot be false together, we can see every state has 1, 3, 4 as its neighbours and these states has either c or d as true.

```
samay@samay-VM:~/slugs$ src/slugs --explicitStrategy --jsonOutput examples/unrealizable1.slugsin
SLUGS: SmaLl bUt complete Gr(1) Synthesis tool (see the documentation for an author list).
RESULT: Specification is unrealizable.
```

Fig. 5.9: Unrealizable Scenario

```
samay@samay-VM:~/slugs$ src/slugs --explicitStrategy --jsonOutput examples/simple_safety_example.slugsin
SLUGS: SmaLl bUt complete Gr(1) Synthesis tool (see the documentation for an author list).
RESULT: Specification is realizable.
{"version": 0,
 "slugs": "0.0.1",

 "variables": ["a", "b", "c"],

 "nodes": {
"0": {
        "rank": 0,
        "state": [0, 0, 1],
        "trans": [1, 3, 4]
},

"1": {
        "rank": 0,
        "state": [0, 1, 1],
        "trans": [1, 3, 4]
},

"2": {
        "rank": 0,
        "state": [1, 0, 1],
        "trans": [1, 3, 4]
},

"3": {
        "rank": 0,
        "state": [1, 0, 0],
        "trans": [1, 3, 4]
},

"4": {
        "rank": 0,
        "state": [1, 1, 0],
        "trans": [1, 3, 4]
}

}}
```

Fig. 5.10: Simple Safety Example

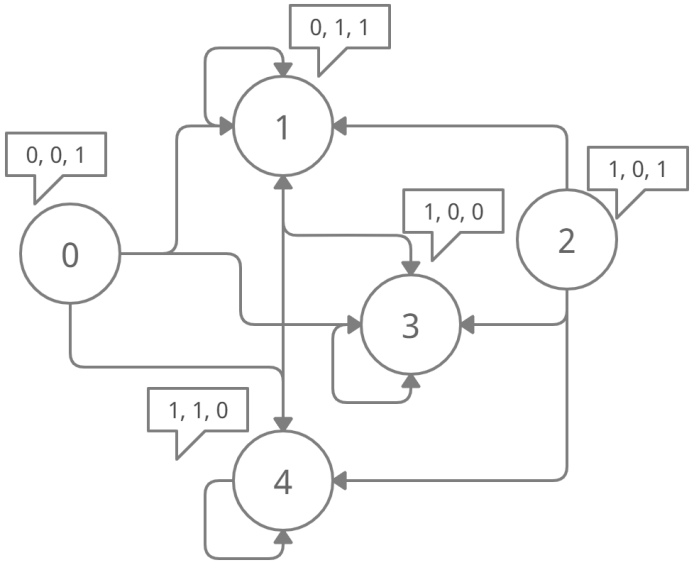

Fig. 5.11: Automata for simple safety example

Chapter 6

Conclusion

6.1 Results

In this paper, we explored the run-time enforcement shield using the DCSynth tool which takes a correctness/critical requirement REQ(I,O), hard deviation constraint HDC (depending on the type of shield to enforce such as e,d shield, k-burst shield etc.), a integer value H called as horizon value to globally minimize the deviation of the shield over the next H steps and a preference order, the tool synthesizes the shields $Det_{ord}(MPS(REQ, HDC))$ called as shield with no deviation minimization and $Det_{ord}(MPHOS(REQ, HDC, H))$ called as shield with deviation minimization. We worked upon some of the QDDC examples e.g. Traffic Light and then checked them whether they are realizable or not using DCVALID tool and then generated the automaton for the same.

We also explored the slugs tool which takes a specification and compute whether it is realizable or not, if it is then compute the automaton/controller for the specification. We worked on multiple case studies, run the tool on them and analyze the final result.

References

[ER16] Rüdiger Ehlers and Vasumathi Raman. Slugs: Extensible gr(1) synthesis. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 333–339, Cham, 2016. Springer International Publishing.

[Pan00] Paritosh K. Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using dcvalid. Technical report, 2000.

[PW19] Paritosh K. Pandya and Amol Wakankar. Specification and optimal reactive synthesis of run-time enforcement shields. *ArXiv*, abs/1909.08541:91–106, 2019.