Theories, Techniques and Tools for High Integrity Heterogeneous Embedded Systems

By

Amol Wakankar

(Enrollment Number: ENGG01201204009)

Bhabha Atomic Research Centre, Mumbai

A thesis submitted to the

Board of Studies in Engineering Sciences

In partial fulfillment of requirements

for the degree of

DOCTOR OF PHILOSOPHY

of

HOMI BHABHA NATIONAL INSTITUTE

July, 2020

Homi Bhabha National Institute

Recommendations of the Viva Voce Board

As members of the Viva Voce Committee, we certify that we have read the dissertation prepared by Amol Wakankar entitled "*Theories, Techniques and Tools for High Integrity Heterogeneous Embedded Systems*" and recommend that it may be accepted as fulfilling the thesis requirement for the award of Degree of Doctor of Philosophy.

Composition	Name	Attended in person or through video; if in person, signature
Chairman	Prof. A.P. Tiwari	20/7/2020
Convener (Guide)	Prof. A. K. Bhattacharjee	20/7/2020
Co-Guide/External Guide (if any)	Prof. Paritosh Pandya	P. K. Pandya 20/7/2020
External Examiner	Prof.. Deepak D'Souza	20/07/2020
Member	Prof. V. H. Patankar	V.H. Patankar 20/07/2020
Member	Prof. S. Kar	SKar 20/07/2020

Final approval and acceptance of this thesis is contingent upon the candidate's submission of the final copies of the thesis to HBNI.

I hereby certify that I have read this thesis prepared under my direction and recommend that it may be accepted as fulfilling the thesis requirement.

Date: 20/07/2020

Place: MUMBAI

A K Bhattacharjee

Guide

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfilment of requirements for an advanced degree at Homi Bhabha National Institute (HBNI) and is deposited in the Library to be made available to borrowers under rules of the HBNI.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the Competent Authority of HBNI when in his or her judgement the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

Amol Wakankar

DECLARATION

I, hereby declare that the investigation presented in the thesis has been carried out by me. The work is original and has not been submitted earlier as a whole or in part for a degree/diploma at this or any other Institution/University.

Amol Wakankar

List of publications arising from the thesis

Journals / Lecture Notes in Computer Science

1. Amol Wakankar, Paritosh Pandya, Raj Mohan Matteplackel, *DCSynth: Guided Reactive Synthesis with Soft Requirements*, In: Lecture Notes in Computer Science, 2019, Vol 12031, pp: 124-142, Springer, doi: 10.1007/978-3-030-41600-3_9

2. Amol Wakankar, Ashutosh Kabra, A.K. Bhattacharjee, Gopinath Karmakar, *Architectural model driven dependability analysis of computer based safety system in nuclear power plant*, In: Nuclear Engineering and Technology Journal, 2018, Vol 51, Issue 2, pp: 463-478, Elsevier, doi: 10.1016/j.net.2018.10.019

3. Raj Mohan Matteplackel, Paritosh Pandya, Amol Wakankar, *Formalizing Timing Diagram Requirements in Discrete Duration Calculus*, In: Lecture Notes in Computer Science, 2017, Vol 10469, pp: 253-268, Springer, doi:10.1007/978-3-319-66197-1_16

Refereed Conference Proceedings

1. Paritosh Pandya, Amol Wakankar, *Logical Specification and Uniform Synthesis of Robust Controllers*, In: 17^{th} ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE), 2019, pp 1-11, ACM, doi:10.1145/3359986.3361213 **(Best Paper Award)**

2. Paritosh Pandya, Amol Wakankar, *Specification and Optimal Reactive Synthesis of Run-time Enforcement Shields*, In: International Symposium on Games, Automata, Logics, and Formal Verification, (GANDALF 2019), Electronic Proceedings in Theoretical Computer Science, 2019. pp. 91-106, doi:10.4204/EPTCS.305.7

3. Amol Wakankar, Ashutosh Kabra, A.K. Bhattacharjee, Gopinath Karmakar, *A Case Study on Architecture Centric Dependability Analysis for Computer Based Safety System in NPP*, Reliability, In: Safety and Hazard Assessment for Risk-Based Technologies, Lecture Notes in Mechanical Engineering, 2019, Springer, doi:10.1007/978-981-13-9008-1_25

Industry Track Presentation

1. Amol Wakankar, Paritosh Pandya, Raj Mohan Matteplackel, *DCSynth: Guided Reactive Synthesis with Soft Requirements and Performance Measurement*, In: International Symposium on Automated Technology for Verification and Analysis, 2017

Signature of the student: Amol

Date: 30/07/2020

Dedicated to my Family, Teachers and Colleagues

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to my guides Prof. Paritosh K. Pandya and Prof. A. K. Bhattacharjee for their invaluable guidance, suggestions and encouragement during the course of this work.

I am grateful to my doctoral committee members Prof. A.P. Tiwari, Prof. V.H. Patankar and Prof. S. Kar for their invaluable suggestions.

I am also thankful to Dr. G. Karmakar for giving me important lessons on paper writing.

I thank to my friends and colleagues Dr. Ajith K.J., Dr. Raj Mohan Matteplackel, Ashutosh Kabra and Prateek Saxena for their help and support. Finally, I would like to thank my parents, my wife Shruti and my daughter Sidiksha for their love and encouragement.

Amol Wakankar

Contents

List of Figures

List of Tables

Chapter 8

Conclusion and Future Work

High Integrity heterogeneous embedded Systems (HIS) are often used in safety critical application domain, which requires high level of correctness guarantee with respect to some quality parameters or critical requirements. In this thesis, we have proposed and experimented with a framework that provides the required correctness guarantee in HIS with respect to *functional* as well as the *non-functional* requirements. The approach is based on formal methods and takes an integrated view of the development of HIS. We have presented our work on development of theories, tools and techniques in (a) logic based formalization of requirements, (b) correct by construction design/synthesis from logical specifications with a focus on optimizing the quality of the controller, and (b) system dependability analysis from its architectural components.

We propose to use a highly expressive logic QDDC for requirement specification and automatic synthesis of implementation. As discussed in Chapter 3, QDDC is a highly succinct logic and it allows compositional specification of requirements, but it has non-elementary decidability.

As our first contribution, we have introduced an elementarily decidable fragment of logic QDDC, called SeCeNL. In fact, our analysis implies that the existing model checking tool DCVALID for QDDC works with elementary complexity on the proposed subset. Many natural heterogeneous specification patterns such as Timing diagrams, MSC, State machines etc., fit into this fragment. In this thesis, we have used SeCeNL to formalize the timing diagram requirements making them amenable to analysis (in Chapter 3). We have also introduced usage modalities to make clear the requirement posed by such specifications. This work has improved the state of the art by providing a logic and a method for modular and succinct formalization of visual requirement notations.

Secondly, we have introduced the concept of soft requirement guided synthesis to produce high quality controllers (in Chapter 4), which was not much looked at in the literature till now. In this approach the functional requirements are categorized as the hard (mandatory) requirements and the soft (desirable) requirements, both of them are specified as regular properties in QDDC. A technique and a tool (DCSynth) has been developed to synthesize a controller which invariantly satisfies the hard requirements and H-Optimally meets the soft requirements. Intuitively, this gives a correct-by-construction controller for hard requirement which improves the frequency of occurrence of soft requirements. Optimizing the controller to meet the soft requirement is the idea behind producing high quality controller, as the soft requirements in our framework are often used to specify the quality parameters. This is in contrast to the traditional approach, where the synthesis is performed from the LTL based specification (or its efficiently synthesizable subsets like GR1) without any explicit soft requirements to ensure quality of the synthesized controller.

To the best of our knowledge DCSynth is one of the first tools which deals with regular specification in QDDC and allows logical specification of soft requirement to guide the synthesis for producing more desirable (high quality) controllers. Expressively, QDDC is equivalent to regular languages and hence every QDDC specification can be translated into a language equivalent DFA. By resorting to the regular properties specified in QDDC, we could achieve more modular specification as well as better scalability. This is because the regular properties (DFA), allows aggressive minimization and determinization during each step of synthesis. Another factor contributing to the scalability is the use of efficient semi-symbolic DFA data structure, initially introduced by tool MONA, to represent the automaton, controller and supervisor. In our method and tool, we have adapted all the synthesis algorithms such as the fixed point computation and the iterative application of Bellman backup for optimal controller synthesis to work symbolically on the semi symbolic DFA representation. See Section 4.4.

This thesis also addresses the problem of synthesizing robust controllers from *assume-guarantee* specifications (Chapter 5). Assume-guarantee specification is notoriously prone to exploiting the fact that they are vacuously true outside the assumption. While previous work have mostly addressed special cases of robust synthesis, we provide a systematic treatment of this issue by giving a logical specification of the nature of robustness by specifying a formula called robustness criterion. The method is based on weakening the user specified assumption using the robustness criterion scheme. We exploit the expressive power of QDDC to systematically specify a wide class of such robustness criterion in terms of the sequences of time instances when the assumptions are allowed to fail. Controller synthesized under such weakened assumption tolerates (limited) assumption vi-

olations and still continues to meet commitment. We further optimize the controller to improve the frequency of occurrence of commitment irrespective of the assumption by specifying the commitment as a soft requirement. Experimental results demonstrate the improvement in robustness of the resulting controller.

Further exploring the theme of robust designs, the thesis provides a framework for logical specification of wide variety of error-correcting run-time enforcement shields along with a generic method to synthesize shields (in Chapter 6). The run-time enforcement shield not only repairs the output produced by a humanly provided controller to meet desired critical requirements, but it also minimizes the deviation of shield output from that of humanly provided controller. This enables humanly implemented optimizations in the controller to be retained as much as possible. In our specification, the hard deviation constraint formulas specify the worst-case deviation where as soft-requirement based optimal synthesis is used to further minimize the deviation.

It is notable that in the synthesis of robust controllers as well as run-time enforcement shields, we have provided a uniform method of synthesis using the tool DCSynth by leveraging the hard and soft-requirement guided synthesis method. The use of logic QDDC with its interval modalities and counting constructs provides a very powerful vocabulary to formulate robustness criteria as well as shield types (hard deviation constraints).

Since we synthesize controllers and shields from various logically specified robustness and shield type criteria, the quality of the resulting controller must be analyzed. We have proposed the methods for qualitative and quantitative comparison between various controllers. The method is based on measuring the expected value of meeting the commitments in long run (assuming the inputs are uniformly

distributed) by the controller and the concept of must dominance. Theoretical and experimental results profiling the quality of the synthesized controllers and shields are given using some case studies.

Finally, we have also investigated model checking based techniques for analysis of non-functional properties such as dependability of large HIS (in Chapter 7). To overcome the state space explosion problem associated with the model checking technique, we have proposed the architecture-centric compositional technique for dependability analysis. This technique is applied to successfully analyze the dependability of an industrial safety system.

We envisage the following research directions to extend the work presented in this thesis.

- The logic SeCeNL proposed in this thesis can also be used to formalize other widely used visual requirement specification notations such as Message sequence charts and State charts. Extending our requirement formulation and analysis framework to incorporate other such requirement specification notions will allow integrated analysis of heterogenous requirements specified as a mixture of such notations.

- In robust synthesis we have logically specified various robustness criteria as well as shield types. Many of these use parameters (e.g. k, b) provided by the user, to dictate the level of robustness. An interesting extension to this work would be to automatically infer the optimal parameters for the criterion which make the controller realizable.

- The application of run-time enforcement shield in the safety critical domain is well known for monitoring and correcting the system output for some crit-

ical properties. We plan to integrate the synthesized run-time enforcement shields as part of system validation process in safety critical application domain. The main advantage of shield synthesis is that it is a scalable formal verification technique and hence can be used in systems of practical interest.

– Another important open research area is to deal with the scalability issue in automatic synthesis. In parametric synthesis one formulates a generic controller as well as its specification with n symmetric processes, where n is a parameter. Theoretical investigations [78, 63] have shown that if correct by construction synthesis is carried out for a small cutoff instance of n, the solution also holds for all larger values of n. This allows circumventing the scalability problem. The cutoff parameter instance n depends on the architecture and the specification formula. An investigation of parametric synthesis for DCSynth specifications would enhance its applicability.

– In Architecture centric dependability analysis framework, currently we find a feasible solution which meets required dependability goal. In case of multiple feasible solution the analysis has to be guided by the user. The framework can be improved to automatically derive the optimal solution based on some user specified optimality criteria. We also envisage the application of architecture centric analysis to re-configurable or adaptive architectures by exploiting more generic error models. The error models may include component specific degraded modes of operation instead of uniform error model (used in this thesis) for each component. We can also incorporate error propagation to indicate communication errors between the components. This will allow us to model and analyze complex re-configurable systems.

SYNOPSIS

The computer-based embedded systems which demand the highest level of dependability and correctness guarantee are called High Integrity heterogeneous embedded Systems (HIS). Typical examples of such systems are the systems required in safety critical application domains such as nuclear power plant, aerospace etc.

The high level of dependability as well as the correctness guarantees required by HIS can only be achieved by incorporating the rigorous analysis of specified *functional* as well as the *non-functional* properties in the development process, which takes an integrated view of the heterogeneous elements during the development of the HIS. The major focus of this thesis is in the development of theories, techniques and the tools which help in analysis and automatic synthesis of complex HIS from high level requirement specifications as well as the architectural components. Therefore, this thesis can be broadly divided in to two parts.

The first part investigates formal analysis and automatic synthesis of discrete controllers from functional requirements. This includes the formalization of functional requirements given as *timing diagrams* in an expressive logic QDDC. We identify an elementarily decidable fragment of QDDC called SeCeNL, which retains most of the advantages of QDDC, and is expressive enough for modeling heterogeneous requirements. We formalize timing diagram properties by a *compositional* translation to SeCeNL, which makes these requirements amenable to analysis like checking for consistency, realizability etc.

One of the major contributions of this thesis is in the automatic synthesis of *high quality* controllers for logic based requirements in QDDC. We propose a technique called *soft requirement* guided synthesis, where soft (desirable) requirements are mainly used to specify *quality* attributes. We synthesize a controller

which invariantly meets the *mandatory* requirements and H-Optimally meets the soft requirements. The technique is efficiently implemented in a tool DCSynth using the MTBDD data structure. We also carry-out the *performance measurement* of the synthesized controllers to assess their quality.

We give a logical specification of various notions of *Robust controllers* and *Run-time enforcement shields* along with a uniform synthesis method for each of these notions. A robust controller continues to function correctly (i.e maintain its requirements) even under the intermittent failure of environmental/plant assumptions. On the other hand, a run-time enforcement shield observes the input and outputs of the system under consideration to check weather a given *critical property* is satisfied. In case of property violation, the shield rectifies the output of system such that the critical property is satisfied. Moreover, it also ensures that the shield output *minimally* deviates from the system output. In this thesis, we develop a theory for logical specification of various notions of *robustness* as well as *shields*, which subsumes few already existing notions and also defines some new notions. Finally, building upon our soft requirement guided synthesis framework, we give a uniform synthesis method to synthesize various types of robust controllers and run-time enforcement shields. The quality is enforced using the soft requirements.

The second part of this thesis deals with probabilistic analysis of *non-functional properties* (e.g. system dependability), where we propose a compositional method for analysis of large HIS. The compositional analysis allows us to deal with systems of practical interest. We show the usefulness of this approach by applying it to some industrial case studies.

It is perceived that the techniques developed in this thesis would allow realiz-

ing high quality HIS from high level specifications through algorithmic synthesis and analysis following the correctness-by-construction approach. Non-functional properties such as dependability of the complex system architectures can also be analyzed with the proposed compositional analysis technique.

Chapter 1

Introduction

The computer-based systems performing safety critical functions in nuclear, transportation and medical domains, demand the highest level of dependability and correctness guarantee. These systems need a high degree of rigour in their design to assure their integrity during operation. Many of these systems involve an orchestration of service layers among intelligent sensors, control elements and digital computation devices to provide a level of optimal control of the plant. The control laws implemented in these computation devices themselves are based on complex algorithms. These systems belong to the category of High Integrity heterogeneous embedded Systems (HIS). Heterogeneity comes from the fact that these systems are composed of components with different characteristics, in terms of the applicable models capturing their functional requirements or in terms of their execution/interaction semantics.

During the development of a complex HIS, several heterogeneous elements at each design phase come into play. Designers prominently use various visual requirement specification notations to capture and specify *functional requirements*

addressing different aspects of the system. For example, *temporal relationship* among different signals are specified using Timing diagrams [82] or Message sequence charts, and State Charts [54] efficiently specify the *event-based dynamic* behavioral aspects of the system. Heterogeneity also comes at the architecture design level of HIS, as the system may consist of several components having diverse execution/interaction semantics. The modeling of non-functional properties such as reliability, availability, etc., which are mainly affected by the system architecture, requires domain specific modelling techniques and analysis. It is imperative to analyze them early during their architectural design to show conformity to dependability attributes, as any change during the detailed design would entail significant cost.

The high level of dependability required by HIS can be realized by rigorous analysis of *functional* requirements, correct by construction synthesis of implementation from functional requirements and an assessment of *non-functional* properties. The focus of this research work is on the development of theories, techniques and tools which will help in automatic synthesis of robust components of HIS from specification and show compliance to dependability requirements of HIS from high level architectural design.

Traditional approaches to show the correctness rely mainly upon simulation and testing which cannot give any firm guarantee on the correctness with respect to the specified requirements. In contrast to this, in the formal methods based design approach, the requirements are first expressed in a suitable logic. This allows rigorous formal analysis of requirements such as checking for consistency, realizability, etc. Formal verification of a system against the specified requirements is also well established in the literature. Furthermore, *automatic*

synthesis of a system from logic based requirements is possible and currently an active research area. Unfortunately, besides these advantages, many existing formal methods also have few disadvantages; (a) often the logical notations used are cumbersome, complex and far removed from practically used visual notations for requirement specification, (b) the automatic synthesis methods and tools from logical specifications do not really consider the quality of the synthesized controller. In addition to analysis and synthesis from functional requirements, the analysis of non-functional requirements such as reliability, availability has also been explored for assessing the system dependability. But component level architecture centric analysis of non-functional requirements for HIS has not been much looked at, which is essential to deal with systems of practical interest.

Considering these challenges in application of formal methods, Henzinger and Sifakis [56] in their survey on "The Embedded System Design Challenge" have emphasized the need for a coherent scientific foundation for design of HIS. This requires encompassing manifestations of *heterogeneous* elements in the system as well as *constructivity* during the system design. Constructivity allows building complex systems that meet the given requirement from the components with known properties. In line with this objective, in this thesis we propose to bridge the gap between visual/semi-formal notations for requirement specification (intuitive but often subject to interpretation) used by designers and the formal logic based (unambiguous) approach. This is achieved by systematically translating the heterogeneous visual requirement specification notations into an expressive logic and investigating the techniques for analysis of these requirements. We investigate the automatic synthesis from such logic based requirements. We also show how to compute dependability attributes of low level components from a high

level architecture, for the required Probability of Failure on Demand (PFD) using a component based modeling paradigm.

This thesis mainly consists of two parts. The first part investigates the logic based formalization, analysis as well as automatic controller synthesis from functional requirements as shown the Figure 1.1. We present the formalization of temporal requirements specified using timing diagrams using an expressive logic in order to make them amenable to the formal analysis. The major focus of this thesis is automatic synthesis of *high quality* controller from the logic based specification. This work is further extended to the synthesis of robust controllers and run-time enforcement shields. The second part of this thesis deals with probabilistic analysis of non-functional (dependability) requirements, where we propose the compositional method for analysis of large HIS as shown in Figure 1.2. These two analyses are complimentary and together they provide an integrated approach to ensure the required correctness guarantees for HIS.

The major cornerstone of our method is the logic used to formalize the requirements and associated techniques for analysis and synthesis. The logic used for requirement formalization, analysis and synthesis is required to have following important properties.

- It should be expressive enough to be able to handle the diverse notations used in specifying HIS.
- It should be amenable to automated analysis,
- It must support efficient synthesis of controllers.

To achieve these objectives, we propose a framework based on a rich *interval temporal logic* Quantified Discrete Duration Calculus(QDDC) [83]. It is a highly

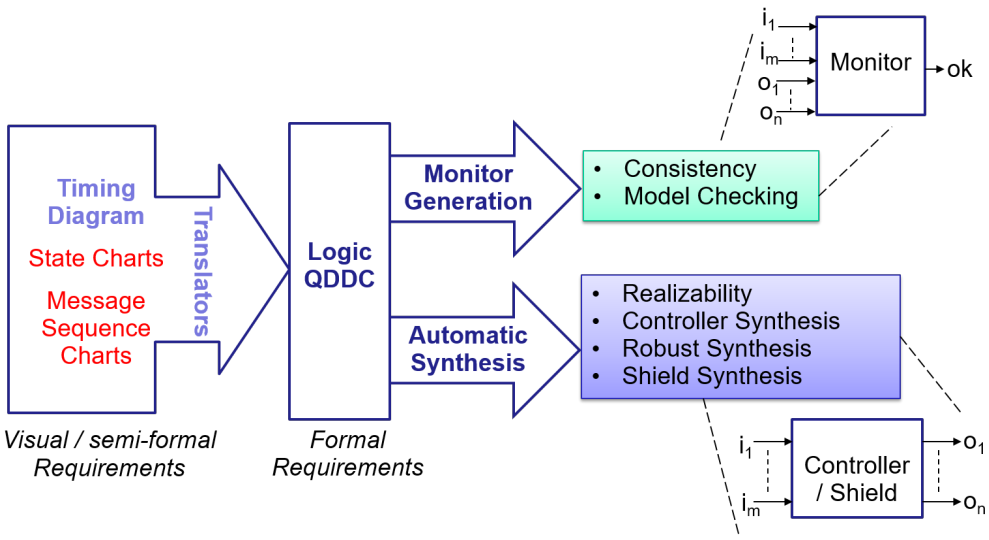


Figure 1.1: Formalization and automatic synthesis from heterogeneous requirements.

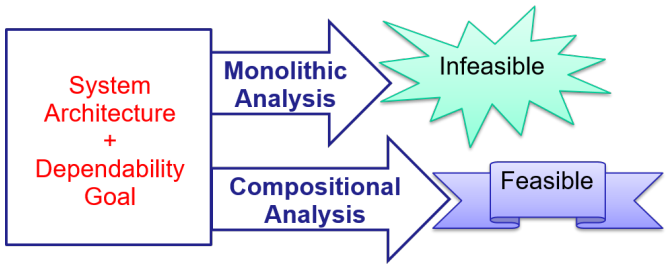


Figure 1.2: Architecture centric compositional dependability analysis.

succinct logic for specifying patterns of behaviours. Formally it has the expressive power equivalent to regular languages. Therefore, the properties specified using QDDC are also termed as the *regular properties*. Its bounded counting features, interval-based modalities, second order quantification and regular expression like primitives allow succinct and modular specification of complex qualitative and quantitative properties frequently occurring in HIS. Prior work [83] shows that any QDDC formula D can be effectively translated into a language equivalent Deterministic Finite State Automata (DFA) $\mathscr{A}(D)$ over finite words. This enables the use of automata theoretic approach to analyze the requirements formalized in this logic. Section 2.1 covers the syntax and semantics of this logic.

Major Contributions:

We propose to use the logic QDDC for formalizing the heterogeneous requirement notations by translating them into this logic. This makes the requirements amenable to the analysis. Logic QDDC has been used previously to formalize the requirements given in State Charts and Live Sequence Charts [33]. In this thesis, we propose an efficient sub-set of logic QDDC to formalize the timing diagram requirements. In general, QDDC has non-elementary worst case complexity of satisfiability checking as well as formula automaton construction. However, confining to the proposed efficient sub-set of logic QDDC, which is expressive enough to formalize the timing diagrams, makes the satisfiability checking elementary.

As our first contribution, we identify an elementarily decidable fragment of QDDC called SeCeNL. We claim that this fragment retains most of the advantages of QDDC, and it is sufficient for modeling heterogeneous requirements. We formalize *timing diagram* properties by a systematic translation to SeCeNL. The

advantage is that, the well-developed automata-theoretic techniques can be used for requirement analysis as well as model checking and controller synthesis from timing diagram specifications. Chapter 3 gives the logic based formalization of timing diagram requirements.

As our second major contribution in this thesis we propose a technique for automatic synthesis of *high quality* controllers from their QDDC logic based specification. Although the automatic synthesis of discrete controllers from logic-based requirements (specified in other widely used specification logic such as LTL, PSL etc.) is well studied in literature, the *quality* of the synthesized controller is still a major concern. This is mainly because of lack of features to specify quality attributes in these logics [79].

We address this problem by introducing the concept of *soft requirement* guided synthesis. These *soft* (desirable) requirements are mainly used to specify the desirable quality parameters. We introduce a technique which allows synthesis of discrete controllers from *regular properties* (QDDC formulas) given as a tuple (I, O, D^h, D^s), where D^h and D^s are QDDC formulas over a set of input and output propositions (I, O). Here, D^h and D^s are the **hard** and the **soft** requirement, respectively. *We synthesize a controller which, (a) invariantly satisfies D^h and (b) it meets the D^s at "as many points as possible".* Meeting D^s "at as many points as possible" is achieved by synthesizing a controller which maximizes (optimizes) the cumulative count of D^s holding in next H steps, averaged over all the inputs of length H. Such a controller is called *H-Optimal* for D^s.

Guided synthesis allows us to obtain more desirable controller out of several candidate controllers possible for a given requirement. It also allows us to deal with under-specified or conflicting requirement specification. Chapter 4 gives the

details of soft requirement guided synthesis approach.

This thesis presents the logical specification and application of the guided synthesis to automatic synthesis of *robust controllers*. Typically, the controller specification consists of a pair of QDDC formulas (D_A, D_C). Formula D_A gives the *assumption* on environment/plant behaviour and the formula D_C gives the *commitment* i.e. the requirements to be satisfied by the synthesized controller. A standard correctness criterion for synthesis is termed as *Be-Correct* [12], which mandates that in all the behaviours of the synthesized controller, at any point the commitment D_C should hold provided the assumption D_A has held invariantly in past (denoted by $pref(D_A)$). This specification can be denoted as $G(pref(D_A) \Rightarrow D_C)$. On the other hand, *robust synthesis* deals with the synthesis of a controller which meets the commitment even under intermittent assumption violations. Thus, *robustness* pertains to the ability of D_C holding even when D_A is violated intermittently in the past [12], which can be specified as a weaker formula than $pref(D_A)$. We propose a framework for logically specifying the assumption weakening by a formula called *robustness criterion*. We formulate a method for automatically relaxing any user specified assumption under such robustness criterion. This is termed as *hard robustness*. The logical formulation of robustness criterion in QDDC allows us to specify various existing as well as new hard robustness notions and devise a uniform synthesis method to obtain robust controllers under these notions. We also optimize the controller to satisfy the commitment H-Optimally irrespective of the assumption. This is called *soft robustness*. We apply our guided controller synthesis to automatically obtain a robust controller from such logical specification as detailed in Chapter 5. To show the effect of our soft requirement *guided synthesis* as well as *robust synthesis* approach, we use Markov-model

based performance measurement methods to compare the quality of the synthesized controllers. This is done by measuring the *expected value* of meeting the formula D_C on long runs by the synthesized controller.

As another major application of our guided synthesis method and the logic QDDC, we also propose a framework for logical specification and synthesis of various notions of *run-time enforcement shield*. A *run-time enforcement shield* for a specified critical requirement $REQ(I,O)$ is a controller which receives both, the inputs and the outputs (I,O) generated by *System/Controller with Sporadic Error*(SSE) under consideration. The shield produces a modified output O' which is guaranteed to invariantly meet the critical requirement $REQ(I,O')$. Moreover, in each run, the shield must deviate from the SSE output "as little as possible". This allows the shield to benefit from system designer's optimizations incorporated in SSE without having to formally handle these in the synthesis. The specification and synthesis of run-time enforcement shield from various existing as well as new notions of the shield is covered in Chapter 6. We measure the quality of the synthesized shields under these various notions in terms the expected value of non-deviation (between system output O and shield output O') in long run and in terms of worst-case burst-deviation latency.

In the second part of this thesis, we deal with modeling and quantitative analysis of non-functional properties such as dependability of HIS. We propose an architecture centric approach, where the model of a system architecture consisting of its sub-components and interaction between them is represented in Architectural Analysis & Design Language (AADL) [44] using OSATE tool [43]. Each component is annexed with a probabilistic automaton which incorporates various operational and fault states of the constituent component(s) along with probabilis-

tic transitions among these states. Such a probabilistic automaton is called the *fault model* of the component. Similarly, a fault model can also be associated with a system. The fault model of a system can be obtained by taking product of fault models of its sub-components. A scheme to translate the fault model of hierarchical model in to a Discrete Time Markov Chain (DTMC) model is developed. This DTMC model can be analyzed for various dependability properties. In this work, the probabilistic model checker PRISM [73] is used as a back-end analysis engine for dependability analysis.

A major challenge in the application of the above model checking technique is the infamous *state space explosion* problem [29], which makes it impractical to analyze the systems of industrial interest. We overcome this problem by adapting the existing techniques to perform the dependability analysis of large HIS with *hierarchical* architecture. This is done by making the analysis compositional. Furthermore, industrial case studies have been carried out to validate the proposed technique. Chapter 7 of this thesis covers the architecture centric compositional dependability analysis approach.

The major contributions of this thesis are summarized below:

1. Formalization and automatic translation of timing diagram requirements in the elementarily decidable fragment of QDDC called SeCeNL. Formal analysis and synthesis of these requirements using associated automata theoretic techniques.

2. Soft-requirement guided synthesis of discrete controllers from the requirements given in QDDC. A tool *DCSynth* to automate the proposed techniques is developed. All synthesis algorithms are adapted to work symbolically on MTBDD based efficient representation of automata and controllers.

3. A method for logical specification of robust controller as well as run-time enforcement shield is given. An application of guided synthesis method to synthesize robust controllers and run-time enforcement shields.

4. A compositional method for architecture centric dependability analysis of large HIS. Compositional analysis allows us to scale the existing analysis methods to apply it for industrial systems of practical interest.

1.1 Relevant Literature

Specification Logics: In formal methods, mathematical logics play an important role for specification and formal analysis of any system. On the other hand, Finite state automata are used to effectively model the system implementation. Several logics have been studied in the literature for unambiguous specification of functional requirements. Temporal logic such as LTL [88] and CTL [30] are amongst the most prominently used logic for requirement specification of reactive systems. Such specification includes the specification of environmental constraints and the requirements to be met by the controller in a reactive system.

LTL and its variants such as PSL [40] which enhances LTL with regular expression like features and MTL [7] which allows specification of real time properties, have been successfully used for specification in some industrial examples [1, 48]. The most desirable properties of LTL which enables its widespread application in requirement specification includes its clean syntax and its intuitive semantics. Over the years some of the major shortcomings of LTL have been identified in the literature. Few of them are the lack of quantitative features which restricts its use in specification of complex quantitative properties, as well as its

inability to effectively express the robustness in the specification [79]. The use of classical logic such as MSO [102] and Duration Calculus [24] to specify desirable requirements has also been studied. QDDC [83] is a discrete time version of duration calculus(DC) originally proposed by Zhou *et al.*, for modelling real-time requirements. The use of DC in real time system design was explored by Zhou *et al.* in [24, 23].

The Expressiveness of these logics are also well studied in literature, e.g. LTL is expressively complete with respect to First Order Logic [10], and it is equivalent to *star free* ω-regular languages [102]. It is possible to get a language equivalent Büchi automata for any LTL formula (but not Vice-Versa as the ω-regular languages are strictly more expressive than LTL). The logic MSO (over finite words) and QDDC are expressively equivalent to regular languages and there exists a language equivalent DFA for each formula specified in these logics. Showing the effective construction of language equivalent automata for these logics are the some of the most ground breaking results in computer science [21, 53], as they enable the automata theoretic algorithmic analysis of these specifications.

Formal Analysis and Verification: The Satisfiability checking of a logical formula and Model checking (also know as formal verification) are some of the most important problems for any mathematical logic. Satisfiability checking of a logical formula, checks for the existence of the model for a given formula. Model checking involves algorithmically answering, whether the set of behaviours exhibited by the given system model (often given as a Finite state automaton), are included in the set of behaviours which satisfy the logical formula. The model checking problem for LTL [103] and MSO [21] against the system specification

given as Finite state automaton is well studied in literature and several efficient tools exist [67, 28, 34]. These tools have significantly contributed to show the importance of formal methods in providing the correctness guarantees required by high integrity systems.

Requirement Notations: The use of mathematical logic for the specification of requirements provides several advantages in terms of unambiguity and formal guarantees it provides, but formalizing requirements in logic is often complex and cumbersome. Therefore, industries came up with several intuitive and visual notations for the requirement specification, capturing different behavioural aspects of the system. The most notable among these are Timing Diagrams, Sequence Diagrams, State Diagrams etc., which are standardised by UML [82]. Although, these notations provide a useful mechanism to effectively specify the requirements and are used widely in the industry, but the analysis and formal verification from these notations is not always possible, because these notions are not completely formal. Therefore, several researchers have tried to formalize these visual notations into logical framework to make them amenable to formal analysis.

For example, there have been several attempts at formalizing the requirements given in timing diagram in the framework of temporal logics such as the *graphical interval logic* [35], *timing diagram logic* [46], with *LTL formulas* [26], and as *synchronous regular timing diagrams* [8]. Moreover, there are industry standard property specification languages such as *PSL-Sugar* and *OVA* for associating temporal assertions to hardware designs [40].

However, the succinctness and compositionality remained the major issue in effectively utilizing the formal analysis techniques on timing diagrams [26]. Also

the specification of the timing/synchronization constraints across the timing diagrams is not very natural in these logics [26]. Moreover, these notation often specify behaviour sequences but it is unclear whether such sequences must/may occur in the behaviour. Rectifying these, Harel [32] has proposed Live Sequence Charts *LSC* with liveness modalities.

Correct by Construction Program Synthesis: Program synthesis deals with the problem of algorithmically obtaining an implementation for a given logical specification, such that all the behaviours of the program meet the specification. The problem was first defined by Church [27] over specification given in MSO and the aim was to synthesize the implementation (controller) as a Mealy machine which realize the given specification. One of the most useful problem in HIS is synthesis from the specifications for reactive programs. Reactive programs are the programs that continuously interact with the environment. Büchi *et al.* [20] and Rabin [91] independently presented solutions to Church's problem.

Specifying the behavior of reactive systems in MSO is cumbersome and hence several alternatives were proposed in the literature. One of the most widely accepted alternatives is LTL, which is used in the formal verification and synthesis community [89]. However, the synthesis from LTL properties was proved to be doubly exponential in the size of the formula by Rosner [97]. To make the synthesis more efficient people have looked at the useful subsets of LTL. Piterman *et al.* [87, 68] have proposed an efficient polynomial time symbolic algorithm to automatically synthesize controllers for the subset of LTL called GR(1). Similarly, Wolff *et al.* [106] have identified a useful subset of LTL for efficient controller synthesis for non-deterministic transition systems and Markov decision processes.

Reactive synthesis from Linear Temporal Logic (LTL) specification has been widely studied and considerable theory and tools exists [12, 5]. While several tools support safety synthesis over circuits (see [62], safety track), the leading tools such as Acacia+[16] and BoSy[42] mainly focus on the future fragment of LTL.

Apart from reactive synthesis, the related problem of supervisory control of a Discrete Event System (DES) was introduced by Ramadge and Wonham [92, 93]. Typically the plant (i.e. the system to be controlled) is modelled using Finite state automaton. The aim is to synthesize a controller for a given specification, which restricts the behaviour of the plant such that the resulting plant behaviour always meets the specification. They proved that the controller can be synthesized in linear time for such specifications. The connection between the reactive synthesis and supervisory control was formally studied by Ehlers *et al.* [38].

Quality aspects in Synthesis:

– **Quantitative Synthesis:** Specifying the quality of a behaviour has been addressed by assigning a quantitative measure to each behaviour, resulting in Quantitative Languages [13]. This allows generalization of correct-by-construction approach to optimal synthesis. Most of the work revolves around the optimal quantitative synthesis, where a weighted arena is assumed to be available, and algorithms for optimal controller synthesis for diverse objectives such as Mean-payoff [13, 17] or energy [18] have been investigated. Also, the techniques for optimal controller synthesis are discussed by Ding *et al.* [36], Wongpiromsarn *et al.* [107] and Raman *et al.* [94], where they have explored the use of receding horizon model predic-

tive control along with temporal logic properties. While considerable theory exists in this area of quantitative synthesis, its practical impact has been limited as it is unclear how to specify quantitative measures using logical formulas in a meaningful way.

– **Robust Synthesis:** Robust controller synthesis deals with the problem to synthesize a controller, which continues to function (i.e maintain its commitment) under the failures of environmental/plant assumptions as much as possible. When such failures are transient, the controller should be able to recover from the failure by re-establishing the commitment in bounded time [76, 14]. Several authors have investigated the notions of robustness [76, 39]. Bloem *et al.* [14] provide a classification of different robustness notions. They have addressed the problem of robust synthesis by proposing the notion of recovery to normal operation if safety assumptions are violated and by optimizing the controller to meet the guarantee formula when liveness assumptions are violated [12]. Ehlers *et al.* [39] propose synthesis of resilient controllers. These provide an ability to tolerate k errors between two periods of recovery of length at-least b. D'Souza *et al.* address the issue in context of conflict-tolerant features [37]. For each of these robustness criteria, a specific synthesis algorithm is developed.

– **Synthesis of run-time enforcement shield:** A *run-time enforcement shield* observes the inputs and outputs of a system under consideration (developed manually) and checks for the correctness with respect to a given critical property in an appropriate logic. The shield rectifies the system generated output when it does not satisfy the given critical property. Finite state au-

tomata as well as LTL are primarily used for specification of critical properties in existing work. The idea of error-correcting run-time enforcement shield was proposed in the pioneering work of Bloem *et al.* [15], where the notion of k-stabilizing shield (with a synthesis algorithm) was proposed. This was further enhanced by Konighofer *et al.* [69]. Wu *et al.* [110, 109] 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.

Analysis of non-functional properties: For the analysis of non-functional properties like dependability, there are many well studied quantitative analysis techniques available in literature, such as reliability block diagrams, fault tree analysis, Markov analysis etc. Rao *et al.* have proposed the use of dynamic fault tree [95] to effectively capture and analyze the behaviour of the components with functional-dependent failures, spares and dynamic redundancy management. Rouvroye *et al.* [98] showed that Markov analysis covers most of the aspects of quantitative safety evaluation (except uncertainty analysis) while taking into account the effect of redundancy, common cause failures, self-diagnostics and on-line/off-line test & repair. Several model checking tools for the analysis of Markov model exist [73] and have been effectively used in the past for analysis. But, in most of these work, one of the major issues in analysis of large HIS is the problem of large state space, which restricts the application of traditional analysis method only to small systems.

1.2 Thesis Organization

This thesis is organized in eight chapters as follows. Chapter two provides the preliminaries including the logic QDDC and associated tools. The elementarily decidable sub-logic SeCeNL of QDDC has been discussed in chapter three. The chapter then provides the formalization of timing diagram requirements in the logic SeCeNL. In chapter four, we present the soft requirement guided synthesis of discrete controllers from hard and soft requirements given in the logic QDDC. In the next two chapters, the thesis explores logical specification of robust controllers as well as run-time enforcement shields. The applications of guided synthesis framework to undertake the automatic synthesis of robust controller and run-time enforcement shields is presented in these chapters. In chapter seven, we apply the probabilistic model checking technique for architecture centric analysis of dependability of large industrial system using a compositional analysis methodology. Each of these chapters concludes with a discussion section where a comparison with related work is provided. The final chapter presents the conclusion and possible future work.

Chapter 2

Preliminaries

2.1 Quantified Discrete Duration Calculus (QDDC) Logic

Let PV be a finite non-empty set of propositional variables. A behaviour σ is 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 $len(\sigma) = n+1$, $dom(\sigma) = \{0,\ldots,n\}$ and $\sigma,[i,j] = P_i \cdots P_j$ and $\sigma[i] = P_i$.

In our logic there are two types of entities; proposition denoted by ϕ and QDDC formula denoted by D. A proposition ϕ is evaluated at a position in the word, whereas a QDDC formula D is evaluated in the interval [b,e]. Let ϕ denote a *propositional formula* over variables PV. 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 operators $\&\&, ||, !$ denoting conjunction, disjunction 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 inductively as follows:

$$\begin{aligned}
&\sigma, i \models true, \\
&\sigma, i \models p \qquad \text{iff} \quad p \in \sigma(i), \\
&\sigma, i \models !p \qquad \text{iff} \quad \sigma, i \not\models p,
\end{aligned}$$

and the satisfaction relation for rest of the Boolean combinations is defined in a natural way.

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

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

where $\varphi \in \Omega(PV)$, $p \in PV$, $c \in \mathbb{N}$ and $\bowtie \ \in \{<, <=, =, >=, >\}$.

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} and let $[b,e] \in Intv(\sigma)$ be an interval. Then the satisfaction relation of a QDDC formula D over PV and interval $[b,e]$ written as $\sigma, [b,e] \models D$, is defined inductively as follows:

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

with Boolean combinations $!D$, $D_1 \mid\mid D_2$ and $D_1 \;\&\&\; D_2$ defined in the expected way.

We call word σ' a p-variant, $p \in PV$, of a word σ if $\forall i \in dom(\sigma), \forall q \neq p : q \in \sigma'[i] \Leftrightarrow q \in \sigma[i]$. Then $\sigma, [b,e] \models ex\ p.\ D \Leftrightarrow \sigma', [b,e] \models D$ for some p-variant σ' of σ and $(all\ p.\ D) \Leftrightarrow (!ex\ p.\ !D)$.

Entities *slen* and *scount* are called *terms*. The term *slen* gives the length of the interval in which it is measured, *scount* φ where $\varphi \in \Omega(PV)$, counts the number of positions including 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 $scount(\sigma, \varphi, [b,e]) = \sum_{i=b}^{i=e} f(i)$, where $f(i) = \begin{cases} 1, & \text{if } \sigma, i \models \varphi, \\ 0, & \text{otherwise.} \end{cases}$

We also define the following derived constructs: $\{\{\phi\}\} = \langle\phi\rangle\hat{}(slen = 1)$, $pt = \langle true\rangle$, $ext = !pt$, $\langle\rangle \mathbf{D} = true\hat{}D\hat{}true$, $[]D = (!\langle\rangle!D)$ and $\mathbf{pref}(\mathbf{D}) = !((!D)\hat{}true)$. 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 $b \leq e' \leq e$.

So far we have evaluated a formula D over an interval [b,e]. Now we define the path satisfaction of a formula, where a formula D holds at a point i in a behaviour provided the **past** of the point i satisfies D. (This definition only applies to QDDC formulas and not to proposition ϕ.)

Definition 1 (Past satisfaction and language of a QDDC formula D)**.** *Let $\sigma, i \models D$ iff $\sigma, [0,i] \models D$, and $\sigma \models D$ iff $\sigma, len(\sigma) - 1 \models D$. We define $L(D) = \{\sigma \mid \sigma \models D\}$, the set of behaviours accepted by D. Formula D is called valid, denoted $\models D$, iff $L(D) = (2^{PV})^{+}$.* □

Example 2. *Consider the behaviour σ over propositional letter A below.*

Pos	0	1	2	3	4	5	6	7	8	9
$\sigma(A)$	1	0	0	1	0	1	0	0	0	1

QDDC formula $D(k)$=`[]([[!A]] => slen < k)` *holds for an interval $[b,e]$ provided for all its sub-intervals $[b',e']$ with $b \leq b' \leq e' \leq e$ if* `!A` *is true throughout the interval then the length of the sub-interval $[b',e']$, i.e. $e'-b'$, must be less than k (i.e. the interval spans at most k cycles). Thus, formula $D(2)$ holds in above behaviour for intervals $[0,5]$ since all sub-intervals with A continuously false span at most 2 cycles. Interval $[5,8]$ does not satisfy $D(2)$ since it has a sub-interval $[6,8]$ of length 2 (i.e. 3 cycles) where A is invariantly false.*

A QDDC formula D holds at a position i if the interval $[0,i]$ spanning its past satisfies D (see Definition 1). Thus, $D(2)$ holds at position 5. It also holds at position 7. But it does not hold at positions 8 or 9. Formally, $\sigma,5 \models D(2)$ but $\sigma,9 \not\models D(2)$.

Theorem 3. *[83] For every formula D over propositional 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.* □

We define the *size of a formula* as the summation of number of literal and operators present in the formula. Note that QDDC formula can include counting constraints. Throughout this thesis, it is assumed that constants occurring in formulas are represented in binary form so that a natural number constant c contributes $log(c)$ to the size of the formula. The minimal size of $\mathscr{A}(D)$ is non-elementary in the size of D.

A tool DCVALID implements this formula automaton construction in an efficient manner by internally using the tool MONA [66]. It gives *minimal, deterministic* automaton (DFA) for the formula D. Detailed description of QDDC and its model checking tool DCVALID can be found in [83, 84].

2.2 Tool DCVALID

The reduction from a QDDC formula to its formula automaton has been implemented into the tool *DCVALID* [83, 84]. The formula automaton it generates is total, deterministic and minimal automaton for the formula. DCVALID can also translate the formula automaton into Lustre/SCADE, Esterel, SMV and Verilog *observer modules*. By connecting this observer module to run synchronously with a system we can reduce model checking of QDDC property to reachability checking in the observer augmented system. See [83, 84] for details.

Example 4. *We now give an example of a QDDC specification of a shared resource arbiter with 2 request lines denoted by req_1 and req_2 and 2 acknowledgement lines denoted by ack_1 and ack_2. The job of an arbiter is to grant an access to the shared resource to exactly one of the requesting clients. The properties of this arbiter are specified in QDDC as given in Figure 2.1.*

The formula "Exclusion" states that only one of the client can be granted access to the shared resource. The formula "Noloss" states that if any request is high, then one of the request should be granted the access by raising the corresponding acknowledgment. The formula "NoSpuriousAck" states that access should be granted only if there is a request and the formula "Response" says that if any request is continuously high for atleast 2 cycles then it should be granted

access atleast once by making the corresponding acknowledgement line high. Finally, the specification is conjunction of all these properties.

Figure 2.2(a) gives an explicit representation of DFA for the automaton of specification given in Figure 2.1. Its alphabet Σ is 4-bit vectors giving value of propositions $(req_1, req_2, ack_1, ack_2)$ and set of states $S = \{1,2,3,4\}$. Being a safety automaton it has a unique reject state 4 and all the missing transitions are directed to it. (State 4 and transitions to it are omitted in Figure 2.2(a) for brevity.)

2.3 Semi-Symbolic DFA Representation

An interesting representation for total and deterministic finite state automata was introduced and implemented by Klarlund *et al.* in the tool MONA[66]. It was used to efficiently compute formula automaton for MSO over finite words. We denote this representation as *Semi-Symbolic DFA* (SSDFA). In this representation, the transition function is encoded as *multi-terminal BDD* (MTBDD).

MTBDD is a generalization of BDD in a sense that BDD is a directed acyclic graph used to encode some Boolean function $f : \mathbb{B}^l \to \mathbb{B}$ for given 'l' possible Boolean variables. MTBDD is a directed acyclic graph used to encode multi-valued function $f : \mathbb{B}^l \to \mathscr{D}$, where $\mathscr{D}$ is the range of this multi-valued function.

We now briefly describe the SSDFA representation of an automaton. Figure 2.2(b) gives the SSDFA for the explicit representation of automaton given in 2.2(a). In SSDFA the states are explicitly represented by terminals in MTBDD, which are assigned an integer number. Note that states are explicitly listed in the

```
var  req1, req2, ack1, ack2;
--Property 1: Mutual Exclusion
  define Exclusion as
    [[( ack1 => !ack2 )   &&  ( ack2 => !ack1 )]];
--Property 2: No lost cycle
  define Noloss as
    [[ (req1 || req2 ) => (ack1 || ack2 )  ]];
--Property 3: No spurious acknowledgement
  define NoSpuriousAck[req, ack] as
    [[ ack => req ]];
--Property 4: Bounded Response
  define Response[req, ack] as
    [](([[req]] && slen=1 => <> <ack>);
-- Specification is conjunction of all properties
infer
  Exclusion && Response[req1,ack1] && Response[req2,ack2] &&
  Noloss &&  NoSpuriousAck[req1, ack1] && NoSpuriousAck[req2, ack2]
```

Figure 2.1: The properties of 2 Client Arbiter in QDDC

array at top and final states are marked as 1 and non-final states marked as -1. (For technical reasons there is an additional state 0 which may be ignored here and state 1 may be treated as the initial state). Figure 2.2(b) represents the SS-DFA that encodes 5 states (State no. 0 to 4) and the transition relation between them. Each state s points to shared MTBDD node encoding the transition function $\delta(s) : \Sigma \rightarrow S$ with each path encoding one transition.Thus, each path encodes an element of Σ and ends in the next state. Each circular node of MTBDD represents a

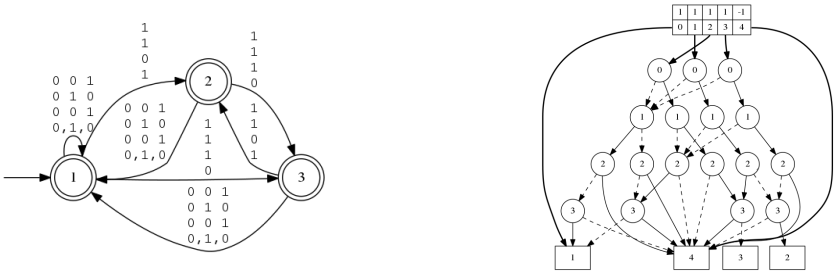


Figure 2.2: Example automaton (a): Explicit representation of Automaton (b): SSDFA format

decision node with indices $0,1,2,3$ denoting variables $req_1, req_2, ack_1, ack_2$. Solid edges lead to true cofactors and dotted edges to false cofactors.

MONA provides a DFA library implementing automata operations including product, complementation, projection, determinization and minimization on SSDFA. Moreover, automata may also be constructed from scratch by giving the list of states and adding transitions one at a time. The original papers [66, 67, 55] may be referred for further details of SSDFA and the MONA DFA library.

Using this, the tool DCVALID computes a minimized, language-equivalent SSDFA, denoted by $\mathscr{A}(D)$, for a QDDC formula D[83, 84]. The formula D is first translated to MSO over finite words of tool MONA; with some optimizations [70]. The tool MONA is invoked on the resulting formula to obtain SSDFA for the formula automaton. MONA constructs the automaton in bottom up fashion by first constructing automata for each sub-formula and then composing these to obtain the automaton for the composite formula. Eager minimization is used at each stage while converting the formula into SSDFA. Several optimizations are used in MONA to efficiently compute this automaton [66].

Although the worst case complexity of automata construction in MONA as well as in DCVALID is non-elementary in the size of formula. But, as pointed out

in [55] the efficient implementation with several optimizations such as formula reductions, DAGification, three-valued logic, eager minimization, SSDFA based automata representations, and cache-conscious data structures, in practice allows to handle several complicated cases using these tools without encountering non-elementary blow up in the automaton. On the other hand worst case complexity also indicates that formula in MONA or DCVALID may be non-elementarily more succinct than a regular expression or an explicit transition table [55]. It may be also noted that SSDFA gives very succinct representation of an automaton, as it allows sharing of BDD nodes across the transition relation of individual states.

Chapter 3

Formalizing Timing Diagram Requirements in QDDC

A *timing diagram* is a collection of binary signals and a set of timing constraints on them. It is a widely used visual formalism in the realm of digital hardware design, communication protocol specification and embedded controller specification. The advantages of timing diagrams in hardware design are twofold, one, since designers can visualize waveforms of signals they are easy to comprehend and two, they are very convenient for specifying ordering and timing constraints between the events.

There have been numerous attempts at formalizing timing diagram constraints in the framework of temporal logics such as the *timing diagram logic* [46], with *LTL formulas* [26], and as *synchronous regular timing diagrams* [8]. Moreover, there are industry standard property specification languages such as *PSL-Sugar* and *OVA* for associating temporal assertions to hardware designs [40]. The main motivation for these attempts was to exploit automatic verification techniques that

these formalisms support for validation and automatic circuit synthesis. However, commenting on their success, Fisler *et al.* state that

> The less than satisfactory adoption of formal methods in timing diagram domain can be partly attributed to the gulf that exists between graphical timing diagrams and textual temporal logic – expressing various timing dependencies that can exist among signals that can be illustrated so naturally in timing diagrams is rather tedious in temporal logics [26].

As a result, hardware designers use timing diagrams informally without any well defined semantics which make them unamenable to automatic design verification techniques.

In this chapter, we define a logic SeCeNL which is elementarily decidable syntactic subset of QDDC. SeCeNL includes quantifier and negation-free fragment of QDDC together with nominals and usage modalities. We claim that SeCeNL provides a natural, modular and succinct formalism for encoding timing diagram requirements, because of following important features.

1. The use of a quantifier and negation-free subset **SeCe** (Semi extended Chop expressions) of QDDC is sufficient for formalizing timing diagram patterns (constraints) imposed by a single waveform.

2. We extend SeCe with *nominals* (i.e. **SeCeN**). These are auxiliary temporal variables which are true at exactly one position. This allows us to mark the occurrence of an event in a waveform. Nominals can be used to specify ordering and synchronization constraints between distinct events[47], which may occur within (see Figure 3.1) or across the waveforms (see Figure 3.2).

For example, the timing diagram in Figure 3.1 stating that P transits from 0 to 1 somewhere in interval u to $u+3$ cycles is captured by the **SeCeN** formula `[!P]^<u>^(slen=3 && [!P]^[[P]])^[[P]]`.

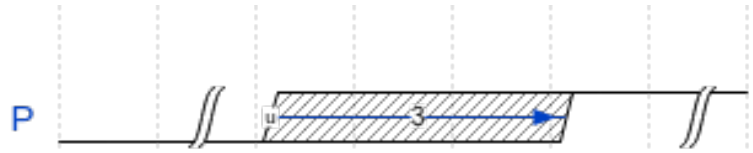


Figure 3.1: Timing diagram with a nominal u and a timing constraint.

3. We also enhance the timing diagram specifications (and the logic SeCeN) with *usage modalities*, giving us **SeCeNL**. While timing diagrams visually specify patterns of occurrence of signals, they do not make precise the modalities of occurrences of such patterns. We explicitly introduce usage modalities such as **a**) initially in the behaviour, a specified pattern must occur, or that **b**) every occurrence of *pattern1* is necessarily and immediately followed by an occurrence of *pattern2*, or that **c**) occurrence of a specified pattern is forbidden anywhere within a behaviour. In this, we are inspired by Allen's Interval Algebra relations [4] as well as the LSC operators of Harel for message sequence charts [32]. We confine ourselves to *usage modalities* where good things are achieved within specified bounds. For example, in specifying a modulo 6 counter, we can say that the counter will stabilize before completion of first 15 cycles. It may be noted that, technically, our usage modalities only give rise to "safety" properties (in the sense of Alpern and Schneider [6]).

We give *language preserving translation of timing diagrams into SeCeNL formulas*. This translation is succinct, in fact, linear time computable in the size of the timing diagram. Moreover, the translation is compositional, i.e. it translates each element of the timing diagram as a sub-formula and overall specification is just the conjunction of such formulas (constraints). Hence, the translation preserves the structure of the diagram.

Subsequently, we give linear time translation of SeCeNL formulas into QDDC, which allows the use of tools such as DCVALID [83, 84] for checking consistency as well as model-checking of timing diagram requirements. Later chapters in this thesis explore how correct-by-construction controllers can be automatically synthesized from such requirements. The translation scheme has been implemented into a tool.

The succinctness and compositionality of logic SeCeNL for modelling of timing diagrams is shown by several examples encoded as SeCeNL formula and comparing them with the formulas in logics such as PSL-Sugar and MTL. PSL-Sugar extends LTL with SERE (regular expressions with intersection) and counting which are similar to SeCe. In spite of this similarity, we show some examples where SeCeNL formula is at least one exponent more succinct as compared to PSL-Sugar. This is essentially due to the use of *nominals*.

In Section 3.1 we formally define the syntax and semantics of logic SeCeNL. Formalization timing diagram using SeCeNL is presented in Section 3.2. We illustrate the formalization of timing diagram requirements by a case study of a Mine-pump controller in Section3.3.

3.1 Logic SeCeNL: Syntax and Semantics

3.1.1 Chop expressions: Ce and SeCe

Definition 5 (Ce and SeCe)**.** *The logic* Semi extended Chop expressions *(SeCe) is a syntactic subset of QDDC in which the quantification operators ex p. D, all p. D and negation operator* ! *are not allowed. The logic* Chop expressions *(Ce) is a sub-logic of SeCe in which conjunction && is not allowed.* □

We define the *size of a formula* as the summation of number of literal and operators present in the formula. Note that Ce and SeCe can include counting constraints. Throughout this thesis, it is assumed that constants occurring in formulas are represented in binary form so that a natural number constant c contributes $log(c)$ to the size of the formula.

Lemma 6. *For any chop expression (Ce) D of size n we can effectively construct a language equivalent DFA $\mathscr{A}$ of size $O(2^{2^n})$.*

Proof. We observe that for any chop expression D we can construct a language equivalent *NFA* which is at most exponential in size of D including the constants appearing in it (for a detailed proof see [9] wherein a similar result has been proved). But this implies there exists a *DFA* of size 2^{2^n} which accepts exactly the set of words σ such that $\sigma \models D$. □

Corollary 7. *For any SeCe D of size n we can effectively construct a language equivalent DFA $\mathscr{A}$ of size $O(2^{2^{2^n}})$.*

Proof. The proof follows from the definition of SeCe, Lemma 6 and from the fact that there can be $O(n)$ conjuncts. Hence, the size of the product of *DFA*s can be

at most exponential in the size of individual *DFA*s. A detailed proof can be found in [9]. □

We now introduce our logic SeCeNL which builds upon SeCe (semi extended chop expressions) by augmenting them with *nominals* and *usage modalities*. Nominals are the auxiliary temporal variable which are $true$ exactly at one position in the behaviour.

Syntax : Let D, D_1 and D_2 range over SeCe formulas and let Θ, Θ_1 and Θ_2 range over subset of propositional variables occurring in SeCe formula. The notation $D : \Theta$, called a *nominated formula*, denotes that the formula D declares set of fresh nominals Θ. The syntax of SeCeNL formula is as follows.

$$\begin{aligned}&\mathbf{initImplies}(D_1 : \Theta_1 \leadsto D_2 : \Theta_2) \mid \mathbf{anti}(D : \Theta) \mid \\ &\mathbf{pref}(D : \Theta) \mid \mathbf{implies}(D_1 : \Theta_1 \leadsto D_2 : \Theta_2)\end{aligned}$$

We require that the sets Θ_1, Θ_2 are mutually disjoint. Here, D is defined over propositions $\Sigma \cup \Theta$, D_1 over propositions $\Sigma \cup \Theta_1$ and D_2 over propositions $\Sigma \cup \Theta_1 \cup \Theta_2$.

A composite SeCeNL specification is a conjunction of SeCeNL formulas of the form above. The operators *initImplies*, *anti*, *pref* and *implies* are called the usage modalities. As a convention, $D : \{\}$ is abbreviated as D when the set of nominals Θ is empty.

Usage Modalities:

For simplicity, we first explain the usage modalities without any nominals. This is subsequently extend to full SeCeNL. It may be recalled that for a given word

σ and a position $i \in dom(\sigma)$, we state that $\sigma, i \models D$ iff $\sigma, [0,i] \models D$. Thus, the interpretation is that the past of the position i in execution σ satisfies D. We say $\sigma' \leq_{prefix} \sigma$ if σ' is a prefix of σ, and $\sigma' <_{prefix} \sigma$ if σ' is a proper prefix of σ. We also define a derived operator over a QDDC formula D denoted as $\Xi(D)$ such that $\Xi(D) = D \;\&\&\; !(D\hat{}ext)$, which says that if $\sigma, [b,e] \models \Xi(D)$ then $\sigma, [b,e] \models D$ and there exists no proper prefix interval $[b,e_1]$, (i. e. $[b,e_1] \in Intv(\sigma)$ and $b \leq e_1 < e$) such that $\sigma, [b,e_1] \models D$.

We first explain the semantics of *usage modalities* assuming that no nominals are used in the specification. Thus, we consider SeCeNL where Θ, Θ_1 and Θ_2 are all empty.

- $L(\mathbf{pref}(D)\ ) = \{\sigma \mid \forall \sigma' \leq_{prefix} \sigma : \sigma' \models D\}$. Operator $\mathbf{pref}(D)$ denotes that D holds invariantly throughout the execution.

- $L(\mathbf{initImplies}(D_1 \rightsquigarrow D_2)) = \{\sigma \mid \forall j : (\sigma, [0,j] \models D_1 \Rightarrow \exists k \leq j :\ \sigma, [0,k] \models D_2)\}$. Operator $\mathbf{initImplies}(D_1 \rightsquigarrow D_2)$ states that if j is the first position which satisfies D_1 in the execution then there exists an $i \leq j$ such that i satisfies D_2. Thus, initially in the behaviour, D_2 holds before the first occurrence of D_1. In case D_1 never holds, then D_2 is not required to hold.

- $L(\mathbf{anti}(D)) = \{\sigma \mid \forall i,j : \sigma, [i,j] \not\models D\}$. Operator $\mathbf{anti}(D)$ states that there is no observation sub interval of the execution which satisfies D.

- $L(\mathbf{implies}(D_1 \rightsquigarrow D_2)) = \{\sigma \mid \forall i,j : (\sigma, [i,j] \models D_1 \Rightarrow \sigma, [i,j] \models D_2)\}$. Operator $\mathbf{implies}(D_1 \rightsquigarrow D_2)$ states all observation intervals which satisfy D_1 will also satisfy D_2.

Based on this semantics, we can translate an atomic SeCeNL formula ζ without nominals into equivalent *QDDC* formula $\aleph(\zeta)$ as follows. The translation is

as follows.

1. $\aleph(\textbf{pref}(D)) \stackrel{\text{def}}{\equiv} !((!D)^{\wedge} true)$.
2. $\aleph(\textbf{initImplies}(D_1 \rightsquigarrow D_2)) \stackrel{\text{def}}{\equiv} \textbf{pref}(D_1 \Rightarrow D_2{}^{\wedge} true)$.
3. $\aleph(\textbf{anti}(D)) \stackrel{\text{def}}{\equiv} !(true\ {}^{\wedge} D^{\wedge} true)$.
4. $\aleph(\textbf{implies}(D_1 \rightsquigarrow D_2)) \stackrel{\text{def}}{\equiv} [](D_1 \Rightarrow D_2)$.

Lemma 8. *For any $\zeta \in$ SeCeNL, if ζ does not use nominals then $\sigma \in L(\zeta)$ iff $\sigma \in L(\aleph(\zeta))$.*

Proof. The proof follows from examination of the semantics of ζ and the definition of $\aleph(\zeta)$. We consider the case of $\zeta = \textbf{implies}(D_1 \rightsquigarrow D_2)$. The other cases are similar and omitted.

From the transformation given above, $\aleph(\textbf{implies}(D_1 \rightsquigarrow D_2)) = [](D_1 \Rightarrow D_2)$. From the semantics of $[]$ operator we get
$L([](D_1 \Rightarrow D_2)) = \{\sigma \mid \forall i,j : \sigma,[i,j] \models (D_1 \Rightarrow D_2)\}$, which is equivalent to $\{\sigma \mid \forall i,j : (\sigma,[i,j] \models D_1 \Rightarrow \sigma,[i,j] \models D_2)\} = L(\textbf{implies}(D_1 \rightsquigarrow D_2))$. □

A set $S \subseteq \Sigma^*$ is *prefix closed* if $\sigma \in S$ then $\forall \sigma' : \sigma' \leq_{prefix} \sigma \Rightarrow \sigma' \in S$.

Lemma 9. *For any $\zeta \in$ SeCeNL without nominals. The language $L(\zeta)$ is prefix closed.*

Proof. From the definition of $L(\zeta)$, it is easy to see that $L(\zeta)$ is always prefix closed. For example, $L(\textbf{implies}(D_1 \rightsquigarrow D_2)) = L([](D_1 \Rightarrow D_2))$ (From Lemma 8).

By definition $\sigma,[b,e] \models []D$ iff $\sigma,[b',e'] \models D$ for all sub-intervals $b \leq b' \leq e' \leq e$. Therefore, if $\sigma,[b,e] \models []D$ then $\sigma,[b,e'] \models []D$ will also be valid for all prefix intervals $b \leq e' \leq e$. Thus, for any formula D, $L([]D)$ is always prefix closed. This proves the result. □

Nominals:

Consider a nominated formula $D : \Theta$ where D is a SeCe formula. As we shall see later, the propositional variables in Θ are treated as fresh "place holder" variables which are meant to be true exactly at one point. Following [49], we call them *nominals*.

Given an interval $[b,e] \in Intv(\mathbb{N})$ we define a *nominal valuation* over $[b,e]$ to be a map $\nu : \Theta \to \{i \mid b \leq i \leq e\}$. It assigns a unique position within $[b,e]$ to each nominal variable belonging to Θ. We can then straightforwardly define $\sigma, [b,e] \models_\nu D$ by constructing a word σ_ν over $\Sigma \cup \Theta$ such that $\forall p \in \Sigma : p \in \sigma_\nu(i) \Leftrightarrow p \in \sigma(i)$ and $\forall u \in \Theta : u \in \sigma_\nu(i) \Leftrightarrow \nu(u) = i$. Then $\sigma_\nu, [b,e] \models D \Leftrightarrow \sigma, [b,e] \models_\nu D$. We state that ν_1 over Θ_1 and ν_2 over Θ_2 are consistent if $\nu_1(u) = \nu_2(u)$ for all $u \in \Theta_1 \cap \Theta_2$. We denote this by $\nu_1 \# \nu_2$.

Now we consider usage modalities where nominals are used and shared between different parts D_1 and D_2 of a usage modality such as $\mathbf{implies}(D_1 : \Theta_1 \rightsquigarrow D_2 : \Theta_2)$, where ν_1, ν_2 are over Θ_1 and Θ_2 respectively. See Example 10 below.

Semantics of SeCeNL: In the following v_i denotes nominal valuation over Θ_i.

- $L(\mathbf{pref}(D_1 : \Theta_1)\ ) = \{\sigma \mid \forall \sigma' \leq_{prefix} \sigma\ \exists \nu_1 :\ \ \sigma' \models_{\nu_1} D_1\}$.
- $L(\mathbf{initImplies}(D_1 : \Theta_1 \rightsquigarrow D_2 : \Theta_2)) = \{\sigma \ \mid\ \forall j \forall \nu_1 :\ \sigma, [0,j] \models_{\nu_1} (D_1 \Rightarrow \exists k \leq j . \exists \nu_2 :\ \sigma, [0,k] \models_{\nu_2} D_2)\}$. Note that D_2 may refer to nominals $\Theta_1 \cup \Theta_2$.
- $L(\mathbf{anti}(D_1 : \Theta_1)) = \{\sigma \mid \forall i,j \forall \nu_1 :\ \sigma, [i,j] \not\models_{\nu_1} D_1\}$.
- $L(\mathbf{implies}(D_1 : \Theta_1 \rightsquigarrow D_2 : \Theta_2)) = \{\sigma \mid \forall i,j \forall \nu_1 : (\sigma, [i,j] \models_{\nu_1} D_1 \Rightarrow \exists \nu_2 : \sigma, [i,j] \models_{\nu_2} D_2)\}$. Note that D_2 may refer to nominals $\Theta_1 \cup \Theta_2$.

Example 10 (lags)**.** *Let* $D_1 : \{u, v\}$ *be the formula* `(<u> ^ [[P]] && ((slen=n) ^ <v> ^ true)` *which holds for an interval where P is true throughout the interval and v marks the* $n+1$ *position (here n is a natural number constant) from u denoting the start of the interval. Let* $D_2 : \{\ \}$ *be the formula* `true ^ <v> ^ [[Q]]`*. Here,* D_2 *refers to nominal v defined in* D_1*. Then,* ***implies***$(D_1 : \{u, v\} \rightsquigarrow D_2 : \{\ \})$ *states that for all observation intervals* $[i, j]$ *and all nominal valuations* ν *over* $[i, j]$ *if* $\sigma, [i, j] \models_\nu D_1$ *then* $\sigma, [i, j] \models_\nu D_2$*. A* live timing diagram[1] *based specification of this example is given in Figure 3.4.*

Based on the above semantics, we now formulate a QDDC formula equivalent to a SeCeNL formula. We will make essential use of quantification $(ex\ p.\ D)$ and $(all\ p.\ D)$. We first define relativized quantifiers to restrict variables in Θ to singletons. Given a set of nominals $\Theta = \{u_1, \ldots, u_n\}$, we define

$$singleton(\Theta) \stackrel{\text{def}}{\equiv} (scount\ u_1 = 1\ \ \&\&\ \cdots\ \&\&\ \ scount\ u_n = 1)$$

It states that in current interval each nominal occurs exactly once. Then, we define derived operators $\forall^1_\Theta : D \stackrel{\text{def}}{\equiv} all\ \Theta.\ (singleton(\Theta)\ \ \Rightarrow\ \ D)$ and $\exists^1_\Theta : D \stackrel{\text{def}}{\equiv} ex\ \Theta.\ (singleton(\Theta)\ \ \&\&\ \ D)$.

SeCeNL to QDDC: We now define the translation $\aleph$ from SeCeNL to QDDC.

1. $\aleph\,(\mathbf{pref}(D_1 : \Theta_1)\ ) \stackrel{\text{def}}{\equiv} \mathbf{pref}(\exists^1_{\Theta_1} : D_1)\ .$
2. $\aleph\,(\mathbf{initImplies}(D_1 : \Theta_1 \rightsquigarrow D_2 : \Theta_2)) \stackrel{\text{def}}{\equiv} \mathbf{pref}(\forall^1_{\Theta_1} :\ (D_1 \Rightarrow ((\exists^1_{\Theta_2} : D_2)\hat{}true)))\ .$
3. $\aleph\,(\mathbf{anti}(D_1 : \Theta_1)) \stackrel{\text{def}}{\equiv} !(true\ \hat{}(\exists^1_{\Theta_1} : D_1)\hat{}true\ ).$
4. $\aleph\,(\mathbf{implies}(D_1 : \Theta_1 \rightsquigarrow D_2 : \Theta_2)) \stackrel{\text{def}}{\equiv} [](\forall^1_{\Theta_1} :\ (D_1 \Rightarrow (\exists^1_{\Theta_2} : D_2)))).$

[1]The illustration was made with WaveDrom and due to its limitation on naming nominals we were forced to rename the nominals *u* and *v* in *Q* as *a* and *b* in the diagram, respectively.

Some additional useful usage modalities can be defined as derived modalities as follows:

– We define the usage modality $\textbf{follows}(D_1 : \Theta_1 \rightsquigarrow D_2 : \Theta_2 / D_3 : \Theta_1)$ which states that if any observation interval $[i, j]$ satisfies D_1 and there is a following interval $[j,k]$ which satisfies D_3 for the first time, then there exists a prefix interval of $[j,k]$ which satisfies D_2. This can be defined using **implies** as shown below.

$$\textbf{implies}(((D_1 : \Theta_1)^\frown\langle u\rangle^\frown(\Xi(D_3) : \Theta_3)) \rightsquigarrow (true^\frown\langle u\rangle^\frown(D_2 : \Theta_2)))$$

Theorem 11. *For any word σ over Σ and any $\zeta \in$ SeCeNL we have that $\sigma \in L(\zeta)$ iff $\sigma \in L(\aleph(\zeta))$. Moreover, the translation $\aleph(\zeta)$ can be computed in time linear in the size of ζ.*

Proof. The proof follows from the semantics of ζ and the definition of $\aleph(\zeta)$. We consider the case of $\zeta = \textbf{implies}(D_1 : \Theta_1 \rightsquigarrow D_2 : \Theta_2)$. The other cases are similar and omitted.

From the transformation $\aleph$ given above, we have $\textbf{implies}(D_1 : \Theta_1 \rightsquigarrow D_2 : \Theta_2) \stackrel{\text{def}}{\equiv} [](\forall^1_{\Theta_1} : (D_1 \Rightarrow (\exists^1_{\Theta_2} : D_2)))$. Let the language $L([](\forall^1_{\Theta_1} : (D_1 \Rightarrow (\exists^1_{\Theta_2} : D_2))))$ be denoted by L_1. Then, from the semantics of [] operator we get
$L_1 = \{\sigma \mid \forall i, j : \sigma, [i, j] \models (\forall^1_{\Theta_1} : (D_1 \Rightarrow (\exists^1_{\Theta_2} : D_2)))\}$,
Assuming that ν_1 and ν_2 provides valuation for Θ_1 and Θ_2 respectively then
$L_1 = \{\sigma \mid \forall i, j \forall \nu_1 : (\sigma, [i, j] \models_{\nu_1} D_1 \Rightarrow \exists \nu_2 : \sigma, [i, j] \models_{\nu_2} D_2)\} = L(\textbf{implies}(D_1 : \Theta_1 \rightsquigarrow D_2 : \Theta_2))$.

It is straightforward to see from transformation $\aleph$ that the size of transformed formula will be linear in the size of SeCeNL formula. □

Theorem 12. *For any $\zeta \in$ SeCeNL the size of the automaton $\mathscr{A}(\zeta)$ for ζ is $O(2^{2^{2^{2^n}}})$ (tower of height 5), where n is the size of ζ.*

Proof. The theorem can be proved for each form of ζ. We consider the case $\zeta = \textbf{implies}(D_1 : \Theta_1 \rightsquigarrow D_2 : \Theta_2)$, which gives the QDDC formula $[](\forall^1_{\Theta_1} : (D_1 \Rightarrow (\exists^1_{\Theta_2} : D_2)))$. As D_1 and D_2 belong to SeCe, the size of automaton for each of them is $O(2^{2^{2^n}})$ (See Corollary 7). The size of automaton for $(\exists^1_{\Theta_2} : D_2)$ is $O(2^{2^{2^{2^n}}})$ because of the existential quantification. Therefore, automaton for $(D_1 \Rightarrow (\exists^1_{\Theta_2} : D_2))$ has the size $O(2^{2^{2^n}} \times 2^{2^{2^{2^n}}})$, which simplifies to $O(2^{2^{2^{2^n}}})$. Finally, the outer $\forall$ quantification and $[]$ operator (both of which are $\forall$ type) together add one more exponent, which proves the result. Similarly, The size for other form of ζ can also be calculated. □

3.2 Formalizing timing diagrams

In this section we give a formal semantics to timing diagrams and formula translation from timing diagrams to SeCeNL. We first give a textual syntax for timing diagrams which is derived from the timing diagram format of WaveDrom [25, 105].

The symbols in a *waveform* come from $\Lambda = \{0, 1, 2, \mathsf{x}, 0|, 1|, 2|, \mathsf{x}|\}$ and Θ, an atomic set of nominals. Let $\Gamma = \Theta \cup \Lambda$. The syntax of a *waveform* over Γ is given by the grammar:

$$\pi := \lambda \mid u : \pi \mid \pi_1 \pi_2,$$

where $u \in \Theta$ and $\lambda \in \Lambda$. We call the elements in Θ the *nominals*. As we shall see later, when we convert a waveform to a SeCeNL formula the nominals that appear in the formula are exactly the nominals in the waveform and hence the name. Let Wf be the set of all waveforms over Γ.

An example of a waveform is 01a:2x011xb:x2|220c:00 with the set of nominals $\Theta = \{\mathsf{a},\mathsf{b},\mathsf{c}\}$. This can be shown graphically as waveform P in Figure 3.2. Intuitively, in a waveform 0 denotes *low*, 1 *high*, 2 and x *don't care*s (there is a subtle difference between 2 and x when stuttering operator "|" is applied to them, as explained in Section 3.2.1) and "|" the *stuttering* operator (it extends previous cycle till any further change in value). Based on these notations the example waveform mentioned above states that the signal is initially *low*, then it becomes *high* in next cycle, then signal goes to don't care and the position of this transition is annotated by a nominal a. In next cycle also signal remains don't care followed by *low*, *high*, *high* and so on.

Let Σ be a set of propositional variables. A *timing diagram over* Σ is a tuple $\langle \mathscr{W}, \Sigma, C, \Theta \rangle$ where $\mathscr{W} = \{W_p \in \mathsf{Wf} \mid p \in \Sigma\}$ and $C \subset \Theta \times \Theta \times Intv(\mathbb{N})$ a set of timing constraints.

Figure 3.2 shows an example timing diagram $T = \langle \{W_p, W_q\}, \{p,q\}, \{(a,b,[10:10]),(a,\ d,[1:8]),(c,d,[20:30])\}, \{a,b,c,d,e,f\}\rangle$ along with its rendering in WaveDrom. WaveDrom renders 0 and 1 as low and high signal respectively. The don't care terms x and 2 are represented by shaded block and unshaded block respectively and the stuttering is shown by drawing a gap on the top. It may be noted that for rendering purpose the shared nominals have to be renamed in WaveDrom, e.g. a and c in W_q have been renamed g and h respectively. As in the case with SeCeNL formulas, nominals act as place holders in timing diagrams which can be shared among multiple waveforms. For example, W_p and W_q share the nominals a and c. As a result a timing constraint in one timing diagram can implicitly induce a timing constraint in the other. For instance, even though there is no direct timing constraint between a and c in W_p the constraints between a and d, and d and c

together impose one on them.

waveform W_p - 01a : 2x011xb : x2|220c : 00

waveform W_q - 00a : 0|d : 11|e : xxx|f : 01c : 11

timing constraints: d-a∈[1:8], c-d∈[20:30], b-a∈[10:10]

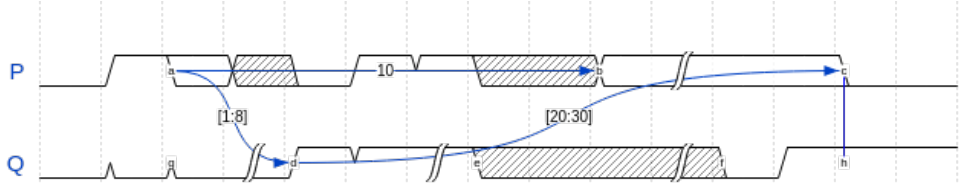


Figure 3.2: Timing diagram T and its WaveDrom rendering.

Let $T = \langle \mathscr{W}, \Sigma, C, \Theta \rangle$, $\mathscr{W} = \{W_p \in \mathsf{Wf} \mid p \in \Sigma\}$, be a timing diagram. Let $\nu : \Theta \to [b,e]$ be a nominal valuation. Let $\sigma : [0,n] \to 2^\Sigma$ be a word over Σ and for all $p \in \Sigma$ let $\sigma_p : [0,n] \to \{0,1\}$ given by $\sigma_p(i) = 1$ iff $p \in \sigma(i)$. Then the satisfaction relation σ_p over a waveform W under the valuation ν is defined as follows.

$$
\begin{aligned}
\sigma_p, [b,e] &\models_\nu 0 && \text{iff} \quad e = b+1 \text{ and } \sigma_p(b) = 0,\\
\sigma_p, [b,e] &\models_\nu 1 && \text{iff} \quad e = b+1 \text{ and } \sigma_p(b) = 1,\\
\sigma_p, [b,e] &\models_\nu \lambda && \text{iff} \quad e = b+1 \text{ and } \lambda \in \{2, \mathsf{x}\},\\
\sigma_p, [b,e] &\models_\nu 0| && \text{iff} \quad \forall b \le i < e : \sigma_p(i) = 0,\\
\sigma_p, [b,e] &\models_\nu 1| && \text{iff} \quad \forall b \le i < e : \sigma_p(i) = 1,\\
\sigma_p, [b,e] &\models_\nu 2| && \text{iff} \quad \forall b \le i < e : \sigma_p(i) \in \{0,1\},\\
\sigma_p, [b,e] &\models_\nu \mathsf{x}| && \text{iff} \quad \forall b \le i < e : \sigma_p(i) = 1 \text{ or } \forall b \le i < e : \sigma_p(i) = 0,\\
\sigma_p, [b,e] &\models_\nu u : W && \text{iff} \quad \nu(u) = b \text{ and } \sigma_p, [b,e] \models_\nu W,\\
\sigma_p, [b,e] &\models_\nu VW && \text{iff} \quad \exists b \le i < e : \sigma_p, [b,i] \models_{\nu_1} V \text{ and } \sigma_p, [i,e] \models_{\nu_2} W,\\
& && \qquad \text{and } \nu_1 \# \nu \text{ and } \nu_2 \# \nu.
\end{aligned}
$$

We say $\nu \models C$ iff $\forall (a,b,\langle l,r\rangle) \in C : \nu(b) - \nu(a) \in \langle l,r\rangle$. We define $\sigma,[b,e] \models_\nu \langle \mathscr{W},\Sigma,C,\Theta\rangle$ iff $\forall p \in \Sigma : \sigma_p,[b,e] \models_\nu W_p$ and $\nu \models C$.

3.2.1 Waveform to SeCeNL translation

We translate a waveform W_p to SeCeNL as follows: every 0 occurring in P is translated to {{! P}}, 1 to {{P}}, 2 and x to slen=1, 0| to pt || [! P], 1| to pt || [P], 2| to true, and x| to pt || [P] || [! P]. A nominal u that is appearing in W_p is translated to <u>. For instance, the waveform W_p=01a:2x011xb:x2|220c:00 in T of Figure 3.2 will be translated to SeCeNL formula as below.

({{! P}}^{{P}}^<a>^(slen=1)^(slen=1)^{{! P}}^{{P}}^{{P}}^(slen=1)^<b>^
(slen=1)^true^(slen=1)^(slen=1) ^{{! P}}^<c>^{{! P}}^{{! P}}).

We denote the translated SeCeNL formula by $\xi(T,W_p)$. Similarly we can translate W_q to get the formula $\xi(T,W_q)$. The timing constraints in C is translated to the SeCeNL formula $\xi(T,C)$ as follows.

((true^<a>^((slen≥ 1) && (slen≤ 8))^<d>^true) &&
(true^<d>^((slen≥ 20) && (slen≤ 30))^<c>^true) &&
(true^<a>^(slen=10)^<b>^true)).

We define $\xi(T) = \xi(T,W_p)$ && $\xi(T,W_q)$ && $\xi(T,C)$. For a timing diagram $T = \langle \mathscr{W},\Sigma,C,\Theta\rangle$, $\mathscr{W} = \{W_p \mid p \in \Sigma\}$ we define $\xi(T) = \bigwedge_{p\in\Sigma} \xi(T,W_p) \bigwedge$ && $\xi(T,C)$.

Theorem 13. *Let T be a timing diagram. Then, for all $\sigma \in \Sigma^+$, for all $[b,e] \in Intv(\sigma)$ and for all nominal valuation ν over $[b,e]$, we have $\sigma,[b,e] \models_\nu T$ iff $\sigma,[b,e] \models_\nu \xi(T) : \Theta$. Also, the translation $\xi(T) : \Theta$ is linear in the size of T.*

Proof. The proof follows straightforwardly by induction on the length of the waveform. □

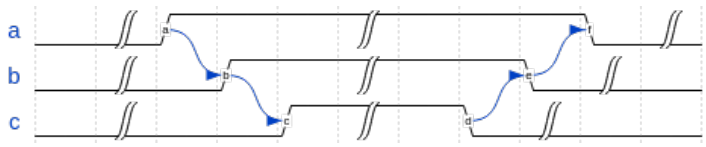


Figure 3.3: Example 1.

Due to the above theorem we can now use timing diagrams in place of nominated formulas $D : \Theta$ within the usage modalities. We call such timing diagrams *live timing diagrams*. For an example of a live timing diagram see Figure 3.4.

3.2.2 Comparison with other temporal logics

In previous section, Lemma 13 showed that timing diagrams can be translated to equivalent SeCeNL formulas with only linear blowup in size. In this section we compare our logic SeCeNL with other relevant logics in the literature viz, LTL [88], discrete time MTL [7], and PSL-Sugar [40]. Of these, PSL-Sugar is the most expressive and discrete time MTL and LTL are its syntactic subset. We show by examples that SeCeNL formulas are more succinct (smaller in size) than PSL-Sugar and we believe that they capture the diagrams more directly.

Example (Ordered Stack) Let us now consider the timing diagram in Figure 3.3 adapted from [26]. Rise and fall of successive signals a, b and c follow a stack discipline. The language described by it is given by the SeCeNL formula:

$$([!a]\,\hat{}\,{<}ua{>}\,\hat{}\,[a]\,\hat{}\,{<}va{>}\,\hat{}\,[!a])\ \&\&\ ([!b]\,\hat{}\,{<}ub{>}\,\hat{}\,[b]\,\hat{}\,{<}vb{>}\,\hat{}\,[!b])\ \&\&$$
$$([!c]\,\hat{}\,{<}uc{>}\,\hat{}\,[c]\,\hat{}\,{<}vc{>}\,\hat{}\,[!c])\ \&\&$$
$$(ext\hat{}\,{<}ua{>}\,\hat{}\,ext\ \hat{}\,{<}ub{>}\,\hat{}\,true)\ \&\&\ (true\hat{}\,{<}ub{>}\,\hat{}\,ext\ \hat{}\,{<}uc{>}\,\hat{}\,true)\ \&\&$$
$$(true\hat{}\,{<}vc{>}\,\hat{}\,ext\ \hat{}\,{<}vb{>}\,\hat{}\,true)\ \&\&\ (true\hat{}\,{<}vb{>}\,\hat{}\,ext\ \hat{}\,{<}va{>}\,\hat{}\,true).$$

Note that first three conjuncts exactly correspond to the three waveforms. The next four conjuncts correspond to the four arrows (ordering constraints) between the waveforms. Here, ua, ub, uc denote the nominals a, b, c and vc, vb, va denote the nominals d, e, f in the Figure 3.3. In general, if n signals are stacked, its SeCeNL specification has size $O(n)$.

An equivalent MTL (or LTL) formula is given by:

$$([\neg a \wedge \neg b \wedge \neg c]\ \mathbf{UU}\ ([a \wedge \neg b \wedge \neg c]\ \mathbf{UU}\ ([a \wedge b \wedge \neg c]\ \mathbf{UU}\ ([a \wedge b \wedge c]\ \mathbf{UU}\ ([a \wedge b \wedge \neg c]\ \mathbf{UU}\ ([a \wedge \neg b \wedge \neg c]\ \mathbf{UU}\ [\neg a \wedge \neg b \wedge \neg c])))))$$

where $a\ \mathbf{UU}\ b$ is the derived modality $a \wedge \mathbf{X}(a\mathbf{U}b)$. For a stack of n signals, the size of the MTL formula is $O(n^2)$. Above formula is also a PSL-Sugar formula. We attempt to specify the pattern as a PSL-Sugar regular expression as follows:

$$((\neg a \wedge \neg b \wedge \neg c;)[+];\ (a \wedge \neg b \wedge \neg c;)[+];\ (a \wedge b \wedge \neg c;)[+];\ (a \wedge b \wedge c;)[+];(a \wedge b \wedge \neg c;)[+];\ (a \wedge \neg b \wedge \neg c;)[+];\ (\neg a \wedge \neg b \wedge \neg c;)[+].$$

For a stack of n signals, the size of the PSL-Sugar SERE expression is $O(n^2)$. We believe that there is no formula of size $O(n)$ in PSL-Sugar which can express the above property. Compare this with size $O(n)$ formula of SeCeNL.

Example (Unordered Stack) In ordered stack signal a turns on first and turns off last followed by signals b, c in that order. We consider a variation of the ordered stack example above where signals turn on and off in first-on-last-off order but there is no restriction on which signal becomes high first. This can be compactly

specified in SeCeNL as follows.

$$([!a]\ \hat{}\ <ua>\ \hat{}\ [a]\ \hat{}\ <va>\ \hat{}\ [!a])\ \&\&\ ([!b]\ \hat{}\ <ub>\ \hat{}\ [b]\ \hat{}\ <vb>\ \hat{}\ [!b])\ \&\&$$
$$([!c]\ \hat{}\ <uc>\ \hat{}\ [c]\ \hat{}\ <vc>\ \hat{}\ [!c])\ \&\&$$
$$(ext\ \hat{}\ <u1>\ \hat{}\ ext\ \hat{}\ <u2>\ \hat{}\ ext\ \hat{}\ <u3>\ \hat{}ext$$
$$\hat{}\ <v3>\ \hat{}\ ext\ \hat{}\ <v2>\ \hat{}\ ext\ \hat{}\ <v1>\ \hat{}\ ext\ )\ \&\&$$
$$Bijection(ua, ub, uc, va, vb, vc, u1, u2, u3, v1, v2, v3)$$

where formula *Bijection* below states that there is one to one correspondence between positions marked by ua, ub, uc, va, vb, vc and positions marked by $u1, u2, u3$, $v1, v2, v3$. Moreover, if u_a maps to say u_3 than v_a must map to v_3 and so on.

$$[[(u1\ ||\ u2\ ||\ u3) \Leftrightarrow (ua\ ||\ ub\ ||\ uc)]]\ \&\&\ [[\bigwedge_{1\leq i,j\leq 3, i\neq j} !(u_i\ \&\&\ u_j)]]$$
$$[[(v1\ ||\ v2\ ||\ v3) \Leftrightarrow (va\ ||\ vb\ ||\ vc)]]\ \&\&\ [[\bigwedge_{1\leq i,j\leq 3, i\neq j} !(v_i\ \&\&\ v_j)]]$$
$$\bigwedge_{1\leq i\leq 3, j\in a,b,c} (true\ \hat{}\ <u_i\ \&\&\ u_j>\ \hat{}\ true \Leftrightarrow\ true\ \hat{}\ <v_i\ \&\&\ v_j>\ \hat{}\ true).$$

Note that, in general, if n signals are stacked, then the above SeCeNL specification has size $O(n^2)$.

Now we discuss encoding of unordered stack in PSL-Sugar. In absence of nominals, it is difficult to state the above behaviour succinctly in logics PSL-Sugar even using its SERE regular expressions. Each order of occurrence of signals has to be enumerated as a disjunction where each disjunct is as in the example ordered stack (where the order was a, b, c). As there are $n!$ orders possible between n signals, the size of the PSL-Sugar formula is also $O(n!)$. We believe that there is no polynomial size formula in PSL-Sugar encoding this property. This shows that SeCeNL is exponentially more succinct as compared to PSL-Sugar.

In general, presence of nominals distinguishes SeCeNL from logics like PSL-Sugar. In formalizing behaviour of hardware circuits, it has been proposed that regular expressions are not enough and operators such as pipelining have been

introduced [26]. These are a form of synchronization and they can be easily expressed using nominals. (See [86].)

3.3 Case study: Mine-pump Specification

We first specify some useful generic timing diagram properties.

- **lags**(P,Q,n): *it is defined by Figure 3.4*. It specifies that in any observation interval if P holds continuously for $n+1$ cycles and persists then Q holds from $(n+1)^{th}$ cycle onwards and persists till P persists.

- **tracks**(P,Q,n): *defined by Figure 3.5*. In any observation interval if P becomes true then Q sustains as long as P sustains or upto n cycles whichever is shorter.

- **sep**(P,n): *Figure 3.6 defines this property*. If an interval which begins with a falling edge of P and ends with next rising edge of P then the length of the interval should be at least n cycles.

- **ubound**(P,n): *Figure 3.7 defines this property*. In any observation interval P can be continuously true for at most n cycles.

Note that we have presented these formulas diagrammatically.

We now state the Mine-pump specification. Imagine a pump which keeps the water level in a mine under control. The pump is driven by a controller which can switch it *on* and *off*. Mines are prone to highly flammable methane leakage trapped underground. So as a safety measure if a methane leakage is detected the controller is not allowed to keep the pump on.

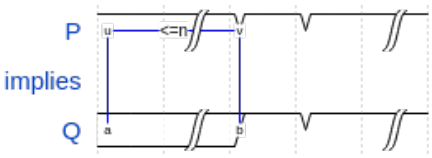


Figure 3.4: **lags**(P,Q,n).

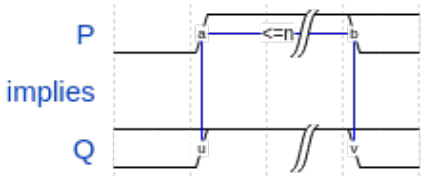


Figure 3.5: **tracks**(P,Q,n).

Figure 3.6: **sep**(P,n).

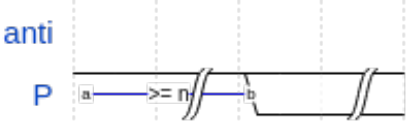


Figure 3.7: **ubound**(P,n).

The controller has two input sensors - HH2O which becomes *true* when water level is high, and HCH4 which is *true* when there is a methane leakage. It can generate two output signals - ALARM which is set to *true* to sound/persist the alarm, and PUMPON which is set to *true* to keep the pump on. The objective of the controller is to *safely* operate the pump and the alarm in such a way that the water level is never dangerous, indicated by the propositional variable DH2O, whenever certain assumptions hold. We have the following assumptions on the mine and the pump.

- Sensor reliability assumption: **pref**$([[\text{DH2O} \Rightarrow \text{HH2O}]])$. If HH2O is false then so is DH2O.

- Water seepage assumptions: **tracks**$(\text{HH2O}, \text{DH2O}, \kappa_1)$. The minimum no. of cycles for water level to become dangerous once it becomes high is κ_1.

- Pump capacity assumption: **lags**$(\text{PUMPON}, !\,\text{HH2O}, \kappa_2)$. If pump is kept on for at least $\kappa_2 + 1$ cycles then water level will not be high after κ_2 cycles.

- Methane release assumptions: **sep**(HCH4, κ_3) and **ubound**(HCH4, κ_4). The minimum separation between the two leaks of methane is κ_3 cycles and the methane leak cannot persist for more than κ_4 cycles.

- Initial condition assumption: **init**(<! HH2O> && <! HCH4>, $slen = 0$). Initially neither the water level is high nor there is a methane leakage.

Let the conjunction of these SeCeNL formulas be denoted as `MineAssume`. The commitments are:

- Alarm control: **lags**(HH2O, ALARM, κ_5) and **lags**(HCH4, ALARM, κ_6) and **lags**(! HH2O && ! HCH4, ! ALARM, κ_7). If the water level is high then alarm will be high after κ_5 cycles and if there is a methane leakage then alarm will be high after κ_6 cycles. If neither the water level is dangerous nor there is a methane leakage, then alarm should be off after κ_7 cycle.

- Safety condition: **pref**([[! DH2O && (HCH4 $\Rightarrow$! PUMPON)]]) . The water level should never become dangerous and whenever there is a methane leakage pump should be off.

Let the conjunction of these commitments be denoted as `MineCommit`. Then, the requirement over the mine-pump controller is given by the formula `MineAssume` $\Rightarrow$ `MineCommit`. Note that the requirement consists of a mixture of timing diagram constraints (such as pump capacity assumption above) as well as SeCeNL formulas (such as safety condition above). Hence the specification is heterogenous.

Given such specifications we have analyzed the specification for checking consistency of the requirements using tool DCVALID. We have also used the tool DCSynth (see Chapter 4) to automatically synthesize a controller for the values,

$\kappa_1 = 10$, $\kappa_2 = 2$, $\kappa_3 = 14$, $\kappa_4 = 2$, and $\kappa_5 = \kappa_6 = \kappa_7 = 1$. For these values, in under 1 second it outputs a SCADE/SMV controller with 140 states meeting the specification.

3.4 Discussion

In this chapter, we defined a logic SeCeNL which includes quantifier and negation-free fragment of QDDC together with usage modalities as well as nominals. SeCeNL provides a natural and convenient formalism for encoding timing diagram requirements. In fact, our proposed encoding extends the Timing Diagram notation with usage modalities which makes precise their use as a formal requirement. The translation of timing diagrams to SeCeNL is succinct and compositional. Hence, the translation preserves the structure of the diagram.

We have also formulated a reduction from a SeCeNL formula to an equivalent QDDC formula. This allows QDDC tools to be used for SeCeNL. It may be noted that, though expressively no more powerful than QDDC, the logic SeCeNL is considerably more efficient for satisfiability and model checking. We find that these problems have elementary complexity as compared with full QDDC which exhibits non-elementary complexity. Also, the presence of usage modalities and nominals makes it more convenient as compared to QDDC for practical use.

By implementing the above reductions, we have constructed a tool which converts a requirement, consisting of a conjunction of timing diagram specifications (augmented with usage modalities) and SeCeNL formulas, into an equivalent QDDC formula. In this sense, we handle heterogeneous specification. We could analyze the resulting formula using the QDDC tools DCVALID [83, 84] and DCSynth for model checking and controller synthesis, respectively.

Chapter 4

Guided Reactive Synthesis for High Quality Controllers

Controller synthesis aims at constructing a controller (say a Mealy Machine) algorithmically from a given temporal logic specification of its desired behaviour. Considerable amount of research has gone into the area of reactive synthesis and several tools are available for experimenting [62].

In this chapter, we propose a framework for *guided* controller synthesis from *regular properties* specified using the interval temporal logic QDDC [83, 84] to synthesize high quality controllers. The study of synthesis of controllers from regular properties was pioneered by Ramadge and Wonham [93]. The guidance is based on some *qualitative* properties, which are also specified in QDDC. Logic QDDC is especially suited for guided synthesis due to its succinctness as well as superiority in dealing with both *qualitative* and *quantitative* specifications.

In practice, user specification may contain certain requirements which cannot be guaranteed, but are *desirable*. We term the desirable properties as *soft require-*

ments. Soft requirements can be effectively used to guide the controller synthesis engine by synthesizing a controller which tries to maximizes the satisfaction of these soft requirements. For example, in a Mine pump controller which removes seepage of water, the soft requirement can state that keep the pump off as much as possible to save the electricity. The soft requirements may also be used when a specification consist of a conjunction of conflicting requirements. In this case, the user may resolve the conflict by making some of these requirements as soft. Therefore, soft requirements give us a capability to synthesize meaningful and practical controllers. However, most existing controller synthesis tools do not have the capability to deal with desirable or conflicting requirements.

Based on the above discussion, it may be noted that in practice we have some requirements which needs to be guaranteed, while others are desirable requirements. The desirable requirements may not be satisfied invariantly, but the frequency of their satisfaction should be maximized. Similarly, the requirements to be guaranteed also need suitable environmental assumptions under which commitments could be guaranteed. These assumptions can also be violated or satisfied intermittently. As a result, such environmental assumptions and the desirable requirements are by nature intermittent. Invariant requirements are non-recoverable i.e. once these requirements are violated at any position they never get satisfied again in future. This limitation has been well recognized in the literature [14, 39] and hence in this thesis we will try to formulate our requirements as intermittent instead of invariant. In section 4.3.2 we formally discuss a generic way to specify intermittent requirements in our notation.

We introduce a technique which allows synthesis of discrete controllers from *regular properties* (QDDC formulas) given as a tuple (I, O, D^h, D^s), where D^h and

D^s are QDDC formulas over a set of input and output propositions (I, O). Here, D^h and D^s are the **hard** and the **soft** requirement, respectively. *We synthesize a controller which, (a) invariantly satisfies D^h and (b) it meets D^s at "as many points as possible".* Meeting D^s "at as many points as possible" is achieved by synthesizing a controller which maximizes (optimizes) the cumulative count of D^s holding in next H moves, averaged over all the inputs of length H. Such a controller is called *H-Optimal* for D^s.

Example 14 (Arbiter for Mutually Exclusive Shared Resource)**.** *The arbiter has inputs r_i (denoting request for access) and outputs a_i (denoting acknowledgement for access) for each client $1 \leq i \leq n$. The specification consist of the following two requirements given as QDDC formulas.*

*−**Mutual Exclusion Requirement** R_1: $[[\ \wedge_{i \neq j}\ \neg(a_i \wedge a_j)\ ]]$, states that at every point in the execution, the access to the shared resource is mutually exclusive.*

*−**k-cycle Response Requirement** R_2: $[](\wedge_i\ ((([[r_i]]\ \&\&\ (slen >= (k-1)))\ \Rightarrow\ (scount\ a_i > 0))$, states that in any observation interval spanning k cycles if request from i^{th} client (r_i) is continuously high during the interval, then that client should get at least one access (a_i) within that observation interval. The property $R2$ is asserted for each client $1 \leq i \leq n$.*

As an example of conflicting requirements, consider the case where $k < n$. In this scenario, no controller can satisfy both requirements (i.e. their conjunction is unrealizable) e.g. when all clients request all the time. We may want to opt for an implementation, which mandatorily satisfies $R1$ and it tries to meet $R2$ "as much as possible". This can be specified in our framework by making $R1$ as hard and $R2$ as a soft requirement.

This chapter gives the algorithm for guided synthesis of a controller from the

specification given as tuple (I, O, D^h, D^s). We first define the term called *supervisor*, which is a *non-blocking* Mealy machine that may non-deterministically produce one or more outputs for each input. A supervisor may be refined to a sub-supervisor by resolving (pruning) the non-deterministic choice of outputs. A *controller* is a deterministic supervisor.

The synthesis algorithm starts by first computing the language equivalent monitor automaton for D^h. In the second step we compute *Maximally Permissive Supervisor* (MPS) from the monitor automaton. The supervisor MPS contains all the behaviours, which invariantly satisfy D^h and is non-blocking. Then *Maximally Permissive H-Optimal Supervisor* (MPHOS) is computed by pruning MPS. This is done by keeping only those outputs of MPS, which *H-Optimally* satisfy D^s. The computation of such H-Optimal supervisor draws upon the technique of finite horizon controller for Markov decision processes, pioneered by Bellman [11]. Finally, MPHOS is turned in to a controller by pruning the choice of outputs based on a user given preference ordering.

We have implemented this guided synthesis algorithm in a tool *DCSynth*. For representing automata, supervisors and controllers, the tool uses an efficient data structure SSDFA (See Section 2.3) originally introduced by the tool MONA [67]. Section 4.4 gives the details of the symbolic algorithms. We compare the performance of our tool with some state of the art tools. We illustrate our specification method and synthesis tool with the help of two case studies[1].

Major Contributions:

1. We developed a technique for the synthesis of controllers from QDDC re-

[1]DCSynth can be downloaded at [104] along with the specification files for the experiments.

quirements. This extends the past work on model checking interval temporal logic QDDC [83, 84, 22, 100, 70] with synthesis abilities.

2. We proposed a method for guided synthesis of controllers based on **soft requirements** which are met in a H-Optimal fashion. Conceptually, this enhances the Ramadge-Wonham framework with optimal controller synthesis.

3. We developed a tool *DCSynth* for guided synthesis. The tool represents and manipulates automata/controllers using BDD-based Semi-Symbolic DFA [67]. It uses eager minimization for efficient synthesis.

4. We experimentally analyzed the impact of soft requirements on the quality of the synthesized controllers using few case studies. The quality is measured for both the *guaranteed* and the *expected* case behaviour of the synthesized controller.

4.1 Supervisors and Controllers

Now we consider QDDC formulas and automata where the set of propositional variables $PV = I \cup O$ is 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.

Definition 15 (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 $\forall s \in F, \forall i \in 2^I$ $\exists o \in 2^O$ 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 thesis. Also, for any input sequence $ii \in (2^I)^*$, we will define $M[ii] = \{oo \mid (ii,oo) \in L(M)\}$.

We now define *cascade product* of two Mealy Machines, where the output of first Mealy Machine can be fed as input to second Mealy Machine but not vice versa (i.e. feedback is not possible).

Definition 16 (Cascade Product of Mealy Machines)**.** *Given two output- nondeterministic Mealy Machines $M_1 = (Q_1, \Sigma_1, s_1, \delta_1, F_1)$ and $M_2 = (Q_2, \Sigma_2, s_2, \delta_2, F_2)$, over input-output alphabets $\Sigma_1 = 2^{I_1} \times 2^{O_1}$ and $\Sigma_2 = 2^{I_2} \times 2^{O_2}$, and unique reject states r_1 and r_2 respectively i.e. the set of input-output variables of M_1 are (I_1, O_1) and input-output variables of M_2 are (I_2, O_2). Moreover, we have a restriction that $I_1 \cap O_2 = \emptyset$. This is because output of M_2 cannot be fed back to M_1. We define the cascade product of M_1 and M_2, as a Melay Machine $M = (Q, \Sigma, s, \delta, F)$ over*

input-output alphabets $\Sigma = 2^{I_1 \cup (I_2 - O_1)} \times 2^{O_1 \cup O_2}$*, denoted as* $M = M_1 \gg M_2$.
Where, $s = (s_1, s_2)$, $Q = \{(q_1, q_2) \mid q_1 \in F_1$ *and* $q_2 \in F_2\} \cup \{(r_1, r_2)\}$,
$F = \{(f_1, f_2) \mid f_1 \in F_1$ *and* $f_2 \in F_2\}$, $r = (r_1, r_2)$.
The transition relation $\delta : Q \times 2^{I_1 \cup (I_2 - O_1)} \times 2^{O_1 \cup O_2} \rightarrow Q$ *is defined as follows.*
$\delta((q_1, q_2), (i_1, i_2), (o_1, o_2)) = (q_1', q_2')$ *iff* $q_1, q_1' \in F_1$, $q_2, q_2' \in F_2$, $i_1 \in I_1$, $i_2 \in (I_2 - O_1)$, $o_1 \in O_1$, $o_2 \in O_2$, $\delta_1(q_1, i_1, o_1) = q_1'$ *and* $\delta_2(q_2, i_2, o_2) = q_2'$. *Otherwise,* $\delta((q_1, q_2), (i_1, i_2), (o_1, o_2)) = (r_1, r_2)$.

It may be noted, that the cascade product of M_1 and M_2 can be obtained by taking standard synchronous product of M_1 and M_2 followed by minimization of resulting automaton.

Controlling a Mealy Machine Our aim is to control an output-nondeterministic Mealy Machine by selecting some good outputs from the available non-deterministic choices using a policy (See Definition 22), in order to ensure that given requirements on input-output behaviour are met. In general a policy can select at every step t a set of outputs from the available output choices. At each step, a deterministic policy (controller) chooses one output which it considers the best. A nondeterministic policy (supervisor) chooses a subset of outputs, all of which it considers equally good. In general, the policy may be history-dependant, needing unbounded memory, but a finite memory policy can be represented as a supervisor. In particular, a memoryless policy just chooses the set of outputs based on the current state and the current input irrespective of the past history.

Definition 17 (Controllers and Supervisors)**.** *An output-nondeterministic Mealy machine which is non-blocking is called a* **supervisor**. *A deterministic supervisor is called a* **controller**. □

It may be noted that a supervisor is in effect an input total non-deterministic Mealy machine. In this thesis, the supervisor is defined as DFA recogniser to exploit the available DFA tools.

The non-deterministic choice of outputs in a supervisor denotes unresolved decision. The intuition is that, in the controlled system, any of the non-determinstic set of outputs allowed by the policy can occur adverserially and must be tolerated. The determinism ordering below allows supervisors to be refined into controllers.

Definition 18 (Determinism Order and Sub-supervisor)**.** *Given two supervisors* S_1, S_2 *we say that* S_2 *is* more deterministic *than* S_1*, denoted* $S_1 \leq_{det} S_2$*, iff* $L(S_2) \subseteq L(S_1)$*. We call* S_2 *to be a* sub-supervisor *of* S_1*.* □

Note that being supervisors, they are both non-blocking, and hence $\emptyset \subset S_2[ii] \subseteq S_1[ii]$ for any $ii \in (2^I)^*$. The supervisor S_2 may make use of additional memory for resolving and pruning the non-determinism in S_1.

For technical convenience, we define the notion of *indicator variable* for a QDDC formula (regular property). The idea is that the indicator variable of a QDDC formula D, witnesses the past satisfaction of D at any point in execution. Thus, the indicator variable is set to $true$ exactly on those points in the behaviour where D is satisfied. The definition is given as follows.

Definition 19. *A propositional variable* w *is called the indicator variable for a QDDC formula* D *denoted by* $Ind(D,w)$ *iff* $pref(EP(w) \Leftrightarrow D)$ *i.e.* $Ind(D,w) = pref(EP(w) \Leftrightarrow D)$*. Here,* $\mathbf{EP}(\mathbf{w}) = (true^\frown\langle w\rangle)$*, i.e.* $EP(w)$ *holds at a point if variable* w *is true at that point.* □

4.2 DCSynth Specification and Controller Synthesis

This section defines the DCSynth specification and presents the algorithm used in our tool DCSynth for soft requirement guided controller synthesis from a DC-Synth specification. The process of synthesizing a controller as discussed in Section 4.2.4 uses three main algorithms given in Sections 4.2.1-4.2.3.

4.2.1 Invariance Properties and Maximally Permissive Supervisor

A QDDC formula D specifies a regular property which may hold intermittently during a behaviour (see Definition 1). An important class of properties, denoted by **inv** D, states that D must hold invariantly during the system behaviour.

Definition 20. *Let* **S realizes inv D** *denote that a supervisor S realizes* invariance *of QDDC formula D over variables* (I, O). *Define* **S realizes inv D** *provided* $L(S) \subseteq L(D)$. *Recall that, by the definition of supervisors, S must be non-blocking. A supervisor S for a formula D is called* **maximally permissive** *iff* $S \leq_{det} S'$ *holds for any supervisor* S' *such that* S' ***realizes*** **inv** *D. This S (when it exists) is unique up to language equivalence of automata, and the minimum state maximally permissive supervisor is denoted as* $\boldsymbol{MPS}(D)$. □

Now, we discuss how the supervisor $MPS(D)$ for a given QDDC formula D is computed.

1. Language equivalent DFA $\mathscr{A}(D) = \langle S, 2^{I \cup O}, s, \delta, F \rangle$ is constructed for formula D (Theorem 3). The standard safety synthesis algorithm [52] over $\mathscr{A}(D)$ gives us the desired $MPS(D)$ as outlined in the following steps.

2. We first compute the *largest* set of winning states $G \subseteq F$ with the following property: $s \in G$ iff $\forall i \exists o : \delta(s,(i,o)) \in G$. Let $Cpre(\mathscr{A}(D),X) = \{s \in X \mid \forall i \exists o : \delta(s,(i,o)) \in X\}$. It may be noted that by definition *Cpre* and hence G retains only states which are non-blocking. Then we iteratively compute G as follows:

   G=F;

   do

   G1=G;

   G=Cpre($\mathscr{A}(D)$,G1);

   while (G != G1);

3. If initial state $s \notin G$, then the specification is *unrealizable*. Otherwise, $MPS(D)$ is obtained by declaring G as the set of final states and retaining all the transitions in $\mathscr{A}(D)$ between states in G and redirecting the remaining transitions of $\mathscr{A}(D)$ to a unique reject state r which is made a sink state.

Proposition 21. *For a given QDDC formula D the above algorithm computes the maximally permissive supervisor $MPS(D)$ if it exists.* □

The proposition follows straightforwardly by combining Theorem 3 with the correctness of standard safety synthesis algorithm [52]. We omit a detailed proof.

4.2.2 Maximally Permissive H-Optimal Supervisor (MPHOS)

Given a supervisor S and a desired soft requirement QDDC formula D which should hold "as much as possible" (both are over input-output propositions (I,O)), we give a method for constructing an "H-Optimal" sub-supervisor of S, which

maximizes the expected value of count of D holding in next H moves when averaged over all the inputs.

Recall the definition of an indicator variable for a QDDC formula as given in Definition 19. We first consider $\mathscr{A}(Ind(D,w))$ which is a deterministic supervisor over input-output propositions $(I \cup O, \{w\})$, such that w is *true* at any position p iff the past behaviour of variables $(I \cup O)$ up to position p (i.e. in interval $[0,p]$) satisfies D. Now we define an *Arena* automaton $\mathscr{A}^{Arena} = S \gg \mathscr{A}(Ind(D,w))$, which is a supervisor over input-output variables $(I, O \cup \{w\})$ (See Definition 16 for cascade product $\gg$ of two Mealy Machines). It augments S by producing an additional output w which witnesses the truth of D. It has the property: $L(\mathscr{A}^{Arena}) \downarrow (I \cup O) = L(S)$. Also for $\sigma \in L(\mathscr{A}^{Arena})$ and $i \in dom(\sigma)$ we have $w \in \sigma[i]$ iff $\sigma, [0,i] \models D$. Thus, every transition of $\mathscr{A}^{Arena}$ is labelled with w iff D holds on taking the transition. Let the weight of transitions labelled with w be 1 and 0 otherwise. Thus, for $o \in 2^{(O \cup \{w\})}$ let $wt(o) = 1$ if $w \in o$ and 0 otherwise. Technically, this makes $\mathscr{A}^{Arena}$ a weighted automaton.

In the supervisor $\mathscr{A}^{Arena} = (Q, \Sigma, s, \delta, Q - \{r\})$, where r is the unique reject state, we define for $(q \in Q) \neq r$ and $i \in 2^I$, set $LegalOutputs(q,i) = \{o \mid \delta(q,i,o) \neq r\}$. A deterministic selection rule for $\mathscr{A}^{Arena}$ is a function f such that $f(q,i) \in LegalOutputs(q,i)$ and a non-deterministic selection rule F for $\mathscr{A}^{Arena}$ is a function F such that $F(q,i) \subseteq LegalOutputs(q,i)$ with $F(q,i) \neq \emptyset$.

Definition 22 (Policy)**.** *A policy* Π *is an infinite sequence* $F_1, F_2, F_3 \ldots$ *of non-deterministic selection rules. A deterministic policy will use only deterministic selection rules. A policy is stationary (memory-less) if* $F_m = F_n \ \forall \ m,n$*. For a stationary policy* Π*, we will denote the selection function for state s and input i by* $\Pi(s,i)$ *such that* $\Pi(s,i) = F_1(s,i)$ *.* □

Given an arena $\mathscr{A}^{Arena}$, a state s, a policy $\Pi = F_1, F_2, \ldots$ and an input sequence $ii \in (2^I)^+$, we define the language $L(\mathscr{A}^{Arena}, ii, s)$ as all runs of $\mathscr{A}^{Arena}$ over the input ii starting from state s. Moreover, we define the language under policy Π, denoted by $L^{\Pi}(\mathscr{A}^{Arena}, ii, s)$, as all runs over input ii starting from s and following the selection rule F_k at step k. Each run in L or L^{Π} has the form (ii, oo). Note that if Π is deterministic then $L^{\Pi}(\mathscr{A}^{Arena}, ii, s)$ is a singleton set admitting a unique run (ii, oo). However, for non-deterministic policies there may be several runs on the same input ii. Let

$$Value(ii, oo) = \Sigma_{1 \leq k \leq \#ii} \ wt(oo[k])). \tag{4.1}$$

where $\#ii$ gives the length of ii. Thus, $Value(ii, oo)$ gives the count of D holding during the behaviour fragment (ii, oo) of $\mathscr{A}^{Arena}$.

H-Optimal Deterministic Policy Given an arena $\mathscr{A}^{Arena}$, a deterministic policy π and a natural number H (called Horizon), we define the following utility value for each state s of $\mathscr{A}^{Arena}$.

$$ValAvg^{\pi}(s, H) \ = \ \mathbb{E}_{ii \in (2^I)^H} \ \{Value(ii, oo) \ \mid \ (ii, oo) \in L^{\pi}(\mathscr{A}^{Arena}, ii, s)\}$$

Note that, given a probability distribution $pr(ii)$ over the set of inputs $(2^I)^H$ of length H, the expected value of $g(ii)$ is defined as usual as

$$\mathbb{E}_{ii \in (2^I)^H} \ g(ii) \ \ = \ \ \Sigma_{ii \in (2^I)^H} \ (pr(ii) \times g(ii))$$

Here, we have assumed that all input sequences ii have equal probability, i.e. the inputs occur in i.i.d. (independent and identically distributed) fashion at every state. Hence, $\mathbb{E}_{ii \in (2^I)^H} \ g(ii)$ operator computes the average of values returned by function g over all the inputs $ii \in (2^I)^H$ of length H.

Definition 23 (Value of the Deterministic Policy)**.** *Given a deterministic policy* $\pi = f_1, f_2, f_3 \ldots$ *and a horizon H, the value of the policy in a state s is given by* $Val^{\pi}(s,H)$ *below.*

$$
\begin{aligned}
&Val^{\pi}(s,0) = 0 \\
&Val^{\pi}(s,H) = \mathbb{E}_{i \in 2^I} \ \{wt(f_1(s,i)) \ + \ Val^{\pi'}(s', (H-1))\} \\
&\qquad\qquad where, \ s' = \delta(s,(i,f_1(s,i))) \\
&\qquad\qquad and, \pi' = f_2, f_3 \ldots
\end{aligned}
$$

□

It is easy to see that $ValAvg^{\pi}(s,H) = Val^{\pi}(s,H)$. This is a well known routine proof and we omit it here (see [90]).

Definition 24 (H-Optimal Deterministic Policy)**.** *For a given natural number (horizon) H, a deterministic policy* π' *is called H-Optimal if for any other deterministic policy* π *we have,*

$$Val^{\pi'}(s,H) \geq Val^{\pi}(s,H) \qquad \forall s, \pi.$$

□

We now explore the construction of such an H-Optimal policy. This will turn out to be a stationary policy. We define a recursive operator to compute optimal utility value of a state s for a given horizon H, which is denoted by $ValOpt(s,H)$. Intuitively it gives the maximal achievable value under any policy π and given horiozn H.

Definition 25 (H-Optimal Utility Value)**.** *Let*

$$
\begin{aligned}
&ValOpt(s,0) = 0 \\
&ValOpt(s,H) = \mathbb{E}_{i \in 2^I} \ max_{o \in 2^{(O \cup \{w\})} \ : \ \delta(s,(i,o)) \neq r} \\
&\qquad\qquad \{wt(o) \ + \ ValOpt(\delta(s,(i,o)), (H-1))\}
\end{aligned}
$$

□

Definition 26. *For a given horizon H, We define a deterministic stationary policy* π^* *as follows:*

$$\pi^*(s,i) \in \underset{o \in 2^{(O \cup \{w\})} : \delta(s,(i,o)) \neq r}{argmax} \{wt(o) + ValOpt(\delta(s,(i,o)),(H-1))\} \qquad \square$$

It may be noted that argmax may return a set of outputs which are equally optimal. In $\pi^*(s,i)$ *we choose any one of them arbitrarily.*

Theorem 27. *The policy* π^* *is H-Optimal i.e.*

$$Val^{\pi^*}(s,H) \geq Val^{\pi}(s,H) \qquad \forall s, \pi.$$

Proof. Let, π be any policy. Proof is done by induction on H.

Base Step (H=0):

$Val^{\pi^*}(s,0) = 0 = Val^{\pi}(s,0)$ for any state s.

Induction Step:

Assume that $Val^{\pi^*}(s,H-1) \geq Val^{\pi}(s,H-1)$ for any policy π and state s.

Then, by Definition 23, 26 and Induction hypothesis, we have

$$wt(\pi^*(s,i)) + Val^{\pi^*}(\delta(s,(i,\pi^*(s,i))),(H-1)) \geq$$
$$wt(f_1(s,i)) + Val^{\pi'}(\delta(s,(i,f_1(s,i))),(H-1)) \text{ for any policy } \pi$$
$$\text{where } \pi = f_1, f_2 \ldots \text{ and } \pi' = f_2, f_3 \ldots.$$

Hence, $Val^{\pi^*}(s,H)$

$$= \mathbb{E}_{i \in 2^I} \{wt(\pi^*(s,i)) + Val^{\pi^*}(\delta(s,(i,\pi^*(s,i))),(H-1))\}$$
$$\geq \mathbb{E}_{i \in 2^I} \{wt(f_1(s,i)) + Val^{\pi'}(\delta(s,(i,f_1(s,i))),(H-1))\}$$
$$= Val^{\pi}(s,H) \qquad \square$$

H-Optimal Non-Deterministic Policy Given an arena $\mathscr{A}^{Arena}$, a non-deterministic policy Π and a natural number H (called Horizon), we define the following utility

value for each state s of $\mathscr{A}^{Arena}$. Recall that for nondeterministic policies, for an input ii the set of behaviours $L^{\Pi}(\mathscr{A}^{Arena}, ii, s)$ may contain one or more elements depending on nondeterminstic choice of output. The policy has no control over which of these outputs occurs, and in worst case outputs can occur adversarially. Hence, we compute the value of a nondeterministic policy by taking worst case view as, $min\ \{Value(ii,oo)\ \mid\ (ii,oo) \in L^{\Pi}(\mathscr{A}^{Arena}, ii, s)\}$. Thus, we define

$$ValAvgMin^{\Pi}(s,H)\ =\ \mathbb{E}_{ii\in(2^I)^H}\ min\ \{Value(ii,oo)\ \mid\ (ii,oo) \in L^{\Pi}(\mathscr{A}^{Arena}, ii, s)\}$$

Definition 28 (Value of the Non-Deterministic Policy)**.** *Given a non-deterministic policy $\Pi = F_1, F_2, F_3 \ldots$ and a horizon H, the value of a policy in a state s is given by $Val^{\Pi}(s,H)$ below.*

$$\begin{aligned}
&ValNonDet^{\Pi}(s,0) = 0\\
&ValNonDet^{\Pi}(s,H) = \mathbb{E}_{i\in 2^I}\ \ min_{o\in F_1(s,i)}\ \{wt(o)\ +\ ValNonDet^{\Pi'}(s',(H-1))\}\\
&\qquad\qquad where,\ s' = \delta(s,(i,o))\\
&\qquad\qquad and, \Pi' = F_2, F_3 \ldots
\end{aligned}$$ □

It is easy to see that $ValAvgMin^{\Pi}(s,H) = ValNonDet^{\Pi}(s,H)$. The proof is similar to that of deterministic policies (see [90]).

Definition 29 (Non-Deterministic H-Optimal Policy)**.** *For a given natural number (horizon) H, a non-deterministic policy Π' is called H-Optimal if for any other non-deterministic policy Π we have,*

$$ValNonDet^{\Pi'}(s,H) \geq ValNonDet^{\Pi}(s,H) \qquad \forall s, \Pi.$$ □

We now explore the construction of such non-deterministic H-Optimal policy. This will turn out to be a stationary policy. We use the recursive operator $ValOpt(s,H)$

given earlier in Definition 25 to compute the optimal utility value of a state s and horizon value H. Intuitively it also gives the maximal achievable value under any non-deterministic policy Π.

Definition 30. *We define a non-deterministic stationary policy* Π^* *as follows:*

$$\Pi^*(s,i) = argmax_{o \in 2^{(O \cup \{w\})} : \delta(s,(i,o)) \neq r} \{wt(o) + ValOpt(\delta(s,(i,o)),(H-1))\} \quad \square$$

It may be noted that argmax returns a set of outputs which are equally optimal. In $\Pi^*(s,i)$ *we keep all the optimal choices.*

Theorem 31. *The policy* Π^* *is H-Optimal i.e.*

$$ValNonDet^{\Pi^*}(s,H) \geq ValNonDet^{\Pi}(s,H) \qquad \forall s, \Pi.$$

Proof. Let, Π be any policy. Proof is done by induction on H.

Base Step (H=0):

$ValNonDet^{\Pi^*}(s,0) = 0 = ValNonDet^{\Pi}(s,0)$ for any state s.

Induction Step:

Assume that $ValNonDet^{\Pi^*}(s,H-1) \geq ValNonDet^{\Pi}(s,H-1)$ for any policy Π and state s.

Then, by Definition 28, 30 and Induction hypothesis, we have

$$min_{o \in \Pi^*(s,i)} \{wt(o) + ValNonDet^{\Pi^*}(\delta(s,(i,o)),(H-1))\} \geq$$
$$min_{o \in \mathbf{F}_1(s,i)} \{wt(o) + ValNonDet^{\Pi'}(\delta(s,(i,o)),(H-1))\} \text{ for any policy } \Pi.$$
$$\text{where } \Pi = F_1, F_2 \ldots \text{ and } \Pi' = F_2, F_3 \ldots.$$

Hence, $ValNonDet^{\Pi^*}(s,H)$

$$= \mathbb{E}_{i \in 2^I} \{ min_{o \in \Pi^*(s,i)} \{wt(o) + ValNonDet^{\Pi^*}(\delta(s,(i,o)),(H-1))\}\}$$
$$\geq \mathbb{E}_{i \in 2^I} \{ min_{o \in \mathbf{F}_1(s,i)} \{wt(o) + ValNonDet^{\Pi'}(\delta(s,(i,o)),(H-1))\}\}$$
$$= ValNonDet^{\Pi}(s,H) \quad \square$$

Obtaining the H-Optimal supervisor We prune the supervisor $\mathscr{A}^{Arena}$ to retain only the transitions with the outputs in set $\Pi^*(s,i)$ (as these are all equally optimal). This gives us *Maximally permissive H-Optimal sub-supervisor of* $\mathscr{A}^{Arena}$ *w.r.t. D*. This supervisor is denoted by $MPHOS(\mathscr{A}^{Arena},H)$ or equivalently $MPHOS(S,D,H)$. The following proposition follows immediately from the construction of $MPHOS(S,D,H)$.

Proposition 32. *For any supervisor S and a QDDC formula D the following relations hold.*

1. $S \leq_{det} MPHOS(S,D,H)$*, for all* H.
2. $MPHOS(S,D,H)$ *is maximally permissive H-Optimal sub-supervisor of S.*
3. *If* $MPHOS(S,D,H) \leq_{det} S'$ *then* S' *is H-Optimal.*

Proof.

1. The construction of $MPHOS(S,D,H)$ starts by constructing $\mathscr{A}^{Arena}$. It has a property that $L(\mathscr{A}^{Arena}) \downarrow (I \cup O) = L(S)$. Therefore $L(\mathscr{A}^{Arena})$ accepts those words, which are accepted by $L(S)$. To obtain $MPHOS(S,D,H)$ we prune some of the non-deterministic choices in $\mathscr{A}^{Arena}$. Therefore, $L(MPHOS) \subseteq L(\mathscr{A}^{Arena})$, hence $S \leq_{det} MPHOS(S,D,H)$.

2. By Theorem 31, $MPHOS(S,D,H)$ is H-Optimal. Moreover, from Definition 30, after assigning H-Optimal utility values to each state, we obtain $MPHOS$ by pruning only those non-deterministic output choices which does not go the optimal utility state. Thus, the $MPHOS$ will contain all H-Optimal transitions and will produce maximally permissive H-Optimal supervisor.

3. The proof for this follows from the construction of $MPHOS$. As $MPHOS$ retains all the transitions which are H-Optimal, so any pruning of non-deterministic choices of outputs in $MPHOS$ will preserve the H-Optimality.

□

4.2.3 From Supervisor to Controller

A controller *Cnt* can be obtained from a supervisor S by resolving output non-determinism in S. We give a rather straightforward mechanism for this. We allow the user to specify a total ordering *Ord* on the set of output variables 2^O. A given supervisor S is determinized by retaining only the highest ordered output among those permitted by S. This is denoted $Det_{Ord}(S)$. The output ordering is specified by giving a lexicographically ordered list of output variable literals. This facility is used to determinize $MPHOS$ and MPS supervisors as required.

Example 33. *For a supervisor S over variables* $(I, \{o_1, o_2\})$*, an example output order can be given as lexicographically ordered list* $(o_1 > \ !o_2)$*. Then, for any transition the determinization step will try to select the highest ordered output (which is allowed by S) from the list* $\{(o_1 = true, o_2 = false)$*,* $(o_1 = true, o_2 = true)$*,* $(o_1 = false, o_2 = false)$*,* $(o_1 = false, o_2 = true)\}$.

4.2.4 Controller Synthesis Algorithm

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 $PV = I \cup O$. Let H be a natural number called Horizon. The objec-

ControllerSynthesis

Input: $S = (I, O, D^h, D^s)$. Horizon H, Output ordering *Ord*

Output: Controller Cnt for S.

1. $\mathscr{A}^{mon} = \mathscr{A}(D^h)$ `/*Monitor automaton for` D^h`*/`
2. $\mathscr{A}^{MPS} = MPS(\mathscr{A}^{mon})$

   `/* If` $\mathscr{A}^{MPS}$ `does not exist then return UNREALIZABLE */`
3. $\mathscr{A}^{MPHOS} = MPHOS(\mathscr{A}^{MPS}, D^s, H)$
4. $Cnt = Det_{ord}(\mathscr{A}^{MPHOS})$.
5. Encode the automaton *Cnt* in an implementation language.

Figure 4.1: Controller Synthesis Algorithm

tive in DCSynth is to synthesize a deterministic controller which (a) *invariantly* satisfies the hard requirement D^h, and (b) is H-Optimal w.r.t. D^s amongst all the controllers satisfying (a).

Given a specification (I, O, D^h, D^s), a horizon value H (a natural number) and a total ordering *Ord* on the set of outputs 2^O, the controller synthesis in DCSynth can be given as Algorithm in Figure 4.1.

Step 1 and 2 are described in MPS construction given in Section 4.2.1. Step 3 uses the MPHOS construction given in Section 4.2.2, whereas Step 4 uses the determinization method of Section 4.2.3.

Theorem 34. *The controller Cnt produced by Algorithm given in Figure 4.1 invariantly satisfies D^h, and it intermittently, but H-Optimally, satisfies D^s.*

Proof. By Proposition 21, $\mathscr{A}^{MPS}$ realizes **inv** D^h. Then, by Proposition 32, $\mathscr{A}^{MPHOS}$ and *Cnt* are sub-supervisors of $\mathscr{A}^{MPS}$ and hence they also realize **inv** D^h. More-

over, by Theorem 31, we get that $\mathscr{A}^{MPHOS}$ is H-Optimal w.r.t. D^s. Hence, by Proposition 32, we get that *Cnt* which is a sub-supervisor of $\mathscr{A}^{MPHOS}$ is also H-Optimal with respect to D^s. □

At all stages of above synthesis, the automata/supervisors $\mathscr{A}(D^h)$, $\mathscr{A}(D^s)$, $\mathscr{A}^{MPS}$ and $\mathscr{A}^{MPHOS}$ and *Cnt* are all represented as Semi-Symbolic DFA (SSDFA) using the MONA [66] DFA data structure. In this representation, the transition function is represented as a multi-terminal BDD. MONA DFA library provides a rich set of automata operations including product, projection, determinization and minimization over the SSDFA. The algorithms discussed in Sections 4.2.1, 4.2.2 and 4.2.3 are implemented over SSDFA. Section 4.4 gives the details of this implementation. Moreover, these algorithms are adapted to work without actually expanding the specification automata into game graph. At each stage of computation, the automata and supervisors are aggressively minimized, which leads to significant improvement in the scalability and computation time.

Generalized DCSynth specification: In general, the tool DCSynth supports specification of soft requirements as an ordered list of formulas with user defined weights. The *Generalized 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, O and D^h have the same meaning. However, the *soft requirement* $\langle D^s_1 : \theta_1, \cdots, D^s_k : \theta_k \rangle$ is now 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. Then, the weight (reward) of a transition is defined as the *sum of weights* of each of the formula D^s_i which holds on taking the transition. The tool DCSynth produces a MPHOS, 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.

Section 4.4 gives the details of symbolic computation of supervisors and controllers using SSDFA. The use of SSDFA leads to significant improvement in the scalability and computation time of the tool.

4.3 Case Studies and Experiments

For a DCSynth specification (I, O, D^h, D^s), D^h and D^s can be any QDDC formulas. While invariance of D^h is guaranteed by the synthesis algorithm, the quality of the controller is controlled by optimizing the outputs for which the soft requirement D^s holds. For example, D^s may specify outputs which save energy, giving an energy efficient controller. The soft requirement can also be used to improve the robustness [12] of the controller, which is discussed in Chapter 5 of this thesis. Below, we consider specifications structured as assumptions and commitments, and their optimized robustness using our soft requirement guided synthesis.

4.3.1 Types of Controller Specification

For many examples, the controller specification can be given as a pair (A, C) of QDDC formulas over input-output variables (I, O). Here, **commitment** C is a formula specifying the desired behaviour which must ideally hold invariantly. But this may be unrealizable, and a suitable **assumption** A on the behaviour of environment may have to be made for C to hold. In case the assumption A does not hold, it is still desirable that controller satisfies C intermittently but "as much as possible". In the next Section 4.3.2, we give a method to specify intermittent requirements, which is useful in many situations. Given this assumption-commitment pair (A, C), we specify four types of derived controller specifications

(I, O, D^h, D^s) as follows.

Type	**Hard Requirement D^h**	**Soft Requirement D^s**
Type0	C	$true$
Type1	$(A \Rightarrow C)$	$true$
Type2	$true$	C
Type3	$(A \Rightarrow C)$	C

Type0 controller gives the best guarantee but it may be unrealizable. *Type1* controller provides a firm but conditional guarantee. *Type2* controller tries to achieve C in H-Optimal fashion irrespective of any assumption and *Type3* Controller provides firm conditional guarantee and it also tries to satisfy C in H-Optimal fashion even when the assumption does not hold.

4.3.2 Invariant v/s Intermittent requirement Specification

In our synthesis framework, QDDC is used to specify interval based as well as point based past time temporal requirements of systems. These requirements are intended to specify assumptions and commitment which may hold intermittently during execution. The *Assumption* and *Commitment* based requirement specification and *Soft requirements*, as discussed in previous section typically are intermittent in nature. Intermittent specification is useful in specifying recoverable behaviours for controller synthesis. The concept of H-Optimality also makes sense when the requirements are intermittent. In following sections, we illustrate the concept of *invariant* and *intermittent* requirements, and methods to specify intermittent requirements in our framework.

Invariant and Prefix Closed Requirement

The satisfaction of an Invariant or Prefix Closed requirement at any position i depends on full past behaviour. Following specifications describe such requirements.

- Given a QDDC formula D, we can specify $[]D$ as full past property which states that all intervals in the past invariantly satisfy D.
- Another way of specifying invariant requirement is $[[P]]$ where P is a proposition. This requirement states that P should be satisfied invariantly at every position in past including the current position.
- Similarly, we can also define $pref(D)$ as full past requirement which states that all prefix intervals in the past satisfy D.

It may be noted that these are prefix closed requirements and hence once these become false at a position i they can never be recovered for any position $j > i$. Typically, for specification of a controllers we would need requirements to be recoverable, which is described in the following paragraph.

Intermittent and Bounded Past Requirements

In general the systems under consideration have bounded memory and the temporal requirements are stated over bounded interval. Bounded past requirements are those properties for which the satisfaction depends only on the last n cycle interval, where n is an integer constant. Following requirements describe such properties.

– Given a QDDC formula D, we can specify $true\hat{}D$ as suffix property which is satisfied if a suffix intervals satisfy D. But full past behaviour is also a suffix interval and hence this property may also become non-intermittent.

– Another way of defining intermittent property is $true\hat{} < P >$ where P is a proposition, which states that P should be satisfied at current position in any behaviour.

– We now give a generic way to derive an intermittent specification from a given QDDC formula D and an integer bound K (which specifies how many past cycles are required to evaluate the satisfaction of D). We define, $KBounded(D,K) = ((slen < K) => D) \;\&\&\; (true\ \hat{}\ (slen = K) => true\ \hat{}\ ((slen = K)\ \&\&\ D))$.
For any QDDC formula D, $\sigma, i \models KBounded(D,n)$ iff (a) $i \leq n$ and $\sigma, i \models D$, OR (b) $i > n$ and $\sigma, [(i-n), i] \models D$.

As the satisfaction of these kind of specification depends only on the bounded past (i.e. looks at only bounded suffix interval), they give rise to intermittent (recoverable) requirements. In our case studies we would prefer to use these kind of specifications.

4.3.3 Performance Metrics: Measuring quality of controllers

For the same assumption commitment pair (A,C), we can synthesize diverse controllers using different specification types, horizon values and output orderings. In order to compare the performance of these different controllers, we define two metrics – i) *Expected Case Performance measure* to compare average case behaviour, and ii) *Must Dominance* to compare the guaranteed behaviour.

i) Expected Case Performance: Given a controller *Cnt* over input-output alphabet (I, O) and a QDDC formula (regular property) C over variables $I \cup O$, we can construct a *Discrete Time Markov Chain (DTMC)*, denoted $M_{unif}(Cnt, C)$, whose analysis allows us to measure the probability of C holding in long runs (steady state) of *Cnt* under random independent and identically distributed (iid) inputs. This value is designated as $\mathbb{E}_{unif}(Cnt, C)$. The construction of the desired DTMC is as follows. The product $Cnt \times \mathscr{A}(C)$ gives a finite state automaton with the same behaviours as *Cnt*. Moreover, it is in accepting state exactly when C holds for the past behaviour. (Here $\mathscr{A}(C)$ works as a total deterministic monitor automaton for C without restricting *Cnt*). By assigning uniform discrete probabilities to all the inputs from any state, we obtain the DTMC $M_{unif}(Cnt, C)$ along with a designated set of accepting states, such that the DTMC is in accepting state precisely when C 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 $\mathbb{E}_{unif}(Cnt, C)$. A leading probabilistic model checking tool MRMC implements this computation [65]. In DCSynth, we provide a facility to compute $M_{unif}(Cnt, C)$ in a format accepted by the tool MRMC. Hence, using DCSynth and MRMC, we are able to compute $\mathbb{E}_{unif}(Cnt, C)$.

ii) Guaranteed Performance as Must-Dominance: Consider two supervisors S_1, S_2 and a regular property C. Define that S_i guarantees C for an input sequence ii, provided for every output sequence $oo \in S_i[ii]$ produced by S_i on ii we have that (ii, oo) satisfies C. We say that S_2 *must dominates* S_1 with respect to the property C provided for every input sequence ii, if S_1 guarantees C then S_2 also guarantees C. Thus, S_2 provides a superior must guarantee of C than S_1.

Definition 35 (Must Dominance)**.** *Given two supervisors S_1, S_2 and a property (formula) C over input-output alphabet (I,O), the must dominance of S_2 over S_1 is defined as $S_1 \leq^C_{dom} S_2$ iff $MustInp(S_1,C) \subseteq MustInp(S_2,C)$, where $MustInp(S_i,C) = \{ii \in (2^I)^+ \mid \forall oo \in (2^O)^+.((ii,oo) \in L(S_i) \Rightarrow (ii,oo) \models C\}$.* □

The following proposition states that any arbitrary resolution of non-determinism in supervisors preserves the must-guarantees.

Proposition 36. *$S_1 \leq_{det} S_2$ implies that $S_1 \leq^D_{must} S_2$ for any $C \in QDDC$.* □

We establish *must dominance* relations among MPHOS supervisors of various types of specifications discussed in Section 4.3.1.

Lemma 37. *For any QDDC formulas A and C, and any horizon H, the following must dominance relations will hold (for any given H)*

1. $MPHOS_1(A,C)) \leq^C_{dom} MPHOS_3(A,C)) \leq^C_{dom} MPHOS_0(A,C))$
2. $MPHOS_2(A,C)) \leq^C_{dom} MPHOS_0(A,C))$

where, $MPHOS_i(A,C)$ denote the maximally permissive H-Optimal supervisor $\mathscr{A}^{MPHOS}$ of Synthesis Algorithm given in Figure 4.1 for the specification $Type_i(A,C)$.

Proof. By definition, $MPHOS_0(A,C)$ invariantly satisfies C for all input sequences. Hence, $MustInp(MPHOS_0(A,C),C) = (2^I)^*$, which immediately gives us that $S \leq^C_{dom} MPHOS_0(A,C))$ for any supervisor S.

Now we prove the remaining relation $MPHOS_1(A,C)) \leq^C_{dom} MPHOS_3(A,C))$. Let $S = MPS(A \Rightarrow C)$. Then, $MPHOS_1(A,C)) = MPHOS(S,true,H) = S$. The second equality holds as soft requirement $true$ does not cause any pruning of outputs in H-Optimal computation. By definition $MPHOS_3(A,C) = MPHOS(S,C,H)$.

By Proposition 32, $S \leq_{det} MPHOS(S,C,H)$ which gives us the required result. □

Note that in general, $MPHOS_2(A,C)$ is theoretically not comparable with $MPHOS_1(A,C)$ and $MPHOS_3(A,C)$, as $MPHOS_2(A,C)$ is a supervisor that does not have to meet any hard requirement, but it optimally meets the soft requirements irrespective of the assumption. However, for specific (A,C) instances, some additional must-dominance relations may hold between $MPHOS_2(A,C)$ and the other supervisors.

4.3.4 Case Studies: Mine-pump and Arbiter Specifications

We have carried out experiments with i) the Mine-pump specification, and ii) an Arbiter specification.

Mine-pump:

The Mine-pump controller (see [83]) has two input sensors: high water level sensor *HH2O* and methane leakage sensor *HCH4*; and one output, *PUMPON* to keep the pump on. The objective of the controller is to *safely* operate the pump in such a way that the water level never remains high continuously for more that w cycles. Thus, Mine-pump controller specification has input and output variables $(\{HH2O, HCH4\}, \{PUMPON\})$.

We have following **assumptions** on the *mine* and the *pump*. Their conjunction is denoted by $MineAssume(\varepsilon, \zeta, \kappa)$ with integer parameters $\varepsilon, \zeta, \kappa$. Being of the form $[]D$ each formula states that the property D (described in text) holds for all observation intervals in past.

- *Pump capacity:* $([]!(slen = \varepsilon \;\&\&\; ([[PUMPON \;\&\&\; HH2O]]\hat{}\langle HH2O\rangle)))$. If the pump is continuously on for ε cycles with water level also continuously high, then water level will not be high at the $\varepsilon + 1$ cycle.

- *Methane release:* $[](([HCH4]\hat{}[!HCH4]\hat{}\langle HCH4\rangle) \Rightarrow (slen > \zeta))$ and $[]([[HCH4]] \Rightarrow slen < \kappa)$. The minimum separation between the two leaks of methane is ζ cycles and the methane leak cannot persist for more than κ cycles.

The **commitments** are as follows. The conjunction of commitments is denoted by $MineCommit(w)$ and they hold intermittently in absence of assumption.

- *Safety conditions:* $true\hat{}\langle((HCH4 \;||\; !HH2O) \Rightarrow\; !PUMPON))\rangle$ states that if there is a methane leak or absence of high water in current cycle, then pump should be off in the current cycle. Formula $!(true\hat{}([[HH2O]] \;\&\&\; slen = w))$ states that the water level does not remain continuously high in last $w+1$ cycles.

The Mine-Pump specification denoted by $MinePump(w, \varepsilon, \zeta, \kappa)$ is given by the assumption-commitment pair $(MineAssume(\varepsilon, \zeta, \kappa), MineCommit(w))$. The four types of DCSynth specifications of Section 4.3.1 can be derived from this.

Arbiter:

We now give the specification of a synchronous bus arbiter. This is an enhanced version of the specification discussed as Example 14. An n-cell synchronous bus arbiter has inputs $\{req_i\}$ and outputs $\{ack_i\}$ where $1 \leq i \leq n$. In any cycle, a subset of $\{req_i\}$ is true and the controller must set one of the corresponding ack_i to true. The **arbiter commitment**, $ArbCommit(n,k)$, is conjunction of the following properties.

$$Mutex(n) = true\hat{}\langle\ \wedge_{i\neq j}\ \neg(ack_i \wedge ack_j)\ \rangle$$
$$NoLoss(n) = true\hat{}\langle\ (\vee_i req_i) \Rightarrow (\vee_j ack_j)\ \rangle$$
$$NoSpurious(n) = true\hat{}\langle\ \wedge_i\ (ack_i \Rightarrow req_i)\ \rangle$$
$$Response(n,k) = (\wedge_{1\leq i\leq n}\ (Resp(req_i, ack_i, k)) \quad \text{where}$$
$$Resp(req, ack, k) = true\hat{}(([[req]]\ \&\&\ (slen = (k-1)))\ \Rightarrow$$
$$true\hat{}(scount\ ack > 0\ \&\&\ (slen = (k-1)))$$

The QDDC formula $true\hat{}\langle P\rangle$ holds at a point i in execution if the proposition P holds at that point. Thus, the formula $Mutex(n)$ gives mutual exclusion of acknowledgments; $NoLoss(n)$ states that if there is at least one request then there must be an acknowledgment; and $NoSpurious$(n) states that acknowledgment is only given to a requesting cell. Formula $true\hat{}(([[req]]\ \&\&\ (slen = (k-1)))$ states that in the last k cycles req is invariantly true. Similarly, the formula $true\hat{}(scount\ ack > 0\ \&\&\ (slen = (k-1)))$ states that in last k cycles the ack has been $true$ at least once. Then, the formula $Resp(req, ack, k)$ states that if req has been continuously true in last k cycles, there must be at least one ack within last k cycles. So, $Response(n,k)$ says that each cell requesting continuously for last k cycles must get an acknowledgment within last k cycles.

A controller can invariantly satisfy $ArbCommit(n,k)$ if $n \leq k$. For example, Tool DCSynth gives us a concrete controller for the commitment $D^h = ArbCommit(6,6)$. It is easy to see that there is no controller which can invariantly satisfy $ArbCommit(n,k)$ if $k < n$. To see this, consider the case when all req_i are continuously true. Then, it is not possible to give response to every cell in less than n cycles due to mutual exclusion of req_i.

To handle such desired but unrealizable requirement we make an assumption. Let the proposition $Atmost(n,i)$ be defined as $\forall S \subseteq \{1 \ldots n\}, |S| \leq i.\ \ \wedge_{j\notin S} \neg req_j$.

It states that at most i out of total n requests can be true simultaneously. Then, the **arbiter assumption** is the formula $ArbAssume(n,i) = true^{\frown}\langle Atmost(n,i)\rangle$, which states that $Atmost(n,i)$ holds at the current cycle.

The specification of the synchronous arbiter is the assumption-commitment pair $(ArbAssume(n,i), ArbCommit(n,k))$, which is denoted by $Arbiter(n,k,i)$. Here, n denotes the number of clients, k is the response time and i is the maximum number of request that can be true simultaneously.

4.3.5 Experimental Evaluation

Given an assumption-commitment pair (A,C) the four types of DCSynth specifications can be derived as given in Section 4.3.1. Given any such specification, a horizon value H, and an ordering of outputs, a controller can be synthesized using our tool DCSynth as described in Section 4.2.4. For the *Mine-pump* instance $MinePump(8,2,6,2)$, we synthesized controllers for all the four derived specification types with horizon value $H = 50$ and output ordering $PUMPON$. These controllers choose to get rid of water aggressively by keeping the pump on whenever possible. Similarly, controllers were also synthesized with the output ordering $!PUMPON$. These controllers save energy by keeping the pump off whenever possible. Note that, in our synthesis method, hard and soft requirements are fulfilled before applying the output orderings.

For the Arbiter instance $Arbiter(5,3,2)$ also, controllers were synthesized for all the four derived specification types with horizon value $H = 50$ and output ordering $ArbDef = (a_1 > a_2 > a_3 > a_4 > a_5)$. This ordering tries to give acknowledgment such that client i has priority higher than client j for all $i < j$.

In Table 4.1 we give the performance of the of tool DCSynth in synthesizing

Table 4.1: Synthesis from Mine-pump(8,2,6,2) and Arbiter(5,3,2) specifications in DCSynth. The last column gives the expected value of commitment in long run on random inputs.

	DCSynth Specification		**Synthesis (States/Time)**			
Sr No	**Controller type**	**Output Ordering**	**MPS Stats**	**MPHOS Stats**	**Controller Stats**	**Expected Value**
MinePump$(\mathbf{8},\mathbf{2},\mathbf{6},\mathbf{2})$						
1	$Type0$	-	Unrealizable			
2	$Type1$	$PUMPON$	70/0.00045	70/0.00254	21/0.00220	0.0
3	$Type2$	$PUMPON$	1/0.00004	10/0.00545	10/0.00033	0.99805
4	$Type3$	$PUMPON$	70/0.00045	75/0.044216	73/0.00081	0.99805
5	$Type1$	$!(PUMPON)$	70/0.00045	70/0.00254	47/0.00230	0.0
6	$Type2$	$!(PUMPON)$	1/0.00004	10/0.00545	10/0.00019	0.99805
7	$Type3$	$!(PUMPON)$	70/0.00045	75/0.044216	73/0.00082	0.99805
Arbiter						
1	$Type0$	-	Unrealizable			
2	$Type1$	$ArbDef$	13/0.000226	13/0.004794	11/0.007048	0.0
3	$Type2$	$ArbDef$	1/0.00001	207/1.864346	201/0.058423	0.9930985
4	$Type3$	$ArbDef$	13/0.000213	207/1.897907	201/0.057062	0.9930985

these controllers. The table gives the time taken at each stage of the synthesis algorithm, and the sizes of the computed supervisors/controllers. The experiments were conducted on Linux (Ubuntu 16.04) system with Intel i5 64 bit, 2.5 GHz processor and 4 GB memory.

Experimental Evaluation of Expected Case Performance: The last column of Table 4.1 gives the expected value of commitment holding in long run for the controllers of various types for both Mine-pump and Arbiter instances. This value is computed as outlined in Section 4.3.3. The results are quite encouraging.

It is observed that in both the case studies *Type0* controller is unrealizable. It is also evident from Table 4.1 that in both the examples, the controllers for *Type1* (i.e., when soft-requirements are not used) specifications have 0 expected value of commitment C. This is because of the strong assumptions used in guaranteeing C, which themselves have expected value 0. In such a case, whenever the assumption fails, the synthesis algorithm has no incentive to try to meet C.

On the other hand, with soft requirement C in *Type2* and *Type3* specifications, the H-Optimal controllers have the expected value of C above 99%. This remarkable increase in the *expected value* of Commitment shows that H-Optimal synthesis is very effective in figuring out controllers which meet the desirable property C as much as possible, irrespective of the assumption.

Experimental Evaluation of Must-Dominance: Given supervisors S_1, S_2 for an assumption-commitment pair (A,C), since both S_1, S_2 are finite state Mealy machines and C is a regular property, an automata theoretic technique can automatically check whether $S_1 \leq^C_{dom} S_2$. This technique is implemented in our tool DCSynth. In case $S_1 \leq^C_{dom} S_2$ does not hold, the tool provides a counter example.

For our case studies, we experimentally compare must dominance of supervisors $MPHOS_i(A,C)$ as defined in Lemma 37. Recall that $MPHOS_i(A,C)$ denotes the maximally permissive H-Optimal supervisor for the specification $Type_i(A,C)$. The results obtained (with $H = 50$) are as follows.

1. Mine-pump instance $Minepump(8,2,6,2)$ denoted by $MP(8,2,6,2)$):

$$MPHOS_1(MP(8,2,6,2)) <^C_{dom} MPHOS_3(MP(8,2,6,2)) =^C_{dom} MPHOS_2(MP(8,2,6,2))$$

2. Arbiter instance $Arb(5,3,2)$:

$$MPHOS_1(Arb(5,3,2)) <^{C}_{dom} MPHOS_2(Arb(5,3,2)) =^{C}_{dom} MPHOS_3(Arb(5,3,2))$$

$MPHOS_3$ must dominates $MPHOS_1$ as expected, as $MPHOS_3$ is a sub-supervisor of $MPHOS_1$. What is interesting and surprising is that in both the case studies Arbiter and Mine-pump, the $MPHOS_2$ and $MPHOS_3$ supervisors are found to be syntactically identical. This is not theoretically guaranteed, as $Type2$ and $Type3$ supervisors are must-incomparable in general. Thus, in these examples, the H-Optimal $MPHOS_2$ already provides all the must-guarantees of the $MPHOS_3$ hard requirements. The H-optimization of C seems to exhibit startling ability to guarantees C without human intervention of providing suitable assumption.

So far we have considered commitment as soft requirement. In general, the soft requirement can be used to optimize MPS w.r.t. any regular property of interest, whereas the hard requirements gives the necessary must guarantees. Such soft requirements may embody performance and quality goals. Hence, it is advisable to use the combination of hard and soft requirement based on the criticality of each requirement.

4.4 Implementation with Semi-Symbolic DFA

Broadly, the computation of controller goes through the steps of (a) Computing hard requirement monitor (b) Computing MPS (c) Computing MPHOS, and (d) Determinization using output preference. This is outlined in Algorithm of Figure 4.1 in Section 4.2. We now consider efficient implementation of these steps using

the semi-symbolic automaton (SSDFA) representation of tool MONA introduced in Section 2.3. We now give the detailed description of this representation, where the transition function is encoded as *multi-terminal BDD* (MTBDD).

MTBDD is a generalization of BDD in a sense that BDD is a directed acyclic graph used to encode some Boolean function $f : \mathbb{B}^l \to \mathbb{B}$ for given 'l' possible Boolean variables. MTBDD is a directed acyclic graph used to encode multi-valued function $f : \mathbb{B}^l \to \mathscr{D}$, where $\mathscr{D}$ is the range of this multi-valued function.

Now, we briefly describe the SSDFA representation of an automaton. Its alphabet is $\Sigma = 2^{IUO}$, where each path π in MTBDD represents an element of Σ. The path π ends in a terminal node which is labelled by $\delta(s,\pi)$. For example, the Figure 4.2(a) gives an explicit DFA. Its alphabet Σ is 4-bit vectors giving value of propositions $(req_1, req_2, ack_1, ack_2)$ and set of states $S = \{1,2,3,4\}$. Being a safety automaton it has a unique reject state 4 and all the missing transitions are directed to it. (State 4 and transitions to it are omitted in Figure 4.2(a) for brevity.)

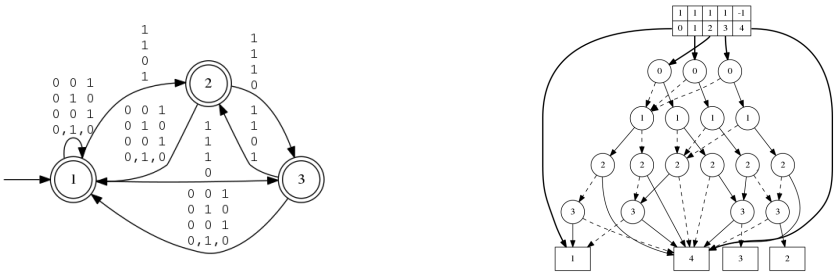


Figure 4.2: Example automaton (a): External format (b): SSDFA format

Figure 4.2(b) gives the SSDFA representation for the above automaton. Note that states numbered from 1 to 4, are explicitly listed in the array (called the *StateArray*). The final states are marked as 1 and non-final states marked as -1.

(For technical reasons there is an additional state 0 which may be ignored here and state 1 may be treated as the initial state). Each state s points to shared MTBDD node encoding the transition function $\delta(s) : \Sigma \rightarrow S$ with each path ending in the next state (represented by square node called the *terminal nodes*). We assume that the MTBDD node pointed by state s can be accessed by *s.bddNode*, which returns the pointer to the corresponding MTBDD node. The MTBDD node which are pointed by states in the automaton are called the *root nodes*. Each circular node of MTBDD represents a *decision node*, in our example the circular nodes with indices $0,1,2,3$ are the decision nodes representing variables $req_1, req_2, ack_1, ack_2$ respectively. Solid edges lead to *true* cofactors and dotted edges to *false* cofactors.

The tool MONA [55] provides a library implementing automata operations including product, complementation, projection, determinization and minimization on SSDFA format. Moreover, automata may be constructed from scratch by giving list of states and adding transitions one at a time. A default transition must be given to make the automaton total.

Remark 1: DFA in Figure 4.2 also denotes a Output-nondeterministic Mealy machine with input alphabet (req_1, req_2) and output alphabet (ack_1, ack_2). Automaton is nondeterministic in its output as $\delta(1,(1,1,1,0)) = 2$ and $\delta(1,(1,1,0,1)) = 3$. We require that *input* variables always occur before the *output* variables.

We use the MTBDD based DFA library in MONA for implementing our tool *DCSynth*. The MTBDD node data structure has been extended for efficient implementation of *DCSynth*. Most of the steps in computation of MPS and MPHOS involve recursive top-down traversal of the MTBDD structure where result at a root node is an aggregate of the nodes/paths encountered. The performance of this can be significantly improved with memoization to avoid repeated compu-

tation at sub-nodes in the DAG data-structure. Additional memory is needed at BDD nodes for storing the intermediate values at interior nodes. Following are the description of the additional fields of a BDD node say **BN** used in the tool implementation.

- **BN.variable** : Gives the variable associated with the decision node.
- **BN.state** : This field in any terminal node stores the state number of the next states in the transition.
- **BN.left** : This field stores the pointer to *false* cofactor (BDD node) of BN.
- **BN.right** : This field stores the pointer to *right* cofactor (BDD node) of BN.
- **BN.visited** : This is also a boolean value to check whether the BN is already explored. This will be used for memoization in MPS and MPHOS computation.
- **BN.status**: This returns a boolean value. Used for obtaining MPS during computation of winning region.
- **BN.optValue** : This field stores a floating point value used for optimal utility value calculation during MPHOS construction.
- **BN.optOutputList** : This field stores the list of all optimal output paths and next state pair from BDD node BN.

4.4.1 Computing Maximally Permissive Supervisor (MPS)

Let the hard requirement automaton be $\mathscr{A}(D^h) = \langle S, 2^{I \cup O}, \delta, F \rangle$. We construct the maximally permissive supervisor by applying $C_{step}(s,X)$ (for $X \subseteq F$) to compute

set of winning states G. The function C_{step}: X $\times$ $2^X \rightarrow \{true, false\}$ for a given *SSDFA* representation of an automaton $\mathscr{A}(D^h)$ and $X \subseteq F$ is defined as follows:

C_{step}(s, X) = $true$ if $\forall\ i \in I, \exists\ o \in O : \delta(s,(i,o)) \in$ X, otherwise $false$.

This requires efficient implementation of $C_{step}(s,X)$ over MTBDD representation of $\mathscr{A}(D^h)$. This allows us to compute $Cpre(\mathscr{A}(D^h),X)$ of Section 4.2.1, by iterating C_{step} over all the states $s \in X$. The process is repeated till the fixed point is reached as given in algorithm of Figure 4.5.

The algorithm for efficient computation of C_{step} is given in Figure 4.3. The algorithm for C_{step} marks, **(a)** each leaf node representing state s by truth value of $s \in X$, **(b)** each decision node associated with an input variable with *AND* of its children's value, and **(c)** each decision node associated with output variable with *OR* of its children's value. The computation is carried out recursively with memoization on $MTBDD$ and takes time of the order of $|MTBDD|$, where $|MTBDD|$ is the number of BDD nodes in it. In contrast the enumerative method for implementation of C_{step} would have taken time of the order of $2^{|I \cup O|}$.

Algorithm $C_{step}(s,X)$, applied to each member of X over MTBDD representation $\mathscr{A}(D^h)$, gives us $Cpre(\mathscr{A}(D^h),X)$ (as given in line 7 to 11 of algorithm of Figure 4.5). Finally $Cpre(\mathscr{A}(D^h),X)$ is used to compute the largest set of winning states $G \subseteq F$ as given in the Section 4.2.1. The pseudo code of this is given in the Figure 4.5. The overall time complexity of *ComputeWinning* is $O(|F| \times (|MTBDD|))$, where $|F|$ is the size of set of final states in $\mathscr{A}(D^h)$. This worst case complexity comes from the fact that there can be maximum $|F|$ iterations to reach the fixed point. In practice the algorithm converges much faster. The loop for clearing the visited field of each BDD node using memoization (See algorithm $ClearVisited$ in Figure 4.4) after each fix point iteration will take time

$O|MTBDD|$, as every BDD node has to be traversed only once. Similarly, the loop for computing new status field with memoization for each BDD node will also take time $O|MTBDD|$.

Next, given a set G of winning state for $\mathscr{A}(D^h)$, we compute the automaton

---

```
1  ComputeCstep
2  Input: n = BDD node for s in A(D^h), X ⊆ F
3  Output: Boolean status (representing whether s is in Cpre(A(D^h), X)).
4  if n.visited then
5      return n.status /* Memoization */
6  else
7      if n is a terminal node then
8          n.status = (n.state ∈ X)
9      else
10         if n.variable is an input variable then
11             n.status = ComputeCstep(n.left,X) AND ComputeCstep(n.right,X)
12         else
13             n.status = ComputeCstep(n.left, X) OR ComputeCstep(n.right, X)
14         end
15     end
16         n.visited = true
17         return n.status
18 end
```

Figure 4.3: Algorithm for computing C_{step} function

$\mathscr{A}^{MPS} = \langle S^{MPS}, 2^{I \cup O}, \delta^{MPS}, F^{MPS} \rangle = \langle G \cup \{r\}, 2^{I \cup O}, G, \delta' \rangle$ by only retaining transitions between the winning states G. Call this step as $MPSSubgraph(\mathscr{A}(D^h), G)$ Here r is the unique reject state introduced to make the automaton total. We consider the following two methods for computing $MPSSubgraph(\mathscr{A}(D^h), G)$.

– *Enumerative method:* $\mathscr{A}^{MPS}$ is constructed from $\mathscr{A}(D^h)$ by adding a transition at a time as follows: for any $s \in G$ if $\delta(s,(i,o)) \in G$ then add the

```
1  ClearVisited
2  Input: n (a MTBDD node)
3  Output: NIL.
4  if n is a terminal node then
5      n.visited = false
6      return
7  end
8  if not n.visited then
9      return /* Memoization */
10 else
11     ClearVisited(n.left)
12     ClearVisited(n.right)
13     n.visited = false
14     return
15 end
```

Figure 4.4: Algorithm for clearing the visited field of MTBDD structure below a given node

transition $(s,(i,o),\delta(s,(i,o)) \in \delta'$ to $\mathscr{A}^{MPS}$. Clearly, this algorithm has lower bound on time complexity of the order of $|S| \times 2^{|I \cup O|}$. Finally, we make A^{MPS} total by adding all the unaccounted transitions from any state to the reject state r.

– *Symbolic method:* in this method, the MTBDD of $\mathscr{A}(D^h)$ is modified so that each edge pointing to a state in $S - G$ is changed to go to the reject

---

1 **ComputeWinning**
2 **Input**: $\mathscr{A}(D^h) = \langle S, 2^{I \cup O}, \delta, F \rangle$
3 **Output**: $G \subseteq F$.
4 $G = F$
5 **do**
6 **for** every state $s \in S$
7 ClearVisited(s.bddNode)
8 **end**
9 $G' = G$
10 **for** each s ∈ G'
11 **if** ComputeCstep(s.bddNode, G') $= false$ **then**
12 remove s from G
13 **end**
14 **end**
15 **while(G != G')**
16 **return** G

Figure 4.5: Algorithm for computing the winning region

state r. Note that this makes states in $S-(G\cup\{r\})$ inaccessible. Now this modified SSDFA is minimized to get rid of inaccessible states and to get smaller MPS. The worst case time complexity of this computation is $O(|MTBDD|)$ for modifying the links and $(max(|MTBDD|,|F|)\times|F|)$ for minimization although in practice the minimization is much faster[55]. It should be noted that typically $|MTBDD| << 2^{|I\cup O|}$. Hence the symbolic algorithm has better worst case complexity.

We give experimental results of computation of $MPSSubgraph(\mathscr{A}(D^h),G)$ using the two algorithms, in Table 4.2. The specifications for $Arbiter(n,k,b)$ and $MinePump(w,\varepsilon,\zeta,\kappa)$ in the table are $Type3$ specification given in section 4.3.4. It can be seen that the symbolic algorithm can be faster by several orders of magnitude. This is because we do not construct the MPS from scratch; instead we only redirect some links in MTBDD of $A(D^h)$ which is already computed.

4.4.2 Computing Maximally Permissive H-Optimal Supervisor (MPHOS)

In this step we compute the $\mathscr{A}^{MPHOS}=\langle S^{MPHOS},2^{I\cup O},\delta^{MPHOS},F^{MPHOS}\rangle$ from $\mathscr{A}^{MPS}=\langle S^{MPS},2^{I\cup O},\delta^{MPS},F^{MPS}\rangle$. For a given maximally permissive supervisor $\mathscr{A}^{MPS}$, a QDDC formula D^s and an integer parameter H. We get the H-Optimal sub-supervisor called $\mathscr{A}^{MPHOS}$ by iteratively computing $Val(s,p+1)$ from $Val(s,p)$ over an automaton $\mathscr{A}^{Arena}=\mathscr{A}^{MPS}\times\mathscr{A}(Ind(D^s,w))$ for $0\leq p<H$ as outlined in Section 4.2.2 [2].

[2]Note that the tool DCSynth in general allows a weighted list of soft requirement (QDDC formulas). The tool DCSynth implements a MPHOS computation based on this lexicographical

Table 4.2: $MPSSubgraph(\mathscr{A}(D^h), G)$ computation: Enumeration vs symbolic method (time in milli seconds and memory in MB). Under the column $\mathscr{A}(D^h)$ we give the number of states in monitor automaton and the computation time for *QDDC* specification of our examples. Under $\mathscr{A}^{MPS}$ We give the number of states and time to compute *MPS* using the above two methods.

Example	$\mathscr{A}(D^h)$		$\mathscr{A}^{MPS}$		
	States	Time (Memory)	States	Time (Memory) (Enumerative)	Time (Memory) (Symbolic)
$Arbiter(4,2,2)$	18	0.03 (9.78)	7	0.65 (3.6)	0.09 (3.6)
$Arbiter(4,3,2)$	44	0.03 (9.90)	11	0.88 (3.6)	0.14 (3.6)
$Arbiter(4,4,2)$	82	0.03 (9.89)	15	1.27 (3.6)	0.26 (3.6)
$Arbiter(5,3,2)$	64	0.05 (9.89)	13	2.18 (4.1)	0.22 (4.1)
$Arbiter(5,4,2)$	124	0.06 (9.92)	18	3.18 (4.1)	0.48 (4.1)
$Arbiter(5,5,2)$	204	0.10 (14.4)	23	4.04 (5.4)	0.94 (5.4)
$Minepump(8,2,6,2)$	222	0.04 (9.8)	70	1.54 (3.6)	0.66 (3.6)
$Minepump(9,3,7,3)$	383	0.05 (9.8)	68	2.00 (3.6)	1.16 (3.6)
$Minepump(10,4,8,3)$	551	0.04 (9.8)	67	2.44 (3.6)	1.63 (3.6)
$Minepump(11,4,8,3)$	606	0.05 (9.8)	82	2.86 (3.6)	1.75 (3.6)

Remark 2: In $\mathscr{A}^{Arena}$ a transition has the form $\delta(s,(i,o,v))$ with $i \in 2^I, o \in 2^O, v \in 2^{\{w\}}$. However, from the definition of $Ind(D^s,w)$, the value of v is uniquely determined by $(s,(i,o))$ in the corresponding automaton $\mathscr{A}(Ind(D^s,w))$. Hence we can abbreviate the transition as $\delta(s,(i,o))$. □

To compute $\mathscr{A}^{MPHOS}$ we again have two methods: enumerative and symbolic. Let $|Q|$ denote the number of states in $\mathscr{A}^{Arena} = \langle Q, 2^{I \cup O \cup w}, \delta, F \rangle$.

(or with explicit weight to each soft requirement) list by using the weight for each transition as the *sum of weights of all the soft requirement* being satisfied on that transition. The tool also allows the discounting factor γ which is used to give higher weight to the requirements being satisfied in near future.

– *Enumerative Method*: As given in Section 4.2.2 of synthesis method, for each state s we need to enumerate all paths starting from s to get $Val(s, p+1)$ from $Val(s, p)$, which will take time of the order of $2^{|I \cup O|}$. Similar complexity will be required to get the list of transitions with optimal values denoted as $optValue$ (Note that there can be multiple transitions with same $optValue$, all such transitions will be included in $\mathscr{A}^{MPHOS}$). Hence, As the algorithm terminates after H iterations the total time complexity of entire algorithm for H iteration is $|Q| \times 2^{|I \cup O|} \times H$.

– *Symbolic method:* For this optimization to be applicable we require that in MTBDD representation of $\mathscr{A}^{Arena}$, all the input variables occur before the output variables O and the indicating variable w. Indicating variable w gives represents the soft requirement D^s.

Frontier Nodes:

We define a node in MTBDD as *frontier node* if it is labelled with an output or a indicating variable, and there exist at-least one path from a source state such that all its ancestors are labelled with input variables only. Similarly, all the nodes labelled with an input variable are called *input nodes* and the nodes labelled with output or an indicating variable are called *output nodes*. Thus, frontier nodes are a special kind of output nodes. For example, in Figure 4.2(b), frontier nodes are the nodes labelled 2 (they happen to occur at same level in this example).

We now give the algorithm for symbolic computation of MPHOS along with its complexity calculation. We define few terminologies and iterator

constructs, which will be used in algorithm description and calculation of complexity results.

Terminologies:

Given an automaton A^{Arena} as semi symbolic automaton, let MTBDD represent its transition relation in MTBDD form.

- $|MTBDD|$: This represents the total number of BDD nodes in $MTBDD$.
- $|MTBDD_{inputs}|$: This represents the number of input nodes $MTBDD$ i.e. these are the nodes above the frontier nodes.
- $|MTBDD_{outputs}|$: This represents the number of output nodes in $MTBDD$ i.e. these are the nodes below and including the frontier nodes.
- $|MTBDD_{totalPaths}|$: This represents the total number of paths in the $MTBDD$ starting from root nodes till the terminal nodes. Recall that root node is a $MTBDD$ node which is pointed to by a state.
- $|MTBDD_{inputPaths}|$: This represents the total number of paths (input paths) in the $MTBDD$ starting from root nodes till frontier nodes.
- $|MTBDD_{outputPaths}|$: This represents the number of paths (output paths) in the $MTBDD$ starting from frontier nodes to the terminal nodes.

Iterators:

We now describe some iterators which will be used to traverse the $MTBDD$.

- Foreach f in Frontier(n)$\{$stmts(f)$\}$: Starting from the source node n, this control structure traverses the $MTBDD$ by exploring the $false$

and $true$ cofactors till a frontier node f is reached. For each such frontier node execute the statements *stmts* which can use the information available at f. Memoization is used to ensure that the statements *stmts* are executed only once for each frontier node, by marking each input as well as frontier node as visited once it is explored. Thus, the time complexity for this operation without considering the time to execute *stmts* would be equivalent to the number of BDD node before the frontier nodes i.e. $O|MTBDD_{inputs}|$.

– Foreach π_{input} in InputPath(n){stmts(π_{input})}: This control structure iterates over all the input path π_{input}, starting from bdd node n. These input paths are found by recursively traversing the $MTBDD$ structure in depth first order. It executes the statements *stmts* for each input path π_{input} ending at some frontier node f. It can be noted that the time complexity for this operation without considering the time to execute *stmts* would be proportional number of input paths in $MTBDD$ structure i.e. $O|MTBDD_{inputPaths}|$.

– Foreach π_{output} in OutputPath(f){stmts(π_{output})}: Starting from a given frontier node f, this control structure traverses all the output paths and executes statements *stmts* for each output path represented by π_{output}. Output paths are found by recursively traversing the $MTBDD$ structure in depth first order. The time complexity for this control structure without considering the time to execute would be proportional number of output paths in $MTBDD$ structure i.e. $O|MTBDD_{outputPaths}|$.

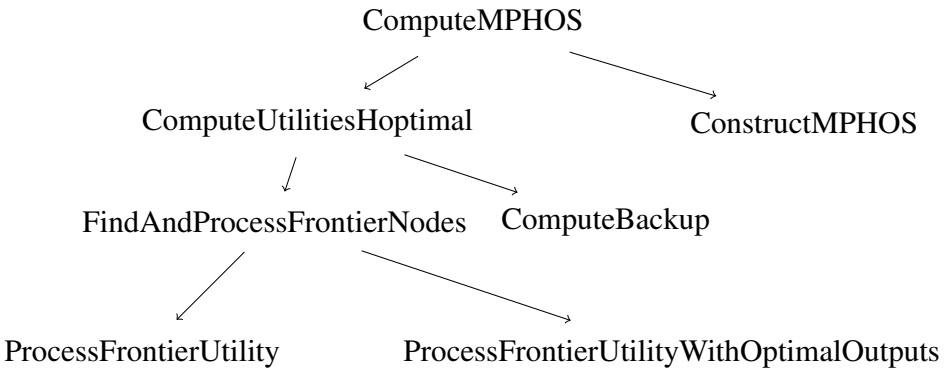


Figure 4.6: Call graph for sub-procedures of ComputeMPHOS function.

---

1 **ComputeMPHOS**

2 **Input**: $\mathscr{A}^{MPS}$, $\mathscr{A}(Ind(D^s,w))$, Natural number H

3 **Output**: $\mathscr{A}^{MPHOS} = \langle S^{MPHOS}, 2^{I \cup O}, \delta^{MPHOS}, F^{MPHOS} \rangle$.

4 `/* Compute` $\mathscr{A}^{Arena}$ `as product of` $\mathscr{A}^{MPS}$ `and` $\mathscr{A}(Ind(D^s,w))$
`Using MONA library function DFAProduct */`

5 $A^{Arena} = DFAProduct(\mathscr{A}^{MPS}, \mathscr{A}(Ind(D^s,w)))$

6 `/* Label the frontier nodes and compute Val array after H iterations */`

7 $A^{Arena}_{labelled}$ = ComputeUtilitiesHoptimal(A^{Arena}, Val, H)

8 `/* Create MPHOS from labelled` $\mathscr{A}^{Arena}$ `*/`

9 $\mathscr{A}^{MPHOS}$ = ConstructMPHOS($A^{Arena}_{labelled}$)

10 **return** $\mathscr{A}^{MPHOS}$

Figure 4.7: Algorithm for Computing $\mathscr{A}^{MPHOS}$ from A^{Arena}

Algorithm for MPHOS computation:

It may be noted that the frontier nodes in $MTBDD$ divide it into two regions, the *input region* containing all the input paths ending at frontier nodes and the *output region* containing all the output paths starting at frontier nodes and ending at the terminal nodes. Therefore, the computation can be naturally divided in to multiple steps as shown in the algorithm *ComputeMPHOS* given in Figure 4.7. The call graph for sub-procedures of *ComputeMPHOS* is shown in Figure 4.6.

ComputeMPHOS Function:

- This function first computes the arena automaton $\mathscr{A}^{Arena}$ by taking product of two automata $\mathscr{A}^{MPS}$ and $\mathscr{A}(Ind(D^s, w))$ using the MONA DFA library function *DFAProduct*.
- We then assign H-Optimal utility values to each state as outlined in Section 4.2.2. This computation is implemented using the algorithm *ComputeUtilitiesHoptimal* given in Figure 4.9, it returns the labelled arena automaton $\mathscr{A}^{Arena}_{labelled}$. In this automaton each frontier node is labelled with H-Optimal utility value and optimal output paths with the corresponding next states. The next paragraph will describe this function in detail.
- Finally, the automaton $\mathscr{A}^{MPHOS}$ is constructed explicitly using the optimal outputs stored in labelled arena automaton $\mathscr{A}^{Arena}_{labelled}$, which is implemented by algorithm *ConstructMPHOS* given in Figure 4.8. This algorithm is pretty straight forward, where construction of $\mathscr{A}^{MPHOS}$

is carried out explicitly using *ConstructMPHOS*. For a given labelled arena automaton $\mathscr{A}^{Arena}_{labelled}$, it traverses all the input paths till frontier node and inserts all the optimal outputs stored in frontier nodes corresponding to the input path to $\mathscr{A}^{MPHOS}$. $\mathscr{A}^{MPHOS}$ is then minimized using MONA DFA library and returned. The time complexity for this

---

1 **ConstructMPHOS**

2 **Input**: $A^{Arena}_{labelled} = \langle Q, 2^{I \cup O \cup w}, \delta, F \rangle$,

3 **Output**: $\mathscr{A}^{MPHOS} = \langle S^{MPHOS}, 2^{I \cup O}, \delta^{MPHOS}, F^{MPHOS} \rangle$.

4 **Pre**: All frontier nodes of $\mathscr{A}^{Arena}$ labelled with optValue and optOutputList

5 **Post**: Minimized $\mathscr{A}^{MPHOS}$ Automaton

6 `/* Create MPHOS from labelled` $\mathscr{A}^{Arena}$ `*/`

7 **for** each state s of A^{Arena}

8 n = s.bddNode

9 **Foreach** π_{input} **in InputPath**(n){

10 Let π_{input} ends at frontier node f

11 **for** each $(\pi_{output}, nextState)$ pair in $f.optOutputList$ list

12 Add transition from s to $nextState$ on $(\pi_{input}, \pi_{output})$ in $\mathscr{A}^{MPHOS}$

13 **end**

14 }

15 **end**

16 Minimize $\mathscr{A}^{MPHOS}$

17 **return** $\mathscr{A}^{MPHOS}$

Figure 4.8: Algorithm for explicit construction of $\mathscr{A}^{MPHOS}$

construction is $O(|MTBDD_{inputPaths}| \times Tr + (max(|MTBDD|, N) \times N)$, where N is the number of states in $\mathscr{A}^{Arena}$, and Tr is the time required to insert one transition in the automaton which is assumed to be constant. Note that $(max(|MTBDD|, N) \times N)$ is the worst case time complexity for minimization in MONA [55].

ComputeUtilitiesHoptimal Function: Now we describe the algorithm for $ComputeUtilitiesHoptimal$, which returns the labelled automaton $A^{Arena}_{labelled}$ after H-Optimal utility computation over A^{Arena}. This computation is as follows:

- The algorithm given in $ComputeUtilitiesHoptimal$ (See Figure 4.9) for H-Optimal output computation has H iterations. We iteratively compute $Val(s, p+1)$ from $Val(s, p)$ (for $0 \leq p < H$) over $MTBDD$ representation of $\mathscr{A}^{Arena}$ using Bellman Backup operation outlined in Section 4.2.2 for each iteration. The $Val(s, 0)$ is initialized with 0.0 for every state in A^{Arena}. Every iteration is further divided into two steps, where the computation is done on the input region and the output region of $MTBDD$ respectively.
- In the first step, $MTBDD$ of A^{Arena} is traversed to find all the frontier node. We then explore all the output paths from each frontier node and store the optimal utility value in that frontier node. In $(p+1)^{th}$ iteration we use $Val(s, p)$, which has the optimal utility value for each state computed in previous (p^{th}) iteration. The algorithm for this step is given by procedure $FindAndProcessFrontierNodes$ (See Figure 4.10).

```
1  ComputeUtilitiesHoptimal
2  Input: 𝒜^Arena = ⟨Q, 2^(I∪O∪w), δ, F⟩ and Val : Q → ℛ, Natural number H
3  Output: A^Arena_labelled
4  /*Initialize the Val array */
5  for each state s of A^Arena
6      Val(s) = 0.0
7  end
8  for count = 0 to H /* Start H Iterations */
9      for every state s ∈ F
10         /* Clear the values from previous iteration*/
11         ClearValues(s.bddNode)
12     end
13     if count = H then   /* i.e. It's a last Iterations */
14         /*Store optimal value along with optimal outputs*/
15         FindAndProcessFrontierNodes(𝒜^Arena, Val, true)
16     else /*Otherwise, store only optimal value*/
17         FindAndProcessFrontierNodes(𝒜^Arena, Val, false)
18     end
19     for every state s ∈ F
20         ClearVisited(s.bddNode)
21     end
22     Val = ComputeBackup(𝒜^Arena)
23 end
24 return A^Arena_labelled    /*Labelled Arena automaton */
```

Figure 4.9: Algorithm for Computing the H-Optimal Utility Values

$FindAndProcessFrontierNodes$ calls procedure $ProcessFrontierUtility$ given in Figure 4.11 over each frontier node, to compute and store the optimal utility values in that iteration. However, in the final (H^{th}) iteration of $ComputeUtilitiesHoptimal$, apart from storing the optimal values, we additionally store the list of optimal outputs and next state in the processed frontier node. These optimal outputs are required to construct the automaton A^{MPHOS} from the labelled automaton. For this, $ProcessFrontierUtilityWithOptimalOutputs$ (See Figure 4.12) is called in the last iteration, this algorithm is similar to $ProcessFrontierUtility$ except that it stores the required additional information on frontier node. We use memoization to ensure that every frontier node is processed only once during each iteration of $ComputeUtilitiesHoptimal$.

The time complexity of $FindAndProcessFrontierNodes$ with memoization on frontier nodes is $O(|MTBDD_{inputs}| + |MTBDD_{outputPaths}|)$, assuming that computation of optimal value on each output path takes a constant time. To see this, notice that the time required to find all the frontier nodes is $O(|MTBDD_{inputs}|$ and time required to process all frontier nodes with memoization, using $ProcessFrontierUtility$ is $O|MTBDD_{outputPaths}|$ as it requires exploring all the output paths once.

– In the second step of each iteration, the function $ComputeBackup$ (See Figure 4.13) is called to update $Val(s, p+1)$ from the optimal values stored at frontier nodes (computed using $Val(s, p)$). Notice that to update $Val(s, p+1)$, we take average of all the optimal values stored on frontier nodes, which can be reached from the input paths starting

```
1  FindAndProcessFrontierNodes
2  Pre: Visited and optValue fields of BDD nodes cleared
3  Post: Mark each frontier node with optimal value
4  Input: 𝒜^Arena = ⟨Q, 2^(I∪O∪w), δ, F⟩, Val : Q → ℛ, storeOptOutputs: boolean
5  Output: 𝒜^Arena_labelled = ⟨Q, 2^(I∪O∪w), δ, F⟩ with all frontier nodes labelled
6  for each state s of A^Arena
7      n = s.bddNode
8     if n.visited then
9        return /* Memoization at root node */
10    end
11    Foreach f in Frontier(n){
12       if f.visited then
13          return /* Memoization at frontier node */
14       else
15          if storeOptOutputs then
16             ProcessFrontierUtilityWithOptimalOutputs(f, 𝒜^Arena, Val)
17          else
18             ProcessFrontierUtility(f, 𝒜^Arena, Val)
19          end
20       end
21    }
22    n.visited = true
23 return 𝒜^Arena_labelled
24 end
```

Figure 4.10: Algorithm for Finding and labeling the frontier nodes

```
1  ProcessFrontierUtility
2  Pre: Visited and optValue fields of BDD nodes cleared
3  Post: Mark the given frontier node with optimal value
4  Input: Frontier Node f, 𝒜^Arena = ⟨Q, 2^(I∪O∪w), δ, F⟩ and Val : Q → ℛ
5  Output: 𝒜^Arena_labelled with frontier node f labelled with optimal value.
6  if f.visited then /*Memoization*/
7      return
8  else
9      Foreach π_output in OutputPath(f){
10         Let π_output ends at state nextState
11         value = wt(π_output) + Val(nextState)
12         if f.optValue < value then
13             f.optValue = value
14         end
15     }
16     f.visited = true
17 end
```

Figure 4.11: Algorithm for computing the utility value from a given frontier node

1 **ProcessFrontierUtilityWithOptimalOutputs**

2 **Pre**: Visited and optValue fields of BDD nodes cleared

3 **Post**: Mark the given frontier node with optimal value

4 **Input**: Frontier Node f, $\mathscr{A}^{Arena} = \langle Q, 2^{I \cup O \cup w}, \delta, F \rangle$ and $Val : Q \to \mathscr{R}$

5 **Output**: $\mathscr{A}^{Arena}_{labelled}$ with frontier node f labelled with optimal outputs.

```
6  if f.visited then /*Memoization*/
7      return
8  else
9      Foreach π_output in OutputPath(f){
10         Let the π_output ends at state nextState
11         value = wt(π_output) + Val(nextState)
12         if f.optValue < value then
13             f.optValue = value
14             f.optOutputList = Initialize with empty list
15             append (π_output, nextState) to f.optOutputList
16         end
17         if f.optValue = value then
18             append (π_output, nextState) to f.optOutputList
19         end
20     }
21     f.visited = true
22 end
```

Figure 4.12: Computing the utility and optimal paths from a given frontier node

1 **ComputeBackup**

2 **Pre**: Visited fields of BDD nodes cleared and all frontier nodes labelled with optValue

3 **Post**: *Val* array containing utility values computed for each state

4 **Input**: $\mathscr{A}^{Arena} = \langle Q, 2^{I \cup O \cup w}, \delta, F \rangle$ with all frontier nodes labelled

5 **Output**: $Val : Q \rightarrow \mathscr{R}$.

6 totalInpPaths = 2^I `/* Total no. of input paths */`

7 **for** each state s of A^{Arena}

8 n = s.bddNode

9 tempVal = 0.0

10 **if** n.visited = false **then** `/*Memoization*/`

11 **Foreach** π_{input} **in InputPath**(n){

12 Let π_{input} ends at frontier node f

13 `/*Count the number of paths represented by` π_{input} `*/`

14 currInpPaths = number of explicit paths represented by π_{input}

15 tempVal = tempVal + f.optValue * (currInpPaths / totalInpPaths)

16 }

17 n.optValue = tempVal

18 n.visited = true

19 **end**

20 Val(s) = n.optValue

21 **end**

Figure 4.13: Algorithm for computing the Bellman Backup

in state s (See line 11 to 16 of the algorithm). The complexity of $ComputeBackup$ is $O(|MTBDD_{inputPaths}|)$ as memoization is used at source node to ensure that every path till frontier node is traversed only once.

- It may be noticed that before each iteration $MTBDD$ nodes are cleared by calling $ClearValues$ (See Figure 4.14), to clear the optimal values and visited field of each BDD node stored during previous iteration. This takes time $O|MTBDD|$ as every node has to be visited only once, which is ensured by memoization.
- Thus, the overall time complexity of $ComputeUtilitiesHoptimal$ is $O(H * (|MTBDD| + |MTBDD_{inputs}| + |MTBDD_{inputPaths}|)) + |MTBDD_{outputPaths}|$, which simplifies to $O(H * (|MTBDD_{inputPaths}| + |MTBDD_{outputPaths}|))$.

4.4.3 Computing Controller: Determinizing MPHOS using Output Preference Ordering

The controller can be computed from MPHOS for an output preference ordering *Ord* given as lexicographically ordered list of output variable literals, as described in Section 4.2.3. $\mathscr{A}^{MPHOS}$ is determinized by retaining only the highest ordered output among those permitted by it. For this we use similar algorithm as used for MPHOS computation, by retaining only those transitions which *locally maximize* the lexicographic weight of *Ord* list. In this case, the frontier nodes will store the lexicographic weight as the optimal value on the frontier nodes, along with the corresponding outputs and the computation is performed for only 1 iteration.

```
1  ClearValues
2  Input: n (a MTBDD node)
3  Output: NIL.
4  if n is a terminal node then
5      n.visited = false
6      n.optValue = -1.0
7      return
8  end
9  if n.visited then
10     return /* Memoization */
11 else
12     ClearValues(n.left)
13     ClearValues(n.right)
14     n.visited = false
15     n.optValue = -1.0
16     return
17 end
```

Figure 4.14: Algorithm for Clearing the fields for MTBDD structure

This gives us the labelled automaton similar to $A^{Arena}_{labelled}$, which has an optimal transition stored at frontier node for each input. The controller is then constructed from this labelled automaton by explicitly adding each transition to the required controller automaton as given in $ConstructMPHOS$. It may be noted that, as the *Ord* provides the literal (either positive or negative) for every output variable, so there will always be a unique output path that will maximally satisfy the weighted list of outputs (See Example 33). Hence, the we will always get a deterministic Mealy machine (controller).

4.5 Discussion

Reactive synthesis from Linear Temporal Logic (LTL) specification is a widely studied area[12] and a considerable number of tools [16, 42] supported by theoretical foundations are available. The leading tools such as Acacia+[16] and BoSy[42] mainly focus on the future fragment of LTL. In contrast, we focus on *invariance* of complex regular properties, denoted by **inv** D^h where D^h is a QDDC formula. For such a property, a maximally permissive supervisor (MPS) can be synthesized. There is an extensive work on requirement modelling using DC, as well as model checking DC and QDDC properties in [84]. However, algorithmic synthesis of controllers from QDDC specification, as presented in this thesis, is new. Fränzle *et al.*[50, 51] presented an early analysis of this problem from duration calculus.

Formally, logics LTL and QDDC have incomparable expressive power. There is increasing evidence that regular properties form an important class of requirements [36, 74]. The IEEE standard PSL extends LTL with regular properties [1].

Wonham and Ramadge in their seminal work [92, 93] first studied the synthesis of maximally permissive supervisors from regular properties. In their supervisory control theory, *MPS* can in fact be synthesized for a richer property class *AGEF* D^h [38]. Tool DCSynth can be easily extended to support such properties too. Riedweg *et al.* [96] give some sub-classes of Quantified Mu-Calculus for which *MPS* can be computed. However, none of these works address soft requirement guided synthesis.

Most of the reactive synthesis tools focus on correct-by-construction synthesis from hard requirements. For example, none of the tools in recent competition on reactive synthesis, SYNTCOMP17 [62], address the issue of guided synthesis which is our main focus. In our approach, we refine the MPS (for hard requirements) to a sub-supervisor optimally satisfying the soft requirements too. Since LTL does not admit MPS, it is unclear how our approach can extend to it.

In quantitative synthesis, a weighted arena is assumed to be available, and algorithms for optimal controller synthesis for diverse objectives such as Mean-payoff [13] or energy [18] have been investigated. In our case, we first synthesize the weighted arena from given hard and soft requirements. Moreover, we use H-Optimality as the synthesis criterion. This criterion has been widely used in reinforcement learning as well as optimal control of Markov Decision Processes (MDPs) [90, 11]. In this theory, the optimal control is computed considering a discounting factor γ such that in computing utility, the reward of a transition after i steps is discounted by a factor γ^i. In this thesis, we have presented the theory assuming $\gamma = 1$. It may be noted that the tool DCSynth allows us to compute MPHOS for a user specified value of γ. The Proposition 32 can be adapted to include such discounted sum. Also, in computing MPHOS, we have assumed that

all inputs are independent and identically distributed (iid). However, the method can easily be adapted to accommodated a finite state Markov model governing the occurrence of inputs (See [90]).

In other related work, techniques for optimal controller synthesis are discussed by Ding *et al.* [36], Wongpiromsarn *et al.* [107] and Raman *et al.* [94], where they have explored the use of receding horizon model predictive control along with temporal logic properties.

Since our focus is on the quality of the controllers, we have also defined metrics and measurement techniques for comparing the controllers for their guaranteed (based on must dominance) and expected case performance. For the expected case measurement, we have assumed that inputs are *iid*. However, the method can easily accommodate a finite state Markov model governing the occurrences of inputs.

DCSynth uses an efficient BDD-based symbolic representation, inherited from tool MONA [66] for storing automata, supervisors and controllers. The use of eager minimization (See Section 4.4 for implementation details) allows us to handle much more complex properties.

We have presented a technique for guided synthesis of controllers from hard and soft requirements specified in logic QDDC. This technique is also implemented in our tool DCSynth. Case studies show that combination of hard and soft requirements provides us with a capability to deal with unrealizable (but desirable), conflicting and default requirements. In context of assumption-commitment based specification, we have shown with case studies that soft requirements improve the expected case performance, where as hard requirements provide certain (but typically conditional) guarantees on the synthesized controller. Hence, the

combination of hard and soft requirements as formulated in *Type*3 specifications offers a superior choice of controller specification. This is confirmed by theoretical analysis as well as experimental results. We have also explored the experimental ability to compare the controller performance using *expected value* and *must dominance* metrics. This helps us in designing better performing controllers.

Chapter 5

Logical Specification and Uniform Synthesis of Robust Controllers

In this chapter, we investigate the synthesis of *robust controllers* from a logical specification of regular properties given in the interval temporal logic QDDC. Our specification encompasses both *hard robustness* and *soft robustness*. Here, *hard robustness* guarantees the invariance of commitment under *relaxed (weakened) assumptions*. A systematic framework for logically specifying the assumption weakening by means of a QDDC formula $Rb(A)$, called *Robustness criterion*, is presented. This can be used with any user specified assumption D_A to obtain a relaxed (weakened) assumption $Rb(D_A)$. A variety of robustness criteria encompassing some existing notions such as k,b resilience as well as some new notions like tolerating non-burst errors and recovery from transient errors are formulated *logically*. The *soft robustness* pertains to the ability of the controller to maintain the commitment for as many inputs as possible, irrespective of any assumption. We present a uniform method for the synthesis of a robust controller which guar-

antees the invariance of specified hard robustness and it optimizes the *expected value* of occurrence of commitment (given as soft robustness) across the input sequences. This uses the framework of soft requirement guided synthesis presented in the previous chapter. Through the case study of a synchronous bus arbiter, we experimentally show the impact of variety of hard robustness criteria as well as the soft robustness on the ability of the synthesized controllers to meet the commitment "as much as possible".

We consider the specification of robust controller using logic QDDC. A controller specification consists of a pair of QDDC formulas (D_A, D_C) giving the *assumption* and the *commitment*, respectively. A standard correctness criterion, termed BeCorrect [12], mandates that in all the behaviours of the synthesized controller, the commitment D_C should hold invariantly provided the assumption D_A holds invariantly. This can be denoted by $(\mathbf{inv}(D_A)) \Rightarrow (\mathbf{inv}(D_C)))$. Criticizing this, a different "strong implication" semantics for BeCorrect has been proposed [68, 87] and this is used by several controller synthesis tools. This can be denoted as $G(pref(D_A) \Rightarrow D_C)$. It may be recalled that $pref(D)$ holds at a point in behaviour if the regular property D has been invariantly true in the past.

Robustness pertains to the ability of D_C holding even when D_A does not hold invariantly in the past [12]. A *relaxed assumption* $Rb(D_A)$ specifies a weaker condition than $pref(D_A)$ and the robust specification can be given by the formula $G(Rb(D_A) \Rightarrow D_C)$. Thus, D_C should hold whenever the relaxed assumption $Rb(D_A)$ holds. We term this as **hard robustness**. For example, given integer parameters k and b, the relaxed assumption $LenCntInt(D_A, k, b)$ holds at a point i if the assumption D_A is violated at most k times in the interval $[i-(b+1), i]$ spanning b previous cycles from the current point i. (This formula is defined for-

mally in Section 5.1.) The controller synthesized under the relaxed assumption $LenCntInt(D_A, k, b)$ would be more robust than *BeCorrect* controller as it will tolerate up to k assumption violations in recent past. Notice that this criterion also allows the controller to *recover* from long but transient violations of the assumption D_A by specifying that the commitment D_C should be re-established once the assumption holds sufficiently in recent past.

We structure the formulation of relaxed assumption $Rb(D_A)$ (used to provide hard robustness) as a pair $(Rb(A), D_A)$ where D_A is the concrete assumption formulated by the user. $Rb(A)$ is a QDDC formula called the **Robustness criterion** which specifies a generic method of relaxing any user specified assumption. A notion of cascade composition $Rb(A) \ll Ind(D_A, A)$ (formalized using logic QDDC), gives the desired relaxed assumption formula $Rb(D_A)$.

We show that logic QDDC can be used to conveniently and systematically formulate a wide variety of robustness criteria $Rb(A)$, such as k,b-resilience of Ehlers and Topku [39] as well as several new notions. These include (a) tolerating "bounded errors in bounded time interval", (b) tolerating non-burst errors, as well as (c) notions of recovery of the commitment within bounded time after recovery of the assumption. For example, the $LenCntInt(D_A, k, b)$ criterion of the previous paragraph has features (a) and (c). It should be noted that the interval logic modalities, bounded counting constraints and second order quantification features of logic QDDC are particularly helpful in the formulation of these robustness criteria. As our experiments will show, robustness criteria have a significant impact on the empirically measured robustness of the resulting controller.

Given a hard robustness specification as a tuple $(D_A, D_C, Rb(A))$, we give a uniform synthesis method for synthesis of corresponding controller using our tool

DCSynth, for various notions of robustness, formulated as QDDC formula $Rb(A)$. This is in contrast to the previous work where specific synthesis algorithms were developed for each proposed robustness criterion [12, 39].

Complementary to hard robustness, the **soft robustness** pertains to the ability of a controller to meet the commitment D_C even when the relaxed assumption $Rb(D_A)$ does not hold. The controller synthesis technique should try to satisfy D_C "as much as possible" irrespective of the assumption. Bloem *et al.* have called this notion "never give up" [14].

To exploit the concept of soft robustness, we synthesize a controller by specifying D_C as the *soft requirement* in our tool DCSynth. As given in the previous chapter, it will maximize the expected value of count of D_C over next H-steps. This count is averaged over all inputs of length H. In this formulation we are guided by the H-Optimal receding horizon controller synthesis of Markov Decision processes [90, 11].

We evaluate our method using our two case studies of a Synchronous bus arbiter and a Mine pump controller specification. We synthesize controllers under various robustness criteria and also with soft requirement optimization. To evaluate the performance of these synthesized controllers, we empirically measure the probability of commitment holding in long run under random (independent and uniformly distributed) inputs. Our experiments show that soft robustness has a marked impact on the performance of the controller although it cannot give the firm (conditional) guarantees provided by hard robustness. Thus, our synthesis technique, combining both the hard and the soft robustness, seems useful.

5.1 Specification and Synthesis of robust controllers

This section formally introduces the notion of Robust specification and a method for synthesis of controller from such a specification. We define the ***Robust specification*** as a triple $(D_A, D_C, Rb(A))$, where the QDDC formulas D_A and D_C specify regular properties over input-output alphabet (I,O) representing the *assumption* and the *commitment*. Moreover, $\mathbf{Rb}(\mathbf{A})$ is a formula over an indicator propositional variable A (See definition 19). These indicator variables can be used as auxiliary propositions in another formula using the notion of cascade composition $\ll$ defined below. Formula $Rb(A)$ is called a ***Robustness criterion***. It specifies a generic method for relaxing any arbitrary assumption formula D_A, using the notion of *cascade composition*.

Definition 38 (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:*

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

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 regular properties given as QDDC formulas.

Example 39. *Let* `Rb(A)` $=$ (`scount !A <= 3`) *which holds at a point provided the proposition* `A` *is false at most 3 times in entire past.*

Consider formula `D' = (true^<!req>^(slen=1)) || (true^<ack>)` *which holds at a point provided either the current point satisfies* `ack` *or the previous point does not satisfy* `req`.
Then, the formula ϕ = `(Rb(A)` $\ll$ `Ind(D',A))` *is equivalent to the formula* `(scount !A <=3) && pref(EP(A) <=> D')`. *This states that the count of* `!A` *in past is at most 3, and* `D'` *holds exactly at those points where* `A` *holds. Thus, the whole formula* ϕ *holds provided* `D'` *is false at most* 3 *times in the entire past.*

Given a robustness criterion $Rb(A)$ and concrete assumption formula D_A, we make use of the cascade composition (see Definition 38) to get the *relaxed assumption* under which the commitment must hold. The desired relaxed assumption is given by

$$Rb(A) \ll Ind(D_A, A)$$

which results in a QDDC formula over the input-output variables $(I, O \cup \{A\})$. See Example 39.

The *Robust specification* $(D_A, D_C, Rb(A))$ gives us the *DCSynth specification* below, which is denoted by $Rb_{Spec}(D_A, D_C, Rb(A))$.

$$(I, (O \cup \{A\}),\ ((Rb(A) \ll Ind(D_A, A)) \Rightarrow D_C),\ D_C) \tag{5.1}$$

The hard requirement states that the commitment D_C **must** hold whenever the relaxed assumption $Rb(A) \ll Ind(D_A, A)$ holds. Moreover, it specifies D_C as the soft-requirement. The controller must be **optimized** so that the soft requirement D_C holds for as many inputs as possible, irrespective of the assumption.

5.1.1 Synthesis from Robust Specification

The robust controller synthesis method supported by tool DCSynth is as follows.

– The user provides the *Robust specification* $(D_A, D_C, Rb(A))$. The user also provides a horizon value H.

– The corresponding DCSynth specification is automatically obtained as outlined in Equation 5.1. The tool DCSynth is used to obtain MPS and $MPHOS$ supervisors as described in Section 4.2. These supervisors are denoted as $MPS(D_A, D_C, Rb(A))$ and $MPHOS(D_A, D_C, Rb(A), H)$ respectively. Both these supervisors may be output non-deterministic. The reader should recall that the MPS supervisor only guarantees the hard-robustness, whereas the $MPHOS$ supervisor further improves MPS by optimizing the soft-robustness (while retaining hard-robustness guarantee).

– The user specifies an output order *ord* as a prioritized list of output literals. This is used to get the deterministic controllers $Det_{ord}(MPS(D_A, D_C, Rb(A)))$ and $Det_{ord}(MPHOS(D_A, D_C, Rb(A), H))$, respectively.

5.1.2 Designing Robustness Criteria (hard robustness)

A key element of our robust specification $(D_A, D_C, Rb(A))$ is the robustness criterion $Rb(A)$, where the truth value of proposition A at any point indicates whether the assumption is satisfied at that point or not. $Rb(A)$ provides a generic method of relaxing any assumption D_A. We propose a systematic methodology for designing robustness criteria $Rb(A)$ based on formulas *Error-Type* and *Error-Scope*. Here Error-Type is a QDDC formula that specifies which intervals in a behaviour are to be treated as erroneous. For example `scount !A > 3` specifies all intervals with more than 3 violations of the assumption. An Error-Scope formula, parameterized by error-type formula Err specifies intervals in the behaviour where error

Pos	0	1	2	3	4	5	6	7	8	9
$\sigma(A)$	1	0	0	1	0	1	0	0	0	1

Figure 5.1: Example behaviour for Error-Types

intervals of type Err are forbidden. For example, no sub-interval within last k cycles must satisfy Err. Together, an Error-Scope formula instantiated with an Error-Type formula gives us a robustness criterion $Rb(A)$

Error-Types: Defining Intervals with Assumption Errors The indicator proposition `A` designates points in the behaviour, where the assumption is satisfied. In terms of this, we define several Error-type formulas, below. Each formula specifies which intervals are to be called erroneous. Each Error-Type formula is explained using the example behaviour given in Figure 5.1, where *Pos* gives the position in the behaviour and assumption violation at a position is indicated by 0 at that position.

1. Intervals where `A` is violated at the end-point of the interval.
   `LocalErr(A) = (true^<!A>)`
   For example, $\sigma,[1,4]$, $\sigma,[0,8]$ satisfies the `LocalErr(A)`, whereas $\sigma,[0,3]$, $\sigma,[2,5]$ does not satisfy it.

2. Intervals having more than k assumption violations (`!A`).
   `CountErr(A,k) = (scount !A > k).`
   For example, $\sigma,[1,6]$ satisfies `CountErr(A,2)` as the number of assumption violations in this interval are 4, but `CountErr(A,4)` is not satisfied in this interval.

3. Burst error interval is an interval where the assumption proposition `A` remains false invariantly. So, the intervals where there is burst error of length more than k is given by the formula

   `BurstErr(A,k) = ([[!A]] && slen >= k).`

   And the intervals containing a sub-interval with `BurstError(A,k)` are given by

   `HasBurstErr(A,k) = (<>(BurstErr(A,k))).`

   For example, $\sigma,[6,8]$ satisfies `BurstErr(A,2)` and all the intervals containing $\sigma,[6,8]$ e.g $\sigma,[2,9]$ satisfies `HasBurstErr(A,k)`.

Each formula in the above list is called an *Error-Type* formula. Proposition 40 gives the relation between various Error-Type formulas.

Proposition 40. *For all Error-type formulas Err, Err_1 and Err_2 we have*
(a) $\models BurstErr(A,k) \Rightarrow HasBurstErr(A,k)$
(b) $\models HasBurstErr(A,k) \Rightarrow CountErr(A,k)$
(c) If $j < k$ *then* $\models CountErr(A,k) \Rightarrow CountErr(A,j)$
and $\models HasBurstErr(A,k) \Rightarrow HasBurstErr(A,j)$

Proof. These implications hold straightforwardly using the QDDC semantics. For example, $HasBurstErr(A,k)$ = <> ([[!A]] && $slen >= k$) (by definition). This formula is satisfied by all the interval having burst assumption violations of length at-least $k+1$, these intervals will essentially satisfy total violation count of more than $k+1$ also. Thus it logically implies $CountErr(A,k)$ giving us (b). We omit the remaining proofs. □

Pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\sigma(A)$	1	1	0	1	1	1	0	0	0	1	1	1	1	1

Figure 5.2: Example behaviour for position and length based Criteria

Error-Scope and Robustness Criteria Let Err be any of the Error-Type formulas giving the erroneous intervals. We now specify the restrictions on occurrence of these erroneous intervals by suitable QDDC formulas. These are called *Error-Scope* formulas. They are parameterized by the specification of an erroneous interval Err. Error-scope formulas are evaluated at a position i in a behaviour σ (see Definition 1). For any position i in the behaviour, the formula indicates for which sub-intervals of $[0,i]$ the given Err should not occur. An *Error-Scope* formula instantiated with a specific *Error-Type* gives rise to a Robustness Criterion. We categorize the various Error-Scope and corresponding Robustness Criteria as position based, length based or the resilience based.

1. **Position based Error-Scope:** We use the following two Error-Scope formulas for this.

   `NeverInPast(Err) = !<>Err.`

   and `NeverInSuffix(Err) = !(true^Err)`

   The first formula states that the error never occurs in past for any interval. Thus, when evaluated at position i, no sub-interval of $[0,i]$ should satisfy Err. Once the Err has occurred at position i, this formula can never be satisfied at any future position $j \geq i$ even if in the successive behaviour Err does not occurred.

   The second formula states that the error does not occur in any suffix interval.

This is an *recoverable* version of the above Error-Scope formula, where the formula starts holding again, if the Err does not occur in successive behaviour.

Position based Robustness Criteria: Instantiating the above scope formulas with the `LocalErr(A)` gives rise to the criteria `BeCorrect(A)` and `BeCurrentlyCorrect(A)` respectively (Row 2 and 3 in Table 5.1).

Criterion `BeCorrect(A)` holds at a point if `!A` never occurs in its past e.g. at position 0 and 1 in Figure 5.2 formula `BeCorrect(A)` is satisfied, but not at the any position beyond 1, as the assumption is violated at 2. Thus, the synthesized controller under this criterion meets the commitments invariantly if the assumptions are met invariantly [14] and if assumption is violated even once then the commitment may never be met again. By contrast `BeCurrentlyCorrect(A)` holds at a point i (irrespective of the past), if `A` holds at position i. Thus, in Figure 5.2, apart from positions 0 and 1, the positions 3, 9 also satisfy `BeCurrentlyCorrect(A)`. This criterion gives rise to the controllers which are *recoverable* i.e. if relaxed assumptions start meeting again then the commitments should also be met immediately. Thus, the later is implied by the former and is more robust against the intermittent assumption violations, (but less realizable).

2. **Length based Error-Scope:** We use the following two Error-Scope formulas for this.

```
NeverInPastLen(b,Err) =
       !<>(slen <= b-1 && Err)
NeverInSuffixLen(b,Err) =
```

```
!(true^((slen <= b-1) && Err))
```

The first formula says that the Err never occur in any past interval of length b or less cycles. This formula is not recoverable because once an Err occurs at any interval in past this formula can never be satisfied again in any successive interval. The second formula says that Err does not occur in an interval spanning last b or less cycles. This formula is recoverable as the satisfiability of this formula does not depend on the behaviour beyond past b cycles.

Length based Robustness Criteria: Instantiating the above scope formulas with the `CountErr(A,k)` as well as `HasBurstErr(A,k)`, gives rise to the criteria `LenCnt(A,k,b)`, `LenCntInt(A,k,b)`, `LenBurst(A,k,b)` and `LenBurstInt(A,k,b)` respectively (Row 4-7 in Table 5.1). We give intuitive description of these criteria.

Criterion `LenCntInt(A,k,b)` with integer parameters k, b holds at a point i provided in last b cycles from i, assumption violation `!A` occurs at most k times. The past beyond last b cycles does not affect its truth. E.g. position 11 in Figure 5.2 satisfies `LenCntInt(A,2,5)` as there is number of assumption violations is only 2 in last 5 positions, however at position 10 the criterion `LenCntInt(A,2,5)` is not satisfied as there are 3 assuption violations in $\sigma, [6, 10]$. Thus, the controllers synthesized under this Robustness Criterion will recover and start meeting the commitments again as soon as the number of assumption violations are at-most k in last b cycles. In contrast, the controller synthesized under the criterion `LenCnt(A,k,b)` will never meet the commitments in future of a position i once their are more than k assumption violations in last b cycles from position i.

Pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$\sigma(A)$	1	1	0	0	0	0	1	1	1	1	0	0	0	1

Figure 5.3: Example behaviour for resilience based Criteria

Criterion `LenBurstInt(A,k,b)` is a new Robustness Criterion based on burst assumption violations. A burst error of length k is said to occurs if `!A` occurs continuously for k points. `LenBurstInt(A,k,b)` holds at point i provided in last b cycles before i, there is no burst error of length k. E.g. in Figure 5.2, position 9 does not satisfies `LenCntInt(A,3,8)` as there are 4 violations between position 1 to 9, but it satisfy `LenBurstInt(A,3,8)` as the maximum length of burst error is not more than 3. This criterion is recoverable.

3. **Resilience based Error-Scope:** An interval is said to be a *recovery interval*, if `A` is invariantly *true* in that interval. The length of this interval is called the *recovery period*. Intervals where all recovery periods are of length less than b is denoted by

   ```
   HasNoRecovery(A,b) =   ([]([[A]] => slen < b-1))
   ```

   We use the following two Error-Scope formulas.

   ```
   NeverInPastRes(b,Err) =
         NeverInPast(Err && HasNoRecovery(A,b))
   NeverInSuffixRes(b,Err) =
         NeverInSuffix(Err && HasNoRecovery(A,b))
   ```

   The first scope formula holds at position i provided there is no sub-interval $[b,e]$ of $[0,i]$ such that *Err* holds for $[b,e]$ but there is no sub-interval $[b',e']$

of $[b,e]$ which is a *recovery interval* of b cycles (length $b-1$). This formula is satisfied by a behaviour, if there has never been an Err without a *recovery period* of length b inside it.

The second formula says that *Err* without a *recovery period* does not occur in any suffix interval. This formula is recoverable as its satisfiability only depends upon the behaviour since the **last** *recovery interval*.

Resilience based Robustness Criteria: Instantiating the above scope formulas with `CountErr(A,k)` and `HasBurstErr(A,k)`, gives rise to four different robustness criteria namely `ResCnt(A,k,b)`, `ResCntInt(A,k,b)`, `ResBurst(A,k,b)` and `ResBurstInt(A,k,b)` respectively (Row 8-11 in Table 5.1). We now compare these criteria with the resilience based criteria proposed in the literature.

Ehlers *et al.* [39] introduced the notion of a resilient controller. A controller is said to be resilient if it meets the commitments invariantly provided there are only bounded assumption violation (given by integer parameter k) between any two successive *recovery intervals* with period b. The k,b-resiliance of [39] can be specified as the robustness criterion `ResCnt(A,k,b)`. This holds at a point i in a behaviour provided that between any two successive *recovery intervals* of length b in the past of i, the count of violation of `A` is at most k. This criterion is not recoverable.

We have also encoded the recoverable version of this criterion, which is denoted by `ResCntInt(A,k,b)`. This criterion holds at a point provided after the last recovery interval of length b, or the start of the behaviour if there is no such recovery period, the count of violation of `A` is at-most k.

The past beyond the last recovery interval of length b does not affect its truth. E.g. in Figure 5.3, the criterion `ResCntInt(A,2,4)` is satisfied up to position 3 from start as the total count of assumption violations is only 2 up to that position. It is not satisfied at any position between 4 and 8 as there are more than 2 errors and is no recovery. A recovery occurs at position 9. Hence position 11 satisfies the criterion as there are only 2 errors since the last recovery which ended at position 9. The formula is not satisfied at position 13 as there are 3 violations since the last recovery. The controller synthesized under this criteria shall start meeting the commitments again as soon as there is a recovery period, and it will maintain commitment till the number of assumption violations are at-most k after the last recovery period of length b.

Following proposition gives the relation between various Error-Scope formulas.

Proposition 41. *For any Error-type formula Err, Err_1 and Err_2 we have*

(a) $\models NeverInPast(Err) \Rightarrow NeverInSuffix(Err)$

(b) $\models NeverInPastLen(b, Err) \Rightarrow NeverInSuffixLen(b, Err)$

(c) $\models NeverInPast(Err) \Rightarrow NeverInPastLen(b, Err)$

(d) $\models NeverInSuffix(Err) \Rightarrow NeverInSuffixLen(b, Err)$

(e) For any scope formula $SCP(Err)$ defined above, we have, if $\models Err_1 \Rightarrow Err_2$, *then* $\models SCP(Err_2) \Rightarrow SCP(Err_1)$

Proof. The proofs of these implications are immediate from definitions using QDDC semantics. For example, $(true\hat{}Err)$ implies $(true\hat{}Err\hat{}true)$ which equals $\langle\rangle Err$. Hence, $NeverInPast(Err)$ which equals $!\langle\rangle Err$ implies $!(true\hat{}Err)$ which equals $NeverInSuffix(Err)$. This gives us (a).

Table 5.1: Robustness criteria $Rb(A)$ defined using Error-types and Error-Scope formulas. There may use additional integer parameters k, b.

Sr.	**Robustness Criteria**	**Definition of** $Rb(A)$
1.	AssumeFalse(A)	`(false)`
2.	BeCorrect(A)	`NeverInPast(LocalErr(A))`
3.	BeCurrentlyCorrect(A)	`NeverInSuffix(LocalErr(A))`
4.	LenCnt(A,k,b)	`NeverInPastLen(b,CountErr(A,k))`
5.	LenCntInt(A,k,b)	`NeverInSuffixLen(b,CountErr(A,k))`
6.	LenBurst(A,k,b)	`NeverInPastLen(b,HasBurstErr(A,k))`
7.	LenBurstInt(A,k,b)	`NeverInSuffixLen(b,HasBurstErr(A,k))`
8.	ResCnt(A,k,b)	`NeverInPastRes(b,CountErr(A,k))`
9.	ResCntInt(A,k,b)	`NeverInSuffixRes(b, CountErr(A,k))`
10.	ResBurst(A,k,b)	`NeverInPastRes(b, HasBurstErr(A,k))`
11.	ResBurstInt(A,k,b)	`NeverInSuffixRes(b, HasBurstErr(A,k))s`
12.	AssumeTrue(A)	`(true)`

As a second example, $\mathit{NeverInPast}(\mathit{Err}) = (!\langle\rangle(\mathit{Err}))$ (by definition), which implies $!\langle\rangle((\mathit{slen} \leq (b-1))\ \&\&\ \mathit{Err})$. This, by definition, equals the formula, $\mathit{NeverInPastLen}(b, \mathit{Err})$. Hence we get (c). We omit the remaining proofs. □

5.1.3 Robustness Order and Comparison

Recall the definition of Maximally Permissive Supervisor (MPS) from Section 4.2.1. By the monotonicity of greatest fixed point computation used for construction of MPS, we get the following result.

Proposition 42 (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.*

Proof.
- As $D_1 \Rightarrow D_2$, we have $L(D_1) \subseteq L(D_2)$. Therefore, $MPS(D_1)$ will also be a supervisor (but may not be maximal) for the formula D_2 and $MPS(D_2)$ may have some additional behaviour which makes it maximal. Thus, $L(MPS(D_1)) \subseteq L(MPS(D_2))$ which is equivalent to $MPS(D_2) \leq_{det} MPS(D_1)$ (by Definition 18).
- From the above argument that $MPS(D_1)$ will also be a supervisor for D_2 and from the fact that if there exist a supervisor for any formula D then $MPS(D)$ also exist. We get the second result.

□

Given robustness criteria $Rb_1(A)$ and $Rb_2(A)$, we say that $Rb_1(A)$ implies $Rb_2(A)$ provided $\models Rb_1(A) \Rightarrow Rb_2(A)$. This gives us the **implication ordering** on robustness criteria. The following theorem shows that implication ordering improves the worst case guarantee of D_C holding but it makes supervisors less realizable.

Theorem 43. *Let $\models Rb_1(A) \Rightarrow Rb_2(A)$. Then for all QDDC formulas D_A, D_C, we have*
(a) $MPS(D_A, D_C, Rb_1(A)) \leq^{D_C}_{must} MPS(D_A, D_C, Rb_2(A))$.
(b) If the specification $(D_A, D_C, Rb_2(A))$ is realizable then $(D_A, D_C, Rb_1(A))$ is also realizable.

Proof. For $1 \leq i \leq 2$, we have $MPS(D_A, D_C, Rb_i(A)) = MPS(\phi_i)$, where $\phi_i = ((Rb_i(A) \wedge pref(EP(A) \Leftrightarrow D_A)) \Rightarrow D_C)$ (by Definition 38 and Equation 5.1).

Now, as $\models Rb_1(A) \Rightarrow Rb_2(A)$, we have $\models (Rb_2(A) \Rightarrow D_C) \quad \Rightarrow \quad (Rb_1(A) \Rightarrow D_C)$ and hence for any formula D, we have $\models ((Rb_2(A) \wedge D) \Rightarrow D_C) \ \Rightarrow \ ((Rb_1(A) \wedge D) \Rightarrow D_C)$. Thus $\models \phi_2 \Rightarrow \phi_1$.

Then, by Proposition 42, we have $MPS(\phi_1) \leq_{det} MPS(\phi_2)$. From this and by applying Proposition 36, we get the desired result $MPS(\phi_1) \leq^{D_C}_{must} MPS(\phi_2)$. We also get (b). □

The various robustness criteria specified above can be ordered by **implication ordering** which also improves the hard robustness of the synthesized controller (See Theorem 43). Figure 5.4 gives the implication ordering between the robustness criteria of Table 5.1. Hence, in specification $(D_A, D_C, Rb(A))$, the user must use the weakest criterion (under the implication ordering) from the Figure 5.4 which makes the specification realizable.

Theorem 44. *All the implications given in Figure 5.4 are valid.*

Proof. The proofs of these implications follow easily from the definitions of the robustness criteria by using Proposition 40 and 41. As an example, we prove that $\models BeCorrect(A) \Rightarrow BeCurrentlyCorrect(A)$.

Formula $BeCorrect(A)$ equals $NeverInPast(LocalErr(A))$ (by definition), which by Proposition 41(a) implies $NeverInSuffix(LocalErr(A))$ which by definition equals $BeCurrentlyCorrect(A)$, thus proving the claim.

Now, we give proof for $\models ResCntInt(A,k,b) \Rightarrow ResBurstInt(A,k,b)$. We have $\models HasBurstErr(A,k) \Rightarrow CountErr(A,k)$ by Proposition 40(b).

Therefore, by Proposition 40(e), we have $\models RecoveryErr(b, HasBurstErr(A,k))$

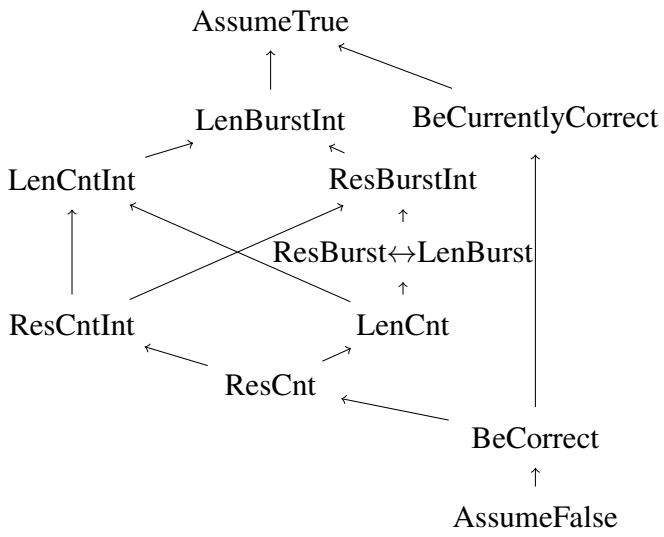


Figure 5.4: Implication order on the robustness criteria of Table 5.1. Here $X \rightarrow Y$ denotes the validity $\models X \Rightarrow Y$. The implication holds for same value of parameters k and b.

$\Rightarrow$ *RecoveryErr*$(b, CountErr(A,k))$. Then, using Proposition 41(e) and instantiating *SCP* by *NeverInSuffix*, we get $\models$ *NeverInSuffix*(*RecoveryErr*$(b, CountErr(A,k)))$ $\Rightarrow$ *NeverInSuffix*(*RecoveryErr*$(b, HasBurstError(A,k)))$. This proves the result.

Finally, we give proof for $\models$ *ResCntInt*$(A,k,b) \Rightarrow$ *LenCntInt*(A,k,b). From Proposition 41(d) we have, $\models$ *NeverInSuffix*$(Err) \Rightarrow$ *NeverInSuffixLen*(b, Err). Also for any formula D, we have $\models$ *NeverInSuffix*$(Err) \Rightarrow$ *NeverInSuffix*$(Err$ && $D)$. Now, by instantiating D with $HasNoRecovery(A,b)$ and Err with $CountErr(A,k)$, we get the result *NeverInSuffix*$(CountErr(A,k)$ && *HasNoRecovery*$(A,b)) \Rightarrow$ *NeverInSuffixLen*$(b, CountErr(A,k))$. This proves the claim by definition.

We omit the remaining proofs. □

5.2 Case Study: A Synchronous Bus Arbiter and Experiments

Please refer to the specification of Synchronous Bus Arbiter given in Section 4.3.4. The specification of the synchronous arbiter is the assumption-commitment pair $(ArbAssume(n,i), ArbCommit(n,k))$, which is denoted by $Arbiter(n,k,i)$. Robust controllers can be synthesized from this with various robustness criteria. For comparison of synthesized controllers, we used the expected value of meeting the commitments in long run by technique presented in Section 4.3.5.

5.2.1 Experimental Results

The synchronous bus arbiter case study specifies the assumption-commitment pair (D_A, D_C) for an $Arbiter(n,k,i)$ consisting of n-cells with response time requirement of k cycles, under the assumption that at most i requests occur in each cycle. Let $detMPS(Arb)$ and $detMPHOS(Arb)$ denote the MPS and the $MPHOS$ controllers for $Arbiter(4,3,2)$ determinized under the output preference ordering `a1 > a2 > a3 > a4`. We have used horizon $H = 50$ in DCSynth for synthesizing the $MPHOS$. The controllers were synthesized for various robustness criteria, and their performance was measured as the corresponding expected values $E_{D_C}(detMPS(Arb))$ and $E_{D_C}(detMPHOS(Arb))$. See Section 4.3.3(i), for the definition of expected value $E_D(Cnt)$ of property D holding for controller Cnt in long run. Table 5.2 provides these values under the columns titled E(Arb-MPS) and E(Arb-MPHOS), respectively.

Following paragraph gives the simulations to illustrating differences in behaviours of some of these controllers.

Simulation of Robust Controllers for Arbiter: The robust controllers synthesized for `BeCorrect`, `BeCurrentlyCorrect`, `LenCntInt` and `ResBurstInt` criteria are encoded as Lustre models. We give the simulation traces (on same inputs) for $Arbiter(4,3,2)$ example using Lustre simulator. It can be seen how each one of them generator different output trace for the same inputs. For example, `BeCorrect` controller does not meet the commitment from cycle 15 and keeps all the acknowledgement lines high, once the assumption is violated in cycle 15. Whereas, the `BeCurrentlyCorrect` controller recovers and start meeting the commitment from cycle 22. Hence it is a more robust controller.

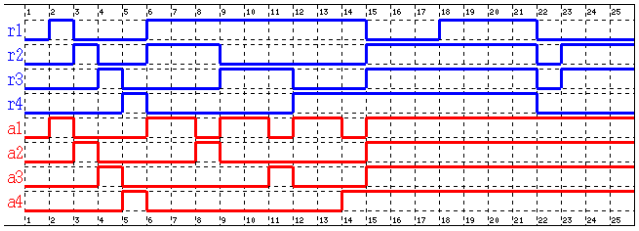


Figure 5.5: Simulation of MPS BeCorrect for Arbiter(4,3,2)

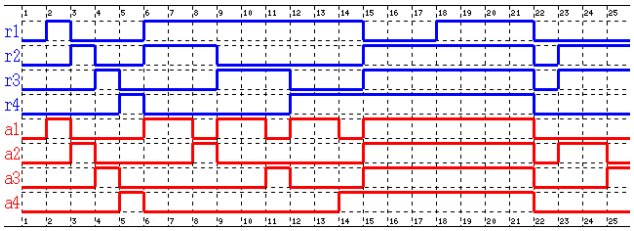


Figure 5.6: Simulation of MPS BeCurrentlyCorrect for Arbiter(4,3,2)

Similarly, for a case study of a Mine-pump Controller specification, we synthesized the determinized *MPS* and *MPHOS* controllers under the preferred output *PumpOn* and various robustness criteria. We measured the performance of

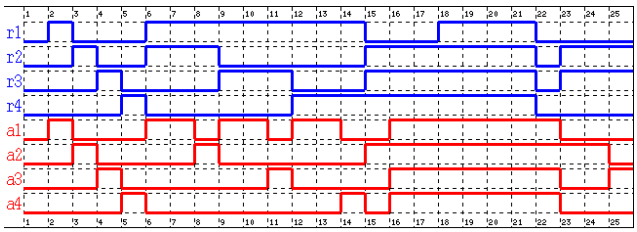


Figure 5.7: Simulation of MPS LenCntInt for Arbiter(4,3,2)

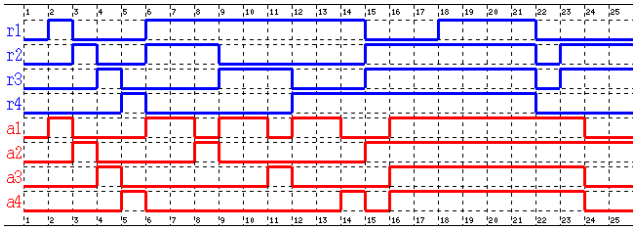


Figure 5.8: Simulation of MPS ResBurstInt for Arbiter(4,3,2)

these controllers as the expected values $E_{D_C}(detMPS(MP))$ and $E_{D_C}(detMPHOS(MP))$. Table 5.2 provides these values under the columns titled E(MP-MPS) and E(MP-MPHOS), respectively. Multiple cells are merged in the table if the obtained supervisors are *syntactically identical*.

In all the above cases, the tool DCSynth performed efficiently by giving the required controllers for Arbiter within 1 seconds and for the Minepump within 3 seconds. All these experiments were done on Ubuntu 18.04 system with Intel i5 64 bit, 2.5 GHz processor and 4 GB memory. It is observed that scalability is mainly limited by the monitor synthesis step.

An examination of Table 5.2 is quite enlightening. We state our main findings.

– In both the case studies, the robustness criterion *AssumeTrue* leads to unrealizable specification whereas all other criteria give realizable specifi-

Table 5.2: Expected value of Commitment D_C holding in Long Runs over random inputs for Controllers synthesized under various Robustness Criteria and integer parameters (k,b).

<table>
<tr><td></td><td colspan="2">Arbiter(4,3,2)</td><td colspan="2">Minepump(8,2,6,2)</td></tr>
<tr><td rowspan="2">Robustness Criteria</td><td>E(ARB-MPS)</td><td>E(ARB-MPHOS)</td><td>E(MP-MPS)</td><td>E(MP-MPHOS)</td></tr>
<tr><td colspan="2">k=1, b=3</td><td colspan="2">k=2, b=8</td></tr>
<tr><td>AssumeFalse</td><td>0.000000</td><td rowspan="10">0.998175</td><td>0.000000</td><td rowspan="2">0.997070</td></tr>
<tr><td>BeCorrect</td><td>0.000000</td><td>0.000000</td></tr>
<tr><td>ResCnt(K,B)</td><td>0.000000</td><td rowspan="4">0.000000</td><td rowspan="4">0.997070</td></tr>
<tr><td>LenCnt(K,B)</td><td>0.000000</td></tr>
<tr><td>ResBurst(K,B)</td><td rowspan="2">0.000000</td></tr>
<tr><td>LenBurst(K,B)</td></tr>
<tr><td>ResCntInt(K,B)</td><td>0.544309</td><td rowspan="2">0.000966</td><td rowspan="2">0.997070</td></tr>
<tr><td>ResBurstInt(K,B)</td><td>0.669069</td></tr>
<tr><td>LenCntInt(K,B)</td><td>0.768066</td><td>0.0027342</td><td rowspan="3">0.997070</td></tr>
<tr><td>LenBurstInt(K,B)</td><td>0.835205</td><td>0.004514</td></tr>
<tr><td>BeCurrentlyCorrect</td><td>0.687500</td><td>0.992647</td><td>0.997070</td></tr>
</table>

cations. Thus, it is advisable to use the weakest criteria *LenBurstInt* or *BeCurrentlyCorrect* for the best hard-robustness.

– In both the case studies, for the *MPS* controllers, the Expected value of commitment D_C increases with the implication ordering given in Figure 5.4. The value ranges from 0 to 83% for the Arbiter and 0 to 99% for the Mine pump. Thus, the robustness criterion has a major effect on the performance of the synthesized *MPS* controller. Also, for many robustness criteria, the Expected D_C value is 0. This happens as these criteria (defined using the Error-Scope formulas *NeverInPast*) are non-recoverable – once the criterion becomes false it remains false in future. Hence, it is desirable to use recoverable criteria defined using the Error-scope formulas *NeverInSuffix*.

– The *MPHOS* controller, which improves the *MPS* controller by optimizing the Expected value of the soft requirement D_C, has an overwhelming impact on the measured expected value of D_C under random inputs. In both case studies, the value is above 99% irrespective of the robustness criterion used. Thus, soft-robustness vastly improves the expected case performance of the controller and should be preferred over *MPS*.

– We often get *MPHOS* controllers with the same/similar expected value of D_C for several robustness criteria. It should be noted that these can be different controllers providing distinct hard-robustness guarantees. For $Minepump(8,2,6,2)$, all the *MPHOS* controllers have same expected value but they are *not* identical. Hence, the circumstances (relaxed assumptions) under which they *guarantee* D_C are quite different.

  In summary, the hard robustness provides the *conditional guarantee* of meeting the commitment D_C under the relaxed assumptions and the soft robustness improves the performance of the controller by optimizing it for meeting commitment D_C even when the relaxed assumptions are not met. Therefore, the combination of hard and soft robustness as supported by our tool DC-Synth is useful.

5.3 Discussion

A robust controller should continue to function (i.e maintain its commitment) under failure of environmental/plant assumptions as much as possible. When such failures are transient, the controller should be able to recover from the failure by reestablishing the commitment in bounded time [76, 14].

Our main contribution is a logical framework for specifying hard and soft robustness and a uniform method for the synthesis of specified controllers. The framework allows a logical specification of Robustness criterion. This provides a generic formula for weakening any user specified assumption. A robust controller which invariantly guarantees the commitment under such weakened assumption (and hence is more robust) is synthesized. Several robustness criteria including the k, b resilience of Ehlers and Topku [39], as well as new notions such tolerating bounded errors in bounded time intervals, tolerating burst errors and recovery from transient errors can be logically specified. Augmenting the above hard robustness, we also optimize the synthesized controller to enhance the expected value of commitment holding irrespective of any assumption. We used the tool DCSynth introduced Chapter 4 for the synthesis of the robust controller. With case studies, we have shown the impact of hard and soft robustness on the expected behaviour of the controller. The experiments show that the combination of hard and soft robustness, as proposed here, is beneficial. Logic QDDC, based on Duration Calculus of Zhou *et al.* [23], provides a very powerful vocabulary for stating robustness properties.

Several authors have investigated the notions of robustness [14, 76, 39]. Bloem *et al.* [14] provide a classification of different robustness notions. The concept of *soft robustness* is termed as "Never Give Up" and *hard robustness* is introduced as "Don't Be Lazy" notion although no logical specification framework or uniform synthesis method is formulated. Ehlers *et al.* [39] propose synthesis of resilient controllers. These provides an ability to tolerate k errors between two periods of recovery of length at-least b. This can be specified as our logical criterion $ResCnt(k, b)$. However, the specified controller cannot recover from

a long transient burst of error. We have generalized this to a recoverable criterion $ResCntInt(k,b)$ by specifying a controller which can tolerate up to k errors after the last recovery period of length b (irrespective of previous history). We also specify new notions of tolerating non-burst errors (criteria *ResBurstInt* and *ResBurst*) as well as length based notions of tolerating k errors in last b cycles (criterion *LenCntInt*). This last criterion is related to the k-robustness of Bloem *et al.* [12], where the ratio between the count of assumption and the count of commitment is stabilized around k. A limitation of our synthesis technique is noteworthy. In our synthesis, concrete values of the parameters k,b have to be specified for synthesis. As compared to this, both Ehlers and Topku [39] as well as Bloem *et al.* [12] give synthesis algorithms which automatically find the (Pareto) optimal values of these parameters and they also synthesize controllers for this optimal value. Inspired by this, our future work will address parametric synthesis of our logical specification.

None of the above cited work analyse the performance of the synthesized controllers. One indicator for this performance is the expected value of commitment holding in average. Another measure is the must-dominance order amongst the synthesized controllers which compares their worst-case behaviour. Such measurement is important as robustness criterion has significant impact on the performance. Moreover, we optimize the controller with soft robustness which also has a very large impact on the expected value of commitment holding in average. In our case studies, we have demonstrated the impact of hard and soft robustness on the expected case and the worst-case performance.

Chapter 6

Specification and Optimal Synthesis of Run-time Enforcement Shields

A system/controller 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 specified here by a QDDC formula $REQ(I,O)$, 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 6.1 on page 153.

In this chapter, we give a method for logical specification of shields using QDDC formulas. We show how our tool DCSynth introduced in Chapter 4, implementing soft requirement guided synthesis can be used for automatic synthesis of shields from a given specification. We then logically formulate several notions of shields on our proposed framework. We give experimental results showing the performance of our shield synthesis tool with a comparison to previous work.

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 of deviation explored in the literature [15, 69, 110, 109]. In their pioneering work, Bloem *et al.* [15] 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**. Konighofer *et al.* [69] have proposed some variants of the k-stabilizing shield 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** HDC using the formulas of an interval temporal logic QDDC. With its counting constructs and interval based modalities, the logic 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.* [110, 109] 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*.

As our second main contribution, we generalize Wu's 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. We call such a shield as *H-Optimally deviation minimizing*. This is a powerful optimization and 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 using our tool DCSynth introduced in Chapter 4. We give an experimental evaluation of the performance of our DCSynth tool for synthesizing shields and compare it with some previously reported studies in the literature. We also compare the performance of various shields by measuring the extent of deviation of the shield output from the SSE output. Towards this, we use two measures of the shield performance.

– We compute the *probability of deviation* in long run using method discussed in Section 4.3.5. 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. A model checking technique implemented in a previously available tool CTLDC [85] allows us to compute this worst case latency.

6.1 Framework for Specification and Synthesis of Run-time Enforcement Shields

For a system with sporadic errors (SSE) over input-output alphabet $\Sigma = 2^I \times 2^O$ (with I and O denoting the set of input propositions and output propositions respectively), let the **correctness requirement** be given as a QDDC formula $REQ(I,O)$, which is also over input-output propositions (I,O). SSE may fail to meet the requirement at some of the points in a behaviour (ii,oo). (The reader may recall Definition 15 in Section 4.1 and its following two paragraphs for the notation.) A **run-time enforcement shield** for the a given SSE and the correctness requirement REQ(I,O) is a Mealy machine with input-output alphabet $\Sigma' = 2^{I \cup O} \times 2^{O'}$, where O' is the output propositions for the shield and $I \cup O$ are the inputs to the shield. Thus, O' denotes the updated outputs computed by the shield for the corresponding system output O. See Figure 6.1 for the diagrametic representation. 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.

In this section, we give a logical framework for specifying various shields

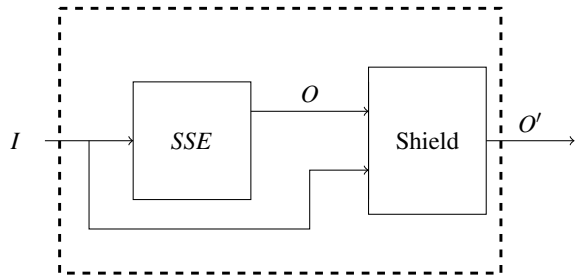


Figure 6.1: Run-time Enforcement Shield.

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).

6.1.1 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 38). 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 6.1.4, 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 6.1 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 38. This allows us to modularize the specification into components REQ and HDC.

$$HShield \;=\; REQ(I,O') \wedge HDC(SSEOK, Deviation) \ll INDDEF \qquad (6.1)$$

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 20). This supervisor can be termed as *shield-supervisor without deviation minimization*, denoted by $MPS(REQ, HDC)$.

6.1.2 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 sec-

tion. Quantitative optimization techniques from Markov Decision Processes can be used. (Stochasticity 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 (See *Generalized DCSynth specification* in Section 4.2.4). 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.

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 $O = \{o_1, \ldots, o_r\}$ having r output propositions, consider the DCSynth soft-requirement

$$Hamming(O, O') = \langle (true^\frown \langle o_1 = o_1' \rangle) : 1, \ \ldots \ , (true^\frown \langle o_r = o_r' \rangle) : 1 \rangle \qquad (6.2)$$

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 r is the number of output propositions and h is the hamming distance between $oo[k]$ and $oo'[k]$, where $oo[k]$ and $oo'[k]$ are vectors of r bits. 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 $MPHOS(REQ, HDC, H)$.

6.1.3 Determinization

It may be noted that both the shield-supervisors $MPS(REQ, HDC)$ as well as $MPHOS(REQ, HDC, H)$ 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 Theorem 34).

As given in section 4.2.3, the user specifies a preference ordering *ord* on the shield outputs $2^{O'}$. A deterministic controller is obtained by retaining only the highest ordered output from the non-deterministic choice of outputs offered by the supervisor. We denote the obtained shields (**deterministic controllers**) as $Det_{ord}(MPS(REQ, HDC))$ 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.

6.1.4 Variety of Hard Deviation Constraints and Shield-Types

In Table 6.1 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 6.1: 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, whereas 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 shield of Wu *et al.* [110] 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 experiments.

– 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.* [69]. By a variation of this, the k-stabilizing shield with fail-safe state [69] can also be specified.

– 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 6.1).

6.2 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 6.2.1. In Section 6.2.2 we define some performance measurement metrics for shields and we use these to compare various shield types.

6.2.1 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 [110]. The results are given in Table 6.2.

Table 6.2: 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 comparison, results for k-stabilizing shield synthesis and Burst-error shield synthesis are reproduced directly from Wu *et al.* [110].

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

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.*[108] were used in place of QDDC formulas for uniformity. For a comparison 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.* [110]. 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 favourably with the results reported in literature [110], 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[15, 110].

6.2.2 Comparison between various shield notions

For comparing the performance of shields synthesized with different shield types, we use 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 in Section 4.3.3.

The expected value of a shield S being in a non-deviating state over long runs can be denoted 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 [85] 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 $MAXLEN([[SSEOK \ \&\& \ Deviation]],S)$.

Experiments and Findings:

We can use the *expected value of deviation* and the *worst case burst-deviation latency*, for comparing the shields obtained using various shield-types defined in Section 6.1.4. We synthesized various shields for the correctness requirement $\varphi_{until}(n)$ given in Example 45 with $n=5$ and the input-output propositions $(\{r\},\{p,q\})$. The output propositions of synthesized shield are $\{p',q'\}$.

Example 45. *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*

Table 6.3: Shield Synthesis for the formula $\varphi_{until}(5)$ of Example 45 for shield types defined in Table 6.1 and their Performance comparison. The *expected value of non-deviation in long run* and *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		

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 6.2 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 6.2,* $\sigma, [5,9] \models until(p,q,3)$ *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 ]]`.

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

Figure 6.2: Example behaviour for $\phi_{until}(3)$

For each shield type V_i given in Table 6.1, the shields V_i_NoDM and V_i_DM were synthesized. 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 as outlined in the last paragraph of Section 4.2.3.

Table 6.3 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

6.1.4 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 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-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 as seen in Section 5.2.1.

6.3 Discussion

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.* [15], where the notion of k-stabilizing shield (with a synthesis algorithm) was proposed. This was further enhanced by Konighofer *et al.* [69]. Extension of shield synthesis to liveness properties was also been explored by them. Wu *et al.* [110, 109] 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. 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 using our tool DC-Synth. Using the proposed technique, we have specified the k-stabilizing shield of Konighofer *et al.* [69], the burst shield of Wu *et al.* [110, 109], 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.* [69] as well as Ehlers and Topku [39] 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.

Chapter 7

Architecture Centric Analysis of Non-Functional Requirements

The success of a HIS not only depends on its functional correctness, but also equally depends on the guarantees it provides w.r.t. the non-functional requirements. One of the most important non-functional properties is *dependability*, which must be assessed for any HIS. An accepted approach to ensure high level of dependability requirements associated with HIS is to follow design methodology guided by the relevant national standards like AERB/SG/D-10 [2], AERB/SG/D-25 [3][1] and international standards such as IEC-61508 [59], IEEE 603 [60], IEEE 7-4.3.2 [61], IAEA [58] etc. These standards [59] recommend the use of formal model based development and rigorous validation. Moreover, the assessment of such requirements should be precise, and needs to be carried-out early in the design life cycle to minimize the adverse consequences of design errors (if any and manifested later).

[1] Applicable to Nuclear Power Plants in India

The application of probabilistic automata based techniques for the analysis of non-functional properties has been recommended [59] and studied in the past. There are tools based on formal methods to assist in such analysis (e.g. probabilistic model checkers [73]) and these are shown to be useful for analysis of properties such as dependability [60]. But these tools have major limitation when the state space associated with the system model is very large. This constrains the applicability of these tools to small systems only.

A large/complex HIS is organized as an assembly of components. The *dependability* of a large HIS is determined by its architecture. This is because, the overall system dependability is a function of the dependability parameters (viz., safety, availability and reliability) of its sub-components. Hence, architecture plays an important role in achieving the required level of system dependability.

Traditionally, conformance to dependability requirement is mostly substantiated by manual arguments, mainly due to the lack of formal semantics for artefacts produced during architectural design. Any deficiency in the architectural design, if found later in the development life cycle, is likely to be expensive in terms of both cost and time.

We propose an architecture centric approach, where the model of a system architecture consisting of its sub-components is represented in Architectural Analysis & Design Language (AADL) [44] using OSATE tool [43]. Each component is annexed with a probabilistic automaton which incorporates various operational and fault states of the constituent component(s) along with probabilistic transitions among these states. Such a probabilistic automaton is called the *fault model* of the component. Similarly, a fault model can also be associated with a system. A tool has been developed to translate the fault model into Discrete Time

Markov Chain (DTMC) model. This DTMC model can be analyzed for various dependability properties like *Probability of Failure on Demand* (PFD), *Spurious Operation Probability* (SOP) etc. In this work, the probabilistic model checker PRISM [73] is used as back-end analysis engine for dependability analysis.

The fault model of a system/composite component can be obtained by taking product of fault models of its sub-component. However, a major challenge in the application of model checking technique is infamous *state space explosion* problem [57, 31, 29], which makes it impractical to analyze the systems of industrial interest.

Overcoming the state space explosion problem in real-world applications is one of the main contributions of this part of the work, where existing techniques are adapted to perform the dependability analysis of large HIS with *hierarchical* architecture, by making the analysis compositional. Thus, given a hierarchical architecture model of a system and the *Target Dependability Attributes* (TDA), the objective is to estimate the *Component Level Specifications* (CLS) of all the atomic components in the architecture that achieves the required TDA. Here, TDA specifies the targeted values of PFD and SOP, and the CLS are the values of failure rates, repair rate and diagnostics coverage for each component.

Compositional top down analysis consists of decomposing the monolithic analysis into steps involving simpler analysis. We categorize the architectural components either as an *atomic* or a *composite* one. *Atomic components* are the basic building blocks of a system architecture and are not divided further, whereas *composite components* are constituted using the existing atomic or composite components. We start with the system under consideration as a composite component and determine the CLS(s) of its sub-component(s) that achieve the required TDA of

the system/composite component. The sub-components are assumed to be atomic by abstracting the hierarchy of components/sub-components below it. If any of the sub-component was a composite component, then the obtained CLS for that component is converted into its TDA using a lookup table. Using this TDA, the sub-component is further analyzed to get the CLS(s) of its sub-component(s). The process is repeated for each of the sub-components, till we obtain the CLS of all the atomic components in the hierarchical architecture.

The compositional analysis methodology proposed in this work is demonstrated with the help of a case study for assessing the dependability of a few I&C safety systems of a Large Pressurized Water Reactor (PWR), which fall under the category of HIS. Following are the main contributions of this part of the work:

1. We present a compositional top down analysis methodology to analyze large (hierarchical) HIS (Section 7.2.1). Compositional analysis also allows to reuse the analysis results, when the same component is being used in multiple system architectures.

   We also extend the OSATE framework for automatic translation of *hierarchical* architecture model into corresponding PRISM DTMC model (Section 7.2.2).

2. Case study of two large HIS has been carried out for assessing the dependability using the proposed method (Section 7.3).

3. Rationalization of proposed architecture for achieving target dependability attributes by comparing it with alternative architectures (Section 7.3.4).

```
device Sensor                    // Declaration of a device
  features                       // Declaration of Ports
    output: out data port;
    ...
end Sensor;
device implementation Sensor.impl
   annex EMV2 {**                //Fault behaviour description
    use types ErrorLibrary;
    use behavior ThreeStateModel;
    properties
    OccurrenceDistribution =>[ProbabilityValue =>Pr ...] applies to Failure;
  **};
end Sensor.impl;
```

Figure 7.1: Example architecture description of an atomic component and associated Error Model in OSATE using AADL

7.1 Preliminaries

7.1.1 System Architecture Modeling in AADL

The Architecture Analysis & Design Language (AADL) [44] is a modelling language defined by the Society of Automotive Engineers (SAE), for specifying the architecture of a system. AADL can be used to model component based architecture along with interaction between the components and their execution behaviour. This language facilitates architectural description of large HIS in a hierarchical manner. In this thesis, we will describe the important features of AADL with the help of examples.

```
system SensingModule
    ...
end SensingModule;
system implementation SensingModule.impl
  subcomponents                //Instances of constituent components
    SN : device Sensor.impl;
    CU : ComparisionUnit.impl
  annex EMV2 {**               //Fault behaviour description
    use types ErrorLibrary;
    use behavior ThreeStateModel;
    composite error behavior
    states  //State of composite component defined using its constituents
      [CU.Failed_USUD or (CU.Operational and SN.Failed_USUD)]->Failed_USUD;
      [CU.Failed_Safe or (CU.Operational and SN.Failed_Safe)] ->Failed_Safe;
  **};
end SensingModule.impl;
```

Figure 7.2: Example architecture description of a compositional component and associated Error Model in OSATE(AADL)

Architectural components in AADL can be categorized as either *atomic* or *composite*. *Atomic components* are the basic building blocks of a system architecture and are not divided further, whereas *composite components* are constituted using the existing atomic or composite components, which describe the system hierarchy. An example of AADL architectural specification of an atomic component *Sensor* having a single output port is shown in Figure 7.1. The keyword *device* specifies that *Sensor* is a hardware device and the interface of this com-

ponent for data and control flow is given in the *features* section. A composite component *SensingModule* built using a *Sensor*(SN) and a *comparison unit*(CU) is shown in Figure 7.2. This component is designated as a *system* to indicate that it is a distinct and self-sufficient unit within the architecture, built by integration of other hardware, software and system components. The *subcomponents* section in the model specifies the instances of other constituent components used. We will use these architectural components in our case study presented in section 7.3.

AADL is an extensible language for modelling the architecture of a system, which provides the flexibility to extend the basic language by introducing sub-languages. The Error Model [99] is one such sub-language, which is used to formally specify the error model of a system/component in terms of its states and the events under which the components undergo transition among these states. For dependability analysis, we specify the *error model* of a component to describe its *anticipated fault model*. Therefore, the terms *error model* and *fault model* may be used interchangeably in rest of this chapter.

AADL Error Model

The error model of a AADL component is described by a state machine consisting of all possible states of the component and transitions among the states. The transitions are triggered on the occurrences of error events. An error event denotes the occurrence of a fault in a component. We associate an error model to each component of the system architectural model.

The syntax of error model is described with the help of an example. Note that any error model should be specified within the construct **annex EMV2** {** $\cdots$ **}. The atomic component *Sensor*, shown in Figure 7.1, is associated with

error behaviour ThreeStateModel

events

λ_2 : error event ; $//Rate\ in\ hr^{-1}$

λ_1 : error event ; $//Rate\ in\ hr^{-1}$

μ : error event (Repair); $//Rate\ in\ hr^{-1}$

$\frac{1}{T_{proof}}$: error event (Surveillance); $//Rate\ in\ hr^{-1}$

states

Operational : initial state;

Failed_Safe : state;

Failed_USUD : state;

transitions

t1 : Operational $-[\lambda_1]->$ Failed_Safe;

t2 : Operational $-[\lambda_2]->$ Failed_USUD;

t3 : Failed_Safe $-[\mu]->$ Operational;

t4 : Failed_USUD $-[\frac{1}{T_{proof}}]->$ Failed_Safe;

$\overbrace{Name}$ $\overbrace{Source\ State}$ $\overbrace{Event\ Rate}$ $\overbrace{Destination\ State}$

end behavior ;

Figure 7.3: Example Error Model (ThreeStateModel)

an error model denoted as *ThreeStateModel* (specified using the keyword *use behavior*), which is presented in Figure 7.3. This error model has three states viz., *Operational*, *Failed_Safe* and *Failed_USUD*, where the initial state is *Operational*. While the component remains in the *Operational* state if it is working as expected, a transition to *Failed_Safe* (or *Failed_USUD*) state will take place in the event of any safe failure (or unsafe-undetectable failure) [59]. A transition between two states represents the rate of the corresponding event – *failure* or *repair*. The nota-

tions used in this error model are described in the Table 7.1.

The error model of a composite component is described in terms of its constituent components. Example of an error model of a composite component *SensingModule*, in terms of its sub-components *Sensor* and *ComparisonUnit*, is shown in Figure 7.2. This example shows that the current state of *SensingModule* is derived by the current states of its sub-components.

During our analysis, the steady state probabilities of component being in the *Failed_Safe* or *Failed_USUD* state are used to derive the dependability attributes of the system. We assume that all the architectural components presented have an associated error model described in Figure 7.3.

7.1.2 Dependability Analysis using AADL error model

For estimating the system dependability, many quantitative analysis techniques, such as reliability block diagram, fault tree analysis, Markov analysis etc., are reported in literature. Rouvroye and Brombacher [98] showed that Markov analysis covers most of the aspects of quantitative safety evaluation (except uncertainty analysis) while taking into account the effect of redundancy, common cause failures, self-diagnostics and on-line/off-line test & repair. We adapted the Markov analysis technique and propose a DTMC based methodology, which makes application of quantitative dependability analysis feasible for large HIS.

Here, we explain the notations used in our error model, which are also summarized in Table 7.1. Cumulative failure rate of a system/component λ can be categorized into λ_S and λ_{US} representing safe and unsafe failure rates, respectively. However, a fraction of unsafe failures is detectable using self-diagnostics, which is denoted by *Diagnostic coverage (DC)*. Based on the self-diagnostics,

Table 7.1: Notations Used

Symbol	Description
λ	Cumulative Failure rate of system/component (hr^{-1})
λ_S	Failure rate due to safe failures (hr^{-1})
λ_{US}	Failure rate due to unsafe failures (hr^{-1})
λ_{USD}	Failure rate due to unsafe detectable failures (hr^{-1})
λ_{USUD}	Failure rate due to unsafe undetectable failures (hr^{-1})
λ_1	Failure rate due to safe or unsafe detectable failures (hr^{-1})
λ_2	Failure rate due to unsafe undetectable failures (hr^{-1})
DC	Fraction of detectable failures using self-diagnostic feature
$MTTR$	Mean Time to Repair (hrs)
μ	Repair Rate (hr^{-1}), which is inverse of MTTR
T_{proof}	Interval between two successive surveillance tests (hrs)

unsafe failures are further categorized as unsafe detectable (USD) failures and unsafe undetectable (USUD) failures. These failure rates are defined as follows.

$$\lambda_{USD} = (DC) \times \lambda_{US} \tag{7.1}$$

$$\lambda_{USUD} = (1 - DC) \times \lambda_{US} \tag{7.2}$$

We consider the *USD failure* also *as safe failure* because, operator can drive the system into a safe state on the basis of diagnostic information. Hence λ_1, the total rate for safe failure in the system, is derived by equation 7.3 and λ_2, the rate for unsafe undetectable failure in the system, is represented by equation 7.4.

$$\lambda_1 = \lambda_S + \lambda_{USD} \tag{7.3}$$

$$\lambda_2 = \lambda_{USUD} \tag{7.4}$$

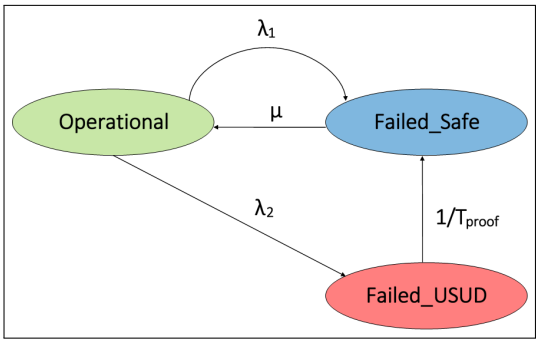


Figure 7.4: Generic Error model (Markov) of the Architectural Components

It can be observed that our error model example (Figure 7.3) has a one-to-one mapping with the generic error model of a component shown in Figure 7.4. A component can undergo a transition from *Operational* to *Failed_Safe* state on the occurrence of an event *Failure_Safe* with a failure rate represented by λ_1, and to *Failed_USUD* on the event of *Failure_USUD* with a failure rate represented by λ_2. Since the component is repairable, it can also undergo a transition from *Failed_Safe* to *Operational* state on the event *Repair* with a rate represented by μ. Further, USUD failure can be detected during surveillance test. Therefore, a component can undergo a transition from *Failed_USUD* to *Failed_Safe* on the event *surveillance*. The rate of this transition is $1/T_{proof}$ for a given surveillance test period (T_{proof}) [71].

For the purpose of dependability analysis, the error model of a component is specified using the parameters λ, DC, μ and T_{proof}. We refer to these parameters collectively as Component Level Specification (CLS).

7.1.3 PRISM Model Checker

PRISM [73, 72] is a probabilistic model checking tool for formal modeling and analysis of systems that exhibit random or probabilistic behavior. PRISM supports the analysis of DTMC and CTMC models. We use DTMC to formally represent the error model of a system, which is subsequently used to perform quantitative analysis for dependability, as discussed in section 7.2. The DTMC model is analyzed with respect to the specification of a formal property given in Probabilistic Computational Tree Logic (PCTL). PCTL allows the specification of temporal probabilistic properties.

An example of PCTL properties, used in this work, is presented here.

– $S =? [F\ SystemFailed]$: This specification demands that the model checker shall compute the steady state probability (S) of a system eventually (F) going into the failed state.

Refer to [73] for syntax and semantics of Prism language and PCTL.

7.2 Model Based Dependability Analysis of Safety Systems

In system engineering, the primary attributes considered for quantitative dependability analysis are safety, availability and reliability. Additional attributes viz., maintainability and testability are also considered during system architecture design [58, 80].

As Computer based I&C safety systems are examples of HIS, we consider analysis of these systems to demonstrate our compositional analysis framework.

Safety systems generally have two modes of operation, viz., *standby* and *active* mode. The safety systems mostly operate in standby mode waiting for the demand for its actuation that may arise in the event of any identifiable and detectable unsafe state of the plant. During standby mode, the safety system is periodically tested and maintained. Once a demand occurs, the safety system operates in active mode for a predefined duration termed as the *required period*. Therefore, the dependability analysis on such systems, can be carried out in two phases, as proposed by Meshkat *e*t al. [77].

- In the first phase, system availability analysis is carried out for *standby* mode, and
- In the second phase, system reliability analysis is carried out for *active* mode of operation, where the system availability obtained in the first phase is used as initial state probability.

In this work the dependability analysis of *Computer Based I&C safety systems* is carried out. These systems are designed mainly for actuation/initiation of safety functions, e.g. reactor trip, actuation of emergency core cooling etc. Dependability of these systems is mainly indicated by their availability during standby mode of operation. This is because, the duration of active mode of operation is negligible involving only actuation/initiation of the demanded safety function. Therefore, we restrict our presentation only to availability (and safety) analysis as required in the first phase of the dependability analysis[2].

The dependability attributes from the point of view of the plant safety and

[2]It is acknowledged that conventional reliability analysis is necessary for safety systems, which require long term operation following its actuation.

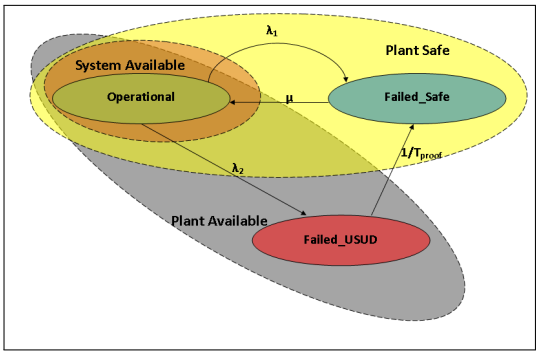


Figure 7.5: Safety and Availability Mapping on Generic Error model

availability are i) (safety) *system availability* and ii) *plant availability*. The correlation among these dependability attributes and the states of generic error model (shown in Figure 7.4) is presented with the help of the Figure 7.5. Here, the probability of system being in *Operational* state indicates the *system availability*. From the perspective of plant safety, the probability of system being in *Operational* or *Failed_Safe* states indicates the *safety* of the plant. However, from the perspective of plant availability, the probability of system being in *Operational* or *Failed_USUD* states indicates the *availability* of the plant. It may be noted here that system in *Failed_USUD* indicates unsafe state of the plant, though the plant is available till there is no demand for actuation of the safety system.

In order to quantify the system dependability attributes during standby mode, we use the following parameters.

– Probability of Failure on Demand (PFD): It is the probability of a system being in a state, where it fails to execute safety function when there is a demand. For error model shown in Figure 7.4, the steady state probability of the system/component being in *Failed_USUD* state represents the PFD

of that system/component.

– Spurious Operation Probability (SOP): It is the probability of a system being in a state, where system performs safety actions spuriously (without any demand) and makes plant unavailable. For error model shown in Figure 7.4, the steady state probability of the system/component being in *Failed_Safe* state represents the SOP of that system/component.

It may be observed from Figure 7.5 that both PFD and SOP should be minimized in order to increase the *system availability*, which in turn increases the *plant safety* and *plant availability*. Therefore, the target dependability requirements are specified in terms of these two attributes and referred to as Target Dependability Attributes (TDA).

It may be noted that as shown in our generic error model (Figure 7.4), the TDA of an atomic component is a function of its CLS, which is defined in Section 7.1.2. The *aim of dependability analysis* is to *estimate the CLS of all atomic components* of the system that *achieves the required TDA* of the system.

In this work, we adapted DTMC based technique to analyze the dependability of a system from its architectural model. We create a formal model of the system architecture annexed with error model in AADL [44]. The model is built hierarchically using bottom-up approach, by integrating the atomic or composite components in each step, as applicable. An open source tool OSATE [43] is used for graphical modelling of system/component architecture along with its error model. The extension of OSATE tool is used for translating the *hierarchical* architectural model to corresponding DTMC model (discussed in Section 7.2.2). The DTMC model is then analyzed using probabilistic model checker (PRISM) [73] for quantitative assessment of system dependability.

PRISM is a state-of-the-art tool which uses symbolic model checking technique. However, analyzing large systems of practical interest using this technique has its own limitations. If the analysis is performed monolithically for the complete system, it faces the challenge of state space explosion. The state space of the DTMC model grows prohibitively large in this case and it becomes infeasible to analyze the system architecture due to the requirement of huge computing resources. For example, state space of our case study of Reactor Trip System (Section 7.3) is 3^{24} ($\approx 10^{12}$) states for one safety train, and the complete system consist of four such trains. System with such a large state space could not be analyzed using symbolic model checking technique supported by PRISM and we observed that it quickly ran out of memory on a server having 250 GB of RAM. One of the main contributions of this work is to present a methodology for compositional analysis of such large HIS using limited computing resources.

7.2.1 Compositional Analysis Methodology

In order to overcome the state space explosion problem, we propose a *compositional dependability analysis* methodology. For a given hierarchical architecture model of a system and the required *TDA* the objective is to estimate the *CLS* of all the atomic components in the architecture, that achieves the required *TDA*.

We follow the compositional top down analysis, which consists of decomposing the monolithic analysis into steps involving simpler analysis that require lesser computing resource. We start with the system under consideration as a composite component and determine the CLS(s) of its sub-component(s) that achieve the required TDA of the system/composite component. The sub-components are assumed to be atomic by abstracting the hierarchy of components/sub-components

below it. If the sub-component is a composite component, then the obtained CLS is converted into its TDA based on a lookup table mapping CLS to the corresponding TDA values (See Table 7.2 and 7.3). Using this TDA, the sub-component is further analyzed to get the CLS(s) of its sub-component(s). The process is repeated for each of the sub-components, till we obtain the CLS of all the atomic components present in the hierarchical architecture of the system.

There can be multiple options of CLS to achieve the required TDA of a composite component. This happens because, the individual parameters of CLS can be experimented with different values, during the analysis. For example, the overall PFD of a component can remain unchanged, if increase (decrease) in failure rate (λ_2) is compensated by increasing (decreasing) the DC accordingly. Also, dependability of overall system will remain unchanged if an increase (decrease) in dependability of one component is compensated by decrease (increase) in dependability by other component(s) in the system. In such cases, the selection of a particular option is guided by the development cost and time of the component(s) involved.

Algorithm 1 gives the steps for compositional dependability analysis. It takes the system hierarchy H and a list denoted by cls_list to store the computed CLS(s) of the atomic component(s) as global inputs. Hierarchy H is used in step 4 of the algorithm to get the list of sub-components of a composite component. When the algorithm terminates, the cls_list will contain the CLS(s) of the atomic component(s), which will meet the required TDA.

The mapping of CLS to TDA for an atomic component is pre-computed and stored in a lookup table (Used in step 12 of Algorithm 1). The following assumptions are made for this mapping.

Table 7.2: PFD Mapping of a component with the error model in Figure 7.4

$\lambda_S = \lambda_{US} \xrightarrow{} DC \downarrow$	10^{-3}	10^{-4}	10^{-5}	10^{-6}
0	2.60×10^{-1}	3.47×10^{-2}	3.59×10^{-3}	3.60×10^{-4}
0.1	2.40×10^{-1}	3.13×10^{-2}	3.23×10^{-3}	3.24×10^{-4}
0.2	2.20×10^{-1}	2.79×10^{-2}	2.87×10^{-3}	2.88×10^{-4}
0.3	1.97×10^{-1}	2.45×10^{-2}	2.51×10^{-3}	2.52×10^{-4}
0.4	1.74×10^{-1}	2.11×10^{-2}	2.15×10^{-3}	2.16×10^{-4}
0.5	1.50×10^{-1}	1.76×10^{-2}	1.80×10^{-3}	1.80×10^{-4}
0.6	1.23×10^{-1}	1.42×10^{-2}	1.44×10^{-3}	1.44×10^{-4}
0.7	9.54×10^{-2}	1.07×10^{-2}	1.08×10^{-3}	1.08×10^{-4}
0.8	6.57×10^{-2}	7.13×10^{-3}	7.19×10^{-4}	7.20×10^{-5}
0.9	3.40×10^{-2}	3.58×10^{-3}	3.60×10^{-4}	3.60×10^{-5}

- $\lambda_s = \lambda_{US}$
- $MTTR = 24hrs (where\ \mu = \frac{1}{MTTR})$
- $T_{proof} = 720hrs$

Table 7.2 and 7.3 give the mapping of PFD and SOP, respectively for a given CLS. The lookup table is applicable for the components having Markov model depicted graphically in Figure 7.4. It may be recalled that the PFD and SOP of a component under consideration is obtained by computing the steady state probabilities of *DTMC model of the component* being in *Failed_USUD* and *Failed_Safe* states, respectively (using PRISM model checker).

Table 7.3: SOP Mapping of a component with the error model in Figure 7.4

$\lambda_S = \lambda_{US} \xrightarrow{} DC \downarrow$	10^{-3}	10^{-4}	10^{-5}	10^{-6}
0	1.73×10^{-2}	2.31×10^{-3}	2.39×10^{-4}	2.40×10^{-5}
0.1	1.78×10^{-2}	2.32×10^{-3}	2.39×10^{-4}	2.40×10^{-5}
0.2	1.83×10^{-2}	2.33×10^{-3}	2.39×10^{-4}	2.40×10^{-5}
0.3	1.88×10^{-2}	2.34×10^{-3}	2.39×10^{-4}	2.40×10^{-5}
0.4	1.94×10^{-2}	2.34×10^{-3}	2.39×10^{-4}	2.40×10^{-5}
0.5	1.99×10^{-2}	2.35×10^{-3}	2.40×10^{-4}	2.40×10^{-5}
0.6	2.05×10^{-2}	2.36×10^{-3}	2.40×10^{-4}	2.40×10^{-5}
0.7	2.12×10^{-2}	2.37×10^{-3}	2.40×10^{-4}	2.40×10^{-5}
0.8	2.19×10^{-2}	2.38×10^{-3}	2.40×10^{-4}	2.40×10^{-5}
0.9	2.26×10^{-2}	2.39×10^{-3}	2.40×10^{-4}	2.40×10^{-5}

We illustrate the compositional analysis methodology using an example shown in Figure 7.6. Figure 7.6 (a) shows the hierarchy of a system with two composite components, viz., $C1$ and $C2$. The remaining components, viz., $C3$, $C4$, $C5$ and $C6$ are atomic.

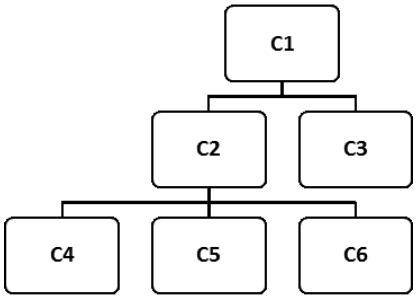
The dependability analysis of this system evaluates the CLS of all the atomic components that achieve the required TDA of the system $C1$. The monolithic analysis can be carried out by translating the error model of complete system into DTMC model, which will have 3^6 states (assuming each of the six components have the error model with 3 states as given in Figure 7.4).

In the proposed top-down compositional analysis methodology, we first carry-out the analysis of system $C1$, by assuming $C2$ as abstracted atomic component

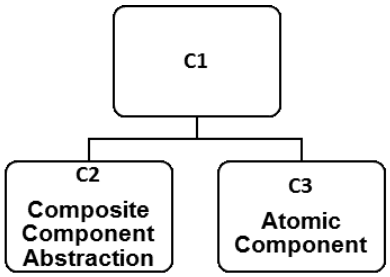
Algorithm 1: *DEPENDABILITY_ANALYSIS*

```
1  Global: H: system architecture hierarchy
2  Global: cls_list: List to contain CLS for all atomic components
   Input: s: component in system architecture hierarchy, initialized with the
          system under consideration
   Input: s_tda: Required TDA for s
   Output: updated cls_list, containing the CLS of each component to
           achieve the required TDA
3  if s is a composite component then
4      Explore all possible values of constituent components that achieve required TDA;
5      Select the suitable CLS values for each sub component from feasible choices;
6      foreach sub-component c ∈ s do
7          c_cls := assign the CLS of c from the selected CLSs;
8          if c is atomic component then
9              Append (c, c_cls) to cls_list;
10         end
11         else
12             c_tda := get the TDA corresponding to c_cls;
13             call DEPENDABILITY_ANALYSIS (c, c_tda);
14         end
15     end
16 end
```

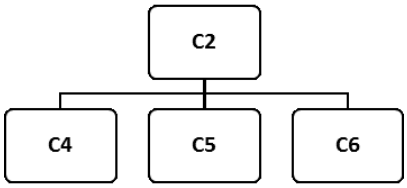
as shown in Figure 7.6 (b). This step involves a state space of only 3^3 states and computes the CLSs of $C2$ and $C3$, required to achieve system TDA. Note that there



(a) *System architecture*



(b) Abstraction of composite component



(c) *Architecture of component C*2

Figure 7.6: Example of Compositional Analysis

may be multiple possible *finite set* of user specified values for CLS of each component, given as a table (See Table 7.4 for example). All these possible combinations are explored and the list of feasible CLS values which achieves the required TDA is maintained. Further analysis is then carried out by choosing one the feasible combination from the list. As $C3$ is an atomic component so no further analysis is required for this component. However, $C2$ being a composite component, we have to compute the CLSs of its sub-components. In order to do so, we first get the TDA of $C2$ from its CLS (selected from the list of feasible CLS values) using the lookup tables (Table 7.2 and 7.3). We then carry-out the analysis of composite component $C2$ to determine the CLS of all its sub-components shown in Figure 7.6 (c). This step involves a state space of 3^4 states only and gives the required CLSs of $C4$, $C5$ and $C6$, which achieve the required TDA for $C2$. As all the sub-

components of $C2$ are atomic, the analysis is terminated here. Thus we get the CLSs of all the atomic components. If the selected CLS of composite component in the hierarchy (e.g. component $C2$ here) does not give any feasible CLS values for its sub-components then another CLS from the list of feasible choices is selected and analysis is carried out again for the hierarchy below that composite component (e.g. below $C2$ here).

It may be noted that maximum state space size involved in compositional analysis of this example is 3^4 as compared to 3^6 states involved in monolithic analysis. This example shows that *compositional methodology* can be *used effectively* for *systems* having large number of components with *deep hierarchical architecture*.

Analysis of the Algorithm

Algorithm 1 works on the DTMC model (in PRISM syntax) of the system under consideration. DTMC model is automatically obtained from the AADL error model using modular approach described in Section 7.2.2. The obtained DTMC model of system is also hierarchical and has one to one correspondence with the AADL error model i.e. each component (atomic or compositional) in AADL error model is translated into corresponding module in DTMC model. Given such a hierarchical DTMC model and required TDA, the algorithm provides a method to compute the values of CLS of all the atomic components that achieves the required TDA for the system. Following assessment provides insight in to our algorithm.

1. The *finite set* of CLS values for each sub-component in the hierarchy is specified by the user as a table. For example, CLS values for safe and unsafe failure rate are provides as order of magnitude (e.g failure rates are specified as 10^{-3} to 10^{-6} etc), and test interval, repair rate and diagnostic coverage

is decided based on failure rate, online testability and redundancy available in the system (See Table 7.4). At every level in the hierarchy all possible combinations are explored and the list of feasible CLS values which achieve the required TDA is constructed. The TDA of a composite component for specific values of CLSs of its constituent components is computed using the PRISM model checker and the CLS values which achieve the required TDA is pushed into the list of feasible CLS values.

Further analysis down the hierarchy is then carried out by choosing one of the feasible combination of CLS values from the list as indicated in step 5 of the algorithm. The selected CLS for composite component is then converted to the corresponding TDA and analysis is carried out to obtain the feasible CLS values of its sub-components. If the TDA of composite component in the hierarchy can not be satisfied by any CLS values for its sub-components then another CLS from for the composite component from the list of feasible choices is selected and analysis is carried out again for the hierarchy below that composite component. To find the list of all feasible CLS values at any level in hierarchy we use PRISM tool (Step 4 in algorithm). PRISM allows to provide the user specified CLS values as lower and upper range. Tool also allows to specify the step length to be considered for varying the CLS values of each constituent component while exploring the given range (See Table 7.4).

2. There is a possibility that more than one combination of CLS exists that achieve required TDA. In such a case, a particular option is selected manually by considering the cost, maintainability and development time of the components. The algorithm does not prescribe any specific approach to

be followed. However, if there does not exist any desirable combination of CLSs of constituent component, then required TDA is declared as *not achievable* with the architecture model under consideration. In such a case, system architecture model has to be re-engineered.

3. Termination of algorithm: The input to the algorithm is a hierarchical DTMC model corresponding to the AADL model **H**, which can be represented as a Tree. Each internal node in this tree represents a composite component and each terminal node represent an atomic component in the system. The children of an internal node represent the sub-components of a composite component identified by that node.

   It can be seen from the Algorithm 1 that it traverses the tree corresponding to hierarchical model **H** in depth first manner, therefore the recursion at step 13 will terminate. It is assumed that any system of practical interest consist of finite number of components and user can assign finite set of possible values for CLS of each component. Therefore, it is possible to explore all combinations of CLS of each sub-component and hence *foreach* loop at step 6 will also terminate, which indicates that the algorithm will always terminate. Also, when the algorithm terminates the list of CLS values of atomic component will surely achieve the required TDA for system i.e. this algorithm is sound. As discussed in point 1, we will keep on exercising alternate CLS values from the list of feasible CLS for composite component at every level in the hierarchy till we get CLS value assignment for each atomic component that achieves required TDA. Hence, algorithm will always find such assignment if it exist.

7.2.2 Automatic Translation of AADL Error model to DTMC model in PRISM

An open source software tool OSATE (an eclipse based framework) provides an editor for AADL models along with the several APIs for parsing and annotation of these models. Being an eclipse based framework, OSATE facilitates extension for any specific analysis requirements through eclipse plug-in mechanism. We used this mechanism to extend the OSATE framework for translation of *hierarchical* architecture model into a PRISM DTMC model.

Our translation scheme systematically translates each component in the architectural hierarchy, which can either be an atomic component or a composite component. For simplicity of exposition, we present an informal translation scheme using examples involving one *atomic* component and one *composite* component, as follows.

Translation of Atomic Component

The error model of atomic component, represented by a state machine having transitions annotated with its rate, is translated into PRISM language syntax. Consider the example of the translation of error model of an atomic component *Sensor*(SN), which uses the *ThreeStateModel* shown in figure 7.3. This error model has three states, viz., *Operational* (initial state), *Failed_Safe* and *Failed_USUD* with four transitions. The translated DTMC model is shown in Figure 7.7.

PRISM models facilitates *formula* definition (similar to macro definition), which comprises a name and an associated PRISM expression. The example of a formula definition can be observed in Figure 7.7, where the Sensor component

formula SN_is_Operational = SN_state=0;

formula SN_is_Failed_USUD = SN_state=1;

formula SN_is_Failed_Safe = SN_state=2;

module SN

SN_state: [0 .. 2] init 0;

[ ] SN_state=0 $\rightarrow \lambda_1$: (SN_state'=2) + $(1-\lambda_1)$:(SN_state'=0);

[ ] SN_state=2 $\rightarrow \mu$: (SN_state'=0) + $(1-\mu)$: (SN_state'=2);

[ ] SN_state=0 $\rightarrow \lambda_2$: (SN_state'=1) + $(1-\lambda_2)$: (SN_state'=0);

[ ] SN_state=1 $\rightarrow \frac{1}{T_{proof}}$: (SN_state'=2) + $(1-\frac{1}{T_{proof}})$: (SN_state'=1);

endmodule

Figure 7.7: Translation of error model of atomic components to PRISM model

is defined to be in operational state (*SN_is_Operational*) if the value of the enumerated variable *SN_state* is equal to 0. Similarly, $SN\_state = 1$ and $SN\_state = 2$ correspond to the Sensor component states *Failed_USUD* and *Failed_Safe*, respectively. The initial value of *SN_state* variable is 0, which indicates that the initial state of Sensor component is assumed to be *Operational*.

The state transition of the AADL error model is translated into PRISM module, which contains two parts: its state variables (say *SV*) and the *commands*. The variables represent the possible states and the commands correspond to the transitions i.e. the way in which the state changes over time. The current state of the system is represented by the state variable *SV* and the next state is represented by

SV'. Every command in the PRISM module has the following syntax.

$$[\,]SV = S1 \rightarrow R2 : SV' = S2 + R3 : SV' = S3 + \cdots + Rn : SV' = Sn \tag{7.5}$$

The command in Expression 7.5 above indicates that state of a component changes from $S1$ to $S2$ with a transition rate $R2$, from $S1$ to $S3$ with transition rate $R3$ and so on. The translated PRISM specification of the *ThreeStateModel* error model is presented in Figure 7.7. It can be observed from Figure 7.7 that the command

$$[\,]SN\_state = 0 \rightarrow \lambda_1 : (SN\_state' = 2) + (1 - \lambda_1) : (SN\_state' = 0)$$

shows that the component *SN* takes transition from *Operational* to *Failed_Safe* with a rate of λ_1 and remains in *Operational* state with a rate of $1 - \lambda_1$, where *SN_state* is its state variable. The PRISM commands for the rest of the transitions in *ThreeStateModel* are also shown in Figure 7.7.

The symbols used for transition probability (e.g λ_1, λ_2, μ, T_{proof}) are translated as symbolic constants in the PRISM model. During the experiments we can assign the range of values to these symbolic constants and plot the results to find out the CLS (i.e. the values of λ_1, λ_2, μ, T_{proof}), which will meet the required TDA.

Translation of Composite Component

The error model for a composite component defines current states of a composite component in terms of the current states of its sub-components. The translated PRISM model of a composite component is derived by the logical combination of formula defined in the translation of its sub-components. An example of the error

```
formula SM_is_Failed_USUD = CU_is_Failed_USUD |
                            (CU_is_Operational & SN_is_Failed_USUD);

formula SM_is_Failed_Safe = CU_is_Failed_Safe |
                            (CU_is_Operational & SN_is_Failed_Safe);
```

Figure 7.8: Translation of error model of composite components to PRISM model

model of a composite component *SensingModule* (*SM*) is shown in Figure 7.2. It shows that *SM* is comprised of the sub-components *Sensor (SN)* and *ComparisionUnit (CU)*.

The translated DTMC model corresponding to the error model specified in Figure 7.2 is shown in Figure 7.8. It can be observed from the translated model that the state of *SensingModule* is *Failed_USUD*, if the *ComparisionUnit* is in *Failed_USUD* state ***or*** the *Sensor* is in *Failed_USUD* state ***with*** *ComparisionUnit* in *Operational* state. Similarly, the state of *SensingModule* is *Failed_Safe* if the *ComparisionUnit* is in *Failed_Safe* state ***or*** the *Sensor* is in *Failed_Safe* state ***with*** *ComparisionUnit* in *Operational* state.

7.3 Case Studies

This section demonstrates the system dependability analysis in our framework for proposed architectures of I&C safety systems for a Pressurized Water Reactor (PWR) as case studies. The quantitative analysis of two systems, viz., Reactor Trip System (RTS) and Engineered Safety Feature Actuation System (ESFAS), was carried out using compositional analysis methodology. The analysis of al-

ternative architectures was also carried out in order to arrive at the most suitable architecture providing desirable dependability.

Based on the design basis requirements of these safety systems [59], the required TDAs for both systems are as follows:

- Probability of Failure on Demand (PFD) $\approx 10^{-5}$
- Spurious Operation Probability (SOP) $\approx 10^{-5}$

7.3.1 Reactor Trip System of a PWR

Proposed Architecture

Reactor Trip System (RTS) is a safety system that monitors the safety parameters and initiates reactor trips. The RTS consists of four redundant and independent safety trains. Each safety train consist of safety sensors, two diverse RPS controllers and two reactor trip breakers (RTBs). The RPS controllers are considered as diverse, because they receive signals, which are functionally diverse. Each RPS controller interfaces with its safety sensors and the RTBs. Architecture of RTS is shown in Figure 7.9.

A RPS controller acquires and processes the input signals and generates *parameter trip* signals by comparing the individual trip parameter values with their respective set-points, in the comparison unit (CU). We define *parameter trip* as the trip signal corresponding to a single parameter in any safety train of RTS. A RPS controller communicates the *parameter trip* information to the corresponding controllers of the three other redundant RTS safety trains using point to point (P2P) data communication link in a *2-out-of-4 (2/4)* configuration.

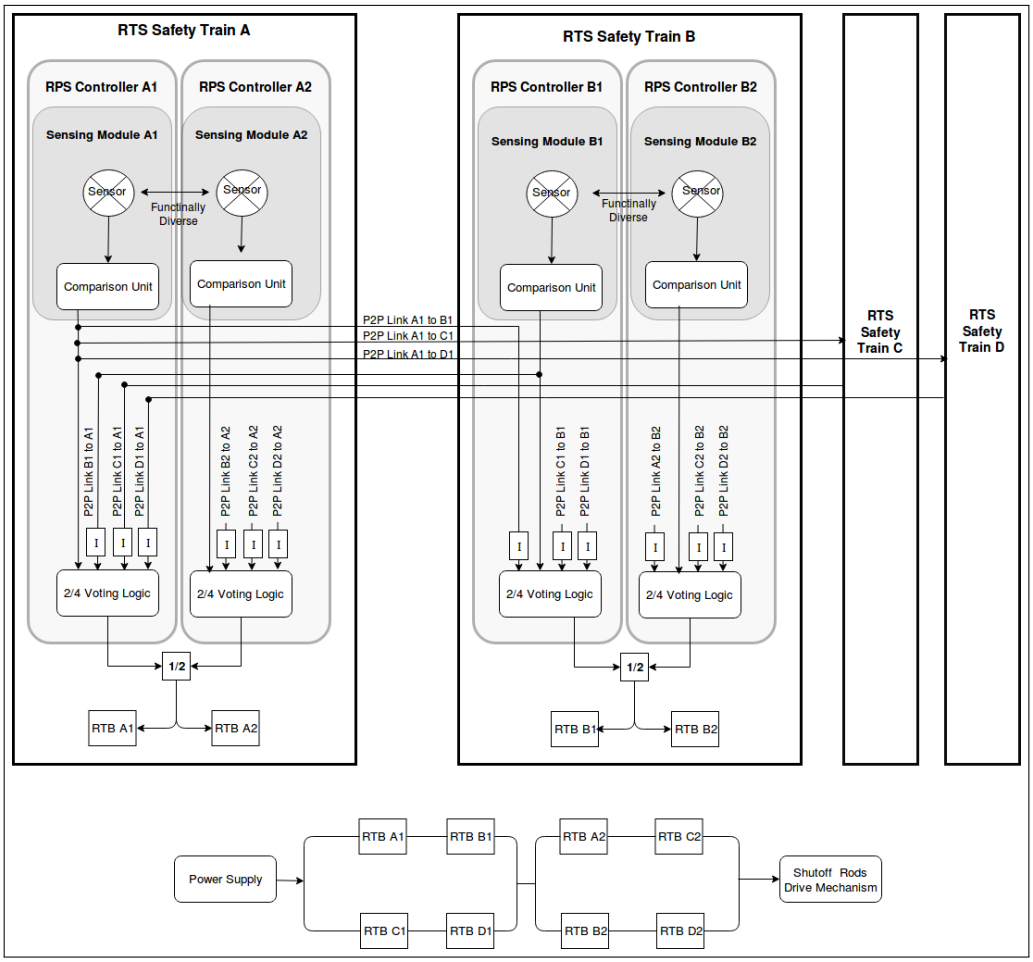


Figure 7.9: Architecture of Reactor Trip System

A *controller level reactor trip* signal on a particular parameter is generated by a diverse controller. This is done based on the *2-out-of-4 (2/4)* voting logic on the *parameter trip* signals generated by the corresponding controllers of the four trains. A *train level reactor trip* signal is generated by applying *1-out-of-2 (1/2)* voting logic between the *controller level reactor trips* generated by the diverse controllers.

A *train level reactor trip* actuates (opens) two RTBs associated with the particular train. The RTBs of all the four RTS trains are configured to implement a *2/4* voting logic as shown in Figure 7.9. According to this logic, opening of RTBs of at least two RTS trains will generate the *final reactor trip* signal. This case study assesses the dependability of RTS described above, with respect to its reactor trip function.

For simplicity of exposition, the following assumptions were made for dependability analysis of the RTS:

- All redundant trains are identical and independent of each other.
- CLS of both the diverse controllers of a RTS trains are same.
- All failure rates are constant with respect to time.
- Surveillance test is perfect, i.e. it detects all USUD failures.
- The system is maintainable during standby mode and the repair rate μ is identical for all trains and constant with respect to time.
- Voting logics are failure free: The voting logic (*1/2* and *2/4*) components used in the model are implemented either using software or using passive hardware components. Hence their failure rates are assumed to be 0.
- Effect of common cause failure is not considered in this analysis.
- When the system is in standby mode, demand for safety action can occur at any time with equal likelihood.
- The standby mode can last for a long time, therefore we have considered steady-state probabilities for computation of SOP and PFD in our analysis.

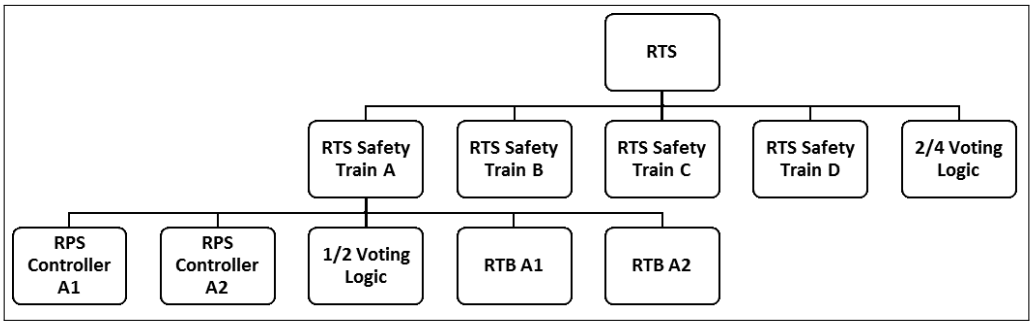


Figure 7.10: Abstract hierarchical model of Reactor Trip System by considering RPS controller as atomic component

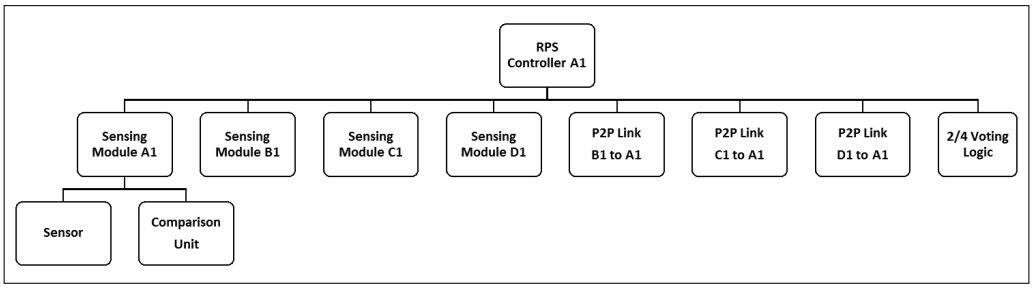


Figure 7.11: Hierarchical model of RPS Controller

Architectural Model

The RTS is modeled using AADL language in OSATE framework. System architecture is developed hierarchically using bottom-up approach i.e. the atomic components of the system are defined first and then these atomic components are gradually integrated to build the large hierarchical system/sub-system.

The total number of states in this model is of the order of 3^{96} ($\approx 10^{48}$), which makes a probabilistic analyse of this model infeasible even using reasonably high-end resources (Intel Xeon E5 Processor with 250 GB of RAM).

To overcome this difficulty, we apply *compositional analysis* methodology, by

decomposing the RTS model into multiple hierarchical models as follows.

1. First model, termed as *abstract RTS* model, is the abstract RTS architectural model, where RPS controller is abstracted as an *atomic component*. Abstract RTS model has four RTS safety train and *2/4* voting logic. Each RTS safety train consists of two *RPS controllers*, *1/2* voting logic module (for the controller outputs) and two *RTBs*. This model is shown in Figure 7.10.

2. Second model, termed as *refined RPS controller* model, is the refinement of the RPS controller, which consist of four *sensing modules*, three *P2P link* and *2/4* voting logic. This model is shown in Figure 7.11.

The above two models have state spaces of 3^{16} and 3^{11} respectively, which is much lesser than the overall state space (3^{96}) and thus it could be analyzed using compositional analysis methodology with the available computing resources.

It may be recalled from section 7.1.1 that for dependability analysis, we annotate each component with the error model (Figure 7.4). The transition rates for all the atomic components (sensor, CU, P2P link and RTB) are given as an input to the back-end probabilistic analysis engine PRISM. For each composite component, the current state is defined by the state of its sub-components. Therefore, there is no transition rate associated explicitly with a composite component.

Now, we define the error model of each composite components present in the architectural hierarchy with the help of the following functions.

– isOperational(Component *C*): Returns true if component *C* is *operational*.

– isFailedSafe(Component *C*): Returns true if component *C* is *Failed_Safe*.

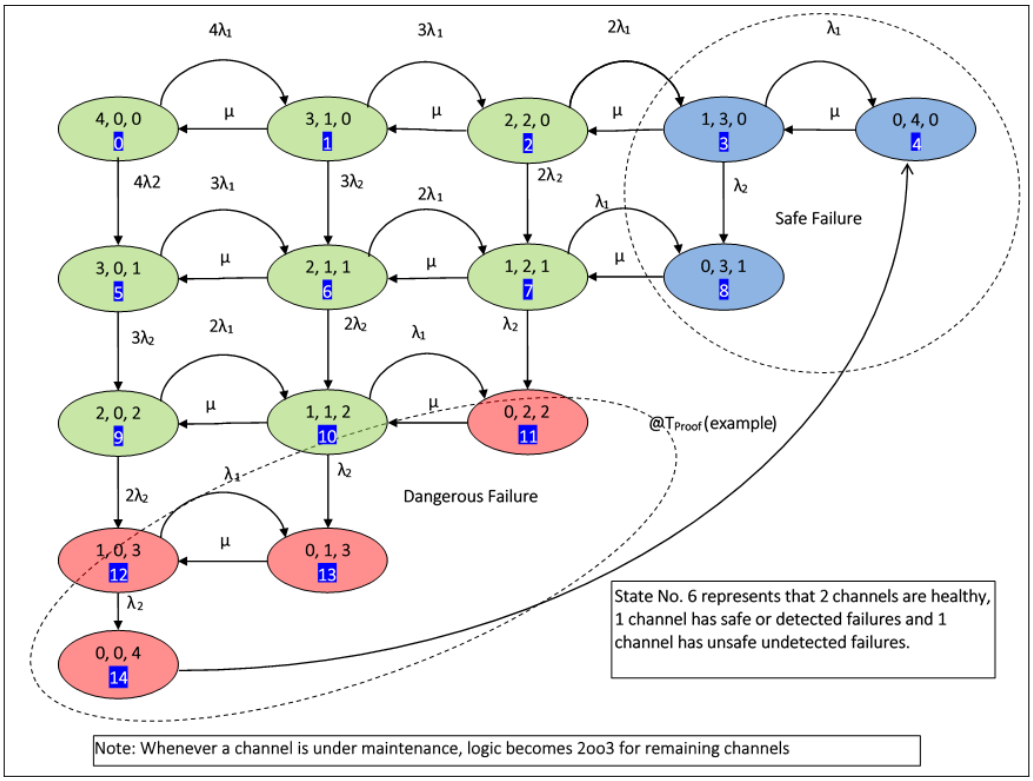


Figure 7.12: Markov model for *2/4* Voting Logic

– isFailedUSUD(Component *C*): Returns true if component *C* is *Failed_USUD*.

– nOperational(Component *C*): Returns the number of *C* components, which are in *operational* state.

– nFailedSafe(Component *C*): Returns the number of *C* components, which are in *Failed_Safe* state.

– nFailedUSUD(Component *C*): Returns the number of *C* components, which are in *Failed_USUD* state.

Error Model of the RTS system

RTS generates the *final reactor trip* based on the *2/4* voting logic. *2/4* voting logic gets inputs from four redundant trains and initiates safety action if at least two out of the four redundant trains demand for it. [64] gives the formal description of *2/4* voting logic and associated Markov model, which is reproduced here for ready reference.

7.3.2 Error Model of 2/4 Voting Logic

2/4 voting logic gets inputs from four redundant trains and initiates safety action if at least two out of four redundant trains demand for it.

The Markov model depicting *2/4* voting logic, used for analysis of RTS and ESFAS is shown in Figure 7.12. Each state in this model is represented by a 3-tuple (x, y, z), where x represents number of healthy trains, y represents number of trains having safe failures, and z represents number of trains having unsafe undetectable failure (USUD). Each state is also assigned a unique state number. Description of transitions is given as follows.

Transitions for safe failures: A healthy train may have safe failure with failure rate of λ_1 (Refer Equation 7.3). For example, the transition rate from state 0 to state 1 is $4\lambda_1$ representing safe failure in any one of the four healthy trains. In case of safe failure of a channel, it is bypassed and considered for maintenance with repair rate μ.

Transitions for unsafe undetectable failure: A healthy channel can also fail with unsafe undetectable failure with failure rate of λ_2 (Refer Equation 7.4). For example, the transition rate from state 0 to state 5 is $4\lambda_2$. All dangerous undetected

failures shall be detected in surveillance test, which is conducted periodically with T_{proof} interval.

Transition for repair of channel: If any state has at least one channel with safe failure, that channel is considered for maintenance with repair rate μ. For example, the transition rate from state 1 to state 0 is μ.

Transition for surveillance test: If any state has at least one channel with unsafe-undetected failure, that channel will remain in that state until next surveillance test, which happens after every T_{proof} time period. As surveillance test is assumed to be perfect, all USUD failures will be detected.

In *2/4* state space (Figure 7.12), states $10, 11, 12, 13$ and 14 represent that the system is in unsafe undetectable failure state, where system shall not be able to initiate safety action on demand. Steady sate probability of system being in these states has been calculated to estimate the PFD. Similarly, states $3, 4$ and 8 represents that the system is in safe failure states. Steady sate probability of system being in these states has been calculated to estimate the SOP.

As per the Markov model of *2/4* voting logic, probability of RTS system failing to provide safety action on demand (PFD), is represented by RTS being in *Failed_USUD* state. The RTS shall be in *Failed_USUD* state (represented by $RTS\_Is\_Failed\_USUD$ in PRISM DTMC model) if following condition holds.

$$(nFailedUSUD(RTS\_Train) >= 3)\ OR$$
$$((nFailedUSUD(RTS\_Train) = 2\ AND$$
$$(nFailedSafe(RTS\_Train) >= 1))$$

The above condition states that RTS will be in *Failed_USUD* state if either atleast 3 RTS trains are in *Failed_USUD* state **or** at-least 1 RTS train is in *Failed_Safe* state in conjunction with exactly 2 RTS trains in *Failed_USUD* state.

The probability of spurious actuation i.e., SOP of RTS is represented by the probability of RTS being in *Failed_Safe* state. RTS shall be in *Failed_Safe* state (represented by $RTS\_Is\_Failed\_Safe$ in PRISM DTMC model), if following condition holds.

$$nFailedSafe(RTS\_Train) >= 3$$

The above condition states that the RTS system will be in *Failed_Safe* state, if at-least 3 RTS trains are in *Failed_Safe* state. It may be noted that this model is applicable for *2/4* voting logic based system, if one of the trains having safe failure(s) is allowed to be bypassed [64] for maintenance.

Error Model of the RTS Safety Train

RTS train generates *train level reactor trip* signal based on *1/2* voting of the outputs of the two RPS controllers. Thus, a RTS safety train shall be in *Failed_USUD* state (represented by $RTS\_Train\_Is\_Failed\_USUD$ in PRISM DTMC model), if

$$(nFailedUSUD(RTB) >= 1)\ OR$$
$$((nOperational(RTB) >= 1)\ AND$$
$$(nFailedUSUD(RPS\_Controller) = 2))$$

A RTS safety train shall be in *Failed_Safe* state (represented by $RTS\_Train\_Is\_Failed\_Safe$ in PRISM DTMC model) if

$$(nFailedSafe(RTB) = 2)\ OR$$
$$((nFailedUSUD(RTB) = 0)\ AND$$
$$(nFailedSafe(RPS\_Controller) >= 1))$$

Error Model of the RPS Controller

RPS Controller generates the *controller level reactor trip* signal based on the *2/4* voting logic. In accordance with the *2/4* voting logic, a RPS controller shall be in *Failed_USUD* state (represented by *RPS_Controller_Is_Failed_USUD* in PRISM DTMC model), if

$$(nFailedUSUD(Sensing\_ModuleOrP2P) >= 3)\ OR$$
$$((nFailedUSUD(Sensing\_ModuleOrP2P) = 2)\ AND$$
$$(nFailedSafe(Sensing\_ModuleOrP2P) >= 1))$$

A RPS controller shall be in *Failed_Safe* state (represented by *RPS_Controller_Is_Failed_Safe* in PRISM DTMC model), if

$$nFailedSafe(Sensing\_ModuleOrP2P) >= 3$$

Here, $nFailedUSUD(Sensing\_ModuleOrP2P)$ represents the number of Sensing Modules or their corresponding P2P links in *Failed_USUD* state.

Error Model of the Sensing Module

Sensing Module shall be in *Failed_USUD* state (represented by *Sensing_Module_Is_Failed_USUD* in PRISM DTMC model), if

$$isFailedUSUD(Comparison\_Unit)\ OR$$
$$(isOperational(Comparison\_Unit)\ AND$$
$$isFailedUSUD(Sensor))$$

Sensing Module shall be in *Failed_Safe* state (represented by *Sensing_Module_Is_Failed_Safe*

in PRISM DTMC model) if

$$isFailedSafe(Comparison\_Unit)\ OR$$
$$(isOperational(Comparison\_Unit)\ AND$$
$$isFailedSafe(Sensor))$$

Compositional Dependability Analysis of the Reactor Trip System

We perform the analysis of each of the two models viz., *abstract RTS* and *refined RPS controller* models described in Section 7.3.1 separately and integrate the results (refer Algorithm 1) to obtain the CLSs of all the atomic components that achieve the required TDA of the RTS.

Analysis of the Abstract RTS Model

In this step, we generated DTMC model from the abstract model of RTS (shown in Figure 7.10) and used PRISM model checker to determine the CLSs of RPS controller and RTB, such that it achieve the required PFD and SOP of RTS.

PFD is the probability of RTS being in Failed_USUD state. The PCTL specification to compute the steady state probability of RTS being in *Failed_USUD* state is

$$S =? [RTS\_Is\_Failed\_USUD]$$

Similarly, SOP is the probability of RTS being in Failed_Safe state. The PCTL specification to compute the steady state probability of RTS being in *Failed_Safe* state is

$$S =? [RTS\_Is\_Failed\_Safe]$$

We carried out the experiments and obtained the plots for PFD and SOP with respect to various values of DC, where $0 \leq DC \leq 0.9$. While doing so, parameters,

viz., failure rates for RPS controller and RTB, were varied with a step size of 10^{-1} as given in Table 7.4. The Error model of RTS does not depend on the failure rate of voting logic as it has been assumed to be 0. Based on the plant operating experience, the most likely configuration for MTTR $(1/\mu)$ and T_{proof} are 24 hours and 720 hours respectively, and hence fixed values of these parameters were used in our experiments. The PFD and SOP values between the two consecutive DC values are estimated by linear interpolation.

The plots obtained were analyzed and only the relevant plots that achieve desired TDA (PFD $< 10^{-5}$ and SOP $< 10^{-5}$) are shown in Figure 7.13 and 7.14. The results are summarized in Table 7.5 and results relevant in achieving the TDA are highlighted. Following are the important observations based on the experimental results (Table 7.5).

- To achieve target dependability for RTS, the RTB should be designed with failure rate of the order of 10^{-5}/hour with DC of at least 0.5. It was also observed that further decreasing the failure rate of RTB (making it more reliable) does not give any significant gain in PFD and SOP.
- RPS controller should be designed with failure rate in the range of 10^{-5}/hour to 10^{-4}/hour, if appropriate DC ($0.1 \leq DC \leq 0.5$) is provided.

Since RTB is an atomic component (refer Figure 7.10), no further analysis is required for this. Whereas, RPS controller being a composite component, its dependability analysis is to be further carried out to obtain the CLS of its sub-components. From the calculated CLS requirement (i.e. $\lambda_S = \lambda_{US} = 10^{-4}$ with $DC = 0.5$), we use Table 7.2 and 7.3 to obtain the TDA ($PFD = 1.76 \times 10^{-2}$ and $SOP = 2.35 \times 10^{-3}$) for RPS Controller. The calculated TDA is used in the next step for analyzing RPS controller.

Table 7.4: CLS values considered for analysis of Reactor Trip System

Parameter Name	Start Value	End Value	Step
λ_S for RTB	10^{-4}	10^{-6}	multiplied by 10^{-1}
λ_{US} for RTB	10^{-4}	10^{-6}	multiplied by 10^{-1}
λ_S for RPS Controller	10^{-3}	10^{-6}	multiplied by 10^{-1}
λ_{US} for RPS Controller	10^{-3}	10^{-6}	multiplied by 10^{-1}
λ_S for Sensor	10^{-3}	10^{-5}	multiplied by 10^{-1}
λ_{US} for Sensor	10^{-3}	10^{-5}	multiplied by 10^{-1}
λ_S for Comparison Unit	10^{-3}	10^{-6}	multiplied by 10^{-1}
λ_{US} for Comparison Unit	10^{-3}	10^{-6}	multiplied by 10^{-1}
λ_S for P2P link	10^{-3}	10^{-5}	multiplied by 10^{-1}
λ_{US} for P2P link	10^{-3}	10^{-5}	multiplied by 10^{-1}
DC	0	0.9	increased by 0.1
T_{proof}	720		
MTTR	24		

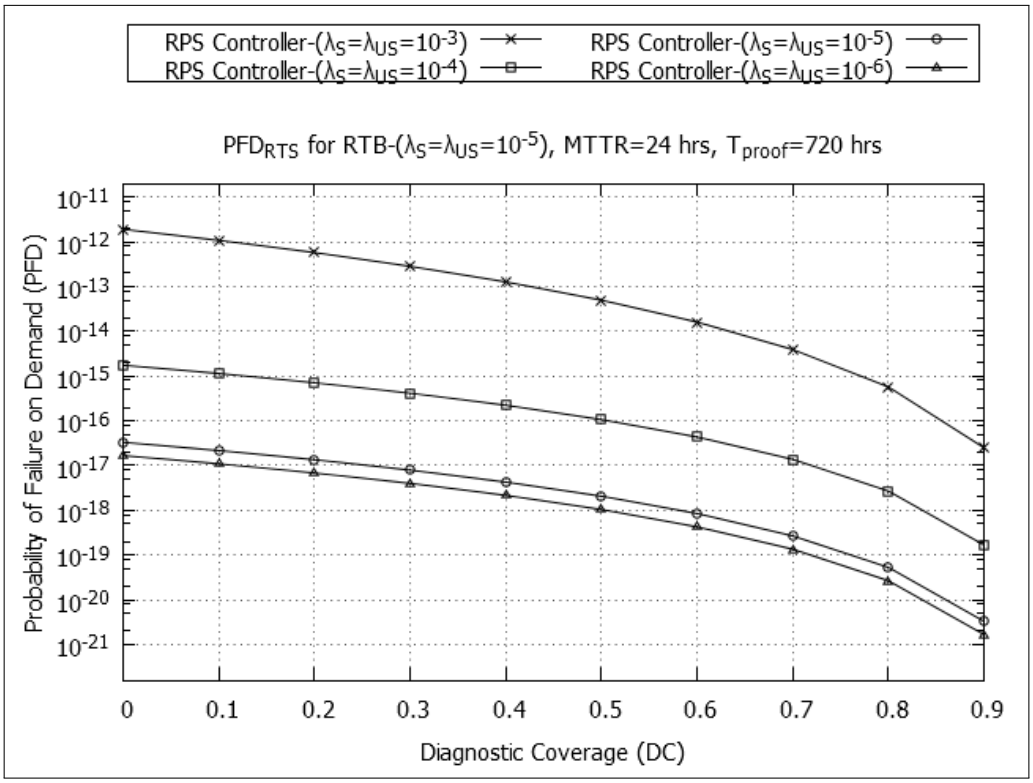


Figure 7.13: PFD of RTS with $\lambda_S = \lambda_{US} = 10^{-5}$ for RTB, MTTR=24 hrs and T_{proof} =720 hrs

Analysis of Refined RPS Controller model

In this step, we generated DTMC model from the *refined RPS Controller model* (shown in Figure 7.11) and used PRISM model checker to determine the CLSs of its sub-components, viz., Sensor, P2P link and CU, which achieve the required TDA ($PFD = 1.76 \times 10^{-2}$ and $SOP = 2.35 \times 10^{-3}$) for RPS controller.

The PCTL specification to determine the PFD and SOP for RPS controller is as follows.

$$S =? [RPS\_Controller\_Is\_Failed\_USUD]$$

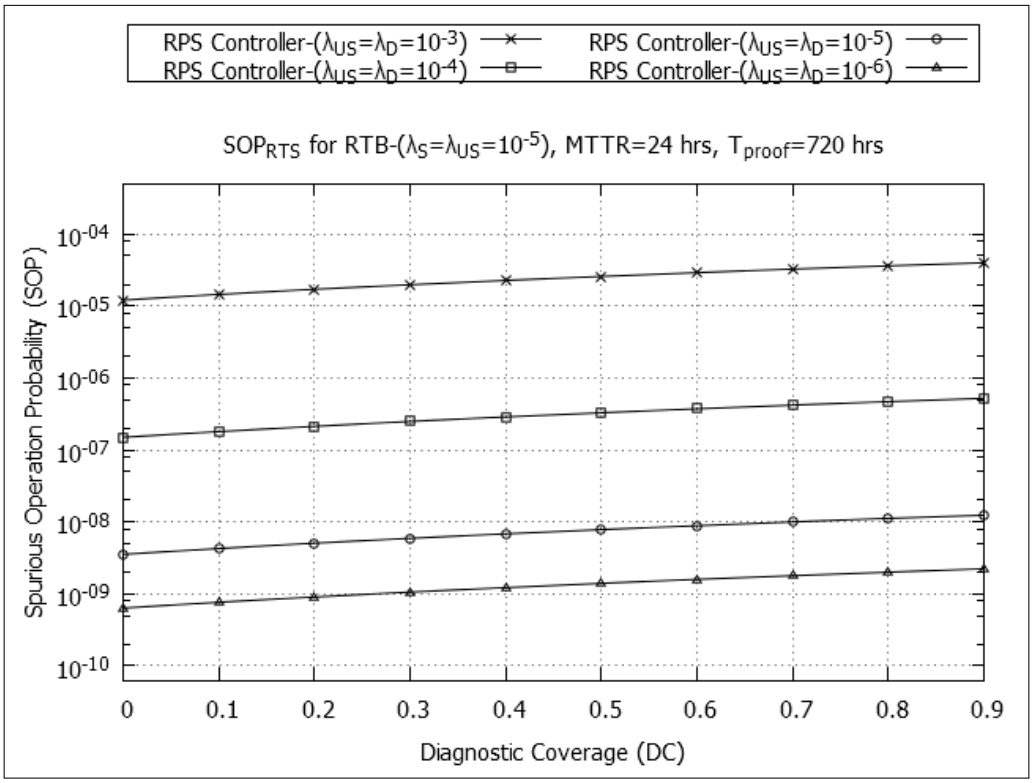


Figure 7.14: SOP of RTS with $\lambda_S = \lambda_{US} = 10^{-5}$ for RTB, MTTR=24 hrs and T_{proof} =720 hrs

$$S =? [RPS\_Controller\_Is\_Failed\_Safe]$$

Following the same process as used in the dependability analysis of RTS, we carried out the experiments and obtained the plots for PFD and SOP of RPS controller with respect to various values of DC, where $0 \leq DC \leq 0.9$. While doing so, parameters, viz., failure rates for Sensor, P2P data link and CU, were varied with a step size of 10^{-1} as given in Table 7.4. The values of MTTR $(1/\mu)$ and T_{proof} are kept constant at 24 hours and 720 hours, respectively.

The plots obtained were analyzed and the relevant results are highlighted in

Table 7.5: Experimental results for RTS architecture analysis

Sr. No.	RTS component failure rate (hr^{-1})		DC	MTTR (hrs)	T_{proof} (hrs)	PFD of RTS	SOP of RTS
	RPS Controller	RTB					
1	10^{-6}	10^{-4}	0.9	24	720	1.05×10^{-5}	1.61×10^{-7}
2	10^{-4}	10^{-5}	0.5	24	720	8.84×10^{-6}	2.76×10^{-6}
3	10^{-5}	10^{-5}	0.1	24	720	6.73×10^{-6}	3.02×10^{-9}
4	10^{-4}	10^{-6}	0	24	720	3.87×10^{-6}	2.54×10^{-6}
5	10^{-5}	10^{-6}	0	24	720	2.05×10^{-8}	3.12×10^{-9}

Table 7.6. Following important observation based on the experimental results can be made from Table 7.6.

- In order to achieve the desired TDA for RPS controller, the sensors, P2P data links and CU modules are required to be designed with failure rate of the order of 10^{-4}/hour with $DC = 0.2$.
- It was also observed that decreasing the failure rate of P2P data link further does not give any significant gain in dependability.

Result

Analysis results of the proposed RTS architecture (Figure 7.9) shows that in order to achieve the target dependability (PFD $\leq 10^{-5}$ and SOP $\leq 10^{-5}$), the following design requirements are to be met.

Table 7.6: Experimental results for RPS controller architecture analysis

Sr. No.	RPS component failure rate (hr^{-1})			DC	MTTR (hrs)	T_{proof} (hrs)	PFD of RPS Controller	SOP of RPS Controller
	Sensor	P2P Data Link	CU					
1	10^{-5}	10^{-5}	10^{-3}	≈0.80	24	720	1.0×10^{-2}	2.36×10^{-4}
2	10^{-4}	10^{-4}	10^{-4}	≈0.20	24	720	1.0×10^{-2}	5.5×10^{-6}
3	10^{-4}	10^{-5}	10^{-4}	0	24	720	1.0×10^{-2}	2.45×10^{-6}
4	10^{-5}	10^{-4}	10^{-4}	0	24	720	1.0×10^{-2}	1.67×10^{-6}
5	10^{-4}	10^{-4}	10^{-4}	≈0.70	24	720	1.0×10^{-3}	6.6×10^{-6}
6	10^{-4}	10^{-5}	10^{-4}	≈0.60	24	720	1.0×10^{-3}	2.93×10^{-6}
7	10^{-5}	10^{-5}	10^{-4}	≈0.20	24	720	1.0×10^{-3}	5.16×10^{-7}

i Failure rates of RTBs should be of $\leq 10^{-5}$/hour with at least 50% DC.

ii Failure rates of sensors, CUs and P2P Data Links are required to be $\leq 10^{-4}$/hour with at least 20% diagnostic coverage.

iii MTTR and T_{proof} are to be kept at most 24 hours and 720 hours respectively for all the components of RTS.

In this section we showed the application of proposed compositional analysis methodology for dependability analysis of large systems. We mainly focused on the *safety* and *availability* attributes as these are the most relevant attributes for dependability of computer based I&C systems as discussed in section 7.2.

The other dependability attributes such as *maintainability* and *testability* are mainly addressed by the qualitative analysis of system architecture. In the four-train architecture of RTS, any one of it can be taken out of service for maintenance and testing without violating *single failure criteria*. This is because, the system will be able to perform its safety functions using *2-out-of-3 (2/3)* voting logic with the remaining three trains. It may be noted that, the error model used for dependability analysis of RTS, incorporates this provision. In this context, it may also be noted that it is a common practice in I&C safety systems of NPP to make provisions for on-line periodic surveillance testing.

7.3.3 Engineered Safety Feature Actuation System of IPWR

ESFAS is a computer based I&C safety system, which senses the accident conditions and actuates emergency safety systems to mitigate the effect of Postulated Initiating Event (PIE), such as Loss of Coolant Accident (LOCA), a steam line break etc.

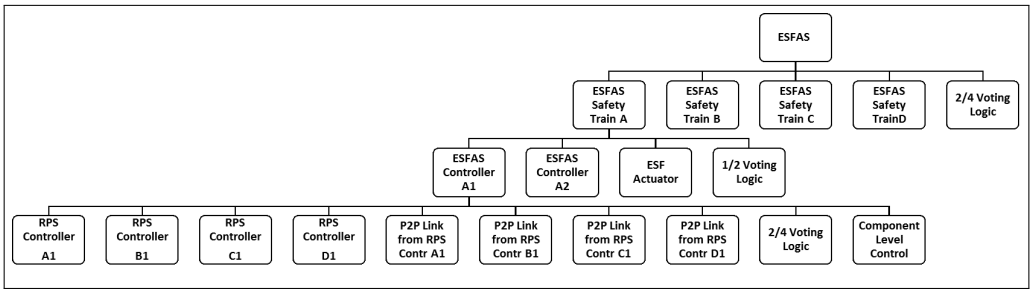


Figure 7.15: Hierarchical model of Engineered Safety Feature Actuation System

In the proposed architecture (Figure 7.15), ESFAS uses the sensors (for common parameters) and RPS controllers of RTS for acquisition of all its field inputs and their processing. It gives the advantage of i) lesser penetrations in reactor coolant system (RCS) due to shared sensors and ii) reduced hardware offering reduced cost & time towards installation, commissioning and maintenance. Using compositional analysis methodology, we could validate that the required TDA for both RPS and ESFAS can be achieved even with shared sensors and controllers.

In the hierarchical architecture of ESFAS, the RPS controllers, P2P Data Link (from RPS to ESFAS), Component Level Control (CLC) modules and ESF Actuators are considered as atomic component for dependability analysis of ESFAS.

Experimental Results

Based on the experimental results, following important observations were made.

– To achieve the target dependability of ESFAS, ESF actuators, CLCs, P2P Data Links and RPS controllers are required to be designed with the failure rates of $\leq 10^{-5}$/hour, $\leq 10^{-5}$/hour, $\leq 10^{-4}$/hour and $\leq 10^{-4}$/hour respectively, with DC value of at least 0.2.

– The MTTR and T_{proof} are to be kept at most 24 hours and 720 hours respectively for all the components of ESFAS.

– Based on the CLS values of sensors and controller obtained, we concluded that the *required TDA* of both RTS and ESFAS could be *achieved* with *shared controllers and sensors*.

7.3.4 Comparison with Alternate Architectures

We also used architecture analysis framework to assess the alternative architectures and their effect on the system dependability. For brevity, we will refer the architectures of RTS and ESFAS, as *RTSArch* and *ESFASArch* respectively.

– *RPS controller without local 2/4 voting logic for RTS:* An alternative architecture without local *2/4* voting logic (*Say RTSArch_NoLocalCoincidence*), was compared with the proposed architecture *RPSArch*.

It is observed that, SOP of *RTSArch_NoLocalCoincidence* is degraded considerably. For example, the SOP of *RTSArch* was found to be 2.76×10^{-6} as against 9.0×10^{-2} in case of *RPSArch_NoLocalCoincidence*. Note that we considered same sensors and controllers (failure rate of $10^{-4}/hour$) for this comparison. This justifies the train level local *2/4* voting logic in *RPSArch*.

– *ESFAS train with one controller:* An alternate architecture with only one controller in ESFAS train, referred to as *ESFASArch_SingleController*, was compared with *ESFASArch*.

It is observed that *ESFASArch_SingleController* provides better SOP 4.61×10^{-7} compared to *ESFASArch* (2.94×10^{-6}), but at the cost of increased

PFD 1.0×10^{-3}, which is not desirable. Further, in order to achieve the target PFD with *ESFASArch_SingleController*, the failure rate of ESFAS controller is required to be $\leq 10^{-5}/hour$, which is difficult to achieve.

Therefore, *ESFASArch* (two controllers in each train with *1/2* voting logic) is a better and feasible option to meet the required TDA.

7.4 Discussion

In this chapter we have proposed a compositional analysis methodology for architecture centric dependability analysis of large HIS. AADL is an architecture modeling language, which along with its error model annexure provides a powerful notation to model inter-component connection, fault in the component and their propagation and recovery. The framework has been successfully used in various kind of analysis such as safety, dependability and performance analysis [45, 19]. These analysis are typically based on automatic generation of fault trees from the architecture and using fault tree analysis methods [41, 19]. However, we translate the AADL error model (fault model) into DTMC model and use PRISM for dependability analysis, which allows us to incorporate and analyze the architecture having temporal error and recovery mechanism. Our main focus is to apply existing formal modeling and analysis technique to systems of practical interest by making analysis compositional, which has not been addressed in earlier work to the best of our knowledge.

AADL has also been used for automatic code generation from distributed models [75]. To apply formal analysis techniques to AADL models like model checking, several transformations to formal notations such as rewriting logic [81] or Timed abstract state machine [111] has been proposed. AADL also has annexes

to deal with other system level aspects such as timing, latency and schedulability analysis [101], which is not in the scope of our work. However, as a future work we are planning to use advanced features of error model annexure such as error propagation and component specific fault modeling to expand the applicability of architecture centric analysis to re-configurable or adaptive systems.

We have shown that, because of the associated state space explosion problem, monolithic approach does not scale for large systems occurring in real-world applications. In order to demonstrate the applicability and effectiveness of the proposed compositional dependability analysis methodology we carried out case studies involving quantitative dependability analysis of the proposed architectures of RTS and ESFAS of a PWR. The results of the case studies clearly demonstrate the following advantages of our compositional analysis methodology.

- It facilitates use of model checking techniques for dependability analysis of large systems to improve the rigor of the analysis, and
- Enables the re-use of analysis results, when architectural components are shared across multiple systems.

This analysis framework enabled us to

- validate the suitability of proposed architecture for achieving desired safety and availability attributes, which strengthens preliminary safety analysis.
- establish the required CLS (failure rate, diagnostic coverage, surveillance test interval etc.) for its subsequent use in system design.
- assess the alternative architectural considerations and its effect on the system dependability.

Bibliography

[1] "IEC 62531:2012(e) (IEEE std 1850-2010): Standard for property specification language (PSL)," *IEC*, pp. 1–184, June 2012.

[2] AERB, "AERB/NPP-PHWR/SG/D-10, AERB safety guide: Safety systems for pressurized heavy water reactors," 2005. [Online]. Available: https://aerb.gov.in/english/publications/codes-guides

[3] AERB, "AERB/NPP-PHWR/SG/D-25, AERB safety guide: Computer based systems of pressurized heavy water reactors," 2010. [Online]. Available: https://aerb.gov.in/english/publications/codes-guides

[4] J. F. Allen, "Maintaining knowledge about temporal intervals," in *Readings in Qualitative Reasoning About Physical Systems*. Morgan Kaufmann, 1990, pp. 361 – 372. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B978148321447450033X

[5] S. Almagor, U. Boker, and O. Kupferman, "Formally reasoning about quality," *JACM*, vol. 63, no. 3, pp. 24:1–24:56, 2016. [Online]. Available: http://doi.acm.org/10.1145/2875421

[6] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distributed Computing*, vol. 2, no. 3, pp. 117–126, 1987. [Online]. Available: http://dx.doi.org/10.1007/BF01782772

[7] R. Alur and T. A. Henzinger, "Real-time logics: complexity and expressiveness," in *LICS*, June 1990, pp. 390–401. [Online]. Available: http://dx.doi.org/10.1109/LICS.1990.113764

[8] N. Amla, E. A. Emerson, R. P. Kurshan, and K. S. Namjoshi, "Model checking synchronous timing diagrams," in *FMCAD*. Springer, 2000, pp. 320–335. [Online]. Available: http://dx.doi.org/10.1007/3-540-40922-X_18

[9] A. Babu and P. K. Pandya, "Chop expressions and discrete duration calculus," pp. 229–256, 2012. [Online]. Available: https://doi.org/10.1142/9789814271059_0008

[10] J. Barwise, "An introduction to first-order logic," in *Handbook of Mathematical Logic*. Elsevier, 1977, vol. 90, pp. 5 – 46. [Online]. Available: https://doi.org/10.1016/S0049-237X(08)71097-8

[11] R. E. Bellman, *Dynamic Programming*. Princeton Univ. Press, 1957.

[12] R. Bloem, K. Chatterjee, K. Greimel, T. A. Henzinger, G. Hofferek, B. Jobstmann, B. Könighofer, and R. Könighofer, "Synthesizing robust systems," *Acta Inf.*, vol. 51, no. 3-4, pp. 193–220, 2014. [Online]. Available: http://dx.doi.org/10.1007/s00236-013-0191-5

[13] R. Bloem, K. Chatterjee, T. A. Henzinger, and B. Jobstmann, "Better quality in synthesis through quantitative objectives," in *CAV*, 2009, pp. 140–156. [Online]. Available: https://doi.org/10.1007/978-3-642-02658-4_14

[14] R. Bloem, R. Ehlers, S. Jacobs, and R. Könighofer, "How to handle assumptions in synthesis," in *Proceedings 3rd Workshop on Synthesis, SYNT*, 2014, pp. 34–50. [Online]. Available: https://doi.org/10.4204/EPTCS.157.7

[15] R. Bloem, B. Könighofer, R. Könighofer, and C. Wang, "Shield synthesis: - runtime enforcement for reactive systems," in *TACAS*, ser. LNCS, C. Baier, Ed., vol. 9035. Springer, 2015, pp. 533–548. [Online]. Available: https://doi.org/10.1007/978-3-662-46681-0_51

[16] A. Bohy, V. Bruyère, E. Filiot, N. Jin, and J. Raskin, "Acacia+, a tool for LTL synthesis," in *CAV*, ser. LNCS, vol. 7358. Springer, 2012, pp. 652–657. [Online]. Available: https://doi.org/10.1007/978-3-642-31424-7_45

[17] A. Bohy, V. Bruyère, E. Filiot, and J.-F. Raskin, "Synthesis from LTL specifications with mean-payoff objectives," in *TACAS*. Springer, 2013, pp. 169–184. [Online]. Available: https://doi.org/10.1007/978-3-642-36742-7_12

[18] P. Bouyer, N. Markey, M. Randour, K. G. Larsen, and S. Laursen, "Average-energy games," *Acta Informatica*, vol. 55, no. 2, pp. 91–127, Mar 2018. [Online]. Available: https://doi.org/10.1007/s00236-016-0274-1

[19] M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri, "Safety, dependability and performance analysis of extended

aadl models," *The Computer Journal*, vol. 54, pp. 754–775, 2011. [Online]. Available: https://doi.org/10.1093/comjnl/bxq024

[20] J. R. Buchi and L. H. Landweber, "Solving sequential conditions by finite-state strategies," *Transactions of the American Mathematical Society*, vol. 138, pp. 295–311, 1969. [Online]. Available: http://doi.org/10.2307/1994916

[21] R. Büchi, "Weak second-order arithmetic and finite automata." Z. Math. Logik Grundlagen Math., 6:6692, 1960, pp. 66–92. [Online]. Available: https://doi.org/10.1002/malq.19600060105

[22] G. Chakravorty and P. K. Pandya, "Digitizing interval duration logic," in *CAV*, ser. LNCS, vol. 2725. Springer, 2003, pp. 167–179. [Online]. Available: https://doi.org/10.1007/978-3-540-45069-6_17

[23] Z. Chaochen and M. R. Hansen, *Duration Calculus - A Formal Approach to Real-Time Systems*, ser. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2004. [Online]. Available: https://doi.org/10.1007/978-3-662-06784-0

[24] Z. Chaochen, C. A. R. Hoare, and A. P. Ravn, "A calculus of durations," *Information Processing Letters*, vol. 40, no. 5, pp. 269–276, 1991. [Online]. Available: https://doi.org/10.1016/0020-0190(91)90122-X

[25] A. Chapyzhenka and J. Probell, "Wavedrom: Rendering beautiful waveforms from plain text," *Synopsys User Group*, 2016. [Online]. Available: http://wavedrom.com/images/SNUG2016_WaveDrom.pdf

[26] H. Chockler and K. Fisler, "Temporal modalities for concisely capturing timing diagrams," in *CHARME*, ser. LNCS, vol. 3725. Springer, 2005, pp. 176–190. [Online]. Available: http://dx.doi.org/10.1007/11560548_15

[27] A. Church, *Logic, arithmetic and automata*. International Congress of Mathematicians, 1962.

[28] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *CAV*, ser. LNCS, vol. 2404. Springer, 2002, pp. 359–364. [Online]. Available: https://doi.org/10.1007/3-540-45657-0_29

[29] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, *Progress on the State Explosion Problem in Model Checking*. Springer, 2001, pp. 176–194. [Online]. Available: https://doi.org/10.1007/3-540-44577-3_12

[30] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Logics of Programs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 52–71. [Online]. Available: https://doi.org/10.1007/BFb0025774

[31] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking*. MIT press, 1999.

[32] W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," *Formal Methods in System Design*, vol. 19, no. 1, pp. 45–80, 2001. [Online]. Available: http://dx.doi.org/10.1023/A:1011227529550

[33] J. Das, "Heterogeneous modeling of live sequence charts and statecharts," *M.Tech. Thesis, IIT Bombay*, 2014.

[34] L. M. de Moura, S. Owre, H. Rueß, J. M. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "Sal 2," in *CAV*, ser. LNCS, vol. 3114. Springer, 2004, pp. 496–500. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-27813-9_45

[35] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna, "Graphical specifications for concurrent software systems," in *ICSE, Melbourne, Australia, May 11-15, 1992.*, T. Montgomery, L. A. Clarke, and C. Ghezzi, Eds. ACM, 1992, pp. 214–224. [Online]. Available: http://doi.acm.org/10.1145/143062.143116

[36] X. C. Ding, M. Lazar, and C. Belta, "LTL receding horizon control for finite deterministic systems," *Automatica*, vol. 50, no. 2, pp. 399–408, 2014. [Online]. Available: https://doi.org/10.1016/j.automatica.2013.11.030

[37] D. D'Souza and M. Gopinathan, "Conflict-tolerant features," in *CAV*, ser. LNCS, vol. 5123. Springer, 2008, pp. 227–239. [Online]. Available: https://doi.org/10.1007/978-3-540-70545-1_22

[38] R. Ehlers, S. Lafortune, S. Tripakis, and M. Y. Vardi, "Supervisory control and reactive synthesis: a comparative introduction," *Discrete Event Dynamic Systems*, vol. 27, no. 2, pp. 209–260, Jun 2017. [Online]. Available: https://doi.org/10.1007/s10626-015-0223-0

[39] R. Ehlers and U. Topcu, "Resilience to intermittent assumption violations in reactive synthesis," in *HSCC*. New York, NY, USA: ACM, 2014, pp. 203–212. [Online]. Available: https://doi.org/10.1145/2562059.2562128

[40] C. Eisner and D. Fisman, "Temporal logic made practical," *Handbook of Model Checking. Springer (Expected 2016)*, 2016.

[41] M. Esteve, J. Katoen, V. Y. Nguyen, B. Postma, and Y. Yushtein, "Formal correctness, safety, dependability, and performance analysis of a satellite," in *ICSE*, 2012, pp. 1022–1031. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227118

[42] P. Faymonville, B. Finkbeiner, and L. Tentrup, "Bosy: An experimentation framework for bounded synthesis," in *CAV*, ser. LNCS, vol. 10427. Springer, 2017, pp. 325–332. [Online]. Available: https://doi.org/10.1007/978-3-319-63390-9_17

[43] P. Feiler, "Open source AADL tool environment (OSATE)," in *AADL Workshop, Paris*, 2004.

[44] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*, 1st ed. Addison-Wesley Professional, 2012.

[45] P. H. Feiler and A. Rugina, "Dependability modeling with the architecture analysis & design language (aadl)," Tech. Rep., 2007.

[46] K. Fisler, "Timing diagrams: Formalization and algorithmic verification," *Journal of Logic, Language and Information*, vol. 8, no. 3, pp. 323–361, 1999. [Online]. Available: http://dx.doi.org/10.1023/A:1008345113376

[47] K. Fisler, "Two-dimensional regular expressions for compositional bus protocols," in *FMCAD*. IEEE, 2007, pp. 154–157. [Online]. Available: http://dx.doi.org/10.1109/FAMCAD.2007.14

[48] L. Fix, *Fifteen Years of Formal Property Verification in Intel*, ser. LNCS. Springer, 2008, vol. 5000, pp. 139–144. [Online]. Available: https://doi.org/10.1007/978-3-540-69850-0_8

[49] M. Franceschet, M. de Rijke, and B. Schlingloff, "Hybrid logics on linear structures: Expressivity and complexity," in *TIME-ICTL*. IEEE, 2003, pp. 166–173. [Online]. Available: http://dx.doi.org/10.1109/TIME.2003.1214893

[50] M. Fränzle, "Synthesizing controllers from duration calculus," in *FTRTFT*, ser. LNCS, vol. 1135. Springer, 1996, pp. 168–187.

[51] M. Fränzle and M. Müller-Olm, *Compilation and Synthesis for Real-Time Embedded Controllers*, ser. LNCS. Springer, 1999, vol. 1710, pp. 256–287. [Online]. Available: https://doi.org/10.1007/3-540-48092-7_12

[52] E. Grädel, W. Thomas, and T. Wilke, Eds., *Automata Logics, and Infinite Games: A Guide to Current Research*, ser. LNCS. Springer, 2002, vol. 2500. [Online]. Available: https://doi.org/10.1007/3-540-36387-4

[53] Y. Gurevich and L. Harrington, "Trees, automata, and games," in *STOC*. ACM, 1982, pp. 60–65. [Online]. Available: http://doi.acm.org/10.1145/800070.802177

[54] D. Harel, “Statecharts: a visual formalism for complex systems,” *Science of Computer Programming*, vol. 8, no. 3, pp. 231 – 274, 1987. [Online]. Available: https://doi.org/10.1016/0167-6423(87)90035-9

[55] J. G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm, “Mona: Monadic second-order logic in practice,” in *TACAS*, ser. LNCS, vol. 1019. Springer, 1995, pp. 89–110. [Online]. Available: https://doi.org/10.1007/3-540-60630-0_5

[56] T. A. Henzinger and J. Sifakis, “The embedded systems design challenge,” in *FM*, ser. LNCS, vol. 4085. Springer, 2006, pp. 1–15. [Online]. Available: https://doi.org/10.1007/11813040_1

[57] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

[58] IAEA, *Dependability Assessment of Software for Safety Instrumentation and Control Systems at Nuclear Power Plants*, ser. IAEA Nuclear Energy Series. Vienna: International Atomic Energy Agency, 2018, no. NP-T-3.27. [Online]. Available: http://www-pub.iaea.org/books/IAEABooks/12232/

[59] IEC, “IEC Std-61508: Functional safety of electrical/electronic/ programmable electronic safety related systems,” 2010.

[60] IEEE, “Std-603: IEEE standard criteria for safety systems for nuclear power generating stations,” Nov 2009.

[61] IEEE, "IEEE standard criteria for programmable digital devices in safety systems of nuclear power generating stations," *IEEE Std 7-4.3.2-2016 (Revision of IEEE Std 7-4.3.2-2010)*, pp. 1–86, Aug 2016.

[62] S. Jacobs and et. al, "The 4th reactive synthesis competition (SYNTCOMP 2017): Benchmarks, participants & results," *CoRR*, vol. abs/1711.11439, 2017. [Online]. Available: http://arxiv.org/abs/1711.11439

[63] S. Jacobs and R. Bloem, "Parameterized synthesis," in *TACAS*. Springer, 2012, pp. 362–376. [Online]. Available: https://doi.org/10.1007/978-3-642-28756-5_25

[64] A. Kabra, G. Karmakar, M. Kumar, and P. P. Marathe, "Sensitivity analysis of safety system architectures," in *ICIC*. IEEE, May 2015, pp. 846–851. [Online]. Available: https://doi.org/10.1109/IIC.2015.7150860

[65] J. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen, "The ins and outs of the probabilistic model checker MRMC," *Performance Evaluation*, vol. 68, pp. 89–220, 2011. [Online]. Available: https://doi.org/10.1016/j.peva.2010.04.001

[66] N. Klarlund and A. Møller, *MONA Version 1.4 User Manual*, 2001. [Online]. Available: http://www.brics.dk/mona/mona14.pdf

[67] N. Klarlund, A. Møller, and M. I. Schwartzbach, "MONA implementation secrets," in *CIAA*, ser. LNCS, vol. 2088. Springer, 2001, pp. 182–194. [Online]. Available: https://doi.org/10.1007/3-540-44674-5_15

[68] U. Klein and A. Pnueli, "Revisiting synthesis of GR(1) specifications," in *HVC*, ser. LNCS, vol. 6504. Springer, 2010, pp. 161–181. [Online]. Available: https://doi.org/10.1007/978-3-642-19583-9_16

[69] B. Könighofer, M. Alshiekh, R. Bloem, L. Humphrey, R. Könighofer, U. Topcu, and C. Wang, "Shield synthesis," *FMSD*, vol. 51, no. 2, pp. 332–361, Nov 2017. [Online]. Available: https://doi.org/10.1007/s10703-017-0276-9

[70] S. N. Krishna and P. K. Pandya, "Modal strength reduction in quantified discrete duration calculus," in *FSTTCS*, ser. LNCS, vol. 3821. Springer, 2005, pp. 444–456. [Online]. Available: https://doi.org/10.1007/11590156_36

[71] M. Kumar, A. K. Verma, and A. Srividya, "Analyzing effect of demand rate on safety of systems with periodic proof-tests," *International Journal of Automation and Computing*, vol. 4, no. 4, pp. 335–341, 2007. [Online]. Available: http://dx.doi.org/10.1007/s11633-007-0335-6

[72] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *CAV*, ser. LNCS, vol. 6806. Springer, 2011, pp. 585–591. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_47

[73] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM: Probabilistic symbolic model checker," in *TOOLS*, ser. LNCS, vol. 2324. Springer, 2002, pp. 200–204. [Online]. Available: https://doi.org/10.1007/3-540-46029-2_13

[74] S. Lafortune, K. Rudie, and S. Tripakis, "Thirty years of the ramadge-wonham theory of supervisory control: A retrospective and future perspectives [conference reports]," *IEEE Control Systems Magazine*, vol. 38, no. 4, pp. 111–112, 2018. [Online]. Available: https://doi.org/10.1109/MCS.2018.2830083

[75] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues, "Ocarina : An environment for aadl models analysis and automatic code generation for high integrity applications," in *Reliable Software Technologies*, ser. LNCS, vol. 5570. Springer, 2009, pp. 237–250. [Online]. Available: https://doi.org/10.1007/978-3-642-01924-1_17

[76] R. Majumdar, E. Render, and P. Tabuada, "Robust discrete synthesis against unspecified disturbances," in *HSCC*. ACM, 2011, pp. 211–220. [Online]. Available: http://doi.acm.org/10.1145/1967701.1967732

[77] L. Meshkat, J. B. Dugan, and J. D. Andrews, "Dependability analysis of systems with on-demand and active failure modes, using dynamic fault trees," *IEEE Transactions on Reliability*, vol. 51, no. 2, pp. 240–251, 2002. [Online]. Available: https://doi.org/10.1109/TR.2002.1011531

[78] K. S. Namjoshi, "Symmetry and completeness in the analysis of parameterized systems," in *VMCAI*. Springer, 2007, pp. 299–313. [Online]. Available: https://doi.org/10.1007/978-3-540-69738-1_22

[79] D. Neider, A. Weinert, and M. Zimmermann, "Robust, expressive, and quantitative linear temporal logics: Pick any two for free," in

GANDALF. Open Publishing Association, 2019, pp. 1–16. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.305.1

[80] V. P. Nelson, "Fault-tolerant computing: Fundamental concepts," *Computer*, vol. 23, no. 7, pp. 19–25, 1990. [Online]. Available: https://dx.doi.org/10.1109/2.56849

[81] P. C. Ölveczky, A. Boronat, and J. Meseguer, "Formal semantics and analysis of behavioral aadl models in real-time maude," in *Formal Techniques for Distributed Systems*, ser. LNCS, vol. 6117. Springer, 2010, pp. 47–62. [Online]. Available: https://doi.org/10.1007/978-3-642-13464-7_5

[82] OMG, "OMG unified modeling language (omg uml) version 2.5.1," 2017. [Online]. Available: https://www.omg.org/spec/UML/

[83] P. K. Pandya, "Specifying and deciding quantified discrete-time duration calculus formulae using dcvalid," in *RTTOOLS (affiliated with CONCUR 2001)*, 2001.

[84] P. K. Pandya, "Model checking CTL*[DC]," in *TACAS*, ser. LNCS, vol. 2031. Springer, 2001, pp. 559–573. [Online]. Available: https://doi.org/10.1007/3-540-45319-9_38

[85] P. K. Pandya, "Finding extremal models of discrete duration calculus formulae using symbolic search," *Electronic Notes in Theoretical Computer Science*, vol. 128, no. 6, pp. 247 – 262, 2005. [Online]. Available: https://doi.org/10.1016/j.entcs.2005.04.015

[86] P. K. Pandya, Y. S. Ramakrishna, and R. K. Shyamasundar, "A Compositional semantics of Esterel in Duration Calculus," in *AMAST workshop on Real-time Systems: Models and Proofs*. Springer, 1995.

[87] N. Piterman, A. Pnueli, and Y. Sa'ar, "Synthesis of reactive(1) designs," in *VMCAI*, ser. LNCS, vol. 3855. Springer, 2006, pp. 364–380. [Online]. Available: https://doi.org/10.1007/11609773_24

[88] A. Pnueli, "The temporal logic of programs," in *SFCS*. IEEE, 1977, pp. 46–57. [Online]. Available: https://doi.org/10.1109/SFCS.1977.32

[89] A. Pnueli and R. Rosner, "On the synthesis of an asynchronous reactive module," in *ICALP*, ser. LNCS, vol. 372. Springer, 1989, pp. 652–671. [Online]. Available: https://doi.org/10.1007/BFb0035790

[90] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, 1st ed. John Wiley & Sons, Inc., 1994. [Online]. Available: https://doi.org/10.1002/9780470316887

[91] M. O. Rabin, *Automata on Infinite Objects and Church's Problem*. American Mathematical Society, 1972.

[92] P. Ramadge and W. Wonham, "Supervisory control of a class of discrete event processes," *SIAM Journal on Control and Optimization*, vol. 25, no. 1, pp. 206–230, 1987. [Online]. Available: https://doi.org/10.1137/0325013

[93] P. Ramadge and W. Wonham, "The Control of Discrete Event Systems," in *Proceedings of IEEE*, vol. 77, 1989, pp. 81–98. [Online]. Available: https://doi.org/10.1109/5.21072

[94] V. Raman, A. Donzé, D. Sadigh, R. M. Murray, and S. A. Seshia, "Reactive synthesis from signal temporal logic specifications," in *HSCC*. ACM, 2015, pp. 239–248. [Online]. Available: http://doi.acm.org/10.1145/2728606.2728628

[95] K. D. Rao, V. Gopika, V. S. Rao, H. Kushwaha, A. Verma, and A. Srividya, "Dynamic fault tree analysis using monte carlo simulation in probabilistic safety assessment," *Reliability Engineering & System Safety*, vol. 94, no. 4, pp. 872 – 883, 2009. [Online]. Available: https://doi.org/10.1016/j.ress.2008.09.007

[96] S. Riedweg and S. Pinchinat, "Quantified mu-calculus for control synthesis," in *MFCS*, ser. LNCS, vol. 2747. Springer, 2003, pp. 642–651. [Online]. Available: https://doi.org/10.1007/978-3-540-45138-9_58

[97] R. Rosner, *Modular Syntheis of Reactive Systems*. Ph.D. Thesis, Weizmann Institute of Science, Rehovot, Israel, 1992.

[98] J. Rouvroye and A. Brombacher, "New quantitative safety standards: different techniques, different results?" *Reliability Engineering and System Safety*, vol. 66, no. 2, pp. 121 – 125, 1999. [Online]. Available: https://doi.org/10.1016/S0951-8320(99)00028-9

[99] SAE, "SAE architecture analysis and design language (AADL) annex volume 1: Annex a: Graphical AADL notation," *Annex C: AADL Meta-Model and Interchange Formats, Annex D: Language Compliance and Application Program Interface Annex E: Error Model Annex, AS5506/1*, 2011.

[100] B. Sharma, P. K. Pandya, and S. Chakraborty, "Bounded validity checking of interval duration logic," in *TACAS*, ser. LNCS, vol. 3440. Springer, 2005, pp. 301–316. [Online]. Available: https://doi.org/10.1007/978-3-540-31980-1_20

[101] F. Singhoff and A. Plantec, "Aadl modeling and analysis of hierarchical schedulers," in *SIGAda*. ACM, 2007, pp. 41–50. [Online]. Available: https://doi.org/10.1145/1315580.1315593

[102] W. Thomas, "Automata on infinite objects," in *Formal Models and Semantics*. Elsevier, 1990, pp. 133 – 191. [Online]. Available: https://doi.org/10.1016/B978-0-444-88074-1.50009-3

[103] M. Y. Vardi and P. Wolper, "An automata-theoretic approach to automatic program verification (preliminary report)," in *LICS*. IEEE, 1986, pp. 332–344. [Online]. Available: http://hdl.handle.net/2268/116609

[104] A. Wakankar, P. K. Pandya, and R. M. Matteplackel, *DCSynth 1.0*, TIFR, Mumbai, 2018. [Online]. Available: http://www.tcs.tifr.res.in/~pandya/dcsynth/dcsynth.html

[105] WaveDrom, "Wavedrom user manual," 2016. [Online]. Available: http://wavedrom.com/tutorial.html

[106] E. M. Wolff, U. Topcu, and R. M. Murray, "Efficient reactive controller synthesis for a fragment of linear temporal logic," in *ICRA*. IEEE, 2013, pp. 5033–5040. [Online]. Available: http://dx.doi.org/10.1109/ICRA.2013.6631296

[107] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Receding horizon temporal logic planning," *IEEE Transactions on Automatic Control*, vol. 57, no. 11, pp. 2817–2830, 2012. [Online]. Available: https://doi.org/10.1109/TAC.2012.2195811

[108] M. Wu, "ishield2 synthesizer," 2016. [Online]. Available: https://bitbucket.org/mengwu/shield-synthesis/

[109] M. Wu, H. Zeng, C. Wang, and H. Yu, "Invited: Safety guard: Runtime enforcement for safety-critical cyber-physical systems," in *DAC*. ACM, June 2017, pp. 1–6. [Online]. Available: https://doi.org/10.1145/3061639.3072957

[110] M. Wu, H. Zeng, and C. Wang, "Synthesizing runtime enforcer of safety properties under burst error," in *NFM*, ser. LNCS, vol. 9690. Springer, 2016, pp. 65–81. [Online]. Available: https://doi.org/10.1007/978-3-319-40648-0_6

[111] Z. Yang, K. Hu, D. Ma, J.-P. Bodeveix, L. Pi, and J.-P. Talpin, "From aadl to timed abstract state machines: A verified model transformation," *Journal of Systems and Software*, vol. 93, pp. 42 – 68, 2014. [Online]. Available: https://doi.org/10.1016/j.jss.2014.02.058

Thesis Highlight

Name of the Student: Amol Wakankar
Name of the CI/OCC: BARC, Mumbai
Enrolment No.: ENGG01201204009

Thesis Title: Theories, Techniques and Tools for High Integrity Heterogeneous Embedded Systems
Discipline: Engineering Sciences **Date of viva voce:** 20/07/2020

The correctness guarantees entailed by the High Integrity Heterogeneous Embedded Systems, necessitate to incorporate formal methods in all phases of the design, spanning over *functional* as well as *non-functional* requirements. This thesis bridges the gap between visual (semi-formal) notation for requirement specification traditionally used by designers and the formal logic-based approach, by formalizing the requirements in an expressive logic (*QDDC*). The automatic synthesis of discrete controllers from such logic-based requirements is well studied in literature, but the *quality* of such controllers is still a major concern.

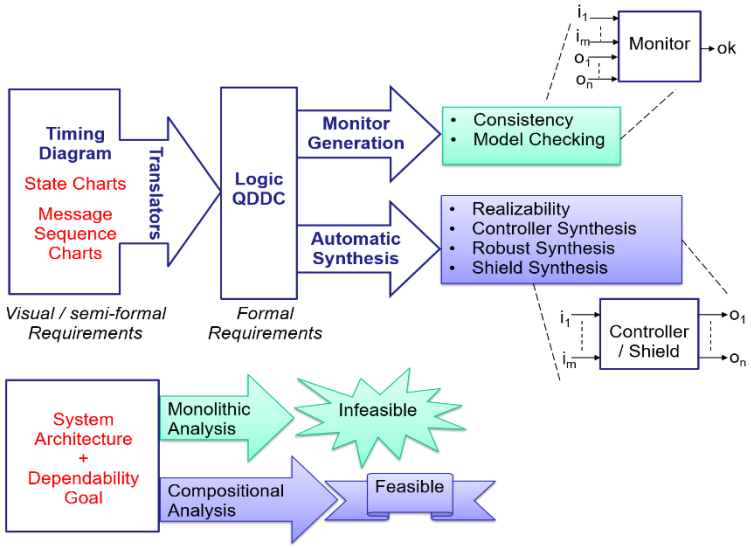


Figure 1: Heterogeneous Requirement Analysis and Synthesis

The thesis provides a novel formalization of timing diagram requirements using logic *QDDC*. An elementarily decidable subset of *QDDC* is used for this. As one of the main contributions, the notion of *quality* in automatic synthesis of controllers is explored, using the *soft requirement* guided synthesis approach proposed in this thesis. Soft requirements are the *QDDC* formulas (typically used to designate qualitative requirements) which must be satisfied *H-optimally*. A tool called *DCSynth* to efficiently construct such a high quality controller has been developed. This synthesis technique is then utilized to give formal specification and automatic synthesis of Robust Controller as well as Runtime Enforcement Shield. Various existing and the newly proposed notions are encoded using our logic-based framework. A *generic* method to synthesize various Robust Controllers and Shields is given using our tool *DCSynth*. Because of the possibility to encode and synthesize various Robust Controllers and Shields, it became possible to compare these various notions. A method based on computing the steady state probability of meeting the Soft requirements in the synthesized controller has been used for the comparison. The thesis has also explored the Architecture centric analysis of non-functional requirements such as *dependability* (guided by parameters like availability, reliability etc.). A *compositional* Markov Model (DTMC) based approach is proposed to overcome the state-space explosion problem and effectively used for analysis of Industrial Case Studies.