Symbolic Hierarchical Interface-based Supervisory Control

Symbolic Synthesis and Verification of Hierarchical Interface-based Supervisory Control

By

Raoguang Song, B.Eng.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree

Master of Applied Science

McMaster University

Master of Applied Science(2006) McMaster University
Computing and Software Hamilton, Ontario

TITLE: Symbolic Synthesis and Verification of Hierarchical Interface-based Supervisory Control

AUTHOR: Raoguang Song, B.Eng.

SUPERVISOR: Dr. Ryan J. Leduc

NUMBER OF PAGES: xvii, 249

Abstract

Hierarchical Interface-based Supervisory Control (HISC) is a method to alleviate the state-explosion problem when verifying the controllable and nonblocking properties of a large discrete event system. By decomposing a system as a number of subsystems according to the HISC method, we can verify the subsystems separately and the global system controllability and nonblocking property are guaranteed, so that potentially great computation effort is saved.

In this thesis, we first present a predicate-based synthesis algorithm for each type of subsystem and then prove the correctness of the algorithms. Then a predicate-based verification algorithm for each type of subsystem is provided. Based on the predicate-based algorithms, a symbolic implementation is proposed by using Binary Decision Diagrams (BDD) and the fact that a subsystem is usually composed of a number of components. With the symbolic implementation, we can handle a much larger subsystem of each type.

Two large and complicated examples (with estimated worst-case state space on the order of 10^{30}) extended from the AIP example are provided for demonstrating the capabilities of the algorithms and the implementation. A software tool for the synthesis and verification using our approach is also developed.

Acknowledgements

First I would like to sincerely thank my supervisor, Dr. Ryan Leduc, for his insightful guidance, great patience and generous support; for his careful review and corrections to the draft of this thesis.

A special thanks to Dr. Chuan Ma for his help on the BDD implementation techniques and STS. I must acknowledge preceding research works on HISC synthesis method done by Pengcheng Dai and many useful discussions with him.

At last, I am very much indebted to my wife Ning Zhou and my daughter Karen. To them, I dedicate this thesis.

Contents

List of Figures **xiii**

List of Tables **xvii**

1 Introduction **1**

1.1 Research Review 3

1.2 Thesis Overview 6

1.3 Thesis Structure 7

2 Preliminaries **9**

2.1 Algebra 9

2.1.1 Posets and Lattices 9

2.1.2 Equivalence Relations 11

2.1.3 Monotone Functions and Fixpoints 11

2.2 Languages 14

2.3 DES 15

2.4 Operations on DES 18

2.4.1 Product and Meet 18

2.4.2 Synchronous Product 18

2.5 Predicates and Predicate Transformers 20

2.5.1 Predicates 20
2.5.2 Predicate Transformers 22
2.5.3 Languages Induced by Predicates 27
2.6 Supervisory Control 32
2.6.1 Controllable Languages 32
2.6.2 Supervisory Control 33
2.6.3 Implementation of MNSC V for $\mathbf{G}$ 35
2.6.4 Supervisor Synthesis 38

3 HISC Overview **39**
3.1 System Structure 39
3.1.1 Command-pair Interfaces 42
3.1.2 Flat System 43
3.2 Local Conditions 44
3.2.1 Interface Consistent 45
3.2.2 Local Conditions for Global Nonblocking 54
3.2.3 Local Conditions for Global Controllability 54
3.3 Level-wise Interface Controllable Supervisor 55

4 Synthesis of HISC **59**
4.1 Introduction 59
4.2 High-level Supervisor Synthesis 62
4.2.1 High-level Interface Controllable Language 62
4.2.2 $\sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H))$ and the Greatest Fixpoint of Ω_H 66
4.2.3 Computing the greatest fixpoint of Ω_H 70
4.2.4 Computing $\sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$ 72
4.2.5 Computing $\Omega_{HNB}(L(\boldsymbol{\mathcal{G}}_H, P))$ 92

4.2.6 The Algorithm to Compute $\sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H))$ 93
4.3 Low-level Supervisor Synthesis 101
4.3.1 The j^{th} Low-level P4 Interface Controllable Language 101
4.3.2 The j^{th} Low-level Interface Controllable Language 110
4.3.3 $\sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}))$ and the Greatest Fixpoint of Ω_{L_j} 113
4.3.4 Computing the Greatest Fixpoint of Ω_{L_j} 121
4.3.5 The Algorithm to Compute $\sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}))$ 136

5 Verification of HISC **143**

6 Symbolic Computation for HISC Synthesis and Verification **145**
6.1 Symbolic Representation of State Subsets and Transitions 146
6.1.1 Symbolic Representation of State Subsets 147
6.1.2 Symbolic Representation of Transitions 151
6.2 Symbolic Computation of Predicate Transformers 154
6.2.1 Computation of Transition and Inverse Transition 155
6.2.2 Computation of R 161
6.2.3 Computation of TR 167
6.2.4 Computation of $\mathcal{CR}$ 167
6.3 Miscellaneous Computation for HISC Synthesis and Verification . . . 168
6.3.1 Computation of $pr(\text{Bad}_H)$ 169
6.3.2 Computation of $pr(\text{Bad}_{L_j})$ 171
6.3.3 Miscellaneous Computation in Algorithm 4.2 172
6.3.4 Verifying Event Partition 173
6.3.5 Verifying Command-pair Interfaces 174
6.4 BDD Implementations of Algorithms 177
6.4.1 State Variable Ordering 178

6.5 Controller Implementation 180
6.5.1 Simplifying Control Predicates 188
6.6 A Small Example 193

7 The AIP Example 199

7.1 Introduction of the AIP 200
7.2 Control Specifications 203
7.3 System Structure 205
7.4 The Interface DES 207
7.5 Low-level Subsystems 208
7.6 The High-level Subsystem 209
7.6.1 Plant Components 210
7.6.2 Supervisor Components 214
7.7 Verifying Properties 222
7.8 Synthesizing Supervisors 226
7.9 Results For the AIP Example 228

8 Multiple AIP Example 229

8.1 System Model 230
8.2 Verifying Properties 234
8.3 Synthesizing Supervisors 236

9 Conclusions and Future Work 239

9.1 Conclusions 239
9.2 Future Work 241
9.2.1 Intermediate BDD Size Problem 241
9.2.2 Complexity Analysis of the Algorithms 242

9.2.3 Symbolic Computation for Timed HISC System 242

A HISC Software Program **251**

A.1 Introduction of HISC Software Program 251

A.2 Source Code 253

List of Figures

2.1 MACH 16

2.2 Example for predicate transformers 26

3.1 HISC block diagram 40

3.2 HISC system structure 42

3.3 Example interface 43

6.1 Small factory example 148

6.2 Example DES $\mathbf{G}_E$ 157

6.3 Example system $\mathbf{M}$ 162

6.4 Control diagram 183

6.5 New control diagram 187

6.6 System diagram for small example 193

6.7 The small example 194

6.8 Synthesized high-level proper supervisor 195

6.9 Synthesized low-level proper supervisor for low-level $m1$ 195

6.10 Synthesized low-level proper supervisor for low-level $m2$ 195

6.11 Control predicates for $m1\_st, m2\_st, m2\_rpr, tu\_st$ 197

6.12 Prime simplified control predicates for $m1\_st, m2\_st, m2\_rpr, tu\_st$ . . 197

6.13 Triple-prime simplified control predicates for $m1\_st, m2\_st, m2\_rpr, tu\_st$ 197

6.14 The control predicates for *m2_cpl*. 198

7.1 The AIP system architecture(from [21]) 200

7.2 Assembly station of external loop X = 1, 2, 3 (from [21]) 201

7.3 Transport unit for external loop X = 1, 2, 3, 4 (from [21]) 202

7.4 The AIP system structure 205

7.5 Interface to *low-level* $w = 1, 2$ (from [23]) 207

7.6 Interface to *low-level* 3 (from [23]) 207

7.7 Interface to *low-level* $v = 4, 5, 7$ (from [23]) 207

7.8 Interface to *low-level* 6 (from [23]) 207

7.9 Interface to *low-level* 8 208

7.10 CapGateEL_2.AS3 208

7.11 OperateGateEL_2.AS3 209

7.12 PalletArvGateSenEL_2.AS3 209

7.13 *Low-level 8 Subsystem* 209

7.14 Component DES in the AIP high-level 210

7.15 ASStoreUpState.k = AS1, AS2 (from [23]) 211

7.16 QueryPalletAtTU.i, i = TU1, TU2, TU3, TU4 211

7.17 QueryPalletAtIO 211

7.18 CapEL12, CapEL22, CapEL32, CapEL42 212

7.19 CapC1, CapC2, CapC3, CapC4 213

7.20 CapEL11, CapEL21, CapEL31, CapEL41 214

7.21 ManageTU1 216

7.22 ManageTU2 216

7.23 ManageTU3 217

7.24 ManageTU4 217

7.25 DetWhichStnUp (from [21]) 218
7.26 HndlComEventsAS(from [21]) 218
7.27 OFProtEL11 220
7.28 OFProtEL21 220
7.29 OFProtEL31 221
7.30 OFProtEL41 221
7.31 OFProtEL12, OFProtEL22, OFProtEL32, OFProtEL42 222
7.32 OFProtC1, OFProtC2, OFProtC3, OFProtC4 223
7.33 AltMvInTypes 223
7.34 ManageIO 223
7.35 OFProtAIP 224

8.1 Block diagram of multiple AIPs 230
8.2 IntfAIP-j(j = 1, 2, 3) 230
8.3 New low-level plants in 3-2 MAIP system(verfication) 231
8.4 Components in low-level subsystem MAIP-j, ($j = 1, 2, 3$) 232
8.5 High-level plant components 233
8.6 High-level supervisor components 234
8.7 New low-level plants for the 5-4 MAIP system(verification) 236
8.8 New low-level plants for the 3-2 MAIP system(synthesis) 237

9.1 Number of BDD nodes and time for CR operation at each iteration . 242

List of Tables

7.1 The AIP example data (verification) 225
7.2 The AIP example data (synthesis) 227

8.1 The multiple AIP example data(verification) 235
8.2 The multiple AIP example data(synthesis) 238

A.1 Source code files in the software library 252
A.2 Source code files for the library usage example 252

Chapter 1

Introduction

A Discrete Event System (DES) is a dynamic system that is discrete in state space and is event-driven. Ramadge and Wonham (RW) [36] proposed the supervisory control theory framework for a class of general DES, in which the DES is represented as an automaton over an event alphabet, and its behavior is described by the language generated by the automaton. The events in the alphabet are partitioned as controllable events which can be disabled and uncontrollable events which can not be disabled. By disabling and enabling controllable events, a system controller can restrict the behavior of a plant. Such a controller is also called a *supervisor*. Two properties, controllability and nonblocking, are usually desirable for the behavior of the controlled system.

Although the supervisory control theory does not require the language to be regular, in practice the language should be, so that the automaton representing the language has a finite state space. The control action in the theory is based on the strings generated by the plant. However, the number of the strings could be infinite. In order to implement such a controller, a supervisor DES can be designed so that the synchronous product of the plant DES and the supervisor DES can generate the

desired behavior. We can design the supervisor DES by hand, and then verify the controllable and nonblocking properties.

In [48], RW provided an algorithm to synthesize an optimal supervisor according to an arbitrary specification language. An optimal supervisor means that it is a minimally restrictive controller that satisfies the specification and is nonblocking and controllable. The complexity of this algorithm is polynomial on the number of the DES states.

The most intuitive but inefficient method to store a DES in a computer is to save the state space and transitions in look-up tables directly. To check the controllable and nonblocking properties, we have to traverse the DES which means we have to go through and search the look-up tables. However, most practical systems have a very large state space since they are usually modeled as the synchronous product of a group of component DES, so the state space size is increased exponentially on the number of the components. Such a problem is known as the *state explosion problem*.

The software TCT [47] is a tool using the above method to store a DES. It provides a synthesizing routine (**supcon**) to compute the optimal supervisor and other routines to help model a system and check properties. Due to the state explosion problem, TCT can only handle some simple systems under the memory limitations (With 1GB RAM, TCT can handle a system on the order of 10^7 states). Therefore, it is necessary to find other efficient approaches to model and represent DES such that for a large system a supervisor can be synthesized and nonblocking and controllable properties can be verified.

1.1 Research Review

In order to overcome the state explosion problem, usually two categories of approaches can be applied: to explore control architectures and to explore internal structures of DES.

Modular control [13,34,49] is an early and widely used method, which implements control action by designing multiple supervisors. Each supervisor is responsible to implement part of the control specifications. In [3, 30, 39, 45, 50], this approach is extended as decentralized control by allowing each supervisor access only to partial observations of the plant. These architectures successfully solved the controllability verification but not for the nonblocking verification.

Another architectural approach is the model aggregation methods or the bottom up methods [8,18,33,41,46,52]. In these methods, the aggregation model(high-level) is obtained from low-level models by language aggregation or state aggregation. The "hierarchical consistency" concept is introduced to make sure that a high-level supervisor can be implemented in low-levels. However, as aggregate models are constructed from low-levels sequentially, a given level can not be constructed until all the levels below it have been constructed. Furthermore, the conditions for this type of approaches are sufficient and necessary, which means that a change on one level may affect all the levels above it.

In [35], RW introduced a static state feedback supervisory control framework by using predicates to store the state space and define the specifications. In [47], a dynamic state feedback control is also introduced by using memory components. However, they did not mention how to represent a predicate.

In the VDES [9, 10, 28, 29] or Petri Net [32, 42, 43] framework, a system state is represented by a vector with integer components, and state transitions by integer

vector addition. The specifications are defined in the form of linear inequalities. The supervisor is synthesized by using linear programming algorithms. These approaches are applied well on the systems with high degree of regularity. Moreover, to verify the nonblocking property or synthesize a nonblocking controller, a reachability graph may need to be constructed first.

STCT [51] utilized the fact that a system is usually modeled as a group of DES components, so that each state of the system can be represented as a vector. It then does the synthesis process based on the synchronous product of the plant and the specification components but does not actually build the synchronous product. By using IDD (Integer Decision Diagram, an extension of Binary Decision Diagram (BDD) [6]) to encode the state space and transitions of the system, STCT can synthesize an optimal controllable and nonblocking supervisor for a system with closed-loop state space size on the order of 10^{23}, but that system is quite artificial, and there is no rigorous theoretical foundation for the algorithms.

Ma [31] presented a top-down multi-level design model called State Tree Structures (STS) which was initiated in [44] by using the idea of state charts [17]. A STS includes a State Tree to store states and holons to store local dynamics. A sub-state-tree is used to store a state set and a basic sub-state-tree is used to represent one state. Set operations such as union could be done by the join operation of two sub-state-trees, and global dynamics could be constructed from the local holons, but there is no easy sub-state-tree operation corresponding to the intersection of two state sets. Furthermore, not all of the state sets can be represented as one sub-state-tree. However, by using STS, one could model a system from the top level and treat each module as a superstate, and later on the module can be further designed in detail. These modeling steps reflect the hierarchical nature of most complex systems, so modeling a system as a STS will be straightforward.

To solve the above problems in STS, Ma used BDDs to represent state sets. Due to the rich structure of STS, the state sets can be represented by BDDs compactly. By using the optimized recursive algorithms, he modeled and synthesized a state-based supervisor for a system with an estimated state space on the order of 10^{24}. However, in order to construct the global transitions from local transitions, STS requires the local coupling condition: no shared events between two holons matched to the OR superstates that are not AND-adjacent to the same AND superstates. This restriction could force a modeler to flatten the STS, so that it makes the STS less structural and some optimization techniques (e.g. coreachable inference) work less efficiently. We will talk more about this in the AIP example chapter. Furthermore, in STS, the synthesis process is still working on the global state space, which grows exponentially on the number of components.

Leduc *et al.* [21–27] proposed the Hierarchical Interface-based Supervisory Control (HISC) by bringing the information hiding theory from software engineering into supervisory control. By using this method, a system is decomposed into one high-level subsystem and multiple low-level subsystems. Between the high-level subsystem and each low-level subsystem, there is a well-defined interface which restricts the interaction of the subsystems. All the communications between the high-level subsystem and low-level subsystems must be done through a specific interface. Therefore, after modeling the interfaces, we can model those subsystems simultaneously. The really valuable thing in HISC is that as long as each subsystem and its interface satisfies certain local conditions, then the global controllable and nonblocking properties can be guaranteed, which means that we only need to work on the states in each subsystem instead of the states of the synchronous product of the whole system. With automata based algorithms, Leduc successfully verified a system with an estimated worst-case closed-loop state space on the order of 10^{21}. Richter *et al.* [37] also modeled and

verified a bottling plant by using the HISC method with estimated state space on the order of 10^{47} (They obtained this number by multiplying the number of states in each component DES).

1.2 Thesis Overview

Because the algorithms in [21] (Leduc) explicitly list the states and transitions, each subsystem could not be very large. With 1GB of RAM, he could handle individual components only as large as 10^6–10^7 states. This might put limitations on modeling a real system especially for the high-level. Dai [14] is developing synthesis algorithms to compute the supervisors for each subsystem. However, his algorithms are still based on the explicit state and transition listing, thus do not save much computation.

As each subsystem in HISC is usually modeled as a group of plant DES and a group of specification/supervisor DES, each subsystem-wide state can be represented as a vector, where each member of the vector is the state of a component DES. Therefore, we can use BDD to represent the state space and transitions for each subsystem, and develop algorithms based on BDD representations to verify or synthesize supervisors for it. By using the BDD representation and algorithms, our program is able to handle much larger subsystem, so the power of HISC is greatly improved.

In this thesis, we need to verify not only the controllable and nonblocking properties but also the interface consistent conditions in [21] for the high-level and low-level subsystems. The conditions we used are from [23, 24], an equivalent version of the conditions in [21]. A proof for the equivalence between the two can be found in [14,24]. We first present the synthesis algorithms based on specifications for each type of subsystem, and then give the verification algorithms based on modular supervisors for

each type of subsystem. The correctness of the algorithms is proved. A software tool to implement these algorithms is also developed.

Two examples of industrial size are provided to show the capability of our algorithms in this thesis. The first one is extended from the AIP example in [21], which synthesizes a high-level supervisor with state space on the order of 10^{15}. In the second example, we build a feedback system by treating each high-level subsystem in the first example as a low-level system, and then synthesize a low-level supervisor.

As the structure of each subsystem could be thought of as a special STS with three levels (Level 0 is the root AND superstate, and each state in Level 1 is an OR superstate corresponding to each component DES, and each state in Level 2 is a simple state corresponding to each state in the component DES), we borrowed many ideas from [31], especially with respect to BDD implementation.

1.3 Thesis Structure

The remaining chapters in this thesis are organized as follows.

In Chapter 2, we present the necessary algebra and language foundations for proofs in this thesis. Basic supervisory control theory of DES and some fundamental operations of DES and propositions are also presented.

In Chapter 3, we give an overview for the HISC method, and then give an equivalent definition of the interface consistent condition for the purpose of proving our synthesis algorithms.

In Chapter 4 and 5, the synthesis and verification algorithms based on predicate operations for an HISC system are presented and proved.

In Chapter 6, we give a BDD-based implementation of the algorithms in Chapter 4 and 5. The encoding techniques and optimization techniques for the BDD imple-

mentations are the main focus of this chapter.

In Chapter 7 and 8, examples to show the capabilities of our algorithms and BDD implementations are presented.

Finally, in Chapter 9, we discuss the future work and conclusions.

Chapter 2

Preliminaries

In this chapter, we first discuss some basic algebraic and linguistic concepts that will be used in the later chapters. Then we introduce the predicate and predicate transformers which are important for our later algorithms. After that, we give an overview of DES and the RW supervisory control theory. We also state and prove several fundamental propositions. This chapter is largely based on [47]. Proposition 2.1 is an extension of Theorem 3.18 from [19] by Huth and Ryan.

2.1 Algebra

2.1.1 Posets and Lattices

Let X be a set, and $\leq$ be a binary relation on X. $\leq$ is a *partial order* if it has the following three properties:

- reflexive: $(\forall x \in X)\ x \leq x$
- transitive: $(\forall x, y, z \in X)\ x \leq y\ \&\ y \leq z \Rightarrow x \leq z$
- antisymmetric: $(\forall x, y \in X)\ x \leq y\ \&\ y \leq x \Rightarrow x = y$

If $\leq$ is a partial order on X, then the pair $(X, \leq)$ is called a *poset*. If $\leq$ is understood, we could call X a poset.

Let $(X, \leq)$ be a poset, $x, y \in X$. An element $l \in X$ is a *meet* of x and y(denote it as $x \sqcap y = l$ [1]), if

$$l \leq x \ \& \ l \leq y \ \& \ ((\forall a \in X)\ a \leq x \ \& \ a \leq y \Rightarrow a \leq l).$$

Dually, an element $u \in X$ is a *join* of x and y(denote it as $x \sqcup y = u$), if

$$x \leq u \ \& \ y \leq u \ \& \ ((\forall b \in X)\ x \leq b \ \& \ y \leq b \Rightarrow u \leq b).$$

Let L be a set. A poset $(L, \leq)$ is a lattice if the meet and join of any two elements in L always exist.

Let S be a nonempty subset of L and $l \in L$. Element l is the *infimum* of S ($\inf S$) if

$$(\forall y \in S)\ l \leq y \ \ \& \ \ (\forall z \in L)((\forall y \in S) z \leq y) \Rightarrow z \leq l$$

Dually, $u \in L$ is the *supremum* of ($\sup(S)$) if

$$(\forall y \in S)\ y \leq u \ \ \& \ \ (\forall z \in L)((\forall y \in S) y \leq z) \Rightarrow u \leq z$$

A *lattice* $(L, \leq)$ is *complete* if, for any nonempty subset S of L, both $\inf S$ and $\sup S$ always exist. Every finite lattice is *complete*.

Let X be a set. Define $Pwr(X)$ as the set consisting of all the subsets of X. $(Pwr(X), \subseteq)$ is a *complete lattice* [15]. This fact is fundamental for the correctness of our algorithms.

[1] In [47], the meet of elements x and y is denoted as $x \wedge y$, and the join of them is denoted as $x \vee y$, but we will use the notation $\wedge$ for predicate operation *and* and $\vee$ for predicate operation *or*, because the predicate operations are heavily used in this thesis.

2.1.2 Equivalence Relations

Let X be a nonempty set, and E a binary relation on X. E is an *equivalence relation* if

- $(\forall x \in X)xEx$ (reflexive)
- $(\forall x, x' \in X)xEx' \Rightarrow x'Ex$ (symmetric)
- $(\forall x, x', x'' \in X)xEx' \ \& \ x'Ex'' \Rightarrow xEx''$ (transitive)

Let $x \in X$, define $[x] \subseteq X$ as $[x] := \{x' \in X | x'Ex\}$. $[x]$ is called the *coset* or *equivalent class* of x with respect to the equivalence relation E.

By $x \in [x]$, $[x]$ is nonempty.

Let $x, y \in X$, then it can be shown that either $[x] = [y]$ or $[x] \cap [y] = \emptyset$. Namely, $[x]$ and $[y]$ are either equal or disjoint.

2.1.3 Monotone Functions and Fixpoints

Let $(X, \leq)$ be a complete lattice. A function $f : X \to X$ is *monotone* if

$$(\forall x, y \in X) \ x \leq y \Rightarrow f(x) \leq f(y).$$

Let X be a set, and $f : X \to X$ be a function. An element $x \in X$ is a *fixpoint* of f if $x = f(x)$. Let $(X, \leq)$ be a complete lattice. The element x is the *greatest fixpoint* of f if

$$(\forall x' \in X) \ x' = f(x') \Rightarrow x' \leq x.$$

Let X be a set, and $f : X \to X$ be a function. For $x \in X$, denote $f^0(x)$ for x, $f^1(x)$ for $f(x)$, $f^2(x)$ for $f(f(x))$, ..., and $f^i(x)$ for

$$\underbrace{f(f(\ldots f}_{i}(x))),\ i \in \{0, 1, 2, \ldots\}.$$

We now state Theorem 2.1 which is the well known Knaster-Tarski Theorem [40].

Theorem 2.1 (Knaster-Tarski Theorem)**.** *Let* $(X, \leq)$ *be a complete lattice, and let* $f : X \to X$ *be a monotone function. Then the* greatest fixpoint *of* f *exists and is equal to* $\sup\{x | x \leq f(x)\}$. □

Note that the set X can be *uncountable infinite*, and that this theorem does not say anything about how to compute the greatest fixpoint. It only states the existence.

For our usage, the complete lattice is $(Pwr(\Sigma^*), \subseteq)$. Σ^* is countable infinite, while $Pwr(\Sigma^*)$ is uncountable infinite.

Proposition 2.1 is a special case of the Knaster-Tarski theorem, and it also gives a method to compute the greatest fixpoint for the special case.

Proposition 2.1. *Let* X *be a finite set with* n *elements,* $n \in \{0, 1, \ldots\}$*. If a function* $f : Pwr(X) \to Pwr(X)$ *is a monotone function with respect to* $\subseteq$*, then there exists* $k \in \{0, 1, 2, \ldots\}$ *such that* $k \leq n$ *and* $f^k(X)$ *is the* greatest fixed point *of* f *with respect to* $(Pwr(X), \subseteq)$*, and* $(\forall i \in \{1, 2, \ldots\})\ i > k \Rightarrow f^i(X) = f^k(X)$.

Proof:

1. Show that there exists $k \leq n$, $f^k(X)$ is a fixpoint of f, i.e. $f^k(X) = f^{k+1}(X)$.

   We know that $f(X) \subseteq X$,

   $\Rightarrow f^2(X) \subseteq f(X)$, as f is monotone.

   $\Rightarrow f^3(X) \subseteq f^2(X)$

   $\ldots$

   $\Rightarrow f^{n+1}(X) \subseteq f^n(X)$

   So, we have

$$f^{n+1}(X) \subseteq f^n(X) \subseteq \cdots \subseteq f^1(X) \subseteq X$$

Assume $\nexists k \leq n, f^k(X) = f^{k+1}(X)$, then

$$f^{n+1}(X) \subset f^n(X) \subset \cdots \subset f^1(X) \subset X$$

This is impossible, because X only contains n elements, and $f^{k+1}(X) \subset f^k(X)$ ($k \in \{0, 1, \ldots\}$) means that $f^{k+1}(X)$ contains at least one less element than $f^k(X)$ does.

2. Show that $f^k(X)$ is the greatest fixpoint of f, i.e. $(\forall S \in Pwr(X))\ S = f(S) \Rightarrow S \subseteq f^k(X)$.

   Let $S \in Pwr(X)$, and assume $S = f(S)$.

   Must show this implies $S \subseteq f^k(X)$.

   We know that $S \subseteq X$

   $\Rightarrow f(S) \subseteq f(X)$, as f is monotone

   $\Rightarrow S \subseteq f(X)$, as $S = f(S)$

   $\Rightarrow f(S) \subseteq f^2(X)$, as f is monotone

   $\Rightarrow S \subseteq f^2(X)$, as $S = f(S)$

   $\ldots$

   $\Rightarrow S \subseteq f^k(X)$

3. Show that $(\forall i \in \{1, 2, \ldots\}) i > k \Rightarrow f^i(X) = f^k(X)$.

   This follows immediately since $f^k(X)$ is a fixpoint of f, i.e. $f^k(X) = f^{k+1}(X)$.

□

2.2 Languages

Let Σ be a finite set of symbols, and we refer to Σ as an *alphabet*. A *string* over Σ is a finite sequence of symbols in Σ. Denote Σ^+ as the set of all such strings over Σ.

Let ϵ ($\notin \Sigma$) be the *empty string* (no symbols), we write $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$.

We define the operation of *concatenation* of strings: $cat : \Sigma^* \times \Sigma^* \to \Sigma^*$ according to

$$cat(\epsilon, s) = cat(s, \epsilon) = s, \qquad s \in \Sigma^*,$$
$$cat(s, t) = st, \qquad s, t \in \Sigma^+.$$

A *language* L is a subset of Σ^*, and thus an element of $Pwr(\Sigma^*)$.

Let $s, t \in \Sigma^*$. t is said to be a *prefix* of s ($t \leq s$) if $(\exists u \in \Sigma^*)\ tu = s$.

Let $L \subseteq \Sigma^*$. The *(prefix) closure* of L is defined as:

$$\overline{L} := \{t \in \Sigma^* | (\exists s \in L)\ t \leq s\}.$$

A language L is *closed* if $L = \overline{L}$.

The following proposition will be used in following chapters.

Proposition 2.2. *Let $L_1, L_2 \in Pwr(\Sigma^*)$. Then the following holds:*

$$\overline{L_1} \cup \overline{L_2} = \overline{L_1 \cup L_2}$$

proof:

$\overline{L_1} \cup \overline{L_2} \subseteq \overline{L_1 \cup L_2}$ follows immediately since $\overline{L_1} \subseteq \overline{L_1 \cup L_2}$ and $\overline{L_2} \subseteq \overline{L_1 \cup L_2}$.

We now show $\overline{L_1 \cup L_2} \subseteq \overline{L_1} \cup \overline{L_2}$.

Let $s \in \overline{L_1 \cup L_2}$ (1)

Must show this implies $s \in \overline{L_1} \cup \overline{L_2}$

By (1), we know $(\exists s' \in \Sigma^*)\ ss' \in L_1 \cup L_2$

$\Rightarrow ss' \in L_1$ *or* $ss' \in L_2$ (2)

If $ss' \in L_1$, then $s \in \overline{L_1}$. We thus have $s \in \overline{L_1} \cup \overline{L_2}$. (3)

Similarly, if $ss' \in L_2$, then $s \in \overline{L_2}$. We thus also have $s \in \overline{L_1} \cup \overline{L_2}$. (4)

By (2), (3) and (4), we can conclude $s \in \overline{L_1} \cup \overline{L_2}$.

□

Let $L \subseteq \Sigma^*$ and $s \in \Sigma^*$. The operator $\text{Elig}_L : \Sigma^* \to Pwr(\Sigma)$ is used to get *the set of eligible events for s with respect to L*.

$$\text{Elig}_L(s) := \{\sigma \in \Sigma | s\sigma \in L\}.$$

Let $L \subseteq \Sigma^*$. The *Nerode equivalence relation on Σ^* with respect to L* is defined as

$$(\forall s, t \in \Sigma^*)\ s \equiv_L t \ \text{ iff } \ (\forall u \in \Sigma^*)\ su \in L \Leftrightarrow tu \in L.$$

Write $\| L \|$ as the cardinality of the set of the equivalence class of $\equiv_L$. If $\| L \|$ is finite, we say L is *regular*. *All the languages in this thesis are regular unless otherwise stated.*

For a regular language L, if we treat each Nerode equivalence class as a state, then we can build a *recognizer* with finite number of states to tell if a string $s \in \Sigma^*$ is in L or not. Such a recognizer is said to be *canonical*.

2.3 DES

A DES $\mathbf{G}$ is a *generator*, and formally defined as a five tuple

$$\mathbf{G} := (Q, \Sigma, \delta, q_0, Q_m),$$

where Q is the state set; $\Sigma = \Sigma_c \dot{\cup} \Sigma_u$, where Σ_c is the set of *controllable* events, and Σ_u is the set of *uncontrollable* events; $\delta : Q \times \Sigma \to Q$ is the *(partial) transition function*; q_0 is the *initial state*, and $Q_m \subseteq Q$ is the set of *marker states*. We always assume Q and Σ are finite.

Let $q \in Q, \sigma \in \Sigma$. We use $\delta(q, \sigma)!$ to mean that $\delta(q, \sigma)$ is defined.

The transition function δ can be extended to $\delta : Q \times \Sigma^* \to Q$ according to

$$\delta(q, \epsilon) = q$$
$$\delta(q, s\sigma) = \delta(\delta(q, s), \sigma)$$

provided $q' := \delta(q, s)!$ and $\delta(q', \sigma)!$. Note that $\delta(q, \epsilon)$ is always defined.

A DES over Σ is the *empty* DES if $Q = \emptyset$, and we denote it as **EMPTY**.

A DES **G** can be represented as a *transition graph*. The nodes(circles) in the graph represent states in Q. The edges represent transitions. The initial state is labeled with an entering arrow ($\rightarrow\circ$), and the marker states are labels with an exiting arrow($\circ\rightarrow$). If the initial state is also a marker state, it will be labeled with a double side arrow($\circ\leftrightarrow$). The controllable transitions are labeled with a slash on the edge. Figure 2.1 is the transition graph representation for the well known **MACH** DES. The DES **MACH** includes three states $s0$, $s1$ and $s2$. State $s0$ is the initial state and

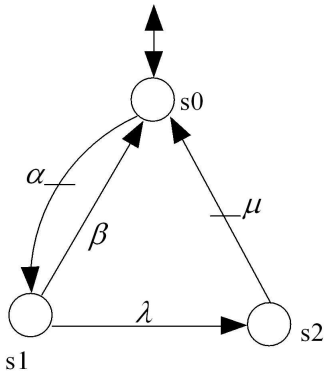


Figure 2.1: MACH

the only marker state. There are four transitions: $(s0, \alpha, s1)$, $(s1, \beta, s0)$, $(s1, \lambda, s2)$, $(s2, \mu, s0)$. Events α, μ are controllable events and events β, λ are uncontrollable events. Note that the event set Σ may include events other than the four events shown in the graph.

A DES $\mathbf{G}$ generates two languages: $L(\mathbf{G})$ and $L_m(\mathbf{G})$, which are called the *closed behavior* of $\mathbf{G}$ and the *marked behavior* of $\mathbf{G}$ respectively, and are defined as follows.

$$L(\mathbf{G}) := \{s \in \Sigma^* | \delta(q_0, s)!\}$$
$$L_m(\mathbf{G}) := \{s \in \Sigma^* | \delta(q_0, s) \in Q_m\}$$

The language $L(\mathbf{G})$ is closed, and $L_m(\mathbf{G}) \subseteq L(\mathbf{G})$.

Let $q \in Q$. State q is *reachable* if $\exists s \in \Sigma^*$ such that $\delta(q_0, s) = q$. State q is *coreachable* if $\exists s \in \Sigma^*$ such that $\delta(q, s) \in Q_m$.

A DES $\mathbf{G}$ is *reachable*, if every state in Q is reachable.

A DES $\mathbf{G}$ is *coreachable*, if every state in Q is coreachable.

A DES $\mathbf{G}$ is *trim*, if every state in Q is reachable and coreachable.

A DES $\mathbf{G}$ is *nonblocking*, if every reachable state in Q is coreachable. i.e. $\overline{L_m(\mathbf{G})} = L(\mathbf{G})$.

We say that DES $\mathbf{G}$ represents a language $K \subseteq \Sigma^*$ if $\mathbf{G}$ is nonblocking and $L_m(\mathbf{G}) = K$. We thus have $L(\mathbf{G}) = \overline{K}$.

In the tuple definition of $\mathbf{G}$, we define that $\delta : Q \times \Sigma \to Q$ is a partial function. We say such a DES $\mathbf{G}$ is *deterministic.* Instead, one can define a DES $\mathbf{T} := (Q, \Sigma, \tau, q_0, Q_m)$, where Q, Σ, q_0 and Q_m are defined as in $\mathbf{G}$, but $\tau : Q \times \Sigma \to Pwr(Q)$ is a total function. We say such a DES $\mathbf{T}$ is *nondeterministic.*

As every nondeterministic DES can be converted to a deterministic DES, we assume that all the DES in this thesis are deterministic.

2.4 Operations on DES

2.4.1 Product and Meet

Let $\mathbf{G_1} = (Q_1, \Sigma, \delta_1, q_{1_0}, Q_{1_m})$ and $\mathbf{G_2} = (Q_2, \Sigma, \delta_2, q_{2_0}, Q_{2_m})$ be two DES defined over the same alphabet Σ. The *product* of two DES is defined as:

$$\mathbf{G_1} \times \mathbf{G_2} := (Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (q_{1_0}, q_{2_0}), Q_{1_m} \times Q_{2_m}),$$

where $\delta_1 \times \delta_2 : Q_1 \times Q_2 \times \Sigma \to Q_1 \times Q_2$ is given by

$$(\delta_1 \times \delta_2)((q_1, q_2), \sigma) := (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)),$$

whenever $\delta_1(q_1, \sigma)!$ and $\delta_2(q_2, \sigma)!$.

From the definition, we have $L(\mathbf{G_1} \times \mathbf{G_2}) = L(\mathbf{G_1}) \cap L(\mathbf{G_2})$, and $L_m(\mathbf{G_1} \times \mathbf{G_2}) = L_m(\mathbf{G_1}) \cap L_m(\mathbf{G_2})$.

The *meet* of DES $\mathbf{G_1}$ and $\mathbf{G_2}$ is defined as $\mathbf{G_1} \times \mathbf{G_2}$ but only including the reachable states, and we denote it as $\mathbf{meet}(\mathbf{G_1}, \mathbf{G_2})$.

Obviously, $L(\mathbf{meet}(\mathbf{G_1}, \mathbf{G_2})) = L(\mathbf{G_1} \times \mathbf{G_2})$ and $L_m(\mathbf{meet}(\mathbf{G_1}, \mathbf{G_2})) = L_m(\mathbf{G_1} \times \mathbf{G_2})$.

2.4.2 Synchronous Product

Synchronous Product on Languages

Let Σ_1, Σ_2 be two alphabets, $\Sigma = \Sigma_1 \cup \Sigma_2$.

Define the *natural projection* $P_i : \Sigma^* \to \Sigma_i^* (i = 1, 2)$ according to

$$P_i(\epsilon) = \epsilon$$

$$P_i(\sigma) = \begin{cases} \epsilon & \text{if } \sigma \notin \Sigma_i \\ \sigma & \text{if } \sigma \in \Sigma_i \end{cases}$$

$$P_i(s\epsilon) = P_i(s)P_i(\sigma) \quad s \in \Sigma^*, \sigma \in \Sigma$$

Clearly, $(\forall s, t \in \Sigma^*)\ P_i(st) = P_i(s)P_i(t)$. In other words, P_i is *catenative*.

Let $P_i^{-1} : Pwr(\Sigma_i^*) \to Pwr(\Sigma^*)$ be the inverse image function of P_i, namely for $H \subseteq \Sigma_i^*$,

$$P_i^{-1}(H) := \{s \in \Sigma^* | P_i(s) \in H\}.$$

Let $L_1 \subseteq \Sigma_1^*, L_2 \subseteq \Sigma_2^*$. The *synchronous product* $L_1 \parallel L_2 \subseteq \Sigma^*$ is defined as

$$L_1 || L_2 := P_1^{-1}(L_1) \cap P_2^{-1}(L_2).$$

Selfloop

Let $\mathbf{G_1} = (Q_1, \Sigma_1, \delta_1, q_{1_0}, Q_{1_m})$ be a DES defined on alphabet Σ_1, and Σ_2 be another alphabet with $\Sigma_1 \cap \Sigma_2 = \emptyset$. The *selfloop* operation on $\mathbf{G_1}$ is used to generate a new DES $\mathbf{G}$ by selflooping each event in Σ_2 on each state of $\mathbf{G_1}$. Formally,

$$\mathbf{G} = \mathbf{selfloop}(\mathbf{G_1}, \Sigma_2) = (Q_1, \Sigma_1 \cup \Sigma_2, \delta_2, q_{1_0}, Q_{1_m}),$$

where $\delta_2 : Q_1 \times (\Sigma_1 \cup \Sigma_2) \to Q_1$ is a partial function and defined as

$$\delta_2(q, \sigma) := \begin{cases} \delta_1(q, \sigma), & \sigma \in \Sigma_1, \delta_1(q, \sigma)! \\ q, & \sigma \in \Sigma_2 \\ \text{undefined}, & \text{otherwise} \end{cases}$$

Let $P_1 : (\Sigma_1 \cup \Sigma_2)^* \to \Sigma_1^*$ be a natural projection, then we have

$$L(\mathbf{selfloop}(\mathbf{G_1}, \Sigma_2)) = P_1^{-1}(L(\mathbf{G_1}))$$
$$L_m(\mathbf{selfloop}(\mathbf{G_1}, \Sigma_2)) = P_1^{-1}(L_m(\mathbf{G_1}))$$

Synchronous Product on DES

Let $\mathbf{G_1} = (Q_1, \Sigma_1, \delta_1, q_{1_0}, Q_{1_m})$, $\mathbf{G_2} = (Q_2, \Sigma_2, \delta_2, q_{2_0}, Q_{2_m})$ be two DES. The *synchronous product* of $\mathbf{G_1}$ and $\mathbf{G_2}$ is defined as [2]

$$\mathbf{G_1} \parallel \mathbf{G_2} := \mathbf{selfloop}(\mathbf{G_1}, \Sigma_2 - \Sigma_1) \times \mathbf{selfloop}(\mathbf{G_2}, \Sigma_1 - \Sigma_2).$$

Note that the alphabet set of $\mathbf{G_1} \parallel \mathbf{G_2}$ is $\Sigma_1 \cup \Sigma_2$. The product of two DES $\mathbf{G_1}, \mathbf{G_2}$ is actually a special case of the synchronous product where $\Sigma_1 = \Sigma_2$.

We define $\mathbf{sync}(\mathbf{G_1}, \mathbf{G_2})$ as the DES $\mathbf{G_1} \parallel \mathbf{G_2}$ but only with reachable states. Then, we have

$$L(\mathbf{G_1} \parallel \mathbf{G_2}) = L(\mathbf{sync}(\mathbf{G_1}, \mathbf{G_2})) = L(\mathbf{G_1}) \parallel L(\mathbf{G_2}) = P_1^{-1}(L(\mathbf{G_1})) \cap P_1^{-1}(L(\mathbf{G_2}))$$
$$L_m(\mathbf{G_1} \parallel \mathbf{G_2}) = L_m(\mathbf{sync}(\mathbf{G_1}, \mathbf{G_2})) = L_m(\mathbf{G_1}) \parallel L_m(\mathbf{G_2}) = P_1^{-1}(L_m(\mathbf{G_1})) \cap P_1^{-1}(L_m(\mathbf{G_2}))$$

Note that our synchronous product operation on DES is associative. Namely, for DES $\mathbf{G_1}$, $\mathbf{G_2}$, $\mathbf{G_3}$, we have

$$((\mathbf{G_1} \parallel \mathbf{G_2}) \parallel \mathbf{G_3}) = (\mathbf{G_1} \parallel (\mathbf{G_2} \parallel \mathbf{G_3}))$$
$$\mathbf{sync}(\mathbf{sync}(\mathbf{G_1}, \mathbf{G_2}), \mathbf{G_3}) = \mathbf{sync}(\mathbf{G_1}, \mathbf{sync}(\mathbf{G_2}, \mathbf{G_3}))$$

2.5 Predicates and Predicate Transformers

2.5.1 Predicates

Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a DES. A *predicate* P defined on Q is a function $P : Q \to \{1, 0\}$. P is identified by a corresponding state subset

[2] The operator is same as the synchronous product operator on languages, but they can be easily distinguished by context.

$$Q_P := \{q \in Q | P(q) = 1\} \subseteq Q.$$

If $q \in Q_P$, we write $q \models P$ and say q *satisfies* P or P *includes* q. So, $q \models P$ if and only if $P(q) = 1$. We write $Pred(Q)$ for the set of all predicates defined on Q, so $Pred(Q)$ can be identified by $Pwr(Q)$. With the predicates, we can build boolean expression using the following operations:

$(\neg P)(q) = 1$	iff	$P(q) = 0$
$(P_1 \wedge P_2)(q) = 1$	iff	$P_1(q) = 1$ and $P_2(q) = 1$
$(P_1 \vee P_2)(q) = 1$	iff	$P_1(q) = 1$ or $P_2(q) = 1$
$(P_1 - P_2)(q) = 1$	iff	$P_1(q) = 1$ and $P_2(q) = 0$

where $P, P_1, P_2 \in Pred(Q)$ and $q \in Q$. Clearly, $P_1 - P_2 = P_1 \wedge \neg P_2$.

The two special predicates *true* and *false* are identified by Q and $\emptyset$ respectively. The predicate P_m is identified by Q_m.

For convenience, if Q is understood, for $Q_1 \subseteq Q$, denote the predicate identified by Q_1 as $pr(Q_1)$. For a predicate $P_1 \in Pred(Q)$, denote the corresponding state subset Q_{P_1} as $st(P_1)$.

The relation $\preceq$ on $Pred(Q)$ is defined as

$$(\forall P_1, P_2 \in Pred(Q))\ P_1 \preceq P_2 \text{ iff } P_1 \wedge P_2 = P_1.$$

It can easily be shown that $\preceq$ is a partial order on $Pred(Q)$. Clearly, $P_1 \preceq P_2$ iff $Q_{P_1} \subseteq Q_{P_2}$, so $(\forall q \in Q)\ q \models P_1 \Rightarrow q \models P_2$.

If $P_1 \preceq P_2$, we say that P_1 is *stronger* than P_2, and we also say that P_1 is a *subpredicate* of P_2.

Under the identification of $Pred(Q)$ with $Pwr(Q)$ and $\preceq$ with $\subseteq$, it is clear that $(Pred(Q), \preceq)$ is a complete lattice. The top element is *true*, and the bottom element

is $false$.

For a predicate $P \in Pred(Q)$, we define $Sub(P)$ to be the set of all the sub-predicates of P. Clearly, $Sub(P)$ is identified by $Pwr(Q_P)$, and $(Sub(P), \preceq)$ is also a complete lattice with top element P and bottom element $false$.

Proposition 2.3. *Let Q be the state set of a DES $\mathbf{G}$ with n states, $n \in \{0, 1, \ldots\}$, and let P be a predicate on Q. If a function $f : Sub(P) \to Sub(P)$ is a monotone function with respect to $\preceq$, then there exists $k \in \{0, 1, \ldots\}$ such that $k \leq n$ and $f^k(P)$ is the* greatest fixed point *of f with respect to $(Sub(P), \preceq)$, and $(\forall i \in \{1, 2, \ldots\})$ $i > k \Rightarrow f^i(P) = f^k(P)$.*

proof:

Immediately from Proposition 2.1 and the identifying relationship of $Sub(P)$ with $Pwr(Q_P)$, $\preceq$ with $\subseteq$, P with Q_P, and $false$ with $\emptyset$.

□

2.5.2 Predicate Transformers

Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a fixed DES. A *predicate transformer* is a function $f : Pred(Q) \to Pred(Q)$. We now introduce several predicate transformers which will be used later on.

- $R(\mathbf{G}, .)$

  Let $P \in Pred(Q)$. The *reachability predicate* $R(\mathbf{G}, P)$ is defined to hold precisely on those states that can be reached in $\mathbf{G}$ from q_0 via states satisfying P. Formally,

  1. $q_0 \models P \Rightarrow q_0 \models R(\mathbf{G}, P)$
  2. $q \models R(\mathbf{G}, P)$ & $\sigma \in \Sigma$ & $\delta(q, \sigma)!$ & $\delta(q, \sigma) \models P \Rightarrow \delta(q, \sigma) \models R(\mathbf{G}, P)$

3. No other states satisfy $R(\mathbf{G}, P)$.

In other words, $q \models R(\mathbf{G}, P)$ if and only if there exists a path in $\mathbf{G}$ from q_0 to q and each state on the path satisfies P. Therefore, $q_0 \not\models P \Rightarrow R(\mathbf{G}, P) = false$.

Clearly, $R(\mathbf{G}, P) \preceq P$ and $R(\mathbf{G}, .)$ is monotone with respect to $\preceq$. Note that $R(\mathbf{G}, true)$ exactly includes all the reachable states in Q.

- $CR(\mathbf{G}, .)$

  Let $P \in Pred(Q)$. The *coreachability predicate* $CR(\mathbf{G}, P)$ is defined to hold precisely on those states that can reach a marker state in $\mathbf{G}$ via states satisfying P. Formally,

  1. $P_m \wedge P = false \Rightarrow CR(\mathbf{G}, P) = false$
  2. $q \models P_m \wedge P \Rightarrow q \models CR(\mathbf{G}, P)$
  3. $q \models CR(\mathbf{G}, P)$ & $q' \models P$ & $\sigma \in \Sigma$ & $\delta(q', \sigma)!$ & $\delta(q', \sigma) = q \;\Rightarrow\; q' \models CR(\mathbf{G}, P)$
  4. No other states satisfy $CR(\mathbf{G}, P)$.

  In other words, $q \models CR(\mathbf{G}, P)$ if and only if there exists a path in $\mathbf{G}$ from q to a marker state satisfying P and each state on the path satisfies P.

  Clearly, $CR(\mathbf{G}, P) \preceq P$ and $CR(\mathbf{G}, .)$ is monotone with respect to $\preceq$. Note that $CR(\mathbf{G}, true)$ exactly includes all the coreachable states in Q.

- $TR(\mathbf{G}, ., \Sigma')$

  Let $\Sigma' \subseteq \Sigma$. With $\mathbf{G}$ and Σ' fixed, $TR(\mathbf{G}, ., \Sigma')$ is a predicate transformer. Let $P \in Pred(Q)$. $TR(\mathbf{G}, P, \Sigma')$ is defined to hold precisely on those states that can reach a state satisfying P in $\mathbf{G}$ only via transitions with events in Σ'. Formally,

1. $P = false \Rightarrow TR(\mathbf{G}, P, \Sigma') = false$
2. $q \models P \Rightarrow q \models TR(\mathbf{G}, P, \Sigma')$
3. $q \models TR(\mathbf{G}, P, \Sigma')$ & $q' \in Q$ & $\sigma \in \Sigma'$ & $\delta(q', \sigma)!$ & $\delta(q', \sigma) = q \ \Rightarrow \ q' \models TR(\mathbf{G}, P, \Sigma')$
4. No other states satisfy $TR(\mathbf{G}, P, \Sigma')$.

In other words, $q \models TR(\mathbf{G}, P, \Sigma')$ if and only if there exists a path in $\mathbf{G}$ from q to a state satisfying P and each transition event in the path is in Σ'. Note that the states on the path (excluding the end point state) do not need to satisfy P.

Clearly, $P \preceq TR(\mathbf{G}, P, \Sigma')$, and $TR(\mathbf{G}, P, \Sigma')$ is monotone with respect to $\preceq$.

- $\mathcal{CR}(\mathbf{G}, P', \Sigma', .)$

  Let $P' \in Pred(Q)$ and $\Sigma' \subseteq \Sigma$. With $\mathbf{G}, P'$ and Σ' fixed, $\mathcal{CR}(\mathbf{G}, P', \Sigma', .)$ is also a predicate transformer. Let $P \in Pred(Q)$. $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$ is defined to hold precisely on those states that can reach a state satisfying P' in $\mathbf{G}$ via states satisfying P and transitions with events in Σ'. Formally,

  1. $P' \wedge P = false \Rightarrow \mathcal{CR}(\mathbf{G}, P', \Sigma', P) = false$
  2. $q \models P' \wedge P \Rightarrow q \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P)$
  3. $q \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P)$ & $q' \models P$ & $\sigma \in \Sigma'$ & $\delta(q', \sigma)!$ & $\delta(q', \sigma) = q \ \Rightarrow \ q' \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P)$
  4. No other states satisfy $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$.

  In other words, $q \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P)$ if and only if there exists a path in $\mathbf{G}$ from q to a state which satisfies P' and each state on the path satisfies P and each transition event is in Σ'.

Clearly, $\mathcal{CR}(\mathbf{G}, P', \Sigma', P) \preceq P$ and $P' \wedge P \preceq \mathcal{CR}(\mathbf{G}, P', \Sigma', P)$. Note that $CR(\mathbf{G}, .)$ is actually a special case of $\mathcal{CR}(\mathbf{G}, P', \Sigma', .)$ with $P' = P_m$ and $\Sigma' = \Sigma$.

Proposition 2.4. *For a fixed* $\mathbf{G}$, *fixed predicate* $P' \in Pred(Q)$, *and* $\Sigma' \subseteq \Sigma$, *the predicate transformer* $\mathcal{CR}(\mathbf{G}, P', \Sigma', .)$ *is monotone with respect to* $\preceq$, *i.e.*

$$(\forall P_1, P_2 \in Pred(Q))P_1 \preceq P_2 \Rightarrow \mathcal{CR}(\mathbf{G}, P', \Sigma', P_1) \preceq \mathcal{CR}(\mathbf{G}, P', \Sigma', P_2)$$

proof:

Let $P_1, P_2 \in Pred(Q)$. Assume $P_1 \preceq P_2$. Must show this implies

$$\mathcal{CR}(\mathbf{G}, P', \Sigma', P_1) \preceq \mathcal{CR}(\mathbf{G}, P', \Sigma', P_2).$$

Let $q \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P_1)$. (1)

We now show implies $q \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P_2)$.

From (1), we know $(\exists k \in \{1, 2, \ldots\})(\exists q_1, q_2, \ldots, q_k \in Q)(\exists \sigma_1, \sigma_2, \ldots, \sigma_{k-1} \in \Sigma')$,

$$q_1 = q,\ q_k \models P'$$
$$q_{i+1} = \delta(q_i, \sigma_i), \quad i = 1, 2, \ldots, k-1$$
$$q_i \models P_1, \quad i = 1, 2, \ldots, k$$

By $P_1 \preceq P_2$, we have

$(\exists k \in \{1, 2, \ldots\})(\exists q_1, q_2, \ldots, q_k \in Q)(\exists \sigma_1, \sigma_2, \ldots, \sigma_{k-1} \in \Sigma')$

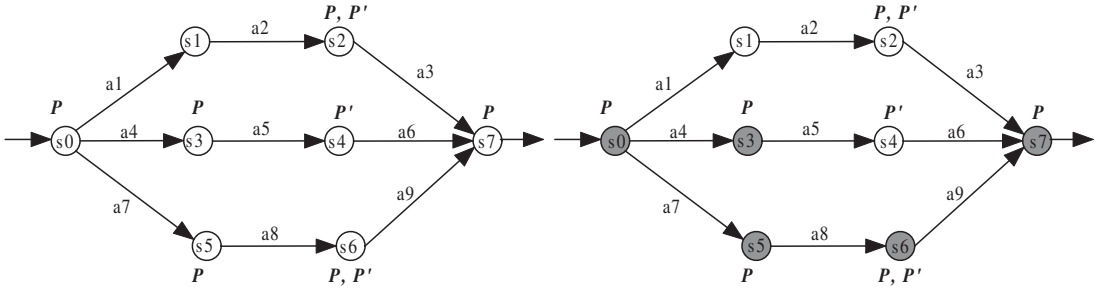
$$q_1 = q,\ q_k \models P'$$
$$q_{i+1} = \delta(q_i, \sigma_i), \quad i = 1, 2, \ldots, k-1$$
$$q_i \models P_2, \quad i = 1, 2, \ldots, k$$

$\Rightarrow q \models \mathcal{CR}(\mathbf{G}, P', \Sigma', P_2)$

□

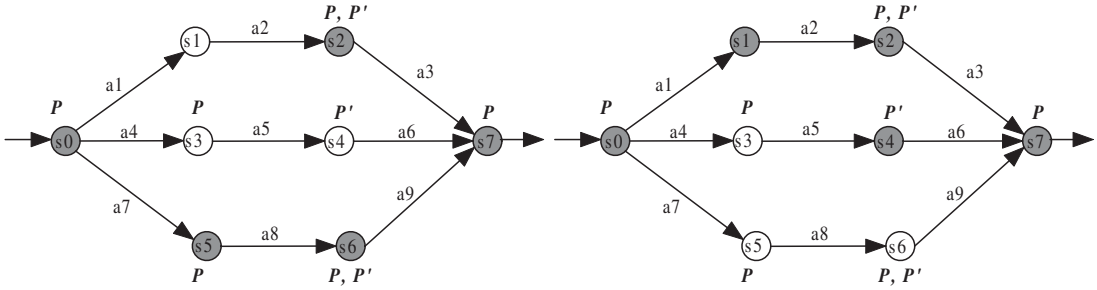
As these predicate transformers play important roles in our algorithms, we give an example for them. Let $\mathbf{G}$ be the DES shown in Figure 2.2(a). A state labeled

with a predicate (P or P') means that it satisfies that predicate. For example, state $s3$ satisfies P but not P', state $s1$ satisfies neither P nor P', and state $s2$ satisfies both P and P'. The other figures in Figure 2.2 show the result of the above predicate transformers. The states filled with gray color satisfy the corresponding predicate transformer result.



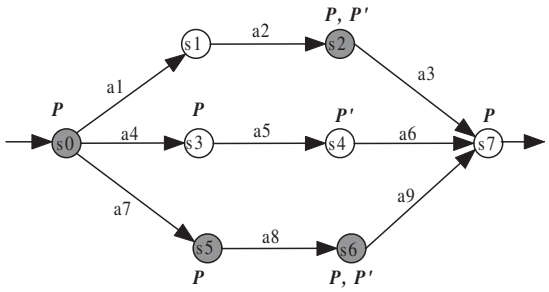
(a) Example **G**

(b) $R(\mathbf{G}, P) = pr(\{s0, s3, s5,\ s6, s7\})$



(c) $CR(\mathbf{G}, P) = pr(\{s0, s2, s5, s6, s7\})$

(d) $TR(\mathbf{G}, pr(\{s7\}), \{a1, a2, a3, a4, a6\}) = pr(\{s0, s1, s2, s4, s7\})$



(e) $\mathcal{CR}(\mathbf{G}, P', \{a1, a2, a4, a5, a7, a8\}, P) = pr(\{s0, s2, s5, s6\})$

Figure 2.2: Example for predicate transformers

2.5.3 Languages Induced by Predicates

Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a DES, and $P \in Pred(Q)$ be a predicate on Q. The language $L(\mathbf{G}, P)$ is the closed language *induced* by P. Informally, $L(\mathbf{G}, P)$ is the closed language generated by the DES $\mathbf{G}$ with only those states satisfying P. Formally,

$$L(\mathbf{G}, P) := \{w \in \Sigma^* | (\forall v \leq w)\delta(q_0, v) \models P\}.$$

Similarly, we define $L_m(\mathbf{G}, P)$ as the marked language induced by P.

$$L_m(\mathbf{G}, P) := \{w \in L(\mathbf{G}, P) | \delta(q_0, w) \in Q_m\}$$

We can define a DES $\mathbf{G}'$ based on DES $\mathbf{G}$ and P as follows. If $q_0 \not\models P$, then $\mathbf{G}'$ is defined as the **EMPTY** DES. Otherwise let $\mathbf{G}' := (Q', \Sigma, \delta', q_0, Q'_m)$, where $Q' := Q_P, Q'_m := Q_P \cap Q_m$, and the partial function $\delta' : Q' \times \Sigma \to Q'$ is defined according to

$$(\forall q' \in Q')(\forall \sigma \in \Sigma)\ \delta'(q', \sigma) := \begin{cases} \delta(q', \sigma), & \text{if } \delta(q', \sigma)! \ \&\ \delta(q', \sigma) \models P \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

Essentially, the DES $\mathbf{G}'$ is the DES $\mathbf{G}$ after removing all the states not satisfying P and all the transitions with source states or target states not satisfying P. Now one may think of $L(\mathbf{G}, P)$ and $L_m(\mathbf{G}, P)$ as $L(\mathbf{G}')$ and $L_m(\mathbf{G}')$ respectively.

The following two propositions will be used in following chapters.

Proposition 2.5. *Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a DES, then the following holds:*

$$(\forall P \in Pred(Q))(\forall s \in \Sigma^*)\ s \in L(\mathbf{G}, P) \Leftrightarrow \delta(q_0, s) \models R(\mathbf{G}, P).$$

proof:

Let $P \in Pred(Q)$ and $s \in \Sigma^*$. Must show implies $s \in L(\mathbf{G}, P) \Leftrightarrow \delta(q_0, s) \models R(\mathbf{G}, P)$.

$s \in L(\mathbf{G}, P)$

$\Leftrightarrow (\forall t \leq s)\ \delta(q_0, t) \models P, \quad$ by definition of $L(\mathbf{G}, P)$

$\Leftrightarrow (\exists n \in \{0, 1, 2, \ldots\})(\exists \sigma_0, \sigma_1, \ldots, \sigma_{n-1} \in \Sigma)(\exists q_1, q_2, \ldots, q_n \in Q)$

$$s = \sigma_0 \sigma_1 \cdots \sigma_{n-1}$$
$$q_n = \delta(q_0, s)$$
$$q_i \models P, \quad i = 0, 1, \ldots, n$$
$$q_{i+1} = \delta(q_i, \sigma_i), \quad i = 0, 1, \ldots, n-1$$

$\Leftrightarrow \delta(q_0, s) \models R(\mathbf{G}, P)$.

□

From this proposition, it is clear that the language $L(\mathbf{G}, P)$ only depends on those states satisfying $R(\mathbf{G}, P)$.

Proposition 2.6. *Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a DES, then the following holds:*

$$(\forall P \in Pred(Q))(\forall s, t \in L(\mathbf{G}, P))\ \delta(q_0, s) = \delta(q_0, t) \Rightarrow s \equiv_{L(\mathbf{G},P)} t\ \&\ s \equiv_{L_m(\mathbf{G},P)} t$$

where $\equiv_{L(\mathbf{G},P)}$ and $\equiv_{L_m(\mathbf{G},P)}$ are the Nerode equivalence relations on Σ^* with respect to $L(\mathbf{G}, P)$ and $L_m(\mathbf{G}, P)$ respectively.

proof:

Let $P \in Pred(Q), s \in L(\mathbf{G}, P)$ and $t \in L(\mathbf{G}, P)$. (1)

Assume $\delta(q_0, s) = \delta(q_0, t)$. (2)

1. Show $s \equiv_{L(\mathbf{G},P)} t$.

   Sufficient to show that $(\forall u \in \Sigma^*) su \in L(\mathbf{G}, P) \Leftrightarrow tu \in L(\mathbf{G}, P)$.

   Let $u \in \Sigma^*$. Must show the following two points.

(a) $su \in L(\mathbf{G}, P) \Rightarrow tu \in L(\mathbf{G}, P)$

Assume $su \in L(\mathbf{G}, P)$. (3)

We now show that $tu \in L(\mathbf{G}, P)$.

By (3) and Proposition 2.5, we know $\delta(q_0, su) \models R(\mathbf{G}, P)$. (4)

If $u = \epsilon$, then $tu = t \in L(\mathbf{G}, P)$ from (1). So, we assume $u \neq \epsilon$.

Let $u := \sigma_1\sigma_2 \cdots \sigma_n$, where $n \in \{1, 2, \ldots\}$ and $\sigma_1, \sigma_2, \ldots, \sigma_n \in \Sigma$.

By (4) and the definition of $R(\mathbf{G}, P)$, we have

$$\delta(q_0, s\sigma_1) \models P, \delta(q_0, s\sigma_1\sigma_2) \models P, \ldots, \delta(q_0, s\sigma_1\sigma_2 \cdots \sigma_n) \models P. \tag{5}$$

By (2) and $\mathbf{G}$ is deterministic, we have

$$\delta(q_0, s\sigma_1) = \delta(q_0, t\sigma_1),\ \delta(q_0, s\sigma_1\sigma_2) = \delta(q_0, t\sigma_1\sigma_2),\ \cdots,$$
$$\delta(q_0, s\sigma_1\sigma_2 \cdots \sigma_n) = \delta(q_0, t\sigma_1\sigma_2 \cdots \sigma_n). \tag{6}$$

By (5)(6), we have

$$\delta(q_0, t\sigma_1) \models P, \delta(q_0, t\sigma_1\sigma_2) \models P, \ldots, \delta(q_0, t\sigma_1\sigma_2 \cdots \sigma_n) \models P. \tag{7}$$

From (1), we know $t \in L(\mathbf{G}, P)$. From this and Proposition 2.5, we have

$$\delta(q_0, t) \models R(\mathbf{G}, P) \tag{8}$$

By (7),(8) and the definition of $R(\mathbf{G}, P)$, we have $\delta(q_0, tu) \models R(\mathbf{G}, P)$.

$\Rightarrow tu \in L(\mathbf{G}, P)$, by Proposition 2.5

(b) $tu \in L(\mathbf{G}, P) \Rightarrow su \in L(\mathbf{G}, P)$

Identical to Part 1(a) by exchanging s and t.

2. Show $s \equiv_{L_m(\mathbf{G},P)} t$.

Sufficient to show that $(\forall u \in \Sigma^*)\ su \in L_m(\mathbf{G}, P) \Leftrightarrow tu \in L_m(\mathbf{G}, P)$.

Let $u \in \Sigma^*$. Must show the following two points.

(a) $su \in L_m(\mathbf{G}, P) \Rightarrow tu \in L_m(\mathbf{G}, P)$

Assume $su \in L_m(\mathbf{G}, P)$. (9)

We now show that $tu \in L_m(\mathbf{G}, P)$.

By (9) and the definition of $L_m(\mathbf{G}, P)$, we have

$su \in L(\mathbf{G}, P)$ & $\delta(q_0, su) \in Q_m$

$\Rightarrow \delta(q_0, su) \models R(\mathbf{G}, P)$ & $\delta(q_0, su) \in Q_m$, by Proposition 2.5 (10)

If $u = \epsilon$, then $tu = t \in L_m(\mathbf{G}, P)$ from (1) and (2). So, we assume $u \neq \epsilon$.

Let $u := \sigma_1\sigma_2 \cdots \sigma_n$, where $n \in \{1, 2, \ldots\}$ and $\sigma_1, \sigma_2, \ldots, \sigma_n \in \Sigma$.

By (10) and the definition of $R(\mathbf{G}, P)$, we have

$$\delta(q_0, s\sigma_1) \models P, \delta(q_0, s\sigma_1\sigma_2) \models P, \ldots, \delta(q_0, s\sigma_1\sigma_2 \cdots \sigma_n) \models P$$
$$\delta(q_0, s\sigma_1\sigma_2 \cdots \sigma_n) \in Q_m \quad (11)$$

By (2) and $\mathbf{G}$ is deterministic, we have

$$\delta(q_0, s\sigma_1) = \delta(q_0, t\sigma_1),\ \delta(q_0, s\sigma_1\sigma_2) = \delta(q_0, t\sigma_1\sigma_2),\ \cdots,$$
$$\delta(q_0, s\sigma_1\sigma_2 \cdots \sigma_n) = \delta(q_0, t\sigma_1\sigma_2 \cdots \sigma_n). \quad (12)$$

By (11)(12), we have

$$\delta(q_0, t\sigma_1) \models P, \delta(q_0, t\sigma_1\sigma_2) \models P, \ldots, \delta(q_0, t\sigma_1\sigma_2 \cdots \sigma_n) \models P.$$
$$\delta(q_0, t\sigma_1\sigma_2 \cdots \sigma_n) \in Q_m \quad (13)$$

From (1), we know $t \in L(\mathbf{G}, P)$. From this and Proposition 2.5, we have

$$\delta(q_0, t) \models R(\mathbf{G}, P) \quad (14)$$

By (13),(14) and the definition of $R(\mathbf{G}, P)$, we have

$\delta(q_0, tu) \models R(\mathbf{G}, P)$ & $\delta(q_0, tu) \in Q_m$

$\Rightarrow tu \in L(\mathbf{G}, P)$ & $\delta(q_0, tu) \in Q_m$, by Proposition 2.5

$\Rightarrow tu \in L_m(\mathbf{G}, P)$, by definition of $L_m(\mathbf{G}, P)$

(b) $tu \in L_m(\mathbf{G}, P) \Rightarrow su \in L_m(\mathbf{G}, P)$

Identical to Part 2(a) by exchanging s and t.

□

Let $s \in L(\mathbf{G}, P)$. We define the following set of strings

$$L^s(\mathbf{G}, P) := \{t \in L(\mathbf{G}, P) | (\exists u \leq t)\ \delta(q_0, u) = \delta(q_0, s)\}.$$

We see that $L^s(\mathbf{G}, P)$ exactly includes all the strings in $L(\mathbf{G}, P)$ reaching or passing through the state $\delta(q_0, s)$.

From the definition of $L^s(\mathbf{G}, P)$, we have

$$L(\mathbf{G}, P) - L^s(\mathbf{G}, P) = \{t \in L(\mathbf{G}, P) | (\forall u \leq t)\ \delta(q_0, u) \neq \delta(q_0, s)\} \tag{2.1}$$

Let $K \subseteq L(\mathbf{G}, P)$. The following definition will be useful later on.

$$L^K(\mathbf{G}, P) = \bigcup_{s \in K} L^s(\mathbf{G}, P).$$

Proposition 2.7. *Let* $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ *be a DES,* $P, P' \in Pred(Q), s \in L(\mathbf{G}, P)$ *and* $q = \delta(q_0, s)$. *Let* $P' := P - pr(\{q\})$, *then the following holds*

$$L(\mathbf{G}, P') = L(\mathbf{G}, P) - L^s(\mathbf{G}, P)$$

proof:

1. Show that $L(\mathbf{G}, P') \subseteq L(\mathbf{G}, P) - L^s(\mathbf{G}, P)$.

   Let $t \in L(\mathbf{G}, P')$. (1)

We now show this implies $t \in L(\mathbf{G}, P) - L^s(\mathbf{G}, P)$.

By (1) and the fact $P' \preceq P$, we have $t \in L(\mathbf{G}, P)$. (2)

By (1) and the definition of $L(\mathbf{G}, P')$, we also have

$(\forall v \leq t)\ \delta(q_0, v) \models P'$

$\Rightarrow (\forall v \leq t)\ \delta(q_0, v) \neq q, \quad$ by definition of P'

$\Rightarrow (\forall v \leq t)\ \delta(q_0, v) \neq \delta(q_0, s)$

$\Rightarrow t \in L(\mathbf{G}, P) - L^s(\mathbf{G}, P), \quad$ by Equation 2.1 and (2)

2. Show that $L(\mathbf{G}, P) - L^s(\mathbf{G}, P) \subseteq L(\mathbf{G}, P')$.

   Let $t \in L(\mathbf{G}, P) - L^s(\mathbf{G}, P)$. (3)

   We now show this implies $t \in L(\mathbf{G}, P')$.

   By (3) and (2.1), we know $(\forall v \leq t)\ \delta(q_0, v) \neq \delta(q_0, s)$

   $\Rightarrow (\forall v \leq t)\ \delta(q_0, v) \neq q$

   $\Rightarrow (\forall v \leq t)\ \delta(q_0, v) \models P', \quad$ as $t \in L(\mathbf{G}, P)$

   $\Rightarrow t \in L(\mathbf{G}, P')$

□

This proposition states that if we remove all the strings in $L^s(\mathbf{G}, P)$, it is equivalent to removing the state $\delta(q_0, s)$ from P.

2.6 Supervisory Control

2.6.1 Controllable Languages

Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a DES with $\Sigma = \Sigma_c \dot{\cup} \Sigma_u$.

Let $K \subseteq \Sigma^*$. K is *controllable with respect to* $\mathbf{G}$ if

$$(\forall s \in \overline{K})(\forall \sigma \in \Sigma_u)\ s\sigma \in L(\mathbf{G}) \Rightarrow s\sigma \in \overline{K}$$

Let $S \subseteq \Sigma^*$, and $\Sigma_0 \subseteq \Sigma$. Denote $S\Sigma_0$ as the set $\{s\sigma | s \in S, \sigma \in \Sigma_0\}$. The condition for language K being controllable with respect to $\mathbf{G}$ can be rewritten as

$$\overline{K}\Sigma_u \cap L(\mathbf{G}) \subseteq \overline{K}$$

We can also use the Elig operator to express that K is controllable with respect to $\mathbf{G}$ as

$$(\forall s \in \overline{K} \cap L(\mathbf{G}))\ \mathrm{Elig}_{L(\mathbf{G})}(s) \cap \Sigma_u \subseteq \mathrm{Elig}_{\overline{K}}(s)$$

The above three representations for controllable language will be used in this thesis interchangeably.

Clearly, $\emptyset, L(\mathbf{G})$ and Σ^* are always controllable with respect to $\mathbf{G}$.

Let $E \subseteq \Sigma^*$ be an arbitrary language. Define the set of all sublanguages of E that are controllable with respect to $\mathbf{G}$ as

$$\mathcal{C}(E) := \{K \subseteq E | K \text{ is controllable with respect to } \mathbf{G}\}$$

Proposition 2.8 (From [47])**.** *$\mathcal{C}(E)$ is nonempty and is closed under arbitrary unions. In particular, the supremal element* $\sup\mathcal{C}(E) \in \mathcal{C}(E)$.

□

This proposition says that $\sup\mathcal{C}(E) \subseteq E$. Therefore, we can compute $\sup\mathcal{C}(E)$ by removing strings from E.

2.6.2 Supervisory Control

Let $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$ be a nonempty DES with $\Sigma = \Sigma_c \dot{\cup} \Sigma_u$. A *control pattern* is the union of Σ_u and a subset of Σ_c. Then, the set all of control patterns is

$$\Gamma := \{\gamma \in Pwr(\Sigma) | \gamma \supseteq \Sigma_u\},$$

where γ is a control pattern.

A *supervisory control for* $\mathbf{G}$ is any map $V : L(\mathbf{G}) \to \Gamma$. We write the pair $(\mathbf{G}, V)$ as $V/\mathbf{G}$. We also refer to $\mathbf{G}$ as the *plant* (DES).

The *closed behavior* of $V/\mathbf{G}$ is defined to be the language $L(V/\mathbf{G}) \subseteq L(\mathbf{G})$, which is described as follows.

1. $\epsilon \in L(V/\mathbf{G})$
2. If $s \in L(V/\mathbf{G}), \sigma \in V(s)$, and $s\sigma \in L(\mathbf{G})$ then $s\sigma \in L(V/\mathbf{G})$
3. No other strings belong to $L(V/\mathbf{G})$.

Clearly, $\epsilon \in L(V/\mathbf{G})$ and $L(V/\mathbf{G})$ is closed and $\{\epsilon\} \subseteq L(V/\mathbf{G}) \subseteq L(\mathbf{G})$.

The *marked behavior* of $V/\mathbf{G}$ is defined as $L_m(V/\mathbf{G}) = L(V/\mathbf{G}) \cap L_m(\mathbf{G})$. We always have $\emptyset \subseteq L_m(V/\mathbf{G}) \subseteq L(\mathbf{G})$.

V is a *nonblocking supervisory control(NSC) for* $\mathbf{G}$ if $\overline{L_m(V/\mathbf{G})} = L(V/\mathbf{G})$.

Usually, given a plant DES $\mathbf{G}$, we want the plant $\mathbf{G}$ to behave in a desired way. That means that we want a sublanguage of $L_m(\mathbf{G})$ which represents a set of desired tasks that the plant is supposed to complete. It is also desired that the controlled system be nonblocking. However, it is not always true that we can find a nonblocking supervisory control for a sublanguage of $L_m(\mathbf{G})$.

Let $K \subseteq L \subseteq \Sigma^*$. We say the language K is *L-closed* if $K = \overline{K} \cap L$.

Theorem 2.2 (From [47])**.** *Let $K \subseteq L_m(\mathbf{G}), K \neq \emptyset$. There exists a NSC V for $\mathbf{G}$ such that $L_m(V/\mathbf{G}) = K$ if and only if K is controllable with respect to $\mathbf{G}$ and K is $L_m(\mathbf{G})$-closed.*

□

The above supervisory control V does not play a role in marking a string, so the desired sublanguage K is required to be $L_m(\mathbf{G})$-*closed*. If we also allow V to include marking, then we can implement a more flexible supervisory control.

Let $K \subseteq L_m(\mathbf{G})$. We redefine the marked behavior of $V/\mathbf{G}$ as $L_m(V/\mathbf{G}) = L(V/\mathbf{G}) \cap K$. The map $V : L(\mathbf{G}) \to \Gamma$ is called a *marking nonblocking supervisory control (MNSC) for the pair* $(K, \mathbf{G})$ if $\overline{L_m(V/\mathbf{G})} = L(V/\mathbf{G})$.

Theorem 2.3 (From [47])**.** *Let $K \subseteq L_m(\mathbf{G}), K \neq \emptyset$. There exists a MNSC V for the pair $(K, \mathbf{G})$ such that $L_m(V/\mathbf{G}) = K$ if and only if K is controllable with respect to* $\mathbf{G}$. □

In this thesis, *we will focus on* MNSC.

2.6.3 Implementation of MNSC V for G

The supervisory control $V : L(\mathbf{G}) \to \Gamma$ is mainly used for theoretical purpose. In reality, it is not convenient to build such a map.

Let $K \subseteq L_m(\mathbf{G})$ be a desired marked sublanguage for a nonempty DES $\mathbf{G} = (Q, \Sigma, \delta, q_0, Q_m)$. Assume K is controllable. By Theorem 2.3, there exists a MNSC V such that $K = L_m(V/\mathbf{G}), \overline{K} = L(V/\mathbf{G})$.

In order to implement the language K, we construct a DES over $\Sigma = \Sigma_c \dot{\cup} \Sigma_u$ which we will refer to as a *supervisor*, say $\mathbf{S}$, such that $K = L_m(\mathbf{S} \times \mathbf{G}) \subseteq L_m(\mathbf{G})$ and $L(\mathbf{S} \times \mathbf{G}) = \overline{K}$. Then we say that $\mathbf{S}$ implements V. In order to ensure that S implements V, by Theorem 2.3 and the MNSC definition, the following *verification* conditions are required.

$$\begin{aligned} &1.\ L_m(\mathbf{S} \times \mathbf{G}) \text{ is controllable with respect to } \mathbf{G}. \\ &2.\ \overline{L_m(\mathbf{S} \times \mathbf{G})} = L(\mathbf{S} \times \mathbf{G}). \end{aligned} \tag{2.2}$$

The first condition can be written as

$$\overline{L_m(\mathbf{S} \times \mathbf{G})}\Sigma_u \cap L(\mathbf{G}) \subseteq \overline{L_m(\mathbf{S} \times \mathbf{G})}$$

As we also require the second condition to hold, the first condition is equivalent to

$$L(\mathbf{S} \times \mathbf{G})\Sigma_u \cap L(\mathbf{G}) \subseteq L(\mathbf{S} \times \mathbf{G})$$

A supervisor $\mathbf{S}$ is *controllable with respect to* $\mathbf{G}$ if it satisfies the first condition. A supervisor $\mathbf{S}$ is *nonblocking for* $\mathbf{G}$ if it satisfies the second condition. The second condition also states that $\mathbf{S} \times \mathbf{G}$ is nonblocking.

In [21], the first condition is defined as $L(\mathbf{S})\Sigma_u \cap L(\mathbf{G}) \subseteq L(\mathbf{S})$, we now show that our condition is equivalent to it.

Proposition 2.9. *Let* $\mathbf{G}$ *be a plant DES,* $\mathbf{S}$ *be a supervisor for* $\mathbf{G}$*, and both* $\mathbf{G}$ *and* $\mathbf{S}$ *are defined over event set* $\Sigma = \Sigma_c \dot{\cup} \Sigma_u$*. Then we have*

$$L(\mathbf{S} \times \mathbf{G})\Sigma_u \cap L(\mathbf{G}) \subseteq L(\mathbf{S} \times \mathbf{G}) \textit{ iff } L(\mathbf{S})\Sigma_u \cap L(\mathbf{G}) \subseteq L(\mathbf{S})$$

proof:

1. (if) Assume $L(\mathbf{S})\Sigma_u \cap L(\mathbf{G}) \subseteq L(\mathbf{S})$. (1)
   Must show this implies $L(\mathbf{S} \times \mathbf{G})\Sigma_u \cap L(\mathbf{G}) \subseteq L(\mathbf{S} \times \mathbf{G})$.

   Let $s \in L(\mathbf{S} \times \mathbf{G})\Sigma_u \cap L(\mathbf{G})$. (2)

   We now show implies $s \in L(\mathbf{S} \times \mathbf{G})$.

   By (2) we know $s \in L(\mathbf{S})\Sigma_u \cap L(\mathbf{G})$ & $s \in L(\mathbf{G})$

   $\Rightarrow s \in L(\mathbf{S})$ & $s \in L(\mathbf{G})$, by (1)

   $\Rightarrow s \in L(\mathbf{S} \times \mathbf{G})$, by definition of $L(\mathbf{S} \times \mathbf{G})$

2. (only if) Assume $L(\mathbf{S} \times \mathbf{G})\Sigma_u \cap L(\mathbf{G}) \subseteq L(\mathbf{S} \times \mathbf{G})$. (3)
   Must show this implies $L(\mathbf{S})\Sigma_u \cap L(\mathbf{G}) \subseteq L(\mathbf{S})$

   Let $s \in L(\mathbf{S})\Sigma_u \cap L(\mathbf{G})$. (4)

   We now show implies $s \in L(\mathbf{S})$.

   By (4), we know $(\exists s' \in L(\mathbf{S}))(\exists \sigma \in \Sigma_u)\ s = s'\sigma\ \&\ s \in L(\mathbf{G})$ (5)

   $\Rightarrow s' \in L(\mathbf{S})\ \&\ s'\sigma \in L(\mathbf{G})$

   $\Rightarrow s' \in L(\mathbf{S})\ \&\ s' \in L(\mathbf{G})\ \&\ s'\sigma \in L(\mathbf{G}),$ as $L(\mathbf{G})$ is closed

   $\Rightarrow s' \in L(\mathbf{S} \times \mathbf{G})\ \&\ s'\sigma \in L(\mathbf{G})$

   $\Rightarrow s'\sigma \in L(\mathbf{S} \times \mathbf{G})\Sigma_u\ \&\ s'\sigma \in L(\mathbf{G})$

   $\Rightarrow s'\sigma \in L(\mathbf{S} \times \mathbf{G}),$ by (3)

   $\Rightarrow s'\sigma \in L(\mathbf{S}),$ by definition of $L(\mathbf{S} \times \mathbf{G})$

   $\Rightarrow s \in L(\mathbf{S}),$ by (5)

□

Except for some trivial systems, usually both the plant $\mathbf{G}$ and the supervisor $\mathbf{S}$ are modeled as a group of *modular(component)* DES. The final $\mathbf{G}$ and $\mathbf{S}$ are the synchronous product of their modular components. For convenience, we sometimes add some artificial events to supervisor DES. The events must be selflooped at each state of the plant $\mathbf{G}$. Similarly, the supervisors may not care about some events in the plant DES, so those events may not appear in any of the supervisor DES. Then, we have to selfloop all the "don't care events" at each state of $\mathbf{S}$.

Let $\mathbf{S} := \mathbf{S}_1 \parallel \cdots \parallel \mathbf{S}_m$, $\mathbf{G} := \mathbf{G}_1 \parallel \cdots \parallel \mathbf{G}_n$, $m, n \in \{1, 2, \ldots\}$. Let $S' := \mathbf{selfloop}(\mathbf{S}, \Sigma_G - \Sigma_S)$ and $\mathbf{G}' := \mathbf{selfloop}(\mathbf{G}, \Sigma_S - \Sigma_G)$, where Σ_S and Σ_G are the event set for $\mathbf{S}$ and $\mathbf{G}$ respectively. The verification conditions (Equation 2.2) become

1. $L(\mathbf{S}' \times \mathbf{G}')\Sigma_u \cap L(\mathbf{G}') \subseteq L(\mathbf{S}' \times \mathbf{G}')$,

2. $\overline{L_m(\mathbf{S}' \times \mathbf{G}')} = L(\mathbf{S}' \times \mathbf{G}'))$.

2.6.4 Supervisor Synthesis

Sometimes, it is difficult to design a controllable and nonblocking supervisor $\mathbf{S}$ for a plant $\mathbf{G}$, especially for a complicated system.

To this end, we need a synthesis method to build a controllable and nonblocking supervisor $\mathbf{S}$ from a specification DES $\mathbf{E}$, which only cares about the system requirements. However, $\mathbf{E}$ is usually not controllable with respect to $\mathbf{G}$ and $\mathbf{E} \times \mathbf{G}$ may block, so we would like to find the supremal sublanguage $\sup \mathcal{C}(L_m(\mathbf{E} \times \mathbf{G}))$. Then according to Theorem 2.3, a MNSC V can be built. If we construct a nonblocking DES **KDES** representing $\sup \mathcal{C}(L_m(\mathbf{E} \times \mathbf{G}))$, then **KDES** implements V.

Chapter 3

HISC Overview

The Hierarchical Interface-based Supervisory Control (HISC) framework was proposed by Leduc *et al.* in [21–27] to alleviate the state explosion problem. In this chapter, we first give an overview of the HISC system structure and then give the definitions of high-level and low-level proper supervisors.

3.1 System Structure

An HISC system currently is a two-level system which includes one *high-level subsystem* and n *low-level subsystems* ($n \geq 1$). The high-level subsystem communicates with each low-level subsystem through a separate *interface*. Figure 3.1 shows the system block diagram and the conceptual flow of information. In [21], if $n = 1$, the system is called a *serial interface system*, otherwise the system is called a n^{th} *degree parallel interface system*. Because the serial interface system is a special case of a parallel interface system, here we only discuss the parallel interface system and call the n^{th} degree parallel interface system as *the* n^{th} *degree interface system*. All the subsystems and interfaces are modeled as DES automata. In order to restrict

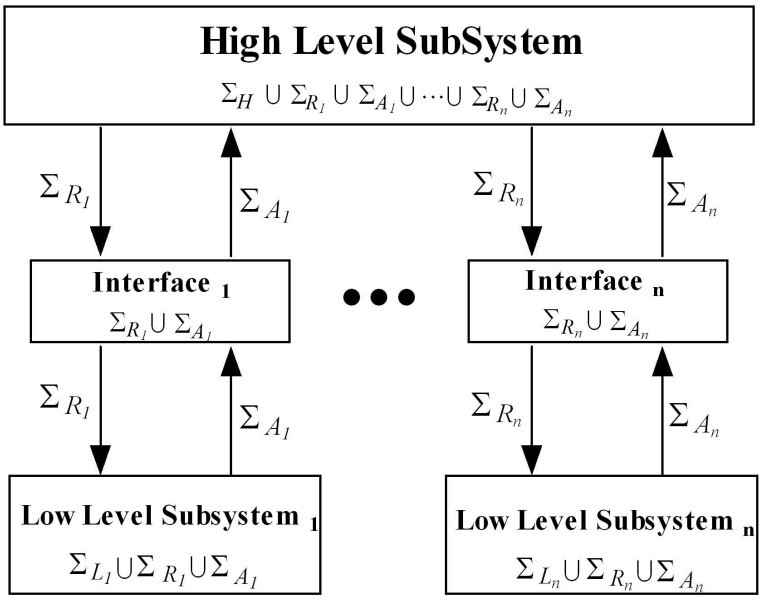


Figure 3.1: HISC block diagram

the information flow at the interface, for an n^{th} degree interface system, the system alphabet is partitioned into pairwise disjoint alphabets:

$$\Sigma := \Sigma_H \dot{\cup} \dot{\bigcup_{k\in\{1,\ldots,n\}}} [\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}] \tag{3.1}$$

In the remainder of this chapter, j is always an index with range $\{1, \ldots, n\}$. The high-level subsystem is modeled by DES $\mathbf{G}_H$, which is the product of *the high-level plant* $\mathbf{G}_H^p$ and *the high-level supervisor* $\mathbf{S}_H$ (both are defined over event set $\Sigma_H \dot{\cup} (\dot{\cup}_{k\in\{1,\ldots,n\}} [\Sigma_{R_k} \dot{\cup} \Sigma_{A_k}])$). The j^{th} low-level subsystem is modeled by DES $\mathbf{G}_{L_j}$, which is the product of *the j^{th} low-level plant* $\mathbf{G}_{L_j}^p$ and *the j^{th} low-level supervisor* $\mathbf{S}_{L_j}$ (both are defined over event set $\Sigma_{L_j} \dot{\cup} \Sigma_{R_j} \dot{\cup} \Sigma_{A_j}$), and the j^{th} interface is modeled by $\mathbf{G}_{I_j}$(defined over event set $\Sigma_{R_j} \dot{\cup} \Sigma_{A_j}$). The description for each event partition is as follows:

Σ_H : The set of *high-level events*, exist only in high-level subsystem.

Σ_{R_j} : The set of *request events* for the j^{th} interface.

Σ_{A_j} : The set of *answer events* for the j^{th} interface.

Σ_{L_j} : The set of j^{th} *low-level events*, exist only in the j^{th} low-level subsystem.

For controllability, the event set Σ is also partitioned as $\Sigma = \Sigma_c \dot{\cup} \Sigma_u$, where Σ_c is the controllable event set and Σ_u is the uncontrollable event set.

We refer to DES $\boldsymbol{\mathcal{G}}_H := \mathbf{G}_H \parallel \mathbf{G}_{I_1} \parallel \ldots \parallel \mathbf{G}_{I_n}$ as the *high-level* and DES $\boldsymbol{\mathcal{G}}_{L_j} := \mathbf{G}_{I_j} \parallel \mathbf{G}_{L_j}$ as the j^{th} *low-level*. For convenience, the following event sets are also defined:

$\Sigma_{I_j} := \Sigma_{R_j} \dot{\cup} \Sigma_{A_j}$	The set of *interface events* for the j^{th} interface
$\Sigma_A := \dot{\cup}_{k \in \{1,\ldots,n\}} \Sigma_{A_k}$	The set of all the answer events
$\Sigma_{IH} := (\dot{\cup}_{k \in \{1,\ldots,n\}} \Sigma_{I_k}) \dot{\cup} \Sigma_H$	The set of *interface and high-level events*
$\Sigma_{IL_j} := \Sigma_{L_j} \dot{\cup} \Sigma_{I_j}$	The set of j^{th} *interface and low-level events*
$\Sigma_{IL} := \dot{\cup}_{k \in \{1,\ldots,n\}} \Sigma_{IL_k}$	The set of all interface and low-level events
$\Sigma_{hu} := \Sigma_{IH} \cap \Sigma_u$	The set of *the high-level uncontrollable events*
$\Sigma_{hc} := \Sigma_{IH} \cap \Sigma_c$	The set of *the high-level controllable events*
$\Sigma_{lu_j} := \Sigma_{IL_j} \cap \Sigma_u$	The set of *the j^{th} low-level uncontrollable events*
$\Sigma_{lc_j} := \Sigma_{IL_j} \cap \Sigma_c$	The set of *the j^{th} low-level controllable events*

Then the high-level subsystem and the high-level are defined over event set Σ_{IH}. The j^{th} interface is defined over event set Σ_{I_j}, and the j^{th} low-level subsystem and the j^{th} low-level are defined over event set Σ_{IL_j}. We also have $\Sigma_{IH} = \Sigma_{hu} \dot{\cup} \Sigma_{hc}$ and $\Sigma_{IL_j} = \Sigma_{lu_j} \dot{\cup} \Sigma_{lc_j}$. The overall system structure is shown in Figure 3.2.

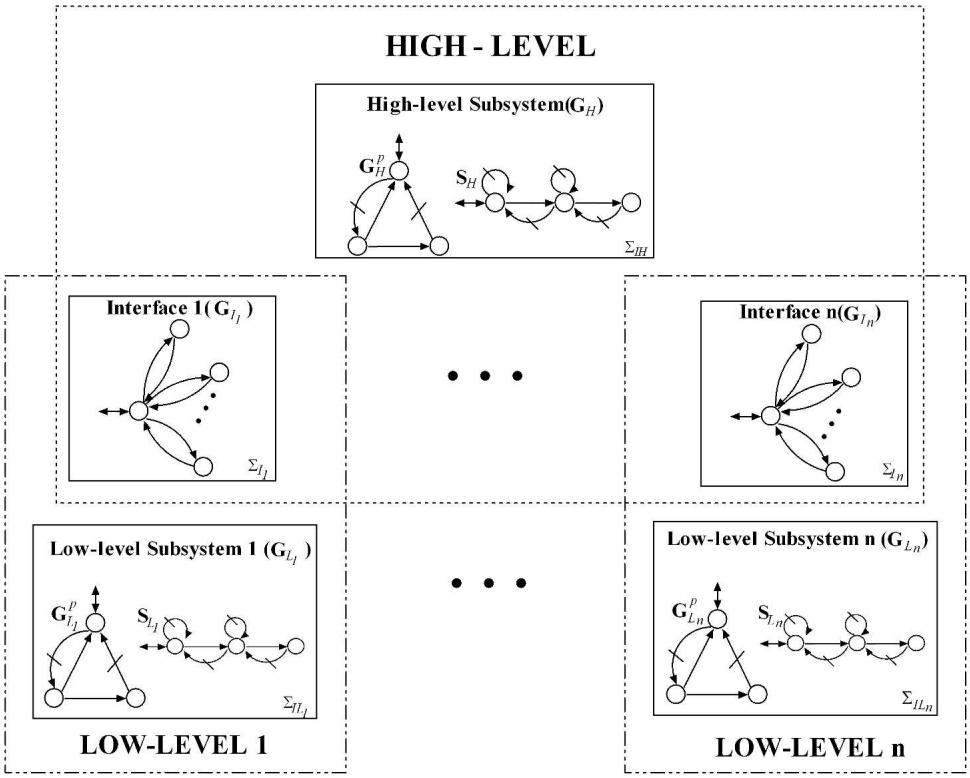


Figure 3.2: HISC system structure

3.1.1 Command-pair Interfaces

The interface DES play a very important role in the HISC structure. The high-level subsystem $\mathbf{G}_H$ send requests through an interface to a low-level subsystem. Once the low-level subsystem completes the requested job, it sends back an answer through the interface to the high-level subsystem. However, until the low-level subsystem completes the requested job, it can not accept another request from the high-level. To enforce such a mechanism, the interface DES is required to be a *command-pair interface* as defined below.

Definition 3.1. For the n^{th} degree interface system composed of plant components $\mathbf{G}_H^p, \mathbf{G}_{L_1}^p, \ldots, \mathbf{G}_{L_n}^p$, supervisors $\mathbf{S}_H, \mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, and interfaces $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$, the j^{th}

interface DES $\mathbf{G}_{I_j} = (X_j, \Sigma_{R_j} \dot{\cup} \Sigma_{A_j}, \xi_j, x_{j_0}, X_{j_m})$ is a *command-pair interface* if [1]:

(A) $L(\mathbf{G}_{I_j}) \subseteq \overline{(\Sigma_{R_j}.\Sigma_{A_j})^*}$

(B) $L_m(\mathbf{G}_{I_j}) = (\Sigma_{R_j}.\Sigma_{A_j})^* \cap L(\mathbf{G}_{I_j})$

$\diamondsuit$

An example command-pair interface from [23], with $\Sigma_{R_j} := \{\rho_i | i = 1, 2, 3\}$ and $\Sigma_{A_j} := \{\alpha_i | i = 1, \ldots, 7\}$, is shown in Figure 3.3.

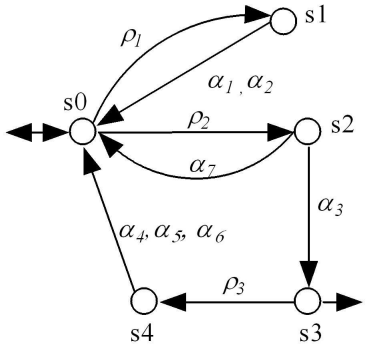


Figure 3.3: Example interface

3.1.2 Flat System

An HISC system is actually a structured *flat system*. The *flat plant* is defined as

$$\mathbf{PLANT} := \mathbf{G}^p_H \parallel \mathbf{G}^p_{L_1} \parallel \cdots \parallel \mathbf{G}^p_{L_n}$$

and the *flat supervisor* is defined as

$$\mathbf{SUP} := \mathbf{S}_H \parallel \mathbf{S}_{L_1} \parallel \cdots \parallel \mathbf{S}_{L_n} \parallel \mathbf{G}_{I_1} \parallel \cdots \parallel \mathbf{G}_{I_n}.$$

Therefore, both **PLANT** and **SUP** are defined over event set Σ. The whole flat system is the product of the flat supervisor and the flat plant.

[1] As we require $\mathbf{G}_{I_j}$ to be expressible as a tuple including initial state x_{j_0}, it thus can not be an empty DES. It follows that the empty string belongs to $L(\mathbf{G}_{I_j})$, and thus to $L_m(\mathbf{G}_{I_j})$ by point B.

$$\mathbf{SYSTEM} := \mathbf{SUP} \times \mathbf{PLANT}$$

As stated in Chapter 2, we want to ensure **SYSTEM** satisfies the following two properties.

1. $L(\mathbf{SYSTEM})\Sigma_u \cap L(\mathbf{PLANT}) \subseteq L(\mathbf{SYSTEM})$
2. $\overline{L_m(\mathbf{SYSTEM})} = L(\mathbf{SYSTEM})$

The first property is called *global controllability*, and the second property is called *global nonblocking*.

3.2 Local Conditions

One of the most important benefits of modeling a system as an HISC system is that we can guarantee global controllability and nonblocking by only checking local conditions on the high-level and each low-level separately. The conditions here are based on [23] and [27]. Please refer to them for a more detailed discussion. All our conditions here are based on either the high-level languages or the low-level languages. We do not directly use the conditions in [23] because our conditions are more convenient to prove the correctness of the predicate algorithms in the next chapters. All we do here mainly is to remove low-level event selfloops from the languages in high-level conditions in [23], and to remove high-level event selfloops from the languages in low-level conditions in [23].

Definition 3.2. For the n^{th} degree interface system that respects the alphabet partition given by 3.1 and is composed of plant components $\mathbf{G}^p_H, \mathbf{G}^p_{L_1}, \ldots, \mathbf{G}^p_{L_n}$, supervisors $\mathbf{S}_H, \mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, and interfaces $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$, for all $j \in \{1, \ldots, n\}$, define

$$\mathbf{G}^h_{I_j} := \mathbf{selfloop}(\mathbf{G}_{I_j}, \Sigma_{IH} - \Sigma_{I_j})$$

$$\mathbf{G}^l_{I_j} := \mathbf{selfloop}(\mathbf{G}_{I_j}, \Sigma_{L_j})$$

$\diamondsuit$

From the definition, $\mathbf{G}^h_{I_j}$ is defined over Σ_{IH}, and $\mathbf{G}^l_{I_j}$ is defined over Σ_{IL_j}. Now we define

$$\mathbf{G}^h_I := \mathbf{G}^h_{I_1} \times \cdots \times \mathbf{G}^h_{I_n}.$$

Therefore, the high-level $\boldsymbol{\mathcal{G}}_H$ and j^{th} low-level $\boldsymbol{\mathcal{G}}_{L_j}$ can be defined as

$$\boldsymbol{\mathcal{G}}_H = \mathbf{S}_H \times \mathbf{G}^p_H \times \mathbf{G}^h_I \qquad \boldsymbol{\mathcal{G}}_{L_j} = \mathbf{S}_{L_j} \times \mathbf{G}^p_{L_j} \times \mathbf{G}^l_{I_j}$$

3.2.1 Interface Consistent

Definition 3.3. The n^{th} degree interface system composed of plant components $\mathbf{G}^p_H, \mathbf{G}^p_{L_1}, \ldots, \mathbf{G}^p_{L_n}$, supervisors $\mathbf{S}_H, \mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, and interfaces $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$ is *interface consistent* with respect to the alphabet partition given by 3.1, if for all $j \in \{1, \ldots, n\}$, the following conditions are satisfied:

- Multi-level Properties

  1. The event set of $\mathbf{G}_H$ is Σ_{IH}, and the event set of $\mathbf{G}_{L_j}$is Σ_{IL_j}.

  2. $\mathbf{G}_{I_j}$ is a command-pair interface.

- High-level Properties

  3. $L(\mathbf{G}_H \times \prod_{\substack{k=1,\ldots,n \\ k\neq j}} \mathbf{G}^h_{I_k})\Sigma_{A_j} \cap P^{-1}_{I_j|\Sigma_{IH}}(L(\mathbf{G}_{I_j})) \subseteq L(\mathbf{G}_H \times \prod_{\substack{k=1,\ldots,n \\ k\neq j}} \mathbf{G}^h_{I_k})$,

     where $\prod_{\substack{k=1,\ldots,n \\ k\neq j}} \mathbf{G}^h_{I_k} := \mathbf{G}^h_{I_1} \times \cdots \times \mathbf{G}^h_{I_{j-1}} \times \mathbf{G}^h_{I_{j+1}} \times \cdots \times \mathbf{G}^h_{I_n}$;

     $P_{I_j|\Sigma_{IH}} : \Sigma^*_{IH} \to \Sigma^*_{I_j}$ is a natural projection.

- Low-level Properties

  4. $L(\mathbf{G}_{L_j})\Sigma_{R_j} \cap P^{-1}_{I_j|\Sigma_{IL_j}}(L(\mathbf{G}_{I_j})) \subseteq L(\mathbf{G}_{L_j})$,

     where $P_{I_j|\Sigma_{IL_j}} : \Sigma^*_{IL_j} \to \Sigma^*_{I_j}$ is a natural projection.

  5. $(\forall s \in \Sigma^*_{IL_j}.\Sigma_{R_j} \cap L(\boldsymbol{\mathcal{G}}_{L_j}))\ \text{Elig}_{L(\boldsymbol{\mathcal{G}}_{L_j})}(s\Sigma^*_{L_j}) \cap \Sigma_{A_j} = \text{Elig}_{L(\mathbf{G}_{I_j})}(P_{I_j}(s)) \cap \Sigma_{A_j}$,

     where $\text{Elig}_{L(\boldsymbol{\mathcal{G}}_{L_j})}(s\Sigma^*_{L_j}) := \cup_{l \in \Sigma^*_{L_j}} \text{Elig}_{L(\boldsymbol{\mathcal{G}}_{L_j})}(sl)$;

     $P_{I_j} : \Sigma^* \to \Sigma^*_{I_j}$ is a natural projection.

  6. $(\forall s \in L(\boldsymbol{\mathcal{G}}_{L_j}))\ \ P_{I_j}(s) \in L_m(\mathbf{G}_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ sl \in L_m(\boldsymbol{\mathcal{G}}_{L_j})$

$\diamondsuit$

For the purpose of later proofs, we will give an equivalent interface consistent definition in Definition 3.5. For convenience to prove the equivalence between Definition 3.3 and Definition 3.5, we first present an intermediate version of the interface consistent definition.

Definition 3.4. The n^{th} degree interface system composed of plant components $\mathbf{G}^p_H, \mathbf{G}^p_{L_1}, \ldots, \mathbf{G}^p_{L_n}$, supervisors $\mathbf{S}_H, \mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, and interfaces $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$ is *interface consistent* with respect to the alphabet partition given by 3.1, if for all $j \in \{1, \ldots, n\}$, the following conditions are satisfied:

- Multi-level Properties

  1. The event set of $\mathbf{G}^p_H$ and $\mathbf{S}_H$ is Σ_{IH}, and the event set of $\mathbf{G}^p_{L_j}$ and $\mathbf{S}_{L_j}$ is Σ_{IL_j}.

  2. $\mathbf{G}_{I_j}$ is a command-pair interface.

- High-level Properties

  3. $L(\boldsymbol{\mathcal{G}}_H)\Sigma_{A_j} \cap L(\mathbf{G}^h_{I_j}) \subseteq L(\boldsymbol{\mathcal{G}}_H)$

- Low-level Properties

  4. $L(\boldsymbol{\mathcal{G}}_{L_j})\Sigma_{R_j} \cap L(\mathbf{G}^l_{I_j}) \subseteq L(\boldsymbol{\mathcal{G}}_{L_j})$

  5. $(\forall s \in \Sigma^*_{IL_j}.\Sigma_{R_j} \cap L(\boldsymbol{\mathcal{G}}_{L_j}))$ $\quad \text{Elig}_{L(\boldsymbol{\mathcal{G}}_{L_j})}(s\Sigma^*_{L_j}) \cap \Sigma_{A_j} = \text{Elig}_{L(\mathbf{G}^l_{I_j})}(s) \cap \Sigma_{A_j}$,
     where $\text{Elig}_{L(\boldsymbol{\mathcal{G}}_{L_j})}(s\Sigma^*_{L_j}) := \cup_{l \in \Sigma^*_{L_j}} \text{Elig}_{L(\boldsymbol{\mathcal{G}}_{L_j})}(sl)$

  6. $(\forall s \in L(\boldsymbol{\mathcal{G}}_{L_j}))$ $\; s \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j}) sl \in L_m(\boldsymbol{\mathcal{G}}_{L_j})$

$\diamondsuit$

Proposition 3.1. *Definition 3.3 is equivalent to Definition 3.4.*

proof:

For the n^{th} degree interface system that respects the alphabet partition given by 3.1 and is composed of plant components $\mathbf{G}^p_H, \mathbf{G}^p_{L_1}, \ldots, \mathbf{G}^p_{L_n}$, supervisors $\mathbf{S}_H, \mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, and interfaces $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$, we prove the equivalence between each point in Definition 3.3 and Definition 3.4.

1. Point 1 in both definitions are equivalent.

   As we defined $\mathbf{G}_H := \mathbf{S}_H \times \mathbf{G}^p_H$ and $\mathbf{G}_{L_j} := \mathbf{S}_{L_j} \times \mathbf{G}^p_{L_j}$ in Section 3.1, Point 1 in both definitions are clearly equivalent.

2. Point 2 in both definitions are equivalent.

   Point 2 in both definitions are exactly the same.

3. Point 3 in both definitions are equivalent.

   By the definitions of $P_{I_j|\Sigma_{IH}}$ and $\mathbf{G}^h_{I_j}$, we have $P^{-1}_{I_j|\Sigma_{IH}}(L(\mathbf{G}_{I_j})) = L(\mathbf{G}^h_{I_j})$. Thus we know Point 3 in Definition 3.3 is equivalent to

   $$L(\mathbf{G}_H \times \prod_{\substack{k=1,\dots,n \\ k\neq j}} \mathbf{G}^h_{I_k})\Sigma_{A_j} \cap L(\mathbf{G}^h_{I_j}) \subseteq L(\mathbf{G}_H \times \prod_{\substack{k=1,\dots,n \\ k\neq j}} \mathbf{G}^h_{I_k}) \tag{1}$$

   We now show that (1) is equivalent to Point 3 in Definition 3.4.

   By the definition of $\boldsymbol{\mathcal{G}}_H$, we have $\boldsymbol{\mathcal{G}}_H = \mathbf{G}_H \times \mathbf{G}^h_{I_1} \times \cdots \times \mathbf{G}^h_{I_n}$.

   The rest proof of this point is identical to the proof of Proposition 2.9 by substituting Σ with Σ_{IH}, Σ_u with Σ_{A_j}, Σ_c with $\Sigma_{IH} - \Sigma_{A_j}$, $\mathbf{S}$ with $\mathbf{G}_H \times \prod_{\substack{k=1,\dots,n \\ k\neq j}} \mathbf{G}^h_{I_k}$, $\mathbf{G}$ with $\mathbf{G}^h_{I_j}$, and $\mathbf{S} \times \mathbf{G}$ with $\boldsymbol{\mathcal{G}}_H$.

4. Point 4 in both definitions are equivalent.

   By the definitions of $P_{I_j|\Sigma_{IL_j}}$ and $\mathbf{G}^l_{I_j}$, we have $P^{-1}_{I_j|\Sigma_{IL_j}}(L(\mathbf{G}_{I_j})) = L(\mathbf{G}^l_{I_j})$. Thus we know Point 4 in Definition 3.3 is equivalent to

   $$L(\mathbf{G}_{L_j})\Sigma_{A_j} \cap L(\mathbf{G}^l_{I_j}) \subseteq L(\mathbf{G}_{L_j}) \tag{2}$$

   We now show that (2) is equivalent to Point 4 in Definition 3.4.

   By the definition of $\boldsymbol{\mathcal{G}}_{L_j}$, we have $\boldsymbol{\mathcal{G}}_{L_j} = \mathbf{G}_{L_j} \times \mathbf{G}_{I_j}$.

   The rest proof of this point is identical to the proof of Proposition 2.9 by substituting Σ with Σ_{IL_j}, Σ_u with Σ_{R_j}, Σ_c with $\Sigma_{IL_j} - \Sigma_{R_j}$, $\mathbf{S}$ with $\mathbf{G}_{L_j}$, $\mathbf{G}$ with $\mathbf{G}^l_{I_j}$, and $\mathbf{S} \times \mathbf{G}$ with $\boldsymbol{\mathcal{G}}_{L_j}$.

5. Point 5 in both definitions are equivalent.

   By comparing Point 5 in Definition 3.3 and Point 5 in Definition 3.4, it is sufficient to show

$$(\forall s \in \Sigma^*_{IL_j}.\Sigma_{R_j} \cap L(\boldsymbol{\mathcal{G}}_{L_j}))\ \mathrm{Elig}_{L(\mathbf{G}^l_{I_j})}(s) \cap \Sigma_{A_j} = \mathrm{Elig}_{L(\mathbf{G}_{I_j})}(P_{I_j}(s)) \cap \Sigma_{A_j}.$$

Let $s \in \Sigma^*_{IL_j}.\Sigma_{R_j} \cap L(\boldsymbol{\mathcal{G}}_{L_j})$. (3)

(a) Show that $\mathrm{Elig}_{L(\mathbf{G}^l_{I_j})}(s) \cap \Sigma_{A_j} \subseteq \mathrm{Elig}_{L(\mathbf{G}_{I_j})}(P_{I_j}(s)) \cap \Sigma_{A_j}$

Let $\alpha \in \mathrm{Elig}_{L(\mathbf{G}^l_{I_j})}(s) \cap \Sigma_{A_j}$. (4)

We now show that $\alpha \in \mathrm{Elig}_{L(\mathbf{G}_{I_j})}(P_{I_j}(s)) \cap \Sigma_{A_j}$.

By (4), we know $s\alpha \in L(\mathbf{G}^l_{I_j})$

$\Rightarrow s\alpha \in P^{-1}_{I_j|\Sigma_{IL_j}}(L(\mathbf{G}_{I_j}))$, by the definitions of $\mathbf{G}^l_{I_j}$ and $P_{I_j|\Sigma_{IL_j}}$

$\Rightarrow P_{I_j|\Sigma_{IL_j}}(s\alpha) \in L(\mathbf{G}_{I_j})$

$\Rightarrow P_{I_j|\Sigma_{IL_j}}(s)\alpha \in L(\mathbf{G}_{I_j})$, as $\alpha \in \Sigma_{A_j}$ by (4)

From (3), we know $s \in \Sigma^*_{IL_j}$. By the definitions of $P_{I_j|\Sigma_{IL_j}}$ and P_{I_j}, we thus have $P_{I_j}(s)\alpha \in L(\mathbf{G}_{I_j})$

$\Rightarrow \alpha \in \mathrm{Elig}_{L(\mathbf{G}_{I_j})}(P_{I_j}(s)) \cap \Sigma_{A_j}$.

(b) Show that $\mathrm{Elig}_{L(\mathbf{G}_{I_j})}(P_{I_j}(s)) \cap \Sigma_{A_j} \subseteq \mathrm{Elig}_{L(\mathbf{G}^l_{I_j})}(s) \cap \Sigma_{A_j}$

Let $\alpha \in \mathrm{Elig}_{L(\mathbf{G}_{I_j})}(P_{I_j}(s)) \cap \Sigma_{A_j}$. (5)

We now show that $\alpha \in \mathrm{Elig}_{L(\mathbf{G}^l_{I_j})}(s) \cap \Sigma_{A_j}$.

By (5), we know $P_{I_j}(s)\alpha \in L(\mathbf{G}_{I_j})$

From (3), we know $s \in \Sigma^*_{IL_j}$. By the definitions of $P_{I_j|\Sigma_{IL_j}}$ and P_{I_j}, we thus have $P_{I_j|\Sigma_{IL_j}}(s)\alpha \in L(\mathbf{G}_{I_j})$

$\Rightarrow P_{I_j|\Sigma_{IL_j}}(s\alpha) \in L(\mathbf{G}_{I_j})$, as $\alpha \in \Sigma_{A_j}$ by (5)

$\Rightarrow s\alpha \in P^{-1}_{I_j|\Sigma_{IL_j}}(L(\mathbf{G}_{I_j}))$

$\Rightarrow s\alpha \in L(\mathbf{G}^l_{I_j})$, by the definitions of $\mathbf{G}^l_{I_j}$ and $P_{I_j|\Sigma_{IL_j}}$

$\Rightarrow \alpha \in \mathrm{Elig}_{L(\mathbf{G}^l_{I_j})}(s) \cap \Sigma_{A_j}$.

6. Point 6 in both definitions are equivalent.

   By comparing Point 6 in Definition 3.3 and Point 6 in Definition 3.4, it is sufficient to show

   $$(\forall s \in L(\boldsymbol{\mathcal{G}}_{L_j}))\ \ s \in L_m(\mathbf{G}^l_{I_j}) \text{ iff } P_{I_j}(s) \in L_m(\mathbf{G}_{I_j})$$

   Let $s \in L(\boldsymbol{\mathcal{G}}_{L_j})$.

   $\Rightarrow s \in \Sigma^*_{IL_j}$, by the fact that $\boldsymbol{\mathcal{G}}_{L_j}$ is defined over Σ_{IL_j}. (6)

   Let function $P_{I_j|\Sigma_{IL_j}} : \Sigma^*_{IL_j} \to \Sigma^*_{I_j}$ be a natural projection.

   (a) Show that $s \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow P_{I_j}(s) \in L_m(\mathbf{G}_{I_j})$.

   Assume $s \in L_m(\mathbf{G}^l_{I_j})$. Must show this implies $P_{I_j}(s) \in L_m(\mathbf{G}_{I_j})$.

   From assumption $s \in L_m(\mathbf{G}^l_{I_j})$, by the definitions of $P_{I_j|\Sigma_{IL_j}}$ and $\mathbf{G}^l_{I_j}$, we have $s \in P^{-1}_{I_j|\Sigma_{IL_j}}(L_m(\mathbf{G}_{I_j}))$.

   $\Rightarrow P_{I_j|\Sigma_{IL_j}}(s) \in L_m(\mathbf{G}_{I_j})$

   $\Rightarrow P_{I_j}(s) \in L_m(\mathbf{G}_{I_j})$, by (6) and the definitions of P_{I_j} and $P_{I_j|\Sigma_{IL_j}}$.

   (b) Show that $P_{I_j}(s) \in L_m(\mathbf{G}_{I_j}) \Rightarrow s \in L_m(\mathbf{G}^l_{I_j})$

   Assume $P_{I_j}(s) \in L_m(\mathbf{G}_{I_j})$. Must show this implies $s \in L_m(\mathbf{G}^l_{I_j})$.

   From assumption $P_{I_j}(s) \in L_m(\mathbf{G}_{I_j})$, by (6) and the definitions of P_{I_j} and $P_{I_j|\Sigma_{IL_j}}$, we have $P_{I_j|\Sigma_{IL_j}}(s) \in L_m(\mathbf{G}_{I_j})$.

   $\Rightarrow s \in P^{-1}_{I_j|\Sigma_{IL_j}}(L_m(\mathbf{G}_{I_j}))$

   $\Rightarrow s \in L_m(\mathbf{G}^l_{I_j})$, by the definitions of $P_{I_j|\Sigma_{IL_j}}$ and $\mathbf{G}^l_{I_j}$.

□

We now give our final interface consistent definition. We will always use this definition in the rest of this thesis.

Definition 3.5. The n^{th} degree interface system composed of plant components $\mathbf{G}^p_H, \mathbf{G}^p_{L_1}, \ldots, \mathbf{G}^p_{L_n}$, supervisors $\mathbf{S}_H, \mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, and interfaces $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$ is *interface consistent* with respect to the alphabet partition given by 3.1, if for all $j \in \{1, \ldots, n\}$, the following conditions are satisfied[2]:

- Multi-level Properties

  1. The event set of $\mathbf{G}^p_H$ and $\mathbf{S}_H$ is Σ_{IH}, and the event set of $\mathbf{G}^p_{L_j}$ and $\mathbf{S}_{L_j}$ is Σ_{IL_j}.
  2. $\mathbf{G}_{I_j}$ is a command-pair interface.

- High-level Properties

  3. $L(\boldsymbol{\mathcal{G}}_H)\Sigma_{A_j} \cap L(\mathbf{G}^h_{I_j}) \subseteq L(\boldsymbol{\mathcal{G}}_H)$

- Low-level Properties

  4. $L(\boldsymbol{\mathcal{G}}_{L_j})\Sigma_{R_j} \cap L(\mathbf{G}^l_{I_j}) \subseteq L(\boldsymbol{\mathcal{G}}_{L_j})$

  5. $(\forall s \in L(\boldsymbol{\mathcal{G}}_{L_j}))(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})$
     $s\rho\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ s\rho l\alpha \in L(\boldsymbol{\mathcal{G}}_{L_j})$

  6. $(\forall s \in L(\boldsymbol{\mathcal{G}}_{L_j}))\ \ s \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ sl \in L_m(\boldsymbol{\mathcal{G}}_{L_j})$

$\diamondsuit$

Proposition 3.2. *Definition 3.5 is equivalent to Definition 3.4.*

proof:

For the n^{th} degree interface system that respects to the alphabet partition given by 3.1 and is composed of plant components $\mathbf{G}^p_H, \mathbf{G}^p_{L_1}, \ldots, \mathbf{G}^p_{L_n}$, supervisors $\mathbf{S}_H, \mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, and interfaces $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$,

[2]This definition is only for proving our synthesis algorithm. For understanding HISC, please see the interface consistent definition in [23] by Leduc.

1. assume for all $j \in \{1, \ldots, n\}$, the system satisfies all the conditions in Definition 3.4. Must show this implies the system satisfies all the conditions in Definition 3.5 as well.

   Point 1, 2, 3, 4 and 6 in both definitions are exactly same, so the system satisfies Point 1, 2, 3, 4 and 6 in Definition 3.5.

   We now show that the system also satisfies Point 5 in Definition 3.5.

   Let $j \in \{1, \ldots, n\}, s \in L(\boldsymbol{\mathcal{G}}_{L_j}), \rho \in \Sigma_{R_j}, \alpha \in \Sigma_{A_j}$. (1)

   Assume $s\rho\alpha \in L(\mathbf{G}^l_{I_j})$. (2)

   Must show implies $(\exists l \in \Sigma^*_{L_j})\ s\rho l\alpha \in L(\boldsymbol{\mathcal{G}}_{L_j})$.

   By (1), we know $s \in L(\boldsymbol{\mathcal{G}}_{L_j}), \rho \in \Sigma_{R_j}$

   $\Rightarrow s\rho \in L(\boldsymbol{\mathcal{G}}_{L_j}),$ by Point 4 in Definition 3.4

   $\Rightarrow s\rho \in \Sigma^*_{IL_j}.\Sigma_{R_j} \cap L(\boldsymbol{\mathcal{G}}_{L_j})$

   $\Rightarrow \mathrm{Elig}_{L(\boldsymbol{\mathcal{G}}_{L_j})}(s\rho\Sigma^*_{L_j}) \cap \Sigma_{A_j} = \mathrm{Elig}_{L(\mathbf{G}^l_{I_j})}(s\rho) \cap \Sigma_{A_j},$ (3)
   by Point 5 in Definition 3.4

   By (2), we have $s\rho\alpha \in L(\mathbf{G}^l_{I_j})$

   $\Rightarrow \alpha \in \mathrm{Elig}_{L(\mathbf{G}^l_{I_j})}(s\rho) \cap \Sigma_{A_j}.$

   $\Rightarrow \alpha \in \mathrm{Elig}_{L(\boldsymbol{\mathcal{G}}_{L_j})}(s\rho\Sigma^*_{L_j}) \cap \Sigma_{A_j},$ by (3)

   $\Rightarrow (\exists l \in \Sigma^*_{L_j})\ s\rho l\alpha \in L(\boldsymbol{\mathcal{G}}_{L_j})$

2. assume for all $j \in \{1, \ldots, n\}$, the system satisfies all the conditions in Definition 3.5, must show this implies the system satisfies all the conditions in Definition 3.4 as well.

   Point 1, 2, 3, 4 and 6 in both definitions are exactly same, so the system satisfies Point 1, 2, 3, 4 and 6 in Definition 3.4.

We now show that the system also satisfies Point 5 in Definition 3.4.

Let $j \in \{1, \ldots, n\}$ and $s \in \Sigma^*_{IL_j}.\Sigma_{R_j} \cap L(\boldsymbol{\mathcal{G}}_{L_j})$. (3)

Must show this implies $\text{Elig}_{L(\boldsymbol{\mathcal{G}}_{L_j})}(s\Sigma^*_{L_j}) \cap \Sigma_{A_j} = \text{Elig}_{L(\mathbf{G}^l_{I_j})}(s) \cap \Sigma_{A_j}$.

(a) Show that $\text{Elig}_{L(\boldsymbol{\mathcal{G}}_{L_j})}(s\Sigma^*_{L_j}) \cap \Sigma_{A_j} \subseteq \text{Elig}_{L(\mathbf{G}^l_{I_j})}(s) \cap \Sigma_{A_j}$.

Sufficient to show that

$(\forall \alpha \in \Sigma_{A_j})(\forall l \in \Sigma^*_{L_j})\ \alpha \in \text{Elig}_{L(\boldsymbol{\mathcal{G}}_{L_j})}(sl) \Rightarrow \alpha \in \text{Elig}_{L(\mathbf{G}^l_{I_j})}(s)$.

Let $\alpha \in \Sigma_{A_j}, l \in \Sigma^*_{L_j}$.

Assume $\alpha \in \text{Elig}_{L(\boldsymbol{\mathcal{G}}_{L_j})}(sl)$. (4)

Must show this implies $\alpha \in \text{Elig}_{L(\mathbf{G}^l_{I_j})}(s)$

By $\boldsymbol{\mathcal{G}}_{L_j} = \mathbf{S}_{L_j} \times \mathbf{G}^p_{L_j} \times \mathbf{G}^l_{I_j}$, we have $L(\boldsymbol{\mathcal{G}}_{L_j}) \subseteq L(\mathbf{G}^l_{I_j})$ (5)

By (4) and (5), we have $\alpha \in \text{Elig}_{L(\mathbf{G}^l_{I_j})}(sl)$

As $\mathbf{G}^l_{I_j} := \mathbf{selfloop}(\mathbf{G}_{I_j}, \Sigma_{L_j})$, we have $\text{Elig}_{L(\mathbf{G}^l_{I_j})}(s) = \text{Elig}_{L(\mathbf{G}^l_{I_j})}(sl)$.

So, $\alpha \in \text{Elig}_{L(\mathbf{G}^l_{I_j})}(s)$.

(b) Show that $\text{Elig}_{L(\mathbf{G}^l_{I_j})}(s) \cap \Sigma_{A_j} \subseteq \text{Elig}_{L(\boldsymbol{\mathcal{G}}_{L_j})}(s\Sigma^*_{L_j}) \cap \Sigma_{A_j}$.

Sufficient to show that $(\forall \alpha \in \Sigma_{A_j}) s\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ sl\alpha \in L(\boldsymbol{\mathcal{G}}_{L_j})$.

From (3), we know $s \in \Sigma^*_{IL_j}.\Sigma_{R_j} \cap L(\boldsymbol{\mathcal{G}}_{L_j})$

$\Rightarrow (\exists s' \in L(\boldsymbol{\mathcal{G}}_{L_j}))(\exists \rho \in \Sigma_{R_j})\ s'\rho = s$

$\Rightarrow (\forall \alpha \in \Sigma_{A_j})\ s'\rho\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ s'\rho l\alpha \in L(\boldsymbol{\mathcal{G}}_{L_j})$,

by Point 5 in Definition 3.5

$\Rightarrow (\forall \alpha \in \Sigma_{A_j})\ s\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ sl\alpha \in L(\boldsymbol{\mathcal{G}}_{L_j})$.

□

Note that although Point 5 in Definition 3.5 is stronger than Point 5 in Definition 3.4, the two definitions are equivalent, because Point 5 in Definition 3.5 only requires some extra conditions which are already present in Point 4 of both definitions.

By Proposition 3.1 and Proposition 3.2, we know that Definition 3.3 and Definition 3.5 are equivalent.

3.2.2 Local Conditions for Global Nonblocking

Definition 3.6. The n^{th} degree interface system composed of plant components $\mathbf{G}^p_H, \mathbf{G}^p_{L_1}, \ldots, \mathbf{G}^p_{L_n}$, supervisors $\mathbf{S}_H, \mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, and interfaces $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$ is said to be *level-wise nonblocking* with respect to the alphabet partition given by 3.1, if the following two conditions are satisfied:

1. *Nonblocking at the high-level*: $\overline{L_m(\boldsymbol{\mathcal{G}}_H)} = L(\boldsymbol{\mathcal{G}}_H)$
2. *Nonblocking at the low-level:* $(\forall j \in \{1, \ldots, n\})\ \overline{L_m(\boldsymbol{\mathcal{G}}_{L_j})} = L(\boldsymbol{\mathcal{G}}_{L_j})$

$\diamondsuit$

Theorem 3.1 (From [24])**.** *If the n^{th} degree interface system composed of plant components $\mathbf{G}^p_H, \mathbf{G}^p_{L_1}, \ldots, \mathbf{G}^p_{L_n}$, supervisors $\mathbf{S}_H, \mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, and interfaces $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$, is level-wise nonblocking and interface consistent with respect to the alphabet partition given by 3.1, then*

$$\overline{L_m(\mathbf{SYSTEM})} = L(\mathbf{SYSTEM})$$

$\square$

3.2.3 Local Conditions for Global Controllability

Definition 3.7. The n^{th} degree interface system composed of plant components $\mathbf{G}^p_H, \mathbf{G}^p_{L_1}, \ldots, \mathbf{G}^p_{L_n}$, supervisors $\mathbf{S}_H, \mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, and interfaces $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$ is said

to be *level-wise controllable* with respect to the alphabet partition given by 3.1, if for all $j \in \{1, \ldots, n\}$ the following two conditions are satisfied:

1. The alphabet of $\mathbf{G}_H^p$ and $\mathbf{S}_H$ is Σ_{IH}, the alphabet of $\mathbf{G}_{L_j}^p$ and $\mathbf{S}_{L_j}$ is Σ_{IL_j}, and the alphabet of $\mathbf{G}_{I_j}$ is Σ_{I_j}

2. *Controllable at the high-level*: $L(\boldsymbol{\mathcal{G}}_H)\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \subseteq L(\boldsymbol{\mathcal{G}}_H)$

3. *Controllable at the low-level*: $L(\boldsymbol{\mathcal{G}}_{L_j})\Sigma_{lu_j} \cap L(\mathbf{G}_{L_j}^p) \subseteq L(\boldsymbol{\mathcal{G}}_{L_j})$

$\diamondsuit$

Theorem 3.2. *If the n^{th} degree interface system composed of plant components $\mathbf{G}_H^p, \mathbf{G}_{L_1}^p, \ldots, \mathbf{G}_{L_n}^p$, supervisors $\mathbf{S}_H, \mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, and interfaces $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$, is level-wise controllable with respect to the alphabet partition given by 3.1, then*

$$L(\mathbf{SYSTEM})\Sigma_u \cap L(\mathbf{PLANT}) \subseteq L(\mathbf{SYSTEM})$$

proof:

Follows immediately from Theorem 2 of [24] and Proposition 2.9.

$\square$

3.3 Level-wise Interface Controllable Supervisor

From Section 3.2, we can see that all the conditions are either based on the high-level or a single low-level. Therefore, we can verify the global controllability and global nonblocking by only verifying the local conditions. For convenience, we now group the conditions by levels and give several corresponding definitions.

Definition 3.8. For the n^{th} degree interface system that respects the alphabet partition given by 3.1 and is composed of plant components $\mathbf{G}_H^p, \mathbf{G}_{L_1}^p, \ldots, \mathbf{G}_{L_n}^p$, supervisors $\mathbf{S}_H, \mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, and interfaces $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$, define the following:

- $\mathbf{S}_H$ is *high-level interface controllable* (HIC), if the following conditions are satisfied:

  1. $L(\boldsymbol{\mathcal{G}}_H)\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \subseteq L(\boldsymbol{\mathcal{G}}_H)$
  2. $(\forall j \in \{1, \ldots, n\})\ L(\boldsymbol{\mathcal{G}}_H)\Sigma_{A_j} \cap L(\mathbf{G}_{I_j}^h) \subseteq L(\boldsymbol{\mathcal{G}}_H)$

- $\mathbf{S}_H$ is a *high-level proper supervisor*, if the following conditions are satisfied:

  1. $\mathbf{S}_H$ is high-level interface controllable.
  2. $\overline{L_m(\boldsymbol{\mathcal{G}}_H)} = L(\boldsymbol{\mathcal{G}}_H)$

- For all $j \in \{1, \ldots, n\}$, $\mathbf{S}_{L_j}$ is j^{th} *low-level interface controllable* (LIC_j), if the following conditions are satisfied:

  1. $L(\boldsymbol{\mathcal{G}}_{L_j})\Sigma_{lu_j} \cap L(\mathbf{G}_{L_j}^p) \subseteq L(\boldsymbol{\mathcal{G}}_{L_j})$
  2. $L(\boldsymbol{\mathcal{G}}_{L_j})\Sigma_{R_j} \cap L(\mathbf{G}_{I_j}^l) \subseteq L(\boldsymbol{\mathcal{G}}_{L_j})$
  3. $(\forall s \in L(\boldsymbol{\mathcal{G}}_{L_j}))(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})$
     $s\rho\alpha \in L(\mathbf{G}_{I_j}^l) \Rightarrow (\exists l \in \Sigma_{L_j}^*)\ s\rho l\alpha \in L(\boldsymbol{\mathcal{G}}_{L_j})$
  4. $(\forall s \in L(\boldsymbol{\mathcal{G}}_{L_j}))\ \ s \in L_m(\mathbf{G}_{I_j}^l) \Rightarrow (\exists l \in \Sigma_{L_j}^*) sl \in L_m(\boldsymbol{\mathcal{G}}_{L_j})$

- For all $j \in \{1, \ldots, n\}$, $\mathbf{S}_{L_j}$ is a j^{th} *low-level proper supervisor*, if the following conditions are satisfied:

  1. $\mathbf{S}_{L_j}$ is j^{th} low-level interface controllable.
  2. $\overline{L_m(\boldsymbol{\mathcal{G}}_{L_j})} = L(\boldsymbol{\mathcal{G}}_{L_j})$

$\diamondsuit$

Proposition 3.3. *For the n^{th} degree interface system that respects the alphabet partition given by 3.1 and is composed of plant components $\mathbf{G}_H^p, \mathbf{G}_{L_1}^p, \ldots, \mathbf{G}_{L_n}^p$, supervisors*

$\mathbf{S}_H, \mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, *and interfaces* $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$, *the flat system is nonblocking and the flat supervisor is controllable for the flat plant if for all* $j \in \{1, \ldots, n\}$, *the following conditions are satisfied:*

1. *The alphabet of* $\mathbf{G}_H^p$ *and* $\mathbf{S}_H$ *is* Σ_{IH}.
2. *The alphabet of* $\mathbf{G}_{L_j}^p$ *and* $\mathbf{S}_{L_j}$ *is* Σ_{IL_j}.
3. $\mathbf{G}_{I_j}$ *is a command-pair interface.*
4. $\mathbf{S}_H$ *is a high-level proper supervisor.*
5. $\mathbf{S}_{L_j}$ *is a* j^{th} *low-level proper supervisor.*

proof:

If the system satisfies all the above five conditions, clearly, it is interface consistent, level-wise nonblocking and level-wise controllable, so the flat system is is nonblocking and the flat supervisor is controllable for the flat plant by Theorem 3.1 and 3.2.

□

Chapter 4

Synthesis of HISC

In [21–27], the supervisors in an HISC system are designed by hand, and then the system is verified for the interface consistent, level-wise controllable and nonblocking conditions. However, for a complicated system it is very desirable to synthesize the supervisors from some specifications which specify the desired system behavior but the system containing them does not necessarily satisfy all the required conditions. In this chapter, we first discuss how to compute the supremal high-level and low-level interface controllable sublanguages, and then give the predicate-based algorithms. These algorithms can easily be implemented by using Binary Decision Diagrams(BDD) as we will discuss in Chapter 6.

4.1 Introduction

In the previous chapter, we specified that a n^{th} degree interface system is composed of plants, supervisors and interfaces. By supervisors, we expect that the system containing them not only have the desired behavior but also is interface consistent, level-wise controllable and level-wise nonblocking. As we discuss the synthesis process

in this chapter, we will assume that a n^{th} degree parallel interface system is composed of plants, specifications and interfaces. By specifications, we mean that they specify the desired behavior but the system containing them is likely not interface consistent, level-wise controllable and level-wise nonblocking.

For the n^{th} degree interface system that respects the alphabet partition given by 3.1 and is composed of plant components $\mathbf{G}_H^p$, $\mathbf{G}_{L_1}^p$, ..., $\mathbf{G}_{L_n}^p$, supervisors $\mathbf{S}_H$, $\mathbf{S}_{L_1}$, ..., $\mathbf{S}_{L_n}$, and interfaces $\mathbf{G}_{I_1}$, ..., $\mathbf{G}_{I_n,}$, if we replace the high-level supervisor $\mathbf{S}_H$ by a high-level specification DES $\mathbf{E}_H$ (defined over Σ_{IH}), and for all $j \in \{1, \ldots, n\}$, we replace the j^{th} low-level supervisor $\mathbf{S}_{L_j}$ by a j^{th} low-level specification DES $\mathbf{E}_{L_j}$ (defined over Σ_{IL_j}), then the resulting system is called a n^{th} *degree specification interface system.* We refer to the original system with supervisors as a n^{th} *degree supervisor interface system.* In case we do not care whether a system is composed of specifications or supervisors, we refer to it as a n^{th} *degree interface system.*

We can apply all the concepts defined for a n^{th} degree supervisor interface system to a n^{th} degree specification interface system by replacing each supervisor DES by its corresponding specification DES. To make things clear, we give the following corresponding definitions for a n^{th} degree specification system.

- The high-level subsystem: $\mathbf{G}_H := \mathbf{E}_H \times \mathbf{G}_H^p$.
- The j^{th} low-level subsystem: $\mathbf{G}_{L_j} := \mathbf{E}_{L_j} \times \mathbf{G}_{L_j}^p, j \in \{1, \ldots, n\}$
- The high-level: $\boldsymbol{\mathcal{G}}_H := \mathbf{E}_H \times \mathbf{G}_H^p \times \mathbf{G}_I^h$.
- The j^{th} low-level: $\boldsymbol{\mathcal{G}}_{L_j} := \mathbf{E}_{L_j} \times \mathbf{G}_{L_j}^p \times \mathbf{G}_{I_j}^l, j \in \{1, \ldots, n\}$

We now give the starting point for the synthesis process.

Definition 4.1. The n^{th} degree parallel interface specification system composed of plant components $\mathbf{G}_H^p, \mathbf{G}_{L_1}^p, \ldots, \mathbf{G}_{L_n}^p$, specifications $\mathbf{E}_H$, $\mathbf{E}_{L_1}, \ldots, \mathbf{E}_{L_n}$, and interfaces

$\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n,}$ is *HISC-valid* with respects to the alphabet partition given by 3.1, if for all $j \in \{1, \ldots, n\}$, the following conditions are satisfied:

1. The alphabet of $\mathbf{G}_H^p$ and $\mathbf{E}_H$ is Σ_{IH}.
2. The alphabet of $\mathbf{G}_{L_j}^p$ and $\mathbf{E}_{L_j}$ is Σ_{IL_j}.
3. $\mathbf{G}_{I_j}$ is a command-pair interface.

$\diamondsuit$

Obviously, the three conditions are exactly same as the first three conditions in Propositions 3.3 except that we use the specification DES instead. Given a n^{th} degree HISC-valid parallel interface specification system, our main objective in this chapter is to develop a method to synthesize a proper high-level supervisor and proper low-level supervisors such that the marked behavior of the constructed n^{th} degree supervisor interface system is as large as possible. That is, we would like to compute the largest marked sublanguages of the high-level and each of the low-levels such that the DES representing each is either a proper high-level supervisor or a proper low-level supervisor, as appropriate. We say such a proper high-level supervisor or low-level supervisor is *locally maximally permissive* for its level.

Let $\mathbf{\Phi}$ be the n^{th} degree HISC-valid specification interface system[1] that respects the alphabet partition given by 3.1 and is composed of plant components $\mathbf{G}_H^p$, $\mathbf{G}_{L_1}^p$, $\ldots$, $\mathbf{G}_{L_n}^p$, specifications $\mathbf{E}_H, \mathbf{E}_{L_1}, \ldots, \mathbf{E}_{L_n}$, and interfaces $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n.}$ We assume that all the DES in system $\mathbf{\Phi}$ have a finite number of states. In this chapter, we always use the system $\mathbf{\Phi}$, and j is always an index with range $\{1, \ldots, n\}$.

For later usage, we give the tuple definitions for the following DES:

$$\boldsymbol{\mathcal{G}}_H := (Q_H, \Sigma_{IH}, \delta_H, q_{H_0}, Q_{H_m}), \;\; \mathbf{G}_H^p := (Y_H, \Sigma_{IH}, \eta_H, y_{H_0}, Y_{H_m}),$$

[1]We use the Greek letter to mean that $\mathbf{\Phi}$ is not a simple DES, but a structured system.

$$\mathbf{E}_H := (Z_H, \Sigma_{IH}, \zeta_H, z_{H_0}, Z_{H_m}),\ \ \mathbf{G}^h_{I_j} := (X^h_j, \Sigma_{IH}, \xi^h_j, x^h_{j_0}, X^h_{j_m}),$$
$$\mathbf{G}^h_I := (X^h, \Sigma_{IH}, \xi^h, x^h_0, X^h_m),$$
$$\boldsymbol{\mathcal{G}}_{L_j} := (Q_{L_j}, \Sigma_{IL_j}, \delta_{L_j}, q_{Lj_0}, Q_{Lj_m}),\quad \mathbf{G}^p_{L_j} := (Y_{L_j}, \Sigma_{IL_j}, \eta_{L_j}, y_{Lj_0}, Y_{Lj_m}),$$
$$\mathbf{E}_{L_j} := (Z_{L_j}, \Sigma_{IL_j}, \zeta_{L_j}, z_{Lj_0}, Z_{Lj_m}),\quad \mathbf{G}^l_{I_j} := (X^l_j, \Sigma_{IL_j}, \xi^l_j, x^l_{j_0}, X^l_{j_m}).$$

So,

$$\boldsymbol{\mathcal{G}}_H = (Z_H \times Y_H \times X^h, \Sigma_{IH}, \zeta_H \times \eta_H \times \xi^h, (z_{H_0}, y_{H_0}, x^h_0), Z_{H_m} \times Y_{H_m} \times X^h_m)$$
$$\boldsymbol{\mathcal{G}}_{L_j} = (Z_{L_j} \times Y_{L_j} \times X^l_j, \Sigma_{IL_j}, \zeta_{L_j} \times \eta_{L_j} \times \xi^l_j, (z_{Lj_0}, y_{Lj_0}, x^l_{j_0}), Z_{Lj_m} \times Y_{Lj_m} \times X^l_{j_m})$$
$$\mathbf{G}^h_I = (X^h_1 \times \cdots \times X^h_n, \Sigma_{IH}, \xi^h_1 \times \cdots \times \xi^h_n, (x^h_{1_0}, \cdots, x^h_{n_0}), X^h_{1_m} \times \cdots \times X^h_{n_m})$$

4.2 High-level Supervisor Synthesis

In this section, we show how to synthesize a locally maximally permissive proper high-level supervisor for system $\mathbf{\Phi}$.

4.2.1 High-level Interface Controllable Language

Definition 4.2. For system $\mathbf{\Phi}$, let $K \subseteq \Sigma^*_{IH}$ be a language. K is *high-level interface controllable*(HIC) *with respect to system* $\mathbf{\Phi}$ if

1. $\overline{K}\Sigma_{hu} \cap L(\mathbf{G}^p_H \times \mathbf{G}^h_I) \subseteq \overline{K}$
2. $(\forall j \in \{1, \ldots, n\})\ \overline{K}\Sigma_{A_j} \cap L(\mathbf{G}^h_{I_j}) \subseteq \overline{K}$

$\diamondsuit$

Obviously, the empty language $\emptyset$ is high-level interface controllable with respect to system $\mathbf{\Phi}$. Note that the definition is based on the language $\overline{K}$, so K is high-level interface controllable with respect to system $\mathbf{\Phi}$ iff $\overline{K}$ is high-level interface controllable with respect to system $\mathbf{\Phi}$.

For an arbitrary language $E \subseteq \Sigma_{IH}^*$, we define the set of all sublanguages of E that are high-level interface controllable with respect to system $\mathbf{\Phi}$ as

$$\mathcal{C}_H(E) := \{K \subseteq E | K \text{ is high-level interface controllable with respect to system } \mathbf{\Phi}\}$$

Clearly, $(\mathcal{C}_H(E), \subseteq)$ is a poset. We now show that the supremum in this poset always exists in $\mathcal{C}_H(E)$.

Proposition 4.1. *For system $\mathbf{\Phi}$, let $E \subseteq \Sigma_{IH}^*$. The set $\mathcal{C}_H(E)$ is nonempty and is closed under arbitrary unions. In particular, $\mathcal{C}_H(E)$ contains a (unique) supremal element,* $\sup \mathcal{C}_H(E) = \cup\{K | K \in \mathcal{C}_H(E)\}$.

proof:

1. Show that $\mathcal{C}_H(E)$ is nonempty

   The empty language $\emptyset$ is high-level interface controllable with respect to system $\mathbf{\Phi}$, so $\emptyset \in \mathcal{C}_H(E)$.

2. Show that $\mathcal{C}_H(E)$ is closed under arbitrary unions.

   Let B be an index set. Assume that $(\forall \beta \in B) K_\beta \in \mathcal{C}_H(E)$. Sufficient to show that $\cup_{\beta \in B} K_\beta \in \mathcal{C}_H(E)$.

   Let $K := \cup_{\beta \in B} K_\beta$. (1)

   Sufficient to show that K is high-level interface controllable with respect to system $\mathbf{\Phi}$. Then by Definition 4.2, we have to show the following:

   (a) Show that $\overline{K}\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \subseteq \overline{K}$

   Let $s \in \overline{K}\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h)$. (2)

   Must show this implies $s \in \overline{K}$.

   From (2), we know $(\exists s' \in \overline{K})(\exists \sigma \in \Sigma_{hu}) s'\sigma = s$ (3)

As $s' \in \overline{K}$, we can conclude $(\exists u \in \Sigma_{IH}^*)\ s'u \in K$

$\Rightarrow (\exists \beta \in B) s'u \in K_\beta$, by (1)

$\Rightarrow s' \in \overline{K_\beta}$, by (1)

$\Rightarrow s'\sigma \in \overline{K_\beta}\Sigma_{hu}$

$\Rightarrow s'\sigma \in \overline{K_\beta}\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h)$, by (3) and (2)

$\Rightarrow s'\sigma \in \overline{K_\beta}$, as $K_\beta \in \mathcal{C}_H(E)$

$\Rightarrow (\exists u' \in \Sigma_{IH}^*) s'\sigma u' \in K_\beta$

$\Rightarrow s'\sigma u' \in K$, by (1)

$\Rightarrow su' \in K$, by (3)

$\Rightarrow s \in \overline{K}$

(b) Show that $(\forall j \in \{1, \ldots, n\})\ \overline{K}\Sigma_{A_j} \cap L(\mathbf{G}_{I_j}^h) \subseteq \overline{K}$

Let $j \in \{1, \ldots, n\}$. Let $s \in \overline{K}\Sigma_{A_j} \cap L(\mathbf{G}_{I_j}^h)$. Must show this implies $s \in \overline{K}$.

The rest is identical to Part 2(a) (from the line labeled with (3)) after substituting Σ_{hu} with Σ_{A_j}, Σ_{IH} with Σ_{IL_j} and $L(\mathbf{G}_H^p \times \mathbf{G}_I^h)$ with $L(\mathbf{G}_{I_j}^h)$.

3. Show that $\sup \mathcal{C}_H(E) = \cup\{K | K \in \mathcal{C}_H(E)\}$.

Let $K_{sup} := \cup\{K | K \in \mathcal{C}_H(E)\}$ (4)

Clearly, for all $K \in \mathcal{C}_H(E)$, we have $K \subseteq K_{sup}$

All that remains is to show:

$(\forall K' \in Pwr(\Sigma_{IH}^*))\ (((\forall K \in \mathcal{C}_H(E))\ K \subseteq K') \Rightarrow K_{sup} \subseteq K')$

Let $K' \in Pwr(\Sigma_{IH}^*)$

Assume $(\forall K \in \mathcal{C}_H(E))\ K \subseteq K'$ (5)

Must show this implies $K_{sup} \subseteq K'$

Let $s \in K_{sup}$ (6)

Must show implies $s \in K'$

By (4) and (6), we know $(\exists K \in \mathcal{C}_H(E))s \in K$

$\Rightarrow s \in K'$, by (5)

4. Show that $\sup \mathcal{C}_H(E) \in \mathcal{C}_H(E)$.

   This immediately follows from Part 3: $\sup \mathcal{C}_H(E) = \cup\{K | K \in \mathcal{C}_H(E)\}$.

□

From Proposition 4.1, we have $\sup \mathcal{C}_H(E) \subseteq E$, which means that we can compute $\sup \mathcal{C}_H(E)$ by removing strings from E.

The following lemma will be used later.

Lemma 4.1. *For system* $\mathbf{\Phi}$, *let* $L \subseteq \Sigma_{IH}^*$. *If* L *is closed then* $\sup \mathcal{C}_H(L)$ *is also closed.*

proof:

Assume $L = \overline{L}$. Must show this implies $\sup \mathcal{C}_H(L) = \overline{\sup \mathcal{C}_H(L)}$.

$\sup \mathcal{C}_H(L) \subseteq \overline{\sup \mathcal{C}_H(L)}$ is automatic. We now show $\overline{\sup \mathcal{C}_H(L)} \subseteq \sup \mathcal{C}_H(L)$.

Sufficient to show that $\overline{\sup \mathcal{C}_H(L)} \subseteq L$ and $\overline{\sup \mathcal{C}_H(L)}$ is high-level interface controllable with respect to system $\mathbf{\Phi}$.

$\overline{\sup \mathcal{C}_H(L)}$ is high-level interface controllable with respect to system $\mathbf{\Phi}$ automatically by the definition of high-level interface controllable language.

By Proposition 4.1, we know that $\sup \mathcal{C}_H(L) \subseteq L$.

$\Rightarrow \overline{\sup \mathcal{C}_H(L)} \subseteq \overline{L}$.

$\Rightarrow \overline{\sup \mathcal{C}_H(L)} \subseteq L$, as L is closed.

□

4.2.2 $\sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H))$ and the Greatest Fixpoint of Ω_H

For system $\mathbf{\Phi}$, if we compute the supremal high-level interface controllable sublanguage of $L_m(\boldsymbol{\mathcal{G}}_H)$, then a DES representing this sublanguage is a maximally permissive proper high-level supervisor. Therefore, we have to find a method to compute $\sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H))$. To this objective, we define the following function.

Definition 4.3. For system $\mathbf{\Phi}$, define the function $\Omega_H : Pwr(\Sigma_{IH}^*) \to Pwr(\Sigma_{IH}^*)$ according to

$$(\forall K \in Pwr(\Sigma_{IH}^*))\ \Omega_H(K) = \Omega_{HNB}(\text{HIC}(K))$$

where Ω_{HNB} and HIC are functions for system $\mathbf{\Phi}$, which are defined as following:

- $\Omega_{HNB} : Pwr(\Sigma_{IH}^*) \to Pwr(\Sigma_{IH}^*)$

$$(\forall K \in Pwr(\Sigma_{IH}^*))\ \ \Omega_{HNB}(K) = L_m(\boldsymbol{\mathcal{G}}_H) \cap K$$

- $\text{HIC} : Pwr(\Sigma_{IH}^*) \to Pwr(\Sigma_{IH}^*)$

$$(\forall K \in Pwr(\Sigma_{IH}^*))\ \ \text{HIC}(K) = \sup \mathcal{C}_H(\overline{K}).$$

$\diamondsuit$

Since the supremal element of $\sup \mathcal{C}_H(\overline{K})$ is unique and must exist by Proposition 4.1, the function HIC is well-defined. The function Ω_{HNB} and HIC both are defined on $Pwr(\Sigma_{IH}^*)$, so we can composite them and the function Ω_H is well-defined.

Clearly, function Ω_{HNB} is monotone with respect to $\subseteq$, and HIC is also monotone with respect to $\subseteq$ by Proposition 4.1, so the function Ω_H is monotone with respect to $\subseteq$ as well.

By the Knaster-Tarski Theorem(Theorem 2.1), the greatest fixpoint of the function Ω_H with respect to $(Pwr(\Sigma_{IH}^*), \subseteq)$ must exist.

Proposition 4.2. *For system* $\mathbf{\Phi}$, $\sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H))$ *is the greatest fixpoint of the function* Ω_H *with respect to* $(Pwr(\Sigma_{IH}^*), \subseteq)$.

proof:

By the definition of Ω_H, we can rewrite it as

$$(\forall K \in Pwr(\Sigma_{IH}^*))\ \Omega_H(K) = L_m(\boldsymbol{\mathcal{G}}_H) \cap \sup \mathcal{C}_H(\overline{K})$$

Let $S = \sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H))$.

1. Show that S is a fixpoint of Ω_H, i.e. $S = \Omega_H(S)$.

   Sufficient to show that $S \subseteq \Omega_H(S)$ and $\Omega_H(S) \subseteq S$.

   $\Omega_H(S) = L_m(\boldsymbol{\mathcal{G}}_H) \cap \sup \mathcal{C}_H(\overline{S})$

   As S is high-level interface controllable with respect to system $\mathbf{\Phi}$, $\overline{S}$ is high-level interface controllable as well. Then,

   $\Omega_H(S) = L_m(\boldsymbol{\mathcal{G}}_H) \cap \overline{S}$ (1)

   By Proposition 4.1, $S \subseteq L_m(\boldsymbol{\mathcal{G}}_H)$. Also $S \subseteq \overline{S}$, so we have $S \subseteq \Omega_H(S)$. (2)

   Now we show that $\Omega_H(S) \subseteq S$.

   Let $t \in \Omega_H(S)$. (3)

   Must show this implies $t \in S$.

   By (3), we have $t \in \Omega_H(S)$

   $\Rightarrow t \in L_m(\boldsymbol{\mathcal{G}}_H)$ and $t \in \overline{S}$, by (1)

   $\Rightarrow \{t\} \subseteq L_m(\boldsymbol{\mathcal{G}}_H)$ and $\{t\} \subseteq \overline{S}$

   $\Rightarrow \{t\} \subseteq L_m(\boldsymbol{\mathcal{G}}_H)$ and $\overline{\{t\}} \subseteq \overline{S}$ (4)

   Let $S' = S \cup \{t\}$.

$\Rightarrow S' \subseteq L_m(\boldsymbol{\mathcal{G}}_H)$, (5)
by (4) and $S \subseteq L_m(\boldsymbol{\mathcal{G}}_H)$

We now show that S' is high-level interface controllable with respect to system $\boldsymbol{\Phi}$.

It is enough to show that

$$\overline{S'}\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \subseteq \overline{S'} \text{ and } (\forall j \in \{1, \ldots, n\})\ \overline{S'}\Sigma_{A_j} \cap L(\mathbf{G}_{I_j}^h) \subseteq \overline{S'}.$$

- Show that $\overline{S'}\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \subseteq \overline{S'}$

  S is high-level interface controllable, so $\overline{S}\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \subseteq \overline{S}$. (6)

  By (4) and (6), we have $\overline{\{t\}}\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \subseteq \overline{S}$ (7)

  From (6) and (7), we have $(\overline{S} \cup \overline{\{t\}})\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \subseteq \overline{S}$

  $\Rightarrow (\overline{S \cup \{t\}})\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \subseteq \overline{S}$, by Proposition 2.2

  $\Rightarrow \overline{S'}\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \subseteq \overline{S}$

  $\Rightarrow \overline{S'}\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \subseteq \overline{S'}$, (8)

  as $S \subseteq S'$ and thus $\overline{S} \subseteq \overline{S'}$

- Show that $(\forall j \in \{1, \ldots, n\})\ \overline{S'}\Sigma_{A_j} \cap L(\mathbf{G}_{I_j}^h) \subseteq \overline{S'}$.

  Let $j \in \{1, \ldots, n\}$. We show $\overline{S'}\Sigma_{A_j} \cap L(\mathbf{G}_{I_j}^h) \subseteq \overline{S'}$.

  S is high-level interface controllable, so $\overline{S}\Sigma_{A_j} \cap L(\mathbf{G}_{I_j}^h) \subseteq \overline{S}$. (9)

  By (4) and (9), we have $\overline{\{t\}}\Sigma_{A_j} \cap L(\mathbf{G}_{I_j}^h) \subseteq \overline{S}$ (10)

  From (9) and (10), we have $(\overline{S} \cup \overline{\{t\}})\Sigma_{A_j} \cap L(\mathbf{G}_{I_j}^h) \subseteq \overline{S}$

  $\Rightarrow (\overline{S \cup \{t\}})\Sigma_{A_j} \cap L(\mathbf{G}_{I_j}^h) \subseteq \overline{S}$, by Proposition 2.2

  $\Rightarrow \overline{S'}\Sigma_{A_j} \cap L(\mathbf{G}_{I_j}^h) \subseteq \overline{S}$

  $\Rightarrow \overline{S'}\Sigma_{A_j} \cap L(\mathbf{G}_{I_j}^h) \subseteq \overline{S'}$, (11)

  as $S \subseteq S'$ and thus $\overline{S} \subseteq \overline{S'}$

By (5), (8), (11), we have $S' \in \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H))$. As S is the supremal high-level interface controllable sublanguage of $L_m(\boldsymbol{\mathcal{G}}_H)$, we get $S' \subseteq S$.

$\Rightarrow \{t\} \subseteq S$

$\Rightarrow t \in S$

We thus have $\Omega_H(S) \subseteq S$. Combining it with (2) gives $\Omega_H(S) = S$.

2. Show that S is the greatest fixpoint of Ω_H. i.e. $(\forall T \in Pwr(\Sigma_{IH}^*))T = \Omega_H(T) \Rightarrow T \subseteq S$.

   Let $T \in Pwr(\Sigma_{IH}^*)$. Assume $T = \Omega_H(T)$. Must show this implies $T \subseteq S$.

   Sufficient to show that T is high-level controllable with respect to system $\boldsymbol{\Phi}$ and $T \subseteq L_m(\boldsymbol{\mathcal{G}}_H)$.

$$\begin{aligned} T &= \Omega_H(T) \\ &= L_m(\boldsymbol{\mathcal{G}}_H) \cap \sup \mathcal{C}_H(\overline{T}) \qquad (12) \\ &\subseteq \sup \mathcal{C}_H(\overline{T}) \end{aligned}$$

   By Lemma 4.1, $\sup \mathcal{C}_H(\overline{T})$ is closed, so combining with (12), we have

$$\overline{T} \subseteq \sup \mathcal{C}_H(\overline{T})$$

   By Proposition 4.1, we know that $\sup \mathcal{C}_H(\overline{T}) \subseteq \overline{T}$. Therefore, we have

$$\overline{T} = \sup \mathcal{C}_H(\overline{T})$$

   So, $\overline{T}$ is high-level controllable with respect to system $\boldsymbol{\Phi}$, which means that T is high-level controllable with respect to system $\boldsymbol{\Phi}$.

   From (12), clearly, $T \subseteq L_m(\boldsymbol{\mathcal{G}}_H)$.

□

4.2.3 Computing the greatest fixpoint of Ω_H

From the above section, we know that we can compute the $\sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H))$ by computing the greatest fixpoint of Ω_H. It is tempting to compute the greatest fixpoint by iteration of Ω_H.

Proposition 4.3. *For system* $\boldsymbol{\Phi}$*, the set theoretic limit* $\lim_{i\to\infty} \Omega_H^i(L(\boldsymbol{\mathcal{G}}_H)), i \in \{1, 2, ...\}$ *exists, and* $\sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H)) \subseteq \lim_{i\to\infty} \Omega_H^i(L(\boldsymbol{\mathcal{G}}_H))$.

proof:

Let $S := \sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H))$.

1. Show that $\lim_{i\to\infty} \Omega_H^i(L(\boldsymbol{\mathcal{G}}_H))$ exists.

   From the definition of Ω_H, we know $\Omega_H(L(\boldsymbol{\mathcal{G}}_H)) = L_m(\boldsymbol{\mathcal{G}}_H) \cap \sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H))$, so by $L_m(\boldsymbol{\mathcal{G}}_H) \subseteq L(\boldsymbol{\mathcal{G}}_H)$ and $\sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H)) \subseteq L(\boldsymbol{\mathcal{G}}_H)$ by Proposition 4.1, we have

   $\Omega_H(L(\boldsymbol{\mathcal{G}}_H)) \subseteq L(\boldsymbol{\mathcal{G}}_H)$

   $\Rightarrow \Omega_H^2(L(\boldsymbol{\mathcal{G}}_H)) \subseteq \Omega_H(L(\boldsymbol{\mathcal{G}}_H)),$ as Ω_H is monotone.

   $\Rightarrow \Omega_H^3(L(\boldsymbol{\mathcal{G}}_H)) \subseteq \Omega_H^2(L(\boldsymbol{\mathcal{G}}_H)),$ as Ω_H is monotone.

   $\cdots$

   Then, we have

   $$L(\boldsymbol{\mathcal{G}}_H) \supseteq \Omega_H(L(\boldsymbol{\mathcal{G}}_H)) \supseteq \Omega_H^2(L(\boldsymbol{\mathcal{G}}_H)) \supseteq \cdots$$

   Therefore, $\lim_{i\to\infty} \Omega_H^i(L(\boldsymbol{\mathcal{G}}_H))$ exists and equals to $\bigcap_{i=0}^{\infty} \Omega_H^i(L(\boldsymbol{\mathcal{G}}_H))$.

2. Show that $S \subseteq \lim_{i\to\infty} \Omega_H^i(L(\boldsymbol{\mathcal{G}}_H))$.

   By Proposition 4.1, we have $S \subseteq L(\boldsymbol{\mathcal{G}}_H)$

$\Rightarrow \Omega_H(S) \subseteq \Omega_H(L(\boldsymbol{\mathcal{G}}_H)), \quad$ as Ω_H is monotone.

$\Rightarrow S \subseteq \Omega_H(L(\boldsymbol{\mathcal{G}}_H)), \quad$ as $S = \Omega_H(S)$ by Proposition 4.2.

$\Rightarrow \Omega_H(S) \subseteq \Omega_H^2(L(\boldsymbol{\mathcal{G}}_H)), \quad$ as Ω_H is monotone.

$\Rightarrow S \subseteq \Omega_H^2(L(\boldsymbol{\mathcal{G}}_H)), \quad$ as $S = \Omega_H(S)$ by Proposition 4.2.

$\ldots$

$\Rightarrow S \subseteq \lim_{i\to\infty} \Omega_H^i(L(\boldsymbol{\mathcal{G}}_H))$.

□

Note that we start the iteration from the closed language $L(\boldsymbol{\mathcal{G}}_H)$. It remains to show the reverse $\lim_{i\to\infty} \Omega_H^i(L(\boldsymbol{\mathcal{G}}_H)) \subseteq \sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H))$. For practical computation, we also have to show that the limit will be reached after a finite number of iterations on Ω_H. In general case, actually $\lim_{i\to\infty} \Omega_H^i(L(\boldsymbol{\mathcal{G}}_H)) \nsubseteq \sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H))$. Please see [48] for details. However, in the case that all the languages are regular, the reverse holds and the number of iterations on Ω_H is finite, as we will show in the following sections.

To iterate the computation on the function Ω_H, we first must know how to compute the function HIC, i.e. to compute the supremal high-level interface controllable sublanguage of a closed language. Here, we only focus on the special class of closed languages. Let $Pred(Q_H)$ be the set of all predicates on Q_H, the state set of $\boldsymbol{\mathcal{G}}_H$. Let $\mathcal{P}_{ha} \in Pred(Q_H)$ be a predicate[2], we now show how to compute $\sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$. Note that $L(\boldsymbol{\mathcal{G}}_H) = L(\boldsymbol{\mathcal{G}}_H, true)$, so at the starting point, we let $\mathcal{P}_{ha} = true$ and then compute $\sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$.

We compute $\sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$ by creating another function, and we will prove that $\sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$ is the greatest fixpoint of that function.

[2] $\mathcal{P}_{ha}$ is just an arbitrary predicate. We use this notation as we need to distinguish a given predicate ($\mathcal{P}_{ha}$) with an arbitrary predicate P in the next subsection.

4.2.4 Computing $\sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$

Definition 4.4. For system $\mathbf{\Phi}$, let $\mathcal{P}_{ha} \in Pred(Q_H)$ be a given predicate. Define the function $\Omega_{HIC} : Pwr(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) \to Pwr(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$ according to

$$
\begin{aligned}
&(\forall K \in Pwr(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})))\\
&\quad \Omega_{HIC}(K) := \{s \in K \mid \overline{\{s\}}\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \subseteq \overline{K} \ \ \&\\
&\qquad\qquad ((\forall j \in \{1, \ldots, n\})\ \overline{\{s\}}\Sigma_{A_j} \cap L(\mathbf{G}_{I_j}^h) \subseteq \overline{K})\}
\end{aligned}
$$

$\diamondsuit$

Clearly Ω_{HIC} is monotone. Note that $\Omega_{HIC}(K) \subseteq K \subseteq L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})$, so this function is well-defined, and it only removes strings from K.

To make it easier to understand and use this function, the definition can be rewritten as

$$
\begin{aligned}
&(\forall K \in Pwr(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})))\\
&\quad \Omega_{HIC}(K) := \{s \in K \mid (\forall t \leq s)\\
&\qquad\qquad ((\forall \sigma_u \in \Sigma_{hu})\ t\sigma_u \in L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \Rightarrow t\sigma_u \in \overline{K})\ \&\\
&\qquad\qquad ((\forall \sigma_{a_1} \in \Sigma_{A_1})\ t\sigma_{a_1} \in L(\mathbf{G}_{I_1}^h) \Rightarrow t\sigma_{a_1} \in \overline{K})\ \&\\
&\qquad\qquad\quad \cdots\\
&\qquad\qquad ((\forall \sigma_{a_n} \in \Sigma_{A_n})\ t\sigma_{a_n} \in L(\mathbf{G}_{I_n}^h) \Rightarrow t\sigma_{a_n} \in \overline{K})\}.
\end{aligned}
$$

Proposition 4.4. *For system $\mathbf{\Phi}$, let $\mathcal{P}_{ha} \in Pred(Q_H)$ be the given predicate for function Ω_{HIC}. Then $\sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$ is the greatest fixpoint of Ω_{HIC} with respect to $(Pwr(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})), \subseteq)$.*

proof:

Let $S = \sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$. By Proposition 4.1, $S \subseteq L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})$, S is a valid input for Ω_{HIC}.

1. Show that $S = \Omega_{HIC}(S)$.

   $$\Omega_{HIC}(S) = \{s \in S \mid \overline{\{s\}}\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \subseteq \overline{S} \ \ \& \\ ((\forall j \in \{1, \ldots, n\})\ \overline{\{s\}}\Sigma_{A_j} \cap L(\mathbf{G}_{I_j}^h) \subseteq \overline{S})\}$$

   By the definition of S, clearly, all the strings in S must satisfy the condition, so $\Omega_{HIC}(S) = S$.

2. Show that S is the greatest fixpoint of Ω_{HIC}, i.e.

   $(\forall T \in Pwr(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})))\ T = \Omega_{HIC}(T) \Rightarrow T \subseteq S$.

   Let $T \in Pwr(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$. Assume $T = \Omega_{HIC}(T)$.

   Must show this implies $T \subseteq S$.

   Sufficient to show that T is high-level interface controllable with respect to system $\mathbf{\Phi}$.

   $$\Omega_{HIC}(T) = \{s \in T \mid \overline{\{s\}}\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \subseteq \overline{T} \ \ \& \\ ((\forall j \in \{1, \ldots, n\})\ \overline{\{s\}}\Sigma_{A_j} \cap L(\mathbf{G}_{I_j}^h) \subseteq \overline{T})\}$$

   So,

   $$T = \{s \in T \mid \overline{\{s\}}\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \subseteq \overline{T} \ \ \& \\ ((\forall j \in \{1, \ldots, n\})\ \overline{\{s\}}\Sigma_{A_j} \cap L(\mathbf{G}_{I_j}^h) \subseteq \overline{T})\}$$

   Then, we have

   $$\overline{T}\Sigma_{hu} \cap L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \subseteq \overline{T} \ \ \& \ \ ((\forall j \in \{1, \ldots, n\})\ \overline{T}\Sigma_{A_j} \cap L(\mathbf{G}_{I_j}^h) \subseteq \overline{T}),$$

   which means T is high-level interface controllable with respect to system $\mathbf{\Phi}$.

□

From Proposition 4.4, we know that we can compute the $\sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$ by computing the greatest fixpoint of Ω_{HIC}. It is also tempting to compute the greatest fixpoint by iteration of Ω_{HIC}.

Proposition 4.5. *For system* $\mathbf{\Phi}$*, let* $\mathcal{P}_{ha} \in Pred(Q_H)$ *be the given predicate for the function* Ω_{HIC}*. The set theoretic limit* $\lim_{i\to\infty} \Omega^i_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})), i \in \{1, 2, ...\}$ *exists, and* $\sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) \subseteq \lim_{i\to\infty} \Omega^i_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$.

proof:

Let $S := \sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$.

1. Show that $\lim_{i\to\infty} \Omega^i_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$ exists.

   From the definition of Ω_{HIC}, we have $\Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) \subseteq L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})$.

   $\Rightarrow \Omega^2_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) \subseteq \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})),$ as Ω_{HIC} is monotone.

   $\Rightarrow \Omega^3_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) \subseteq \Omega^2_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})),$ as Ω_{HIC} is monotone.

   $\cdots$

   Then, we have

   $$L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}) \supseteq \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) \supseteq \Omega^2_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) \supseteq \cdots$$

   So, $\lim_{i\to\infty} \Omega^i_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$ exists and equals to $\bigcap_{i=0}^{\infty} \Omega^i_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$.

2. Show that $S \subseteq \lim_{i\to\infty} \Omega^i_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$.

   By Proposition 4.1, we have $S \subseteq L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})$

   $\Rightarrow \Omega_{HIC}(S) \subseteq \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})),$ as Ω_{HIC} is monotone.

   $\Rightarrow S \subseteq \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})),$ as $S = \Omega_{HIC}(S)$ by Proposition 4.4.

   $\Rightarrow \Omega_{HIC}(S) \subseteq \Omega^2_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})),$ as Ω_{HIC} is monotone.

   $\Rightarrow S \subseteq \Omega^2_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})),$ as $S = \Omega_{HIC}(S)$ by Proposition 4.4.

   $\cdots$

   $\Rightarrow S \subseteq \lim_{i\to\infty} \Omega^i_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$.

□

It remains to show that $\lim_{i\to\infty} \Omega^i_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) \subseteq \sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$ and that we only need to iterate a finite number of iterations in the case that $\boldsymbol{\mathcal{G}}_H$ only has finite number of states.

Definition 4.5. For system $\boldsymbol{\Phi}$, define the function $W_H : Pred(Q_H) \to Pwr(\Sigma^*_{IH})$ according to

$(\forall P \in Pred(Q_H))$

$$W_H(P) := \big\{ s \in L(\boldsymbol{\mathcal{G}}_H, P) \mid \big((\exists \sigma_u \in \Sigma_{hu})\ s\sigma_u \in L(\mathbf{G}^p_H \times \mathbf{G}^h_I)\ \&\ s\sigma_u \notin L(\boldsymbol{\mathcal{G}}_H, P) \big)\ or \\ \big((\exists j \in \{1, \ldots, n\}) (\exists \sigma_{a_j} \in \Sigma_{A_j})\ s\sigma_{a_j} \in L(\mathbf{G}^h_{I_j})\ \&\ s\sigma_{a_j} \notin L(\boldsymbol{\mathcal{G}}_H, P) \big) \big\}$$

$\diamondsuit$

Note that $W_H(P) \subseteq L(\boldsymbol{\mathcal{G}}_H, P)$. The following two lemmas will be used in the proof of the next proposition.

Lemma 4.2. *For system* $\boldsymbol{\Phi}$, *the following holds:*

$(\forall P \in Pred(Q_H))(\forall s, t \in L(\boldsymbol{\mathcal{G}}_H, P))$

$$s \in W_H(P)\ \&\ (\delta_H(q_{H_0}, s) = \delta_H(q_{H_0}, t)) \Rightarrow t \in W_H(P),$$

where q_{H_0} *is the initial state of* $\boldsymbol{\mathcal{G}}_H$ *as defined in section 4.1.*

proof:

Let $P \in Pred(Q_H)$, and $s, t \in L(\boldsymbol{\mathcal{G}}_H, P)$. (1)

Assume $\delta_H(q_{H_0}, s) = \delta_H(q_{H_0}, t)$ and $s \in W_H(P)$. (2)

Must show this implies $t \in W_H(P)$.

Let $\equiv_{L(\boldsymbol{\mathcal{G}}_H, P)}$, $\equiv_{L(\mathbf{G}^p_H \times \mathbf{G}^h_I)}$, $\equiv_{L(\mathbf{G}^h_{I_j})}$ $(j \in \{1, \ldots, n\})$ be the Nerode equivalence relation on Σ^*_{IH} with respect to $L(\boldsymbol{\mathcal{G}}_H, P), L(\mathbf{G}^p_H \times \mathbf{G}^h_I)$, and $L(\mathbf{G}^h_{I_j})$ respectively.

From (2), we have $\delta_H(q_{H_0}, s) = \delta_H(q_{H_0}, t)$

By Proposition 2.6, we thus have $s \equiv_{L(\boldsymbol{\mathcal{G}}_H, P)} t$ (3)

As $\boldsymbol{\mathcal{G}}_H = \mathbf{E}_H \times \mathbf{G}_H^p \times \mathbf{G}_{I_1} \times \cdots \times \mathbf{G}_{I_n}$, we can conclude $\eta_H \times \xi^h((y_{H_0}, x_0^h), s) = \eta_H \times \xi^h((y_{H_0}, x_0^h), t)$ and $(\forall j \in \{1, \ldots, n\})\ \xi_j^h(x_{j_0}^h, s) = \xi_j^h(x_{j_0}^h, t)$.

By Proposition 2.6 and the fact $L(\mathbf{G}_H^p \times \mathbf{G}_I^h, true) = L(\mathbf{G}_H^p \times \mathbf{G}_I^h)$ and $L(\mathbf{G}_{I_j}^h, true) = L(\mathbf{G}_{I_j}^h)$, we thus have $s \equiv_{L(\mathbf{G}_H^p \times \mathbf{G}_I^h)} t$ and $(\forall j \in \{1, \ldots, n\})\ s \equiv_{L(\mathbf{G}_{I_j}^h)} t$ (4)

From (1), we have $s \in W_H(P)$

$\Rightarrow \big((\exists \sigma_u \in \Sigma_{hu}) s\sigma_u \in L(\mathbf{G}_H^p \times \mathbf{G}_I^h)\ \&\ s\sigma_u \notin L(\boldsymbol{\mathcal{G}}_H, P)\big)$ *or*

$\quad \big((\exists j \in \{1, \ldots, n\})(\exists \sigma_{a_j} \in \Sigma_{A_j})\ s\sigma_{a_j} \in L(\mathbf{G}_{I_j}^h)\ \&\ s\sigma_{a_j} \notin L(\boldsymbol{\mathcal{G}}_H, P)\big)$

$\Rightarrow \big((\exists \sigma_u \in \Sigma_{hu})\ t\sigma_u \in L(\mathbf{G}_H^p \times \mathbf{G}_I^h)\ \&\ t\sigma_u \notin L(\boldsymbol{\mathcal{G}}_H, P)\big)$ *or*

$\quad \big((\exists j \in \{1, \ldots, n\})(\exists \sigma_{a_j} \in \Sigma_{A_j})\ t\sigma_{a_j} \in L(\mathbf{G}_{I_j}^h)\ \&\ t\sigma_{a_j} \notin L(\boldsymbol{\mathcal{G}}_H, P)\big)$, by (3)(4)

$\Rightarrow t \in W_H(P)$

□

Lemma 4.3. *For system* $\boldsymbol{\Phi}$, *let* $\mathcal{P}_{ha} \in Pred(Q_H)$ *be the given predicate for* Ω_{HIC}. *Then the following holds:*

$(\forall P \in Sub(\mathcal{P}_{ha}))(\forall s \in L(\boldsymbol{\mathcal{G}}_H, P))$

$$((\exists t \leq s)\ t \in W_H(P)) \Leftrightarrow s \notin \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P))$$

proof:

Let $P \in Sub(\mathcal{P}_{ha})$ and $s \in L(\boldsymbol{\mathcal{G}}_H, P)$. (1)

1. Show that $((\exists t \leq s)\ t \in W_H(P)) \Rightarrow s \notin \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P))$

   Assume $(\exists t \leq s)\ t \in W_H(P)$. (2)

   Must show this implies $s \notin \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P))$.

   From (2), we have

   $(\exists t \leq s)\big((\exists \sigma_u \in \Sigma_{hu})\ t\sigma_u \in L(\mathbf{G}_H^p \times \mathbf{G}_I^h)\ \&\ t\sigma_u \notin L(\boldsymbol{\mathcal{G}}_H, P)\big)$ *or*

   $\big((\exists j \in \{1, \ldots, n\})\ (\exists \sigma_{a_j} \in \Sigma_{A_j})\ t\sigma_{a_j} \in L(\mathbf{G}_{I_j}^h)\ \&\ t\sigma_{a_j} \notin L(\boldsymbol{\mathcal{G}}_H, P)\big)$

$$\Rightarrow \neg\big((\forall t \leq s)\ ((\forall \sigma_u \in \Sigma_{hu})\ t\sigma_u \in L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \Rightarrow t\sigma_u \in L(\boldsymbol{\mathcal{G}}_H, P))\ \&$$
$$((\forall \sigma_{a_1} \in \Sigma_{A_1})\ t\sigma_{a_1} \in L(\mathbf{G}_{I_1}^h) \Rightarrow t\sigma_{a_1} \in L(\boldsymbol{\mathcal{G}}_H, P))\ \&$$
$$\cdots$$
$$((\forall \sigma_{a_n} \in \Sigma_{A_n})\ t\sigma_{a_n} \in L(\mathbf{G}_{I_n}^h) \Rightarrow t\sigma_{a_n} \in L(\boldsymbol{\mathcal{G}}_H, P))\big)$$
$$\Rightarrow s \notin \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P)), \quad \text{by definition of } \Omega_{HIC}$$

2. Show that $s \notin \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P)) \Rightarrow (\exists t \leq s)\ t \in W_H(P)$.

   Assume $s \notin \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P))$. (3)

   Must show this implies $(\exists t \leq s)\ t \in W_H(P)$.

   From (1), we know $s \in L(\boldsymbol{\mathcal{G}}_H, P)$. From (3), we thus have

$$\Rightarrow \neg\big((\forall t \leq s)\ ((\forall \sigma_u \in \Sigma_{hu})\ t\sigma_u \in L(\mathbf{G}_H^p \times \mathbf{G}_I^h) \Rightarrow t\sigma_u \in L(\boldsymbol{\mathcal{G}}_H, P))\ \&$$
$$((\forall \sigma_{a_1} \in \Sigma_{A_1})\ t\sigma_{a_1} \in L(\mathbf{G}_{I_1}^h) \Rightarrow t\sigma_{a_1} \in L(\boldsymbol{\mathcal{G}}_H, P))\ \&$$
$$\cdots$$
$$((\forall \sigma_{a_n} \in \Sigma_{A_n})\ t\sigma_{a_n} \in L(\mathbf{G}_{I_n}^h) \Rightarrow t\sigma_{a_n} \in L(\boldsymbol{\mathcal{G}}_H, P))\big)$$
$$\Rightarrow (\exists t \leq s)\big((\exists \sigma_u \in \Sigma_{hu})\ t\sigma_u \in L(\mathbf{G}_H^p \times \mathbf{G}_I^h)\ \&\ t\sigma_u \notin L(\boldsymbol{\mathcal{G}}_H, P)\big)\ \textit{or}$$
$$\big((\exists j \in \{1, \ldots, n\})\ (\exists \sigma_{a_j} \in \Sigma_{A_j})\ t\sigma_{a_j} \in L(\mathbf{G}_{I_j}^h)\ \&\ t\sigma_{a_j} \notin L(\boldsymbol{\mathcal{G}}_H, P)\big) \quad (4)$$

   As $s \in L(\boldsymbol{\mathcal{G}}_H, P)$ and $L(\boldsymbol{\mathcal{G}}_H, P)$ is closed, we have $t \in L(\boldsymbol{\mathcal{G}}_H, P)$ (5)

   By (4) and (5), we have $(\exists t \leq s)\ t \in W_H(P)$.

□

The above two lemmas state a very important fact. If a string $s \in L(\boldsymbol{\mathcal{G}}_H, P)$ is in $W_H(P)$ with $P \preceq \mathcal{P}_{ha}$, Lemma 4.2 says that all strings in $L(\boldsymbol{\mathcal{G}}_H, P)$ that go from q_{H_0} to the state $\delta_H(q_{H_0}, s)$ are also in $W_H(P)$, while Lemma 4.3 says that all strings in $L(\boldsymbol{\mathcal{G}}_H, P)$ that are extended from a string in $W_H(P)$ are not in $\Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P))$. It means that if a string $s \in L(\boldsymbol{\mathcal{G}}_H, P)$ is in $W_H(P)$, then all the strings in $L(\boldsymbol{\mathcal{G}}_H, P))$

that lead from q_{H_0} to or pass through the state $\delta_H(q_{H_0}, s)$ are not in $\Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P))$. The fact is formally proved as follows by using these two lemmas.

Proposition 4.6. *For system* $\boldsymbol{\Phi}$, *let* $\mathcal{P}_{ha} \in Pred(Q_H)$ *be the given predicate for* Ω_{HIC}, *then*

$$(\forall P \in Sub(\mathcal{P}_{ha}))\ L(\boldsymbol{\mathcal{G}}_H, P) - \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P)) = L^{W_H(P)}(\boldsymbol{\mathcal{G}}_H, P),$$

where $L^{W_H(P)}(\boldsymbol{\mathcal{G}}_H, P) = \cup_{s \in W_H(P)} L^s(\boldsymbol{\mathcal{G}}_H, P)$ *as defined in Section 2.5.3.*

proof:

Let $P \in Sub(\mathcal{P}_{ha})$.

1. Show that $L^{W_H(P)}(\boldsymbol{\mathcal{G}}_H, P) \subseteq L(\boldsymbol{\mathcal{G}}_H, P) - \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P))$.

   Let $s \in L^{W_H(P)}(\boldsymbol{\mathcal{G}}_H, P)$. (1)

   Must show this implies $s \in L(\boldsymbol{\mathcal{G}}_H, P) - \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P))$.

   From (1), we know $(\exists t \in W_H(P))\ s \in L^t(\boldsymbol{\mathcal{G}}_H, P)$

   $\Rightarrow (\exists t \in W_H(P))\ s \in \{u \in L(\boldsymbol{\mathcal{G}}_H, P) | (\exists v \leq u)\ \delta_H(q_{H_0}, v) = \delta_H(q_{H_0}, t)\}$

   $\Rightarrow s \in L(\boldsymbol{\mathcal{G}}_H, P)\ \&\ (\exists v \leq s)(\exists t \in W_H(P))\ \delta_H(q_{H_0}, v) = \delta_H(q_{H_0}, t)$

   $\Rightarrow s \in L(\boldsymbol{\mathcal{G}}_H, P)\ \&\ (\exists v \leq s)\ v \in W_H(P)$,

   by Lemma 4.2 and the fact $L(\boldsymbol{\mathcal{G}}_H, P)$ is closed.

   $\Rightarrow s \in L(\boldsymbol{\mathcal{G}}_H, P)\ \&\ s \notin \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P))$, by Lemma 4.3

   $\Rightarrow s \in L(\boldsymbol{\mathcal{G}}_H, P) - \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P))$

2. Show that $L(\boldsymbol{\mathcal{G}}_H, P) - \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P)) \subseteq L^{W_H(P)}(\boldsymbol{\mathcal{G}}_H, P)$.

   Let $s \in L(\boldsymbol{\mathcal{G}}_H, P) - \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P))$. (2)

   Must show this implies $s \in L^{W_H(P)}(\boldsymbol{\mathcal{G}}_H, P)$.

Sufficient to show that $(\exists t \in W_H(P))\ s \in L^t(\boldsymbol{\mathcal{G}}_H, P)$

From (2), we know $s \in L(\boldsymbol{\mathcal{G}}_H, P)$ & $s \notin \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P))$.

$\Rightarrow s \in L(\boldsymbol{\mathcal{G}}_H, P)$ & $(\exists t \leq s)$ & $t \in W_H(P)$, by Lemma 4.3

$\Rightarrow (\exists t \in W_H(P))\ s \in L^t(\boldsymbol{\mathcal{G}}_H, P)$, as t is a prefix of s and $L(\boldsymbol{\mathcal{G}}_H, P)$ is closed.

□

Now, if we look at Proposition 2.7, it seems that we can compute $\Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P))$ by removing all the states in $\{q \in Q_H | (\exists s \in W_H(P))\delta_H(q_{H_0}, s) = q\}$ from the state set Q_H of $\boldsymbol{\mathcal{G}}_H$.

For any $q \in Q_H$, as $\boldsymbol{\mathcal{G}}_H = \mathbf{E}_H \times \mathbf{G}_H^p \times \mathbf{G}_I^h$, there must exist unique $z \in Z_H, y \in Y_H$ and $x \in X^h$ such that $q = (z, y, x)$. For the state $x \in X^h$, as $\mathbf{G}_I^h = \mathbf{G}_{I_1}^h \times \cdots \times \mathbf{G}_{I_n}^h$, there must also exist unique $x_1 \in X_1^h, \ldots, x_n \in X_n^h$ such that $x = (x_1, \ldots, x_n)$, so

$$q = (z, y, x) = (z, y, x_1, \ldots, x_n) \tag{4.1}$$

Definition 4.6. For system $\boldsymbol{\Phi}$, let $\mathcal{P}_{ha} \in Pred(Q_H)$ be a given predicate. Define the function $\Pi_{HIC} : Sub(\mathcal{P}_{ha}) \to Sub(\mathcal{P}_{ha})$ according to

$(\forall P \in Sub(\mathcal{P}_{ha}))$

$\Pi_{HIC}(P) = P\text{–}pr(\text{Bad}_{RH} \cup \{q \models R(\boldsymbol{\mathcal{G}}_H, P) \mid (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q, \sigma) \models \neg P\})$, where $\text{Bad}_{RH} = \{q \models R(\boldsymbol{\mathcal{G}}_H, P) |$

$((\exists \sigma_u \in \Sigma_{hu})\ (\eta_H \times \xi^h((y, x), \sigma_u)!\ \ \&\ \ \zeta_H(z, \sigma_u)\ \not!))$ *or*

$\big((\exists j \in \{1, \ldots, n\})(\exists \sigma_{a_j} \in \Sigma_{A_j})\ (\xi_j^h(x_j, \sigma_{a_j})!\ \&$

$\zeta_H \times \eta_H \times \xi_1^h \times \cdots \times \xi_{j-1}^h \times \xi_{j+1}^h \times \cdots \times \xi_n^h((z, y, x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n), \sigma_{a_j})\ \not!)\big)\big\}$

and $q = (z, y, x) = (z, y, x_1, \ldots, x_n)$ as in equation (4.1).

♢

Note that $\Pi_{HIC}(P) \preceq P \preceq \mathcal{P}_{ha}$, so the function is well-defined.

Proposition 4.7. *For system* $\mathbf{\Phi}$, *let* $\mathcal{P}_{ha} \in Pred(Q_H)$ *be the given predicate for* Ω_{HIC}, *then*

1. $(\forall P \in Sub(\mathcal{P}_{ha}))\ \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P)) = L(\boldsymbol{\mathcal{G}}_H, \Pi_{HIC}(P))$

2. $(\forall i \in \{0, 1, \ldots\})\ \Omega^i_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) = L(\boldsymbol{\mathcal{G}}_H, \Pi^i_{HIC}(\mathcal{P}_{ha}))$

proof:

1. Let $P \in Sub(\mathcal{P}_{ha})$. Must show that $\Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P)) = L(\boldsymbol{\mathcal{G}}_H, \Pi_{HIC}(P))$.

   We prove this proposition by a series of transformation.

   $\Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P))$

   $= L(\boldsymbol{\mathcal{G}}_H, P) - L^{W_H(P)}(\boldsymbol{\mathcal{G}}_H, P)$, by Proposition 4.6

   $= L(\boldsymbol{\mathcal{G}}_H, P) - \cup_{w \in W_H(P)} L^w(\boldsymbol{\mathcal{G}}_H, P)$

   $= L\big(\boldsymbol{\mathcal{G}}_H, P - pr(\cup_{w \in W_H(P)} \{\delta_H(q_{H_0}, w)\})\big)$, by Proposition 2.7

   $= L\big(\boldsymbol{\mathcal{G}}_H, P - pr(\{q \in R(\boldsymbol{\mathcal{G}}_H, P) | \delta_H(q_{H_0}, w) = q \text{ for some } w \in W_H(P)\})\big)$,

   as $W_H(P) \subseteq L(\boldsymbol{\mathcal{G}}_H, P)$ and by Proposition 2.5

   $= L\Big(\boldsymbol{\mathcal{G}}_H, P - pr\big(\{q \in R(\boldsymbol{\mathcal{G}}_H, P) | (\exists w \in L(\boldsymbol{\mathcal{G}}_H, P))\ \delta_H(q_{H_0}, w) = q\ \&$

   $\big((\exists \sigma_u \in \Sigma_{hu})\ w\sigma_u \in L(\mathbf{G}^p_H \times \mathbf{G}^h_I)\ \&\ w\sigma_u \notin L(\boldsymbol{\mathcal{G}}_H, P)\big)$ *or*

   $((\exists j \in \{1, \ldots, n\})(\exists \sigma_{a_j} \in \Sigma_{A_j})\ w\sigma_{a_j} \in L(\mathbf{G}^h_{I_j})\ \&\ w\sigma_{a_j} \notin L(\boldsymbol{\mathcal{G}}_H, P))\}\big)\Big)$,

   by definition of $W_H(P)$

   $= L\Big(\boldsymbol{\mathcal{G}}_H, P - pr\big(\{q \models R(\boldsymbol{\mathcal{G}}_H, P) |$

   $\big((\exists \sigma_u \in \Sigma_{hu})\ \eta_H \times \xi^h((y, x), \sigma_u)!\ \&$

   $(\delta_H(q, \sigma_u)\ \not!\ or\ (\delta_H(q, \sigma_u)!\ \&\ \delta_H(q, \sigma_u) \not\models P))\big)$ *or*

   $\big((\exists j \in \{1, \ldots, n\})(\exists \sigma_{a_j} \in \Sigma_{A_j})\ \xi^h_j(x_j, \sigma_{a_j})!\ \&$

$$(\delta_H(q,\sigma_{a_j}) \not! \; or \; (\delta_H(q,\sigma_{a_j})! \;\&\; \delta_H(q,\sigma_{a_j}) \not\models P)))\})\Big),$$

where $q = (z, y, x) = (z, y, x_1, \ldots, x_n)$ as in equation (4.1).

$$= L\Big(\boldsymbol{\mathcal{G}}_H, P - pr\big(\{q \models R(\boldsymbol{\mathcal{G}}_H, P)|$$
$$\big((\exists\sigma_u \in \Sigma_{hu})\ (\eta_H \times \xi^h((y,x),\sigma_u)! \;\&\; \delta_H(q,\sigma_u) \not!) \; or$$
$$(\eta_H \times \xi^h((y,x),\sigma_u)! \;\&\; \delta_H(q,\sigma_u)! \;\&\; \delta_H(q,\sigma_u) \not\models P)\big) \; or$$
$$\big((\exists j \in \{1,\ldots,n\})(\exists\sigma_{a_j} \in \Sigma_{A_j})\ (\xi^h_j(x_j,\sigma_{a_j})! \;\&\; \delta_H(q,\sigma_{a_j}) \not!) \; or$$
$$(\xi^h_j(x_j,\sigma_{a_j})! \;\&\; \delta_H(q,\sigma_{a_j})! \;\&\; \delta_H(q,\sigma_{a_j}) \not\models P)\big)\}\big)\Big),$$

by logical distribution law.

$$= L\Big(\boldsymbol{\mathcal{G}}_H, P - pr\big(\{q \models R(\boldsymbol{\mathcal{G}}_H, P)|$$
$$\big((\exists\sigma_u \in \Sigma_{hu})\ (\eta_H \times \xi^h((y,x),\sigma_u)! \;\&\; \delta_H(q,\sigma_u) \not!) \; or$$
$$(\delta_H(q,\sigma_u)! \;\&\; \delta_H(q,\sigma_u) \not\models P)\big) \; or$$
$$\big((\exists j \in \{1,\ldots,n\})(\exists\sigma_{a_j} \in \Sigma_{A_j})\ (\xi^h_j(x_j,\sigma_{a_j})! \;\&\; \delta_H(q,\sigma_{a_j}) \not!) \; or$$
$$(\delta_H(q,\sigma_{a_j})! \;\&\; \delta_H(q,\sigma_{a_j}) \not\models P)\big)\}\big)\Big),$$

as $\delta_H(q,\sigma_u)! \Rightarrow \eta_H \times \xi^h((y,x),\sigma_u)!$ and $\delta_H(q,\sigma_{a_j})! \Rightarrow \xi^h_j(x_j,\sigma_{a_j})!$

by definition of δ_H.

$$= L\Big(\boldsymbol{\mathcal{G}}_H, P - pr\big(\{q \models R(\boldsymbol{\mathcal{G}}_H, P)|$$
$$\big((\exists\sigma_u \in \Sigma_{hu})\ (\eta_H \times \xi^h((y,x),\sigma_u)! \;\&\; \delta_H(q,\sigma_u) \not!) \; or \; \delta_H(q,\sigma_u) \not\models P\big) \; or$$
$$\big((\exists j \in \{1,\ldots,n\})(\exists\sigma_{a_j} \in \Sigma_{A_j})\ (\xi^h_j(x_j,\sigma_{a_j})! \;\&\; \delta_H(q,\sigma_{a_j}) \not!) \; or$$
$$\delta_H(q,\sigma_{a_j}) \not\models P)\}\big)\Big),$$

as $\delta_H(q,\sigma_u) \not\models P \Leftrightarrow \delta_H(q,\sigma_u)! \;\&\; \delta_H(q,\sigma_u) \not\models P$ and

$\delta_H(q,\sigma_{a_j}) \not\models P \Leftrightarrow \delta_H(q,\sigma_{a_j})! \;\&\; \delta_H(q,\sigma_{a_j}) \not\models P.$

$$= L\Big(\boldsymbol{\mathcal{G}}_H, P - pr\big(\{q \models R(\boldsymbol{\mathcal{G}}_H, P)|$$
$$\big((\exists\sigma_u \in \Sigma_{hu})(\eta_H \times \xi^h((y,x),\sigma_u)! \;\;\&\;\; \zeta_H(z,\sigma_u) \not!) \; or \; \delta_H(q,\sigma_u) \not\models P\big) \; or$$
$$\big((\exists j \in \{1,\ldots,n\})(\exists\sigma_{a_j} \in \Sigma_{A_j})\ (\xi^h_j(x_j,\sigma_{a_j})! \;\&$$
$$\zeta_H \times \eta_H \times \xi^h_1 \times \cdots \times \xi^h_{j-1} \times \xi^h_{j+1} \times \cdots \times \xi^h_n((z,y,x_1,\ldots,x_{j-1},x_{j+1},\ldots,x_n),\sigma_{a_j}) \not!) \; or$$

$\delta_H(q,\sigma_{a_j}) \not\models P)\}\big)\Big)$, by definition of δ_H.

$= L\Big(\boldsymbol{\mathcal{G}}_H, P - pr\big(\{q \models R(\boldsymbol{\mathcal{G}}_H,P)|$

$\big((\exists \sigma_u \in \Sigma_{hu})\ (\eta_H \times \xi^h((y,x),\sigma_u)! \ \&\ \zeta_H(z,\sigma_u)\ \not!)$ *or* $\delta_H(q,\sigma_u) \models \neg P\big)$ *or*

$\big((\exists j \in \{1,\ldots,n\})\ (\exists \sigma_{a_j} \in \Sigma_{A_j})\ (\xi^h_j(x_j,\sigma_{a_j})!\ \&$

$\zeta_H \times \eta_H \times \xi^h_1 \times \cdots \times \xi^h_{j-1} \times \xi^h_{j+1} \times \cdots \times \xi^h_n((z,y,x_1,\ldots,x_{j-1},x_{j+1},\ldots,x_n),\sigma_{a_j})\ \not!)$ *or*

$\delta_H(q,\sigma_{a_j}) \models \neg P)\}\big)\Big)$, by $q' \not\models P \Leftrightarrow q' \models \neg P$

$= L\Big(\boldsymbol{\mathcal{G}}_H, P - pr\big(\{q \models R(\boldsymbol{\mathcal{G}}_H,P)|$

$((\exists \sigma_u \in \Sigma_{hu})\ \eta_H \times \xi^h((y,x),\sigma_u)!\ \ \&\ \ \zeta_H(z,\sigma_u)\ \not!)$ *or*

$((\exists \sigma_u \in \Sigma_{hu})\ \delta_H(q,\sigma_u) \models \neg P)$ *or*

$((\exists j \in \{1,\ldots,n\})(\exists \sigma_{a_j} \in \Sigma_{A_j})\ \xi^h_j(x_j,\sigma_{a_j})!\ \&$

$\zeta_H \times \eta_H \times \xi^h_1 \times \cdots \times \xi^h_{j-1} \times \xi^h_{j+1} \times \cdots \times \xi^h_n((z,y,x_1,\ldots,x_{j-1},x_{j+1},\ldots,x_n),\sigma_{a_j})\ \not!)$ *or*

$((\exists j \in \{1,\ldots,n\})(\exists \sigma_{a_j} \in \Sigma_{A_j})\ \delta_H(q,\sigma_{a_j}) \models \neg P)\}\big)\Big)$, by regrouping conditions

$= L(\boldsymbol{\mathcal{G}}_H, P - pr(\mathrm{Bad}_{RH}\ \cup \{q \models R(\boldsymbol{\mathcal{G}}_H,P)|$

$((\exists \sigma_u \in \Sigma_{hu})\ \delta_H(q,\sigma_u) \models \neg P)$ *or*

$((\exists j \in \{1,\ldots,n\})(\exists \sigma_{a_j} \in \Sigma_{A_j})\ \delta_H(q,\sigma_{a_j}) \models \neg P)\})),$

by definition of Bad_{RH}

$= L(\boldsymbol{\mathcal{G}}_H, P - pr(\mathrm{Bad}_{RH}\ \cup \{q \models R(\boldsymbol{\mathcal{G}}_H,P)|$

$((\exists \sigma_u \in \Sigma_{hu})\ \delta_H(q,\sigma_u) \models \neg P)$ *or*

$((\exists \sigma_a \in \Sigma_A)\ \delta_H(q,\sigma_a) \models \neg P)\}))$

$= L(\boldsymbol{\mathcal{G}}_H, P - pr(\mathrm{Bad}_{RH}\ \cup \{q \models R(\boldsymbol{\mathcal{G}}_H,P)\ |\ (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q,\sigma) \models \neg P\}))$

$= L(\boldsymbol{\mathcal{G}}_H, \Pi_{HIC}(P)).$

2. Show that $(\forall i \in \{0,1,\ldots\})\ \Omega^i_{HIC}(L(\boldsymbol{\mathcal{G}}_H,\mathcal{P}_{ha})) = L(\boldsymbol{\mathcal{G}}_H,\Pi^i_{HIC}(\mathcal{P}_{ha}))$

We prove this by induction.

(a) Base case: $(i = 0)$

By the definitions of $\Omega^0_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$ and $\Pi^0_{HIC}(\mathcal{P}_{ha})$ (see section 2.1.3), we have

$\Omega^0_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) = L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})$

$L(\boldsymbol{\mathcal{G}}_H, \Pi^0_{HIC}(\mathcal{P}_{ha})) = L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})$

So, $\Omega^0_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) = L(\boldsymbol{\mathcal{G}}_H, \Pi^0_{HIC}(\mathcal{P}_{ha}))$

(b) Inductive step:

Let $k \in \{0, 1, \ldots\}$. Assume $\Omega^k_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) = L(\boldsymbol{\mathcal{G}}_H, \Pi^k_{HIC}(\mathcal{P}_{ha}))$. Must show this implies $\Omega^{k+1}_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) = L(\boldsymbol{\mathcal{G}}_H, \Pi^{k+1}_{HIC}(\mathcal{P}_{ha}))$.

$$\begin{aligned}
\Omega^{k+1}_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) &= \Omega_{HIC}(\Omega^k_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))) \\
&= \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \Pi^k_{HIC}(\mathcal{P}_{ha}))), \quad \text{by assumption} \\
&= L(\boldsymbol{\mathcal{G}}_H, \Pi^{k+1}_{HIC}(\mathcal{P}_{ha})), \quad \text{by taking } \Pi^k_{HIC}(\mathcal{P}_{ha}) \text{ as our} \\
&\qquad \text{predicate } P \text{ in Point 1 of this proposition.}
\end{aligned}$$

□

Now, if we take a look at the definition of the function Π_{HIC}, we can see that for any $P \preceq \mathcal{P}_{ha}$, the function Π_{HIC} only removes states in $R(\boldsymbol{\mathcal{G}}_H, P)$ from P. Precisely, Π_{HIC} only removes all the states in $pr(\text{Bad}_{RH} \cup \{q \models R(\boldsymbol{\mathcal{G}}_H, P) \mid (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q, \sigma) \models \neg P\})$, which is a subpredicate of $R(\boldsymbol{\mathcal{G}}_H, P)$. If we also remove any state that is not in $R(\boldsymbol{\mathcal{G}}_H, P)$, then the language $L(\boldsymbol{\mathcal{G}}_H, P)$ is not affected at all, because the $L(\boldsymbol{\mathcal{G}}_H, P)$ is only dependent on $R(\boldsymbol{\mathcal{G}}_H, P)$. Similarly, removing any state that is not in $R(\boldsymbol{\mathcal{G}}_H, P)$ does not affect the language $L(\boldsymbol{\mathcal{G}}_H, \Pi_{HIC}(P))$ either, because it is only dependent on a subpredicate of $R(\boldsymbol{\mathcal{G}}_H, P)$. The following function Γ_{HIC} is a replacement of the function Π_{HIC}.

Definition 4.7. For system $\mathbf{\Phi}$, let $\mathcal{P}_{ha} \in Pred(Q_H)$ be a given predicate. Define the function $\Gamma_{HIC} : Sub(\mathcal{P}_{ha}) \to Sub(\mathcal{P}_{ha})$ according to

$(\forall P \in Sub(\mathcal{P}_{ha}))$

$$\Gamma_{HIC}(P) := P - pr(\text{Bad}_H \ \cup \{q \in Q_H \mid (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q, \sigma) \models \neg P\}),$$

where $\text{Bad}_H := \{q \in Q_H |$

$((\exists \sigma_u \in \Sigma_{hu})\ (\eta_H \times \xi^h((y, x), \sigma_u)! \ \ \& \ \ \zeta_H(z, \sigma_u)\ \not!))$ *or*

$\big((\exists j \in \{1, \ldots, n\})(\exists \sigma_{a_j} \in \Sigma_{A_j})\ (\xi^h_j(x_j, \sigma_{a_j})! \ \&$

$\quad \zeta_H \times \eta_H \times \xi^h_1 \times \cdots \times \xi^h_{j-1} \times \xi^h_{j+1} \times \cdots \times \xi^h_n((z, y, x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n), \sigma_{a_j})\ \not!)\big)\}$

and $q = (z, y, x) = (z, y, x_1, \ldots, x_n)$ as in equation (4.1).

$\diamondsuit$

Note that for system $\mathbf{\Phi}$, Bad_H is constant.

Corollary 4.1. *For system* $\mathbf{\Phi}$, *let* $\mathcal{P}_{ha} \in Pred(Q_H)$ *be the given predicate for the function* Ω_{HIC}, Π_{HIC} *and* Γ_{HIC}. *Then the following holds:*

1. $(\forall P \in Sub(\mathcal{P}_{ha}))\ \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P)) = L(\boldsymbol{\mathcal{G}}_H, \Gamma_{HIC}(P))$

2. $(\forall i \in \{0, 1, \ldots\})\ \Omega^i_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) = L(\boldsymbol{\mathcal{G}}_H, \Gamma^i_{HIC}(\mathcal{P}_{ha}))$

proof:

1. Let $P \in Sub(\mathcal{P}_{ha})$, show that $\Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P)) = L(\boldsymbol{\mathcal{G}}_H, \Gamma_{HIC}(P))$

   By the definitions of Π_{HIC} and Γ_{HIC}, and from the above description, we know that removing states that are not in $R(\boldsymbol{\mathcal{G}}_H, P)$ does not affect the language $L(\boldsymbol{\mathcal{G}}_H, \Pi_{HIC}(P))$. We thus have

   $$L(\boldsymbol{\mathcal{G}}_H, \Pi_{HIC}(P)) = L(\boldsymbol{\mathcal{G}}_H, \Gamma_{HIC}(P)). \tag{1}$$

   By Point 1 of Proposition 4.7, we know

$$\Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P)) = L(\boldsymbol{\mathcal{G}}_H, \Pi_{HIC}(P)). \tag{2}$$

By (1) and (2), we have $\Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, P)) = L(\boldsymbol{\mathcal{G}}_H, \Gamma_{HIC}(P))$.

2. Show that $(\forall i \in \{0, 1, \ldots\})\ \Omega^i_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) = L(\boldsymbol{\mathcal{G}}_H, \Pi^i_{HIC}(\mathcal{P}_{ha}))$

   Identical to the proof of Part 2 of Proposition 4.7 after substituting Π_{HIC} with Γ_{HIC}.

□

The following lemma will be used in the next proposition.

Lemma 4.4. *For system* $\boldsymbol{\Phi}$, *let* $\mathcal{P}_{ha} \in Pred(Q_H)$ *be the given predicate for the function* Γ_{HIC}. *The function* Γ_{HIC} *is monotone with respect to* $\preceq$, *i.e.*

$$(\forall P_1, P_2 \in Sub(\mathcal{P}_{ha}))\ P_1 \preceq P_2 \Rightarrow \Gamma_{HIC}(P_1) \preceq \Gamma_{HIC}(P_2)$$

proof:

Let $P_1, P_2 \in Sub(\mathcal{P}_{ha})$. Assume $P_1 \preceq P_2$. Must show implies $\Gamma_{HIC}(P_1) \preceq \Gamma_{HIC}(P_2)$.

$$\begin{aligned}
\Gamma_{HIC}(P_1) &= P_1 - pr(\mathrm{Bad}_H \cup \{q \in Q_H \mid (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\delta_H(q,\sigma) \models \neg P_1\}) \\
&= P_1 \wedge \neg pr(\mathrm{Bad}_H \cup \{q \in Q_H \mid (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\delta_H(q,\sigma) \models \neg P_1\}) \\
&= P_1 \wedge (\neg pr(\mathrm{Bad}_H)) \wedge \neg(pr(\{q \in Q_H \mid ((\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q,\sigma) \models \neg P_1)\})) \\
&= P_1 \wedge (\neg pr(\mathrm{Bad}_H)) \wedge pr(\{q \in Q_H \mid ((\forall \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q,\sigma)! \Rightarrow \delta_H(q,\sigma) \models P_1)\}) \\
&\preceq P_2 \wedge (\neg pr(\mathrm{Bad}_H)) \wedge pr(\{q \in Q_H \mid ((\forall \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q,\sigma)! \Rightarrow \delta_H(q,\sigma) \models P_2)\}), \\
&\qquad \text{by assumption } P_1 \preceq P_2. \\
&= P_2 \wedge (\neg pr(\mathrm{Bad}_H)) \wedge \neg(pr(\{q \in Q_H \mid ((\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q,\sigma) \models \neg P_2)\})) \\
&= P_2 \wedge \neg pr(\mathrm{Bad}_H \cup \{q \in Q_H \mid (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\delta_H(q,\sigma) \models \neg P_2\}) \\
&= P_2 - pr(\mathrm{Bad}_H \cup \{q \in Q_H \mid (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q,\sigma) \models \neg P_2\}) \\
&= \Gamma_{HIC}(P_2)
\end{aligned}$$

□

Proposition 4.8. *For system* $\mathbf{\Phi}$*, let* $\mathcal{P}_{ha} \in Pred(Q_H)$ *be the given predicate for the function* Ω_{HIC} *and* Γ_{HIC}*, then,*

1. *There exists* $k \in \{0, 1, 2, \ldots\}$ *with* $k \leq |st(\mathcal{P}_{ha})|$ *such that* $\Gamma^k_{HIC}(\mathcal{P}_{ha})$ *is the greatest fixpoint of the function* Γ_{HIC} *with respect to* $(Sub(\mathcal{P}_{ha}), \preceq)$*, and where* $st(\mathcal{P}_{ha})$ *is the identifying state subset of* Q_H *for* $\mathcal{P}_{ha}$*.*

2. $\sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) = L(\boldsymbol{\mathcal{G}}_H, \Gamma^k_{HIC}(\mathcal{P}_{ha}))$

proof:

1. Show that there exists $k \leq |st(\mathcal{P}_{ha})|$ such that the greatest fixpoint of the function Γ_{HIC} is equal to $\Gamma^k_{HIC}(\mathcal{P}_{ha})$.

   As we assume the number of states in $\boldsymbol{\mathcal{G}}_H$ is finite (Section 4.1), the number of states in $st(\mathcal{P}_{ha})$ is also finite. By Lemma 4.4, we know that Γ_{HIC} is monotone with respect to $\preceq$, so by Proposition 2.3, there exists $k \leq |st(\mathcal{P}_{ha})|$ such that the greatest fixpoint of the function Γ_{HIC} is equal to $\Gamma^k_{HIC}(\mathcal{P}_{ha})$.

2. Show that $\sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) = L(\boldsymbol{\mathcal{G}}_H, \Gamma^k_{HIC}(\mathcal{P}_{ha}))$

   (a) Show that $\sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) \subseteq L(\boldsymbol{\mathcal{G}}_H, \Gamma^k_{HIC}(\mathcal{P}_{ha}))$

   By Proposition 2.3, we know that

   $(\forall i \in \{0, 1, \ldots\})\ i \geq k \Rightarrow L(\boldsymbol{\mathcal{G}}_H, \Gamma^i_{HIC}(\mathcal{P}_{ha})) = L(\boldsymbol{\mathcal{G}}_H, \Gamma^k_{HIC}(\mathcal{P}_{ha}))$

   $\Rightarrow \lim_{i\to\infty} L(\boldsymbol{\mathcal{G}}_H, \Gamma^i_{HIC}(\mathcal{P}_{ha})) = L(\boldsymbol{\mathcal{G}}_H, \Gamma^k_{HIC}(\mathcal{P}_{ha}))$

   $\Rightarrow \lim_{i\to\infty} \Omega^i_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) = L(\boldsymbol{\mathcal{G}}_H, \Gamma^k_{HIC}(\mathcal{P}_{ha}))$, by Corollary 4.1

   $\Rightarrow \sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) \subseteq L(\boldsymbol{\mathcal{G}}_H, \Gamma^k_{HIC}(\mathcal{P}_{ha}))$, by Proposition 4.5

   (b) Show that $L(\boldsymbol{\mathcal{G}}_H, \Gamma^k_{HIC}(\mathcal{P}_{ha})) \subseteq \sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$

   From Point 1 of this proposition and Proposition 2.3, we have

$$\Gamma^k_{HIC}(\mathcal{P}_{ha}) = \Gamma^{k+1}_{HIC}(\mathcal{P}_{ha})$$
$$\Rightarrow L(\boldsymbol{\mathcal{G}}_H, \Gamma^k_{HIC}(\mathcal{P}_{ha})) = L(\boldsymbol{\mathcal{G}}_H, \Gamma^{k+1}_{HIC}(\mathcal{P}_{ha})) \quad (1)$$
$$\Rightarrow L(\boldsymbol{\mathcal{G}}_H, \Gamma^k_{HIC}(\mathcal{P}_{ha})) = \Omega^{k+1}_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})), \quad \text{by Corollary 4.1}$$
$$\Rightarrow L(\boldsymbol{\mathcal{G}}_H, \Gamma^k_{HIC}(\mathcal{P}_{ha})) = \Omega_{HIC}(\Omega^k_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})))$$
$$\Rightarrow L(\boldsymbol{\mathcal{G}}_H, \Gamma^k_{HIC}(\mathcal{P}_{ha})) = \Omega_{HIC}(L(\boldsymbol{\mathcal{G}}_H, \Gamma^k_{HIC}(\mathcal{P}_{ha}))), \quad \text{by Corollary 4.1}$$

So, $L(\boldsymbol{\mathcal{G}}_H, \Gamma^k_{HIC}(\mathcal{P}_{ha}))$ is a fixpoint of the function Ω_{HIC}. By Proposition 4.4, $\sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$ is the greatest fixpoint of Ω_{HIC}, thus,

$$L(\boldsymbol{\mathcal{G}}_H, \Gamma^k_{HIC}(\mathcal{P}_{ha})) \subseteq \sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$$

□

We are now able to write a computer program to compute $\sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha}))$ by computing $\Gamma^k_{HIC}(\mathcal{P}_{ha})$ ($k \in \{0, 1, 2, \ldots\}$), and we know that after at most $|st(\mathcal{P}_{ha})|$ number of iterations, a fixpoint of Γ_{HIC} will be reached, which is the greatest fixpoint of Γ_{HIC} by Proposition 2.3. However, such a program based on function Γ_{HIC} is not efficient enough. For instance, each time when we implement the Γ_{HIC} function, we have to do the predicate subtracting operation. The following alternative method can avoid this.

Definition 4.8. For system $\boldsymbol{\Phi}$, define the function $\mathrm{PHIC} : Pred(Q_H) \to Pred(Q_H)$ according to

$$(\forall P \in Pred(Q_H))\ \mathrm{PHIC}(P) := \neg TR(\boldsymbol{\mathcal{G}}_H, \neg P \vee pr(\mathrm{Bad}_H), \Sigma_{hu} \cup \Sigma_A).$$

♢

The following lemma will be used in the next proposition.

Lemma 4.5. *For system* $\boldsymbol{\Phi}$, *the following two points hold:*

1. *The function* PHIC *is monotone with respect to* $\preceq$, *i.e.*

   $(\forall P_1, P_2 \in Pred(Q_H))\ P_1 \preceq P_2 \Rightarrow \mathrm{PHIC}(P_1) \preceq \mathrm{PHIC}(P_2)$.

2. $(\forall P \in Pred(Q_H))$ PHIC$(P) \preceq P$.

proof:

1. Show that PHIC is monotone.

   Let $P_1, P_2 \in Pred(Q_H)$.

   Assume $P_1 \preceq P_2$. Must show this implies PHIC$(P_1) \preceq$ PHIC(P_2).

   By assumption $P_1 \preceq P_2$, we have $\neg P_2 \preceq \neg P_1$.

   $\Rightarrow TR(\boldsymbol{\mathcal{G}}_H, \neg P_2 \vee pr(\text{Bad}_H), \Sigma_{hu} \cup \Sigma_A) \preceq TR(\boldsymbol{\mathcal{G}}_H, \neg P_1 \vee pr(\text{Bad}_H), \Sigma_{hu} \cup \Sigma_A)$,

   as TR is monotone

   $\Rightarrow \neg$PHIC$(P_2) \preceq \neg$PHIC(P_1)

   $\Rightarrow$ PHIC$(P_1) \preceq$ PHIC(P_2)

2. Show that $(\forall P \in Pred(Q_H))$ PHIC$(P) \preceq P$.

   Let $P \in Pred(Q_H)$. We will now show that PHIC$(P) \preceq P$.

   PHIC$(P) = \neg TR(\boldsymbol{\mathcal{G}}_H, \neg P \vee pr(\text{Bad}_H), \Sigma_{hu} \cup \Sigma_A)$

   $\Rightarrow \neg$PHIC$(P) = TR(\boldsymbol{\mathcal{G}}_H, \neg P \vee pr(\text{Bad}_H), \Sigma_{hu} \cup \Sigma_A)$

   $\Rightarrow \neg P \preceq \neg$PHIC$(P)$, by definition of TR and fact $\neg P \preceq \neg P \vee pr(\text{Bad}_H)$

   $\Rightarrow$ PHIC$(P) \preceq P$.

□

Proposition 4.9. *For system* $\boldsymbol{\Phi}$, *the following holds:*

$$(\forall \mathcal{P}_{ha} \in Pred(Q_H)) \ \sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \mathcal{P}_{ha})) = L(\boldsymbol{\mathcal{G}}_H, \text{PHIC}(\mathcal{P}_{ha})).$$

proof:

Let $\mathcal{P}_{ha} \in Pred(Q_H)$ be the given predicate for the function Γ_{HIC} (i.e. $Sub(\mathcal{P}_{ha})$ is the domain and codomain for Γ_{HIC}).

From Lemma 4.5, we know $\text{PHIC}(\mathcal{P}_{ha}) \preceq \mathcal{P}_{ha}$, so $\text{PHIC}(\mathcal{P}_{ha}) \in Sub(\mathcal{P}_{ha})$ and is thus a valid input of Γ_{HIC}.

From Proposition 4.8, sufficient to show $\text{PHIC}(\mathcal{P}_{ha})$ is the greatest fixpoint of Γ_{HIC}.

1. Show that $\text{PHIC}(\mathcal{P}_{ha})$ is a fixpoint of Γ_{HIC}, i.e. $\text{PHIC}(\mathcal{P}_{ha}) = \Gamma_{HIC}(\text{PHIC}(\mathcal{P}_{ha}))$.

   $\Gamma_{HIC}(\text{PHIC}(\mathcal{P}_{ha})) = \text{PHIC}(\mathcal{P}_{ha}) -$

   $$pr(\text{Bad}_H \ \cup \{q \in Q_H \mid (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q, \sigma) \models \neg\text{PHIC}(\mathcal{P}_{ha})\})$$

   To show $\Gamma_{HIC}(\text{PHIC}(\mathcal{P}_{ha})) = \text{PHIC}(\mathcal{P}_{ha})$, sufficient to show that

   $$(\forall q' \in Q_H)\ q' \models \text{PHIC}(\mathcal{P}_{ha}) \Rightarrow q' \not\models pr(\text{Bad}_H)\ \&$$
   $$q' \notin \{q \in Q_H \mid (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q, \sigma) \models \neg\text{PHIC}(\mathcal{P}_{ha})\}$$

   Let $q' \in Q_H$. Assume $q' \models \text{PHIC}(\mathcal{P}_{ha})$. (1)

   Must show the following two points.

   (a) Show $q' \not\models pr(\text{Bad}_H)$.

   By (1), we have $q' \models \text{PHIC}(\mathcal{P}_{ha})$

   $\Rightarrow q' \not\models TR(\boldsymbol{\mathcal{G}}_H, \neg\mathcal{P}_{ha} \vee pr(\text{Bad}_H), \Sigma_{hu} \cup \Sigma_A)$

   $\Rightarrow q' \not\models pr(\text{Bad}_H)$, by definition of TR

   (b) Show $q' \notin \{q \in Q_H \mid (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q, \sigma) \models \neg\text{PHIC}(\mathcal{P}_{ha})\}$

   We prove this by contradiction.

   Assume $q' \in \{q \in Q_H \mid (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q, \sigma) \models \neg\text{PHIC}(\mathcal{P}_{ha})\}$.

   $\Rightarrow q' \in \{q \in Q_H \mid$

   $$(\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q, \sigma) \models TR(\boldsymbol{\mathcal{G}}_H, \neg\mathcal{P}_{ha} \vee pr(\text{Bad}_H), \Sigma_{hu} \cup \Sigma_A)\}$$

$\Rightarrow (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q', \sigma) \models TR(\boldsymbol{\mathcal{G}}_H, \neg \mathcal{P}_{ha} \vee pr(\text{Bad}_H), \Sigma_{hu} \cup \Sigma_A)$

$\Rightarrow q' \models TR(\boldsymbol{\mathcal{G}}_H, \neg \mathcal{P}_{ha} \vee pr(\text{Bad}_H), \Sigma_{hu} \cup \Sigma_A), \quad$ by definition of TR

$\Rightarrow q' \models \neg \text{PHIC}(\mathcal{P}_{ha})$,

which contradicts with (1): the assumption $q' \models \text{PHIC}(\mathcal{P}_{ha})$, so

$q' \notin \{q \in Q_H \mid (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q, \sigma) \models \neg \text{PHIC}(\mathcal{P}_{ha})\}$.

2. Show that $\text{PHIC}(\mathcal{P}_{ha})$ is the greatest fixpoint of Γ_{HIC}, i.e.

$$(\forall P' \in Sub(\mathcal{P}_{ha}))\ P' = \Gamma_{HIC}(P') \Rightarrow P' \preceq \text{PHIC}(\mathcal{P}_{ha}).$$

Let $P' \in Sub(\mathcal{P}_{ha})$. Assume $P' = \Gamma_{HIC}(P')$. (2)

Must show this implies $P' \preceq \text{PHIC}(\mathcal{P}_{ha})$.

We show this by contradiction.

Assume $P' \not\preceq \text{PHIC}(\mathcal{P}_{ha})$

$\Rightarrow (\exists q' \models P') q' \not\models \text{PHIC}(\mathcal{P}_{ha})$

$\Rightarrow (\exists q' \models P') q' \models TR(\boldsymbol{\mathcal{G}}_H, \neg \mathcal{P}_{ha} \vee pr(\text{Bad}_H), \Sigma_{hu} \cup \Sigma_A)$ (3)

By the definition of Γ_{HIC}, we have

$$\begin{aligned}
\Gamma_{HIC}(P') &= P' - pr(\text{Bad}_H \cup \{q \in Q \mid (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q, \sigma) \models \neg P'\}) \\
&= P' \wedge \neg pr(\text{Bad}_H \cup \{q \in Q \mid (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q, \sigma) \models \neg P'\}) \\
&= P' \wedge \neg pr(\text{Bad}_H) \wedge \neg pr(\{q \in Q \mid (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q, \sigma) \models \neg P'\}) \\
&= P' - pr(\text{Bad}_H) - pr(\{q \in Q \mid (\exists \sigma \in \Sigma_{hu} \cup \Sigma_A)\ \delta_H(q, \sigma) \models \neg P'\}) \quad (4)
\end{aligned}$$

From (3), by the definition of TR, one of the following two conditions must be satisfied.

(a) $q' \models \neg \mathcal{P}_{ha} \vee pr(\text{Bad}_H)$

By (3), we also have $q' \models P'$

$\Rightarrow q' \models \mathcal{P}_{ha}, \quad$ as $P' \in Sub(\mathcal{P}_{ha})$

$\Rightarrow q' \models \neg(\neg\mathcal{P}_{ha})$

$\Rightarrow q' \not\models \neg\mathcal{P}_{ha}$

$\Rightarrow q' \models pr(\text{Bad}_H), \quad$ as $q' \models \neg\mathcal{P}_{ha} \vee pr(\text{Bad}_H)$.

As $q' \models pr(\text{Bad}_H)$ and $\Gamma_{HIC}(P') = P'$, by (4), we thus have $q' \not\models P'$, which contradicts $q' \models P'$ in (3).

(b) $q' \not\models \neg\mathcal{P}_{ha} \vee pr(\text{Bad}_H)$

$\Rightarrow (\exists s \in (\Sigma_{hu} \cup \Sigma_A)^+)\ \delta_H(q', s) \models \neg\mathcal{P}_{ha} \vee pr(\text{Bad}_H)$, by definition of TR

$\Rightarrow (\exists k \in \{1, 2, \ldots\})\ (\exists q_1, q_2, \ldots, q_{k+1} \in Q_H)\ (\exists \sigma_1, \sigma_2, \ldots, \sigma_k \in \Sigma_{hu} \cup \Sigma_A)$

$$
\begin{aligned}
q_1 &:= q' \\
q_{k+1} &\models \neg\mathcal{P}_{ha} \vee pr(\text{Bad}_H) \qquad (5) \\
\delta_H(q_i, \sigma_i) &= q_{i+1}, i = 1, 2, \ldots, k \\
s &= \sigma_1\sigma_2 \ldots \sigma_k
\end{aligned}
$$

First,we show that $q_{k+1} \models \neg P'$.

By (5), we have $q_{k+1} \models \neg\mathcal{P}_{ha} \vee pr(\text{Bad}_H)$.

If $q_{k+1} \models \neg\mathcal{P}_{ha}$, as $P' \preceq \mathcal{P}_{ha}$, we have $\neg\mathcal{P}_{ha} \preceq \neg P'$; thus $q_{k+1} \models \neg P'$.

If $q_{k+1} \models pr(\text{Bad}_H)$, by (4) and assumption $\Gamma_{HIC}(P') = P'$, we also have $q_{k+1} \models \neg P'$.

Thus, in both cases we have $q_{k+1} \models \neg P'$. (6)

Now we show that $q_k \models \neg P'$.

By (5), we know $\delta_H(q_k, \sigma_k) = q_{k+1}$. By (6), we know $q_{k+1} \models \neg P'$. By (4) and assumption $\Gamma_{HIC}(P') = P'$, we thus have $q_k \models \neg P'$.

Similarly, we can prove that $q_i \models \neg P'$ for $i = k-1, \ldots, 1$. As $q_1 = q'$ by (5), we have $q' \models \neg P'$, which contradicts $q' \models P'$ in (3).

As Part (a) and Part (b) both will cause contradictions, we have $P' \preceq \text{PHIC}(\mathcal{P}_{ha})$.

□

4.2.5 Computing $\Omega_{HNB}(L(\boldsymbol{\mathcal{G}}_H, P))$

For a predicate $P \in Pred(Q_H)$, in the previous section, we have found a way to compute $\sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, P))$. To complete the computation for the function Ω_H, we now show the method to compute $\Omega_{HNB}(L(\boldsymbol{\mathcal{G}}_H, P))$, *i.e.* the method to compute $L_m(\boldsymbol{\mathcal{G}}_H) \cap L(\boldsymbol{\mathcal{G}}_H, P)$.

Proposition 4.10. *For system* $\boldsymbol{\Phi}$, *the following holds:*

$$(\forall P \in Pred(Q_H))\ L_m(\boldsymbol{\mathcal{G}}_H) \cap L(\boldsymbol{\mathcal{G}}_H, P) = L_m(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P))$$

proof:

Let $P \in Pred(Q_H)$.

1. Show that $L_m(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P)) \subseteq L_m(\boldsymbol{\mathcal{G}}_H) \cap L(\boldsymbol{\mathcal{G}}_H, P)$

   Clearly, $L_m(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P)) \subseteq L_m(\boldsymbol{\mathcal{G}}_H)$. (1)

   By the definition of $CR(\boldsymbol{\mathcal{G}}_H, .)$, we know $CR(\boldsymbol{\mathcal{G}}_H, P) \preceq P$.

   $\Rightarrow L_m(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P)) \subseteq L_m(\boldsymbol{\mathcal{G}}_H, P)$

   $\Rightarrow L_m(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P)) \subseteq L(\boldsymbol{\mathcal{G}}_H, P)$ (2)

   By (1) and (2), $L_m(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P)) \subseteq L_m(\boldsymbol{\mathcal{G}}_H) \cap L(\boldsymbol{\mathcal{G}}_H, P)$.

2. Show that $L_m(\boldsymbol{\mathcal{G}}_H) \cap L(\boldsymbol{\mathcal{G}}_H, P) \subseteq L_m(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P))$

   Let $s \in L_m(\boldsymbol{\mathcal{G}}_H) \cap L(\boldsymbol{\mathcal{G}}_H, P)$. (3)

Must show implies $s \in L_m(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P))$.

Sufficient to show $s \in L(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P))$ and $\delta_H(q_{H_0}, s) \in Q_{H_m}$, where Q_{H_m} is the marker state set of $\boldsymbol{\mathcal{G}}_H$ as defined in Section 4.1.

From (3), we know $s \in L_m(\boldsymbol{\mathcal{G}}_H)$

$$\Rightarrow \delta_H(q_{H_0}, s) \in Q_{H_m} \tag{4}$$

From (3), we also know $s \in L(\boldsymbol{\mathcal{G}}_H, P)$

$$\Rightarrow (\exists n \in \{0, 1, \ldots\})(\exists q_0, q_1, \ldots, q_n \in Q_H)(\exists \sigma_0, \sigma_1, \ldots, \sigma_{n-1} \in \Sigma_{IH})$$

$$\begin{aligned} q_0 &= q_{H_0} \quad (q_{H_0} \text{ is the initial state of } \boldsymbol{\mathcal{G}}_H) \\ q_i &\models P, \quad i = 0, 1, \ldots, n \\ \delta_H(q_i, \sigma_i) &= q_{i+1}, \quad i = 0, 1, \ldots, n-1 \\ s &= \sigma_0 \sigma_1 \cdots \sigma_{n-1} \\ q_n &= \delta_H(q_{H_0}, s). \end{aligned} \tag{5}$$

By (4) and (5), we know $(\forall i \in \{0, 1, \ldots, n\})\ q_i \models CR(\boldsymbol{\mathcal{G}}_H, P)$

$\Rightarrow s \in L(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P))$, as $q_0 = q_{H_0}$ by (5)

□

4.2.6 The Algorithm to Compute $\sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H))$

It is time to put everything for the high-level together and to obtain our algorithm.

Definition 4.9. For system $\boldsymbol{\Phi}$, define the function $\Gamma_H : Pred(Q_H) \rightarrow Pred(Q_H)$ according to

$$(\forall P \in Pred(Q_H))\ \ \Gamma_H(P) := CR(\boldsymbol{\mathcal{G}}_H, \text{PHIC}(P))$$

♢

As PHIC (by Lemma 4.5) and the predicate transformer $CR(\boldsymbol{\mathcal{G}}_H, .)$ are monotone, the function Γ_H is monotone.

Lemma 4.6. *For system* $\boldsymbol{\Phi}$, *the following holds:*

$$(\forall P \in Pred(Q_H))\ \overline{L_m(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P))} = L(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P))$$

proof:

Let $P \in Pred(Q_H)$.

$\overline{L_m(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P))} \subseteq L(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P))$ is automatic.

We now show $L(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P)) \subseteq \overline{L_m(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P))}$.

Let $s \in L(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P))$. (1)

Must show this implies $s \in \overline{L_m(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P))}$.

Sufficient to show $(\exists s' \in \Sigma_{IH}^*)\ ss' \in L_m(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P))$.

By (1), we know $s \in L(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P))$

$\Rightarrow (\exists k \in \{0, 1, \ldots\})(\exists q_0, q_1, \ldots, q_k \in Q_H)(\exists \sigma_0, \sigma_1, \ldots, \sigma_{k-1} \in \Sigma_{IH})$

$$\begin{aligned} q_0 &= q_{H_0} \\ q_i &\models CR(\boldsymbol{\mathcal{G}}_H, P), \quad i = 0, 1, \ldots, k \\ \delta_H(q_i, \sigma_i) &= q_{i+1}, \quad i = 0, 1, \ldots, k-1 \\ s &= \sigma_0 \sigma_1 \cdots \sigma_{k-1}. \end{aligned} \tag{2}$$

As $q_k \in CR(\boldsymbol{\mathcal{G}}_H, P)$, it follows that

$((\exists n \in \{0, 1, \ldots\})\ n \geq k)(\exists q_{k+1}, \ldots, q_n \in Q_H)(\exists \sigma_k, \sigma_{k+1}, \ldots, \sigma_{n-1} \in \Sigma_{IH})$

$$\begin{aligned} q_n &\in Q_{H_m} \\ q_i &\models CR(\boldsymbol{\mathcal{G}}_H, P), \quad i = k, k+1, \ldots, n \\ \delta_H(q_i, \sigma_i) &= q_{i+1}, \quad i = k, k+1, \ldots, n-1 \end{aligned} \tag{3}$$

Let $s' = \sigma_k \sigma_{k+1} \dots \sigma_{n-1}$. Combining (2) and (3), we thus have

$$(\exists n \in \{0, 1, \dots\})(\exists q_0, q_1, \dots, q_n \in Q_H)(\exists \sigma_0, \sigma_1, \dots, \sigma_{n-1} \in \Sigma_{IH})$$

$$q_0 = q_{H_0}$$

$$q_n \in Q_{H_m}$$

$$q_i \models CR(\boldsymbol{\mathcal{G}}_H, P), \quad i = 0, 1, \dots, n$$

$$\delta_H(q_i, \sigma_i) = q_{i+1}, \quad i = 0, 1, \dots, n-1$$

$$ss' = \sigma_0 \sigma_1 \cdots \sigma_{n-1}$$

$$\Rightarrow ss' \in L_m(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P))$$

□

Lemma 4.7. *For system* $\boldsymbol{\Phi}$, *the following holds:*

$$(\forall P \in Pred(Q_H))(\forall i \in \{1, 2, \dots\})\ \overline{L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^i(P))} = L(\boldsymbol{\mathcal{G}}_H, \Gamma_H^i(P))$$

proof:

Let $i \in \{1, 2, \dots\}$, and $P \in Pred(Q_H)$. By definition of Γ_H, we have

$$\begin{aligned}
\overline{L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^i(P))} &= \overline{L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H(\Gamma_H^{i-1}(P)))} \\
&= \overline{L_m(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, \text{PHIC}(\Gamma_H^{i-1}(P))))}, \quad \text{by definition of } \Gamma_H \\
&= \overline{L_m(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P'))} \quad \text{by letting } P' = \text{PHIC}(\Gamma_H^{i-1}(P)) \\
&= L(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, P')), \quad \text{by Lemma 4.6} \\
&= L(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, \text{PHIC}(\Gamma_H^{i-1}(P)))) \\
&= L(\boldsymbol{\mathcal{G}}_H, \Gamma_H^i(P)), \quad \text{by definition of } \Gamma_H
\end{aligned}$$

□

Proposition 4.11. *For system* $\boldsymbol{\Phi}$, *the following two points hold:*

1. $(\forall P \in Pred(Q_H))\ \Omega_H(L(\boldsymbol{\mathcal{G}}_H, P)) = L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H(P))$

2. $(\forall P \in Pred(Q_H))(\forall i \in \{1, 2, \dots\})\ \Omega_H^i(L(\boldsymbol{\mathcal{G}}_H, P)) = L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^i(P))$

proof:

1. Show that $(\forall P \in Pred(Q_H))\ \Omega_H(L(\boldsymbol{\mathcal{G}}_H, P)) = L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H(P))$

   Let $P \in Pred(Q_H)$.

   $$\begin{aligned}
   \Omega_H(L(\boldsymbol{\mathcal{G}}_H, P)) &= L_m(\boldsymbol{\mathcal{G}}_H) \cap \sup \mathcal{C}_H(\overline{L(\boldsymbol{\mathcal{G}}_H, P)}), \quad \text{by definition of } \Omega_H \\
   &= L_m(\boldsymbol{\mathcal{G}}_H) \cap \sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, P)) \\
   &= L_m(\boldsymbol{\mathcal{G}}_H) \cap L(\boldsymbol{\mathcal{G}}_H, \text{PHIC}(P)), \quad \text{by Proposition 4.9} \\
   &= L_m(\boldsymbol{\mathcal{G}}_H, CR(\boldsymbol{\mathcal{G}}_H, \text{PHIC}(P))), \quad \text{by Proposition 4.10} \\
   &= L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H(P)), \quad \text{by definition of } \Gamma_H
   \end{aligned}$$

2. Show that $(\forall P \in Pred(Q_H))(\forall i \in \{1, 2, \ldots\})\ \Omega_H^i(L(\boldsymbol{\mathcal{G}}_H, P)) = L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^i(P))$

   Let $P \in Pred(Q_H)$. We prove this by induction.

   (a) Base Case: $i = 1$

   By Point 1 of this proposition, we know $\Omega_H(L(\boldsymbol{\mathcal{G}}_H, P)) = L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H(P))$

   $\Rightarrow \Omega_H^1(L(\boldsymbol{\mathcal{G}}_H, P)) = L_m^1(\boldsymbol{\mathcal{G}}_H, \Gamma_H(P))$

   (b) Inductive step.

   Let $k \in \{1, 2, \ldots\}$. Assume $\Omega_H^k(L(\boldsymbol{\mathcal{G}}_H, P)) = L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^k(P))$. Must show $\Omega_H^{k+1}(L(\boldsymbol{\mathcal{G}}_H, P)) = L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^{k+1}(P))$.

   $$\begin{aligned}
   \Omega_H^{k+1}(L(\boldsymbol{\mathcal{G}}_H, P)) &= \Omega_H(\Omega_H^k(L(\boldsymbol{\mathcal{G}}_H, P))) \\
   &= \Omega_H(L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^k(P))), \quad \text{by inductive assumption} \\
   &= L_m(\boldsymbol{\mathcal{G}}_H) \cap \sup \mathcal{C}_H(\overline{L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^k(P))}), \quad \text{by definition of } \Omega_H \\
   &= L_m(\boldsymbol{\mathcal{G}}_H) \cap \sup \mathcal{C}_H(L(\boldsymbol{\mathcal{G}}_H, \Gamma_H^k(P))), \quad \text{by Lemma 4.7} \\
   &= \Omega_H(L(\boldsymbol{\mathcal{G}}_H, \Gamma_H^k(P))), \quad \text{by definition of } \Omega_H \\
   &= L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^{k+1}(P)), \quad \text{by Point 1 of this proposition.}
   \end{aligned}$$

□

Theorem 4.1. *For system* $\mathbf{\Phi}$*, the following two points hold:*

1. *There exists* $k \in \{0, 1, \ldots\}$ *such that* $k \leq |Q_H|$ *and* $\Gamma_H^k(true)$ *is the greatest fixpoint of the function* Γ_H *with respect to* $(Pred(Q_H), \preceq)$.

2. $\sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H)) = \sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H, true)) = L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^k(true))$.

proof:

1. Show that there exists $k \in \{0, 1, \ldots\}$ such that $k \leq |Q_H|$ and $\Gamma_H^k(true)$ is the greatest fixpoint of the function Γ_H with respect to $(Pred(Q_H), \preceq)$.

   As Γ_H is monotone and $|Q_H|$ is assumed to be finite (Section 4.1), we know immediately from Proposition 2.3 that there exists $k \in \{0, 1, \ldots\}$ such that $k \leq |Q_H|$ and $\Gamma_H^k(true)$ is the greatest fixpoint of the function Γ_H.

2. Show that $\sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H)) = \sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H, true)) = L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^k(true))$.

   $\sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H)) = \sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H, true))$ is automatic as $L_m(\boldsymbol{\mathcal{G}}_H) = L_m(\boldsymbol{\mathcal{G}}_H, true)$.

   (a) Show that $\sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H, true)) \subseteq L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^k(true))$

   By Proposition 2.3, we know that

   $(\forall i \in \{1, 2, \ldots\})\ i > k \Rightarrow L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^i(true)) = L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^k(true))$

   $\Rightarrow \lim_{i \to \infty} L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^i(true)) = L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^k(true))$

   $\Rightarrow \lim_{i \to \infty} \Omega_H^i(L(\boldsymbol{\mathcal{G}}_H, true)) = L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^k(true))$, by Proposition 4.11

   $\Rightarrow \sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H, true)) \subseteq L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^k(true))$, by Proposition 4.3

   (b) Show that $L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^k(true)) \subseteq \sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H, true))$.

   Based on the value of k, we show this in two cases:

- Case 1: $k \in \{1, 2, \ldots\}$

  By Point 1 of this theorem and Proposition 2.3, we have

  $\Gamma^k_H(true) = \Gamma^{k+1}_H(true)$

  $\Rightarrow L_m(\boldsymbol{\mathcal{G}}_H, \Gamma^k_H(true)) = L_m(\boldsymbol{\mathcal{G}}_H, \Gamma^{k+1}_H(true))$

  $\Rightarrow L_m(\boldsymbol{\mathcal{G}}_H, \Gamma^k_H(true)) = \Omega^{k+1}_H(L(\boldsymbol{\mathcal{G}}_H, true)),$ by Proposition 4.11

  $\Rightarrow L_m(\boldsymbol{\mathcal{G}}_H, \Gamma^k_H(true)) = \Omega_H(\Omega^k_H(L(\boldsymbol{\mathcal{G}}_H, true)))$

  $\Rightarrow L_m(\boldsymbol{\mathcal{G}}_H, \Gamma^k_H(true)) = \Omega_H(L_m(\boldsymbol{\mathcal{G}}_H, \Gamma^k_H(true))),$

  by Proposition 4.11 and $k \in \{1, 2, \ldots\}$

- Case 2: $k = 0$

  If $\Gamma^0_H(true)$ is the greatest fixpoint of Γ_H, then by Proposition 2.3 we know that

  $$\Gamma^1_H(true) = \Gamma^0_H(true), \tag{1}$$

  so $\Gamma^1_H(true)$ is also the greatest fixpoint of Γ_H.

  $\Rightarrow L_m(\boldsymbol{\mathcal{G}}_H, \Gamma^1_H(true)) = \Omega_H(L_m(\boldsymbol{\mathcal{G}}_H, \Gamma^1_H(true))),$ by Case 1

  $\Rightarrow L_m(\boldsymbol{\mathcal{G}}_H, \Gamma^0_H(true)) = \Omega_H(L_m(\boldsymbol{\mathcal{G}}_H, \Gamma^0_H(true))),$ by (1)

From Case 1 and Case 2, we know that $L_m(\boldsymbol{\mathcal{G}}_H, \Gamma^k_H(true))$ is always a fixpoint of the function Ω_H.

By Proposition 4.2, $\sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H, true))$ is the greatest fixpoint of Ω_H, so $L_m(\boldsymbol{\mathcal{G}}_H, \Gamma^k_H(true)) \subseteq \sup \mathcal{C}_H(L_m(\boldsymbol{\mathcal{G}}_H, true))$.

□

Corollary 4.2. *For system* $\boldsymbol{\Phi}$, *let* $\mathbf{S}_H$ *be a DES defined over event set* Σ_{IH} *with* $L(\mathbf{S}_H) = L(\boldsymbol{\mathcal{G}}_H, \Gamma^k_H(true))$ *and* $L_m(\mathbf{S}_H) = L_m(\boldsymbol{\mathcal{G}}_H, \Gamma^k_H(true))$, *where* $k \in \{0, 1, \ldots\}$ *and* $\Gamma^k_H(true)$ *is the greatest fixpoint of* Γ_H *with respect to* $(Pred(Q_H), \preceq)$. *Then for the* n^{th} *degree parallel interface system composed of* $\mathbf{G}^p_H$, $\mathbf{G}^p_{L_1}$, …, $\mathbf{G}^p_{L_n}$, $\mathbf{G}_{I_1}$, …,

$\mathbf{G}_{I_n}$, $\mathbf{S}_H$, $\mathbf{E}_{L_1}$, ..., $\mathbf{E}_{L_n}$ *with respect to the alphabet partition in Equation 3.1,* $\mathbf{S}_H$ *is a high-level proper supervisor.*

proof:

We first note that by Theorem 4.2, an appropriate $k \in \{0, 1, \ldots\}$ exists such that $\Gamma_H^k(true)$ is the greatest fixpoint of Γ_H with respect to $(Pred(Q_H), \preceq)$.

1. Show that $\overline{L_m(\mathbf{S}_H \times \mathbf{G}_H^p \times \mathbf{G}_I^h)} = L(\mathbf{S}_H \times \mathbf{G}_H^p \times \mathbf{G}_I^h)$.

   $L_m(\mathbf{S}_H) = L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^k(true))$

   $\Rightarrow L_m(\mathbf{S}_H) \subseteq L_m(\boldsymbol{\mathcal{G}}_H)$

   $\Rightarrow L_m(\mathbf{S}_H) \subseteq L_m(\mathbf{E}_H \times \mathbf{G}_H^p \times \mathbf{G}_I^h)$

   $\Rightarrow L_m(\mathbf{S}_H) \subseteq L_m(\mathbf{G}_H^p \times \mathbf{G}_I^h)$

   $\Rightarrow L_m(\mathbf{S}_H) = L_m(\mathbf{S}_H \times \mathbf{G}_H^p \times \mathbf{G}_I^h)$ (1)

   Similarly, we can show that $L(\mathbf{S}_H) = L(\mathbf{S}_H \times \mathbf{G}_H^p \times \mathbf{G}_I^h)$ (2)

   If $k \in \{1, 2, \ldots\}$ and $\Gamma_H^k(true)$ is the greatest fixpoint of Γ_H, then by Lemma 4.7 we know that $\overline{L_m(\mathbf{S}_H)} = L(\mathbf{S}_H)$.

   If $k = 0$ and $\Gamma_H^0(true)$ is the greatest fixpoint of Γ_H, then as $|Q_H|$ is assumed to be finite (Section 4.1), by Proposition 2.3 we know that $\Gamma_H^1(true) = \Gamma_H^0(true)$.

   $\Rightarrow L(\mathbf{S}_H) = L(\boldsymbol{\mathcal{G}}_H, \Gamma_H^1(true))$ and $L_m(\mathbf{S}_H) = L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^1(true))$

   $\Rightarrow \overline{L_m(\mathbf{S}_H)} = L(\mathbf{S}_H)$, by Lemma 4.7.

   Therefore, we always have

   $$\overline{L_m(\mathbf{S}_H)} = L(\mathbf{S}_H), \tag{3}$$

   when $k \in \{0, 1, \ldots\}$ and $\Gamma_H^k(true)$ is the greatest fixpoint of Γ_H.

   Combining (1), (2) and (3), we can conclude $\overline{L_m(\mathbf{S}_H \times \mathbf{G}_H^p \times \mathbf{G}_I^h)} = L(\mathbf{S}_H \times \mathbf{G}_H^p \times \mathbf{G}_I^h)$.

2. Show that $\mathbf{S}_H$ is high-level interface controllable.

   By Part 1, Definition 3.8 and Definition 4.2, it is sufficient to show that $L_m(\mathbf{S}_H \times \mathbf{G}_H^p \times \mathbf{G}_I^h)$ is high-level interface controllable.

   From Theorem 4.1, we know $L_m(\mathbf{S}_H)$ is high-level interface controllable.

   By (1), we can conclude $L_m(\mathbf{S}_H \times \mathbf{G}_H^p \times \mathbf{G}_I^h)$ is also high-level interface controllable.

□

The supervisor $\mathbf{S}_H$ can be built by trimming off states from $\boldsymbol{\mathcal{G}}_H$ that do not satisfy $\Gamma_H^k(true)$. A trim supervisor DES $\mathbf{S}_H$ can be built by trimming states from $\boldsymbol{\mathcal{G}}_H$ that do not satisfy $R(\boldsymbol{\mathcal{G}}_H, \Gamma_H^k(true))$.

Algorithm 4.1 shows how to compute $\Gamma_H^k(true)$, where $k \in \{0, 1, \ldots\}$ and $\Gamma_H^k(true)$ is the greatest fixpoint of Γ_H with respect to $(Pred(Q_H), \preceq)$. By Point 1 of Theorem 4.1, we know that the greatest fixpoint will be reached after finite number of iterations.

Algorithm 4.1 Computing the greatest fixpoint of Γ_H w.r.t. $(Pred(Q_H), \preceq)$

1: $P_{bad_H} \leftarrow pr(\mathrm{Bad}_H)$;
2: $P_1 \leftarrow true$;
3: **repeat**
4: $\quad P_2 \leftarrow P_1$;
5: $\quad P_1 \leftarrow CR(\boldsymbol{\mathcal{G}}_H, \neg TR(\boldsymbol{\mathcal{G}}_H, \neg P_2 \vee P_{bad_H}, \Sigma_{hu} \cup \Sigma_A))$;
6: **until** $P_1 = P_2$
7: return P_1;

Line 5 computes $\Gamma_H(P_2)$, because $CR(\boldsymbol{\mathcal{G}}_H, \neg TR(\boldsymbol{\mathcal{G}}_H, \neg P_2 \vee P_{bad_H}, \Sigma_{hu} \cup \Sigma_A)) = CR(\boldsymbol{\mathcal{G}}_H, \mathrm{PHIC}(P_2)) = \Gamma_H(P_2)$.

4.3 Low-level Supervisor Synthesis

In this section, we show how to synthesize a maximally permissible proper low-level supervisor for the j^{th} low-level in system $\mathbf{\Phi}, j \in \{1, \ldots, n\}$. Unlike the high-level supervisor synthesis, the low-level supervisor synthesis is much more complicated. We first only discuss part of the conditions for the j^{th} low-level.

4.3.1 The j^{th} Low-level P4 Interface Controllable Language

Definition 4.10. Let $K \subseteq \Sigma^*_{IL_j}$ be a language. K is *j^{th} low-level P4 interface controllable (*LPC_j*) with respect to system* $\mathbf{\Phi}$ if the following conditions are satisfied:

1. $\overline{K}\Sigma_{lu_j} \cap L(\mathbf{G}^p_{L_j}) \subseteq \overline{K}$
2. $\overline{K}\Sigma_{R_j} \cap L(\mathbf{G}^l_{I_j}) \subseteq \overline{K}$

$\diamondsuit$

Clearly, the empty language $\emptyset$ is j^{th} low-level P4 interface controllable and the definition is based on the language $\overline{K}$, so K is j^{th} low-level P4 interface controllable with respect to system $\mathbf{\Phi}$ iff $\overline{K}$ is j^{th} low-level P4 interface controllable with respect to system $\mathbf{\Phi}$.

The low-level P4 interface controllable language definition is quite simple. Notice that the first condition here is identical to the first condition in the the high-level interface controllable language definition(Definition 4.2) by substituting Σ_{IH} with Σ_{IL_j}, Σ_{hu} with Σ_{lu_j}, $L(\mathbf{G}^p_H \times \mathbf{G}^h_I)$ with $L(\mathbf{G}^p_{L_j})$. The second condition in Definition 4.2 is actually composed of n sub-conditions. The second condition here is identical to the j^{th} sub-conditions of the second condition in Definition 4.2 by substituting Σ_{IH} with Σ_{IL_j}, Σ_{A_j} with Σ_{R_j}, $L(\mathbf{G}^h_{I_j})$ with $L(\mathbf{G}^l_{I_j})$. Also notice that all the sub-conditions in the second condition of Definition 4.2 are independent. Therefore, the

following propositions or lemmas are provided without detailed proofs, but with the corresponding propositions or lemmas in Section 4.2. The detailed proofs can be easily obtained by appropriate substitutions from the corresponding propositions or lemmas.

For an arbitrary language $E \subseteq \Sigma^*_{IL_j}$, we define the set of all sublanguages of E that are j^{th} low-level P4 interface controllable with respect to system $\mathbf{\Phi}$ as

$\mathcal{LPC}_j(E) := \{K \subseteq E \mid K$ is j^{th} low-level
P4 interface controllable with respect to system $\mathbf{\Phi}$. $\}$

Clearly, $(\mathcal{LPC}_j(E), \subseteq)$ is a poset. The following proposition shows that the supremum in this poset always exists in $\mathcal{LPC}_j(E)$.

Proposition 4.12. *For system* $\mathbf{\Phi}$, $\mathcal{LPC}_j(E)$ *is nonempty and is closed under arbitrary unions. In particular,* $\mathcal{LPC}_j(E)$ *contains a (unique) supremal element,* $\sup \mathcal{LPC}_j(E) = \cup\{K | K \in \mathcal{LPC}_j(E)\}$.

proof: Similar to Proposition 4.1.

□

From Proposition 4.12, we have $\sup \mathcal{LPC}_j(E) \subseteq E$, which means that we can compute $\sup \mathcal{LPC}_j(E)$ by removing strings from E.

Lemma 4.8. *For system* $\mathbf{\Phi}$, *let* $L \subseteq \Sigma^*_{IL_j}$. *If* L *is closed then* $\sup \mathcal{LPC}_j(L)$ *is also closed.*

proof: Similar to Lemma 4.1.

□

Let $\mathcal{P}_{la_j} \in Pred(Q_{L_j})$ be a given predicate, we now give our method to compute $\sup \mathcal{LPC}_j(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j}))$.

Definition 4.11. For system $\mathbf{\Phi}$, let $\mathcal{P}_{la_j} \in Pred(Q_{L_j})$ be a given predicate. Define the function $\Omega_{LPC_j} : Pwr(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j})) \to Pwr(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j}))$ according to

$$(\forall K \in Pwr(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j})))$$
$$\Omega_{LPC_j}(K) := \{s \in K \mid \overline{\{s\}}\Sigma_{lu_j} \cap L(\mathbf{G}^p_{L_j}) \subseteq \overline{K} \;\; \& \;\; (\overline{\{s\}}\Sigma_{R_j} \cap L(\mathbf{G}^l_{I_j}) \subseteq \overline{K})\}$$

$\diamondsuit$

Clearly Ω_{LPC_j} is monotone. Note that $\Omega_{LPC_j}(K) \subseteq K \subseteq L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j})$, so this function is well-defined, and it only removes strings from K.

To make it easier to understand and use this function, the definition can be rewritten as

$$(\forall K \in Pwr(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j})))$$
$$\Omega_{LPC_j}(K) := \{s \in K \mid (\forall t \leq s)((\forall \sigma_u \in \Sigma_{lu_j})\ t\sigma_u \in L(\mathbf{G}^p_{L_j}) \Rightarrow t\sigma_u \in \overline{K}) \;\&$$
$$((\forall \sigma_{r_j} \in \Sigma_{R_j})\ t\sigma_{r_j} \in L(\mathbf{G}^l_{I_j}) \Rightarrow t\sigma_{r_j} \in \overline{K})\}.$$

Proposition 4.13. *For system* $\mathbf{\Phi}$, *let* $\mathcal{P}_{la_j} \in Pred(Q_{L_j})$ *be the given predicate for* Ω_{LPC_j}. *Then* $\sup \mathcal{LPC}_j(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j}))$ *is the greatest fixpoint of* Ω_{LPC_j} *with respect to* $(Pwr(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j})), \subseteq)$.

proof: Similar to Proposition 4.4.

□

From Proposition 4.13, we know that we can compute the $\sup \mathcal{LPC}_j(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j}))$ by computing the greatest fixpoint of Ω_{LPC_j}. It is also tempting to compute the greatest fixpoint by iteration of Ω_{LPC_j}.

Proposition 4.14. *For system* $\mathbf{\Phi}$, *let* $\mathcal{P}_{la_j} \in Pred(Q_{L_j})$ *be the given predicate for the function* Ω_{LPC_j}. *The set theoretic limit* $\lim_{i\to\infty} \Omega^i_{LPC_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j})), i \in \{1, 2, ...\}$ *exists, and* $\sup \mathcal{LPC}_j(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j})) \subseteq \lim_{i\to\infty} \Omega^i_{LPC_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j}))$.

proof: Similar to Proposition 4.5.

□

It remains to show that $\lim_{i\to\infty} \Omega^i_{LPC_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j})) \subseteq \sup \mathcal{LPC}_j(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j}))$ and that we only need to iterate a finite number of iterations in the case that $\boldsymbol{\mathcal{G}}_{L_j}$ has a finite number of states.

Definition 4.12. For system $\boldsymbol{\Phi}$, define the function $W_{L_j} : Pred(Q_{L_j}) \to Pwr(\Sigma^*_{IL_j})$ according to

$(\forall P \in Pred(Q_{L_j}))$

$$W_{L_j}(P) := \big\{ s \in L(\boldsymbol{\mathcal{G}}_{L_j}, P) \mid \big((\exists \sigma_u \in \Sigma_{lu_j})\ s\sigma_u \in L(\mathbf{G}^p_{L_j})\ \&\ s\sigma_u \notin L(\boldsymbol{\mathcal{G}}_{L_j}, P)\big)\ or \\ \big((\exists \sigma_{r_j} \in \Sigma_{R_j})\ s\sigma_{r_j} \in L(\mathbf{G}^l_{I_j})\ \&\ s\sigma_{r_j} \notin L(\boldsymbol{\mathcal{G}}_{L_j}, P)\big)\big\}$$

♢

Note that $W_{L_j}(P) \subseteq L(\boldsymbol{\mathcal{G}}_{L_j}, P)$. The following two lemmas will be used in the proof of the next proposition.

Lemma 4.9. *For system* $\boldsymbol{\Phi}$, *the following holds:*

$(\forall P \in Pred(Q_{L_j}))(\forall s, t \in L(\boldsymbol{\mathcal{G}}_{L_j}, P))$

$$s \in W_{L_j}(P)\ \&\ (\delta_{L_j}(q_{Lj_0}, s) = \delta_{L_j}(q_{Lj_0}, t)) \Rightarrow t \in W_{L_j}(P),$$

where q_{Lj_0} *is the initial state of* $\boldsymbol{\mathcal{G}}_{L_j}$ *as defined in section 4.1.*

proof: Similar to Lemma 4.2.

□

Lemma 4.10. *For system* $\boldsymbol{\Phi}$, *let* $\mathcal{P}_{la_j} \in Pred(Q_{L_j})$ *be the given predicate for* Ω_{LPC_j}. *Then the following holds:*

$(\forall P \in Sub(\mathcal{P}_{la_j}))(\forall s \in L(\boldsymbol{\mathcal{G}}_{L_j}, P))$

$$((\exists t \leq s)\ t \in W_{L_j}(P)) \Leftrightarrow s \notin \Omega_{LPC_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)).$$

proof: Similar to Lemma 4.3.

□

The above two lemmas state a very important fact. If a string $s \in L(\boldsymbol{\mathcal{G}}_{L_j}, P)$ is in $W_{L_j}(P)$ with $P \preceq \mathcal{P}_{la_j}$, Lemma 4.9 says that all strings in $L(\boldsymbol{\mathcal{G}}_{L_j}, P)$ that go from q_{Lj_0} to the state $\delta_{L_j}(q_{Lj_0}, s)$ are also in $W_{L_j}(P)$, while Lemma 4.10 says that all strings in $L(\boldsymbol{\mathcal{G}}_{L_j}, P)$ that are extended from a string in $W_{L_j}(P)$ are not in $\Omega_{LPC_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$. It means that if a string $s \in L(\boldsymbol{\mathcal{G}}_{L_j}, P)$ is in $W_{L_j}(P)$, then all the strings in $L(\boldsymbol{\mathcal{G}}_{L_j}, P))$ that lead from q_{Lj_0} to or pass through the state $\delta_{L_j}(q_{Lj_0}, s)$ are not in $\Omega_{LPC_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$. The fact is formally proved as follows by using these two lemmas.

Proposition 4.15. *For system* $\boldsymbol{\Phi}$, *let* $\mathcal{P}_{la_j} \in Pred(Q_{L_j})$ *be the given predicate for* Ω_{LPC_j}, *then*

$$(\forall P \in Sub(\mathcal{P}_{la_j}))\ L(\boldsymbol{\mathcal{G}}_{L_j}, P) - \Omega_{LPC_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) = L^{W_{L_j}(P)}(\boldsymbol{\mathcal{G}}_{L_j}, P),$$

where $L^{W_{L_j}(P)}(\boldsymbol{\mathcal{G}}_{L_j}, P) = \cup_{s \in W_{L_j}(P)} L^s(\boldsymbol{\mathcal{G}}_{L_j}, P)$ *as defined in Section 2.5.3.*

proof: Similar to Proposition 4.6.

□

Now, if we look at Proposition 2.7, it seems that we can compute $\Omega_{LPC_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$ by removing all the states in $\{q \in Q_{L_j} | (\exists s \in W_{L_j}(P)) \delta_{L_j}(q_{Lj_0}, s) = q\}$ from the state set Q_{L_j} of $\boldsymbol{\mathcal{G}}_{L_j}$.

For any $q \in Q_{L_j}$, as $\boldsymbol{\mathcal{G}}_{L_j} = \mathbf{E}_{L_j} \times \mathbf{G}_{L_j} \times \mathbf{G}^l_{I_j}$, there must exist $z \in Z_{L_j}, y \in Y_{L_j}$ and $x \in X^l_j$ such that

$$q = (z, y, x) \tag{4.2}$$

Definition 4.13. For system $\boldsymbol{\Phi}$, let $\mathcal{P}_{la_j} \in Pred(Q_{L_j})$ be a given predicate. Define the function $\Pi_{LPC_j} : Sub(\mathcal{P}_{la_j}) \rightarrow Sub(\mathcal{P}_{la_j})$ according to

$(\forall P \in Sub(\mathcal{P}_{la_j}))$

$$\Pi_{LPC_j}(P) = P - pr(\mathrm{Bad}_{RL_j} \cup \{q \models R(\boldsymbol{\mathcal{G}}_{L_j}, P) \mid (\exists \sigma \in \Sigma_{lu_j} \cup \Sigma_{R_j})\ \delta_{L_j}(q,\sigma) \models \neg P\}),$$

where $\mathrm{Bad}_{RL_j} = \{q \models R(\boldsymbol{\mathcal{G}}_{L_j}, P) |$

$$((\exists \sigma_u \in \Sigma_{lu_j})\ \eta_{L_j}(y,\sigma_u)!\ \ \&\ \ \zeta_{L_j} \times \xi^l_j((z,x),\sigma_u)\ \not!)\ \textit{or}$$

$$((\exists \sigma_{r_j} \in \Sigma_{R_j})\ \xi^l_j(x,\sigma_{r_j})!\ \&\ \zeta_{L_j} \times \eta_{L_j}((z,y),\sigma_{r_j})\ \not!)\}$$

and $q = (z,y,x)$ as in equation (4.2).

$\diamondsuit$

Note that $\Pi_{LPC_j}(P) \preceq P \preceq \mathcal{P}_{la_j}$, so the function is well-defined. Let $P \in Pred(Q_{L_j})$. Function Π_{LPC_j} removes all the states in $\{q \in Q_{L_j} | (\exists s \in W_{L_j}(P))\delta_{L_j}(q_{Lj_0}, s) = q\}$ from P. The following proposition shows this.

Proposition 4.16. *For system* $\boldsymbol{\Phi}$, *let* $\mathcal{P}_{la_j} \in Pred(Q_{L_j})$ *be the given predicate for* Ω_{LPC_j}, *then*

1. $(\forall P \in Sub(\mathcal{P}_{la_j}))\ \Omega_{LPC_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) = L(\boldsymbol{\mathcal{G}}_{L_j}, \Pi_{LPC_j}(P))$

2. $(\forall i \in \{0,1,\ldots\})\ \Omega^i_{LPC_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j})) = L(\boldsymbol{\mathcal{G}}_{L_j}, \Pi^i_{LPC_j}(\mathcal{P}_{la_j}))$

proof: Similar to Proposition 4.7.

□

Now, if we take a look at the definition of the function Π_{LPC_j}, we can see that for any $P \preceq \mathcal{P}_{la_j}$, the function Π_{LPC_j} only removes states in $R(\boldsymbol{\mathcal{G}}_{L_j}, P)$ from P. Precisely, Π_{LPC_j} only removes all the states in $pr(\mathrm{Bad}_{RL_j} \cup \{q \models R(\boldsymbol{\mathcal{G}}_{L_j}, P) \mid (\exists \sigma \in \Sigma_{lu_j} \cup \Sigma_{R_j})\ \delta_{L_j}(q,\sigma) \models \neg P\})$, which is a subpredicate of $R(\boldsymbol{\mathcal{G}}_{L_j}, P)$. If we also remove any state that is not in $R(\boldsymbol{\mathcal{G}}_{L_j}, P)$, then the language $L(\boldsymbol{\mathcal{G}}_{L_j}, P)$ is not affected at all, because the $L(\boldsymbol{\mathcal{G}}_{L_j}, P)$ is only dependent on $R(\boldsymbol{\mathcal{G}}_{L_j}, P)$. Similarly, removing any state that is not in $R(\boldsymbol{\mathcal{G}}_{L_j}, P)$ does not affect the language $L(\boldsymbol{\mathcal{G}}_{L_j}, \Pi_{LPC_j}(P))$ either,

because it is only dependent on a subpredicate of $R(\boldsymbol{\mathcal{G}}_{L_j}, P)$. The following function Γ_{LPC_j} is a replacement of the function Π_{LPC_j}.

Definition 4.14. For system $\boldsymbol{\Phi}$, let $\mathcal{P}_{la_j} \in Pred(Q_{L_j})$ be a given predicate. Define the function $\Gamma_{LPC_j} : Sub(\mathcal{P}_{la_j}) \to Sub(\mathcal{P}_{la_j})$ according to

$(\forall P \in Sub(\mathcal{P}_{la_j}))$

$\Gamma_{LPC_j}(P) = P - pr(\mathrm{Bad}_{L_j} \cup \{q \models Q_{L_j} \mid (\exists \sigma \in \Sigma_{lu_j} \cup \Sigma_{R_j})\ \delta_{L_j}(q, \sigma) \models \neg P\})$,

where $\mathrm{Bad}_{L_j} = \{q \in Q_{L_j} |$

$((\exists \sigma_u \in \Sigma_{lu_j})\ \eta_{L_j}(y, \sigma_u)!\ \ \&\ \ \zeta_{L_j} \times \xi^l_j((z, x), \sigma_u)\ \not!)$ *or*

$((\exists \sigma_{r_j} \in \Sigma_{R_j})\ \xi^l_j(x, \sigma_{r_j})!\ \&\ \zeta_{L_j} \times \eta_{L_j}((z, y), \sigma_{r_j})\ \not!)\}$

and $q = (z, y, x)$ as in equation (4.2).

♢

Note that for system $\boldsymbol{\Phi}$, Bad_{L_j} is constant.

Corollary 4.3. *For system* $\boldsymbol{\Phi}$, *let* $\mathcal{P}_{la_j} \in Pred(Q_{L_j})$ *be the given predicate for the function* Ω_{LPC_j}, Π_{LPC_j} *and* Γ_{LPC_j}. *Then the following holds:*

1. $(\forall P \in Sub(\mathcal{P}_{la_j}))\ \Omega_{LPC_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) = L(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{LPC_j}(P))$
2. $(\forall i \in \{0, 1, \ldots\})\ \Omega^i_{LPC_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j})) = L(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^i_{LPC_j}(\mathcal{P}_{la_j}))$

proof: Similar to Corollary 4.1.

□

The following lemma will be used in the next proposition.

Lemma 4.11. *For system* $\boldsymbol{\Phi}$, *let* $\mathcal{P}_{la_j} \in Pred(Q_{L_j})$ *be the given predicate for the function* Γ_{LPC_j}. *The function* Γ_{LPC_j} *is monotone with respect to* $\preceq$, *i.e.*

$$(\forall P_1, P_2 \in Sub(\mathcal{P}_{la_j}))\ P_1 \preceq P_2 \Rightarrow \Gamma_{LPC_j}(P_1) \preceq \Gamma_{LPC_j}(P_2)$$

proof: Similar to Lemma 4.4.

□

Proposition 4.17. *For system* $\mathbf{\Phi}$*, let* $\mathcal{P}_{la_j} \in Pred(Q_{L_j})$ *be the given predicate for the function* Ω_{LPC_j} *and* Γ_{LPC_j}*, then,*

1. *There exists* $k \in \{0, 1, 2, \ldots\}$ *with* $k \leq |st(\mathcal{P}_{la_j})|$ *such that* $\Gamma^k_{LPC_j}(\mathcal{P}_{la_j})$ *is the greatest fixpoint of the function* Γ_{LPC_j} *with respect to* $(Sub(\mathcal{P}_{la_j}), \preceq)$*, and where* $st(\mathcal{P}_{la_j})$ *is the identifying state subset of* Q_{L_j} *for* $\mathcal{P}_{la_j}$*.*

2. $\sup \mathcal{LPC}_j(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j})) = L(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{LPC_j}(\mathcal{P}_{la_j}))$

proof: Similar to Proposition 4.8.

□

Now, we are able to write a computer program to compute $\sup \mathcal{LPC}_j(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j}))$ by computing $\Gamma^k_{LPC_j}(\mathcal{P}_{la_j})$ ($k \in \{0, 1, 2, \ldots\}$), and we know that after at most $|st(\mathcal{P}_{la_j})|$ number of iterations, a fixpoint of Γ_{LPC_j} will be reached, which is the greatest fixpoint of Γ_{LPC_j} by Proposition 2.3. However, such a program based on function Γ_{LPC_j} is not efficient enough. For instance, each time when we implement the Γ_{LPC_j} function, we have to do the predicate subtracting operation. The following alternative method can avoid this.

Definition 4.15. For system $\mathbf{\Phi}$, define the function $\mathrm{PLPC}_j : Pred(Q_{L_j}) \to Pred(Q_{L_j})$ according to

$$(\forall P \in Pred(Q_{L_j}))\ \mathrm{PLPC}_j(P) := \neg TR(\boldsymbol{\mathcal{G}}_{L_j}, \neg P \vee pr(\mathrm{Bad}_{L_j}), \Sigma_{lu_j} \cup \Sigma_{R_j}).$$

♢

Lemma 4.12. *For system* $\mathbf{\Phi}$*, the following holds:*

1. *The function* PLPC_j *is monotone, i.e.*

   $(\forall P_1, P_2 \in Pred(Q_{L_j}))\ P_1 \preceq P_2 \Rightarrow \mathrm{PLPC}_j(P_1) \preceq \mathrm{PLPC}_j(P_2).$

2. $(\forall P \in Pred(Q_{L_j}))\ \mathrm{PLPC}_j(P) \preceq P$.

proof: Similar to Lemma 4.5.

□

Proposition 4.18. *For system* $\mathbf{\Phi}$, *the following holds:*

$$(\forall \mathcal{P}_{la_j} \in Pred(Q_{L_j}))\ \sup \mathcal{LPC}_j(L(\boldsymbol{\mathcal{G}}_{L_j}, \mathcal{P}_{la_j})) = L(\boldsymbol{\mathcal{G}}_{L_j}, \mathrm{PLPC}_j(\mathcal{P}_{la_j})).$$

proof: Similar to Proposition 4.9.

□

The following lemma is obvious, but for later convenience, we give a proof here.

Lemma 4.13. *For system* $\mathbf{\Phi}$, *the following holds:*

$$(\forall P \in Pred(Q_{L_j}))(\forall q \in Q_{L_j})\ q \models \mathrm{PLPC}_j(P) \Rightarrow ((\forall \rho \in \Sigma_{R_j})\ \xi_j^l(x,\rho)! \Rightarrow \delta_{L_j}(q,\rho)!)$$

proof:

Let $P \in Pred(Q_{L_j})$ and $q \in Q_{L_j}$.

Assume $q \models \mathrm{PLPC}_j(P)$. (1)

Must show this implies $(\forall \rho \in \Sigma_{R_j})\ \xi_j^l(x,\rho)! \Rightarrow \delta_{L_j}(q,\rho)!$.

Let $\rho \in \Sigma_{R_j}$. Assume $\xi_j^l(x,\rho)!$. We now show $\delta_{L_j}(q,\rho)!$.

From (1), we have $q \models \mathrm{PLPC}_j(P)$

$\Rightarrow q \models \neg TR(\boldsymbol{\mathcal{G}}_{L_j}, \neg P \vee pr(\mathrm{Bad}_{L_j}), \Sigma_{lu_j} \cup \Sigma_{R_j})$

$\Rightarrow q \not\models TR(\boldsymbol{\mathcal{G}}_{L_j}, \neg P \vee pr(\mathrm{Bad}_{L_j}), \Sigma_{lu_j} \cup \Sigma_{R_j})$

$\Rightarrow q \not\models pr(\mathrm{Bad}_{L_j}),$ by definition of TR

$\Rightarrow \zeta_{L_j} \times \eta_{L_j}((z,y),\rho)!,$ by definition of Bad_{L_j} and $q = (z,y,x)$ and $\xi_j^l(x,\rho)!$

$\Rightarrow \delta_{L_j}(q,\rho)!,$ by $\delta_{L_j} = \zeta_{L_j} \times \eta_{L_j} \times \xi_j^l$ and $\xi_j^l(x,\rho)!$.

□

4.3.2 The j^{th} Low-level Interface Controllable Language

Definition 4.16. Let $K \subseteq \Sigma^*_{IL_j}$. K is j^{th} *low-level interface controllable (*LIC_j*) with respect to system* $\mathbf{\Phi}$ if the following conditions are satisfied:

1. K is j^{th} low-level P4 interface controllable.

2. $(\forall s \in \overline{K})(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\ s\rho\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ s\rho l\alpha \in \overline{K}$

3. $(\forall s \in \overline{K})\ \ s \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ sl \in K$

$\diamondsuit$

Obviously, the empty language $\emptyset$ is j^{th} low-level interface controllable with respect to system $\mathbf{\Phi}$. Note that the first and the second conditions are based on the closed language $\overline{K}$.

For a language $K \subseteq \Sigma^*_{IL_j}$, if K satisfies the second condition of LIC_j, we say that K is j^{th} *low-level P5-satisfied*, and if K satisfies the third condition of LIC_j, we say that K is j^{th} *low-level P6-satisfied.*

For an arbitrary language $E \subseteq \Sigma^*_{IL_j}$, we define the set of all sublanguages of E that are j^{th} low-level interface controllable with respect to system $\mathbf{\Phi}$ as

$$\mathcal{C}_{L_j}(E) := \{K \subseteq E | K \text{ is } j^{th} \text{ low-level interface controllable with respect to system } \mathbf{\Phi}\}$$

Clearly, $(\mathcal{C}_{L_j}(E), \subseteq)$ is a poset. We now show that the supremum always exists in $\mathcal{C}_{L_j}(E)$.

Proposition 4.19. *Let* $E \subseteq \Sigma^*_{IL_j}$. *The set* $\mathcal{C}_{L_j}(E)$ *is nonempty and is closed under arbitrary unions. In particular,* $\mathcal{C}_{L_j}(E)$ *contains a (unique) supremal element,* $\sup \mathcal{C}_{L_j}(E) = \cup\{K | K \in \mathcal{C}_{L_j}(E)\}$.

proof:

1. Show that $\mathcal{C}_{L_j}(E)$ is nonempty

   The empty language $\emptyset$ is j^{th} low-level interface controllable with respect to system $\mathbf{\Phi}$, so $\emptyset \in \mathcal{C}_{L_j}(E)$.

2. Show that $\mathcal{C}_{L_j}(E)$ is closed under arbitrary unions.

   Let B be an index set. Assume that $(\forall \beta \in B)\ K_\beta \in \mathcal{C}_{L_j}(E)$. Sufficient to show that $\cup_{\beta \in B} K_\beta \in \mathcal{C}_{L_j}(E)$.

   Let $K := \cup_{\beta \in B} K_\beta \in \mathcal{C}_{L_j}(E)$. It is sufficient to show that K is j^{th} low-level interface controllable with respect to system $\mathbf{\Phi}$. By Definition 4.16, we have to show the following:

   (a) Show that $\overline{K}\Sigma_{lu_j} \cap L(\mathbf{G}^p_{L_j}) \subseteq \overline{K}$

   Identical to the proof of Proposition 4.1(Part 2(a)) by substituting Σ_{hu} with Σ_{lu_j}, $L(\mathbf{G}^p_H \times \mathbf{G}^h_I)$ with $L(\mathbf{G}^p_{L_j})$ and $\mathcal{C}_H$ with $\mathcal{C}_{L_j}$.

   (b) Show that $\overline{K}\Sigma_{R_j} \cap L(\mathbf{G}^l_{I_j}) \subseteq \overline{K}$

   Identical to the proof of Proposition 4.1 (Part 2(a)) by substituting Σ_{hu} with Σ_{R_j}, $L(\mathbf{G}^p_H \times \mathbf{G}^h_I)$ with $L(\mathbf{G}^l_{I_j})$ and $\mathcal{C}_H$ with $\mathcal{C}_{L_j}$.

   (c) Show that $(\forall s \in \overline{K})(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\ s\rho\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ s\rho l\alpha \in \overline{K}$.

   Let $s \in \overline{K}, \rho \in \Sigma_{R_j}, \alpha \in \Sigma_{A_j}$. (1)

   Assume $s\rho\alpha \in L(\mathbf{G}^l_{I_j})$. (2)

   Must show this implies $(\exists l \in \Sigma^*_{L_j})\ s\rho l\alpha \in \overline{K}$.

   By (1), we have $s \in \overline{K}$

   $\Rightarrow (\exists s' \in \Sigma^*_{IL_j})\ ss' \in K$

$\Rightarrow (\exists \beta \in B)\ ss' \in K_\beta$

$\Rightarrow s \in \overline{K_\beta}$ (3)

By (3), (2) and $K_\beta \in \mathcal{C}_{L_j}(E)$, we have $(\exists l \in \Sigma^*_{L_j})\ s\rho l\alpha \in \overline{K_\beta}$

$\Rightarrow (\exists l \in \Sigma^*_{L_j})\ s\rho l\alpha \in \overline{K}, \quad$ as $K_\beta \subseteq K$

(d) Show that $(\forall s \in \overline{K})\ s \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j}) sl \in K$

Let $s \in \overline{K}$. (4)

Assume $s \in L_m(\mathbf{G}^l_{I_j})$. (5)

Must show this implies $(\exists l \in \Sigma^*_{L_j})\ sl \in K$.

By (4), we have $s \in \overline{K}$

$\Rightarrow (\exists s' \in \Sigma^*_{IL_j})\ ss' \in K$

$\Rightarrow (\exists \beta \in B)\ ss' \in K_\beta$

$\Rightarrow s \in \overline{K_\beta}$ (6)

By (6), (5) and $K_\beta \in \mathcal{C}_{L_j}(E)$, we have $(\exists l \in \Sigma^*_{L_j})\ sl \in K_\beta$

$\Rightarrow (\exists l \in \Sigma^*_{L_j})\ sl \in K, \quad$ as $K_\beta \subseteq K$

3. Show that $\sup \mathcal{C}_{L_j}(E) = \cup\{K | K \in \mathcal{C}_{L_j}(E)\}$.

   Identical to the proof of Proposition 4.1 (Part 3) by substituting $\mathcal{C}_H$ with $\mathcal{C}_{L_j}$ and Σ_{IH} with Σ_{IL_j}.

4. Show that $\sup \mathcal{C}_{L_j}(E) \in \mathcal{C}_{L_j}(E)$.

   This immediately follows from Part 2 and Part 3.

□

From Proposition 4.19, we have $\sup \mathcal{C}_{L_j}(E) \subseteq E$, which means that we can compute $\sup \mathcal{C}_{L_j}(E)$ by removing strings from E.

4.3.3 $\sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}))$ and the Greatest Fixpoint of Ω_{L_j}

For system $\mathbf{\Phi}$, if we compute the supremal j^{th} low-level interface controllable sublanguage of $L_m(\boldsymbol{\mathcal{G}}_{L_j})$, then a DES representing this sublanguage is a maximally permissible j^{th} proper low-level supervisor. Therefore, we need to find a method to compute $\sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}))$. For this purpose, we define the following function.

Definition 4.17. For system $\mathbf{\Phi}$, define the function $\Omega_{L_j} : Pwr(\Sigma^*_{IL_j}) \to Pwr(\Sigma^*_{IL_j})$ according to

$$(\forall K \in Pwr(\Sigma^*_{IL_j}))\ \Omega_{L_j}(K) := \Omega_{LNB_j}(\Omega_{p6_j}(\Omega_{p5_j}(\mathrm{LPC}_j(K))))$$

where $\Omega_{LNB_j}, \Omega_{p6_j}, \Omega_{p5_j}, \mathrm{LPC}_j$ are functions for system $\mathbf{\Phi}$, which are defined as follows:

- $\mathrm{LPC}_j : Pwr(\Sigma^*_{IL_j}) \to Pwr(\Sigma^*_{IL_j})$

  $$(\forall K \in Pwr(\Sigma^*_{IL_j}))\ \mathrm{LPC}_j(K) := \sup \mathcal{LPC}_j(\overline{K})$$

- $\Omega_{p5_j} : Pwr(\Sigma^*_{IL_j}) \to Pwr(\Sigma^*_{IL_j})$

  $(\forall K \in Pwr(\Sigma^*_{IL_j}))$

  $$\Omega_{p5_j}(K) := \{s \in K \mid (\forall t \leq s)(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j}) \\ t\rho\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ t\rho l\alpha \in K\}$$

- $\Omega_{p6_j} : Pwr(\Sigma^*_{IL_j}) \to Pwr(\Sigma^*_{IL_j})$

  $(\forall K \in Pwr(\Sigma^*_{IL_j}))$

  $$\Omega_{p6_j}(K) := \{s \in K | (\forall t \leq s)\ t \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j}) tl \in L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap K\}$$

- $\Omega_{LNB_j} : Pwr(\Sigma^*_{IL_j}) \to Pwr(\Sigma^*_{IL_j})$

  $$(\forall K \in Pwr(\Sigma^*_{IL_j}))\ \Omega_{LNB_j}(K) := L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap K$$

$\diamondsuit$

As all the functions in Ω_{L_j} are defined on $Pwr(\Sigma^*_{IL_j})$, so we can compose them together.

Clearly, $\Omega_{p5_j}, \Omega_{p6_j}, \Omega_{LNB_j}$ are monotone with respect to $\subseteq$. By Proposition 4.12, it is also clear that LPC$_j$ is monotone with respect to $\subseteq$. Therefore, the function Ω_{L_j} is monotone with respect to $\subseteq$ as well.

Note that $(\forall K \in Pwr(\Sigma^*_{IL_j}))$ LPC$_j(K) \subseteq \overline{K}, \Omega_{p5_j}(K) \subseteq K, \Omega_{p6_j}(K) \subseteq K$ and $\Omega_{LNB_j}(K) \subseteq K$.

By the Knaster-Tarski Theorem(Theorem 2.1), the greatest fixpoint of the function Ω_{L_j} with respect to $(Pwr(\Sigma^*_{IL_j}), \subseteq)$ must exist. The following lemmas will be used later on.

Lemma 4.14. *For system* $\mathbf{\Phi}$, *let* $L \subseteq \Sigma^*_{IL_j}$ *be a closed language, then* $\Omega_{p5_j}(L)$ *and* $\Omega_{p6_j}(L)$ *are also closed.*

proof:

Assume $L = \overline{L}$. Must show implies $\overline{\Omega_{p5_j}(L)} = \Omega_{p5_j}(L)$ and $\overline{\Omega_{p6_j}(L)} = \Omega_{p6_j}(L)$.

$\Omega_{p5_j}(L) \subseteq \overline{\Omega_{p5_j}(L)}$ and $\Omega_{p6_j}(L) \subseteq \overline{\Omega_{p6_j}(L)}$ are automatic.

1. Show that $\overline{\Omega_{p5_j}(L)} \subseteq \Omega_{p5_j}(L)$

   Let $s \in \overline{\Omega_{p5_j}(L)}$. (1)

   Must show this implies $s \in \Omega_{p5_j}(L)$.

   From (1), we have $(\exists s' \in \Sigma^*_{IL_j})\ ss' \in \Omega_{p5_j}(L)$

   $\Rightarrow ss' \in L$ & $((\forall t \leq ss')(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})$

   $$t\rho\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ t\rho l\alpha \in L)$$

   $\Rightarrow s \in L$ & $((\forall t \leq s)(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})$

   $$t\rho\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ t\rho l\alpha \in L), \quad \text{as } L \text{ is closed.}$$

   $\Rightarrow s \in \Omega_{p5_j}(L)$

2. Show that $\overline{\Omega_{p6_j}(L)} \subseteq \Omega_{p6_j}(L)$.

   Let $s \in \overline{\Omega_{p6_j}(L)}$. (2)

   Must show this implies $s \in \Omega_{p6_j}(L)$.

   From (2), we have $(\exists s' \in \Sigma^*_{IL_j})\ ss' \in \Omega_{p6_j}(L)$

   $\Rightarrow ss' \in L$ & $((\forall t \leq ss')\ t \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j}) tl \in L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L)$

   $\Rightarrow s \in L$ & $((\forall t \leq s)\ t \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j}) tl \in L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L)$,

   as L is closed.

   $\Rightarrow s \in \Omega_{p6_j}(L)$.

□

Lemma 4.15. *For system* $\mathbf{\Phi}$ *and for all* $T \in Pwr(\Sigma^*_{IL_j})$,

1. $\overline{T} = \sup \mathcal{LPC}_j(\overline{T})$ *iff* T *is* j^{th} *low-level P4 interface controllable.*

2. $\overline{T} = \Omega_{p5_j}(\overline{T})$ *iff* T *is* j^{th} *low-level P5-satisfied.*

proof:

Let $T \in Pwr(\Sigma^*_{IL_j})$.

1. Show that $\overline{T} = \sup \mathcal{LPC}_j(\overline{T})$ iff T is j^{th} low-level P4 interface controllable.

   We first show $\overline{T} = \sup \mathcal{LPC}_j(\overline{T})$ iff $\overline{T}$ is j^{th} low-level P4 interface controllable.

   (a) Show $(\overline{T} = \sup \mathcal{LPC}_j(\overline{T})) \Rightarrow (\overline{T}$ is j^{th} low-level P4 interface controllable).

   Assume $\overline{T} = \sup \mathcal{LPC}_j(\overline{T})$. (1)

   Must show this implies $\overline{T}$ is j^{th} low-level P4 interface controllable.

   Sufficient to show that $\overline{T} \in \mathcal{LPC}_j(\overline{T})$.

   By Proposition 4.12, we know $\sup \mathcal{LPC}_j(\overline{T}) = \cup\{K | K \in \mathcal{LPC}_j(\overline{T})\}$.

$\Rightarrow \overline{T} = \cup\{K | K \in \mathcal{LPC}_j(\overline{T})\}$, by (1)

$\Rightarrow \overline{T} \in \mathcal{LPC}_j(\overline{T})$, by Proposition 4.12

(b) Show ($\overline{T}$ is j^{th} low-level P4 interface controllable) $\Rightarrow$ ($\overline{T} = \sup \mathcal{LPC}_j(\overline{T})$).

Assume $\overline{T}$ is j^{th} low-level P4 interface controllable. (2)

Must show this implies $\overline{T} = \sup \mathcal{LPC}_j(\overline{T})$.

By (2), we know $\overline{T} \in \mathcal{LPC}_j(\overline{T})$

$\Rightarrow \overline{T} \subseteq \sup \mathcal{LPC}_j(\overline{T})$ (3)

By Proposition 4.12, we know $\sup \mathcal{LPC}_j(\overline{T}) \subseteq \overline{T}$ (4)

From (3) and (4), we have $\overline{T} = \sup \mathcal{LPC}_j(\overline{T})$.

From Definition 4.10, we know that T is j^{th} low-level P4 interface controllable iff $\overline{T}$ is j^{th} low-level P4 interface controllable.

So, we have $\overline{T} = \sup \mathcal{LPC}_j(\overline{T})$ iff T is j^{th} low-level P4 interface controllable.

2. Show that $\overline{T} = \Omega_{p5_j}(\overline{T})$ iff T is j^{th} low-level P5-satisfied.

T is j^{th} low-level P5-satisfied

$\Leftrightarrow (\forall s \in \overline{T})(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\ s\rho\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ s\rho l\alpha \in \overline{T}$

$\Leftrightarrow (\forall s \in \overline{T})(\forall t \leq s)(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})$

$t\rho\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ t\rho l\alpha \in \overline{T}$, as $\overline{T}$ is closed

$\Leftrightarrow (\forall s \in \overline{T})\ s \in \Omega_{p5_j}(\overline{T})$

$\Leftrightarrow \overline{T} \subseteq \Omega_{p5_j}(\overline{T})$

$\Leftrightarrow \overline{T} = \Omega_{p5_j}(\overline{T})$, as $\Omega_{p5_j}(\overline{T}) \subseteq \overline{T}$ by definition of Ω_{p5_j}.

□

Proposition 4.20. *For system* $\mathbf{\Phi}$, $\sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}))$ *is the greatest fixpoint of the function* Ω_{L_j} *with respect to* $(Pwr(\Sigma^*_{IL_j}), \subseteq)$.

proof:

Let $S := \sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}))$.

1. Show that S is a fixpoint of Ω_{L_j}, i.e. $S = \Omega_{L_j}(S)$

   (a) Show that $\overline{S} = \sup \mathcal{LPC}_j(\overline{S})$ and $\overline{S} = \Omega_{p5_j}(\overline{S})$.

   As S is j^{th} low-level interface controllable, S is j^{th} low-level P4 interface controllable and j^{th} low-level P5-satisfied.

   By Lemma 4.15, we have $\overline{S} = \sup \mathcal{LPC}_j(\overline{S})$ and $\overline{S} = \Omega_{p5_j}(\overline{S})$.

   (b) Show that $S = L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \overline{S}$

   Sufficient to show that $S \subseteq L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \overline{S}$ and $L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \overline{S} \subseteq S$.

   By Proposition 4.19, we have $S \subseteq L_m(\boldsymbol{\mathcal{G}}_{L_j})$. (1)

   $\Rightarrow S \subseteq L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \overline{S}$, as $S \subseteq \overline{S}$

   Now we show $L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \overline{S} \subseteq S$.

   Let $t \in L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \overline{S}$. (2)

   Must show this implies $t \in S$.

   From (2), we have $t \in L_m(\boldsymbol{\mathcal{G}}_{L_j})$ and $t \in \overline{S}$

   $\Rightarrow \{t\} \subseteq L_m(\boldsymbol{\mathcal{G}}_{L_j})$ and $\{t\} \subseteq \overline{S}$

   $\Rightarrow \{t\} \subseteq L_m(\boldsymbol{\mathcal{G}}_{L_j})$ and $\overline{\{t\}} \subseteq \overline{S}$ (3)

   Let $S' := S \cup \{t\}$.

   $\Rightarrow S' \subseteq L_m(\boldsymbol{\mathcal{G}}_{L_j})$, by (1) and (3) (4)

   We now show S' is j^{th} low-level interface controllable with respect to system $\mathbf{\Phi}$.

i. Show that $\overline{S'}\Sigma_{lu_j} \cap L(\mathbf{G}^p_{L_j}) \subseteq \overline{S'}$

S is j^{th} low-level interface controllable, so $\overline{S}\Sigma_{lu_j} \cap L(\mathbf{G}^p_{L_j}) \subseteq \overline{S}$. (5)

By (3) and (5), we have $\overline{\{t\}}\Sigma_{lu_j} \cap L(\mathbf{G}^p_{L_j}) \subseteq \overline{S}$ (6)

From (5) and (6), we have $(\overline{S} \cup \overline{\{t\}})\Sigma_{lu_j} \cap L(\mathbf{G}^p_{L_j}) \subseteq \overline{S}$

$\Rightarrow (\overline{S \cup \{t\}})\Sigma_{lu_j} \cap L(\mathbf{G}^p_{L_j}) \subseteq \overline{S}$, by Proposition 2.2

$\Rightarrow \overline{S'}\Sigma_{lu_j} \cap L(\mathbf{G}^p_{L_j}) \subseteq \overline{S}$

$\Rightarrow \overline{S'}\Sigma_{lu_j} \cap L(\mathbf{G}^p_{L_j}) \subseteq \overline{S'}$, as $S \subseteq S'$ by definition of S' (7)

ii. Show that $\overline{S'}\Sigma_{R_j} \cap L(\mathbf{G}^l_{I_j}) \subseteq \overline{S'}$

S is j^{th} low-level interface controllable, so $\overline{S}\Sigma_{R_j} \cap L(\mathbf{G}^l_{I_j}) \subseteq \overline{S}$. (8)

By (3) and (8), we have $\overline{\{t\}}\Sigma_{R_j} \cap L(\mathbf{G}^l_{I_j}) \subseteq \overline{S}$ (9)

From (8) and (9), we have $(\overline{S} \cup \overline{\{t\}})\Sigma_{R_j} \cap L(\mathbf{G}^l_{I_j}) \subseteq \overline{S}$

$\Rightarrow (\overline{S \cup \{t\}})\Sigma_{R_j} \cap L(\mathbf{G}^l_{I_j}) \subseteq \overline{S}$, by Proposition 2.2

$\Rightarrow \overline{S'}\Sigma_{R_j} \cap L(\mathbf{G}^l_{I_j}) \subseteq \overline{S}$

$\Rightarrow \overline{S'}\Sigma_{R_j} \cap L(\mathbf{G}^l_{I_j}) \subseteq \overline{S'}$, as $S \subseteq S'$ by definition of S' (10)

iii. Show that $(\forall s \in \overline{S'})(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\ s\rho\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ s\rho l\alpha \in \overline{S'}$

S is j^{th} low-level interface controllable, so

$(\forall s \in \overline{S})(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\ s\rho\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ s\rho l\alpha \in \overline{S}$ (11)

By (3) and (11), $(\forall s \in \overline{\{t\}})(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})$

$$s\rho\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ s\rho l\alpha \in \overline{S} \quad (12)$$

By (11) and (12), $(\forall s \in \overline{S} \cup \overline{\{t\}})(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})$

$$s\rho\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ s\rho l\alpha \in \overline{S}$$

$\Rightarrow (\forall s \in \overline{S \cup \{t\}})(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})$

$$s\rho\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ s\rho l\alpha \in \overline{S}, \quad \text{by Proposition 2.2}$$

$\Rightarrow (\forall s \in \overline{S'})(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})$

$$s\rho\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ s\rho l\alpha \in \overline{S'}. \quad (13)$$

iv. Show that $(\forall s \in \overline{S'})\ \ s \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j}) sl \in S'$

S is j^{th} low-level interface controllable, so

$(\forall s \in \overline{S})\ s \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ sl \in S.$ (14)

By (3) and (14), we have

$(\forall s \in \overline{\{t\}})\ s \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ sl \in S.$ (15)

By (14) and (15),we have

$(\forall s \in \overline{S} \cup \overline{\{t\}})\ s \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ sl \in S$

$\Rightarrow (\forall s \in \overline{S \cup \{t\}})\ s \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ sl \in S$, by Proposition 2.2

$\Rightarrow (\forall s \in \overline{S'})\ s \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ sl \in S'$ (16)

By (4), (7), (10), (13) and (16), we have $S' \in \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}))$.

As $S = \sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}))$, we then have $S' \subseteq S$.

$\Rightarrow \{t\} \subseteq S$

$\Rightarrow t \in S$

(c) Show that $\overline{S} = \Omega_{p6_j}(\overline{S})$

Sufficient to show that

$(\forall s \in \overline{S})(\forall t \leq s)\ t \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ tl \in L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \overline{S}$

Let $s \in \overline{S}$ and $t \leq s$. (17)

Assume $t \in L_m(\mathbf{G}^l_{I_j})$. Must show this implies $(\exists l \in \Sigma^*_{L_j})\ tl \in L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \overline{S}$

From (17) and the assumption $t \in L_m(\mathbf{G}^l_{I_j})$, by the definition of S, we have

$(\exists l \in \Sigma^*_{L_j})\ tl \in S$

$\Rightarrow (\exists l \in \Sigma^*_{L_j})\ tl \in L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \overline{S}$, by Part 1(b)

By combining (a), (b), (c) and the definition of Ω_{L_j}, we know $S = \Omega_{L_j}(S)$.

2. Show that S is the greatest fixpoint of Ω_{L_j}. i.e.

$$(\forall T \in Pwr(\Sigma^*_{IL_j}))\ T = \Omega_{L_j}(T) \Rightarrow T \subseteq S.$$

Let $T \in Pwr(\Sigma^*_{IL_j})$. Assume $T = \Omega_{L_j}(T)$. Must show this implies $T \subseteq S$.

Sufficient to show that T is j^{th} low-level controllable with respect to system $\mathbf{\Phi}$ and $T \subseteq L_m(\boldsymbol{\mathcal{G}}_{L_j})$.

$T = \Omega_{L_j}(T)$

$= \Omega_{LNB_j}(\Omega_{p6_j}(\Omega_{p5_j}(\mathrm{LPC}_j(T))))$

$= L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \Omega_{p6_j}(\Omega_{p5_j}(\sup \mathcal{LPC}_j(\overline{T})))$ (18)

$\subseteq \Omega_{p6_j}(\Omega_{p5_j}(\sup \mathcal{LPC}_j(\overline{T})))$

As $\overline{T}$ is closed, by Lemma 4.8, $\sup \mathcal{LPC}_j(\overline{T})$ is closed. Then by Lemma 4.14, $\Omega_{p6_j}(\Omega_{p5_j}(\sup \mathcal{LPC}_j(\overline{T})))$ is also closed. So by (18), we have

$$\overline{T} \subseteq \Omega_{p6_j}(\Omega_{p5_j}(\sup \mathcal{LPC}_j(\overline{T}))) \tag{19}$$

For any $L \in Pwr(\Sigma^*_{IL_j})$, we have $\sup \mathcal{LPC}_j(L) \subseteq L$ by Proposition 4.12, $\Omega_{p5_j}(L) \subseteq L$ by the definition of Ω_{p5_j} and $\Omega_{p6_j}(L) \subseteq L$ by the definition of Ω_{p6_j}, so we know

$$\Omega_{p6_j}(\Omega_{p5_j}(\sup \mathcal{LPC}_j(\overline{T}))) \subseteq \overline{T} \tag{20}$$

By (19) and (20), we have

$$\overline{T} = \Omega_{p6_j}(\Omega_{p5_j}(\sup \mathcal{LPC}_j(\overline{T}))) \tag{21}$$

Function Ω_{p6_j} and Ω_{p5_j} only remove strings from their inputs, and $\sup \mathcal{LPC}_j(\overline{T}) \subseteq \overline{T}$, so from (21) we have

$$\overline{T} = \sup \mathcal{LPC}_j(\overline{T}),\ \overline{T} = \Omega_{p5_j}(\overline{T}) \text{ and } \overline{T} = \Omega_{p6_j}(\overline{T}) \tag{22}$$

From $\overline{T} = \sup \mathcal{LPC}_j(\overline{T})$ and $\overline{T} = \Omega_{p5_j}(\overline{T})$, by Lemma 4.15, we know that T is j^{th} low-level P4 interface controllable and j^{th} low-level P5-satisfied.

It remains to show that T is j^{th} low-level P6-satisfied.

Sufficient to show that $(\forall u \in \overline{T})\ u \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ ul \in T$.

Let $u \in \overline{T}$. Assume $u \in L_m(\mathbf{G}^l_{I_j})$. Must show this implies $(\exists l \in \Sigma^*_{L_j})\ ul \in T$.

From (18) and (21), we also have $T = L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \overline{T}$ (23)

From (22), we have $\overline{T} = \Omega_{p6_j}(\overline{T})$

$\Rightarrow \overline{T} = \{s \in \overline{T} \mid (\forall t \leq s)\ t \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ tl \in L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \overline{T}\}$

$\Rightarrow u \in \{s \in \overline{T} \mid (\forall t \leq s)\ t \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ tl \in L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \overline{T}\}$, as $u \in \overline{T}$

$\Rightarrow (\forall t \leq u)\ (t \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ tl \in L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \overline{T})$

$\Rightarrow u \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ ul \in L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \overline{T}, \quad$ as $u \leq u$

$\Rightarrow (\exists l \in \Sigma^*_{L_j})\ ul \in L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \overline{T}, \quad$ by assumption $u \in L_m(\mathbf{G}^l_{I_j})$

$\Rightarrow (\exists l \in \Sigma^*_{L_j})\ ul \in T, \quad$ by (23)

From (18), clearly, $T \subseteq L_m(\boldsymbol{\mathcal{G}}_{L_j})$.

□

4.3.4 Computing the Greatest Fixpoint of Ω_{L_j}

From Proposition 4.20, we know that we can compute $\sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}))$ by computing the greatest fixpoint of Ω_{L_j}. It is tempting to compute the greatest fixpoint by iteration of Ω_{L_j}.

Proposition 4.21. *For system* $\mathbf{\Phi}$, *the set theoretic limit* $\lim_{i\to\infty}(L(\boldsymbol{\mathcal{G}}_{L_j})), i \in \{1, 2, \ldots\}$ *exists, and* $\sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j})) \subseteq \lim_{i\to\infty} \Omega^i_{L_j}(L(\boldsymbol{\mathcal{G}}_{L_j}))$.

proof:

Identical to the proof of Proposition 4.3 by substituting $\boldsymbol{\mathcal{G}}_H$ with $\boldsymbol{\mathcal{G}}_{L_j}$, $\sup \mathcal{C}_H$ with $\sup \mathcal{C}_{L_j}$, Ω_H with Ω_{L_j}, Proposition 4.1 with Proposition 4.19, and Proposition 4.2 with Proposition 4.20.

□

Similar to the high-level, here we start the iteration from the closed language $L(\boldsymbol{\mathcal{G}}_{L_j})$. It remains to show the reverse $\lim_{i\to\infty} \Omega^i_{L_j}(L(\boldsymbol{\mathcal{G}}_{L_j})) \subseteq \sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}))$ and that the limit will be reached after a finite number of iterations on Ω_{L_j}(i.e. $i < \infty$). Again, in general case, the reverse does not hold, but in our case where all the languages are regular, the reverse does hold and the number of iterations on Ω_{L_j} is finite, as we will show.

For a given predicate $P \in Pred(Q_{L_j})$, we already know the method to compute $\sup \mathcal{LPC}_j(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$ from Proposition 4.18. We now show the method to compute $\Omega_{p5_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$ and $\Omega_{p6_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$.

Computing $\Omega_{p5_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$

Definition 4.18. For system $\boldsymbol{\Phi}$, define the function $W_{p5_j} : Pred(Q_{L_j}) \to Pwr(\Sigma^*_{IL_j})$ according to

$$(\forall P \in Pred(Q_{L_j}))\ W_{p5_j}(P) := \{s \in L(\boldsymbol{\mathcal{G}}_{L_j}, P) \mid (\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j}) \\ s\rho\alpha \in L(\mathbf{G}^l_{I_j})\ \&\ (\forall l \in \Sigma^*_{L_j})\ s\rho l\alpha \notin L(\boldsymbol{\mathcal{G}}_{L_j}, P)\}$$

♢

Clearly, for all $P \in Pred(Q_{L_j}), W_{p5_j}(P) \subseteq L(\boldsymbol{\mathcal{G}}_{L_j}, P)$.

Lemma 4.16. *For system* $\boldsymbol{\Phi}$, *the following holds:*

$$(\forall P \in Pred(Q_{L_j}))(\forall s, t \in L(\boldsymbol{\mathcal{G}}_{L_j}, P)) \\ (s \in W_{p5_j}(P)\ \&\ (\delta_{L_j}(q_{Lj_0}, s) = \delta_{L_j}(q_{Lj_0}, t))) \Rightarrow t \in W_{p5_j}(P).$$

proof:

Let $P \in Pred(Q_{L_j})$, and $s, t \in L(\boldsymbol{\mathcal{G}}_{L_j}, P)$. (1)

Assume $\delta(q_{Lj_0}, s) = \delta(q_{Lj_0}, t)$ and $s \in W_{p5_j}(P)$. (2)

Must show this implies $t \in W_{p5_j}(P)$.

Let $\equiv_{L(\boldsymbol{\mathcal{G}}_{L_j},P)}, \equiv_{L(\mathbf{G}^l_{I_j})}$ be the Nerode equivalence relation on $\Sigma^*_{IL_j}$ with respect to $L(\boldsymbol{\mathcal{G}}_{L_j}, P)$ and $L(\mathbf{G}^l_{I_j})$ respectively.

From (2), we have $\delta(q_{Lj_0}, s) = \delta(q_{Lj_0}, t)$

$\Rightarrow s \equiv_{L(\boldsymbol{\mathcal{G}}_{L_j},P)} t,$ (3)

by Proposition 2.6

As $\boldsymbol{\mathcal{G}}_{L_j} = \mathbf{E}_{L_j} \times \mathbf{G}^p_{L_j} \times \mathbf{G}^l_{I_j}$ and $\delta_{L_j}(q_{Lj_0}, s) = \delta_{L_j}(q_{Lj_0}, t)$, we have

$\xi^l_j(x^l_{j0}, s) = \xi^l_j(x^l_{j0}, t)$

$\Rightarrow s \equiv_{L(\mathbf{G}^l_{I_j})} t,$ (4)

by Proposition 2.6 and fact $L(\mathbf{G}^l_{I_j}, true) = L(\mathbf{G}^l_{I_j})$.

From (2), we have $s \in W_{p5_j}(P)$

$\Rightarrow (\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j})\ s\rho\alpha \in L(\mathbf{G}^l_{I_j})\ \&\ (\forall l \in \Sigma^*_{L_j})\ s\rho l\alpha \notin L(\boldsymbol{\mathcal{G}}_{L_j}, P)$

$\Rightarrow (\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j})\ t\rho\alpha \in L(\mathbf{G}^l_{I_j})\ \&\ (\forall l \in \Sigma^*_{L_j})\ t\rho l\alpha \notin L(\boldsymbol{\mathcal{G}}_{L_j}, P)$, by (3)(4)

$\Rightarrow t \in W_{p5_j}(P), \quad$ as $t \in L(\boldsymbol{\mathcal{G}}_{L_j}, P)$.

□

Lemma 4.17. *For system* $\mathbf{\Phi}$, *the following holds:*

$(\forall P \in Pred(Q_{L_j}))(\forall s \in L(\boldsymbol{\mathcal{G}}_{L_j}, P))\ ((\exists t \leq s)\ t \in W_{p5_j}(P)) \Leftrightarrow s \notin \Omega_{p5_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)))$.

proof:

Let $P \in Pred(Q_{L_j})$ and $s \in L(\boldsymbol{\mathcal{G}}_{L_j}, P)$. (1)

1. Show that $((\exists t \leq s)\ t \in W_{p5_j}(P)) \Rightarrow s \notin \Omega_{p5_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$.

   Assume $(\exists t \leq s)\ t \in W_{p5_j}(P)$. (2)

Must show this implies $s \notin \Omega_{p5_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$.

From (2), we have

$(\exists t \leq s)(\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j})\ t\rho\alpha \in L(\mathbf{G}^l_{I_j})\ \&\ (\forall l \in \Sigma^*_{L_j})\ t\rho l\alpha \notin L(\boldsymbol{\mathcal{G}}_{L_j}, P)$

$\Rightarrow \neg((\forall t \leq s)(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})$

$$t\rho\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ t\rho l\alpha \in L(\boldsymbol{\mathcal{G}}_{L_j}, P))$$

$\Rightarrow s \notin \Omega_{p5_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$.

2. Show that $s \notin \Omega_{p5_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) \Rightarrow (\exists t \leq s)\ t \in W_{p5_j}(P)$.

   Assume $s \notin \Omega_{p5_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$. (3)

   Must show this implies $(\exists t \leq s)\ t \in W_{p5_j}(P)$.

   From (1), we know $s \in L(\boldsymbol{\mathcal{G}}_{L_j}, P)$. Combine this with (3), we thus have

   $\neg((\forall t \leq s)(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})$

   $$t\rho\alpha \in L(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ t\rho l\alpha \in L(\boldsymbol{\mathcal{G}}_{L_j}, P))$$

   $\Rightarrow (\exists t \leq s)(\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j})$

   $$t\rho\alpha \in L(\mathbf{G}^l_{I_j})\ \&\ (\forall l \in \Sigma^*_{L_j})\ t\rho l\alpha \notin L(\boldsymbol{\mathcal{G}}_{L_j}, P) \quad (4)$$

   As $s \in L(\boldsymbol{\mathcal{G}}_{L_j}, P)$ and $L(\boldsymbol{\mathcal{G}}_{L_j}, P)$ is closed, we know $t \in L(\boldsymbol{\mathcal{G}}_{L_j}, P)$. (5)

   By (4) and (5), we thus have $(\exists t \leq s)\ t \in W_{p5_j}(P)$.

□

Proposition 4.22. *For system* $\boldsymbol{\Phi}$*, the following holds*

$$(\forall P \in Pred(Q_{L_j}))\ L(\boldsymbol{\mathcal{G}}_{L_j}, P) - \Omega_{p5_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) = L^{W_{p5_j}(P)}(\boldsymbol{\mathcal{G}}_{L_j}, P),$$

where $L^{W_{p5_j}(P)}(\boldsymbol{\mathcal{G}}_{L_j}, P) = \cup_{s \in W_{p5_j}(P)} L^s(\boldsymbol{\mathcal{G}}_{L_j}, P)$.

proof:

Identical to the proof of Proposition 4.6 after substituting $Sub(\mathcal{P}_{ha})$ with $Pred(Q_{L_j})$, Ω_{HIC} with Ω_{p5_j}, $W_H(P)$ with $W_{p5_j}(P)$, $\boldsymbol{\mathcal{G}}_H$ with $\boldsymbol{\mathcal{G}}_{L_j}$, δ_H with δ_{L_j}, q_{H_0} with q_{Lj_0}, Lemma 4.2 with Lemma 4.16 and Lemma 4.3 with Lemma 4.17.

□

Definition 4.19. For system $\mathbf{\Phi}$, define the function $\Gamma_{p5_j} : Pred(Q_{L_j}) \to Pred(Q_{L_j})$ according to

$(\forall P \in Pred(Q_{L_j}))$

$$\Gamma_{p5_j}(P) := P - pr(\{q \in Q_{L_j} | (\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j}) \\ \xi_j^l(x, \rho\alpha)! \ \& \ \delta_{L_j}(q, \rho) \models \neg\, \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_\alpha, \Sigma_{L_j}, P)\}).$$

where $P_\alpha := pr(\{q' \in Q_{L_j} | \delta_{L_j}(q', \alpha) \models P\})$ and

$q = (z, y, x)$ as in equation 4.2 (Page 105).

♢

The following lemma will be used later on.

Lemma 4.18. *For system* $\mathbf{\Phi}$, Γ_{p5_j} *is monotone with respect to* $\preceq$, *i.e.*

$$(\forall P_1, P_2 \in Pred(Q_{L_j}))\ P_1 \preceq P_2 \Rightarrow \Gamma_{p5_j}(P_1) \preceq \Gamma_{p5_j}(P_2).$$

proof:

Let $P_1, P_2 \in Pred(Q_{L_j})$. Assume $P_1 \preceq P_2$.

Must show this implies $\Gamma_{p5_j}(P_1) \preceq \Gamma_{p5_j}(P_2)$.

$$\Gamma_{p5_j}(P_1) = P_1 - pr(\{q \in Q_{L_j} | (\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j}) \\ \xi_j^l(x, \rho\alpha)! \ \& \ \delta_{L_j}(q, \rho) \models \neg\, \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_\alpha, \Sigma_{L_j}, P_1)\})$$

$$= P_1 \wedge \neg pr(\{q \in Q_{L_j} | (\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j}) \\ \xi_j^l(x, \rho\alpha)! \ \& \ \delta_{L_j}(q, \rho) \models \neg\, \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_\alpha, \Sigma_{L_j}, P_1)\})$$

$$= P_1 \wedge pr(\{q \in Q_{L_j} | (\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j}) \\ \xi_j^l(x, \rho\alpha)! \Rightarrow \delta_{L_j}(q, \rho) \models \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_\alpha, \Sigma_{L_j}, P_1)\})$$

$$\preceq P_2 \wedge pr(\{q \in Q_{L_j} | (\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j}) \\ \xi_j^l(x, \rho\alpha)! \Rightarrow \delta_{L_j}(q, \rho) \models \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_\alpha, \Sigma_{L_j}, P_2)\}), \\ \text{as } P_1 \preceq P_2 \text{ and } \mathcal{CR} \text{ is monotone.}$$

$$= P_2 \wedge \neg pr(\{q \in Q_{L_j} | (\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j}) \\ \xi_j^l(x, \rho\alpha)! \ \& \ \delta_{L_j}(q, \rho) \models \neg\ \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_\alpha, \Sigma_{L_j}, P_2)\})$$

$$= P_2 - pr(\{q \in Q_{L_j} | (\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j}) \\ \xi_j^l(x, \rho\alpha)! \ \& \ \delta_{L_j}(q, \rho) \models \neg\ \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_\alpha, \Sigma_{L_j}, P_2)\})$$

$$= \Gamma_{p5_j}(P_2)$$

□

The following lemma will be used in the proof of next proposition.

Lemma 4.19. *For system* $\mathbf{\Phi}$, *let* $P \in Pred(Q_{L_j}), w \in \Sigma^*_{IL_j}$ *and* $q \in Q_{L_j}$. *If* $q \models R(\boldsymbol{\mathcal{G}}_{L_j}, P)$ *and* $q = \delta_{L_j}(q_{Lj_0}, w)$, *then*

$(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\ \delta_{L_j}(q, \rho) \models \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_\alpha, \Sigma_{L_j}, P)$ *iff* $(\exists l \in \Sigma^*_{L_j})\ w\rho l\alpha \in L(\boldsymbol{\mathcal{G}}_{L_j}, P)$,

where $P_\alpha := pr(\{q' \in Q_{L_j} | \delta_{L_j}(q', \alpha) \models P\})$ *as defined in Definition 4.19.*

proof:

Assume $q \models R(\boldsymbol{\mathcal{G}}_{L_j}, P)$ and $q = \delta_{L_j}(q_{Lj_0}, w)$.

Let $\rho \in \Sigma_{R_j}$ and $\alpha \in \Sigma_{A_j}$.

$\delta_{L_j}(q, \rho) \models \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_\alpha, \Sigma_{L_j}, P)$

$\Leftrightarrow (\exists k \in \{0, 1, \ldots\})(\exists \sigma_1, \ldots, \sigma_k \in \Sigma_{L_j})(\exists q_1, \ldots, q_{k+2} \in Q_{L_j})$

$$q_1 = \delta_{L_j}(q, \rho)$$

$$q_{k+1} \models P_\alpha$$

$$q_1, \ldots, q_{k+2} \models P$$

$$q_{i+1} = \delta_{L_j}(q_i, \sigma_i), i = 1, 2, \ldots, k$$

$$q_{k+2} = \delta_{L_j}(q_{k+1}, \alpha),$$

by the definitions of $\mathcal{CR}$ and P_α.

$\Leftrightarrow (\exists k \in \{0,1,\ldots\})(\exists \sigma_1,\ldots,\sigma_k \in \Sigma_{L_j})$

$$\delta_{L_j}(q, \rho\sigma_1\cdots\sigma_k\alpha) \models R(\boldsymbol{\mathcal{G}}_{L_j}, P), \quad \text{as } q \models R(\boldsymbol{\mathcal{G}}_{L_j}, P)$$

$\Leftrightarrow (\exists l \in \Sigma^*_{L_j})\ w\rho l\alpha \in L(\boldsymbol{\mathcal{G}}_{L_j}, P)$,

by letting $l = \sigma_1\cdots\sigma_k$ and the assumption $q \models R(\boldsymbol{\mathcal{G}}_{L_j}, P)$ and $q = \delta_{L_j}(q_{Lj_0}, w)$.

□

Proposition 4.23. *For system* $\boldsymbol{\Phi}$, *the following holds:*

$(\forall P \in Pred(Q_{L_j}))$

$\big((\forall q \models P)(\forall \rho \in \Sigma_{R_j})\ \xi^l_j(x,\rho)! \Rightarrow \delta_{L_j}(q,\rho)!\big) \Rightarrow \big(\Omega_{p5_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) = L(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{p5_j}(P))\big)$,

where $q = (z, y, x)$ *as in equation (4.2) (Page 105).*

proof:

Let $P \in Pred(Q_{L_j})$.

Assume $(\forall q \models P)(\forall \rho \in \Sigma_{R_j})\ \xi^l_j(x,\rho)! \Rightarrow \delta_{L_j}(q,\rho)!$. (1)

Must show this implies $\Omega_{p5_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) = L(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{p5_j}(P))$.

We first claim the following holds by (1).

$(\forall q \models P)(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\ \xi^l_j(x,\rho\alpha)! \Rightarrow \delta_{L_j}(q,\rho)!$. (2)

Proof of the claim:

Let $q \models P, \rho \in \Sigma_{R_j}$ and $\alpha \in \Sigma_{A_j}$.

Assume $\xi^l_j(x,\rho\alpha)!$. (3)

Must show implies $\delta_{L_j}(q,\rho)!$.

From (3) and the definition of ξ^l_j, we know that $\xi^l_j(x,\rho)!$ and $\xi^l_j(\xi^l_j(x,\rho),\alpha)!$.

$\Rightarrow \xi^l_j(x,\rho)!$

$\Rightarrow \delta_{L_j}(q,\rho)!$, by (1)

Claim proven.

We now prove this proposition by a series of transformation.

$\Omega_{p5_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P) - L^{W_{p5_j}(P)}(\boldsymbol{\mathcal{G}}_{L_j}, P)$, by Proposition 4.22

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P) - \cup_{w \in W_{p5_j}(P)} L^w(\boldsymbol{\mathcal{G}}_{L_j}, P)$

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - pr(\cup_{w \in W_{p5_j}(P)} \{\delta_{L_j}(q_{L_{j0}}, w)\}))$, by Proposition 2.7

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - pr(\{q \in R(\boldsymbol{\mathcal{G}}_{L_j}, P) | \delta_{L_j}(q_{L_{j0}}, w) = q \text{ for some } w \in W_{p5_j}(P)\}))$,

as $W_{p5_j}(P) \subseteq L(\boldsymbol{\mathcal{G}}_{L_j}, P)$ and by Proposition 2.5

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - pr(\{q \in R(\boldsymbol{\mathcal{G}}_{L_j}, P) | (\exists w \in L(\boldsymbol{\mathcal{G}}_{L_j}, P))(\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j})$

$\delta_{L_j}(q_{L_{j0}}, w) = q \;\&\; w\rho\alpha \in L(\mathbf{G}^l_{I_j}) \;\&\; (\forall l \in \Sigma^*_{L_j})\; w\rho l\alpha \notin L(\boldsymbol{\mathcal{G}}_{L_j}, P)\}))$,

by definition of W_{p5_j}

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - pr(\{q \in R(\boldsymbol{\mathcal{G}}_{L_j}, P) | (\exists w \in L(\boldsymbol{\mathcal{G}}_{L_j}, P))(\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j})$

$\delta_{L_j}(q_{L_{j0}}, w) = q \;\&\; \xi^l_j(x, \rho\alpha)! \;\&$

$(\delta_{L_j}(q, \rho)! \text{ or } \delta_{L_j}(q, \rho) \not!) \;\&(\forall l \in \Sigma^*_{L_j})\; w\rho l\alpha \notin L(\boldsymbol{\mathcal{G}}_{L_j}, P)\}))$,

as $(\delta_{L_j}(q, \rho)! \text{ or } \delta_{L_j}(q, \rho) \not!)$ is always true, where $q = (z, y, x)$ as in equation (4.2)

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - pr(\{q \in R(\boldsymbol{\mathcal{G}}_{L_j}, P) | (\exists w \in L(\boldsymbol{\mathcal{G}}_{L_j}, P))(\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j})$

$\delta_{L_j}(q_{L_{j0}}, w) = q \;\&\; \xi^l_j(x, \rho\alpha)! \;\&$

$(\delta_{L_j}(q, \rho) \not! \text{ or } (\delta_{L_j}(q, \rho)! \;\&\; (\forall l \in \Sigma^*_{L_j})\; w\rho l\alpha \notin L(\boldsymbol{\mathcal{G}}_{L_j}, P)))\}))$,

by logical distribution law and the fact that $\delta_{L_j}(q, \rho) \not! \Rightarrow (\forall l \in \Sigma^*_{L_j})\; w\rho l\alpha \notin L(\boldsymbol{\mathcal{G}}_{L_j}, P)$.

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - pr(\{q \models R(\boldsymbol{\mathcal{G}}_{L_j}, P) | (\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j})$

$\xi^l_j(x, \rho\alpha)! \;\&\; (\delta_{L_j}(q, \rho) \not! \text{ or } (\delta_{L_j}(q, \rho)! \;\&\; \delta_{L_j}(q, \rho) \not\models \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_\alpha, \Sigma_{L_j}, P)))\}))$,

by Lemma 4.19, where $P_\alpha := pr(\{q' \in Q_{L_j} | \delta_{L_j}(q', \alpha) \models P\})$ as defined in Definition 4.19.

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - pr(\{q \models R(\boldsymbol{\mathcal{G}}_{L_j}, P) | (\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j})$

$\xi^l_j(x, \rho\alpha)! \;\&\; (\delta_{L_j}(q, \rho) \not! \text{ or } \delta_{L_j}(q, \rho) \models \neg\, \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_\alpha, \Sigma_{L_j}, P))\}))$,

as $(\delta_{L_j}(q, \rho) \not\models \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_\alpha, \Sigma_{L_j}, P)) \Leftrightarrow (\delta_{L_j}(q, \rho)! \;\&\; \delta_{L_j}(q, \rho) \models \neg\, \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_\alpha, \Sigma_{L_j}, P))$

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - pr(\{q \models R(\boldsymbol{\mathcal{G}}_{L_j}, P) | (\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j})$

$(\xi^l_j(x, \rho\alpha)! \;\&\; \delta_{L_j}(q, \rho) \not!) \text{ or } (\xi^l_j(x, \rho\alpha)! \;\&\; \delta_{L_j}(q, \rho) \models \neg\, \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_\alpha, \Sigma_{L_j}, P))\}))$,

by logical distribution law.

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - pr(\{q \models R(\boldsymbol{\mathcal{G}}_{L_j}, P)|$

$(\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j})\ \xi_j^l(x, \rho\alpha)! \ \&\ \delta_{L_j}(q, \rho) \models \neg\, \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_\alpha, \Sigma_{L_j}, P)\})),$

as $(\xi_j^l(x, \rho\alpha)! \ \&\ \delta_{L_j}(q, \rho)\ \not!)$ is always false by (2).

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - pr(\{q \in Q_{L_j}|$

$(\exists \rho \in \Sigma_{R_j})(\exists \alpha \in \Sigma_{A_j})\ \xi_j^l(x, \rho\alpha)! \ \&\ \delta_{L_j}(q, \rho) \models \neg\, \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_\alpha, \Sigma_{L_j}, P)\})),$

as removing the states satisfying $(true - R(\boldsymbol{\mathcal{G}}_{L_j}, P))$ from P

does not change the resulting language, by Proposition 2.5.

$= L(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{p5_j}(P))$

□

In this proposition, we put the assumption $(\forall q \models P)(\forall \rho \in \Sigma_{R_j})\ \xi_j^l(x, \rho)! \Rightarrow \delta_{L_j}(q, \rho)!$, because in the function Ω_{L_j}, the input language for the function Ω_{p5_j} is always j^{th} low-level P4 interface controllable. Algorithm 4.2 shows how to compute $\Gamma_{p5_j}(P)$, where $P \in Pred(Q_{L_j})$.

Algorithm 4.2 $\Gamma_{p5_j}(P)$

1: $P_{bad5} \leftarrow false$;
2: **for** each $\alpha \in \Sigma_{A_j}$ **do**
3: $\quad P_\alpha \leftarrow pr(\{q' \in Q_{L_j} \mid \delta_{L_j}(q', \alpha) \models P\})$;
4: $\quad P_{\mathcal{CR}_\alpha} \leftarrow \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_\alpha, \Sigma_{L_j}, P)$;
5: $\quad$ **for** each $\rho \in \Sigma_{R_j}$ **do**
6: $\quad\quad P_{bad5} \leftarrow P_{bad5} \vee pr(\{q \in Q_{L_j} \mid \xi_j^l(x, \rho\alpha)! \ \&\ \delta_{L_j}(q, \rho) \models \neg P_{\mathcal{CR}_\alpha}\})$;
7: $\quad$ **end for**
8: **end for**
9: return $P - P_{bad5}$;

In Line 6, $q = (z, y, x)$ is as in equation (4.2). In our BDD implementations, the computation of $P_{\mathcal{CR}_\alpha}$ is a very costly step. However, usually in an HISC system the number of answer events is small. Furthermore, as the computation of each answer event is independent, we can compute the P_{bad5} for each answer event in parallel and then combine them together.

Computing $\Omega_{p6_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$

Definition 4.20. For system $\mathbf{\Phi}$, define the function $W_{p6_j} : Pred(Q_{L_j}) \to Pwr(\Sigma^*_{IL_j})$ according to

$(\forall P \in Pred(Q_{L_j}))$

$W_{p6_j}(P) := \{s \in L(\boldsymbol{\mathcal{G}}_{L_j}, P) \mid s \in L_m(\mathbf{G}^l_{I_j}) \;\&\; (\forall l \in \Sigma^*_{L_j})\; sl \notin L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L(\boldsymbol{\mathcal{G}}_{L_j}, P)\}$

$\diamondsuit$

Clearly, for all $P \in Pred(Q_{L_j}), W_{p6_j}(P) \subseteq L(\boldsymbol{\mathcal{G}}_{L_j}, P)$.

Lemma 4.20. *For system* $\mathbf{\Phi}$, *the following holds:*

$(\forall P \in Pred(Q_{L_j}))(\forall s, t \in L(\boldsymbol{\mathcal{G}}_{L_j}, P))$

$$(s \in W_{p6_j}(P) \;\&\; (\delta_{L_j}(q_{Lj_0}, s) = \delta_{L_j}(q_{Lj_0}, t))) \Rightarrow t \in W_{p6_j}(P).$$

proof:

Let $P \in Pred(Q_{L_j})$, and $s, t \in L(\boldsymbol{\mathcal{G}}_{L_j}, P)$. (1)

Assume $\delta_{L_j}(q_{Lj_0}, s) = \delta_{L_j}(q_{Lj_0}, t)$ and $s \in W_{p6_j}(P)$. (2)

Must show this implies $t \in W_{p6_j}(P)$.

Let $\equiv_{L(\boldsymbol{\mathcal{G}}_{L_j}, P)}, \equiv_{L_m(\mathbf{G}^l_{I_j})}$, and $\equiv_{L_m(\boldsymbol{\mathcal{G}}_{L_j})}$ be the Nerode equivalence relation on $\Sigma^*_{IL_j}$ with respect to $L(\boldsymbol{\mathcal{G}}_{L_j}, P), L_m(\mathbf{G}^l_{I_j})$ and $L_m(\boldsymbol{\mathcal{G}}_{L_j})$ respectively.

From (2), we have $\delta(q_{Lj_0}, s) = \delta(q_{Lj_0}, t)$

$\Rightarrow s \equiv_{L(\boldsymbol{\mathcal{G}}_{L_j}, P)} t \;\&\; s \equiv_{L_m(\boldsymbol{\mathcal{G}}_{L_j})} t,$ (3)

by Proposition 2.6 and fact $L_m(\boldsymbol{\mathcal{G}}_{L_j}) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, true)$.

As $\boldsymbol{\mathcal{G}}_{L_j} = \mathbf{E}_{L_j} \times \mathbf{G}^p_{L_j} \times \mathbf{G}^l_{I_j}$ and $\delta_{L_j}(q_{Lj_0}, s) = \delta_{L_j}(q_{Lj_0}, t)$, we have

$\xi^l_j(x^l_{j0}, s) = \xi^l_j(x^l_{j0}, t)$

$\Rightarrow s \equiv_{L_m(\mathbf{G}^l_{I_j})} t,$ (4)

by Proposition 2.6 and fact $L_m(\mathbf{G}^l_{I_j}) = L_m(\mathbf{G}^l_{I_j}, true)$.

From (2), we have $s \in W_{p6_j}(P)$

$\Rightarrow s \in L_m(\mathbf{G}^l_{I_j})$ & $(\forall l \in \Sigma^*_{L_j})$ $sl \notin L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L(\boldsymbol{\mathcal{G}}_{L_j}, P)$

$\Rightarrow t \in L_m(\mathbf{G}^l_{I_j})$ & $(\forall l \in \Sigma^*_{L_j})$ $tl \notin L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L(\boldsymbol{\mathcal{G}}_{L_j}, P)$, by (3)(4)

$\Rightarrow t \in W_{p6_j}(P)$, as $t \in L(\boldsymbol{\mathcal{G}}_{L_j}, P)$.

□

Lemma 4.21. *For system* $\mathbf{\Phi}$, *the following holds:*

$$(\forall P \in Pred(Q_{L_j}))(\forall s \in L(\boldsymbol{\mathcal{G}}_{L_j}, P))\ ((\exists t \leq s)\ t \in W_{p6_j}(P)) \Leftrightarrow s \notin \Omega_{p6_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))).$$

proof:

Let $P \in Pred(Q_{L_j})$ and $s \in L(\boldsymbol{\mathcal{G}}_{L_j}, P)$. (1)

1. Show that $((\exists t \leq s)\ t \in W_{p6_j}(P)) \Rightarrow s \notin \Omega_{p6_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$.

   Assume $(\exists t \leq s)\ t \in W_{p6_j}(P)$. (2)

   Must show this implies $s \notin \Omega_{p6_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$.

   From (2), we have

   $(\exists t \leq s)\ t \in L_m(\mathbf{G}^l_{I_j})$ & $(\forall l \in \Sigma^*_{L_j})$ $tl \notin L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L(\boldsymbol{\mathcal{G}}_{L_j}, P)$

   $\Rightarrow \neg((\forall t \leq s)\ t \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ tl \in L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L(\boldsymbol{\mathcal{G}}_{L_j}, P))$

   $\Rightarrow s \notin \Omega_{p6_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$

2. Show that $s \notin \Omega_{p6_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) \Rightarrow ((\exists t \leq s)\ t \in W_{p6_j}(P))$.

   Assume $s \notin \Omega_{p6_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$. (3)

   Must show this implies $(\exists t \leq s)\ t \in W_{p6_j}(P)$.

   From (1), we know $s \in L(\boldsymbol{\mathcal{G}}_{L_j}, P)$. Combine this with (3), we thus have

   $\neg((\forall t \leq s)\ t \in L_m(\mathbf{G}^l_{I_j}) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ tl \in L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L(\boldsymbol{\mathcal{G}}_{L_j}, P))$

   $\Rightarrow (\exists t \leq s)\ t \in L_m(\mathbf{G}^l_{I_j})$ & $(\forall l \in \Sigma^*_{L_j})$ $tl \notin L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L(\boldsymbol{\mathcal{G}}_{L_j}, P)$ (4)

As $s \in L(\boldsymbol{\mathcal{G}}_{L_j}, P)$ and $L(\boldsymbol{\mathcal{G}}_{L_j}, P)$ is closed, we know $t \in L(\boldsymbol{\mathcal{G}}_{L_j}, P)$. (5)

By (4) and (5), we thus have $(\exists t \leq s)\ t \in W_{p6_j}(P)$.

□

Proposition 4.24. *For system* $\boldsymbol{\Phi}$*, the following holds*

$$(\forall P \in Pred(Q_{L_j}))\ L(\boldsymbol{\mathcal{G}}_{L_j}, P) - \Omega_{p6_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) = L^{W_{p6_j}(P)}(\boldsymbol{\mathcal{G}}_{L_j}, P),$$

where $L^{W_{p6_j}(P)}(\boldsymbol{\mathcal{G}}_{L_j}, P) = \cup_{s \in W_{p6_j}(P)} L^s(\boldsymbol{\mathcal{G}}_{L_j}, P)$.

proof: Identical to the proof of Proposition 4.6 after substituting $Sub(\mathcal{P}_{ha})$ with $Pred(Q_{L_j})$, Ω_{HIC} with Ω_{p6_j}, $W_H(P)$ with $W_{p6_j}(P)$, $\boldsymbol{\mathcal{G}}_H$ with $\boldsymbol{\mathcal{G}}_{L_j}$, δ_H with δ_{L_j}, q_{H_0} with q_{Lj_0}, Lemma 4.2 with Lemma 4.20, Lemma 4.3 with Lemma 4.21.

□

Definition 4.21. For system $\boldsymbol{\Phi}$, define the function $\Gamma_{p6_j} : Pred(Q_{L_j}) \to Pred(Q_{L_j})$ according to

$(\forall P \in Pred(Q_{L_j}))\ \Gamma_{p6_j}(P) := P - (P_{X_{j_m}} - \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_{j_m}}, \Sigma_{L_j}, P))$,

where $P_{X_{j_m}} := pr(\{q \in Q_{L_j} | x \in X^l_{j_m}\})$, $P_{L_{j_m}} := pr(Q_{L_{j_m}})$,

$q = (z, y, x)$ as in equation 4.2 (Page 105),

and $X^l_{j_m}$ and $Q_{L_{j_m}}$ are the marker state sets of $\mathbf{G}^l_{I_j}$ and $\boldsymbol{\mathcal{G}}_{L_j}$ respectively as defined in Section 4.1.

♢

Note that for system $\boldsymbol{\Phi}$, $P_{X_{j_m}}$ and $P_{L_{j_m}}$ are constant. The following lemma will be used later on.

Lemma 4.22. *For system* $\boldsymbol{\Phi}$*,* Γ_{p6_j} *is monotone with respect to* $\preceq$*, i.e.*

$$(\forall P_1, P_2 \in Pred(Q_{L_j}))\ P_1 \preceq P_2 \Rightarrow \Gamma_{p6_j}(P_1) \preceq \Gamma_{p6_j}(P_2).$$

proof:

Let $P_1, P_2 \in Pred(Q_{L_j})$. Assume $P_1 \preceq P_2$.

Must show this implies $\Gamma_{p6_j}(P_1) \preceq \Gamma_{p6_j}(P_2)$.

$\Gamma_{p6_j}(P_1)$

$= P_1 - (P_{X_{j_m}} - \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_{j_m}}, \Sigma_{L_j}, P_1))$, by definition of $\Gamma_{p6_j}(P_1)$

$= P_1 \wedge \neg(P_{X_{j_m}} \wedge \neg\, \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_{j_m}}, \Sigma_{L_j}, P_1))$

$= P_1 \wedge (\neg P_{X_{j_m}} \vee \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_{j_m}}, \Sigma_{L_j}, P_1))$

$\preceq P_2 \wedge (\neg P_{X_{j_m}} \vee \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_{j_m}}, \Sigma_{L_j}, P_2))$, as $P_1 \preceq P_2$ and $\mathcal{CR}$ is monotone

$= P_2 \wedge \neg(P_{X_{j_m}} \wedge \neg\, \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_{j_m}}, \Sigma_{L_j}, P_2))$

$= P_2 - (P_{X_{j_m}} - \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_{j_m}}, \Sigma_{L_j}, P_2))$

$= \Gamma_{p6_j}(P_2)$

□

In order to show the method to compute Ω_{p6_j}, we need the following proposition and lemma.

Proposition 4.25. *For system* $\boldsymbol{\Phi}$, *the following holds:*

$$(\forall P \in Pred(Q_{L_j}))\ L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L(\boldsymbol{\mathcal{G}}_{L_j}, P) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, CR(\boldsymbol{\mathcal{G}}_{L_j}, P)).$$

proof:

Identical to the proof of Proposition 4.10 after substituting $\boldsymbol{\mathcal{G}}_H$ with $\boldsymbol{\mathcal{G}}_{L_j}$, Q_H with Q_{L_j}, Q_{H_m} with $Q_{L_{j_m}}$, δ_H with δ_{L_j}, Σ_{IH} with Σ_{IL_j}.

□

Lemma 4.23. *For system* $\boldsymbol{\Phi}$, *the following holds:*

$$(\forall P \in Pred(Q_{L_j}))\ L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L(\boldsymbol{\mathcal{G}}_{L_j}, P) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, P).$$

proof:

Let $P \in Pred(Q_{L_j})$.

1. Show that $L_m(\boldsymbol{\mathcal{G}}_{L_j}, P) \subseteq L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L(\boldsymbol{\mathcal{G}}_{L_j}, P)$.

   Clearly, $L_m(\boldsymbol{\mathcal{G}}_{L_j}, P) \subseteq L_m(\boldsymbol{\mathcal{G}}_{L_j})$ and $L_m(\boldsymbol{\mathcal{G}}_{L_j}, P) \subseteq L(\boldsymbol{\mathcal{G}}_{L_j}, P)$, so $L_m(\boldsymbol{\mathcal{G}}_{L_j}, P) \subseteq L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L(\boldsymbol{\mathcal{G}}_{L_j}, P)$.

2. Show that $L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L(\boldsymbol{\mathcal{G}}_{L_j}, P) \subseteq L_m(\boldsymbol{\mathcal{G}}_{L_j}, P)$.

   By Proposition 4.25, we know $L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L(\boldsymbol{\mathcal{G}}_{L_j}, P) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, CR(\boldsymbol{\mathcal{G}}_{L_j}, P))$.

   By the definition of CR, we also know that $L_m(\boldsymbol{\mathcal{G}}_{L_j}, CR(\boldsymbol{\mathcal{G}}_{L_j}, P)) \subseteq L_m(\boldsymbol{\mathcal{G}}_{L_j}, P)$.

   So, $L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L(\boldsymbol{\mathcal{G}}_{L_j}, P) \subseteq L_m(\boldsymbol{\mathcal{G}}_{L_j}, P)$.

□

The following lemma will be used in the proof of next proposition.

Lemma 4.24. *For system* $\boldsymbol{\Phi}$, *let* $P \in Pred(Q_{L_j}), w \in \Sigma^*_{IL_j}$ *and* $q \in Q_{L_j}$. *If* $q = \delta(q_{Lj_0}, w)$ *and* $q \models R(\boldsymbol{\mathcal{G}}_{L_j}, P)$, *then*

$$q \models \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_{Lj_m}, \Sigma_{L_j}, P) \text{ if and only if } (\exists l \in \Sigma^*_{L_j})\ wl \in L_m(\boldsymbol{\mathcal{G}}_{L_j}, P),$$

where $P_{L_{j_m}} := pr(Q_{L_{j_m}})$ *as defined in Definition 4.21.*

proof:

Assume $q = \delta(q_{Lj_0}, w)$ and $q \models R(\boldsymbol{\mathcal{G}}_{L_j}, P)$.

$q \models \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_{Lj_m}, \Sigma_{L_j}, P)$

$\Leftrightarrow (\exists k \in \{0, 1, \ldots\})(\exists \sigma_1, \ldots, \sigma_k \in \Sigma_{L_j})(\exists q_1, \ldots, q_{k+1} \in Q_{L_j})$

$$q_1 = q$$

$$q_{k+1} \models P_{Lj_m}$$

$$q_1, \ldots, q_{k+1} \models P$$

$$q_{i+1} = \delta_{L_j}(q_i, \sigma_i), i = 1, 2, \ldots, k$$

$\Leftrightarrow (\exists k \in \{0, 1, \ldots\})(\exists \sigma_1, \ldots, \sigma_k \in \Sigma_{L_j})$

$\delta_{L_j}(q, \sigma_1 \ldots \sigma_k) \models R(\boldsymbol{\mathcal{G}}_{L_j}, P)$ & $\delta_{L_j}(q, \sigma_1 \ldots \sigma_k) \in Q_{Lj_m}$,

by $q \models R(\boldsymbol{\mathcal{G}}_{L_j}, P)$ and the definition of P_{Lj_m}.

$\Leftrightarrow (\exists l \in \Sigma^*_{L_j})\ wl \in L_m(\boldsymbol{\mathcal{G}}_{L_j}, P)$,

by letting $l = \sigma_1 \ldots \sigma_k$ and the assumption $q = \delta(q_{Lj_0}, w)$ and $q \models R(\boldsymbol{\mathcal{G}}_{L_j}, P)$.

□

Proposition 4.26. *For system* $\boldsymbol{\Phi}$, *the following holds:*

$(\forall P \in Pred(Q_{L_j}))\ \Omega_{p6_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) = L(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{p6_j}(P))$.

proof:

Let $P \in Pred(Q_{L_j})$. We prove this proposition by a series of transformation.

$\Omega_{p6_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P) - L^{W_{p6_j}(P)}(\boldsymbol{\mathcal{G}}_{L_j}, P)$, by Proposition 4.24

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P) - \cup_{w \in W_{p6_j}(P)} L^w(\boldsymbol{\mathcal{G}}_{L_j}, P)$

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - pr(\cup_{w \in W_{p6_j}(P)} \{\delta_{L_j}(q_{Lj_0}, w)\}))$, by Proposition 2.7

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - pr(\{q \in R(\boldsymbol{\mathcal{G}}_{L_j}, P) | \delta_{L_j}(q_{Lj_0}, w) = q \text{ for some } w \in W_{p6_j}(P)\}))$,

as $W_{p6_j}(P) \subseteq L(\boldsymbol{\mathcal{G}}_{L_j}, P)$ and by Proposition 2.5

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - pr(\{q \in R(\boldsymbol{\mathcal{G}}_{L_j}, P) | (\exists w \in L(\boldsymbol{\mathcal{G}}_{L_j}, P))\ \delta_{L_j}(q_{Lj_0}, w) = q$ &

$w \in L_m(\mathbf{G}^l_{I_j})$ & $(\forall l \in \Sigma^*_{L_j})\ wl \notin L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L(\boldsymbol{\mathcal{G}}_{L_j}, P)\}))$,

by definition of $W_{p6_j}(P)$

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - pr(\{q \in R(\boldsymbol{\mathcal{G}}_{L_j}, P) | (\exists w \in L(\boldsymbol{\mathcal{G}}_{L_j}, P))\ \delta_{L_j}(q_{Lj_0}, w) = q$ &

$w \in L_m(\mathbf{G}^l_{I_j})$ & $(\forall l \in \Sigma^*_{L_j})\ wl \notin L_m(\boldsymbol{\mathcal{G}}_{L_j}, P)\}))$, by Lemma 4.23

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - pr(\{q \models R(\boldsymbol{\mathcal{G}}_{L_j}, P) | x \in X^l_{j_m}$ & $q \not\models \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_{j_m}}, \Sigma_{L_j}, P)\}))$,

by Lemma 4.24,

where $P_{L_{j_m}} := pr(Q_{L_{j_m}})$ as defined in Definition 4.21,

and $q = (z, y, x)$ as in Equation 4.2 (Page 105).

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - pr(\{q \in Q_{L_j} | x \in X^l_{j_m} \ \&\ q \not\models \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_{j_m}}, \Sigma_{L_j}, P)\}))$,

as removing the states satisfying $(true - R(\boldsymbol{\mathcal{G}}_{L_j}, P))$

does not change the resulting language, by Proposition 2.5.

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - (pr(\{q \in Q_{L_j} | x \in X^l_{j_m}\}) - \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_{j_m}}, \Sigma_{L_j}, P)))$

$= L(\boldsymbol{\mathcal{G}}_{L_j}, P - (P_{X_{j_m}} - \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_{j_m}}, \Sigma_{L_j}, P)))$, where $P_{X_{j_m}}$ is as in Definition 4.21.

$= L(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{p6_j}(P))$.

□

4.3.5 The Algorithm to Compute $\sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}))$

For a predicate $P \in Pred(Q_{L_j})$, we know from Proposition 4.25 and the definition of Ω_{LNB_j} that

$$\Omega_{LNB_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) = L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L_(\boldsymbol{\mathcal{G}}_{L_j}, P) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, CR(\boldsymbol{\mathcal{G}}_{L_j}, P)).$$

We now put all the predicate functions together to show the method to compute $\sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}))$.

Definition 4.22. For system $\boldsymbol{\Phi}$, define the function $\Gamma_{L_j} : Pred(Q_{L_j}) \to Pred(Q_{L_j})$ according to

$$(\forall P \in Pred(Q_{L_j}))\ \Gamma_{L_j}(P) := CR(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{p6_j}(\Gamma_{p5_j}(\mathrm{PLPC}_j(P))))$$

♢

By Lemma 4.12, PLPC_j is monotone; by Lemma 4.18, Γ_{p5_j} is monotone; by Lemma 4.22, Γ_{p6_j} is monotone; the predicate transformer $CR(\boldsymbol{\mathcal{G}}_{L_j}, .)$ is monotone; thus the function Γ_{L_j} is also monotone.

Lemma 4.25. *For system* $\boldsymbol{\Phi}$, *the following holds:*

$$(\forall P \in Pred(Q_{L_j}))\ \overline{L_m(\boldsymbol{\mathcal{G}}_{L_j}, CR(\boldsymbol{\mathcal{G}}_{L_j}, P))} = L(\boldsymbol{\mathcal{G}}_{L_j}, CR(\boldsymbol{\mathcal{G}}_{L_j}, P))$$

proof:

Identical to the proof of Lemma 4.6 by substituting $\boldsymbol{\mathcal{G}}_H$ with $\boldsymbol{\mathcal{G}}_{L_j}$, Q_H with Q_{L_j}, Q_{H_m} with $Q_{L_{j_m}}$, Σ_{IH} with Σ_{IL_j}, δ_H with δ_{L_j}.

Lemma 4.26. *For system* $\mathbf{\Phi}$, *the following holds*

$$(\forall i \in \{1,2,\ldots\})(\forall P \in Pred(Q_{L_j}))\ \overline{L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^i_{L_j}(P))} = L(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^i_{L_j}(P))$$

proof:

Let $i \in \{1,2,\ldots\}$, and $P \in Pred(Q_{L_j})$. By definition of Γ_{L_j}, we have

$$\begin{aligned}
\overline{L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^i_{L_j}(P))} &= \overline{L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{L_j}(\Gamma^{i-1}_{L_j}(P)))} \\
&= \overline{L_m(\boldsymbol{\mathcal{G}}_{L_j}, CR(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{p6_j}(\Gamma_{p5_j}(\mathrm{PLPC}_j(\Gamma^{i-1}_{L_j}(P))))))}, \\
&\quad \text{by definition of } \Gamma_{L_j} \\
&= \overline{L_m(\boldsymbol{\mathcal{G}}_{L_j}, CR(\boldsymbol{\mathcal{G}}_{L_j}, P'))}, \\
&\quad \text{by letting } P' := \Gamma_{p6_j}(\Gamma_{p5_j}(\mathrm{PLPC}_j(\Gamma^{i-1}_{L_j}(P)))) \\
&= L(\boldsymbol{\mathcal{G}}_{L_j}, CR(\boldsymbol{\mathcal{G}}_{L_j}, P')), \quad \text{by Lemma 4.25} \\
&= L(\boldsymbol{\mathcal{G}}_{L_j}, CR(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{p6_j}(\Gamma_{p5_j}(\mathrm{PLPC}_j(\Gamma^{i-1}_{L_j}(P)))))) \\
&= L(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^i_{L_j}(P)), \quad \text{by definition of } \Gamma_{L_j}
\end{aligned}$$

□

Proposition 4.27. *For system* $\mathbf{\Phi}$, *the following two points hold:*

1. $(\forall P \in Pred(Q_{L_j}))\ \Omega_{L_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{L_j}(P))$
2. $(\forall P \in Pred(Q_{L_j}))(\forall i \in \{1,2,\ldots\})\ \Omega^i_{L_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^i_{L_j}(P))$

proof:

1. Show that $(\forall P \in Pred(Q_{L_j}))\ \Omega_{L_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{L_j}(P))$

   Let $P \in Pred(Q_{L_j})$.

$\Omega_{L_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))$

$= \Omega_{LNB_j}(\Omega_{p6_j}(\Omega_{p5_j}(\text{LPC}_j(L(\boldsymbol{\mathcal{G}}_{L_j}, P))))))$, by definition of Ω_{L_j}

$= L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \Omega_{p6_j}(\Omega_{p5_j}(\sup \mathcal{LPC}_j(\overline{L(\boldsymbol{\mathcal{G}}_{L_j}, P)})))$, by definitions of LPC_j and Ω_{LNB_j}

$= L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \Omega_{p6_j}(\Omega_{p5_j}(\sup \mathcal{LPC}_j(L(\boldsymbol{\mathcal{G}}_{L_j}, P))))$, as $L(\boldsymbol{\mathcal{G}}_{L_j}, P)$ is closed

$= L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \Omega_{p6_j}(\Omega_{p5_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, \text{PLPC}_j(P))))$, by Proposition 4.18.

$= L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap \Omega_{p6_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{p5_j}(\text{PLPC}_j(P))))$, by Lemma 4.13 and Proposition 4.23.

$= L_m(\boldsymbol{\mathcal{G}}_{L_j}) \cap L(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{p6_j}(\Gamma_{p5_j}(\text{PLPC}_j(P))))$, by Proposition 4.26

$= L_m(\boldsymbol{\mathcal{G}}_{L_j}, CR(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{p6_j}(\Gamma_{p5_j}(\text{PLPC}_j(P)))))$, by Proposition 4.25

$= L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{L_j}(P))$

2. Show that $(\forall P \in Pred(Q_{L_j}))(\forall i \in \{1, 2, \ldots\})\ \Omega^i_{L_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^i_{L_j}(P))$

   Let $P \in Pred(Q_{L_j})$. We now prove this by induction.

   (a) Base Case: $i = 1$

   By Point 1, we have $\Omega^1_{L_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) = L^1_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{L_j}(P))$

   (b) Inductive step.

   Let $k \in \{1, 2, \ldots\}$. Assume $\Omega^k_{L_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(P))$.

   Must show $\Omega^{k+1}_{L_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^{k+1}_{L_j}(P))$.

$$\begin{aligned}
\Omega^{k+1}_{L_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P)) &= \Omega_{L_j}(\Omega^k_{L_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, P))) \\
&= \Omega_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(P))), \text{ by inductive assumption} \\
&= \Omega_{LNB_j}(\Omega_{p6_j}(\Omega_{p5_j}(\sup \mathcal{LPC}_j(\overline{L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(P))})))), \\
&\qquad \text{by definition of } \Omega_{L_j} \\
&= \Omega_{LNB_j}(\Omega_{p6_j}(\Omega_{p5_j}(\sup \mathcal{LPC}_j(L(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(P)))))), \\
&\qquad \text{by Lemma 4.26}
\end{aligned}$$

$= \Omega_{L_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(P)))$, by the definition of Ω_{L_j}.

$= L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^{k+1}_{L_j}(P))$, by Point 1 of this proposition.

□

Theorem 4.2. *For system* $\boldsymbol{\Phi}$*, the following two points hold:*

1. *There exists* $k \in \{0, 1, \ldots\}$ *such that* $k \leq |Q_{L_j}|$ *and* $\Gamma^k_{L_j}(true)$ *is the greatest fixpoint of the function* Γ_{L_j} *with respect to* $(Pred(Q_{L_j}), \preceq)$.

2. $\sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j})) = \sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}, true)) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true))$

proof:

1. Show that there exists $k \in \{0, 1, \ldots\}$ such that $k \leq |Q_{L_j}|$ and $\Gamma^k_{L_j}(true)$ is the greatest fixpoint of the function Γ_{L_j} with respect to $(Pred(Q_{L_j}), \preceq)$.

   As Γ_{L_j} is monotone and $|Q_{L_j}|$ is assumed to be finite (Section 4.1), we know immediately from Proposition 2.3 that there exists $k \in \{0, 1, \ldots\}$ such that $k \leq |Q_{L_j}|$ and $\Gamma^k_{L_j}(true)$ is the greatest fixpoint of the function Γ_{L_j}.

2. Show that $\sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j})) = \sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}, true)) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true))$.

   $\sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j})) = \sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}, true))$ is automatic as $L_m(\boldsymbol{\mathcal{G}}_{L_j}) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, true)$.

   (a) Show that $\sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}, true)) \subseteq L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true))$

   By Proposition 2.3, we know that

   $(\forall i \in \{1, 2, \ldots\})\ i > k \Rightarrow L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^i_{L_j}(true)) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true))$

   $\Rightarrow \lim_{i\to\infty} L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^i_{L_j}(true)) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true))$

   $\Rightarrow \lim_{i\to\infty} \Omega^i_{L_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, true)) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true))$, by Proposition 4.27

   $\Rightarrow \sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}, true)) \subseteq L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true))$, by Proposition 4.21

(b) Show that $L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true)) \subseteq \sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}, true))$.

Based on the value of k, we now show this in two cases:

- Case 1: $k \in \{1, 2, \ldots\}$

  By Point 1 of this theorem and Proposition 2.3, we have

  $\Gamma^k_{L_j}(true) = \Gamma^{k+1}_{L_j}(true)$

  $\Rightarrow L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true)) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^{k+1}_{L_j}(true))$

  $\Rightarrow L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true)) = \Omega^{k+1}_{L_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, true))$, by Proposition 4.27

  $\Rightarrow L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true)) = \Omega_{L_j}(\Omega^k_{L_j}(L(\boldsymbol{\mathcal{G}}_{L_j}, true)))$

  $\Rightarrow L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true)) = \Omega_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true)))$,

  by Proposition 4.27 and $k \in \{1, 2, \ldots\}$

- Case 2: $k = 0$

  If $\Gamma^0_{L_j}(true)$ is the greatest fixpoint of Γ_{L_j}, then by Proposition 2.3 we know that

  $$\Gamma^1_{L_j}(true) = \Gamma^0_{L_j}(true), \tag{1}$$

  so $\Gamma^1_{L_j}(true)$ is also the greatest fixpoint of Γ_{L_j}.

  $\Rightarrow L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^1_{L_j}(true)) = \Omega_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^1_{L_j}(true)))$, by Case 1

  $\Rightarrow L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^0_{L_j}(true)) = \Omega_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^0_{L_j}(true)))$, by (1)

From Case 1 and Case 2, we know that $L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true))$ is always a fixpoint of the function Ω_{L_j}.

From Proposition 4.20, $\sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}, true))$ is the greatest fixpoint of the function Ω_{L_j}, so $L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true)) \subseteq \sup \mathcal{C}_{L_j}(L_m(\boldsymbol{\mathcal{G}}_{L_j}, true))$.

□

Corollary 4.4. *For system* $\boldsymbol{\Phi}$, *let* $\mathbf{S}_{L_j}$ *be a DES defined over event set* Σ_{IL_j} *with* $L_m(\mathbf{S}_{L_j}) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true))$ *and* $L(\mathbf{S}_{L_j}) = L(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true))$, *where* $k \in \{0, 1, \ldots\}$

and $\Gamma^k_{L_j}(true)$ *is the greatest fixpoint of* Γ_{L_j} *with respect to* $(Pred(Q_{L_j}), \preceq)$. *Then for the* n^{th} *degree parallel interface system composed of* $\mathbf{G}^p_H$, $\mathbf{G}^p_{L_1}$, …, $\mathbf{G}^p_{L_n}$, $\mathbf{G}_{I_1}$, …, $\mathbf{G}_{I_n}$, $\mathbf{E}_H$, $\mathbf{E}_{L_1}$, …, $\mathbf{E}_{L_{j-1}}$, $\mathbf{S}_{L_j}$, $\mathbf{E}_{L_{j+1}}$, …, $\mathbf{E}_{L_n}$ *with respect to the alphabet partition in Equation 3.1,* $\mathbf{S}_{L_j}$ *is a* j^{th} *low-level proper supervisor.*

proof:

We first note that by Theorem 4.2, an appropriate $k \in \{0, 1, \ldots\}$ exists such that $\Gamma^k_{L_j}(true)$ is the greatest fixpoint of Γ_{L_j} with respect to $(Pred(Q_{L_j}), \preceq)$.

1. Show that $\overline{L_m(\mathbf{S}_{L_j} \times \mathbf{G}^p_{L_j} \times \mathbf{G}^l_{I_j})} = L(\mathbf{S}_{L_j} \times \mathbf{G}^p_{L_j} \times \mathbf{G}^l_{I_j})$.

   $L_m(\mathbf{S}_{L_j}) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true))$

   $\Rightarrow L_m(\mathbf{S}_{L_j}) \subseteq L_m(\boldsymbol{\mathcal{G}}_{L_j})$

   $\Rightarrow L_m(\mathbf{S}_{L_j}) \subseteq L_m(\mathbf{E}_{L_j} \times \mathbf{G}^p_{L_j} \times \mathbf{G}^l_{I_j})$

   $\Rightarrow L_m(\mathbf{S}_{L_j}) \subseteq L_m(\mathbf{G}^p_{L_j} \times \mathbf{G}^l_{I_j})$

   $\Rightarrow L_m(\mathbf{S}_{L_j}) = L_m(\mathbf{S}_{L_j} \times \mathbf{G}^p_{L_j} \times \mathbf{G}^l_{I_j})$ (1)

   Similarly, we can show that $L(\mathbf{S}_{L_j}) = L(\mathbf{S}_{L_j} \times \mathbf{G}^p_{L_j} \times \mathbf{G}^l_{I_j})$ (2)

   If $k \in \{1, 2, \ldots\}$ and $\Gamma^k_{L_j}(true)$ is the greatest fixpoint of Γ_{L_j}, then by Lemma 4.26 we know that $\overline{L_m(\mathbf{S}_{L_j})} = L(\mathbf{S}_{L_j})$.

   If $k = 0$ and $\Gamma^0_{L_j}(true)$ is the greatest fixpoint of Γ_{L_j}, then as $|Q_{L_j}|$ is assumed to be finite (Section 4.1), by Proposition 2.3 we know that $\Gamma^1_{L_j}(true) = \Gamma^0_{L_j}(true)$.

   $\Rightarrow L(\mathbf{S}_{L_j}) = L(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^1_{L_j}(true))$ and $L_m(\mathbf{S}_{L_j}) = L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^1_{L_j}(true))$

   $\Rightarrow \overline{L_m(\mathbf{S}_{L_j})} = L(\mathbf{S}_{L_j})$, by Lemma 4.26.

   Therefore, we always have

   $$\overline{L_m(\mathbf{S}_{L_j})} = L(\mathbf{S}_{L_j}), \tag{3}$$

   when $k \in \{0, 1, \ldots\}$ and $\Gamma^k_{L_j}(true)$ is the greatest fixpoint of Γ_{L_j}.

Combining (1), (2) and (3), we can conclude $\overline{L_m(\mathbf{S}_{L_j} \times \mathbf{G}^p_{L_j} \times \mathbf{G}^l_{I_j})} = L(\mathbf{S}_{L_j} \times \mathbf{G}^p_{L_j} \times \mathbf{G}^l_{I_j})$.

2. Show that $\mathbf{S}_{L_j}$ is the j^{th} low-level interface controllable.

   By Part 1 and Definition 3.8, Definition 4.10, and Definition 4.16, it is sufficient to show that $L_m(\mathbf{S}_{L_j} \times \mathbf{G}^p_{L_j} \times \mathbf{G}^l_{I_j})$ is j^{th} low-level interface controllable.

   From Theorem 4.2, we know that $L_m(\mathbf{S}_{L_j})$ is j^{th} low-level interface controllable. By (1), we can conclude $L_m(\mathbf{S}_{L_j} \times \mathbf{G}^p_{L_j} \times \mathbf{G}^l_{I_j})$ is also j^{th} low-level interface controllable. □

The supervisor $\mathbf{S}_{L_j}$ can be constructed by removing states from $\boldsymbol{\mathcal{G}}_{L_j}$ that do not satisfy $\Gamma^k_{L_j}(true)$. A trim supervisor DES $\mathbf{S}_{L_j}$ can be built by removing states from $\boldsymbol{\mathcal{G}}_{L_j}$ that do not satisfy $R(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma^k_{L_j}(true))$.

Algorithm 4.3 shows how to compute $\Gamma^k_{L_j}(true)$, where $k \in \{0, 1, \ldots\}$ and $\Gamma^k_{L_j}(true)$ is the greatest fixpoint of Γ_{L_j} with respect to $(Pred(Q_{L_j}), \preceq)$. By Point 1 of Theorem 4.2, we know the greatest fixpoint will be reached after finite number of iterations.

Algorithm 4.3 Computing the greatest fixpoint of Γ_{L_j} w.r.t. $(Pred(Q_{L_j}), \preceq)$

```
1: P_{bad_{L_j}} ← pr(Bad_{L_j});
2: P_1 ← true;
3: repeat
4:     P_2 ← P_1;
5:     P_1 ← ¬TR(G_{L_j}, ¬P_1 ∨ P_{bad_{L_j}}, Σ_{lu_j} ∪ Σ_{R_j});
6:     P_1 ← Γ_{p5_j}(P_1);
7:     P_1 ← P_1 − (P_{X_{j_m}} − CR(G_{L_j}, P_{L_{j_m}}, Σ_{L_j}, P_1));
8:     P_1 ← CR(G_{L_j}, P_1);
9: until P_1 = P_2
10: return P_1;
```

In the algorithm, P_{Xj_m} and P_{Lj_m} are as defined in Definition 4.21. Line 5 computes $\text{PLPC}_j(P_1)$. $\Gamma_{p5_j}(P_1)$ in Line 6 can be computed by using Algorithm 4.2. Line 7 computes $\Gamma_{p6_j}(P_1)$. Line 8 is for the function Ω_{LNB_j}.

Chapter 5

Verification of HISC

In this chapter, we briefly discuss the method to verify a n^{th} degree supervisor interface system and we give the verification algorithms.

In the previous chapter, we proved that for system $\mathbf{\Phi}$, we can synthesize a high-level proper supervisor by trimming off states from $\boldsymbol{\mathcal{G}}_H$ and the j^{th} low-level proper supervisor by trimming off states from $\boldsymbol{\mathcal{G}}_{L_j}$.

For the n^{th} degree supervisor interface system that respects the alphabet partition given by 3.1 and is composed of plant components $\mathbf{G}^p_H, \mathbf{G}^p_{L_1}, \ldots, \mathbf{G}^p_{L_n}$, supervisors $\mathbf{S}_H, \mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, and interfaces $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$, we can treat all the supervisors as their corresponding specifications (assuming the system is HISC-valid) and apply the synthesis algorithm to the system. If $\mathbf{S}_H$ is a high-level proper supervisor, then there would be no *reachable* state that must be removed from $\boldsymbol{\mathcal{G}}_H$. Similarly, if $\mathbf{S}_{L_j}$ is a j^{th} low-level proper supervisor($j \in \{1, \ldots, n\}$), then there would be no *reachable* state that must be removed from $\boldsymbol{\mathcal{G}}_{L_j}$.

Algorithm 5.1 shows the verification algorithm for the high-level. Bad_H in Line 2 is a state subset as defined in Definition 4.7, and $pr(\text{Bad}_H)$ denotes the predicate identified by Bad_H.

Algorithm 5.1 High-level verification

1: $P_R \leftarrow R(\boldsymbol{\mathcal{G}}_H, true)$;
2: $P_{bad_H} \leftarrow pr(\text{Bad}_H)$;
3: **if** $(P_R \wedge P_{bad_H})$ **then**
4: return "The high-level supervisor is not controllable or the high-level fails to satisfy interface consistent condition Point 3.";
5: **end if**
6: **if** $(P_R \neq P_R \wedge CR(\boldsymbol{\mathcal{G}}_H, true))$ **then**
7: return "The high-level is blocking.";
8: **end if**
9: return "The high-level supervisor is high-level interface controllable and the high-level is nonblocking.";

Algorithm 5.2 shows the verification algorithm for the j^{th} low-level. Bad_{L_j} in Line 2 is a state subset as defined in Definition 4.14, and $pr(\text{Bad}_{L_j})$ is the predicate identified by Bad_{L_j}. $\Gamma_{p5_j}(true)$ in Line 6 can be computed by using Algorithm 4.2. Line 9 verifies the interface consistent condition Point 6. Line 12 verifies the nonblocking property.

Algorithm 5.2 The j^{th} low-level verification

1: $P_R \leftarrow R(\boldsymbol{\mathcal{G}}_{L_j}, true)$;
2: $P_{bad_{L_j}} \leftarrow pr(\text{Bad}_{L_j})$;
3: **if** $(P_R \wedge P_{bad_{L_j}})$ **then**
4: return "The j^{th} low-level supervisor is not controllable or the j^{th} low-level fails to satisfy interface consistent condition Point 4.";
5: **end if**
6: **if** $P_R \neq P_R \wedge \Gamma_{p5_j}(true)$ **then**
7: return "The j^{th} low-level fails to satisfy interface consistent condition Point 5.";
8: **end if**
9: **if** $P_R \neq P_R \wedge \neg(P_{X_{j_m}} - \mathcal{CR}(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_{j_m}}, \Sigma_{L_j}, true))$ **then**
10: return "The j^{th} low-level fails to satisfy interface consistent condition Point 6.";
11: **end if**
12: **if** $P_R \neq P_R \wedge CR(\boldsymbol{\mathcal{G}}_{L_j}, true)$ **then**
13: return "The j^{th} low-level is blocking.";
14: **end if**
15: return "The j^{th} low-level supervisor is j^{th} low-level interface controllable and the j^{th} low-level is nonblocking.";

Usually, the verification process is faster than the synthesis process, because the verification algorithms do not need to compute the greatest fixpoints as Algorithm 4.1 and Algorithm 4.3 need to do.

Chapter 6

Symbolic Computation for HISC Synthesis and Verification

In the previous two chapters, we developed the algorithms for the synthesis and verification of an HISC system. The efficiency of these algorithms is dominated by the computation of the four predicate transformers: R, CR, TR and $\mathcal{CR}$. In this chapter, we first discuss how to use logic formulas to represent state subsets and transitions in a system, and then present how to compute the predicate transformers from the logic formulas. After that, we briefly introduce a method of using Reduced Ordered Binary Decision Diagram (ROBDD, or simply BDD) [6] to implement the algorithms. We then present an approach for controller implementation. Finally, we give a small tutorial example to demonstrate the algorithms.

Most of the ideas in this chapter are originally invented by Ma in [31] for the State Tree Structure (STS) or researchers in the model checking area. The approach to represent the state subsets and transitions for the HISC high-level or low-levels as logic formulas and BDD can be thought of as a special case of the approach for STS. The optimization technique using the tautology (Equation 6.2) to reduce the

intermediate BDD is also applied in our implementations. What is new here is that we simplified the logic formula representation for a flat system composed of component DES and applied these ideas and techniques for the HISC synthesis and verification algorithms.

6.1 Symbolic Representation of State Subsets and Transitions

In Chapter 2, we discussed predicates defined on the state set of a DES, but we did not discuss how to define a predicate other than its corresponding state subset. From now on, we call a predicate that are defined on the state set of a DES as a *state predicate*. A state predicate essentially is a *boolean function* whose domain is the state set of the DES and range is $\{0, 1\}$. If we use the identifying state subset to define a state predicate, then it is no better than directly using the state subset in our algorithms. In this section, for a given HISC system, we present how to define a state predicate as a logic formula, so the formula also represents a state subset. We say that the logic formula is a *symbolic* representation of the state subset.

In order to represent the high-level and low-levels in an HISC system symbolically, we also need to define predicates that are defined on the transitions of a DES, which we will call *transition predicates*. Then we will discuss how to use logic formulas to define the transition predicates.

The high-level or low-levels in an HISC system can be thought of as a special flat system which is composed of component DES: plant(s), supervisor(s)/specification(s) and interface(s). Because our symbolic representation for state subsets and transitions does not care about the type of a component DES, in this section, we only talk about

the symbolic representation for a flat system composed of component DES.

6.1.1 Symbolic Representation of State Subsets

Let the system $\mathbf{G} := (Q, \Sigma, \delta, q_0, Q_m)$ be the cross product of component DES $\mathbf{G}_1 := (Q_1, \Sigma, \delta_1, q_{1_0}, Q_{1_m}), \ldots, \mathbf{G}_n := (Q_n, \Sigma, \delta_n, q_{n_0}, Q_{n_m})$, where $n \in \{1, 2, \ldots\}$. i.e. $\mathbf{G} := \mathbf{G}_1 \times \cdots \times \mathbf{G}_n$.

Let $q \in Q$, then there exists a unique $q_1 \in Q_1, \ldots, q_n \in Q_n$ such that $q = (q_1, \ldots, q_n)$. We say q_1 is the first component part of q; $\ldots$; q_n is the n^{th} component part of q.

Definition 6.1. For the system $\mathbf{G}$ composed of components $\mathbf{G}_1, \ldots, \mathbf{G}_n$, the *state variable* $v_i (i \in \{1, \ldots, n\})$ for the i^{th} component DES $\mathbf{G}_i$ is a variable whose domain is Q_i. The *state variable vector* $\mathbf{v}$ for $\mathbf{G}$ is $(v_1, \ldots, v_n)$, where $v_1, \ldots, v_n$ are the state variables for each component.

♢

Let $i \in \{1, \ldots, n\}$ and $q_i \in Q_i$, $v_i = q_i$ returns 1 if the state variable v_i has been assigned value q_i, otherwise it returns 0. Let $A \subseteq Q$ be a state subset, then the predicate P_A for A can be written by[1]

$$P_A(\mathbf{v}) := \bigvee_{q \in A} (v_1 = q_1 \wedge v_2 = q_2 \wedge \cdots \wedge v_n = q_n). \tag{6.1}$$

For convenience, if $\mathbf{v}$ is understood, we write P_A instead of $P_A(\mathbf{v})$. Clearly, for all $q \in Q$, $q \in A$ if and only if $P_A(q) \equiv 1$. Here we use $\equiv$ to stand for logical equivalence, because $=$ has been used to test if v_i has been assigned value q_i.

[1]As a flat system composed of component DES is a special case of STS, we will not give a soundness and completeness proof for the logic formula representation, interested readers can find a proof for STS in [31].

For all $i \in \{1, \ldots, n\}$, if we treat $v_i = q_i$ as a propositional atom, then the formula generated by Equation 6.1 is a propositional logic formula. This is a very important aspect, because a predicate logic formula can not be represented as a BDD.

This formula does not look promising, but if we notice the following tautology, we will see it is possible that the formula can be greatly simplified as the following examples show.

$$\left(\bigvee_{q_i \in Q_i} (v_i = q_i) \right) \equiv 1 \tag{6.2}$$

Figure 6.1 is the simplified small factory example from [47] with buffer size two. The event set for the system is $\Sigma = \{\alpha_1, \alpha_2, \beta_1, \beta_2\}$. All the states in each component should also be added selfloops with events not appearing on the component.

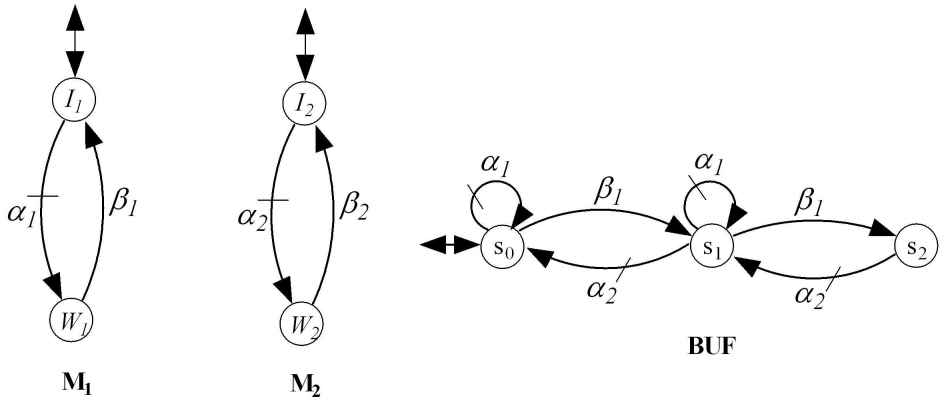


Figure 6.1: Small factory example

Let $v_{M_1}, v_{M_2}, v_{BUF}$ be the state variables for the component DES $\mathbf{M}_1, \mathbf{M}_2$ and **BUF** respectively, then the state variable vector for the small factory is $\mathbf{v} = (v_{M_1}, v_{M_2}, v_{BUF})$. Let $A \subseteq \{I_1, W_1\} \times \{I_2, W_2\} \times \{s_0, s_1, s_2\}$ be a state subset of the small factory. Let P_A be the state predicate for A.

First let $A := \{(I_1, I_2, s_0)\}$, then by Equation 6.1, we have $P_A(\mathbf{v}) := (v_{M_1} = I_1 \wedge v_{M_2} = I_2 \wedge v_{BUF} = s_0)$. This formula for the predicate P_A is not simpler than a state subset.

Now let $A := \{(I_1, I_2, s_0), (W_1, I_2, s_0), (I_1, I_2, s_1), (W_1, I_2, s_1), (I_1, I_2, s_2), (W_1, I_2, s_2)\}$, then by Equation 6.1, we have

$$\begin{aligned}
P_A(\mathbf{v}) := {} & (v_{M_1} = I_1 \wedge v_{M_2} = I_2 \wedge v_{BUF} = s_0) \vee (v_{M_1} = W_1 \wedge v_{M_2} = I_2 \wedge v_{BUF} = s_0) \vee \\
& (v_{M_1} = I_1 \wedge v_{M_2} = I_2 \wedge v_{BUF} = s_1) \vee (v_{M_1} = W_1 \wedge v_{M_2} = I_2 \wedge v_{BUF} = s_1) \vee \\
& (v_{M_1} = I_1 \wedge v_{M_2} = I_2 \wedge v_{BUF} = s_2) \vee (v_{M_1} = W_1 \wedge v_{M_2} = I_2 \wedge v_{BUF} = s_2) \\
\equiv {} & (v_{M_1} = I_1 \vee v_{M_1} = W_1) \wedge (v_{M_2} = I_2) \wedge (v_{BUF} = s_0 \vee v_{BUF} = s_1 \vee v_{BUF} = s_2), \\
& \qquad \text{by the distributive law of propositional logic} \\
\equiv {} & 1 \wedge (v_{M_2} = I_2) \wedge 1 \\
\equiv {} & (v_{M_2} = I_2)
\end{aligned}$$

We can see that $P_A(\mathbf{v}) := (v_{M_2} = I_2)$ is much simpler than the state subset A. Furthermore, the formula clearly says that A includes all the states whose $\mathbf{M_2}$ part is I_2.

With the given formula representation for a state predicate, we can easily check if a state is in the corresponding state subset. Let $P_A(\mathbf{v}) := (v_{M_2} = I_2)$ be a predicate for the small factory system. For example, the state $(I_1, I_2, s_0) \in A$ because

$$P_A(I_1, I_2, s_0) := (I_2 = I_2) \equiv 1,$$

and the state $(I_1, W_2, s_1) \notin A$ because

$$P_A(I_1, W_2, s_1) := (W_2 = I_2) \equiv 0.$$

With the formula representation for a state predicate, the set operations such as intersection, union and complement can be done by the logic operations $\wedge$, $\vee$ and $\neg$ on formulas. We now give examples for the small factory system.

Let $A_1, A_2 \subseteq \{I_1, W_1\} \times \{I_2, W_2\} \times \{s_0, s_1, s_2\}$ be two state subsets of the small factory, and

$A_1 = \{(I_1, I_2, s_0), (W_1, I_2, s_0), (I_1, I_2, s_2), (W_1, I_2, s_2)\}$,

$A_2 = \{(I_1, I_2, s_0),\ (I_1, I_2, s_1),\ (I_1, W_2, s_0),\ (I_1, W_2, s_1), (W_1, I_2, s_0),\ (W_1, I_2, s_1),$
$(W_1, W_2, s_0),\ (W_1, W_2, s_1)\}.$

Let P_{A_1} and P_{A_2} be the state predicates for A_1 and A_2 respectively, then by Equation 6.1, we have the simplified formulas as

$$P_{A_1}(\mathbf{v}) := (v_{M_2} = I_2) \wedge (v_{BUF} = s_0 \vee v_{BUF} = s_2)$$
$$P_{A_2}(\mathbf{v}) := (v_{BUF} = s_0 \vee v_{BUF} = s_1).$$

- $A_1 \cap A_2$

  By set intersection, $A_1 \cap A_2 = \{(I_1, I_2, s_0),\ (W_1, I_2, s_0)\}$, and

  $P_{A_1} \wedge P_{A_2} \equiv (v_{M_2} = I_2) \wedge (v_{BUF} = s_0 \vee v_{BUF} = s_2) \wedge (v_{BUF} = s_0 \vee v_{BUF} = s_1)$

  $\equiv (v_{M_2} = I_2) \wedge (v_{BUF} = s_0).$

  One can see that $P_{A_1} \wedge P_{A_2}$ really represents the state subset $A_1 \cap A_2$.

- $A_1 \cup A_2$

  By set union,

  $A_1 \cup A_2 = \{(I_1, I_2, s_0), (W_1, I_2, s_0), (I_1, I_2, s_2), (W_1, I_2, s_2), (I_1, I_2, s_1), (I_1, W_2, s_0),$
  $(I_1, W_2, s_1),\ (W_1, I_2, s_1),\ (W_1, W_2, s_0),\ (W_1, W_2, s_1)\},$

  and

  $P_{A_1} \vee P_{A_2} \equiv ((v_{M_2} = I_2) \wedge (v_{BUF} = s_0 \vee v_{BUF} = s_2)) \vee (v_{BUF} = s_0 \vee v_{BUF} = s_1)$

  $\equiv (v_{M_2} = I_2) \vee (v_{BUF} = s_0 \vee v_{BUF} = s_1).$

  $P_{A_1} \vee P_{A_2}$ also represents the state subset $A_1 \cup A_2$.

- $\overline{A_2}$

  By set complement,

  $\overline{A_2} = \{(I_1, I_2, s_2), (I_1, W_2, s_2), (W_1, I_2, s_2), (W_1, W_2, s_2)\},$

and

$$\neg P_{A_2} \equiv \neg(v_{BUF} = s_0 \vee v_{BUF} = s_1)$$

$$\equiv (v_{BUF} = s_2).$$

So, $\neg P_{A_2}$ represents the state subset $\overline{A_2}$.

6.1.2 Symbolic Representation of Transitions

Let $\mathbf{G} := (Q, \Sigma, \delta, q_0, Q_m)$ be the system in Section 6.1.1 composed of $\mathbf{G}_1, \ldots, \mathbf{G}_n$. For a given event $\sigma \in \Sigma$, the transition function δ is essentially a subset of the set $Q \times Q$.

Definition 6.2. For the system $\mathbf{G}$ composed of components $\mathbf{G}_1, \ldots, \mathbf{G}_n$, for a given event $\sigma \in \Sigma$, a *transition predicate* N_σ defined on $Q \times Q$ is a boolean function $N_\sigma : Q \times Q \to \{1, 0\}$ according to

$$(\forall q, q' \in Q)\ N_\sigma(q, q') := \begin{cases} 1, & \delta(q,\sigma)! \ \&\ \delta(q,\sigma) = q' \\ 0, & \text{otherwise.} \end{cases}$$

$\diamondsuit$

Let $\sigma \in \Sigma$ be an event, then N_σ identifies all the transitions for σ in $\mathbf{G}$. Like the representation of a state subset, we can also use the a logic formula to represent the subset of $Q \times Q$ (thus a transition predicate), but now we need two different sets of state variables for each component DES. One variable will be for the source state of a transition, the other will be for the target state of a transition.

Definition 6.3. For the system $\mathbf{G}$ composed of components $\mathbf{G}_1, \ldots, \mathbf{G}_n$, the *normal state variable* $v_i, (i \in \{1, \ldots, n\})$ and the *prime state variable* v_i' for each component DES $\mathbf{G}_i$ are two variables whose domains both are Q_i. The *normal state variable*

vector $\mathbf{v}$ is $(v_1, \ldots, v_n)$ and the *prime state variable vector* $\mathbf{v}'$ is $(v'_1, \ldots, v'_n)$, where $v_1, \ldots, v_n$ are normal state variables for each component in $\mathbf{G}$, and $v'_1, \ldots, v'_n$ are prime state variables for each component in $\mathbf{G}$.

$\diamondsuit$

For a given event $\sigma \in \Sigma$ in the system $\mathbf{G}$ composed of components $\mathbf{G}_1, \ldots, \mathbf{G}_n$, the transition predicate for N_σ can be written as

$$N_\sigma(\mathbf{v}, \mathbf{v}') := \bigwedge_{i \in \{1,2,\ldots,n\}} \left(\bigvee_{\substack{(q_i, q'_i) \in Q_i \times Q_i, \\ \delta_i(q_i, \sigma) = q'_i}} (v_i = q_i \wedge v'_i = q'_i) \right). \tag{6.3}$$

Note the following fact. Let $i \in \{1, \ldots, n\}$ and $\sigma \in \Sigma$. If for all $q_i \in Q_i, \delta_i(q_i, \sigma) \not\downarrow$, then

$$\left(\bigvee_{\substack{(q_i, q'_i) \in Q_i \times Q_i, \\ \delta_i(q_i, \sigma) = q'_i}} (v_i = q_i \wedge v'_i = q'_i) \right) \equiv 0$$

Therefore, $N_\sigma(\mathbf{v}, \mathbf{v}') \equiv 0$. This is in line with that the system will block an event if one of the component DES blocks the event.

Equation 6.3 assumes that every component DES is defined on the event set Σ. However, when we design a system with multiple component DES, each component DES is defined on its own event set. The system event set is the union of all the component event set and the system is the synchronous product of all the component DES. Therefore, Equation 6.3 requires that each component DES must be selfflooped with all the events that are not in its own event set, on each state. This not only creates a lot of clutter, but also makes the formula to represent the transitions much more complicated.

To solve this problem, for each $\sigma \in \Sigma$, we use the *transition tuple* $(\boldsymbol{v}_\sigma, \boldsymbol{v}'_\sigma, N_\sigma)$ to

represent the transitions on σ, where $\boldsymbol{v}_\sigma$ is a *normal state variable set* including all the normal state variables whose corresponding component DES event set contains σ, and $\boldsymbol{v}'_\sigma$ is a *prime state variable set* including all the corresponding prime state variables of the normal state variables in $\boldsymbol{v}_\sigma$. For those state variables that are not in the set $\boldsymbol{v}_\sigma$, we know that σ must be selflooped on each state of their corresponding component DES.

For the system $\mathbf{G}$ composed of components $\mathbf{G}_1, \ldots, \mathbf{G}_n$, let $\mathbf{G}'_1 := (Q_1, \Sigma_1, \delta'_1, q_{1_0}, Q_{1_m}), \ldots, \mathbf{G}'_n := (Q_n, \Sigma_n, \delta'_n, q_{n_0}, Q_{n_m})$ be the component DES such that $\mathbf{G}_1 = \mathbf{selfloop}(\mathbf{G}'_1, \Sigma - \Sigma_1), \ldots, \mathbf{G}_n = \mathbf{selfloop}(\mathbf{G}'_n, \Sigma - \Sigma_n)$. So, we have $\mathbf{G} = \mathbf{G}'_1 \parallel \cdots \parallel \mathbf{G}'_n$ and $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_n$.

For a given event $\sigma \in \Sigma$ in the system $\mathbf{G}$ with components $\mathbf{G}'_1, \ldots, \mathbf{G}'_n$, the tuple $(\boldsymbol{v}_\sigma, \boldsymbol{v}'_\sigma, N_\sigma)$ can be written by [2]

$$\boldsymbol{v}_\sigma := \{v_i \in \mathbf{v} | \sigma \in \Sigma_i\}, \quad \boldsymbol{v}'_\sigma := \{v'_i \in \mathbf{v}' | \sigma \in \Sigma_i\} \tag{6.4}$$

$$N_\sigma(\mathbf{v}, \mathbf{v}') := \bigwedge_{\{i \in \{1,2,\ldots,n\} | \sigma \in \Sigma_i\}} \left(\bigvee_{\substack{(q_i, q'_i) \in Q_i \times Q_i, \\ \delta'_i(q_i, \sigma) = q'_i}} (v_i = q_i \wedge v'_i = q'_i) \right). \tag{6.5}$$

Although the tuple $(\boldsymbol{v}_\sigma, \boldsymbol{v}'_\sigma, N_\sigma)$ are created from unselflooped components, note that the tuple actually expresses the selfloop information, which means that by creating the tuple we automatically selfloop σ for each component whose own event set does not contain σ. Note also that Equation 6.4 and 6.5 can be used for creating the transition tuples for selflooped components as well.

We now create the transition tuples for all the events in the small factory example shown in Figure 6.1 to demonstrate how to use Equation 6.4 and 6.5.

[2] Here we are abusing the notation a bit, as $\mathbf{v}$ and $\mathbf{v}'$ should not be treated as state variable sets.

- $(\boldsymbol{v}_{\alpha_1}, \boldsymbol{v}'_{\alpha_1}, N_{\alpha_1})$

  $\boldsymbol{v}_{\alpha_1} := \{v_{M_1}, v_{BUF}\},\ \boldsymbol{v}'_{\alpha_1} := \{v'_{M_1}, v'_{BUF}\}$

  $N_{\alpha_1} := (v_{M_1} = I_1 \wedge v'_{M_1} = W_1) \wedge$
  $((v_{BUF} = s_0 \wedge v'_{BUF} = s_0) \vee (v_{BUF} = s_1 \wedge v'_{BUF} = s_1))$

- $(\boldsymbol{v}_{\beta_1}, \boldsymbol{v}'_{\beta_1}, N_{\beta_1})$

  $\boldsymbol{v}_{\beta_1} := \{v_{M_1}, v_{BUF}\}, \boldsymbol{v}'_{\beta_1} := \{v'_{M_1}, v'_{BUF}\}$

  $N_{\beta_1} := (v_{M_1} = W_1 \wedge v'_{M_1} = I_1) \wedge$
  $((v_{BUF} = s_0 \wedge v'_{BUF} = s_1) \vee (v_{BUF} = s_1 \wedge v'_{BUF} = s_2))$

- $(\boldsymbol{v}_{\alpha_2}, \boldsymbol{v}'_{\alpha_2}, N_{\alpha_2})$

  $\boldsymbol{v}_{\alpha_2} := \{v_{M_2}, v_{BUF}\}, \boldsymbol{v}'_{\alpha_2} := \{v'_{M_2}, v'_{BUF}\}$

  $N_{\alpha_2} := (v_{M_2} = I_2 \wedge v'_{M_2} = W_2) \wedge$
  $((v_{BUF} = s_1 \wedge v'_{BUF} = s_0) \vee (v_{BUF} = s_2 \wedge v'_{BUF} = s_1))$

- $(\boldsymbol{v}_{\beta_2}, \boldsymbol{v}'_{\beta_2}, N_{\beta_2})$

  $\boldsymbol{v}_{\beta_2} := \{v_{M_2}\}, \boldsymbol{v}'_{\beta_2} := \{v'_{M_2}\}$

  $N_{\beta_2} := (v_{M_2} = W_2 \wedge v'_{M_2} = I_2)$

6.2 Symbolic Computation of Predicate Transformers

With the logic formula representation for state subsets and transitions, we now talk about how to compute the four predicate transformers TR, R, CR and $\mathcal{CR}$. Because CR is a special case of $\mathcal{CR}$, we only discuss $\mathcal{CR}$ in this section.

In this section, we will use the system $\mathbf{G}$ composed of components $\mathbf{G}'_1, \ldots, \mathbf{G}'_n$ defined in Section 6.1.2.

6.2.1 Computation of Transition and Inverse Transition

For the system $\mathbf{G} := (Q, \Sigma, \delta, q_0, Q_m)$ composed of components $\mathbf{G}'_1, \ldots, \mathbf{G}'_n$, to compute the predicate transformer R, we must know how to compute $\delta(q, \sigma)$, where $q \in Q$ and $\sigma \in \Sigma$. A state predicate $P \in Pred(Q)$ can represent a state subset $Q_P \subseteq Q$. To compute all the states that can be reached by a σ transition from any state in Q_P, we can compute $\cup_{q \in Q_P}\{\delta(q, \sigma)\}$. However, this method is rather time consuming, especially when the state space of a system is large. We hope we can instead directly compute the function $\widehat{\delta} : Pred(Q) \times \Sigma \to Pred(Q)$, defined as follows:

$$(\forall P \in Pred(Q))(\forall \sigma \in \Sigma)\ \widehat{\delta}(P, \sigma) := pr(\{q' \in Q \mid (\exists q \models P)\ \delta(q, \sigma) = q'\}).$$

For the system $\mathbf{G}$, to compute the predicate transformers TR and $\mathcal{CR}$, we define a function $\delta^{-1} : Q \times \Sigma \to Pwr(Q)$ according to

$$(\forall q \in Q)(\sigma \in \Sigma)\ \delta^{-1}(q, \sigma) := \{q' \in Q \mid \delta(q', \sigma) = q\}.$$

Let $P \in Pred(Q)$ be a predicate. To compute all the states that have a σ transition to any state satisfying P, we can compute $\cup_{q \models P}\delta^{-1}(q, \sigma)$. Similarly, we hope we can instead directly compute the function $\widehat{\delta^{-1}} : Pred(Q) \times \Sigma \to Pred(Q)$, defined as follows:

$$(\forall P \in Pred(Q))(\forall \sigma \in \Sigma)\ \widehat{\delta^{-1}}(P, \sigma) := pr(\{q \in Q \mid \delta(q, \sigma) \models P\}).$$

As in [31], we will use a method similar to the *relational product*, developed by the researchers in the model checking area [11], to compute $\widehat{\delta}(P, \sigma)$ and $\widehat{\delta^{-1}}(P, \sigma)$.

Definition 6.4. For the system $\mathbf{G}$ composed of components $\mathbf{G}'_1, \ldots, \mathbf{G}'_n$, for all $\sigma \in \Sigma$, let $(\boldsymbol{v}_\sigma, \boldsymbol{v}'_\sigma, N_\sigma)$ be the transition tuple for σ in $\mathbf{G}$.

For all $i \in \{1, \ldots, n\}$, if $v_i \in \boldsymbol{v}_\sigma, v'_i \in \boldsymbol{v}'_\sigma$, then define $\exists v_i N_\sigma$ and $\exists v'_i N_\sigma$ as

$$\exists v_i N_\sigma := \bigvee_{q_i \in Q_i} N_\sigma[q_i/v_i],$$

$$\exists v'_i N_\sigma := \bigvee_{q_i \in Q_i} N_\sigma[q_i/v'_i],$$

where $N_\sigma[q_i/v_i]$ is the resulting predicate by assigning q_i to v_i; $N_\sigma[q_i/v'_i]$ is the resulting predicate by assigning q_i to v'_i.

$\diamondsuit$

The above definition of $\exists v_i N_\sigma$ and $\exists v'_i N_\sigma$ is based on the *existential quantifier elimination* method for finite domain [2]. That is, $\exists v_i$ and $\exists v'_i$ eliminate the variable v_i and v'_i from N_σ respectively. The results of $\exists v_i N_\sigma$ and $\exists v'_i N_\sigma$ are still propositional logic formulas, which can be represented by BDD.

Let $\sigma \in \Sigma$ and $(\boldsymbol{v}_\sigma, \boldsymbol{v}'_\sigma, N_\sigma)$ be the transition tuple for σ in $\mathbf{G}$. Assume $\boldsymbol{v}_\sigma = \{\widehat{v}_1, \widehat{v}_2, \ldots \widehat{v}_m\}$ and $\boldsymbol{v}'_\sigma = \{\widehat{v}'_1, \widehat{v}'_2, \ldots \widehat{v}'_m\}$, where $m \in \{1, \ldots, n\}$. For convenience, we define the shorthand $\exists \boldsymbol{v}_\sigma N_\sigma$ and $\exists \boldsymbol{v}'_\sigma N_\sigma$ as follows:

$$\exists \boldsymbol{v}_\sigma N_\sigma := \exists \widehat{v}_1 (\exists \widehat{v}_2 \cdots (\exists \widehat{v}_m N_\sigma) \cdots),$$
$$\exists \boldsymbol{v}'_\sigma N_\sigma := \exists \widehat{v}'_1 (\exists \widehat{v}'_2 \cdots (\exists \widehat{v}'_m N_\sigma) \cdots).$$

The resulting logic formula for $\exists \boldsymbol{v}_\sigma N_\sigma$ contains only the prime variables in $\boldsymbol{v}'_\sigma$. If we replace all the prime variables as normal variables, denoted it as $\exists \boldsymbol{v}_\sigma N_\sigma[\boldsymbol{v}'_\sigma \to \boldsymbol{v}_\sigma]$, then the resulting predicate $\exists \boldsymbol{v}_\sigma N_\sigma[\boldsymbol{v}'_\sigma \to \boldsymbol{v}_\sigma]$ represents the *target* state set of all the σ transitions in $\mathbf{G}$ (i.e. Each state in the target state set has a σ transition entering it). We need to do the variable replacement because we use the normal state variables to express the logic formula of a state subset predicate.

As the resulting logic formula for $\exists \boldsymbol{v}'_\sigma N_\sigma$ contains only the normal variables in $\boldsymbol{v}_\sigma$, we do not need to do the variable replacement. $\exists \boldsymbol{v}'_\sigma N_\sigma$ represents the *source* state set of all the σ transitions in $\mathbf{G}$ (i.e. Each state in the target state set has a σ transition leaving it).

We now give an example to demonstrate $\exists \boldsymbol{v}_\sigma N_\sigma$ and $\exists \boldsymbol{v}'_\sigma N_\sigma$. Figure 6.2 shows a system with only one component $\mathbf{G}_E$. Let v_E, v'_E be the normal and prime state variable for the component $\mathbf{G}_E$.

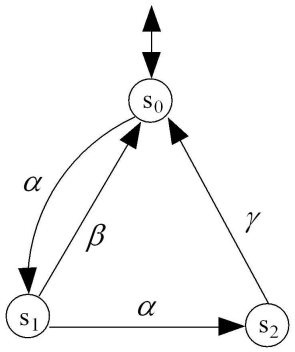


Figure 6.2: Example DES $\mathbf{G}_E$

- $\exists \boldsymbol{v}_\alpha N_\alpha$, $\exists \boldsymbol{v}'_\alpha N_\alpha$

  For the event α, we have

  $\boldsymbol{v}_\alpha := \{v_E\},\ \boldsymbol{v}'_\alpha := \{v'_E\}$

  $N_\alpha := (v_E = s_0 \wedge v'_E = s_1) \vee (v_E = s_1 \wedge v'_E = s_2)$.

  $\exists \boldsymbol{v}_\alpha N_\alpha := \exists v_E N_\alpha$

  $$\equiv (s_0 = s_0 \wedge v'_E = s_1) \vee (s_0 = s_1 \wedge v'_E = s_2) \vee$$
  $$(s_1 = s_0 \wedge v'_E = s_1) \vee (s_1 = s_1 \wedge v'_E = s_2) \vee$$
  $$(s_2 = s_0 \wedge v'_E = s_1) \vee (s_2 = s_1 \wedge v'_E = s_2)$$
  $$\equiv v'_E = s_1 \vee v'_E = s_2$$

$\exists \boldsymbol{v}_\alpha N_\alpha[\boldsymbol{v}'_\alpha \rightarrow \boldsymbol{v}_\alpha] := v_E = s_1 \vee v_E = s_2$.

The logic formula $v_E = s_1 \vee v_E = s_2$ represents the state subset $\{s_1, s_2\}$, which exactly includes the target states of all the α transitions in $\mathbf{G}_E$.

$\exists \boldsymbol{v}'_\alpha N_\alpha := \exists v'_E N_\alpha$

$\equiv (v_E = s_0 \wedge s_0 = s_1) \vee (v_E = s_1 \wedge s_0 = s_2) \vee$

$(v_E = s_0 \wedge s_1 = s_1) \vee (v_E = s_1 \wedge s_1 = s_2) \vee$

$(v_E = s_0 \wedge s_2 = s_1) \vee (v_E = s_1 \wedge s_2 = s_2)$

$\equiv v_E = s_0 \vee v_E = s_1$

The logic formula $v_E = s_0 \vee v_E = s_1$ represents the state subset $\{s_0, s_1\}$, which exactly includes the source states of all the α transitions in $\mathbf{G}_E$.

- $\exists \boldsymbol{v}_\beta N_\beta$, $\exists \boldsymbol{v}'_\beta N_\beta$

  For the event β, we have

  $\boldsymbol{v}_\beta := \{v_E\},\ \boldsymbol{v}'_\beta := \{v'_E\}$

  $N_\beta := (v_E = s_1 \wedge v'_E = s_0)$.

  $\exists \boldsymbol{v}_\beta N_\beta := \exists v_E N_\beta$

  $\equiv (s_0 = s_1 \wedge v'_E = s_0) \vee (s_1 = s_1 \wedge v'_E = s_0) \vee (s_2 = s_1 \wedge v'_E = s_0)$

  $\equiv v'_E = s_0$

  $\exists \boldsymbol{v}_\beta N_\beta[\boldsymbol{v}'_\beta \rightarrow \boldsymbol{v}_\beta] := v_E = s_0$.

  $\exists \boldsymbol{v}'_\beta N_\beta := \exists v'_E N_\beta$

  $\equiv (v_E = s_1 \wedge s_0 = s_0) \vee (v_E = s_1 \wedge s_1 = s_0) \vee (v_E = s_1 \wedge s_2 = s_0)$

  $\equiv v_E = s_1$

We now give an example to compute the state set consisting of target states of all the β_1 transitions in the small factory example shown in Figure 6.1.

$\boldsymbol{v}_{\beta_1} := \{v_{M_1}, v_{BUF}\},\ \boldsymbol{v}'_{\beta_1} := \{v'_{M_1}, v'_{BUF}\}.$

$$N_{\beta_1} := (v_{M_1} = W_1 \wedge v'_{M_1} = I_1) \wedge$$
$$((v_{BUF} = s_0 \wedge v'_{BUF} = s_1) \vee (v_{BUF} = s_1 \wedge v'_{BUF} = s_2))$$

$$\exists \boldsymbol{v}_{\beta_1} N_{\beta_1} := \exists v_{M_1}(\exists v_{BUF} N_{\beta_1})$$
$$\equiv \exists v_{M_1}((v_{M_1} = W_1 \wedge v'_{M_1} = I_1) \wedge$$
$$((s_0 = s_0 \wedge v'_{BUF} = s_1) \vee (s_0 = s_1 \wedge v'_{BUF} = s_2) \vee$$
$$(s_1 = s_0 \wedge v'_{BUF} = s_1) \vee (s_1 = s_1 \wedge v'_{BUF} = s_2) \vee$$
$$(s_2 = s_0 \wedge v'_{BUF} = s_1) \vee (s_2 = s_1 \wedge v'_{BUF} = s_2)))$$
$$\equiv \exists v_{M_1}((v_{M_1} = W_1 \wedge v'_{M_1} = I_1) \wedge (v'_{BUF} = s_1 \vee v'_{BUF} = s_2))$$
$$\equiv ((I_1 = W_1 \wedge v'_{M_1} = I_1) \vee (W_1 = W_1 \wedge v'_{M_1} = I_1)) \wedge$$
$$(v'_{BUF} = s_1 \vee v'_{BUF} = s_2)$$
$$\equiv (v'_{M_1} = I_1) \wedge (v'_{BUF} = s_1 \vee v'_{BUF} = s_2)$$

$$\exists \boldsymbol{v}_{\beta_1} N_{\beta_1}[\boldsymbol{v}'_{\beta_1} \rightarrow \boldsymbol{v}_{\beta_1}] := (v_{M_1} = I_1) \wedge (v_{BUF} = s_1 \vee v_{BUF} = s_2).$$

For the system $\mathbf{G}$ composed of components $\mathbf{G}'_1, \ldots, \mathbf{G}'_n$, let $\sigma \in \Sigma$. From the above definitions and examples, it is clear that $\exists \boldsymbol{v}_\sigma N_\sigma[\boldsymbol{v}'_\sigma \rightarrow \boldsymbol{v}_\sigma]$ exactly computes the predicate representing the state subset $\{q' \in Q \mid (\exists q \in Q)\ \delta(q, \sigma) = q'\}$ and $\exists \boldsymbol{v}'_\sigma N_\sigma$ exactly computes the predicate representing the state subset $\{q \in Q \mid \delta(q, \sigma)!\}$.

With the help of existential quantifier elimination method, we now can compute $\widehat{\delta}$ and $\widehat{\delta^{-1}}$ symbolically. For system $\mathbf{G}$, let $P \in Pred(Q)$ and $\sigma \in \Sigma$. $\widehat{\delta}(P, \sigma)$ requires computing the predicate representing the state subset $\{q' \in Q | (\exists q \models P)\ \delta(q, \sigma) = q'\}$, while $\widehat{\delta^{-1}}(P, \sigma)$ requires computing the predicate representing the state subset

$\{q \in Q | \delta(q, \sigma) \models P\}$. So, $\widehat{\delta}(P, \sigma)$ and $\widehat{\delta^{-1}}(P, \sigma)$ can be computed as follows:

$$\widehat{\delta}(P, \sigma) := (\exists \boldsymbol{v}_\sigma (N_\sigma \wedge P))[\boldsymbol{v}'_\sigma \rightarrow \boldsymbol{v}_\sigma] \tag{6.6}$$

$$\widehat{\delta^{-1}}(P, \sigma) := \exists \boldsymbol{v}'_\sigma (N_\sigma \wedge (P[\boldsymbol{v}_\sigma \rightarrow \boldsymbol{v}'_\sigma])) \tag{6.7}$$

In Equation 6.6, by first computing $N_\sigma \wedge P$, we restrict the σ transitions to those whose source states satisfy P.

In Equation 6.7, note that P is first transformed to the prime variable format, because we want to restrict the σ transitions to those whose target states satisfy P.

For convenience, we omit the hat on $\widehat{\delta}(P, \sigma)$ and $\widehat{\delta^{-1}}(P, \sigma)$, because from the parameters we can clearly tell what the functions are.

We now give examples again for the small factory example shown in Figure 6.1 to demonstrate the computing of $\delta(P, \sigma)$ and $\delta^{-1}(P, \sigma)$.

Let P be a state predicate defined on the state set of the small factory example shown in Figure 6.1, and let $P := (v_{M_1} = W_1 \wedge v_{BUF} = s_1)$. We now demonstrate how to compute $\delta(P, \alpha_2)$ and $\delta^{-1}(P, \alpha_1)$.

- $\delta(P, \alpha_2)$

$$\begin{aligned} N_{\alpha_2} \wedge P := {} & (v_{M_2} = I_2 \wedge v'_{M_2} = W_2) \wedge \\ & ((v_{BUF} = s_1 \wedge v'_{BUF} = s_0) \vee (v_{BUF} = s_2 \wedge v'_{BUF} = s_1)) \wedge \\ & (v_{M_1} = W_1 \wedge v_{BUF} = s_1). \\ \equiv {} & (v_{M_2} = I_2 \wedge v'_{M_2} = W_2) \wedge \\ & (v_{BUF} = s_1 \wedge v'_{BUF} = s_0 \wedge v_{M_1} = W_1). \end{aligned}$$

$$\begin{aligned} \exists \boldsymbol{v}_{\alpha_2} (N_{\alpha_2} \wedge P) := {} & \exists v_{M_2} (\exists v_{BUF} ((v_{M_2} = I_2 \wedge v'_{M_2} = W_2) \wedge \\ & \qquad (v_{BUF} = s_1 \wedge v'_{BUF} = s_0 \wedge v_{M_1} = W_1))) \\ \equiv {} & \exists v_{M_2} ((v_{M_2} = I_2 \wedge v'_{M_2} = W_2) \wedge (v'_{BUF} = s_0 \wedge v_{M_1} = W_1)) \end{aligned}$$

$$\equiv (v'_{M_2} = W_2) \wedge (v'_{BUF} = s_0) \wedge (v_{M_1} = W_1)$$

$\delta(P, \alpha_2) := (\exists \boldsymbol{v}_{\alpha_2}(N_{\alpha_2} \wedge P))[\boldsymbol{v}'_{\alpha_2} \rightarrow \boldsymbol{v}_{\alpha_2}]$

$$\equiv ((v'_{M_2} = W_2) \wedge (v'_{BUF} = s_0) \wedge (v_{M_1} = W_1))[\boldsymbol{v}'_{\alpha_2} \rightarrow \boldsymbol{v}_{\alpha_2}]$$

$$\equiv (v_{M_2} = W_2) \wedge (v_{BUF} = s_0) \wedge (v_{M_1} = W_1)$$

- $\delta^{-1}(P, \alpha_1)$

$$N_{\alpha_1} \wedge (P[\boldsymbol{v}_{\alpha_1} \rightarrow \boldsymbol{v}'_{\alpha_1}]) := (v_{M_1} = I_1 \wedge v'_{M_1} = W_1) \wedge ((v_{BUF} = s_0 \wedge v'_{BUF} = s_0) \vee (v_{BUF} = s_1 \wedge v'_{BUF} = s_1)) \wedge (v'_{M_1} = W_1 \wedge v'_{BUF} = s_1)$$

$$\equiv (v_{M_1} = I_1) \wedge (v'_{M_1} = W_1) \wedge (v_{BUF} = s_1) \wedge (v'_{BUF} = s_1)$$

$\delta^{-1}(P, \alpha_1) := \exists \boldsymbol{v}'_{\alpha_1}(N_{\alpha_1} \wedge (P[\boldsymbol{v}_{\alpha_1} \rightarrow \boldsymbol{v}'_{\alpha_1}]))$

$$\equiv \exists v'_{M_1}(\exists v'_{BUF}((v_{M_1} = I_1) \wedge (v'_{M_1} = W_1) \wedge (v_{BUF} = s_1) \wedge (v'_{BUF} = s_1)))$$

$$\equiv (v_{M_1} = I_1) \wedge (v_{BUF} = s_1)$$

6.2.2 Computation of R

For the system $\mathbf{G}$ composed of components $\mathbf{G}'_1, \ldots, \mathbf{G}'_n$, let $P \in Pred(Q)$. From the definition of $R(\mathbf{G}, .)$, we can write a straightforward algorithm for $R(\mathbf{G}, P)$ as shown in Algorithm 6.1.

Algorithm 6.1 $R(\mathbf{G}, P)$(Straightforward)

1: $P_1 \leftarrow P \wedge pr(\{q_0\})$;
2: **repeat**
3: $\quad P_2 \leftarrow P_1$;
4: $\quad P_1 \leftarrow P_1 \vee \left(\bigvee_{\sigma \in \Sigma}(\delta(P_1, \sigma) \wedge P)\right)$;
5: **until** $P_1 \equiv P_2$
6: return P_1;

In Line 4 of the algorithm, we never remove states from P_1. As the number of states in $\mathbf{G}$ is finite, after finite number of steps we will have $P1 \equiv P2$; thus the algorithm will terminate.

Algorithm 6.1 works like a breadth first search. As found by Ma in [31], a breath first search method could generate a very complicated intermediate result, even though the final result is quite simple because of the tautology in Equation 6.2. An algorithm (shown below) working like a depth first search can greatly reduce the intermediate result. The following example is almost directly taken from [31] for the sake of content completeness, the difference is that the example there is used for explaining the algorithm of a predicate transformer similar to our TR predicate transformer.

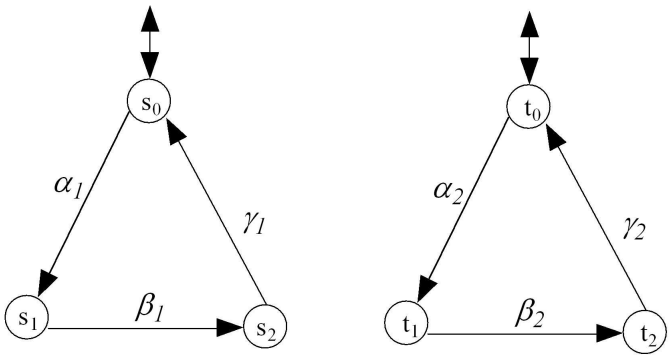


Figure 6.3: Example system $\mathbf{M}$

Figure 6.3 shows a system $\mathbf{M}$ composed of two component DES $\mathbf{M}_1$ and $\mathbf{M}_2$. For simplicity, let $P := true$ be the input predicate of $R(\mathbf{M}, .)$, i.e. we now compute $R(\mathbf{M}, true)$. Let v_{M_1} and v_{M_2} be the state variables for $\mathbf{M}_1$ and $\mathbf{M}_2$ respectively.

Initially, $P_1 \equiv (v_{M_1} = s_0 \wedge v_{M_2} = t_0)$;

- First iteration

$$P_1 := P_1 \vee (v_{M_1} = s_1 \wedge v_{M_2} = t_0) \vee (v_{M_1} = s_0 \wedge v_{M_2} = t_1)$$

$$\equiv (v_{M_1} = s_1 \wedge v_{M_2} = t_0) \vee (v_{M_1} = s_0 \wedge (v_{M_2} = t_0 \vee v_{M_2} = t_1))$$

- Second iteration

$$P_1 := P_1 \vee (v_{M_1} = s_1 \wedge v_{M_2} = t_1) \vee (v_{M_1} = s_2 \wedge v_{M_2} = t_0) \vee (v_{M_1} = s_0 \wedge v_{M_2} = t_2)$$

$$\equiv (v_{M_1} = s_0) \vee (v_{M_1} = s_1 \wedge (v_{M_2} = t_0 \vee v_{M_2} = t_1)) \vee (v_{M_1} = s_2 \wedge v_{M_2} = t_0)$$

- Third iteration

$$P_1 := P_1 \vee (v_{M_1} = s_2 \wedge v_{M_2} = t_1) \vee (v_{M_1} = s_1 \wedge v_{M_2} = t_2)$$

$$\equiv (v_{M_1} = s_0) \vee (v_{M_1} = s_1) \vee (v_{M_1} = s_2 \wedge (v_{M_2} = t_0 \vee v_{M_2} = t_1))$$

$$\equiv (v_{M_1} = s_0) \vee (v_{M_1} = s_1) \vee (v_{M_1} = s_2 \wedge (v_{M_2} = t_0 \vee v_{M_2} = t_1))$$

- Fourth iteration

$$P_1 := P_1 \vee (v_{M_1} = s_2 \wedge v_{M_2} = t_2)$$

$$\equiv 1$$

Although the result can be represented as 1, one can see that the intermediate logic formula is much more complicated than the result during the computation.

To reduce the intermediate result, we can take a look at equation 6.2. Only when a state variable can be assigned to all the values in its domain, the part related to the state variable can be reduced to 1. In the above example, for instance, if we first compute the transitions $\alpha_1, \beta_1, \gamma_1$, then v_{M_1} can be assigned to all the states in its domain $\{s_0, s_1, s_2\}$, so this part can be reduced to 1.

For the system $\mathbf{G}$ composed of components $\mathbf{G}'_1, \ldots, \mathbf{G}'_n$, Algorithm 6.2 formally states the above idea. The algorithm chooses the events component by component.

Note that in Line 7, the event set is Σ_i, but the transition function is δ. This algorithm makes great use of the tautology of Equation 6.2 to reduce the intermediate result. The difference between this algorithm and the previous one is the order of

Algorithm 6.2 $R(\mathbf{G}, P)$

1: $P_1 \leftarrow P \wedge pr(\{q_0\})$;
2: **repeat**
3: $P_2 \leftarrow P_1$;
4: **for** $i \leftarrow 1$ to n **do**
5: **repeat**
6: $P_3 \leftarrow P_1$;
7: $P_1 \leftarrow P_1 \vee \left(\bigvee_{\sigma \in \Sigma_i}(\delta(P_1, \sigma) \wedge P)\right)$;
8: **until** $P_1 \equiv P_3$
9: **end for**
10: **until** $P_1 \equiv P_2$
11: return P_1;

choosing events. In Line 7 of the algorithm, we never remove states from P_1. As the number of states in $\mathbf{G}$ is finite, after finite number of steps we will have $P1 \equiv P3$ for the `repeat` loop from Line 5 to Line 8 and $P1 \equiv P2$ for the `repeat` loop from Line 2 to Line 10; thus the algorithm will terminate.

Now we compute $R(\mathbf{M}, true)$ according to Algorithm 6.2, where $\mathbf{M}$ is shown in Figure 6.3. The phrase "outer `repeat`" means the one on Line 2, and "inner `repeat`" means the one on Line 5.

Initially, $P_1 \equiv (v_{M_1} = s_0 \wedge v_{M_2} = t_0)$;

- First outer `repeat` iteration

  $i := 1$; // For the component $\mathbf{M}_1$

  - First inner `repeat` iteration

    $P_1 := P_1 \vee (v_{M_1} = s_1 \wedge v_{M_2} = t_0)$

    $\equiv (v_{M_1} = s_0 \vee v_{M_1} = s_1) \wedge v_{M_2} = t_0$
  - Second inner `repeat` iteration

    $P_1 := P_1 \vee (v_{M_1} = s_2 \wedge v_{M_2} = t_0)$

    $\equiv v_{M_2} = t_0$
  - Third inner `repeat` iteration

    $P_1 := v_{M_2} = t_0$

  $i := 2$; // For the component $\mathbf{M}_2$

  - First inner `repeat` iteration
    $P_1 := P_1 \vee (v_{M_2} = t_1)$
    $\equiv (v_{M_2} = t_0 \vee v_{M_2} = t_1)$
  - Second inner `repeat` iteration
    $P_1 := P_1 \vee (v_{M_2} = t_2)$
    $\equiv 1$
  - Third inner `repeat` iteration
    $P_1 := 1$

- Second outer `repeat` iteration
  $P_1 := 1$

During the computation, we can see the intermediate predicates are much shorter than those of the previous algorithm. This is very important for our BDD-based algorithms, because the running time of BDD operations directly depends on the number of BDD nodes, and the number of nodes a BDD contains is proportional to the length of its corresponding logic formula (simplified).

For the system $\mathbf{G}$ composed of components $\mathbf{G}'_1, \ldots, \mathbf{G}'_n$, let $P \in Pred(Q)$. We now discuss another optimization technique for the computation of $R(\mathbf{G}, P)$.

Notice that during the computation of $R(\mathbf{G}, P)$, we always compute the new reachable states from all the states that have been reached. This is obviously not efficient in an automata-based algorithm, because we can compute the new reachable states from the states that were found most recently (or in last loop). In our symbolic algorithms, however, computing from all the states that have reached could make the algorithm run faster because the tautology could dramatically reduce the size of the intermediate logic formulas.

However, in our experience, under some cases, computing the new reachable states from the states most recently found is faster than computing them from all the states that have been reached. For instance, in the AIP example in next chapter, computing from the most recently found states is faster.

Algorithm 6.3 shows how to compute $R(\mathbf{G}, P)$ from the most recently found states. Again, In Line 9 of the algorithm, we never remove states from P_1. As the number of states in $\mathbf{G}$ is finite, after finite number of steps we will have $P1 \equiv P3$ for the `repeat` loop from Line 6 to Line 10 and $P1 \equiv P2$ for the `repeat` loop from Line 3 to Line 12; thus the algorithm will terminate.

Algorithm 6.3 $R(\mathbf{G}, P)$(Experimental)

1: $P_1 \leftarrow P \wedge pr(\{q_0\})$;
2: $P_2 \leftarrow false$;
3: **repeat**
4: $\quad P_{new} \leftarrow P_1 - P_2$;
5: $\quad P_2 \leftarrow P_1$;
6: $\quad$ **for** $i \leftarrow 1$ to n **do**
7: $\quad\quad$ **repeat**
8: $\quad\quad\quad P_3 \leftarrow P_1$;
9: $\quad\quad\quad P_1 \leftarrow P_1 \vee \left(\bigvee_{\sigma \in \Sigma_i} (\delta(P_{new}, \sigma) \wedge P)\right)$;
10: $\quad\quad$ **until** $P_1 \equiv P_3$
11: $\quad$ **end for**
12: **until** $P_1 \equiv P_2$
13: return P_1;

As this technique does not always make the algorithm work more efficient, we only give the algorithm for the predicate transformer R. For predicate transformers TR and $\mathcal{CR}$, the algorithms applying this technique can easily be obtained. Furthermore, when we say the algorithm for the predicate transformer R later on, we mean Algorithm 6.2.

As we need to do synthesis and verification on the AIP example in the next chapter, in the current version of our software tool, we use Algorithm 6.3 to compute the R predicate transformer. Similarly, we also apply this optimization technique to compute TR and $\mathcal{CR}$ in our current software tool.

6.2.3 Computation of TR

For the system $\mathbf{G}$ composed of components $\mathbf{G}'_1, \ldots, \mathbf{G}'_n$, let $\Sigma' \subseteq \Sigma$ be a fixed event subset, $P \in Pred(Q)$. From the definition of $TR(\mathbf{G}, ., \Sigma')$, we can write a straightforward algorithm for $TR(\mathbf{G}, \Sigma', P)$ as shown in Algorithm 6.4. Like the algorithm for $R(\mathbf{G}, P)$, this algorithm will terminate in finite number of steps.

Algorithm 6.4 $TR(\mathbf{G}, P, \Sigma')$(Straightforward)

1: $P_1 \leftarrow P$;
2: **repeat**
3: $\quad P_2 \leftarrow P_1$;
4: $\quad P_1 \leftarrow P_1 \vee \left(\bigvee_{\sigma \in \Sigma'} \delta^{-1}(P_1, \sigma)\right)$;
5: **until** $P_1 \equiv P_2$
6: return P_1;

An improved algorithm using the tautology of Equation 6.2 is given in Algorithm 6.5.

Algorithm 6.5 $TR(\mathbf{G}, P, \Sigma')$

1: $P_1 \leftarrow P$;
2: **repeat**
3: $\quad P_2 \leftarrow P_1$;
4: $\quad$ **for** $i \leftarrow 1$ to n **do**
5: $\quad\quad$ **repeat**
6: $\quad\quad\quad P_3 \leftarrow P_1$;
7: $\quad\quad\quad P_1 \leftarrow P_1 \vee \left(\bigvee_{\sigma \in \Sigma' \cap \Sigma_i} \delta^{-1}(P_1, \sigma)\right)$;
8: $\quad\quad$ **until** $P_1 \equiv P_3$
9: $\quad$ **end for**
10: **until** $P_1 \equiv P_2$
11: return P_1;

6.2.4 Computation of $\mathcal{CR}$

For the system $\mathbf{G}$ composed of components $\mathbf{G}'_1, \ldots, \mathbf{G}'_n$, let $\Sigma' \subseteq \Sigma$ and $P' \in Pred(Q)$ be fixed, and let $P \in Pred(Q)$. From the definition of $\mathcal{CR}(\mathbf{G}, P', \Sigma', .)$, we

can write a straightforward algorithm for $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$ as shown in Algorithm 6.6. Like the algorithm for $R(\mathbf{G}, P)$, this algorithm will terminate in finite number of steps.

Algorithm 6.6 $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$(Straightforward)

1: $P_1 \leftarrow P' \wedge P$;
2: **repeat**
3: $\quad P_2 \leftarrow P_1$;
4: $\quad P_1 \leftarrow P_1 \vee \Big(\bigvee_{\sigma\in\Sigma'}(\delta^{-1}(P_1, \sigma) \wedge P)\Big)$;
5: **until** $P_1 \equiv P_2$
6: return P_1;

An improved algorithm using the tautology of Equation 6.2 is given in Algorithm 6.7.

Algorithm 6.7 $\mathcal{CR}(\mathbf{G}, P', \Sigma', P)$

1: $P_1 \leftarrow P' \wedge P$;
2: **repeat**
3: $\quad P_2 \leftarrow P_1$;
4: $\quad$ **for** $i \leftarrow 1$ to n **do**
5: $\quad\quad$ **repeat**
6: $\quad\quad\quad P_3 \leftarrow P_1$;
7: $\quad\quad\quad P_1 \leftarrow P_1 \vee \Big(\bigvee_{\sigma\in\Sigma'\cap\Sigma_i}(\delta^{-1}(P_1, \sigma) \wedge P)\Big)$;
8: $\quad\quad$ **until** $P_1 \equiv P_3$
9: $\quad$ **end for**
10: **until** $P_1 \equiv P_2$
11: return P_1;

6.3 Miscellaneous Computation for HISC Synthesis and Verification

Let $\mathbf{\Phi}$ be the n^{th} degree interface system that respects the alphabet partition given by Equation 3.1 and is composed of plant components $\mathbf{G}^p_H, \mathbf{G}^p_{L_1}, \ldots, \mathbf{G}^p_{L_n}$, specifica-

tions $\mathbf{E}_H, \mathbf{E}_{L_1}, \ldots, \mathbf{E}_{L_n}$, and interfaces $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$. Let j be an index with range $\{1, \ldots, n\}$. For system $\mathbf{\Phi}$, from Section 6.2 we already know how to symbolically compute the predicate transformers that we used in the algorithms we defined in the previous two chapters. However, to complete the full task of HISC synthesis and verification symbolically, we also have to compute $pr(\text{Bad}_H)$ for the high-level and $pr(\text{Bad}_{L_j})$ for the j^{th} low-level for each $j \in \{1, \ldots, n\}$, to verify that all the interface DES are command-pair interface and to verify that system $\mathbf{\Phi}$ respects the alphabet partition given by Equation 3.1.

6.3.1 Computation of $pr(\text{Bad}_H)$

For system $\mathbf{\Phi}$, we now copy the definition of $pr(\text{Bad}_H)$ from Chapter 4 here for convenience.

$$
\begin{aligned}
\text{Bad}_H := \{q \in Q_H \mid\ & \\
& ((\exists \sigma_u \in \Sigma_{hu})\ (\eta_H \times \xi^h((y,x),\sigma_u)!\ \ \&\ \ \zeta_H(z,\sigma_u)\ \not!)) \ or \\
& ((\exists j \in \{1,\ldots,n\})(\exists \sigma_{a_j} \in \Sigma_{A_j})\ (\xi^h_j(x_j,\sigma_{a_j})!\ \& \\
& \quad \zeta_H \times \eta_H \times \xi^h_1 \times \cdots \times \xi^h_{j-1} \times \xi^h_{j+1} \times \cdots \times \xi^h_n((z,y,x_1,\ldots,x_{j-1},x_{j+1},\ldots,x_n),\sigma_{a_j})\ \not!))\},
\end{aligned}
$$

where $q = (z, y, x) = (z, y, x_1, \ldots, x_n)$ as in Equation 4.1 (Page 79).

For system $\mathbf{\Phi}$, let $\mathbf{v}_{IH}$ and $\mathbf{v}'_{IH}$ be the normal state variable vector and the prime state variable vector for the high-level $\boldsymbol{\mathcal{G}}_H$, respectively.

For system $\mathbf{\Phi}$, for each $\sigma \in \Sigma_{IH}$, let $(\boldsymbol{v}_\sigma, \boldsymbol{v}'_\sigma, N_\sigma)$ be the transition tuple for σ in the high level $\boldsymbol{\mathcal{G}}_H$; For each $\sigma_u \in \Sigma_{hu}$, let $(\boldsymbol{v}_{pI,\sigma_u}, \boldsymbol{v}'_{pI,\sigma_u}, N_{pI,\sigma_u})$ be the transition tuple for the component DES of $\mathbf{G}^p_H \times \mathbf{G}^h_I$; For each $\sigma_u \in \Sigma_{hu}$, let $(\boldsymbol{v}_{s,\sigma_u}, \boldsymbol{v}'_{s,\sigma_u}, N_{s,\sigma_u})$ be the transition tuple for the component DES of $\mathbf{E}_H$; For each $j \in \{1, \ldots, n\}$ and each $\alpha \in \Sigma_{A_j}$, let $(\boldsymbol{v}_{(H-I_j),\alpha}, \boldsymbol{v}'_{(H-I_j),\alpha}, N_{(H-I_j),\alpha})$ be the transition tuple for the component DES of $\mathbf{E}_H \times \mathbf{G}^p_H \times \mathbf{G}^h_{I_1} \times \cdots \times \mathbf{G}^h_{I_{j-1}} \times \mathbf{G}^h_{I_{j+1}} \times \cdots \times \mathbf{G}^h_{I_n}$. For each $j \in \{1, \ldots, n\}$

and each $\alpha \in \Sigma_{A_j}$, let v_{I_j} and v'_{I_j} respectively be the normal and prime state variable for the interface $\mathbf{G}_{I_j}$ and $N_{I_j,\alpha}$ be the transition predicate for α in the interface DES $\mathbf{G}_{I_j}$.

Now, for system $\mathbf{\Phi}$, we can compute $pr(\text{Bad}_H)$ symbolically by

$$pr(\text{Bad}_H) := \left(\bigvee_{\sigma_u \in \Sigma_{hu}} (\exists \boldsymbol{v}'_{pI,\sigma_u} N_{pI,\sigma_u} \wedge \neg(\exists \boldsymbol{v}'_{s,\sigma_u} N_{s,\sigma_u})) \right) \vee \left(\bigvee_{j \in \{1,\ldots,n\}} \bigvee_{\alpha \in \Sigma_{A_j}} \left(\exists v'_{I_j} N_{I_j,\alpha} \wedge \neg(\exists \boldsymbol{v}'_{(H-I_j),\alpha} N_{(H-I_j),\alpha}) \right) \right)$$

In order to save memory space, for $\sigma_u \in \Sigma_{hu}$, we do not define N_{s,σ_u} in our software tool directly. We compute $\exists(\boldsymbol{v}_{pI,\sigma_u} \cup \boldsymbol{v}'_{pI,\sigma_u}) N_{\sigma_u}$ instead of directly defining N_{s,σ_u}. In most cases, $\exists(\boldsymbol{v}_{pI,\sigma_u} \cup \boldsymbol{v}'_{pI,\sigma_u}) N_{\sigma_u}$ represents the transition predicate for σ_u in $\mathbf{E}_H$, i.e. N_{s,σ_u}. However, if σ_u is blocked by DES $\mathbf{G}^p_H \times \mathbf{G}^h_I$ (e.g. σ_u is in the event set of a component DES, but there is no σ_u transition in that DES) and there exist σ_u transitions in the component DES of $\mathbf{E}_H$ and σ_u is never blocked by any component DES or combinations of component DES of $\mathbf{E}_H$, then by Equation 6.3, we have $N_{pI,\sigma_u} \equiv 0$, $N_{\sigma_u} \equiv 0$ and $N_{s,\sigma_u} \not\equiv 0$, but $\exists(\boldsymbol{v}_{pI,\sigma_u} \cup \boldsymbol{v}'_{pI,\sigma_u}) N_{\sigma_u} \equiv 0$. However, because $N_{pI,\sigma_u} \equiv 0$, we have $\exists \boldsymbol{v}'_{pI,\sigma_u} N_{pI,\sigma_u} \equiv 0$. Then we always have

$$\exists \boldsymbol{v}'_{pI,\sigma_u} N_{pI,\sigma_u} \wedge \neg(\exists \boldsymbol{v}'_{s,\sigma_u} N_{s,\sigma_u}) \equiv 0$$

regardless of the value of $\neg(\exists \boldsymbol{v}'_{s,\sigma_u} N_{s,\sigma_u})$. Therefore, by replacing N_{s,σ_u} with $\exists(\boldsymbol{v}_{pI,\sigma_u} \cup \boldsymbol{v}'_{pI,\sigma_u}) N_{\sigma_u}$ does not change the result of $pr(\text{Bad}_H)$.

Similarly, for $j \in \{1,\ldots,n\}$ and $\alpha \in \Sigma_{A_j}$, we compute $\exists v_{I_j}(\exists v'_{I_j} N_\alpha)$ instead of directly defining $N_{(H-I_j),\alpha}$. For the similar reason as above, this replacement does not affect the result of $pr(\text{Bad}_H)$.

6.3.2 Computation of $pr(\mathrm{Bad}_{L_j})$

For system $\mathbf{\Phi}$ with $j \in \{1, \ldots, n\}$, we now discuss how to compute $pr(\mathrm{Bad}_{L_j})$ for the j^{th} low-level in system $\mathbf{\Phi}$.

For convenience, we reprint the definition of Bad_{L_j} from Chapter 4 here.

$$\mathrm{Bad}_{L_j} := \{q \in Q_{L_j} | ((\exists \sigma_{lu_j} \in \Sigma_{lu_j})\ \eta_{L_j}(y, \sigma_{lu_j})! \ \&\ \zeta_{L_j} \times \xi^l_j((z,x), \sigma_{lu_j})\ \not!) \ \textit{or}$$
$$((\exists \rho \in \Sigma_{R_j})\ \xi^l_j(x, \rho)! \ \&\ \zeta_{L_j} \times \eta_{L_j}((z,y), \rho)\ \not!)\},$$

where $q = (z, y, x)$ as in Equation 4.2 (Page 105).

For system $\mathbf{\Phi}$, let $\mathbf{v}_{IL_j}$ and $\mathbf{v}'_{IL_j}$ be the normal state variable vector and the prime state variable vector for the j^{th} low-level $\boldsymbol{\mathcal{G}}_{L_j}$, respectively.

For system $\mathbf{\Phi}$, for each $\sigma \in \Sigma_{IL_j}$, let $(\boldsymbol{v}_\sigma, \boldsymbol{v}'_\sigma, N_\sigma)$ be the transition tuple for σ in the j^{th} low-level $\boldsymbol{\mathcal{G}}_{L_j}$; For each $\sigma_u \in \Sigma_{lu_j}$, let $(\boldsymbol{v}_{p,\sigma_u}, \boldsymbol{v}'_{p,\sigma_u}, N_{p,\sigma_u})$ be the transition tuple for the component DES of $\mathbf{G}^p_{L_j}$; For each $\sigma_u \in \Sigma_{lu_j}$, let $(\boldsymbol{v}_{sI,\sigma_u}, \boldsymbol{v}'_{sI,\sigma_u}, N_{sI,\sigma_u})$ be the transition tuple for the component DES of $\mathbf{E}_{L_j} \times \mathbf{G}^l_{I_j}$; For each $\rho \in \Sigma_{R_j}$, let $(\boldsymbol{v}_{(L_j - I_j),\rho}, \boldsymbol{v}'_{(L_j-I_j),\rho}, N_{(L_j-I_j),\rho})$ be the transition tuple for the component DES of $\mathbf{E}_{L_j} \times \mathbf{G}^p_{L_j}$; For each $\rho \in \Sigma_{R_j}$, let v_{I_j} and v'_{I_j} respectively be the normal and prime state variable for the interface $\mathbf{G}_{I_j}$ and $N_{I_j,\rho}$ be the transition predicate for ρ in the interface DES $\mathbf{G}_{I_j}$.

For the j^{th} low-level in system $\mathbf{\Phi}$, we can now compute $pr(\mathrm{Bad}_{L_j})$ symbolically by

$$pr(\mathrm{Bad}_{L_j}) := \left(\bigvee_{\sigma_u \in \Sigma_{lu_j}} (\exists \boldsymbol{v}'_{p,\sigma_u} N_{p,\sigma_u} \wedge \neg(\exists \boldsymbol{v}'_{sI,\sigma_u} N_{sI,\sigma_u})) \right) \vee$$
$$\left(\bigvee_{\rho \in \Sigma_{R_j}} \left(\exists v'_{I_j} N_{I_j,\rho} \wedge \neg(\exists \boldsymbol{v}'_{(L_j - I_j),\rho} N_{(L_j - I_j),\rho}) \right) \right)$$

Similarly to how we compute $pr(\mathrm{Bad}_H)$, for $\sigma_u \in \Sigma_{lu_j}$, we compute $\exists(\boldsymbol{v}_{p,\sigma_u} \cup \boldsymbol{v}'_{p,\sigma_u}) N_{\sigma_u}$ instead of directly defining N_{sI,σ_u}, and for $\rho \in \Sigma_{R_j}$, we compute $\exists v_{I_j} (\exists v'_{I_j} N_\rho)$

instead of directly defining $N_{(L_j - I_j),\rho}$. For similar reasons as for the computation of $pr(\text{Bad}_H)$, we know that these replacements will not change the result of $pr(\text{Bad}_{L_j})$.

6.3.3 Miscellaneous Computation in Algorithm 4.2

For system $\mathbf{\Phi}$ with $j \in \{1, \ldots, n\}$, we now show how to compute P_α for $\alpha \in \Sigma_{A_j}$ and $P \in Pred(Q_{L_j})$ in Algorithm 4.2 and how to compute $pr(\{q \in Q_{L_j} \mid \xi^l_j(x, \rho\alpha)! \;\&\; \delta_{L_j}(q, \rho) \models \neg P_{\mathcal{CR}_\alpha}\})$ in Line 6 of Algorithm 4.2, where $q = (z, y, x)$ as in Equation 4.2 (Page 105).

For convenience, we copy here P_α in Definition 4.19 here

$$P_\alpha := pr(\{q' \in Q_{L_j} | \delta_{L_j}(q', \alpha) \models P\})$$

For system $\mathbf{\Phi}$, let $\mathbf{v}_{IL_j}$ and $\mathbf{v}'_{IL_j}$ be the normal state variable vector and the prime state variable vector for the j^{th} low-level $\boldsymbol{\mathcal{G}}_{L_j}$ respectively.

For system $\mathbf{\Phi}$, for each $\sigma \in \Sigma_{IL_j}$, let $(\boldsymbol{v}_\sigma, \boldsymbol{v}'_\sigma, N_\sigma)$ be the transition tuple for σ in the j^{th} low-level $\boldsymbol{\mathcal{G}}_{L_j}$; For each $\sigma_{I_j} \in \Sigma_{I_j}$, let v_{I_j} and v'_{I_j} respectively be the normal and prime state variable for the interface $\mathbf{G}_{I_j}$ and $N_{I_j,\sigma_{I_j}}$ be the transition predicate for σ_{I_j} in the interface DES $\mathbf{G}_{I_j}$.

For system $\mathbf{\Phi}$, let $\alpha \in \Sigma_{A_j}$ and $P \in Pred(Q_{L_j})$. From the definition of P_α, we know $P_\alpha = \delta^{-1}_{L_j}(P, \alpha)$. So, we can compute P_α symbolically by

$$P_\alpha := \exists \boldsymbol{v}'_\alpha (N_\alpha \wedge (P[\boldsymbol{v}_\alpha \rightarrow \boldsymbol{v}'_\alpha]))$$

For system $\mathbf{\Phi}$, let $\rho \in \Sigma_{R_j}, \alpha \in \Sigma_{A_j}$ and $P_{\mathcal{CR}_\alpha} \in Pred(Q_{L_j})$. We can compute $pr(\{q \in Q_{L_j} | \xi^l_j(x, \rho\alpha)! \;\&\; \delta_{L_j}(q, \rho) \models \neg P_{\mathcal{CR}_\alpha}\})$ symbolically by

$$\exists v'_{I_j}(N_{I_j,\rho} \wedge ((\exists v'_{I_j} N_{I_j,\alpha})[v_{I_j} \rightarrow v'_{I_j}])) \wedge$$
$$\exists \boldsymbol{v}'_\rho (N_\rho \wedge ((\neg P_{\mathcal{CR}_\alpha})[\boldsymbol{v}_\rho \rightarrow \boldsymbol{v}'_\rho])).$$

$\exists v'_{I_j} N_{I_j,\alpha}$ computes the state predicate including all the states in Q_{L_j} with α transition defined. So, $\exists v'_{I_j}(N_{I_j,\rho} \wedge ((\exists v'_{I_j} N_{I_j,\alpha})[v_{I_j} \rightarrow v'_{I_j}]))$ computes the state predicate including all the states in Q_{L_j} with ρ transitions defined, and the target states of these ρ transitions are the source states of α transitions in Q_{L_j}. That is to say, $\exists v'_{I_j}(N_{I_j,\rho} \wedge ((\exists v'_{I_j} N_{I_j,\alpha})[v_{I_j} \rightarrow v'_{I_j}]))$ computes the predicate identifying the set $\{q \in Q_{L_j} \mid \xi^l_j(x, \rho\alpha)!\}$.

6.3.4 Verifying Event Partition

One of the most fundamental properties in the HISC architecture is that an HISC system must respect the event partition defined in Equation 3.1.

As it is usually the case that the number of events in an HISC system is very limited compared to the number of states (e.g. only hundreds in the AIP example in next chapter), the efficiency of the algorithm to verify this property is not critical.

The method used in our software tool is as follows. For each event in the system, we encode it as an integer value. From the integer value, we are able to know which event set the event belongs to. We save the event name and its encoded integer value for all events in a balanced binary search tree[3] with the event name as the key.

Initially, the tree is empty. When we read a component DES file, we can determine which event partition each event belongs to based on the information in the DES file and where the DES file belongs in the system (e.g. part of the high-level, the j^{th} interface, the j^{th} low-level, etc.). Thus we can encode it as an integer value. Then we search for this event in the tree. If the event does not exist, we insert this event into the tree. Otherwise, we compare the integer value associated with the event in the tree and the value we just encoded. If they are the same, then there is no problem. Otherwise, it means that this event is in two different event partitions. After we finish

[3]The data structure used in our software tool is GNU C++ STL map, which is a red-black tree.

reading all the DES, if there are no two events having the same name but different integer values, then we know the system respects the event partition.

As the data structure to save all the events is a balanced binary search tree, the worst-case searching and insertion time for each event are both $O(\log_2 n)$, where n is the number of events in the system. Thus the total time for handling each event is still $O(\log_2 n)$, and the worst-case verification time for event partition is $O(mn \log_2 n)$, where m is the number of DES components in the system.

6.3.5 Verifying Command-pair Interfaces

For the HISC system $\mathbf{\Phi}$, we also need to verify if all the interfaces in the system are command-pair interfaces.

Let $j \in \{1, \ldots, n\}$. We now discuss the algorithm to verify the j^{th} interface DES $\mathbf{G}_{I_j} := (X_j, \Sigma_{I_j}, \xi_j, x_{j_0}, X_{j_m})$. Because $\mathbf{G}_{I_j}$ is usually composed of single component DES, efficiency is not the main concern for the verification algorithm.

For all $\sigma \in \Sigma_{I_j}$, let $N_{I_j,\sigma}$ be the transition predicate for σ in $\mathbf{G}_{I_j}$. Let v_{I_j} and v'_{I_j} be the normal state variable and prime state variable for $\mathbf{G}_{I_j}$ respectively.

Algorithm 6.8 shows how to verify if the j^{th} interface DES $\mathbf{G}_{I_j}$ is a command-pair interface.

Line 1 to 4 says that $\mathbf{G}_{I_j}$ can not be an empty DES. Line 5 computes the predicate including all the reachable states. Line 6 computes the predicate including all the reachable marker states.

Line 7 and 8 computes the predicate including all the reachable states which are the target states of answer event transitions plus the initial state. The states satisfying P_{R_j} should only be the source states of request event transitions.

Line 9 and 10 computes the predicate including all the reachable states which are

Algorithm 6.8 Verifying if $\mathbf{G}_{I_j}$ is a command-pair interface

1: $P_{I_{j_0}} \leftarrow pr(\{x_{j_0}\})$;
2: **if** $(P_{I_{j_0}} \equiv false)$ **then**
3: return "$\mathbf{G}_{I_j}$ has no initial state.";
4: **end if**
5: $P_{I_j} \leftarrow R(\mathbf{G}_{I_j}, true)$;
6: $P_{I_{j_m}} \leftarrow pr(X_{j_m}) \wedge P_{I_j}$;
7: $P_{R_j} \leftarrow \bigvee_{\alpha \in \Sigma_{A_j}} ((\exists v_{I_j} N_{I_j,\alpha})[v'_{I_j} \rightarrow v_{I_j}])$;
8: $P_{R_j} \leftarrow (P_{R_j} \wedge P_{I_j}) \vee P_{I_{j_0}}$;
9: $P_{A_j} \leftarrow \bigvee_{\rho \in \Sigma_{R_j}} ((\exists v_{I_j} N_{I_j,\rho})[v'_{I_j} \rightarrow v_{I_j}])$;
10: $P_{A_j} \leftarrow P_{A_j} \wedge P_{I_j}$;
11: $P_{RA_j} \leftarrow \bigvee_{\alpha \in \Sigma_{A_j}} \xi_j(P_{R_j}, \alpha)$;
12: $P_{AR_j} \leftarrow \bigvee_{\rho \in \Sigma_{R_j}} \xi_j(P_{A_j}, \rho)$;
13: **if** $(P_{AR_j} \not\equiv false)$ *or* $(P_{RA_j} \not\equiv false)$ **then**
14: return "$\mathbf{G}_{I_j}$ fails Point A of the command-pair interface definition";
15: **end if**
16: **if** $(P_{R_j} \not\equiv P_{I_{j_m}})$ **then**
17: return "$\mathbf{G}_{I_j}$ fails Point B of the command-pair interface definition";
18: **end if**
19: return "$\mathbf{G}_{I_j}$ is a command-pair interface.";

the target states of request event transitions. The states satisfying P_{A_j} should only be the source states of answer event transitions.

Line 11 computes the predicate including all the reachable states which are the target states of answer event transitions with source states satisfying P_{R_j}.

Line 12 computes the predicate including all the reachable states which are the target states of request event transitions with source states satisfying P_{A_j}.

Now we informally explain the correctness of the algorithm.

From Line 1 to 4, the algorithm makes sure that $\mathbf{G}_{I_j}$ is not empty.

In the following, we always assume $\mathbf{G}_{I_j}$ is not empty.

We first discuss Point A of Definition 3.1. To show the correctness of the algorithm

for Point A, it is sufficient to show that

$(\forall s \in L(\mathbf{G}_{I_j}))\ s \in \overline{(\Sigma_{R_j}.\Sigma_{A_j})^*}$ if and only if $P_{RA_j} \equiv false$ and $P_{AR_j} \equiv false$.

Equivalently, we can show that $(\exists s \in L(\mathbf{G}_{I_j}))\ s \notin \overline{(\Sigma_{R_j}.\Sigma_{A_j})^*}$ if and only if $P_{RA_j} \not\equiv false$ or $P_{AR_j} \not\equiv false$.

1. Show $((\exists s \in L(\mathbf{G}_{I_j}))\ s \notin \overline{(\Sigma_{R_j}.\Sigma_{A_j})^*}) \Rightarrow P_{RA_j} \not\equiv false\ or\ P_{AR_j} \not\equiv false$.

   Let $s \in L(\mathbf{G}_{I_j})$. Assume $s \notin \overline{(\Sigma_{R_j}.\Sigma_{A_j})^*}$. Must show this implies $P_{RA_j} \not\equiv false\ or\ P_{AR_j} \not\equiv false$.

   From $s \notin \overline{(\Sigma_{R_j}.\Sigma_{A_j})^*}$, we know that one or more of the following three conditions must be satisfied.

   (a) $s \in \Sigma_{I_j}^*.\Sigma_{A_j}.\Sigma_{A_j}.\Sigma_{I_j}^*$

   $\Rightarrow (\exists u, v \in \Sigma_{I_j}^*)(\exists \alpha_1, \alpha_2 \in \Sigma_{A_j})\ s = u\alpha_1\alpha_2 v$.

   From Line 7 and 8, we know that $\xi_j(x_{j_0}, u\alpha_1) \models P_{R_j}$.

   From Line 11, we know that $\xi_j(x_{j_0}, u\alpha_1\alpha_2) \models P_{RA_j}$, so $P_{RA_j} \not\equiv false$.

   (b) $s \in \Sigma_{I_j}^*.\Sigma_{R_j}.\Sigma_{R_j}.\Sigma_{I_j}^*$

   $\Rightarrow (\exists u, v \in \Sigma_{I_j}^*)(\exists \rho_1, \rho_2 \in \Sigma_{R_j})\ s = u\rho_1\rho_2 v$.

   From Line 9 and 10, we know that $\xi_j(x_{j_0}, u\rho_1) \models P_{A_j}$.

   From Line 12, we know that $\xi_j(x_{j_0}, u\rho_1\rho_2) \models P_{AR_j}$, so $P_{AR_j} \not\equiv false$.

   (c) $s \in \Sigma_{A_j}.\Sigma_{I_j}^*$

   $\Rightarrow (\exists \alpha \in \Sigma_{A_j})(\exists u \in \Sigma_{I_j}^*)\ s = \alpha u$.

   From Line 8, we know that $x_{j_0} \models P_{R_j}$.

   From Line 11, we know that $\xi_j(x_{j_0}, \alpha) \models P_{RA_j}$, so $P_{RA_j} \not\equiv false$.

   From (a), (b) and (c), we know that Point 1 holds.

2. Show $P_{RA_j} \not\equiv false$ *or* $P_{AR_j} \not\equiv false \Rightarrow ((\exists s \in L(\mathbf{G}_{I_j}))\ s \notin \overline{(\Sigma_{R_j}.\Sigma_{A_j})^*})$.

   From Line 7, 8 and 11, we know that if $P_{RA_j} \not\equiv false$ then $(\exists s \in L(\mathbf{G}_{I_j}))\ s \in \Sigma^*_{I_j}.\Sigma_{A_j}.\Sigma_{A_j}.\Sigma^*_{I_j}$ or $s \in \Sigma_{A_j}.\Sigma^*_{I_j}$.

   From Line 9, 10 and 12, we know that if $P_{AR_j} \not\equiv false$ then $(\exists s \in L(\mathbf{G}_{I_j}))\ s \in \Sigma^*_{I_j}.\Sigma_{R_j}.\Sigma_{R_j}.\Sigma^*_{I_j}$.

   We can thus conclude that Point 2 holds.

We now discuss Point B of Definition 3.1. Assuming Point A of Definition 3.1 holds, to show the correctness of Point B, it is sufficient to show that $\mathbf{G}_{I_j}$ satisfies Point B if and only if $P_{R_j} \equiv P_{I_{j_m}}$.

From Point B, we know $\mathbf{G}_{I_j}$ satisfies Point B if and only if all the target states (reachable) of all answer event transitions in $\mathbf{G}_{I_j}$ and the initial state of $\mathbf{G}_{I_j}$ must be marked (i.e. $P_{R_j} \preceq P_{I_{j_m}}$.), and there is no other states being marked (i.e. $P_{I_{j_m}} \preceq P_{R_j}$.). Therefore, we have $\mathbf{G}_{I_j}$ satisfies Point B if and only if $P_{R_j} \equiv P_{I_{j_m}}$.

6.4 BDD Implementations of Algorithms

So far, we have shown how to represent an HISC system as logic formulas and the method to implement the HISC synthesis and verification algorithms from the logic formulas. The logic formulas we used can be directly represented as an Integer Decision Diagram(IDD) [16, 51]. We use Binary Decision Diagram (BDD) [6] to represent the formulas, as a BDD software package is readily available now. The BDD software package we used is *BuDDy 2.4* developed by Jørn Lind-Nielsen. Due to space and time constraints, we will not give an introduction for BDD or the software package here. Readers are referred to [19] for an introduction to BDD and its application to symbolic model checking, and to [1] for the BDD package implementation details.

Each state variable in our logic formula is a *finite domain variable* (simply means that a variable can only be assigned finite number of values), which can be represented by a block of binary variables. For instance, a state variable for a 3-state DES can be represented by two binary variables, and a state variable for a 5-state DES can be represented by three binary variables. Assume the number of states in a DES is m, by replacing the finite domain state variable for this DES with a block of binary variables, we actually extend the variable domain to 2^k, where $k = \lceil \log_2 m \rceil$. That is, we actually extend the number of states in the DES to 2^k. However, because of the transition predicate, those extended states will be treated as unreachable states, which do not affect the language result at all.

All the logic operations such as $\wedge, \vee, \neg$ and $\exists$ can be done by corresponding BDD operations directly. For the $\exists$ operation of a finite domain variable, we have to do $\exists$ operation on all of its corresponding binary variables. For example, a state variable v with domain $\{0,1,2\}$ is represented by two binary variables v_1 and v_2. The operation $\exists v$ can be done by $\exists v_1(\exists v_2)$ instead. A very good aspect in the software package $BuDDy$ is that it provides a series of functions with prefix "$fdd\_$", which allow developers to be able to implement the logic operations on finite domain variables directly. In other words, we can implement operations in terms of finite domain variables (state variables in our case), and $BuDDy$ package will automatically translate the operations into ones on their corresponding binary variables. Please refer to the source code of our software tool in Appendix A for more details.

6.4.1 State Variable Ordering

A well-known fact of BDD is that the order of variables in a BDD can dramatically change the number of the BDD nodes. Because the efficiency of all the BDD

operations directly depends on the number of the BDD nodes, the variable order in a BDD plays an important role in the algorithm's actual efficiency. However, to find the optimal order for BDD is NP-Complete [4], so heuristics on variable orders are usually used to reduce the BDD size.

In our case, we always keep binary variables in a block for each finite domain variable together because of the tautology in Equation 6.2. Therefore, we only care about the order of the finite domain variables, i.e. the state variables.

Although the software package provides a so-called dynamic BDD variable ordering function [38] to optimize the order, an initial variable order is critical, otherwise the dynamic ordering function will not have as much effect as expected.

The method we used is that two state variables should be kept closer if their corresponding DES have more shared events. This method was found by Zhang in [51]. The central idea behind this method is that a BDD can be reduced more if two state variables that share more information are kept closer.

Assuming there are k component DES in the high-level or a low-level of an HISC system, for example, we can arrange the state variables for all the DES by trying to make the value of $Cross$ in the following expression as small as possible. The C in the expression is a matrix storing the number of shared events between the i^{th} component and the j^{th} component in the current order.[4]

$$Cross := \sum_{i=1}^{k} \sum_{j=i+2}^{k} C[i,j] * (j - i - 1)$$

For a high-level or low-level with k DES, there are $k!$ different orders. It is impractical to compute $Cross$ for all the possible orders on a personal computer for a even moderately large k(e.g. $k = 50, k! = 3 \times 10^{64}$). Therefore, we use the *sifting*

[4]This expression is quite experimental, but it works well in our experience.

algorithm [38] to compute only a very small number of the orders, and then choose the order with the smallest value for $Cross$ among them as the initial order. Here we only adopt how the sifting algorithm swaps the variables. We then compute $Cross$ for the order after each swapping. As a result, it has much less computation than to swap two variables in a BDD and then count the number of BDD nodes.

6.5 Controller Implementation

Let $\mathbf{\Phi}$ be the n^{th} degree HISC-valid specification interface system that respects the alphabet partition given by Equation 3.1 and is composed of plant components $\mathbf{G}_H^p, \mathbf{G}_{L_1}^p, \ldots, \mathbf{G}_{L_n}^p$, specifications $\mathbf{E}_H, \mathbf{E}_{L_1}, \ldots, \mathbf{E}_{L_n}$, and interfaces $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$. From Chapter 4, we can synthesize a high-level proper supervisor $\mathbf{S}_H$ by making $\mathbf{S}_H$ represent $L_m(\boldsymbol{\mathcal{G}}_H, \Gamma_H^k(true))$, where $k \in \{0, 1, \ldots\}$ and $\Gamma_H^k(true)$ is the greatest fixpoint of Γ_H with respect to $(Pred(Q_H), \preceq)$, and synthesize a j^{th} low-level proper supervisor $\mathbf{S}_{L_j}$ by making $\mathbf{S}_{L_j}$ represent $L_m(\boldsymbol{\mathcal{G}}_{L_j}, \Gamma_{L_j}^{k_j}(true))$ for all $j \in \{1, \ldots, n\}$, where $k_j \in \{0, 1, \ldots\}$ and $\Gamma_{L_j}^{k_j}(true)$ is the greatest fixpoint of Γ_{L_j} with respect to $(Pred(Q_{L_j}), \preceq)$. The supervisor **SUP** for the whole system would be $\mathbf{SUP} := \mathbf{S}_H \parallel \mathbf{S}_{L_1} \parallel \cdots \parallel \mathbf{S}_{L_n} \parallel \mathbf{G}_{I_1} \parallel \cdots \parallel \mathbf{G}_{I_n}$. However, the automata-based supervisors $\mathbf{S}_H, \mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$ could easily be very large, so that they are very difficult to implement. For example, $\mathbf{S}_H$ for the 3-2 AIP example in the next chapter has a state space size on the order of 10^{12}.

From the synthesis algorithms in Chapter 4, we can see that $\mathbf{S}_H$ can be obtained by trimming off states from the high-level $\boldsymbol{\mathcal{G}}_H$, and $\mathbf{S}_{L_j}$ can be obtained by trimming off states from the j^{th} low-level $\boldsymbol{\mathcal{G}}_{L_j}, j \in \{1, \ldots, n\}$. Therefore, we can implement the supervisor for system $\mathbf{\Phi}$ in another way as follows.

For system $\mathbf{\Phi}$, let P_H be the resulting predicate of the Algorithm 4.1 for the high-

level, i.e. $P_H := \Gamma_H^k(true)$ and P_{L_j} be the result predicate of the Algorithm 4.3 for the j^{th} low-level($j \in \{1, \ldots, n\}$), i.e. $P_{L_j} := \Gamma_{L_j}^{k_j}(true)$. Assume[5] $P_H \not\equiv false$ and $P_{L_j} \not\equiv false$ for all $j \in \{1, \ldots, n\}$. In the following, j is always an index with range $\{1, \ldots, n\}$.

Let $\mathbf{G} := (Q, \Sigma, \delta, q_0, Q_m)$ be the DES tuple for the synchronous product of all the DES in system $\mathbf{\Phi}$, then Σ is defined as in Equation 3.1 and $\Sigma := \Sigma_c \dot{\cup} \Sigma_u$. A state $q \in Q$ can be represented as a tuple

$$(z_H, y_H, z_{L_1}, \ldots, z_{L_n}, y_{L_1}, \ldots, y_{L_n}, x_1, \ldots, x_n), \tag{6.8}$$

where $z_H \in Z_H, y_H \in Y_H, z_{L_1} \in Z_{L_1}, \ldots, z_{L_n} \in Z_{L_n}, y_{L_1} \in Y_{L_1}, \ldots, y_{L_n} \in Y_{L_n}, x_1 \in X_1, \ldots, x_n \in X_n$ (see Section 4.1 for more information).

For each $\sigma \in \Sigma$, define the *control predicate* $f_\sigma \in Pred(Q)$ for σ as following:

$$(\forall q \in Q)\ f_\sigma(q) := \begin{cases} 1, & (\sigma \in \Sigma_H \ \&\ \delta_H((z_H, y_H, x_1, \ldots, x_n), \sigma) \models P_H)\ or \\ & \\ & (\sigma \in \Sigma_{L_1} \ \&\ \delta_{L_1}((z_{L_1}, y_{L_1}, x_1), \sigma) \models P_{L_1})\ or \\ & \cdots\ or \\ & (\sigma \in \Sigma_{L_n} \ \&\ \delta_{L_n}((z_{L_n}, y_{L_n}, x_n), \sigma) \models P_{L_n})\ or \\ & \\ & \big(\sigma \in \Sigma_{I_1} \ \&\ (\delta_H((z_H, y_H, x_1, \ldots, x_n), \sigma) \models P_H)\ \& \\ & \qquad (\delta_{L_1}((z_{L_1}, y_{L_1}, x_1), \sigma) \models P_{L_1})\big)\ or \\ & \cdots\ or \\ & \big(\sigma \in \Sigma_{I_n} \ \&\ (\delta_H((z_H, y_H, x_1, \ldots, x_n), \sigma) \models P_H)\ \& \\ & \qquad (\delta_{L_n}((z_{L_n}, y_{L_n}, x_n), \sigma) \models P_{L_n})\big) \\ 0, & \text{otherwise} \end{cases} \tag{6.9}$$

Equation 6.9 is based on the event partition in Equation (3.1), and the fact that $\boldsymbol{\mathcal{G}}_H$ is defined over Σ_{IH} and $\boldsymbol{\mathcal{G}}_{L_j}$ is defined over Σ_{IL_j}.

For $q \in Q$ and $\sigma \in \Sigma$, σ should then be enabled at state q if $f_\sigma(q) = 1$. Therefore, from the control predicate f_σ for all $\sigma \in \Sigma$, we can decide the control action. To

[5] If P_H or any of $P_{L_j}(j \in \{1, \ldots, n\})$ are equal to $false$, this would mean no supervisor exists, so we would have nothing to implement.

implement, we only need to use the predicates for $\sigma \in \Sigma_c$, as our supervisor is controllable by our synthesis process and Theorem 3.2. Thus it will never try to disable $\sigma \in \Sigma_u$ when σ is possible in the plant. Note that each of these predicates can also be represented as a BDD, and usually the BDD representation is much smaller than the automata representation. However, to evaluate f_σ, we need to know the current state of $\mathbf{G}$.

We will use all the component automata models (component DES) $\mathbf{G}^p_H$, $\mathbf{G}^p_{L_1}$, ..., $\mathbf{G}^p_{L_n}$, $\mathbf{E}_H$, $\mathbf{E}_{L_1}$, ..., $\mathbf{E}_{L_n}$, $\mathbf{G}_{I_1}$, ..., $\mathbf{G}_{I_n}$ in system $\mathbf{\Phi}$ to trace the state of the physical plant. In other words, we will use the automata models as a kind of observer to provide our controller with the needed system state information. Note that the plant DES and specification DES are usually the synchronous product of component DES again. For example, the high-level plant in the AIP example in the next chapter is composed of 19 component DES. Although the synchronous product of them could be very large, the individual DES are usually very small.

Let $\Sigma_c = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ ($k \in \{0, 1, \ldots\}$). Figure 6.4 shows the overall picture of what our controller will look like. In the diagram, the plant models $\mathbf{G}^p_H, \mathbf{G}^p_{L_1}, \ldots, \mathbf{G}^p_{L_n}$ and specification models $\mathbf{E}_H, \mathbf{E}_{L_1}, \ldots, \mathbf{E}_{L_n}$ would be replaced by their own component DES. We also assume that all the events exist as part of the physical plant. There are many issues about the controller implementations which are beyond the scope of this thesis; here we only want to show that we do not need to build the automata supervisors $\mathbf{S}_H$ and $\mathbf{S}_{L_j}$ ($j \in \{1, \ldots, n\}$) to implement our controller. Interested readers are referred to [20] to see an example.

Equation 6.9 can actually be simplified. Notice that Point 3 and Point 4 in the interface consistent definition are very similar to the definitions of the controllable language. As the supervisor of the system never disable the uncontrollable events, the supervisor for the high-level should never disable the answer events, and the

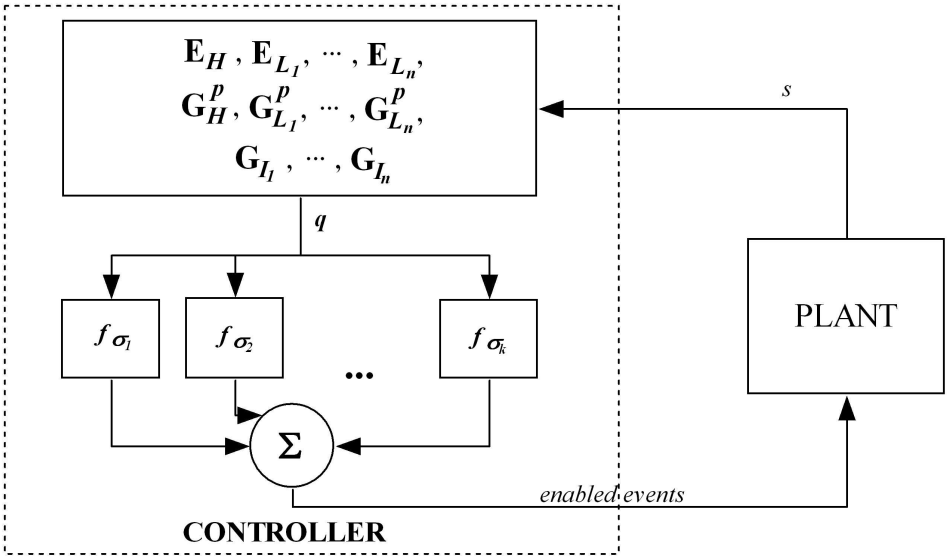


Figure 6.4: Control diagram

supervisor for the j^{th} low-level should never disable the request events in the j^{th} low-level.

Proposition 6.1. *For system* $\mathbf{\Phi}$, *let* $\mathbf{G} := (Q, \Sigma, \delta, q_0, Q_m)$ *be the DES tuple for the synchronous product of all the DES in system* $\mathbf{\Phi}$. *Let* $q \in Q$ *be a reachable state with a tuple as in Equation 6.8,* $s \in \Sigma^*$ *and* $q = \delta(q_0, s)$. *If the following three points are true:*

$$\left.\begin{array}{l} \bullet\ s := \sigma_1\sigma_2\cdots\sigma_k, \\ \quad \textit{where } k \in \{0, 1, \ldots\} (k = 0 \textit{ means } s = \epsilon) \textit{ and } \sigma_1, \sigma_2, \ldots, \sigma_k \in \Sigma; \\ \bullet\ q_1 := \delta(q_0, \sigma_1), q_2 := \delta(q_1, \sigma_2), \ldots, q = q_k := \delta(q_{k-1}, \sigma_k), \\ \quad \textit{where } q_1, q_2, \ldots, q_k \in Q; \\ \bullet\ f_{\sigma_1}(q_0) \equiv 1, f_{\sigma_2}(q_1) \equiv 1, \ldots, f_{\sigma_k}(q_{k-1}) \equiv 1, \end{array}\right\} (1)$$

then the following two points hold:

1. $(\forall j \in \{1, \ldots, n\})(\forall \rho \in \Sigma_{R_j})$

   $(\delta_H((z_H, y_H, x_1, \ldots, x_n), \rho) \models P_H) \Rightarrow (\delta_{L_j}((z_{L_j}, y_{L_j}, x_j), \rho) \models P_{L_j})$

2. $(\forall j \in \{1, \ldots, n\})(\forall \alpha \in \Sigma_{A_j})$
   $(\delta_{L_j}((z_{L_j}, y_{L_j}, x_j), \alpha) \models P_{L_j}) \Rightarrow (\delta_H((z_H, y_H, x_1, \ldots, x_n), \alpha) \models P_H)$

proof:

1. Show Point 1 holds.

   Let $j \in \{1, \ldots, n\}, \rho \in \Sigma_{R_j}$.

   Assume $\delta_H((z_H, y_H, x_1, \ldots, x_n), \rho) \models P_H$. (2)

   Must show this implies $\delta_{L_j}((z_{L_j}, y_{L_j}, x_j), \rho) \models P_{L_j}$.

   Let function $\text{PRJ}_{ILj} : \Sigma^* \to \Sigma^*_{IL_j}$ be a natural projection.

   From (2), we have $\delta_H((z_H, y_H, x_1, \ldots, x_n), \rho) \models P_H$

   $\Rightarrow \delta_H((z_H, y_H, x_1, \ldots, x_n), \rho)!$

   $\Rightarrow \xi^h_j(x_j, \rho)!, \quad$ as $\boldsymbol{\mathcal{G}}_H := \mathbf{E}_H \times \mathbf{G}^p_H \times \mathbf{G}^h_{I_1} \times \cdots \times \mathbf{G}^h_{I_n}$

   $\Rightarrow \xi_j(x_j, \rho)!, \quad$ by definition of $\mathbf{G}^h_{I_j}$ (see Section 3.2) and the fact $\rho \in \Sigma_{R_j}$

   $\Rightarrow \xi^l_j(x_j, \rho)!, \quad$ by definition of $\mathbf{G}^l_{I_j}$

   $\Rightarrow \text{PRJ}_{ILj}(s)\rho \in L(\mathbf{G}^l_{I_j}), \quad$ by (1) and definition of $\mathbf{G}$ (3)

   From the definition in Equation 6.9 and (1), we have

   $$\text{PRJ}_{ILj}(s) \in L(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_j}). \tag{4}$$

   By Theorem 4.2, we know that $L_m(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_j})$ is j^{th} low-level interface controllable. From Definition 4.16, we know that $L_m(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_j})$ is j^{th} low-level P4 interface controllable. From Definition 4.10, we know that $\overline{L_m(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_j})}$ is j^{th} low-level P4 interface controllable. By Corollary 4.4, we have $\overline{L_m(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_j})} = L(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_j})$. Thus we know that $L(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_j})$ is j^{th} low-level P4 interface controllable. By Definition 4.10, (3) and (4), we now have

$\text{PRJ}_{ILj}(s)\rho \in L(\boldsymbol{\mathcal{G}}_{L_j}, P_{L_j})$

$\Rightarrow \delta_{L_j}((z_{L_j}, y_{L_j}, x_j), \rho) \models P_{L_j}$

2. Show Point 2 holds.

   Let $j \in \{1, \ldots, n\}, \alpha \in \Sigma_{A_j}$.

   Assume $\delta_{L_j}((z_{L_j}, y_{L_j}, x_j), \alpha) \models P_{L_j}$. (5)

   Must show this implies $\delta_H((z_H, y_H, x_1, \ldots, x_n), \alpha) \models P_H$.

   Let function $\text{PRJ}_{IH} : \Sigma^* \to \Sigma^*_{IH}$ be a natural projection.

   From (5), we have $\delta_{L_j}((z_{L_j}, y_{L_j}, x_j), \alpha) \models P_{L_j}$

   $\Rightarrow \delta_{L_j}((z_{L_j}, y_{L_j}, x_j), \alpha)!$

   $\Rightarrow \xi^l_j(x_j, \alpha)!, \quad$ as $\boldsymbol{\mathcal{G}}_{L_j} := \mathbf{E}_{L_j} \times \mathbf{G}^p_{L_j} \times \mathbf{G}^l_{I_j}$

   $\Rightarrow \xi_j(x_j, \alpha)!, \quad$ by definition of $\mathbf{G}^l_{I_j}$ (see Section 3.2) and the fact $\alpha \in \Sigma_{A_j}$

   $\Rightarrow \xi^h_j(x_j, \alpha)!, \quad$ by definition of $\mathbf{G}^h_{I_j}$

   $\Rightarrow \text{PRJ}_{IH}(s)\alpha \in L(\mathbf{G}^h_{I_j}), \quad$ by (1) and definition of $\mathbf{G}$ (6)

   From the definition in Equation 6.9 and (1), we have

   $$\text{PRJ}_{IH}(s) \in L(\boldsymbol{\mathcal{G}}_H, P_H). \tag{7}$$

   By Theorem 4.1, we know that $L_m(\boldsymbol{\mathcal{G}}_H, P_H)$ is high-level interface controllable. By Definition 4.2, we know that $\overline{L_m(\boldsymbol{\mathcal{G}}_H, P_H)}$ is high-level interface controllable. By Corollary 4.2, we have $\overline{L_m(\boldsymbol{\mathcal{G}}_H, P_H)} = L(\boldsymbol{\mathcal{G}}_H, P_H)$. Thus we have $L(\boldsymbol{\mathcal{G}}_H, P_H)$ is high-level interface controllable. By Definition 4.2, (6) and (7), we now have

   $\text{PRJ}_{IH}(s)\alpha \in L(\boldsymbol{\mathcal{G}}_H, P_H)$

   $\Rightarrow \delta_H((z_H, y_H, x_1, \ldots, x_n), \alpha) \models P_H$.

□

From the above proposition, now for all $\sigma \in \Sigma_c$, we can redefine the predicate $f_\sigma \in Pred(Q)$ as following:

$$(\forall q \in Q)\ f_\sigma(q) := \begin{cases} 1, & (\sigma \in \Sigma_H \cup \Sigma_{R_1} \cup \cdots \cup \Sigma_{R_n}\ \&\ \delta_H((z_H, y_H, x_1, \ldots, x_n), \sigma) \models P_H)\ or \\ & (\sigma \in \Sigma_{L_1} \cup \Sigma_{A_1}\ \&\ \delta_{L_1}((z_{L_1}, y_{L_1}, x_1), \sigma) \models P_{L_1})\ or \\ & \cdots\ or \\ & (\sigma \in \Sigma_{L_n} \cup \Sigma_{A_n}\ \&\ \delta_{L_n}((z_{L_n}, y_{L_n}, x_n), \sigma) \models P_{L_n}) \\ 0, & \text{otherwise} \end{cases} \tag{6.10}$$

This allows us to replace the set of global control predicates f_σ with the set of *local control predicates* f_{H_σ} and $f_{L_{j_\sigma}}$.

For each $\sigma \in \Sigma_c \cap (\Sigma_H \cup \Sigma_{R_1} \cup \cdots \cup \Sigma_{R_n})$, we define the predicate $f_{H\sigma} \in Pred(Q_H)$ as

$$(\forall q \in Q)\ f_{H\sigma}(z_H, y_H, x_1, \ldots, x_n) := \begin{cases} 1, & \delta_H((z_H, y_H, x_1, \ldots, x_n), \sigma) \models P_H \\ 0, & \text{otherwise} \end{cases} \tag{6.11}$$

For each $j \in \{1, \ldots, n\}$, $\sigma \in \Sigma_c \cap (\Sigma_{L_j} \cup \Sigma_{A_j})$, we define the predicate $f_{L_{j_\sigma}} \in Pred(Q_{L_j})$ as

$$(\forall q \in Q)\ f_{L_j\sigma}(z_{L_j}, y_{L_j}, x_j) := \begin{cases} 1, & \delta_{L_j}((z_{L_j}, y_{L_j}, x_j), \sigma) \models P_{L_j} \\ 0, & \text{otherwise} \end{cases} \tag{6.12}$$

Let $\Sigma_c \cap (\Sigma_H \cup \Sigma_{R_1} \cup \cdots \cup \Sigma_{R_n}) := \{\sigma_1, \sigma_2, \ldots, \sigma_{k_H}\}$ $(k_H \in \{0, 1, \ldots\})$ and $\Sigma_c \cap (\Sigma_{L_j} \cup \Sigma_{A_j}) := \{\sigma_{j-1}, \sigma_{j-2}, \ldots, \sigma_{j-k_j}\}$ $(k_j \in \{0, 1, \ldots\})$ for $j \in \{1, \ldots, n\}$. The control diagram in Figure 6.4 can be modified as shown in Figure 6.5. Again, we assume all the events in the system exist as part of the physical plant.

For system $\mathbf{\Phi}$, for each $\sigma \in \Sigma_c \cap (\Sigma_H \cup \Sigma_{R_1} \cup \cdots \cup \Sigma_{R_n})$, the predicate $f_{H\sigma}$ can be computed by

$$f_{H\sigma} := \delta_H^{-1}(P_H, \sigma).$$

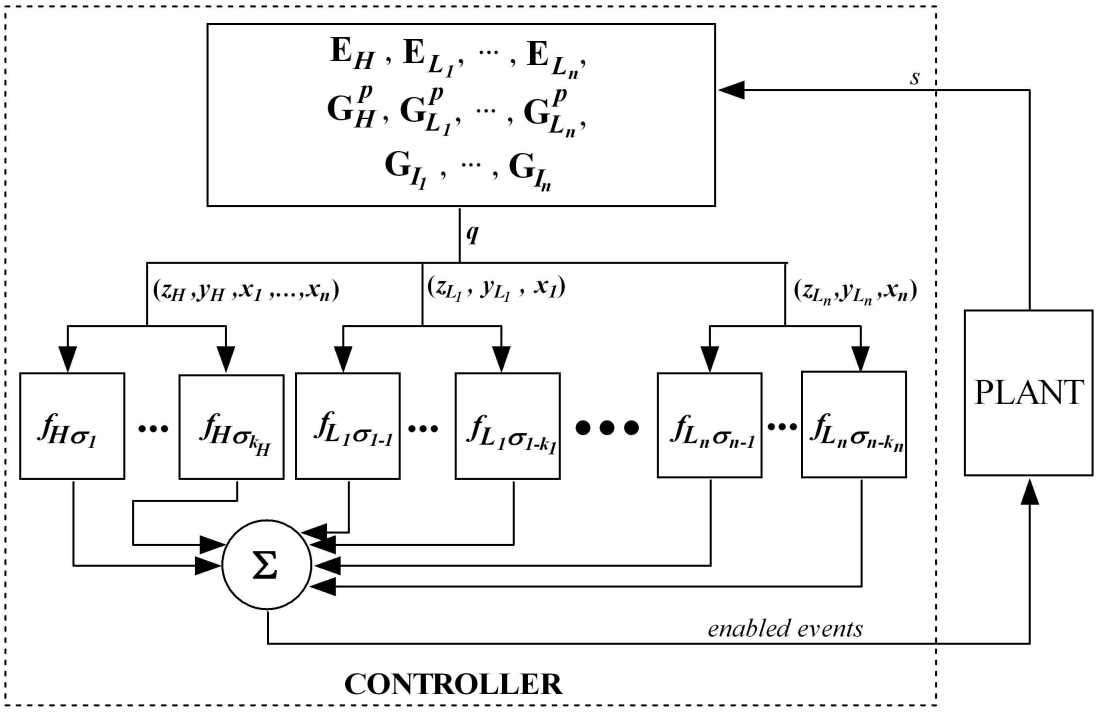


Figure 6.5: New control diagram

Let $(\boldsymbol{v}_{H_\sigma}, \boldsymbol{v}'_{H_\sigma}, N_{H_\sigma})$ be the transition tuple for σ in the high-level $\boldsymbol{\mathcal{G}}_H$, then the predicate $f_{H\sigma}$ can be computed symbolically by

$$f_{H\sigma} = \exists \boldsymbol{v}'_{H_\sigma}(N_{H_\sigma} \wedge (P_H[\boldsymbol{v}_{H_\sigma} \rightarrow \boldsymbol{v}'_{H_\sigma}])).$$

For system $\boldsymbol{\Phi}$, for each $j \in \{1, \ldots, n\}$, $\sigma \in \Sigma_c \cap (\Sigma_{L_j} \cup \Sigma_{A_j})$, the predicate $P_{L_{j_\sigma}}$ can be computed by

$$f_{L_{j_\sigma}} := \delta^{-1}_{L_j}(P_{L_j}, \sigma).$$

Let $(\boldsymbol{v}_{L_{j_\sigma}}, \boldsymbol{v}'_{L_{j_\sigma}}, N_{L_{j_\sigma}})$ be the transition tuple for σ in the j^{th} low-level $\boldsymbol{\mathcal{G}}_{L_j}$, then the predicate $f_{L_j\sigma}$ can be computed symbolically by

$$f_{L_j\sigma} = \exists \boldsymbol{v}'_{L_{j_\sigma}}(N_{L_{j_\sigma}} \wedge (P_{L_j}[\boldsymbol{v}_{L_{j_\sigma}} \rightarrow \boldsymbol{v}'_{L_{j_\sigma}}])).$$

6.5.1 Simplifying Control Predicates

Let $\mathbf{G} := (Q, \Sigma, \delta, q_0, Q_m)$ be the DES tuple for the synchronous product of all the DES in system $\mathbf{\Phi}$. If we use local control predicates as in Figure 6.5, then a state $q \in Q$ (in the form of Equation 6.8) can be reached in the controlled system only if $(z_H, y_H, x_1, \ldots, x_n) \models P_H$ and for all $j \in \{1, \ldots, n\}, (z_{L_j}, y_{L_j}, x_j) \models P_{L_j}$. Based on this fact, we can simplify the control predicate for all $\sigma \in \Sigma_c$ as follows:

1. $\sigma \in \Sigma_c \cap (\Sigma_H \cup \Sigma_{R_1} \cup \cdots \cup \Sigma_{R_n})$

   If $\sigma \in \Sigma_c \cap (\Sigma_H \cup \Sigma_{R_1} \cup \cdots \cup \Sigma_{R_n})$, f_{H_σ} is required only for the state set $\{q \in Q | (z_H, y_H, x_1, \ldots, x_n) \models P_H\}$. For the other states in Q, the return value of f_{H_σ} can either be 0 or 1, as these states are unreachable.

   For a state $q \in Q$ in the form of Equation 6.8, let $q_H := (z_H, y_H, x_1, \ldots, x_n)$ be the high-level part of q. Let $d_{H_\sigma} \in Pred(Q_H)$ and $d_{H_\sigma} := P_H$. A predicate $f'_{H_\sigma} \in Pred(Q_H)$ satisfying $d_{H_\sigma} \wedge f'_{H_\sigma} \equiv d_{H_\sigma} \wedge f_{H_\sigma}$ will have the same effect as f_{H_σ} has, because for state q, if $d_{H_\sigma}(q_H) \equiv 1$, then $f_{H_\sigma} \equiv f'_{H_\sigma}$, otherwise we do not care the value of $f_{H_\sigma}(q_H)$ as q is unreachable. In other words, f'_{H_σ} and f_{H_σ} are *equal on the domain defined by the constraint* d_{H_σ} [12].

   Predicate f'_{H_σ} can be computed by the function $bdd\_simplify$ in the BDD package we are using when passed d_{H_σ} and f_{H_σ} as arguments.[6]

2. $\sigma \in \Sigma_c \cap (\Sigma_{L_j} \cup \Sigma_{A_j}), j \in \{1, \ldots, n\}$

   If $\sigma \in \Sigma_c \cap (\Sigma_{L_j} \cup \Sigma_{A_j})$, then $f_{L_{j\sigma}}$ is required only for the state set $\{q \in Q | (z_{L_j}, y_{L_j}, x_{L_j}) \models P_{L_j}\}$. For the other states in Q, the return value of $f_{L_{j_\sigma}}$ can either be 0 or 1, as those states are unreachable.

[6] Usually, the BDD for f'_{H_σ} is smaller than the BDD for f_{H_σ}, but according to the documentation of the $BuDDy$ package, it is not guaranteed.

For a state $q \in Q$ in the form of Equation 6.8, let $q_{L_j} := (z_{L_j}, y_{L_j}, x_{L_j})$ be the j^{th} low-level part of q. Let $d_{L_{j\sigma}} \in Pred(Q_{L_j})$ and $d_{L_{j\sigma}} := P_{L_j}$. A predicate $f'_{L_{j\sigma}} \in Pred(Q_{L_j})$ satisfying $d_{L_{j\sigma}} \wedge f'_{L_{j\sigma}} \equiv d_{L_{j\sigma}} \wedge f_{L_{j\sigma}}$ will have the same effect as $f_{L_{j\sigma}}$ has, because for state q, if $d_{L_{j\sigma}}(q_{L_j}) \equiv 1$, then $f_{L_{j\sigma}} \equiv f'_{L_{j\sigma}}$, otherwise we do not care the value of $f_{L_{j\sigma}}(q_{L_j})$ as q is unreachable. That is, $f'_{L_{j\sigma}}$ and $f_{L_{j\sigma}}$ are equal on the domain defined by the constraint $d_{L_{j\sigma}}$.

Predicate $f'_{L_{j\sigma}}$ can be computed by the function $bdd\_simplify$ in the BDD package we are using when passed $d_{L_{j\sigma}}$ and $f_{L_{j\sigma}}$ as arguments.

For system $\mathbf{\Phi}$, let $\sigma \in \Sigma_c$. If σ is not permitted at a state in a plant component means that σ can not happen at that state physically, then the control predicates $f_{H\sigma}$ or $f_{L_{j_\sigma}}$ ($j \in \{1, \ldots, n\}$) can be further simplified. In the control diagram in Figure 6.5, if $\sigma \in \Sigma_c \cap (\Sigma_H \cup \Sigma_{R_1} \cup \cdots \cup \Sigma_{R_n})$, $f_{H\sigma}$ has an actual effect only at a state $q \in Q$ (in the form of Equation 6.8) with $\eta_H(y_H, \sigma)!$, i.e. σ transition is defined at the high-level plant part of q. Also if $\sigma \in \Sigma_c \cap (\Sigma_{L_j} \cup \Sigma_{A_j})$, then $f_{L_{j_\sigma}}$ has an actual effect only when a state $q \in Q$ (of the form of Equation 6.8) with $\eta_{L_j}(y_{L_j}, \sigma)!$ i.e. σ transition is defined at the j^{th} low-level plant part of q.

For system $\mathbf{\Phi}$, let $\sigma \in \Sigma_c$. Assume σ can not happen if a plant component does not permit it. By the above analysis, we have the following:

1. $\sigma \in \Sigma_c \cap (\Sigma_H \cup \Sigma_{R_1} \cup \cdots \cup \Sigma_{R_n})$

   If $\sigma \in \Sigma_c \cap (\Sigma_H \cup \Sigma_{R_1} \cup \cdots \cup \Sigma_{R_n})$, f_{H_σ} is required only for the state set $\{q \in Q | (z_H, y_H, x_1, \ldots, x_n) \models P_H \ \& \ \eta_H(y_H, \sigma)!\}$. For the other states in Q, the return value of f_{H_σ} can either be 0 or 1 as either the states are unreachable or σ can not physically occur in the high-level plant.

   For a state $q \in Q$ in the form of Equation 6.8, let $q_H := (z_H, y_H, x_1, \ldots, x_n)$ be the high-level part of q. Let $d'_{H_\sigma} \in Pred(Q_H)$ and $d'_{H_\sigma} := P_H \wedge pr(\{q_H \in$

$Q_H|\eta_H(y_H,\sigma)!\}$). A predicate $f''_{H_\sigma} \in Pred(Q_H)$ satisfying $d'_{H_\sigma} \wedge f''_{H_\sigma} \equiv d'_{H_\sigma} \wedge f_{H_\sigma}$ will have the same effect as f_{H_σ} has, because for the state q, if $d'_{H_\sigma}(q_H) \equiv 1$, then $f_{H_\sigma} \equiv f''_{H_\sigma}$, otherwise we do not care the value of $f_{H_\sigma}(q_H)$ as either q is unreachable or σ can not physically occur in the high-level plant. That is, f''_{H_σ} and f_{H_σ} are equal on the domain defined by the constraint d'_{H_σ}.

We can also simplify f_{H_σ} as f'''_{H_σ} in the following proposition.

Proposition 6.2. *For system* $\mathbf{\Phi}$, *let* $\sigma \in \Sigma_c \cap (\Sigma_H \cup \Sigma_{R_1} \cup \cdots \cup \Sigma_{R_n})$. *For each* $q \in Q$ *in the form of Equation 6.8, let* $q_H := (z_H, y_H, x_1, \ldots, x_n)$ *be the high-level part of* q. *Let* $d''_{H_\sigma} \in Pred(Q_H)$ *and* $d''_{H_\sigma} := P_H \wedge pr(\{q_H \in Q_H|\delta_H(q_H,\sigma)!\})$. *Let* $F_{H_\sigma} \in Pred(Q_H)$ *satisfying* $d''_{H_\sigma} \wedge F_{H_\sigma} \equiv d''_{H_\sigma} \wedge f_{H_\sigma}$, *then* $f'''_{H_\sigma} := F_{H_\sigma} \wedge pr(\{q_H \in Q_H|\zeta_H \times \xi_1^h \times \ldots \times \xi_n^h((z_H, x_1, \ldots, x_n), \sigma)!\})$ *and* f_{H_σ} *are also equal on the domain defined by the constraint* d'_{H_σ}.

proof:

Sufficient to show that $d'_{H_\sigma} \wedge f'''_{H_\sigma} \equiv d'_{H_\sigma} \wedge f_{H_\sigma}$. We show this by a series of transformations starting from the left hand side of the equation.

$d'_{H_\sigma} \wedge f'''_{H_\sigma}$

$\equiv d'_{H_\sigma} \wedge F_{H_\sigma} \wedge pr(\{q_H \in Q_H|\zeta_H \times \xi_1^h \times \ldots \times \xi_n^h((z_H, x_1, \ldots, x_n), \sigma)!\})$

$\equiv P_H \wedge pr(\{q_H \in Q_H|\eta_H(y_H,\sigma)!\}) \wedge F_{H_\sigma} \wedge$

$\quad pr(\{q_H \in Q_H|\zeta_H \times \xi_1^h \times \ldots \times \xi_n^h((z_H, x_1, \ldots, x_n), \sigma)!\})$, by definition of d'_{H_σ}

$\equiv P_H \wedge pr(\{q_H \in Q_H|\delta_H((z_H, y_H, x_1, \ldots, x_n), \sigma)!\}) \wedge F_{H_\sigma}$

$\equiv d''_{H_\sigma} \wedge F_{H_\sigma}$, by definition of d''_{H_σ}

$\equiv d''_{H_\sigma} \wedge f_{H_\sigma}$, as $d''_{H_\sigma} \wedge F_{H_\sigma} \equiv d''_{H_\sigma} \wedge f_{H_\sigma}$

$\equiv P_H \wedge pr(\{q_H \in Q_H|\delta_H(q_H,\sigma)!\}) \wedge f_{H_\sigma}$, by definition of d''_{H_σ} (1)

By the definition of f_{H_σ}, we know that for all $q_H \in Q_H$, $f_{H_\sigma}(q_H) \equiv 1$ if and only if $\delta_H(q_H, \sigma) \models P_H$, which implies that $\delta_H(q_H, \sigma)!$. By the definition of $\preceq$, we thus have

$$f_{H_\sigma} \preceq pr(\{q_H \in Q_H | \delta_H(q_H, \sigma)!\}) \tag{2}$$

By (2), we can conclude that (1) is equivalent to $P_H \wedge f_{H_\sigma}$ (3)

We know that for all $q_H \in Q_H, \delta_H(q_H, \sigma)! \Rightarrow \eta_H(y_H, \sigma)!$, so we have

$$pr(\{q_H \in Q_H | \delta_H(q_H, \sigma)!\}) \preceq pr(\{q_H \in Q_H | \eta_H(y_H, \sigma)!\}) \tag{4}$$

By (2) and (4), we have

$$f_{H_\sigma} \preceq pr(\{q_H \in Q_H | \eta_H(y_H, \sigma)!\}). \tag{5}$$

By (5), we know that (3) is equivalent to

$P_H \wedge pr(\{q_H \in Q_H | \eta_H(y_H, \sigma)!\}) \wedge f_{H_\sigma}$

$\equiv d'_{H_\sigma} \wedge f_{H_\sigma}$, by definition of d'_{H_σ}.

□

2. $\sigma \in \Sigma_c \cap (\Sigma_{L_j} \cup \Sigma_{A_j}), j \in \{1, \ldots, n\}$

If $\sigma \in \Sigma_c \cap (\Sigma_{L_j} \cup \Sigma_{A_j})$, then $f_{L_{j\sigma}}$ is required only for the state set $\{q \in Q | (z_{L_j}, y_{L_j}, x_{L_j}) \models P_{L_j} \ \& \ \eta_{L_j}(y_{L_j}, \sigma)!\}$. For the other states in Q, the return value of $f_{L_{j_\sigma}}$ can either be 0 or 1, as either the states are unreachable or σ can not physically occur in the j^{th} low-level plant.

For a state $q \in Q$ in the form of Equation 6.8, let $q_{L_j} := (z_{L_j}, y_{L_j}, x_{L_j})$ be the j^{th} low-level part of q. Let $d'_{L_{j\sigma}} \in Pred(Q_{L_j})$ and $d'_{L_{j\sigma}} := P_{L_j} \wedge pr(\{q_{L_j} \in Q_{L_j} | \eta_{L_j}(y_{L_j}, \sigma)!\})$. A predicate $f''_{L_{j\sigma}} \in Pred(Q_{L_j})$ satisfying $d'_{L_{j\sigma}} \wedge f''_{L_{j\sigma}} \equiv d'_{L_{j\sigma}} \wedge$

$f_{L_{j\sigma}}$ will have the same effect as $f_{L_{j\sigma}}$ has, because for the state q, if $d'_{L_{j\sigma}}(q_{L_j}) \equiv 1$, then $f_{L_{j\sigma}} \equiv f''_{L_{j\sigma}}$, otherwise we do not care the value of $f_{L_{j\sigma}}(q_{L_j})$ as either q is unreachable or σ can not physically occur in the j^{th} low-level plant. That is, $f''_{L_{j\sigma}}$ and $f_{L_{j\sigma}}$ are equal on the domain defined by the constraint $d'_{L_{j\sigma}}$.

We can also simplify $f_{L_{j\sigma}}$ as $f'''_{L_{j\sigma}}$ in the following proposition.

Proposition 6.3. *For system* $\mathbf{\Phi}$, *let* $j \in \{1, \ldots, n\}$ *and* $\sigma \in \Sigma_c \cap (\Sigma_{L_j} \cup \Sigma_{A_j})$. *For each* $q \in Q$ *in the form of Equation 6.8, let* $q_{L_j} := (z_{L_j}, y_{L_j}, x_{L_j})$ *be the* j^{th} *low-level part of* q. *Let* $d''_{L_{j\sigma}} := P_{L_j} \wedge pr(\{q_{L_j} \in Q_{L_j} | \delta_{L_j}(q_{L_j}, \sigma)!\})$. *Let* $F_{L_{j_\sigma}} \in Pred(Q_{L_j})$ *satisfying* $d''_{L_{j\sigma}} \wedge F_{L_{j_\sigma}} \equiv d''_{L_{j\sigma}} \wedge f_{L_{j_\sigma}}$, *then* $f'''_{L_{j_\sigma}} := F_{L_{j_\sigma}} \wedge pr(\{q_{L_j} \in Q_{L_j} | \zeta_{L_j} \times \xi^l_j((z_{L_j}, x_j), \sigma)!\})$ *and* $f_{L_{j_\sigma}}$ *are also equal on the domain defined by the constraint* $d'_{L_{j\sigma}}$.

proof:

Identical to the proof of Proposition 6.2 by substituting all the high-level predicates with their corresponding j^{th} low-level predicates, δ_H with δ_{L_j}, ζ_H with ζ_{L_j}, η_H with η_{L_j}, and $\xi^h_1 \times \cdots \xi^h_n$ with ξ^l_j.

□

Our software tool (source code in Appendix A) produces f_{H_σ}, f'_{H_σ} and f'''_{H_σ} for all events in $\Sigma_c \cap (\Sigma_H \cup \Sigma_{R_1} \cup \cdots \cup \Sigma_{R_n})$, and $f_{L_{j_\sigma}}$, $f'_{L_{j_\sigma}}$ and $f'''_{L_{j_\sigma}}$ for all events in $\Sigma_c \cap (\Sigma_{L_j} \cup \Sigma_{A_j})$ for all $j \in \{1, \ldots, n\}$. In the rest of this thesis, we will refer to f'_{H_σ} and $f'_{L_{j_\sigma}}$ as *prime simplified control predicates* and refer to f'''_{H_σ} and $f'''_{L_{j_\sigma}}$ as *triple-prime simplified control predicates.*

6.6 A Small Example

In this section, we present a small example for HISC synthesis and provide the final automata supervisors and control predicates. This example is only for the purpose of demonstrating algorithms, so it is a bit contrived.

This example is inspired from the transfer line example from [47]. The system diagram is shown in Figure 6.6. Components $m1$ and $m2$ are two machines. Component tu is a test unit. Components $b1$ and $b2$ are two buffers with capacity 1.

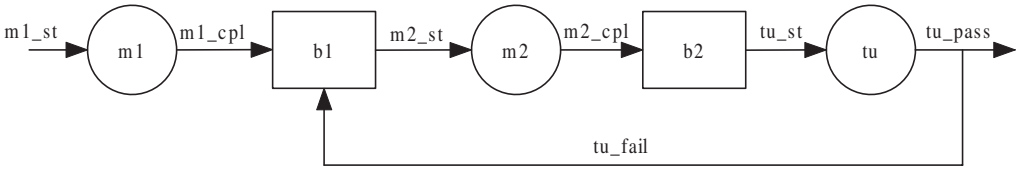


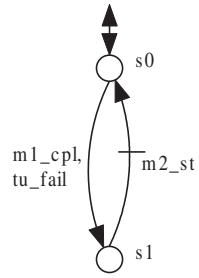
Figure 6.6: System diagram for small example

We model the system as an HISC system as follows. We treat $m1$ and $m2$ as two low-level subsystems with interfaces **intfm1** and **intfm2** as shown in Figure 6.7(e) and Figure 6.7(f), respectively.

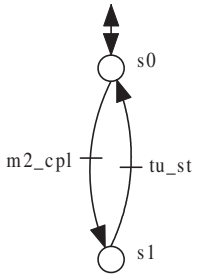
The low-level subsystem for low-level $m1$ is composed of one plant component, **low_m1**(Figure 6.7(g)), and the low-level subsystem for low-level $m2$ is composed of one plant component, **low_m2**(Figure 6.7(h)).

The high-level subsystem is composed of one plant component **high_tu** as shown in Figure 6.7(d) and three specification components **high_m2**, **high_b1** and **high_b2** as shown in Figure 6.7(c), Figure 6.7(a) and Figure 6.7(b) respectively. The specification **high_m2** tells that a part must be completely processed by $m2$. The specification **high_b1** and **high_b2** are used to control the underflow and overflow of buffer b1 and buffer b2.

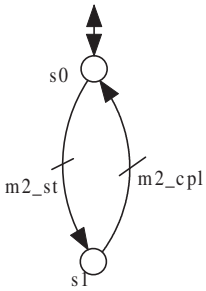
The event partition is shown as follows:



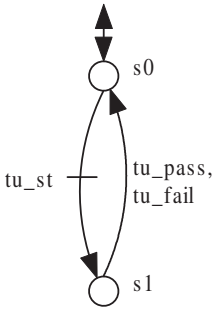
(a) high_b1 (spec)



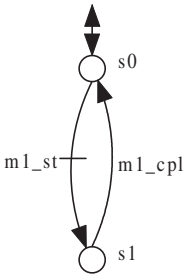
(b) high_b2 (spec)



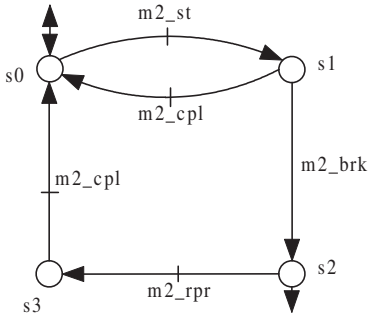
(c) high_m2 (spec)



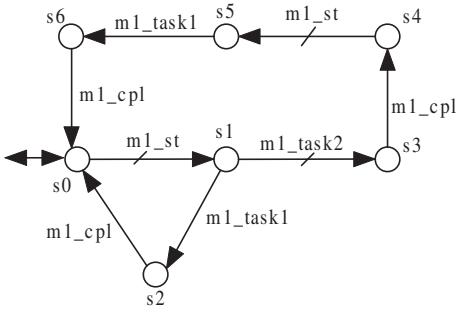
(d) high_tu (plant)



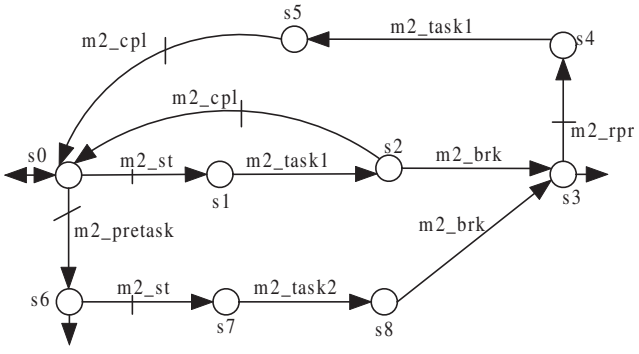
(e) Interface to low level m1



(f) Interface to low level m2



(g) low_m1 (plant)



(h) low_m2 (plant)

Figure 6.7: The small example

$\Sigma_H := \{tu\_st, tu\_pass, tu\_fail\}$

$\Sigma_{R_1} := \{m1\_st\}$, $\Sigma_{A_1} := \{m1\_cpl\}$

$\Sigma_{R_2} := \{m2\_st, m2\_rpr\}$, $\Sigma_{A_2} := \{m2\_cpl, m2\_brk\}$

$\Sigma_{L_1} := \{m1\_task1, m1\_task2\}$, $\Sigma_{L_2} := \{m2\_pretask, m2\_task1, m2\_task2\}$

By using our software tool, we synthesized trim automata supervisors for the high-level and low-levels $m1$ and $m2$. They are shown in Figure 6.8, Figure 6.9, and Figure 6.10 respectively.

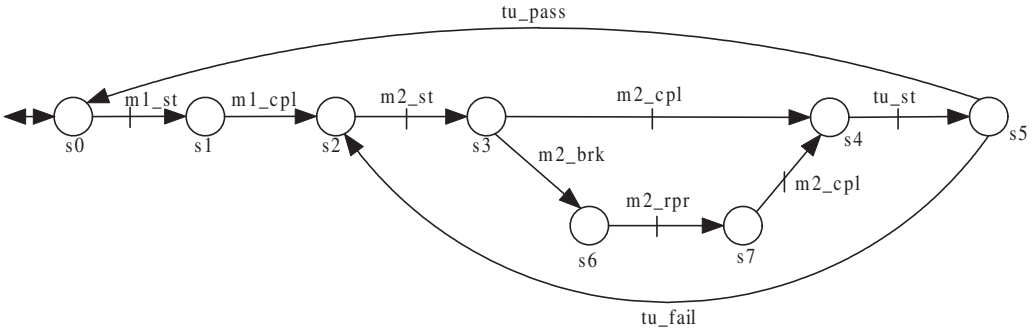


Figure 6.8: Synthesized high-level proper supervisor

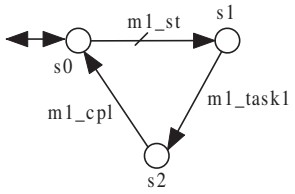


Figure 6.9: Synthesized low-level proper supervisor for low-level $m1$

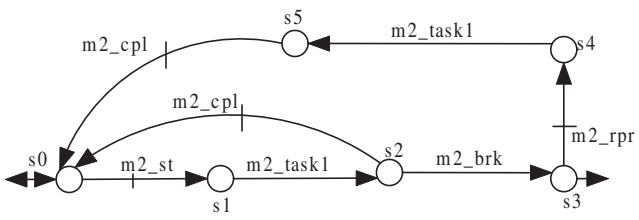


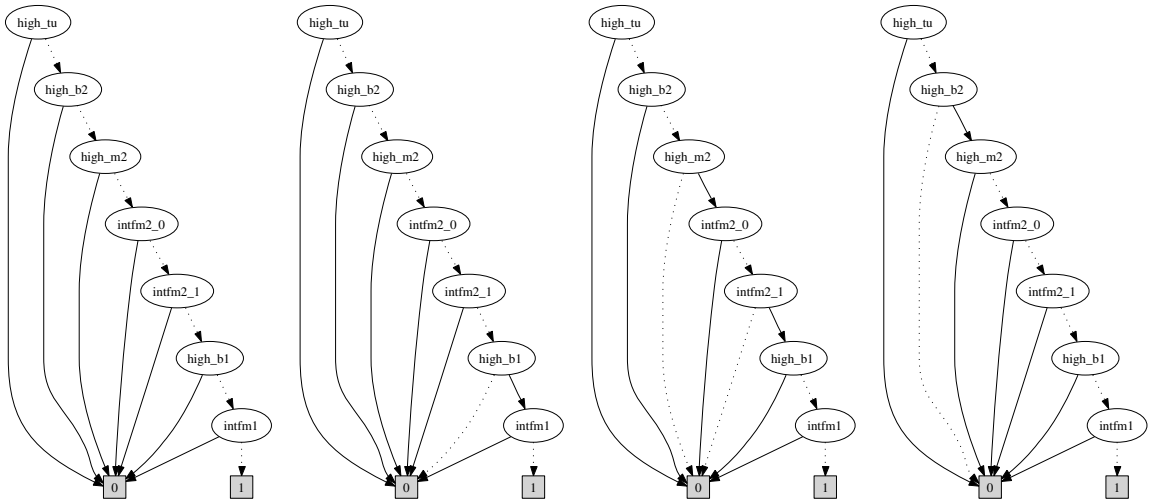
Figure 6.10: Synthesized low-level proper supervisor for low-level $m2$

Although event $m2\_cpl$ is controllable, notice that the high-level supervisor can not disable it when **intfm2** is at state $s1$ or $s3$. This is because $m2\_cpl$ is an answer event and the high-level containing this supervisor must satisfy Point 3 of the interface-consistent condition (Definition 3.5). However, if we treat this HISC system as a flat system, a synthesized supervisor could allow event $m2\_st$ when buffer $b2$ is full. In order to prevent the overflow of buffer $b2$, the supervisor could then disable event $m2\_cpl$. This would have the effect of allowing the low-level to do the actual operation but not reporting the result. Thus in general, we can not use a normal flat synthesis algorithm to synthesize a flat supervisor for an HISC system model.

From the low-level DES **low_m1** and interface DES **intfm1**, it is clear that **low_m1** $\parallel$ **intfm1** = **low_m1**. By inspecting the DES **low_m1**, we see that a string reaching state $s4$ can not reach a marker state by a string composed of only low-level($m1$) events. Therefore, we need to trim off state $s4$. The resulting trim DES is the final supervisor for low-level $m1$.

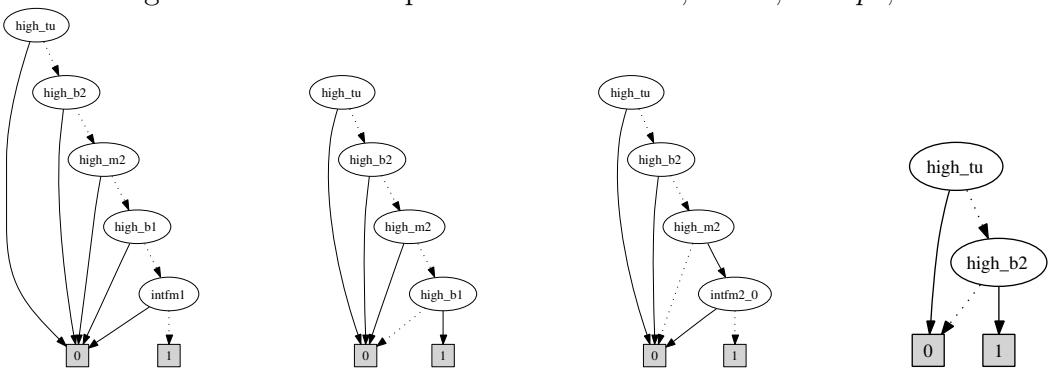
From the low-level DES **low_m2** and interface DES **intfm2**, we also have **low_m2** $\parallel$ **intfm2** = **low_m2**. By inspecting the DES **low_m2**, we can not find a string l composed of only low-level ($m2$) events such that $m2\_pretask\ m2\_st\ l\ m2\_cpl$ belongs to the closed language of low-level $m2$. Therefore, $s6$ must be trimmed off. The resulting trim DES is the final supervisor for low-level $m2$.

The BDD representations for control predicates $f_{H_{m1\_st}}, f_{H_{m2\_st}}, f_{H_{m2\_rpr}}, f_{H_{tu\_st}}$ are shown in Figure 6.11. In the diagrams, dotted line means the variable at its source is assigned to be 0 and solid line means the variable at its source is assigned to be 1. The states in each of the component DES are encoded as the number in their names.(e.g. s0 is encoded as 0, s1 is encoded as 1, ...). As **intfm2** contains four states, we need two binary variables for it. The binary value for a state is encoded with least significant bit first (e.g. $s1$ is encoded as 10 (intfm2_0 = 1, intfm2_1 = 0)



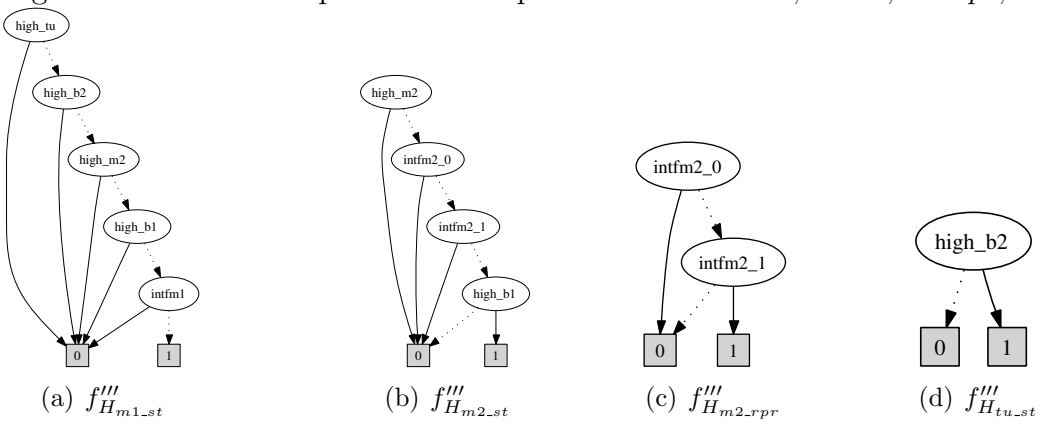
(a) $f_{H_{m1\_st}}$ (b) $f_{H_{m2\_st}}$ (c) $f_{H_{m2\_rpr}}$ (d) $f_{H_{tu\_st}}$

Figure 6.11: Control predicates for $m1\_st, m2\_st, m2\_rpr, tu\_st$



(a) $f'_{H_{m1\_st}}$ (b) $f'_{H_{m2\_st}}$ (c) $f'_{H_{m2\_rpr}}$ (d) $f'_{H_{tu\_st}}$

Figure 6.12: Prime simplified control predicates for $m1\_st, m2\_st, m2\_rpr, tu\_st$



(a) $f'''_{H_{m1\_st}}$ (b) $f'''_{H_{m2\_st}}$ (c) $f'''_{H_{m2\_rpr}}$ (d) $f'''_{H_{tu\_st}}$

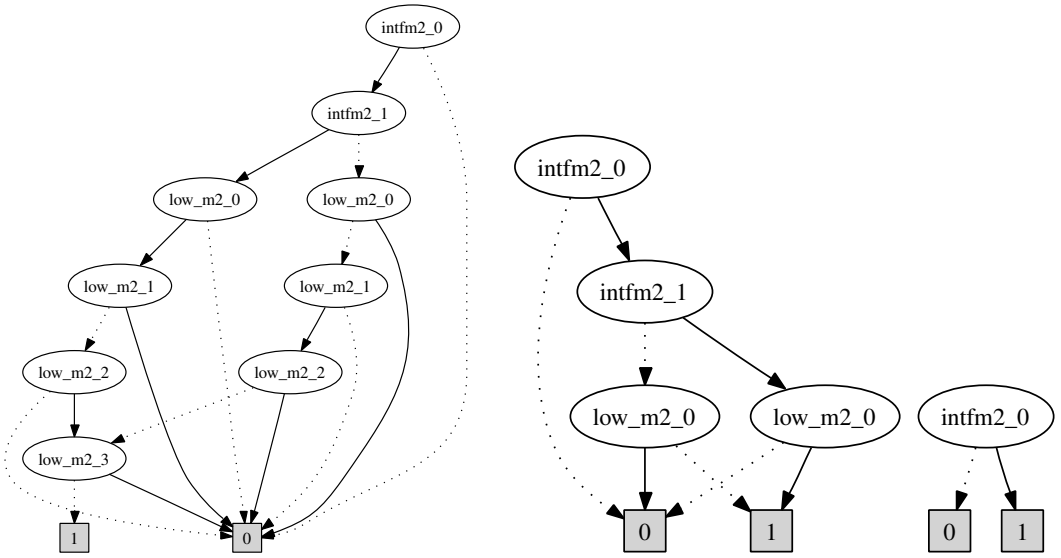
Figure 6.13: Triple-prime simplified control predicates for $m1\_st, m2\_st, m2\_rpr, tu\_st$

and $s2$ is encoded as 01 (intfm2_0 = 0, intfm2_1 = 1) in DES **intfm2**).

The BDD representations for prime simplified control predicates and triple-prime simplified control predicates are shown in Figure 6.12 and Figure 6.13, respectively.

For low-level $m1$, the only control predicate is $f_{L_{1m1\_task2}}$, which is always equal to $false$, so are $f'_{L_{1m1\_task2}}$ and $f'''_{L_{1m1\_task2}}$.

For low-level $m2$, the control predicate $f_{L_{2m2\_pretask}}$ is also always $false$. The BDD representations for the control predicate $f_{L_{2m2\_cpl}}$ and its simplified versions $f'_{L_{2m2\_cpl}}$ and $f'''_{L_{2m2\_cpl}}$ are shown in Figure 6.14.



(a) $f_{L_{2m2\_cpl}}$ (b) $f'_{L_{2m2\_cpl}}$ (c) $f'''_{L_{2m2\_cpl}}$

Figure 6.14: The control predicates for $m2\_cpl$.

Chapter 7

The AIP Example

In order to demonstrate our approach, in this chapter we give an example with a large and complex high-level subsystem, which is modified from the AIP example[1] in [21,23,26]. First we give an introduction of the AIP and our new control specifications, which are extensions of the one used in the original example. Next we present a set of plant DES and a set of modular supervisor DES and then verify each subsystem satisfies its corresponding conditions.After that, we relax the system by removing one restriction of the modular supervisors in the high-level and then synthesize a high-level proper supervisor. Finally, we report our results. The reason we focus on the high-level is that it is usually the limiting factor as it is often more complex than the low-levels and usually increases in complexity as we add new low-levels to an existing system. In next chapter, we will give an example with large and complex low-level subsystems.

[1] In this chapter, for convenience we cite some diagrams directly or with some minor changes from [21, 23, 26] with permission.

7.1 Introduction of the AIP

The AIP is a highly automated manufacturing system, and was first modeled as a discrete event system using modular supervisory control theory in [5] (Brandin and Charbonnier) and [7](Charbonnier).[2] Leduc then modeled the AIP with more detailed behavior added in [21,23,26] and verified the properties of controllability and nonblocking using the HISC method.

The AIP system includes a central loop (CL) conveyor, and four external loop (EL) conveyors. Between each external loop and the central loop, there is a transport

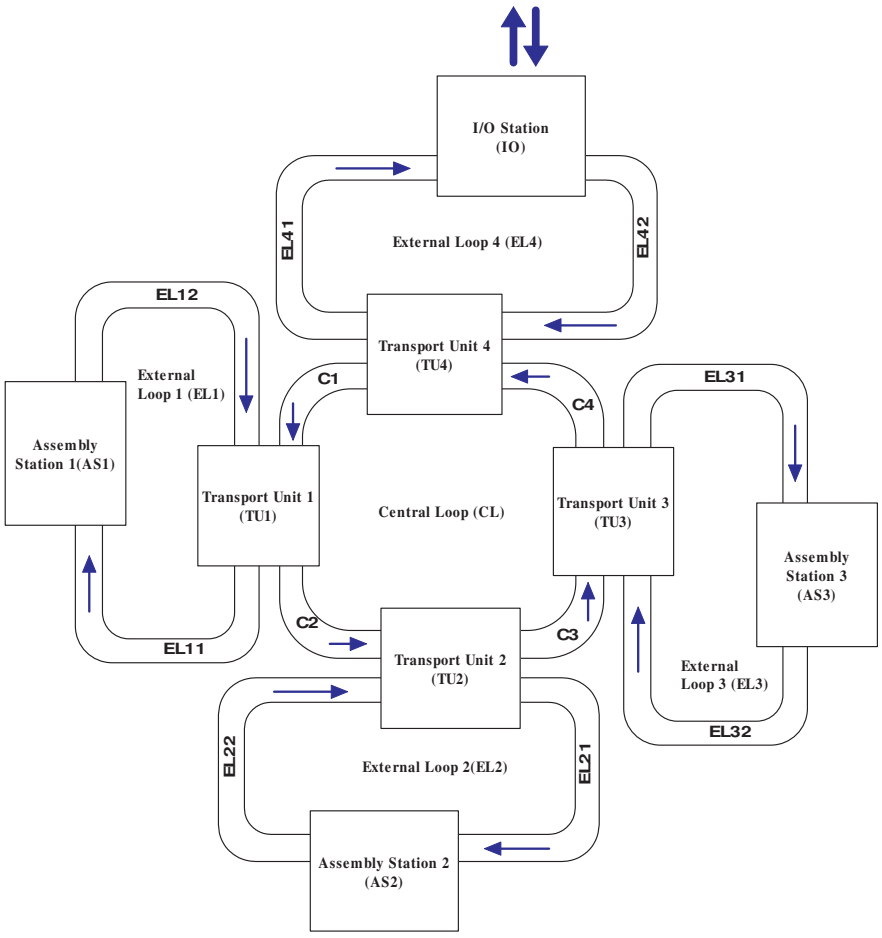


Figure 7.1: The AIP system architecture(from [21])

[2]What the author really read is an English version translated by R. Leduc.

unit (TU) to transport pallets between them. At external loops 1, 2 and 3, they each have an assembly station (AS) including a robot to process pallets. At external loop 4, an Input/Output (I/O) station is there to allow pallets to enter and leave the system. Two types of pallets, Type1 and Type2, can be processed by the AIP system. The system architecture is shown in Figure 7.1. In the diagram, the arrows indicate the direction a pallet can move on a given loop.

The four transport units separate the central loop into four areas: C1, C2, C3 and C4. For each external loop conveyor, there are also two areas separated by either an assembly station (external loop 1, 2 and 3) or by an I/O station (external loop 4). These areas could be thought of as a buffer area. See Figure 7.1 for labels for these areas.

All three assembly stations have the same components and structure (see Figure 7.2) but different functions. The assembly station AS1 is capable of doing Task1A

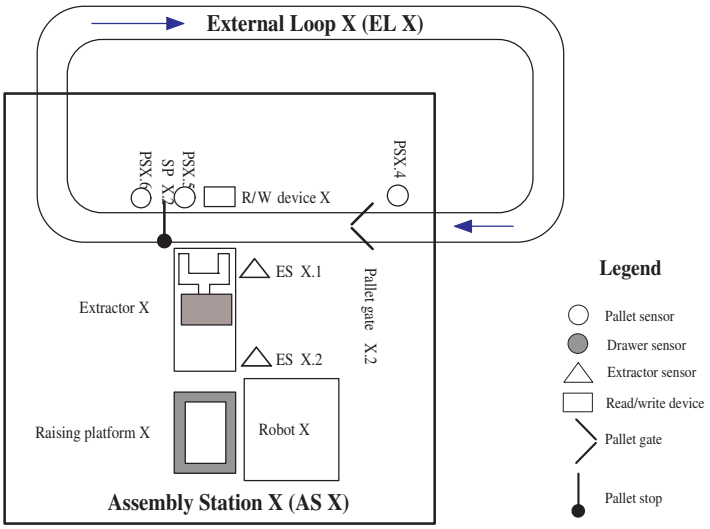


Figure 7.2: Assembly station of external loop X = 1, 2, 3 (from [21])

and Task1B, and AS2 is capable of doing Task2A and Task2B, while AS3 can do all the tasks. AS3 also repairs assembly errors to pallets. AS1 and AS2 can break down,

while AS3 is assumed to never break down. AS3 can also substitute for AS1 and AS2 when they break down. In an assembly station, there are three pallet sensors (PS X.4, PS X.5, PS X.6) to detect a pallet arriving at the pallet gate, arriving at the pallet stop, or leaving the assembly station, respectively. Pallet gate X.2 can prevent other pallets from entering the assembly station while the station is busy. R/W device X can read and write information on the label of a pallet. Pallet stop SP X.2 is used to prevent the pallet from leaving the station when it has not yet been processed. Extractor X is able to transfer a pallet between the conveyor and the raising platform. Two extractor sensors (ES X.1, ES X.2) are used to detect the location of the extractor. Raising platform X is used to feed the pallet to the robot (Robot X) to be processed, and move the processed pallet to the extractor.

In a transport unit (see Figure 7.3), there are three pallet sensors (PS 5.X.1, PS 5.X.2, PS5.X.3) close to the central loop side to detect a pallet arriving at pallet

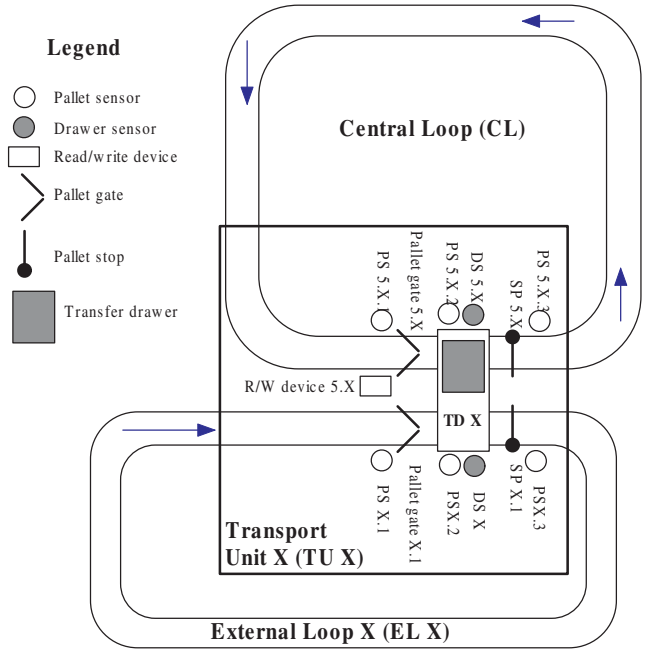


Figure 7.3: Transport unit for external loop X = 1, 2, 3, 4 (from [21])

gate 5.X, arriving at the transfer drawer (TD X) and leaving the transport unit to next area of the central loop, respectively. There are another three pallet sensors (PS X.1, PS X.2, PS X.3) close to the external loop X side to detect a pallet arriving at pallet gate X.1, arriving at transfer drawer (TD X), and leaving the transfer unit to the external loop X, respectively. A pallet coming from the central loop can be transferred to the external loop X or liberated to the next area of the central loop, while a pallet coming from the external loop can only be transferred to the central loop. Pallet gate 5.X and pallet gate X.1 can prevent a pallet from entering the transfer unit when it is busy. Two pallet stops (SP5.X, SP X.1) are used to control pallets leaving the transfer unit. Transfer drawer (TD X) moves pallets between the central loop and the external loop X. Two drawer sensors can detect the location of the drawer.

The I/O station should have similar components and structure as an assembly station except there is no robot, extractor and raising platform. Due to lack of detailed information and time limit, here we assume that there is a sensor to detect if a pallet is ready to enter the system and a sensor to detect if a pallet is ready to leave the system.

7.2 Control Specifications

Our control specifications for the AIP system are extended from the specifications in [21, 23]. For convenience, here we list all the specifications but the new ones begin with a '*'. We also assume that initially the AIP system is empty (i.e. no pallet in the system).

1. ***Input:** The type of the pallets entering the system must alternate, starting with Type1.

2. **Output:** The type of the pallets leaving the system must alternate starting with Type1.

3. **Routing:** Type1 pallet must first go to AS1 to undergo Task1 and then go to AS2 to undergo Task2. Type2 pallet must first go to AS2 to undergo Task2 and then go to AS1 to undergo Task1. Both types of pallets are allowed to leave the system only when both tasks are done.

4. **Assembly errors:** When the robot in AS1 or AS2 makes an assembly error, the R/W device in AS1 or AS2 will write assembly error information on the pallet label, and then AS3 will repair the pallet and AS1 and AS2 can do assembly task again.

5. **Assembly station breakdown:** When either AS1 or AS2 breaks down, all the pallets for that station will be routed to AS3 for assembly. However, when AS1 or AS2 is repaired, all the pallets not already in external loop 3 are rerouted to the original station.

6. **Assembly task ordering:** Assembly tasks are performed in a different order for pallets of different types. For a Type1 pallet, Task1A is performed before Task1B, and Task2A is performed before Task2B. For a Type2 pallet, Task1B is performed before task 1A, and Task2B is performed before task 2A.

7. **Maximum capacity of assembly stations:** At any time, only one pallet is allowed in a given assembly station.

8. ***Maximum capacity of I/O station:** At any time, only one pallet is allowed in the I/O station.

9. ***Maximum capacity of external loops:** At each area of each external loop, the maximum allowed number of pallets at a given time is three.

10. ***Maximum capacity of the central loops:** At each area of the central loop, the maximum allowed number of pallets at a given time is two.

7.3 System Structure

In [21, 23], the AIP system is modeled as a bi-level HISC system with one high-level and seven low-levels. The low-levels represent AS1, AS2, AS3, TU1, TU2, TU3, and TU4. In our example, we keep all the low-levels and add one more low-level for the I/O station. Therefore, in total we have one high-level and eight low-levels. The system structure is shown in Figure 7.4. As a large portion of our example will be the same as [21, 23, 26], we will only present what is new and direct readers to [21, 23, 26] for the remaining details.

The high-level is composed of the high-level subsystem $\mathbf{G}_H$ and 8 interfaces $\mathbf{G}_{I_j}, j \in \{1, \ldots, 8\}$. Each low-level j contains the j^{th} low-level subsystem $\mathbf{G}_{L_j}$ and

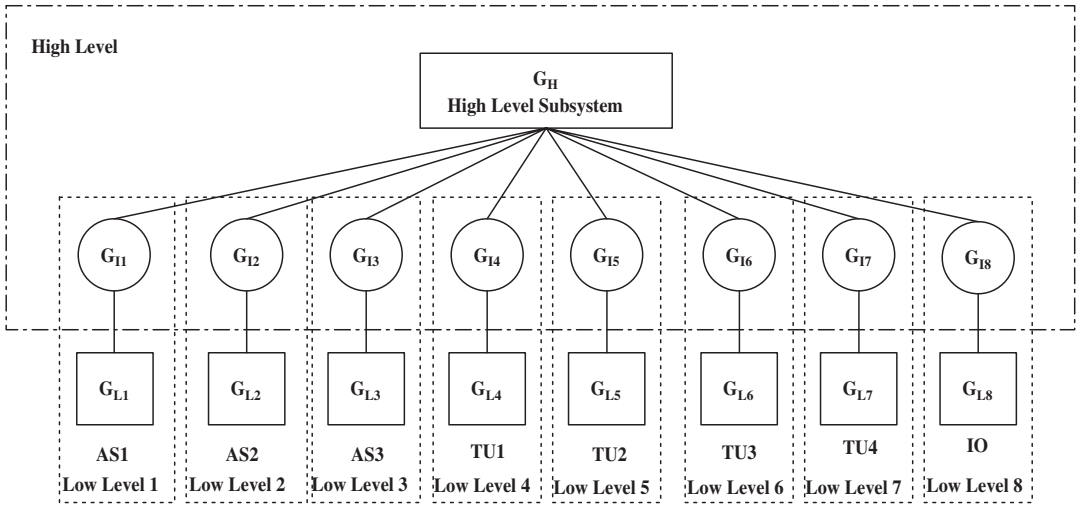


Figure 7.4: The AIP system structure

the j^{th} interface $\mathbf{G}_{I_j}$.

For the I/O station, we model the two sensors to detect that pallets are ready to enter or leave the system in the high-level. As we do not have the internal structure of the I/O station, we only provide an interface DES for the low-level 8 (I/O station) and have the low-level 8 subsystem containing only one plant component DES and set $\Sigma_{L_8} = \varnothing$. The plant DES contains only an initial state, which is also marked.

The event set for the whole AIP system $\Sigma := \dot{\cup}_{j\in\{1,\ldots,8\}}[\Sigma_{L_j}\dot{\cup}\Sigma_{R_j}\dot{\cup}\Sigma_{A_j}]\dot{\cup}\Sigma_H$ is defined based on the event partition given in [21]. The primed event set below stands for the corresponding event set from the AIP example in [21].

$\Sigma_H := (\Sigma'_H - \{PalletArvGEL\_2.AS3\}) \cup$

$\{QPalletOut.IO, IsPalletOut.IO, NoPalletOut.IO, QPalletType1In.IO,$

$QPalletType2In.IO, IsPalletType1In.IO, IsPalletType2In.IO,$

$NoPalletType1In.IO, NoPalletType2In.IO\}$

$\Sigma_{L_i} := \Sigma'_{L_i}$, where $i \in \{1, 2, 4, 5, 6, 7\}$

$\Sigma_{L_3} := \Sigma'_{L_3} \cup \{PalletArvGEL\_2.AS3\}$

$\Sigma_{L_8} := \varnothing$

$\Sigma_{R_k} := \Sigma'_{R_k}$, where $k \in \{1, \ldots, 7\}$

$\Sigma_{R_8} := \{MvInType1Pallet.IO, MvInType2Pallet.IO, MvOutPallet.IO\}$

$\Sigma_{A_k} := \Sigma'_{A_k}$, where $k \in \{1, \ldots, 7\}$

$\Sigma_{A_8} := \{CplMvInPallet.IO, CplMvOutPallet.IO\}$

In the DES diagrams in this chapter, initial states are identified by a thick circle, and marker states are filled in with gray. Uncontrollable events are shown in italic font, and controllable events are shown in normal font. For a given DES, its event set is taken to be that of the event labels shown on transitions in the diagram unless explicitly state otherwise.

7.4 The Interface DES

For low-level 1 to 7, we use the same interface as in [23, 26]. The interfaces for low-level 1 and 2 are shown in Figure 7.5 with k = AS1, AS2. The interface for low-level 3 (AS3) is shown in Figure 7.6. The interfaces for low-levels 4, 5, and 7 are shown in Figure 7.7 with q = TU1, TU2, TU4. The interface for low-level 6 (TU3) is shown in Figure 7.8. Finally, the interface for low-level 8 (I/O) is shown in Figure 7.9.

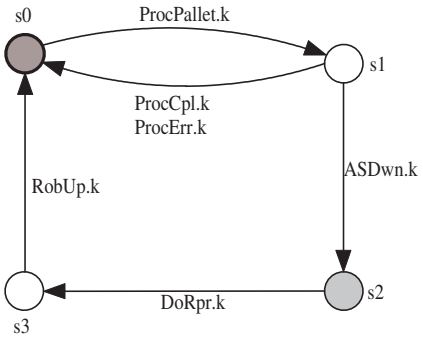


Figure 7.5: Interface to *low-level* $w = 1, 2$ (from [23])

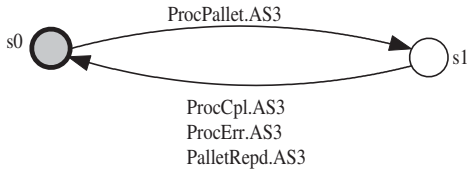


Figure 7.6: Interface to *low-level* 3 (from [23])

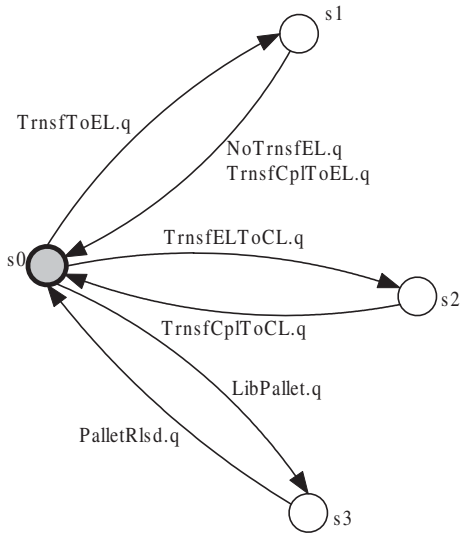


Figure 7.7: Interface to *low-level* $v = 4, 5, 7$ (from [23])

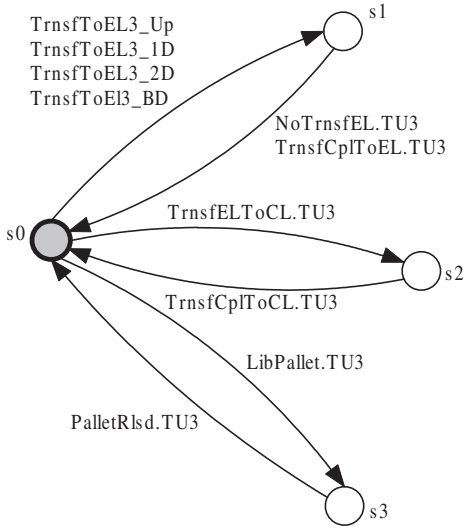


Figure 7.8: Interface to *low-level* 6 (from [23])

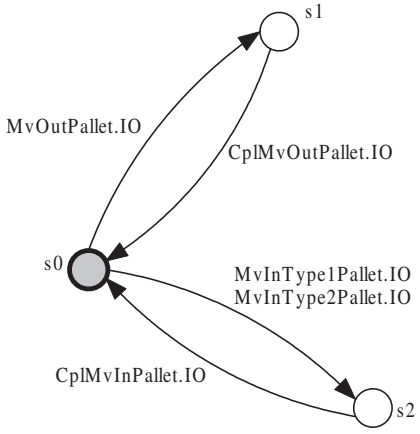


Figure 7.9: Interface to *low-level* 8

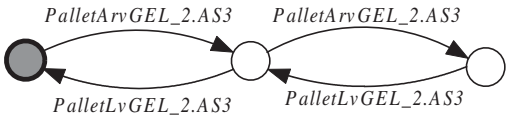


Figure 7.10: CapGateEL_2.AS3

7.5 Low-level Subsystems

For low-level 1 (AS1), 2 (AS2), 4 (TU1), 5 (TU2), 6 (TU3) and 7 (TU4), we use exactly the same models as in [23, 26]. For low-level 3 (AS3), we make some minor modifications. The reason for these modifications is that there was no capacity restriction on external loop 3 in [23]. In order to show the relationship between the event *ProcPallet.AS3* and *PalletArvGEL_2.AS3*, Leduc created a high-level plant component DES that was in Figure 12.3 in [21] (**PalletArvGateSenEL_2.AS3**). However, in this thesis, we will enforce a capacity restriction on external loop 3. This allows us to move this functionality to low-level 3 (AS3). The following shows how we modified the models for low-level 3.

- Replace the plant component **CapGateEL_2.AS3** by the one in Figure 7.10.
- Replace the supervisor component **OperateGateEL_2.AS3** by the one in Figure 7.11.
- Add a plant component **PalletArvGateSenEL_2.AS3** as shown in Figure 7.12.

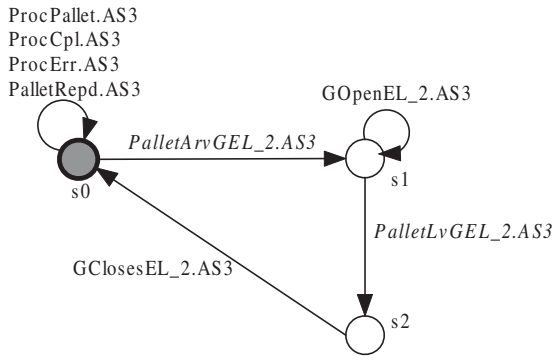


Figure 7.11: OperateGateEL_2.AS3

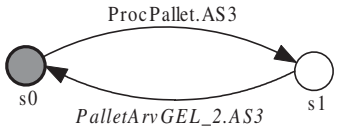


Figure 7.12: PalletArvGateSenEL_2.AS3

For low-level 8 (I/O), as described in section 7.3, we lack the information and time needed to model the subsystem in more detail, so we model its subsystem containing only a one-state plant DES as shown in Figure 7.13

Figure 7.13: *Low-level 8 Subsystem*

7.6 The High-level Subsystem

In this section, we present the high-level subsystem $\mathbf{G}_H$ for the AIP. The high-level controls the global behavior of the system, such as controlling the capacity of each area of the central loop and external loops, detecting the status of AS1 and AS2 and reporting the status to other low-levels. As our newly added control specifications primarily focus on the capacity of the conveyor areas, the high-level is significantly larger and more complicated than the one in [23].

To demonstrate our high-level verification algorithms and synthesis algorithms, we designed all the modular supervisors by hand to meet the control specifications and then verified that the system under the control of these supervisors satisfies all the

high-level conditions (e.g. level-wise controllable, level-wise nonblocking and interface consistent). In the next section, we will relax the restrictions and then synthesize a supervisor to satisfy the control specifications.

The AIP high-level subsystem $\mathbf{G}_H$ is composed of 19 plant components and 21 supervisor components, as shown in Figure 7.14. The high-level plant $\mathbf{G}_H^p$ and supervisor $\mathbf{S}_H$ are defined to be the synchronous product of the indicated automata.

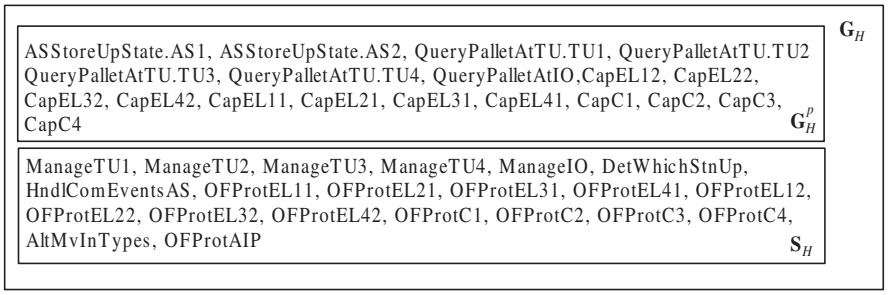


Figure 7.14: Component DES in the AIP high-level

7.6.1 Plant Components

We start from the set of DES **ASStoreUpSate.**k, where k = AS1, AS2, shown in Figure 7.15. These two DES are used to check the status of AS1 and AS2 and tell when the high-level is allowed to send the requests *ProcPallet.k* and *DoRpr.k*.[3]

The next set of DES we introduce are **QueryPalletAtTU.**i, where i = TU1, TU2, TU3, TU4, shown in Figure 7.16. The DES **QueryPalletAtTU.**i can tell if a pallet is waiting to enter the transport unit i from the central loop (by detecting the gate sensor PS 5.X.1 (X = 1 when i = TU1, ..., X = 4 when i = TU4)) or the related external loop (by detecting the gate sensor PS X.1).

[3]Actually we could remove those selfloop transitions on these two DES, since we now use *command-pair interface* for AS1 and AS2. The interfaces to low-level 1 and low-level 2 guarantee when the request events *ProcPallet.k* and *DoRpr.k* can happen.

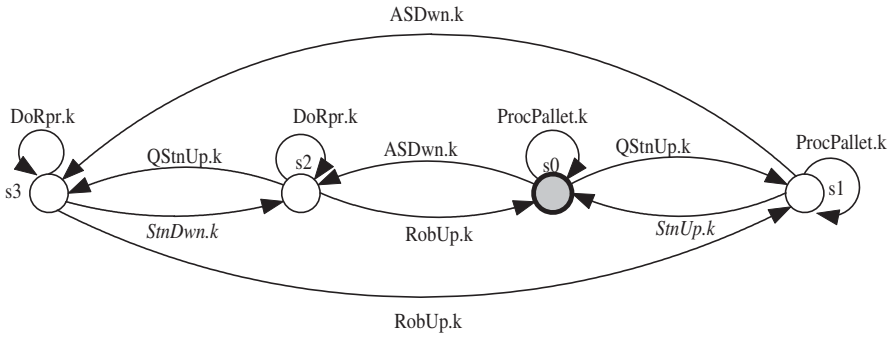


Figure 7.15: ASStoreUpState.k = AS1, AS2 (from [23])

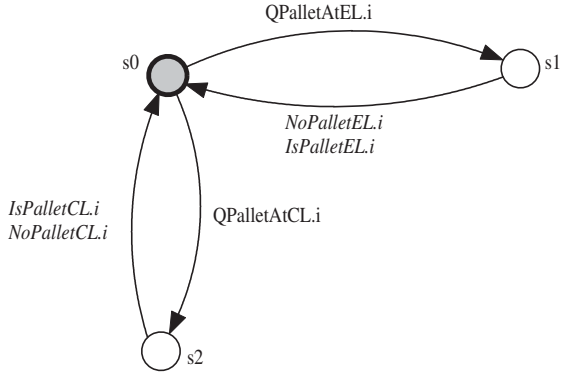


Figure 7.16: QueryPalletAtTU.i, i = TU1, TU2, TU3, TU4

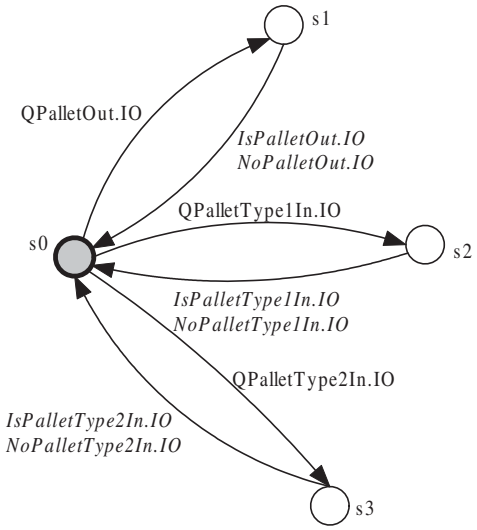
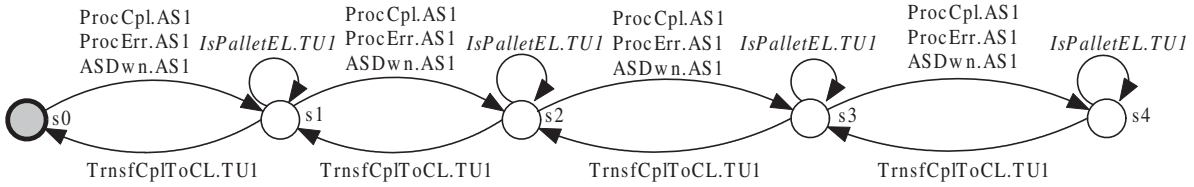


Figure 7.17: QueryPalletAtIO

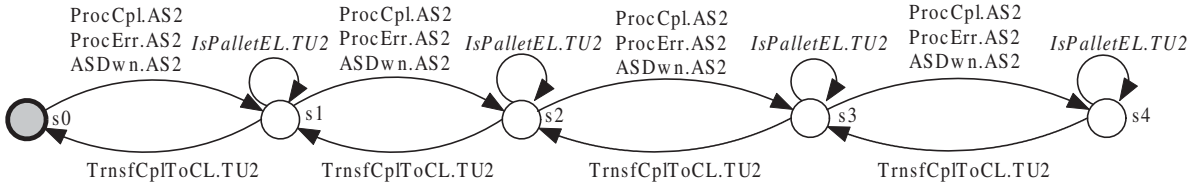
We now describe the DES **QueryPalletAtIO**, shown in Figure 7.17. This DES models the behavior of the the sensors in the I/O station. It provides a way to determine if a pallet is ready to leave the system or if a Type1 or Type2 pallet is ready to enter the system.

The next series of DES represent the fact that a pallet on an external loop can arrive at a transport unit only if it has been processed by the associated assembly station or brought in from the I/O station. They are **CapEL12, CapEL22, CapEL32** and **CapEL42** as shown in Figure 7.18. Theoretically, each of the four DES should

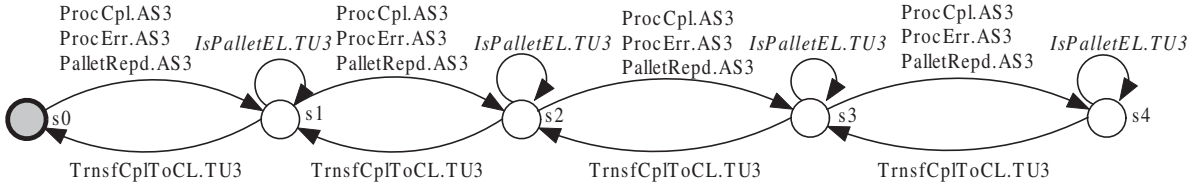
have infinite states. However, because the supervisors will limit the capacity of each of the conveyor areas EL12, EL22, EL32 and EL42 to three, we are safe to make each of them contain five states. Readers might ask why these DES have capacity of four, this is because we would like to show that the supervisors really prevent the fourth pallet from entering any of the above areas.



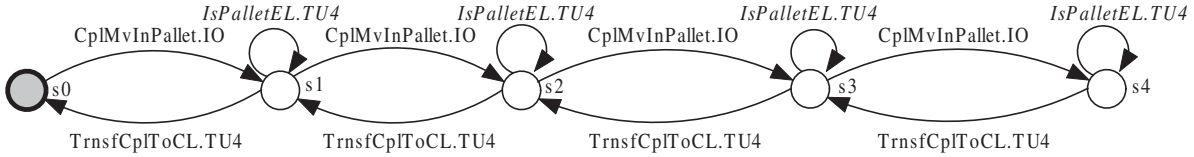
(a) CapEL12



(b) CapEL22



(c) CapEL32

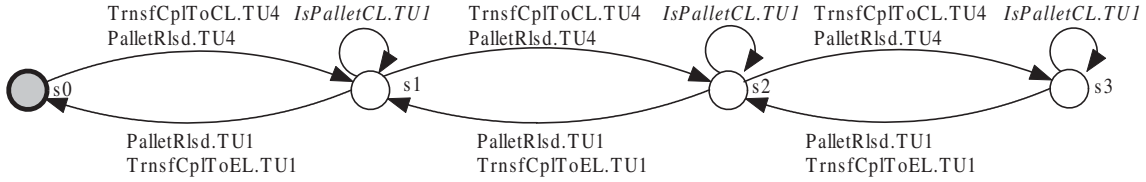


(d) CapEL42

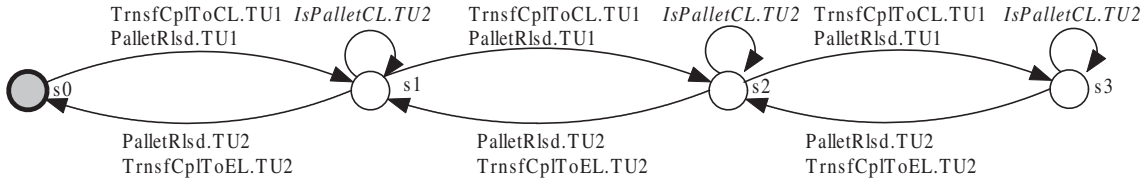
Figure 7.18: CapEL12, CapEL22, CapEL32, CapEL42

Now we describe the series of DES representing the fact that a pallet on the center loop can arrive at a transport unit from the central loop only if it has been liberated or

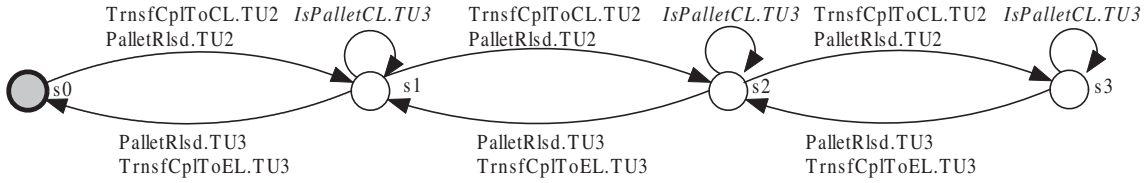
transferred from the external loop by the previous transport unit. They are **CapC1, CapC2, CapC2** and **CapC4** as shown in Figure 7.19. Each of them should also have infinite states, but because the supervisors will limit the capacity of each central loop area to two, we are safe to make each of them contain only 4 states.



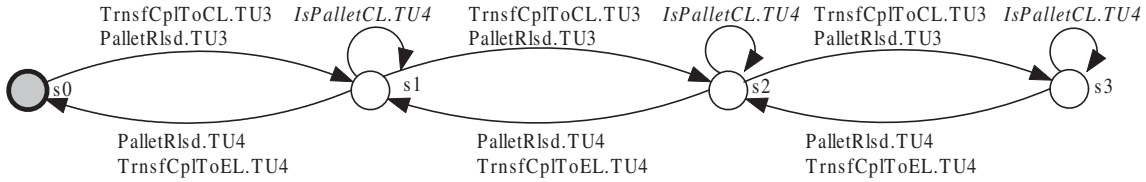
(a) CapC1



(b) CapC2



(c) CapC3

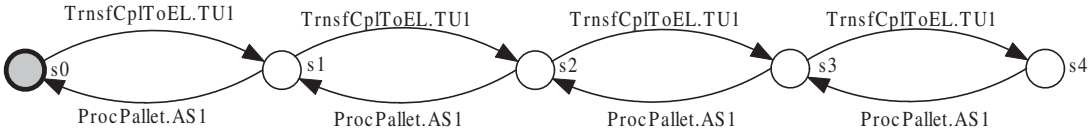


(d) CapC4

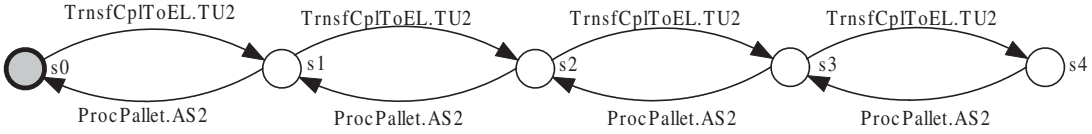
Figure 7.19: CapC1, CapC2, CapC3, CapC4

The plant components **CapEL11**, **CapEL21** and **CapEL31** (Figure 7.20(a), 7.20(b), 7.20(c)) represents the fact that a pallet can be processed by an assembly station only if it has been transferred to the associated external loop. The plant

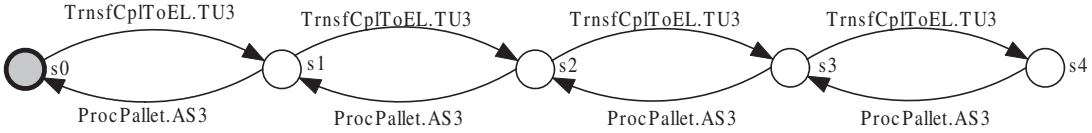
component **CapEL41** (Figure 7.20(d)) represents the fact that a pallet can arrive at the I/O station only if it has been transferred to external loop 4 by transport unit TU4. Again, these DES should have infinite states theoretically, but the supervisors will limit the capacity of these areas to 3, so we are safe to make it contain only 5 states.



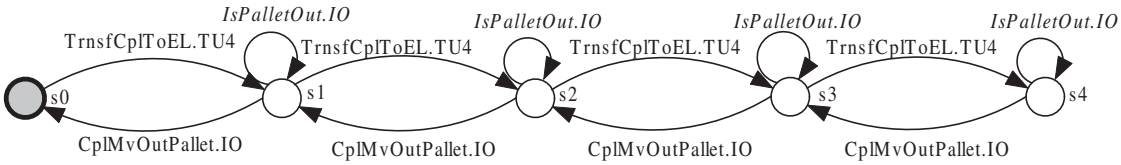
(a) CapEL11



(b) CapEL21



(c) CapEL31



(d) CapEL41

Figure 7.20: CapEL11, CapEL21, CapEL31, CapEL41

7.6.2 Supervisor Components

We now discuss the supervisor components for the AIP high-level subsystem. We first introduce the group of four supervisors which control the transport units, **Man-**

ageTU1, ManageTU2, ManageTU3 and **ManageTU4**. The following explains what they do.

- **ManageTU1** : If a pallet appears before TU1 at the central loop, it could be transferred to external loop 1 or liberated. It can be liberated if AS1 breaks down, or the conveyor area EL11 is at capacity (i.e. 3, determined by **OFProtEL11**), or low-level 4 (TU1) says to do so through the event *NoTransfEL.TU1*. If a pallet appears before TU1 at the external loop 1, then it must be transferred to the central loop. See Figure 7.21.

- **ManageTU2**: If a pallet appears before TU2 at the central loop, it could be transferred to external loop 2 or liberated. It can be liberated if AS2 breaks down, or the conveyor area EL21 is at capacity (i.e. 3, determined by **OFProtEL21**), or low-level 5 (TU2) says to do so through the event *NoTransfEL.TU2*. If a pallet appears before TU2 at the external loop 2, then it must be transferred to the central loop. See Figure 7.22.

- **ManageTU3**: If a pallet appears before TU3 at the central loop, it could be transferred to external loop 3 or liberated. If the conveyor area EL31 is at capacity (i.e. 3, determined by **OFProtEL31**), then the pallet should be liberated. Otherwise the break down status of AS1 and AS2 will be checked through supervisor component **DetWhichStnUp**. Then a corresponding request event with the status of AS1 and AS2 encoded into it will be sent to low-level 6 (TU3). The pallet should be liberated if low-level 5 responds with a *NoTransfEL.TU3* event. If a pallet appears before TU3 at external loop 3, then it must be transferred to the central loop. See Figure 7.23.

- **ManageTU4**: If a pallet appears before TU4 at the central loop, it could

be transferred to external loop 4 or liberated. It can be liberated if the conveyor area EL41 is at capacity (i.e. 3, determined by **OFProtEL41**), or low-level 7 (TU4) says to do so through the event *NoTransfEL.TU4* (e.g. not all tasks have been performed on the pallet). If a pallet appears before TU4 at external loop 4, then it must be transferred to the central loop. See Figure 7.24.

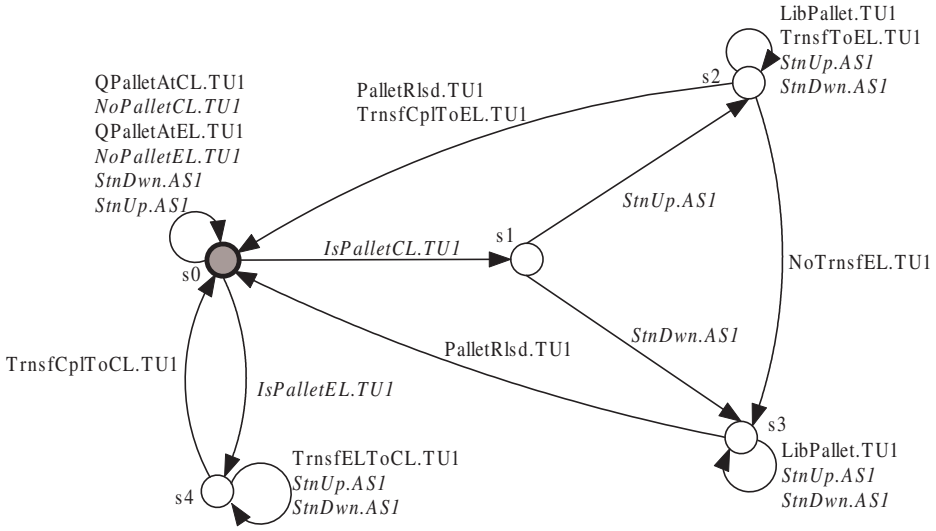


Figure 7.21: ManageTU1

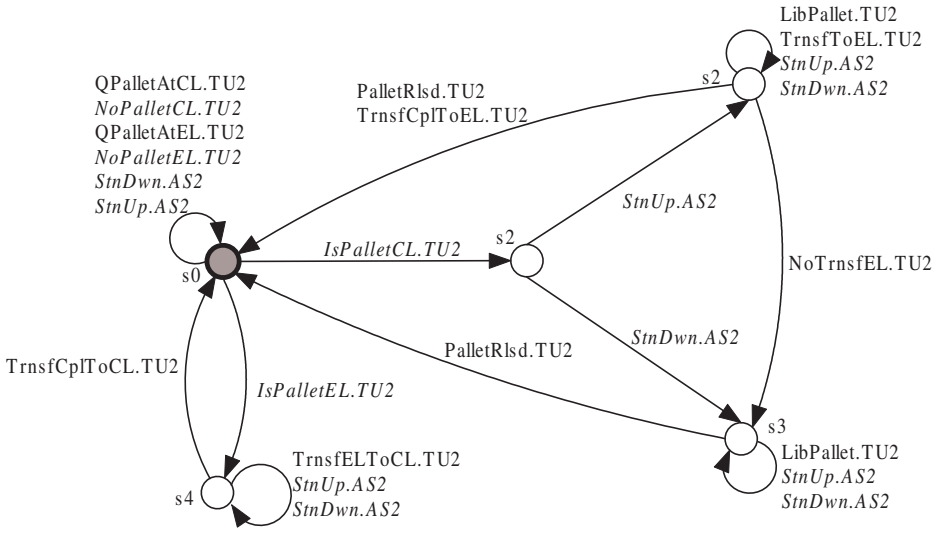


Figure 7.22: ManageTU2

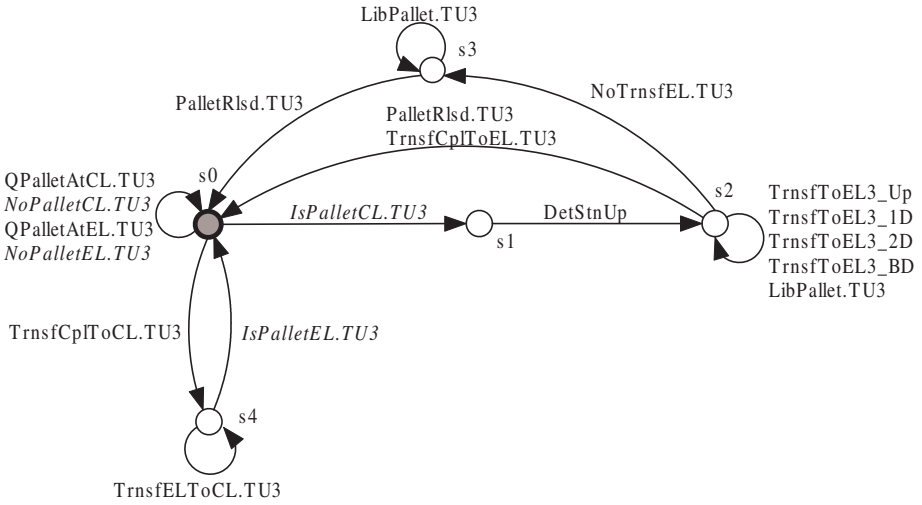


Figure 7.23: ManageTU3

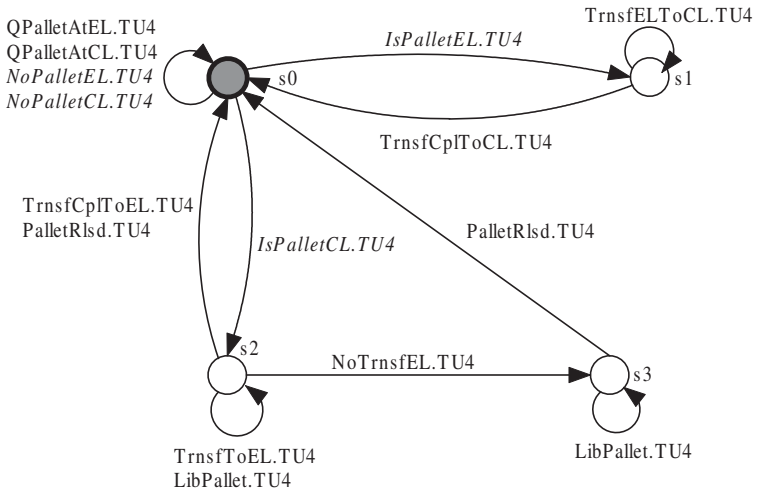


Figure 7.24: ManageTU4

Next we introduce two supervisor DES directly obtained from [21]. The first is **DetWhichStnUp**, shown in Figure 7.25. This supervisor is used to detect the breakdown status of AS1 and AS2, and then the corresponding request event that encodes this status will be chosen to be sent to low-level 6 (TU3). For example, if AS1 breaks down and AS2 does not, then request event *LibPallet.TU3* or *TrnsfToEL3_1D*

will be chosen. The second DES is **HndlComEventsAS**, shown in Figure 7.26. This supervisor is used to solve a blocking problem among the supervisors **ManageTU1, ManageTU2** and **DetWhichStnUp** due to controllable events they use in common.

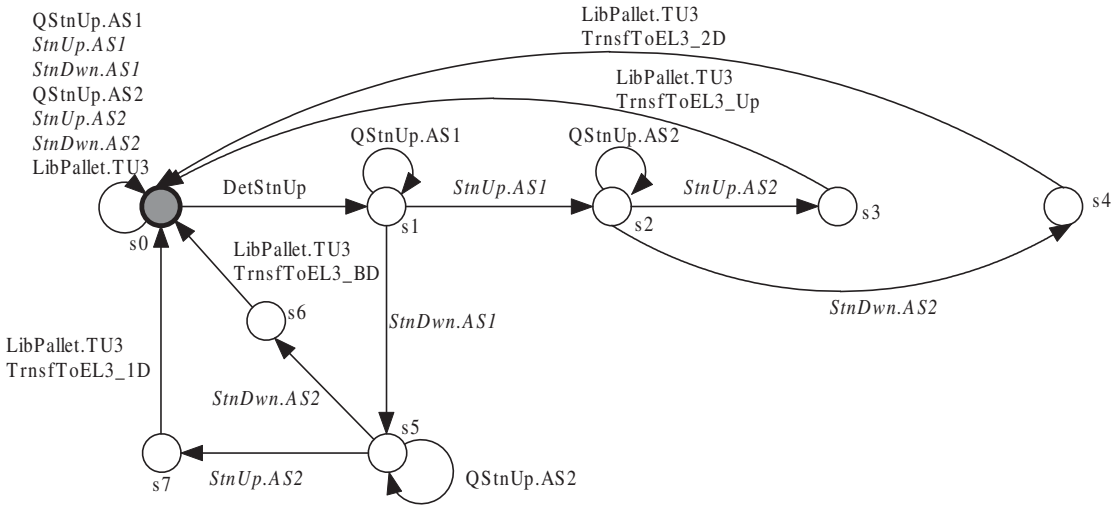


Figure 7.25: DetWhichStnUp (from [21])

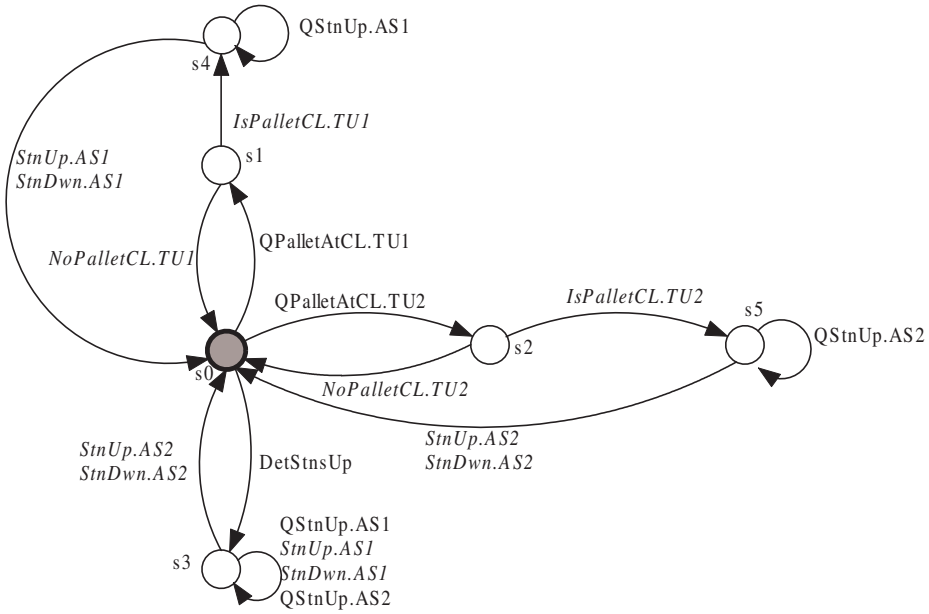


Figure 7.26: HndlComEventsAS(from [21])

We now discuss the supervisors for preventing overflow of conveyor areas EL11, EL21, EL31 and EL41. **OFProtEL11** (see Figure 7.27) is used to prevent overflow of area EL11, but it also enforces that a pallet can not be processed by AS1 until it has been transferred to EL11. **OFProtEL21** (see Figure 7.28) is identical to **OFProtEL11** up to relabeling. **OFProtEL31** (see Figure 7.29) is similar to the previous two. It prevents overflow of area EL31 and enforces that a pallet can not be processed or repaired by AS3 until it has been transferred to EL31. **OFProtEL41**(see Figure 7.30) prevents overflow of area EL41. However, it does not have the selfloop transitions of event *MvOutPallet.IO* at all states except $s0$ and $s10$, because the plant component **CapEL41** (Figure 7.20(d)) and the component supervisor **ManageIO** (Figure 7.34) will ensure that the event *MvOutPallet.IO* will not happen at state $s0$ and $s10$.

The next four component supervisors are used to prevent overflow of conveyor areas EL12, EL22, EL32 and EL42, shown in Figure 7.31. **OFProtEL12, OFProtEL22, OFProtEL32** are identical up to relabeling. **OFProtEL42** disables events *QPalletType1In.IO and QPalletType2In.IO* at state $s3$ instead of event *MvInPallet.IO*, because of the supervisor component **ManageIO**(Figure 7.34). If we use event *MvInType1Pallet.IO*(or MvInType2Pallet.IO), then the system will be blocked when the area EL42 is full (at state $s3$) and **ManageIO** is at state $s1$ (or $s2$).

The four supervisors **OFProtC1**, **OFProtC2**, **OFProtC3**, and **OFProtC4** are used to prevent overflow of the conveyor areas C1, C2, C3 and C4 in the central loop, respectively. They are identical up to relabeling and are shown in Figure 7.32.

We now describe two supervisors for the I/O station. **AltMvInTypes** is used to enforce that the type of pallets entering the system must alternates, starting with Type1 pallets. It is shown in Figure 7.33. Note that both of the states are marked, because the last pallet entering the system can be either Type1 or Type2. **ManageIO**

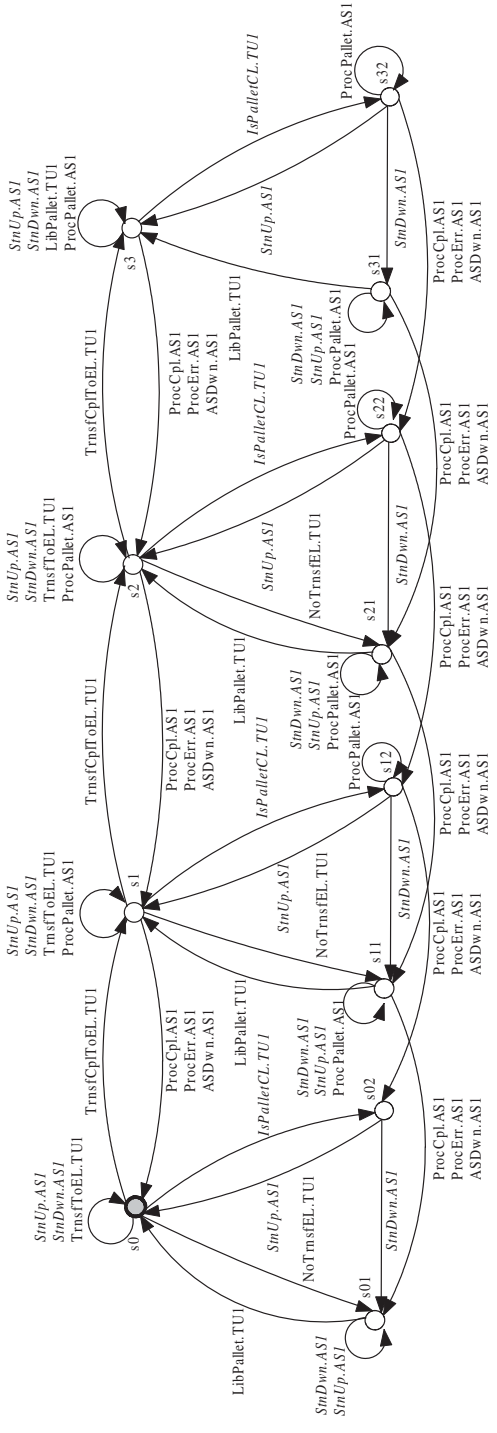


Figure 7.27: OFProtEL11

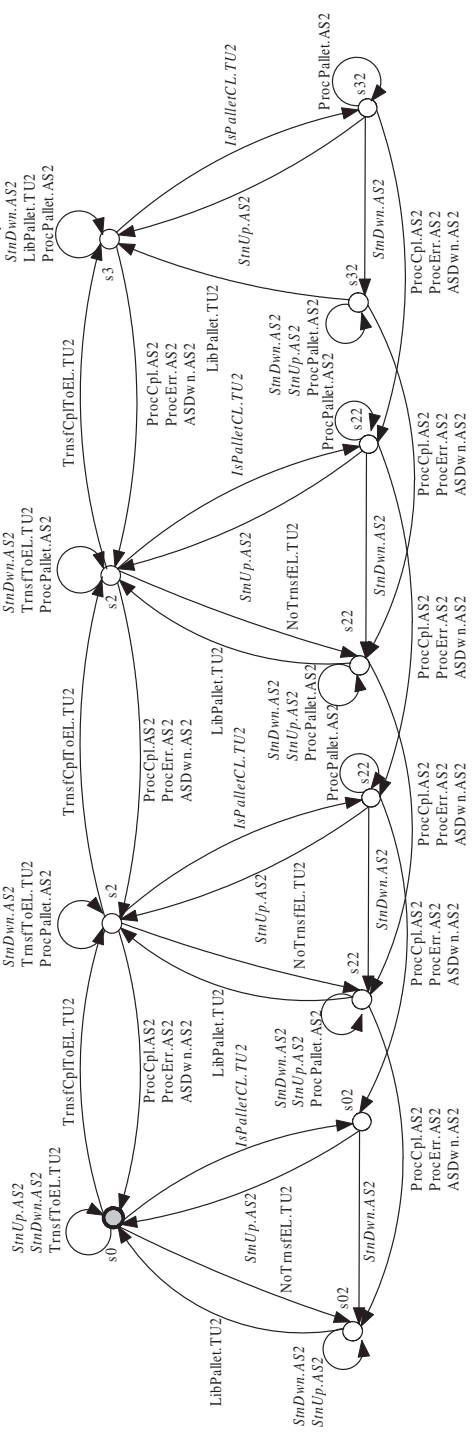


Figure 7.28: OFProtEL21

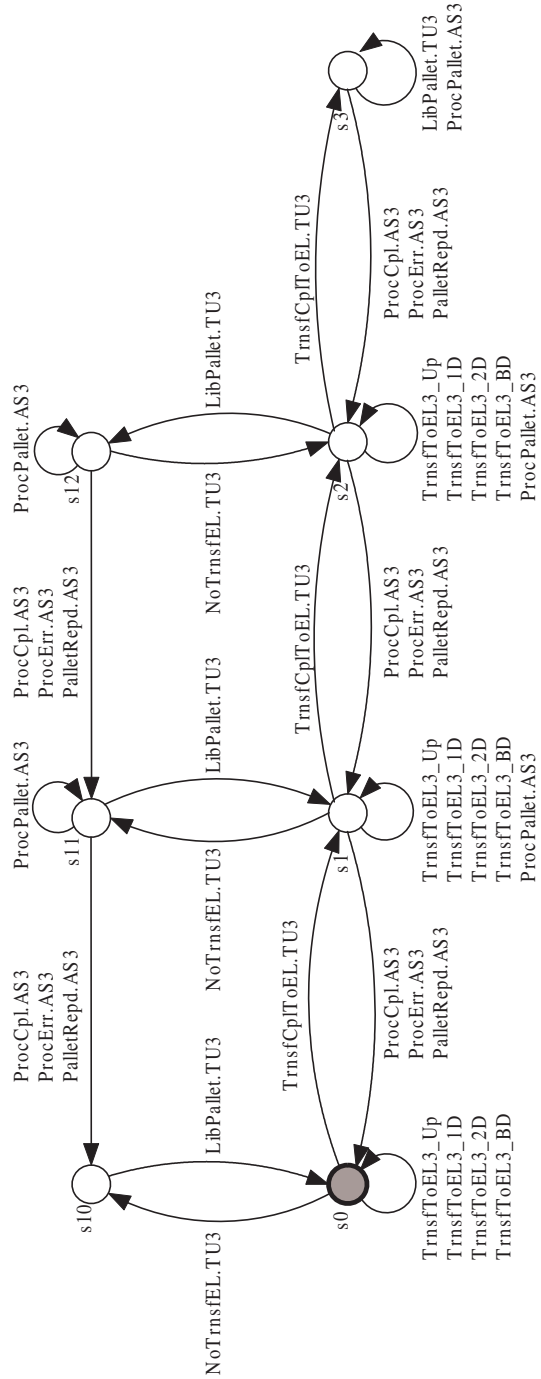


Figure 7.29: OFProtEL31

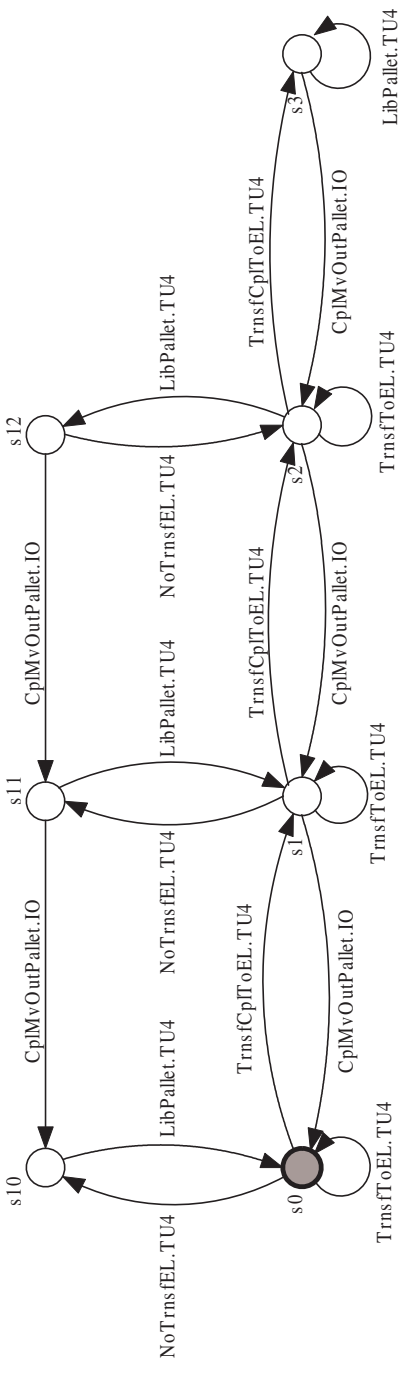
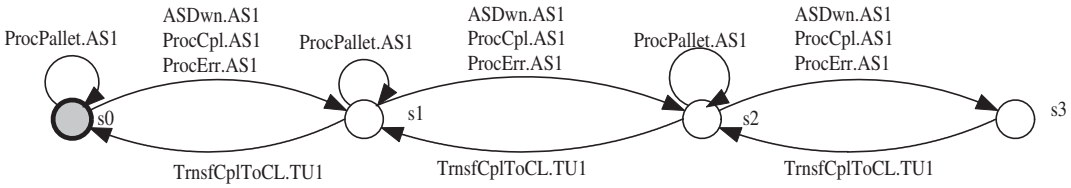
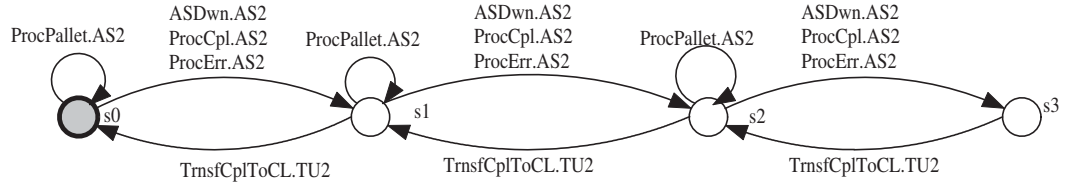


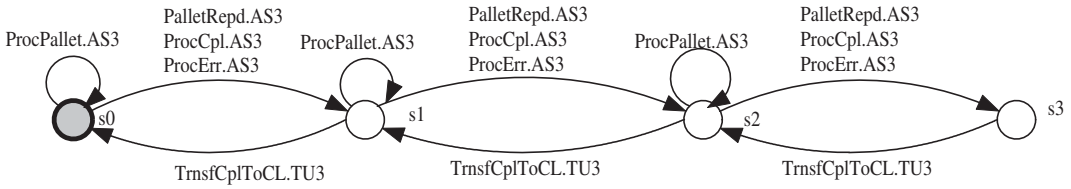
Figure 7.30: OFProtEL41



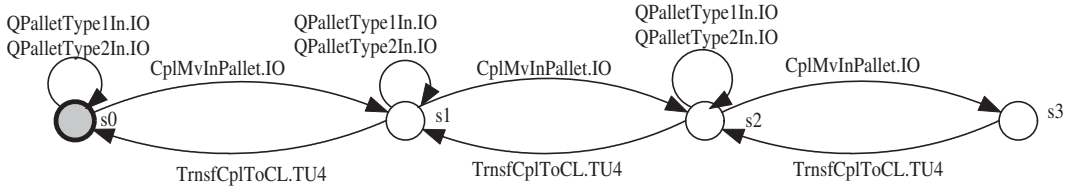
(a) OFProtEL12



(b) OFProtEL22



(c) OFProtEL32



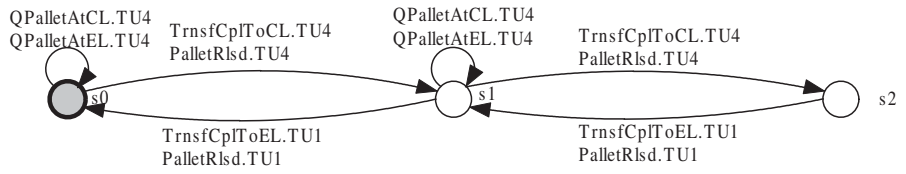
(d) OFProtEL42

Figure 7.31: OFProtEL12, OFProtEL22, OFProtEL32, OFProtEL42

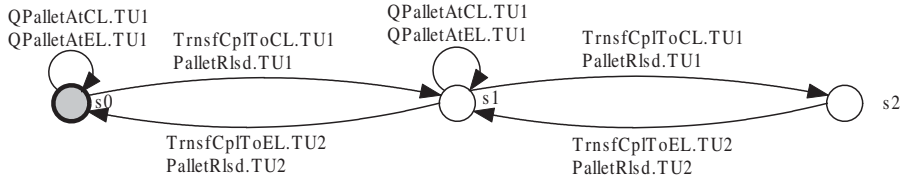
is used to control that the I/O station can only either move out a pallet or move in a pallet at a given time. **ManageIO** is shown in Figure 7.34.

7.7 Verifying Properties

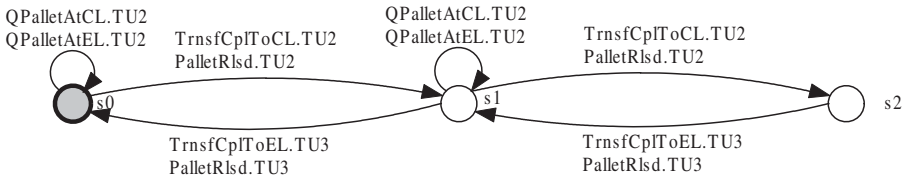
So far, we have discussed all the necessary supervisors to meet the control specifications described in Section 7.2. However, when we put everything together and



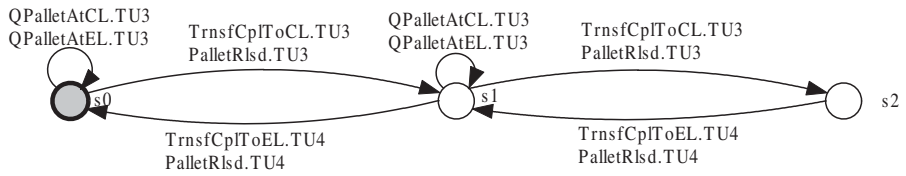
(a) OFProtC1



(b) OFProtC2



(c) OFProtC3



(d) OFProtC4

Figure 7.32: OFProtC1, OFProtC2, OFProtC3, OFProtC4

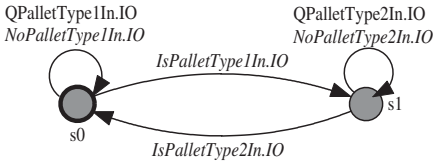


Figure 7.33: AltMvInTypes

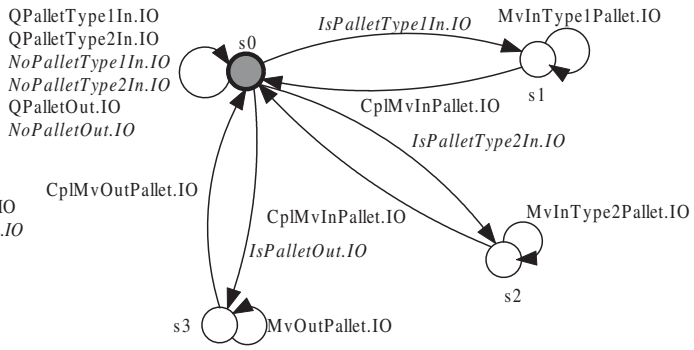


Figure 7.34: ManageIO

use our software tool to verify the high-level and all the low-levels, we find that the high-level is blocking. For example, when there are two pallets in each area of the central loop, the system will be blocked, because all the events *QPalletCL.X* and *QPalletEL.X* (where X = TU1, TU2, TU3, TU4) have been disabled.

Therefore, we add one more component supervisor **OFProtAIP** to restrict the number of pallets in the system (excluding external loop 4) at once to seven, shown in Figure 7.35.

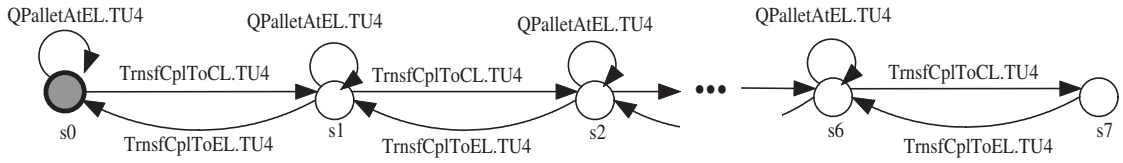


Figure 7.35: OFProtAIP

We then verified the system again using our program, and found that it satisfied all the conditions (level-wise nonblocking, level-wise controllable and interface consistent). We thus can conclude that the flat system is nonblocking by Theorem 3.1 and the flat supervisor of the system is controllable with respect to the flat plant of the system by Theorem 3.2. The system was verified on a Dell Dimension 3000 desktop PC (Pentium 4 2.8 GHz CPU, 512MB DDR 400 memory) running Fedora Core 2. It took 151 seconds to verify the high-level and less than 1 second to verify each low-level. All the computational time data in the rest of this thesis is based on this machine. The reachable state space sizes of the high-level and low-levels are listed in Table 7.1 under the column *3-2 AIP system*[4]. The total estimated worst-case state space size[5] of the AIP system is 3.61×10^{25}. The program initializes the size of the

[4]3-2 comes from the capacity (= 3) of each external loop conveyor area and the capacity (= 2) of each each central loop conveyor area

[5]The estimated size is calculated by multiplying the high-level state space size and all the low-level state space sizes together. We use this way to estimate the system state space size because the whole system is too large to be verified as a flat system. It is likely that the actual size of the system is much smaller.

BDD node table as 2,000,000 and the cache table size as 1,000,000 for this example, and it did not increase both sizes during the verification computing. As each BDD node and each entry in the cache table in the BDD package we used occupies 20 bytes of memory, and the package uses 6 such cache tables for BDD operations, we actually used $(2,000,000 + 1,000,000 \times 6) \times 20 = 160$MB memory[6]. We also treated this AIP example as a flat system and tried to verify it using our BDD-based flat system tool. The program soon used up all the memory (RAM) by increasing the node and cache table and began to use the swap space, so the speed dramatically decreased. The program ran for 24 hours and still couldn't finish the computation so we thought there is no need to let it run any more.

Level/System	3-2 AIP system			5-4 AIP system		
	Number of Reachable States	Computing Time (s)	Memory for BDD node and cache Table (MB)	Number of Reachable States	Computing Time (s)	Memory for BDD node and cache Table (MB)
High-level	$2.65 * 10^{10}$	151	160	$5.16 * 10^{13}$	1543	160
Low-level 1(AS1)	120	<1	160	120	<1	160
Low-level 2(AS2)	120	<1	160	120	<1	160
Low-level 3(AS3)	106	<1	160	106	<1	160
Low-level 4(TU1)	98	<1	160	98	<1	160
Low-level 5(TU2)	98	<1	160	98	<1	160
Low-level 6(TU3)	204	<1	160	204	<1	160
Low-level 7(TU4)	152	<1	160	152	<1	160
Low-level 8(IO)	3	<1	160	3	<1	160
Total Estimated	$3.61 * 10^{25}$			$7.04 * 10^{28}$		

Table 7.1: The AIP example data (verification)

However, our BDD-based flat system tool can verify the original HISC model of the AIP system in [23] using 160MB memory (RAM) and taking 246 seconds. The size of the total reachable state space for this system was 7.32×10^{13}. Our BDD-based

[6]Actually, our software tool always initializes the size of BDD node table as 2,000,000 and the size of cache table size as 1,000,000, because a personal computer with more than 160MB was very common at the time the author was writing this thesis. For some small subsystems(e.g. the low-level subsystems of the 3-2 AIP system here), one can decrease the initial size and may not affect the speed of the computation at all.

HISC tool can verify the original high-level in 2 seconds. The size of the reachable state space for the original high-level is 3.31×10^6.

We also extend the system by increasing the capacity of each area in the central loop to 4 and the capacity of each area in each of the external loop to 5. To do this, we need to extend plant DES **CapEL**X**1**, **CapEL**X**2**, **CapC**X, and supervisor DES **OFProtEL**X**1**, **OFProtEL**X**2**, **OFProtC**X, where $X = 1, 2, 3, 4$. We then extend the supervisor **OFProtAIP** to restrict the number of pallets in the system (excluding external loop 4) at once to be 15. The method to extend these DES to the new capacity should be obvious. We call this system as the *5-4 AIP system*. For the 5-4 AIP system, the reachable state space size for the high-level was 5.16×10^{13}, and the total estimated worst-case reachable state space size was 7.04×10^{28} (See Table 7.1). We do not present the 5-4 AIP system in this thesis in detail because some of the DES diagrams for it are too large to be put on a single page.

7.8 Synthesizing Supervisors

In the previous section, we added a supervisor **OFProtAIP** to restrict the number of pallets in the AIP system (excluding external loop 4) to ensure the high-level was nonblocking. This supervisor obviously restricts the behavior of the system. However, it is difficult to design supervisors by hand to solve the blocking problem and not to restrict the behavior in the mean time as the high-level is so large. Therefore, we can use our synthesis method to produce the control predicates by treating all the supervisor components in the system as specifications. We of course exclude **OFProtAIP** from this system.

The reachable state space size for the high-level of the 3-2 AIP system under the synthesized control predicates is 1.10×10^{12}, and it took 1415 seconds (24 minutes) to

complete the synthesis process for the high-level. This is much longer than the time to verify the 3-2 AIP system. One reason is that the high-level of this system has a bigger state space. Another reason is that the verification algorithm (Algorithm 5.1) does not have a similar `repeat` loop as the synthesis algorithm (Algorithm 4.1) has. The total estimated worst-case reachable state space size of the system under the control predicates is 1.50×10^{27}. Table 7.2 shows all the related data for the 3-2 AIP system and the 5-4 AIP system.

Level/System	3-2 AIP system			5-4 AIP system		
	Number of Reachable States	Computing Time (s)	Memory for BDD node and cache Table (MB)	Number of Reachable States	Computing Time (s)	Memory for BDD node and cache Table (MB)
High-level	$1.10 * 10^{12}$	1415	160	$1.14 * 10^{15}$	7679	160
Low-level 1(AS1)	120	<1	160	120	<1	160
Low-level 2(AS2)	120	<1	160	120	<1	160
Low-level 3(AS3)	106	<1	160	106	<1	160
Low-level 4(TU1)	98	<1	160	98	<1	160
Low-level 5(TU2)	98	<1	160	98	<1	160
Low-level 6(TU3)	204	<1	160	204	<1	160
Low-level 7(TU4)	152	<1	160	152	<1	160
Low-level 8(IO)	3	<1	160	3	<1	160
Total Estimated	$1.50 * 10^{27}$			$1.51 * 10^{30}$		

Table 7.2: The AIP example data (synthesis)

In total there are 37 controllable high-level and request events in the AIP system. For the 3-2 AIP system, the biggest BDD for the prime simplified control predicates (see description in section 6.5.1) is for event $QPalletAtEL.TU4$ ($f'_{H_{QPalletAtEL.TU4}}$), which includes 516 BDD nodes. The BDD for the triple-prime simplified control predicate for this event includes 200 BDD nodes ($f'''_{H_{QPalletAtEL.TU4}}$). For the 5-4 AIP system, the biggest BDD for the prime control predicates is also for event $QPalletAtEL.TU4$, which includes 715 BDD nodes. The BDD for the triple-prime control predicate for this event includes 266 BDD nodes.

There are 206 controllable low-level and answer events in total. Since all the

low-levels already satisfy their corresponding conditions, the BDD for the control predicates are rather small. For the 3-2 AIP system (or the 5-4 AIP system), the biggest BDD representation for the prime simplified control predicates is for event $tu3\_DwnOpNeeded.TU3$ ($f'_{L_{6tu3\_DwnOpNeeded.TU3}}$), which includes only 28 BDD nodes. The BDD representation of the triple-prime simplified control predicate for this event includes 20 BDD nodes ($f'''_{L_{6tu3\_DwnOpNeeded.TU3}}$).

7.9 Results For the AIP Example

In this chapter, we extended the AIP example from [21,23] by adding more control requirements. The extended system is much bigger and more complicated than the original system which has a state space on the order of 10^{13}. The high-level in the 5-4 AIP system alone has a state space on the order of 10^{15}, and its estimated worst-case state space is on the order of 10^{30}. In particular, the capacity restrictions of conveyor areas make the system highly coupled. If we extend the STS of the AIP example in [31] to reflect the capacity restrictions of conveyor areas, the specification DES for them would likely need to be put on level 1 of the STS. Consequently, the STS will become much flatter, and it will make the STS less structured.

Even though our predicate-based algorithm with BDD implementation can handle a larger system than an automata-based algorithm, our BDD-based software tool for a flat system can not handle the whole 3-2 AIP system. However, with the HISC method, we can easily verify the 5-4 AIP system and synthesize supervisors on a common personal computer.

Chapter 8

Multiple AIP Example

To demonstrate the ability of our low-level synthesis and verification algorithms, we construct a system as shown in Figure 8.1, where each MAIP-j ($j \in \{1, 2, 3\}$) is the high-level of the 3-2 AIP system in the previous chapter with appropriate event relabeling[1] but used as low-levels (we will discuss how to relabel events in Section 8.1). Other than the MAIPs, the system also includes a testing unit, a packaging unit and three 4×6 buffers. By 4×6 buffers, we mean that each buffer has 4 slots, and the capacity of each slot is 6 (pallets). One slot must be filled or emptied completely, which means that pallets can only be loaded into or taken out of a MAIP six at a time. We also assume that the system always puts three Type1 pallets and three Type2 Pallets into the input buffer each time. This example is based upon the parallel manufacturing example in [21].

As in the previous chapter, we first give a system with supervisors so that we can examine our verification algorithms, then we relax one of the restrictions to perform synthesis.

[1]We do not use the whole AIP, because our software can not handle one subsystem as large as the whole 3-2 AIP.

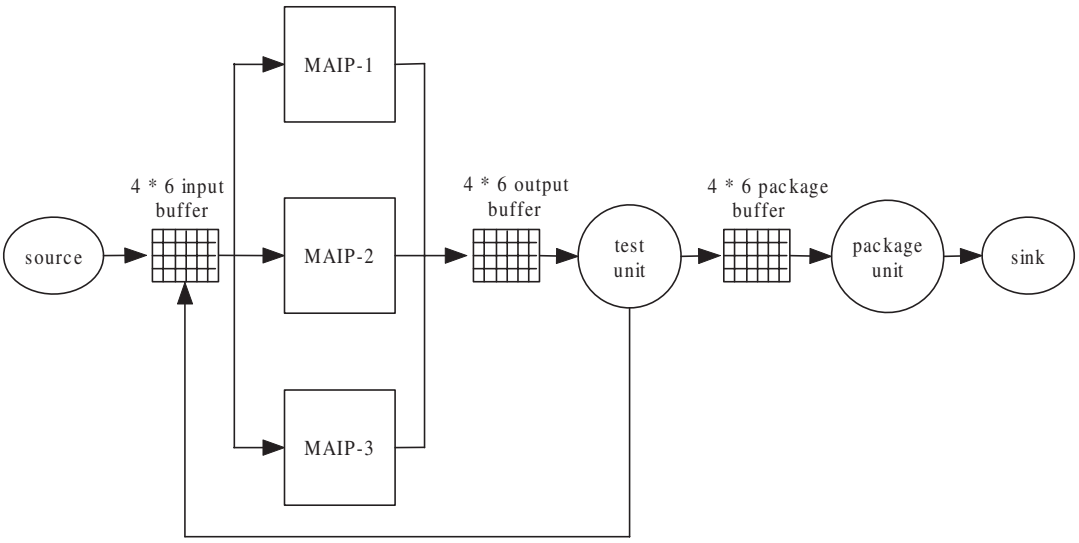


Figure 8.1: Block diagram of multiple AIPs

8.1 System Model

We model this system as a bi-level HISC system with three low-levels MAIP-1, MAIP-2, and MAIP-3. As all the low-levels are identical up to relabeling, their interfaces are as well. The interface for MAIP-$j(j \in \{1, 2, 3\})$ is shown in Figure 8.2. Event $startpallets - j$ means that MAIP takes six pallets from an input buffer slot and starts processing them. Event $finishpallets - j$ means that six processed pallets are put into an output buffer slot.

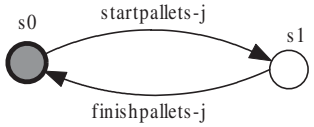
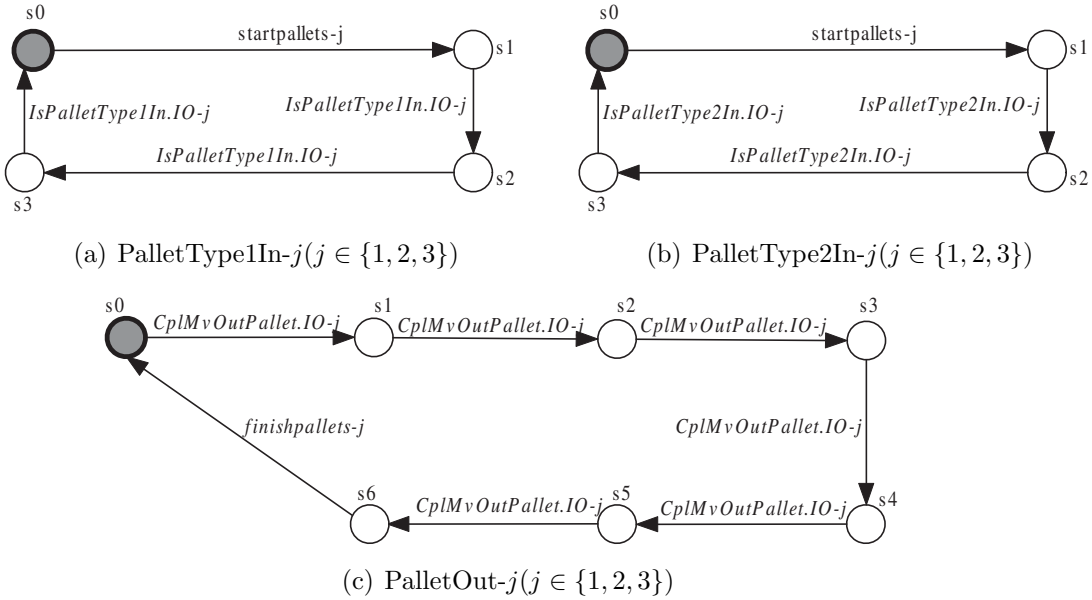


Figure 8.2: IntfAIP-j(j = 1, 2, 3)

As we said before, the low-level MAIP-$j(j \in \{1, 2, 3\})$ here is the high-level of the 3-2 AIP system in the previous chapter with event relabeling. All the events(high-level, request and answer events) in the original high-level of the 3-2 AIP system

are treated as low-level events in MAIP-j. We also treat all the answer events in the original system as uncontrollable low-level events in MAIP-j to capture the fact that we can not just disable them. The other events keep their original controllable or uncontrollable classification. Each event name in MAIP-j is suffixed with -j to capture the fact that events in each MAIP actually are different events, and so that the event partition requirements are satisfied.

The high-level plant components and all the interfaces in the original 3-2 AIP system are treated as low-level plant components in MAIP-j, and the high-level supervisors in the original system are treated as low-level supervisors except that we do not include **OFProtAIP** in MAIP-j. We also add three plant components **PalletType1In-j, PalletType2In-j** and **PalletOut-j** in low-level MAIP-j, as shown in Figure 8.3. These plant components state that MAIP-j processes six pallets at a time.



(a) PalletType1In-$j(j \in \{1,2,3\})$ (b) PalletType2In-$j(j \in \{1,2,3\})$

(c) PalletOut-$j(j \in \{1,2,3\})$

Figure 8.3: New low-level plants in 3-2 MAIP system(verfication)

In total, we have 30 plant components and 20 supervisor components in the low-

level MAIP-j. The components in MAIP-j are shown in Figure 8.4. The plant $\mathbf{G}^p_{L_j}$ and supervisor $\mathbf{S}_{L_j}$ of MAIP-j are defined to be the synchronous product of the indicated automata.

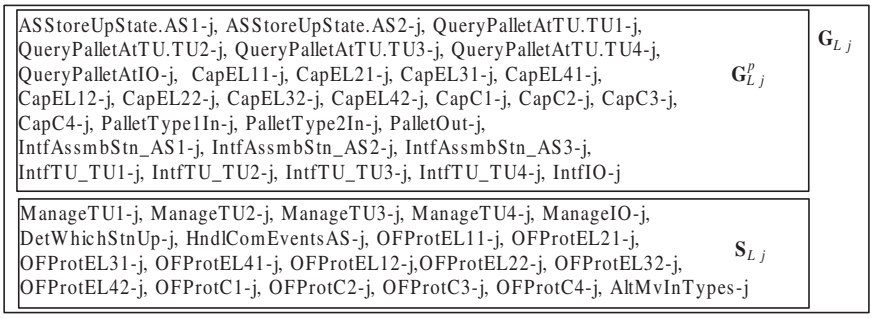
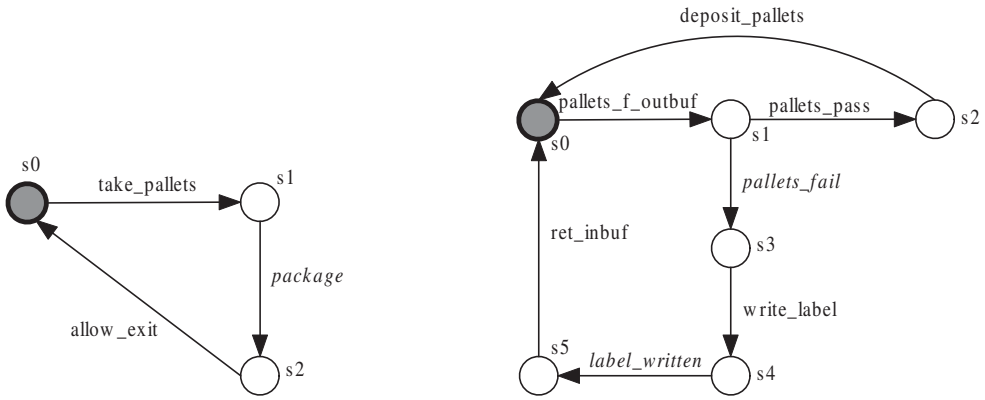


Figure 8.4: Components in low-level subsystem MAIP-j, $(j = 1, 2, 3)$

The high-level plant components are shown in Figure 8.5. Plant component **Source** contains only one event, *new_pallets*, which means that six new pallets are put into one input buffer slot. The test unit (**TestUnit**) takes six pallets from the output buffer and then tests if the assembly of those pallets is correct. If any of the six pallets can not pass the test, information will be written on the labels of the failed pallets and the six pallets altogether must be sent back to the input buffer. For those pallets in the group that have passed the test, they will just go through the central loop of the MAIP-j undergoing no assembly operations as their labels were not altered and are allowed to exist via the I/O station. If all six pallets pass the test, they are placed in the package buffer. Plant component **PackSys** is straightforward. Event *take_pallets* means that the package unit takes out all of the six successfully processed pallets in one package buffer slot. Plant component **Sink** also contains only one event, *allow_exits*, which means that the system allows the six packaged pallets to exit from the system.

The high-level supervisors are shown in Figure 8.6. **InBuf, OutBuf** and **PackBuf** are designed to prevent their corresponding buffers from overflow and underflow.



(a) PackSys (b) TestUnit

(c) Source (d) Sink

Figure 8.5: High-level plant components

NoBlock is to make sure that the system is nonblocking.

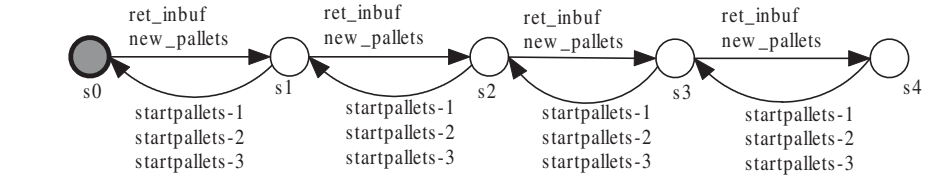
The event partition is defined as follows, where $j = 1, 2, 3$, Σ''_H is the high-level event set, $\Sigma''_{R_1}, \ldots, \Sigma''_{R_8}$ are the request event sets, and $\Sigma''_{A_1}, \ldots, \Sigma''_{A_8}$ are the answer event sets for the 3-2 AIP example in the previous chapter. σ-j means that the name of event σ is suffixed with "-j".

$$\Sigma_H := \{take\_pallets, package, allow\_exit, new\_pallets, pallets\_f\_outbuf,$$
$$pallets\_pass, deposit\_pallets, pallets\_fail, write\_label, label\_written,$$
$$ret\_inbuf\}$$

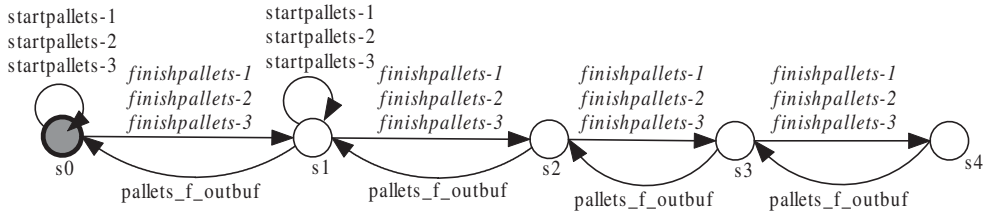
$$\Sigma_{R_j} := \{startpallets\text{-}j\}$$

$$\Sigma_{A_j} := \{finishpallets\text{-}j\}$$

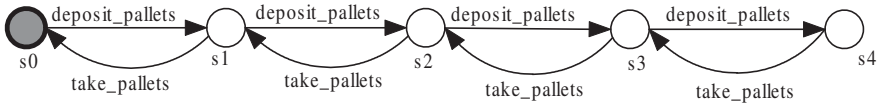
$$\Sigma_{L_j} := \{\sigma\text{-}j | \sigma \in \{\Sigma''_H \cup \Sigma''_{R_1} \cup \cdots \cup \Sigma''_{R_8} \cup \Sigma''_{A_1} \cup \cdots \cup \Sigma''_{A_8}\}\}$$



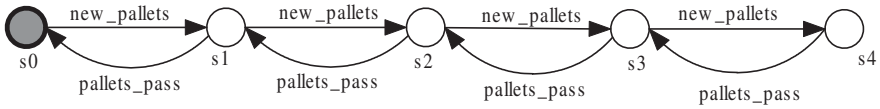
(a) InBuf



(b) OutBuf



(c) PackBuf



(d) NoBlock

Figure 8.6: High-level supervisor components

8.2 Verifying Properties

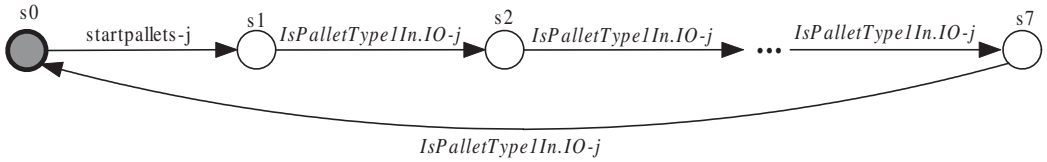
We now use our software tool to verify this system, and find that the system satisfies all the HISC local conditions (level-wise nonblocking, level-wise controllable and interface consistent). We thus can conclude that the flat system is nonblocking by Theorem 3.1 and the flat supervisor of the system is controllable with respect to the flat plant of the system by Theorem 3.2. The resulting computational data is shown in Table 8.1 under the column *3-2 MAIP system.* We also treat this system as a flat system and use our BDD-based software tool for flat systems to verify it. The tool soon used up all the available memory (RAM) and there was still no result after

24 hours.

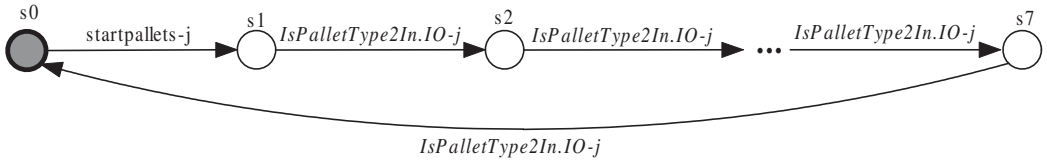
Level/System	3-2 MAIP system			5-4 MAIP system		
	Number of Reachable States	Computing Time (s)	Memory for BDD node and cache Table (MB)	Number of Reachable States	Computing Time (s)	Memory for BDD node and cache Table (MB)
High-level	4260	<1	160	4260	<1	160
Low-level MAIP-j j = 1,2, 3	$6.88 * 10^8$	66	160	$3.04 * 10^{12}$	1397	160
Total Estimated	$1.39 * 10^{30}$			$1.20 * 10^{41}$		

Table 8.1: The multiple AIP example data(verification)

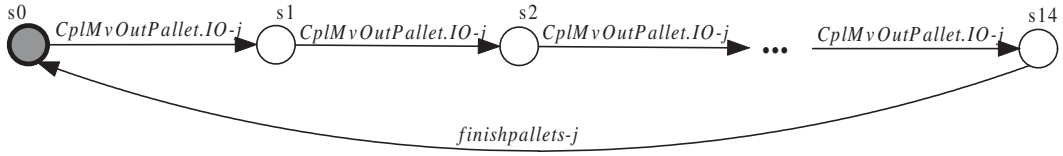
We also built a system by reusing the high-level in the 5-4 AIP system in the previous chapter in a similar fashion. We refer to this system as *5-4 MAIP system*. For the 5-4 MAIP system, we increase the size of each buffer to 4×14 (pallets), and assume that the system always put seven Type1 pallets and seven Type2 Pallets into a 14-pallet slot of the input buffer each time. The high-level model and interfaces of the 5-4 MAIP system are the same as the high-level model and interfaces of the 3-2 MAIP system. The low-levels can be built by using the same method of building low-levels in the 3-2 MAIP system. The plant components **PalletType1In-**j**,** **PalletType2In-**j and **PalletOut-**j in low-level MAIP-j of the 5-4 MAIP system are shown in Figure 8.7. The event partition is same as that of the 3-2 MAIP system. This 5-4 MAIP system also satisfies all the HISC local conditions (level-wise nonblocking, level-wise controllable and interface consistent). We thus can conclude that the flat system is nonblocking and the flat supervisor of the system is controllable with respect to the flat plant of the system. The resulting computational data is shown in Table 8.1 under the column 5-4 MAIP system.



(a) PalletType1In-$j(j \in \{1,2,3\})$



(b) PalletType2In-$j(j \in \{1,2,3\})$



(c) PalletOut-$j(j \in \{1,2,3\})$

Figure 8.7: New low-level plants for the 5-4 MAIP system(verification)

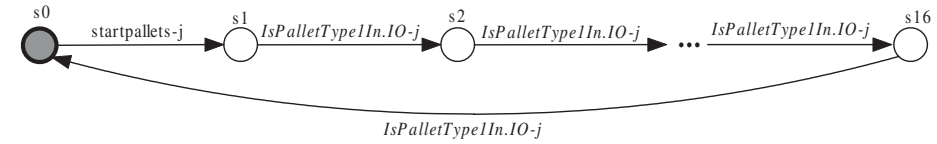
8.3 Synthesizing Supervisors

In the above 3-2 MAIP system, the low-level MAIP-j ($j \in \{1,2,3\}$) processes pallets six at a time. In the previous chapter, the supervisor component **OFProtAIP** was used to make sure that the number of the pallets in the 3-2 AIP system (excluding external loop 4) is at most seven at any given time. Here we require that the above 3-2 MAIP system processes six pallets at a time, so it respects the requirement of the component **OFProtAIP** in the 3-2 AIP system. We set the size of each slot as six because we want the number of Type1 pallets and the number of Type2 pallets in each slot to be equal.

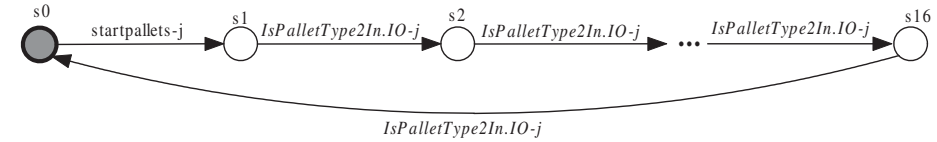
Now we increase the capacity of each slot in each buffer in the 3-2 MAIP system[2]

[2]Recall that in the 3-2 AIP system in the previous chapter, the capacity of each area in the central loop is 2 and the capacity of each area in an external loop is 3, so in total the capacity of the conveyors is $2 \times 4 + 3 \times 2 \times 4 = 32$.

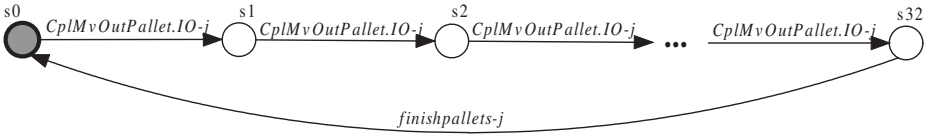
to 32. Consequently, the plant component **PalletType1In-**j should have 16 continuous $IsPalletType1In.IO\text{-}j$ transitions; the plant component **PalletType2In-**j should have 16 continuous $IsPalletType2In.IO\text{-}j$ transitions, and the plant component **PalletOut-**j should have 32 continuous $CplMvOutPallet.IO\text{-}j$ transitions. They are shown in Figure 8.8.



(a) PalletType1In-$j(j \in \{1,2,3\})$



(b) PalletType2In-$j(j \in \{1,2,3\})$



(c) PalletOut-$j(j \in \{1,2,3\})$

Figure 8.8: New low-level plants for the 3-2 MAIP system(synthesis)

When we use our software tool to verify this system, we find that it each low-level MAIP-j ($j \in \{1,2,3\}$) blocks. We then synthesize a low-level proper supervisor and its corresponding control predicates for each low-level MAIP-j. The resulting computational data for this system is shown in Table 8.2 under the column 3-2 MAIP system.

Similarly, we also increase the capacity of each slot in each buffer to 56 for the 5-4 MAIP system. Therefore, the plant component **PalletType1In-**j should have 28 continuous $IsPalletType1In.IO\text{-}j$ transitions; the plant component **PalletType2In-**j should have 28 continuous $IsPalletType2In.IO\text{-}j$ transitions, and the plant com-

ponent **PalletOut-**j should have 56 continuous $CplMvOutPallet.IO$-j transitions. One can easily draw the diagrams for them from Figure 8.8. The low-levels of this 5-4 MAIP system also block. Therefore, we can synthesize a low-level proper supervisor and its corresponding control predicates for each low-level MAIP-j. The resulting computational data for this system is shown in Table 8.2 under the column 5-4 MAIP system. Note that the memory for BDD node and cache table for synthesizing the low-level MAIP-j in this system has increased to 220MB, and all the other systems we examined in this thesis never increased the predefined BDD node and cache table size. Note also that the computing time is very long (26.7 hours). We decided to let the synthesizing process run this long time because it did not use up all the available memory (RAM) in our computer (512MB).

Level/System	3-2 MAIP system			5-4 MAIP system		
	Number of Reachable States	Computing Time (s)	Memory for BDD node and cache Table (MB)	Number of Reachable States	Computing Time (s)	Memory for BDD node and cache Table (MB)
High-level	4260	<1	160	4260	<1	160
Low-level MAIP-j j = 1,2, 3	$9.48 * 10^{12}$	16453 (4.57 hours)	160	$1.61 * 10^{16}$	96126 (26.7 hours)	220
Total Estimated	$3.63 * 10^{42}$			$1.78 * 10^{52}$		

Table 8.2: The multiple AIP example data(synthesis)

Chapter 9

Conclusions and Future Work

9.1 Conclusions

In this thesis, by applying symbolic method to an HISC system, we have developed algorithms for synthesis and verification. We also gave large examples (e.g. the 5-4 AIP system has worst-case estimated state space 1.51×10^{30}) to demonstrate the potential abilities of the algorithms. We now discuss some conclusions and contributions of this thesis.

For an HISC system, a per level supervisor synthesis method based on predicate operations was proposed. This was not provided by Leduc *et al.* in [21–25, 27] before, as they mainly focused on the verification method.

With the help of symbolic computation, we can handle much larger individual subsystems for verification and synthesis (e.g. the synthesized high-level supervisor has 1.14×10^{15} states in the 5-4 AIP system and can be computed with less than 160MB of RAM) than before. With explicit state and transition listings, a software typically can only handle subsystems as large as 10^7 states with 1GB of RAM.

For the synthesized supervisors, a controller implementation scheme for an HISC

system was provided based on a group of small local control predicates which can be represented as BDDs (The largest prime simplified control predicate for the 5-4 AIP system has 715 BDD nodes). As each control predicate is fairly small, the speed of this BDD-based controller can be guaranteed.

Note here that although the system containing the synthesized supervisors has locally maximally permissible behavior, the behavior of the flat system containing these supervisors in general is not globally maximally permissive. This is because of the HISC conditions, and this is the price we pay for synthesizing supervisors that respect the interface conditions. However, it is worthwhile to do so because we could design a very large system with great scalability.

In this thesis, we have talked about both synthesis and verification methods for an HISC system. With the synthesis method, a system designer can mainly focus on modeling system requirements and then compute the controller automatically. However, the synthesized supervisor is often too large to be understood by simple inspection by designers, especially for large systems (e.g. the AIP examples). With the hand-designed supervisor, nevertheless, one is much more confident about the behavior of a system. Furthermore, to verify a system is usually faster than to synthesize a supervisor for the system, as there is no need to compute the greatest fixpoint by repeatedly applying functions. However, sometimes it is very difficult to design all the supervisors by hand directly and not to restrict the system behavior too much in the mean time. For example, to make the 3-2 AIP example (verification version) nonblocking, we had to add a supervisor (**OFProtAIP**, See Figure 7.35) to restrict the system behavior. In the AIP examples we designed, we mainly design supervisors to reflect the control requirements, but to solve the blocking problem, we prefer to use the synthesis method. By designing system this way, we are confident that the controlled system behavior is what we expect but not to restrict the system

too much.

9.2 Future Work

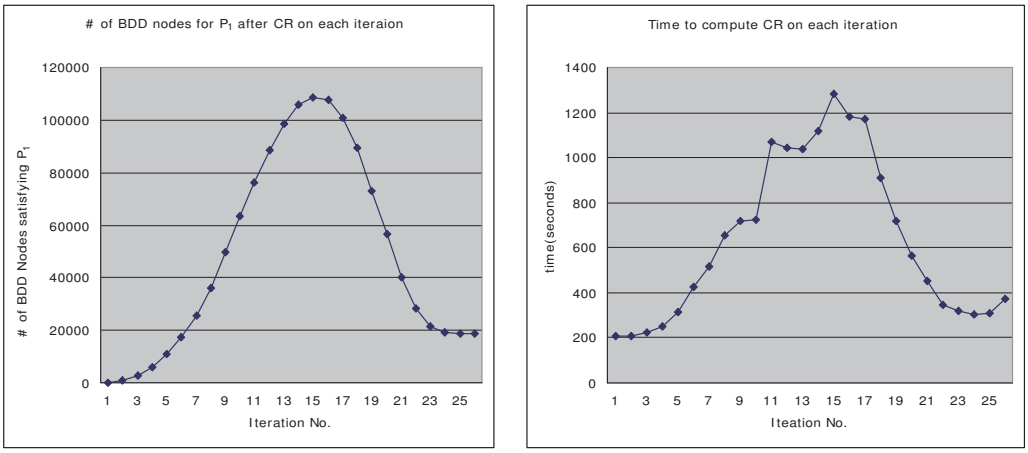
We now discuss some future works of using symbolic method for HISC systems.

9.2.1 Intermediate BDD Size Problem

In Algorithm 4.1 and Algorithm 4.3, there is a `repeat` loop. We only proved that the algorithm can terminate after finite number of iterations which is no more than the size of the state space in the high-level or low-levels. However, for practical systems, the number of iterations is likely to be very small. For example, the 3-2 AIP example in Chapter 7 only takes 3 iterations.

However, once the number of iterations gets bigger, we will have an intermediate BDD size problem. For example, in a system we worked on, the number of iterations was 26. For this system, the size of the BDD for predicate P_1 during iterations 11-18 was much bigger than its size in other iterations, although the number of states satisfying P_1 was becoming smaller. Figure 9.1 shows how the number of BDD nodes changed, and the time to complete the CR operation at each iteration in the synthesis process of that system. We can see that the time to complete each middle iteration is much longer than the time to complete each iteration at the beginning or end.

Usually the specification DES is composed of many specification components. For future work, we may synthesize supervisors first using only part of the specification components, then use these intermediate supervisors as specifications to synthesize a final supervisor. By doing things this way, we may be able to control the size of intermediate BDD. This idea was inspired by Exercise 3.7.13 in [47].



(a) Number of BDD nodes (b) Time to complete CR operation

Figure 9.1: Number of BDD nodes and time for CR operation at each iteration

9.2.2 Complexity Analysis of the Algorithms

Although we use BDD to represent a subsystem of the HISC system and use BDD operations to do the computations, the worst case of the computational complexity is still based on the state space size of the subsystem, which is exponential on the number of components. It is worthwhile to do a more detailed complexity analysis, although the actual computing time of a BDD implementation of the algorithms is highly dependent on the system being examined.

9.2.3 Symbolic Computation for Timed HISC System

Currently, the HISC method is only for untimed discrete event systems. As timed systems usually have much bigger state-spaces, we may apply the idea of the HISC to design a method for timed system and then develop the algorithms suitable for symbolic synthesis and verification. By combining these two methods, we may be able to handle more complicated timed systems.

Bibliography

[1] H. R. Andersen. *An Introduction to Binary Decision Diagrams*. Lecture Notes, IT University of Copenhagen. [ONLINE] Available: http://www.itu.dk/people/hra.

[2] D. S. Arnon. A bibliography of quantifier elimination for real closed fields. *Journal of Symbolic Computation*, 5(1-2):267–274, February/April 1988.

[3] G. Barrett and S. Lafortune. Decentralized supervisory control with communicating controllers. *IEEE Trans. Automatic Control*, 45(9):1620–1638, 2000.

[4] B. Bollig and I. Wegener. Improving the Variable Ordering of OBDDs is NP-Complete. *IEEE Trans. Computers*, 45(9):993–1002, Sept. 1996.

[5] B. A. Brandin and F. Charbonnier. The supervisory control of the automated manufacturing system of the AIP. In *Proc. Rensselaer's 1994 Fourth International Conference on Computer Integrated Manufacturing and Automation Technology*, pages 319–324, Troy, Oct 1994.

[6] R. E. Bryant. Graph-Based algorithm for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, Aug. 1986.

[7] F. Charbonnier. Commande par supervision des systèmes à événements discrets: application à un site expérimental l'Altelier Inter-établissement de Productique.

Technical report, Laboratoire d'Automatique de Grenoble, Grenoble, France, 1994.

[8] H. Chen and H.-M. Hanisch. Model aggregation for hierarchical control synthesis of discrete event systems. In *Proc. 39th Conf. Decision Contr.*, pages 418–423, Sydney, Australia, December 2000.

[9] S.-L. Chen. Existence and design of supervisors for vector discrete event systems. Master's thesis, Department of Electrical Engineering, University of Toronto, Toronto, Ont, 1992.

[10] S.-L. Chen. *Control of Discrete-Event Systems of Vector and Mixed Structural Type*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ont, 1996.

[11] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Jan. 2000.

[12] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373, J. Sifakis, editor, Berlin, Germany, June 1989. Springer-Verlag.

[13] M. Courvoisier, M.Combacau, and A. de Bonneval. Control and monitoring of large discrete event systems: a generic approach. In *Proc. of ISIE 93*, pages 571–576, Budapest, 1993.

[14] Pengcheng Dai. Synthesis method for hierarchical interface-based supervisory control. Master's thesis (to appear), Department of Computing and Software, McMaster University, Hamilton, Ont.

[15] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[16] Johan Gunnarsson. *Symbolic Methods and Tools for Discrete Dynamic Systems.* Ph.D. Thesis, Linkoping Studies in Science and Technology, 1997.

[17] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, Jun. 1987.

[18] P. Hubbard and P.E. Caines. Dynamic consistency in hierarchical supervisory control. *IEEE Transactions On Automatic Control*, 47(1):37–52, Jan. 2002.

[19] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Nov. 1999.

[20] R. J. Leduc. PLC implementation of a DES supervisor for a manufacturing testbed: An implementation perspective. Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ont., 1996.

[21] R. J. Leduc. *Hierarchical Interface-based Supervisory Control*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ont., 2002. [ONLINE] Available: http://www.cas.mcmaster.ca/~leduc.

[22] R. J. Leduc, B.A. Brandin, M. Lawford, and W. Murray Wonham. Hierarchical interface-based supervisory control, part I: Serial case. *IEEE Trans. Automatic Control*, 50(9):1322–1335, Sept. 2005. See also SQRL Report No. 12, Dept.

of Computing and Software, McMaster University, Hamilton, ON. [ONLINE] http://www.cas.mcmaster.ca/sqrl/sqrl_reports.html.

[23] R. J. Leduc and M. Lawford. Hierarchical interface-based supervisory control of flexible manufacturing system. *Submitted to the IEEE Transactions on Control Systems Technology Journal*, Jan. 2004.

[24] R. J. Leduc, M. Lawford, and P. Dai. Hierarchical interface-based supervisory control of a flexible manufacturing system. Technical Report No. 32, Software Quality Research Laboratory, Dept. of Computing and Software, McMaster University, Hamilton, ON, Canada, Dec. 2005.

[25] R. J. Leduc, M. Lawford, and W. Murray Wonham. Hierarchical interface-based supervisory control,part II: Parallel case. *IEEE Trans. Automatic Control*, 50(9):1336–1348, Sept. 2005. See also SQRL Report No. 13, Dept. of Computing and Software, McMaster University, Hamilton, ON. [ONLINE] http://www.cas.mcmaster.ca/sqrl/sqrl_reports.html.

[26] R.J. Leduc. Hierarchical interface-based supervisory control: Command-pair interfaces (see extended version). In *Proc. of the Third International DCDIS Conference on Engineering Applications and Computational Algorithms*, pages 323–329, Guelph, Ontario, Canada, May 15–18, 2003. [ONLINE] Available: http://www.cas.mcmaster.ca/˜leduc.

[27] R.J. Leduc, W.M. Wonham, and M. Lawford. Hierarchical interface-based supervisory control: Bi-level systems. Technical report No. 0103, Systems Control Group, University of Toronto, Toronto, ON, Canada, Nov. 2001.

[28] Y. Li and W. M. Wonham. Control of vector discrete-event systems: I - the base model. *IEEE Transactions on Automatic Control*, 38(8):1214–1227, Aug. 1993.

[29] Y. Li and W. M. Wonham. Control of vector discrete-event systems: II- controller synthesis. *IEEE Transactions on Automatic Control*, 39(3):512–531, Mar. 1994.

[30] F. Lin and W. M. Wonham. Decentralized control and coordination of discrete-event systems with partial observations. *IEEE Transacitons on Automatic Control*, 35(122):1330–1337, 1990.

[31] Chuan Ma. *Nonblocking Supervisory Control of State Tree Structures*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ont., 2004.

[32] John O. Moody and Panos J. Antsaklis. *Supervisory Control of Discrete Event Systems using Petri Nets*. Kluwer Academic Publishers, 1998.

[33] Ken Qian Pu. Modeling and control of discrete-event systems with hierarchical abstraction. Master's thesis, Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, Ont, 2000.

[34] M. Queiroz and J. Cury. Modular supervisory control of large scale discrete event systems. In *Proceedings of WODES 2000*, pages 103–110, Ghent, Belgium, Aug 2000.

[35] P. J. Ramadge and W. M. Wonham. Modular feedback logic for discrete event systems. *SIAM Journal on Control Optimization*, 25(5):1202–1218, Sep. 1987.

[36] P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete-event processes. *SIAM Journal on Control Optimization*, 25(1):206–230, Jan. 1987.

[37] J.H. Richter and F. Wenck. Hierachical interface-based supervisory control of a bottling plant. In *Proceedings of the 16th IFAC World Congress*, Prague, Czech Republic, July 2005.

[38] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *IEEE/ACM International Conference on CAD*, pages 42–47, Santa Clara, California, Nov. 1993. ACM/IEEE, IEEE Computer Society Press.

[39] Karen Rudie and W.M. Wonham. Think globally, act locally: Decentralized control. *IEEE Transactions on Automatic Control*, 37(11):1692–1708, 1992.

[40] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[41] César R. C. Torrico and José E. R. Cury. Hierarchical supervisory control of discrete-event systems based on state aggregation. In *Proc. of Fifteenth Triennial World Congress of the International Federation of Automatic Control*, Barcelona, Spain, July 2002.

[42] M. Uzam. *Petri-net-based Supervisory Control of Discrete Event Systems and their ladder logic diagram implementations*. PhD thesis, University of Salford, Salford, UK, 1998.

[43] M. Uzam. An optimal deadlock prevention policy for flexible manufacturing systems using Petri net models with resources and the theory of regions. *Int. J. Adv. Manuf. Technol.*, 19:192–208, 2002.

[44] Bing Wang. Top-down design for RW supervisory control theory. Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ont., 1995.

[45] K. C. Wong and J. H. van Schuppen. Decentralized supervisory control of discrete event systems with communication. In *Proc. of WODES 1996*, pages 284–289, Edinburgh, UK, Aug 1996.

[46] K.C. Wong and W. M. Wonham. Hierarchical control of discrete-event systems. *Discrete-Event Dynamic Systems: Theory and Applications*, 6(3):241–273, July 1996.

[47] W. M. Wonham. *Supervisory Control of Discrete-Event Systems*. Dept. of Electrical and Computer Engineering, University of Toronto, Jul. 2005. Monograph and TCT software can be downloaded at http://www.control.toronto.edu/DES/.

[48] W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal on Control Optimization*, 25(3):637–659, May. 1987.

[49] W. M. Wonham and P.J. Ramadge. Modular supervisory control of discrete event systems. *Mathematics of Control, Signal and Systems*, 1(1):13–30, 1988.

[50] T. Yoo and S. Lafortune. A general architecture for decentralized supervisory control of discrete-event systems. In *Proc. of WODES 2000*, pages 111–118, Ghent, Belgium, Aug 2000.

[51] Zhonghua Zhang. Smart TCT (STCT): An efficient algorithm for supervisory control design. Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ont., Apr. 2001.

[52] H. Zhong and W. M. Wonham. On the consistency of hiearchical supervision in discrete-event systems. *IEEE Transactions on Automatic Control*, 35(10):1125–1134, 1990.

Appendix A

HISC Software Program

In this appendix, we first briefly introduce the running and development environment, and then we list all the source code of the program.

A.1 Introduction of HISC Software Program

The HISC software program is implemented by using C++ and STL (Standard Template Library), and it is compiled by using GNU g++ 3.3.3. The software was developed by using Eclipse 3.0 with CDT (C/C++ Development Tools) plug-in on Fedora Core 2. The BDD package we used is BuDDy 2.4 developed by Jørn Lind-Nielsen.

The HISC software program is implemented as a function library. For the interface of the library, please see BddHisc.h in the source code list. All the function interfaces are compatible with C or C++.

A very simple text user interface to demonstrate how to use the functions provided by the library is listed in main.cpp.

Table A.1 and Table A.2 are the lists of all the source code files.

No.	File Name	Description	Page
1	BddHisc.h	The interface of all the functions in the library	253
2	BddHisc.cpp	Implementation of the functions in BddHisc.h	255
3	Project.h	Header file of Project.cpp	260
4	Project.cpp	Process Project files (.prj)	261
5	Sub.h	Header file for Sub*.cpp	267
6	Sub.cpp	Process high-level or low-level subsystem file(.sub)	269
7	Sub1.cpp	Reorder DES BDD variables in high-level or low-level	273
8	Sub2.cpp	Save synthesized automata-based supervisor, or synthesized local control predicates, or the synchronous product of a verified system	279
9	HighSub.h	Header file for HighSub*.cpp	285
10	HighSub.cpp	Read high-level DES files and initialize high-level BDDs	286
11	HighSub1.cpp	Some misc functions for high-level such as printing	294
12	HighSub2.cpp	Verification and synthesis functions for high-level	298
13	LowSub.h	Header file for LowSub*.cpp	308
14	LowSub.cpp	Read high-level DES files and initialize high-level BDDs	309
15	LowSub1.cpp	Some misc functions for low-level such as printing	317
16	LowSub2.cpp	Verify low-level interface (Command-pair)	321
17	LowSub3.cpp	Verification and synthesis functions for high-level	323
18	DES.h	Header file for DES.cpp	334
19	DES.cpp	Process DES files	335
20	type.h	self-defined data types	344
21	errmsg.h	Error Code symbolic constants	345
22	pubfunc.h	Header file for pubfunc.cpp	346
23	pubfunc.cpp	Contains utility functions used in the program	347

Table A.1: Source code files in the software library

No.	File Name	Description	Page
1	main.h	Header file for main.cpp	352
2	main.cpp	A text user interface example to show how to use functions in BDD HISC library	353

Table A.2: Source code files for the library usage example

A.2 Source Code

```
/*************************************************************************
  FILE:  BddHisc.h
  DESCR: C/C++ interface for the HISC synthesis and verifcation package
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*************************************************************************/
#ifndef _BDDHISC_H_
#define _BDDHISC_H_
/**
 * DESCR:   Initialize the HISC enviorment
 * PARA:    None
 * RETURN:  0: sucess  < 0: fail
 * */
int init_hisc();
/**
 * DESCR:   Clear the HISC enviorment
 * PARA:    None
 * RETURN:  0
 * */
int close_hisc();
/**
 * DESCR:  Load a HISC project
 * PARA:   prjfile: HISC project file name (input)
 *         errmsg: returned errmsg (output)
 * RETURN: 0: sucess  < 0: fail
 * */
int load_prj(const char *prjfile, char *errmsg);
/**
 * DESCR:   close opened HISC project
 * PARA:    errmsg: returned errmsg (output)
 * RETURN:  0: sucess  < 0: fail
 * */
int close_prj(char *errmsg);
/**
 * DESCR:   Save the project in the memory to a text file, just for verifying
 *          the loaded project.
 * PARA:    filename: where to save the text file (input)
 *          errmsg: returned errmsg (output)
 * RETURN:  0: sucess  < 0: fail
 * */
int print_prj(const char *filename, char *errmsg);
/**
 * A structure for storing computing result information
 * */
typedef struct Hisc_SuperInfo
{
    double statesize;   /*state size*/
    int nodesize;       /*bdd node size*/
    int time;           /*computing time (seconds)*/
}HISC_SUPERINFO;
/**
 * HISC_SAVESUPER_NONE: Not to save the synthesized supervisor
 * HISC_SAVESUPER_BDD: Save synthesized BDD control predicates
 * HISC_SAVESUPER_AUTOMATA: Save synthesized automata-based supervisor
 * HISC_SAVESUPER_BOTH: Save both
 * */
enum HISC_SAVESUPERTYPE {HISC_SAVESUPER_NONE = 0, HISC_SAVESUPER_BDD = 1,
    HISC_SAVESUPER_AUTOMATA = 2, HISC_SAVESUPER_BOTH = 3};
/**
 * To show a path from the initial state to one bad state or not
 * Currently HISC_SHOW_TRACE is only for telling if a blocking state is
 * deadlock or livelock
 * */
enum HISC_TRACETYPE {HISC_NO_TRACE = 0, HISC_SHOW_TRACE = 1};
/**
 * To save the syn-product for a verified subsystem or not
 * */
enum HISC_SAVEPRODUCTTYPE{HISC_NOTSAVEPRODUCT = 0, HISC_SAVEPRODUCT = 1};
/**
 * To synthesize on reachable statespace or not
 * */
enum HISC_COMPUTEMETHOD{HISC_ONCOREACHABLE = 0, HISC_ONREACHABLE = 1};
/**
 * DESCR:   Synthesize a supervisor for a specified low level
 * PARA:    computemethod: computemethod,synthesize on reachable states or
 *                         on coreachable states (input).
 *          subname: low level name ("all" means all the low levels) (input).
 *          errmsg:  returned errmsg (output)
```

```
 *          pinfo:   returned supervisor info (output)
 *          pnextlow: next low level sub index(initially,it must be 0, mainly
 *                    used for "all")(input)
 *          savetype:  how to save the supervisors (input)
 *          savepath:  where to save the supervisors(input)
 * RETURN:  0: successful < 0: error happened (See errmsg.h)
 * */
int syn_lowsuper(HISC_COMPUTEMETHOD computemethod,
                    char *subname,
                    char *errmsg,
                    HISC_SUPERINFO *pinfo,
                    int* pnextlow,
                    const HISC_SAVESUPERTYPE savetype,
                    const char *savepath);
/**
 * DESCR:   synthesize high level supervisor
 * PARA:    computemethod: synthesize on reachable states or on coreachable
 *                         states (input)
 *          errmsg: returned errmsg (output)
 *          pinfo: returned supervisor infomation(output)
 *          savetype: how to save the supervisors (input)
 *          savepath: where to save the supervisor (input)
 * RETURN: 0: successful < 0: error happened (See errmsg.h)
 */
int syn_highsuper(HISC_COMPUTEMETHOD computemethod,
                    char *errmsg,
                    HISC_SUPERINFO *pinfo,
                    const HISC_SAVESUPERTYPE savetype,
                    const char *savepath);
/**
 * DESCR:   verify high level
 * PARA:    showtrace: show a trace to the bad state (not implemented)(input)
 *          errmsg: returned errmsg (output)
 *          pinfo: returned system infomation (output)
 *          saveproduct: whether to save the syn-product (input)
 *          savepath: where to save the syn-product (input)
 * RETURN:  successful < 0: error happened (See errmsg.h)
 * */
int verify_high(
            HISC_TRACETYPE showtrace,
            char *errmsg,
            HISC_SUPERINFO *pinfo,
            const HISC_SAVEPRODUCTTYPE saveproduct,
            const char *savepath);
/**
 * DESCR:   verify low level
 * PARA:    showtrace: show a trace to the bad state (not implemented) (input)
 *          subname: low level name ("all" means all the low levels) (input)
 *          errmsg: returned errmsg (output)
 *          pinfo: returned system infomation (output)
 *          pnextlow: next low level sub index(initially,it must be 0, mainly
 *                    used for "all") (input)
 *          saveproduct: whether to save syn-product (input)
 *          savepath: where to save syn-product (input)
 * RETURN: 0: successful < 0: error happened (See errmsg.h)
 * */
int verify_low(
        HISC_TRACETYPE showtrace,
        char *subname,
        char *errmsg,
        HISC_SUPERINFO *pinfo,
        int* pnextlow,
        const HISC_SAVEPRODUCTTYPE saveproduct,
        const char *savepath);
#endif
```

```
/*************************************************************************
  FILE:  BddHisc.cpp
  DESCR: Implementation for C/C++ interface for the HISC synthesis and
         verifcation package
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*************************************************************************/
#include "BddHisc.h"
#include "Project.h"
#include <string>
#include <fstream>
#include "errmsg.h"
#include <cassert>
#include "Sub.h"
#include "LowSub.h"
#include "HighSub.h"
#include <sys/time.h>
using namespace std;
CProject *pPrj = NULL;
/**
 * DESCR:   Initialize the HISC enviorment
 * PARA:    None
 * RETURN:  0: sucess  < 0: fail
 * */
int init_hisc()
{
    pPrj = new CProject();
    if (pPrj == NULL)
        return HISC_NOT_ENOUGH_MEMORY;
    return 0;
}
/**
 * DESCR:  Load a HISC project
 * PARA:   prjfile: HISC project file name (input)
 *         errmsg: returned errmsg (output)
 * RETURN: 0: sucess  < 0: fail
 * */
int load_prj(const char *prjfile, char *errmsg)
{
    int iRet = 0;
    assert(prjfile != NULL);
    assert(errmsg != NULL);
    pPrj->InitPrj();
    pPrj->LoadPrj(prjfile);
    strcpy(errmsg, pPrj->GetErrMsg());
    iRet = pPrj->GetErrCode();
    if (pPrj->GetErrCode() < 0)
    {
        if (pPrj->GetErrCode() > HISC_WARN_BLOCKEVENTS) //error happened
            pPrj->ClearPrj();
        //else only a warning
    }
    return iRet;
}
/**
 * DESCR:   close opened HISC project
 * PARA:    errmsg: returned errmsg (output)
 * RETURN:  0: sucess  < 0: fail
 * */
int close_prj(char *errmsg)
{
    int iRet = 0;
    pPrj->ClearPrj();
    pPrj->InitPrj();
    strcpy(errmsg, pPrj->GetErrMsg());
    iRet = pPrj->GetErrCode();
    if (pPrj->GetErrCode() < 0)
    {
        pPrj->ClearPrj();
    }
    return iRet;
}
/**
 * DESCR:  clear the HISC enviorment
 * PARA:   none
 * RETURN: 0
 * */
int close_hisc()
{
    delete pPrj;
```

```
    pPrj = NULL;
    return 0;
}
/**
 * DESCR:   Save the project in the memory to a text file, just for verifying
 *          the loaded project.
 * PARA:    filename: where to save the text file (input)
 *          errmsg: returned errmsg (output)
 * RETURN:  0: sucess  < 0: fail
 * */
int print_prj(const char *filename, char *errmsg)
{
    int iRet = 0;
    assert(filename != NULL);
    assert(errmsg != NULL);
    pPrj->PrintPrjAll(filename);
    strcpy(errmsg, pPrj->GetErrMsg());
    iRet = pPrj->GetErrCode();
    pPrj->ClearErr();
    return iRet;
}
/**
 * DESCR:   Synthesize a supervisor for a specified low level
 * PARA:    computemethod: computemethod,synthesize on reachable states or
 *                         on coreachable states (input).
 *          subname: low level name ("all" means all the low levels) (input).
 *          errmsg:  returned errmsg (output)
 *          pinfo:   returned supervisor info (output)
 *          pnextlow: next low level sub index(initially,it must be 0, mainly
 *                    used for "all")(input)
 *          savetype:  how to save the supervisors (input)
 *          savepath:  where to save the supervisors(input)
 * RETURN:  0: successful < 0: error happened (See errmsg.h)
 * */
int syn_lowsuper(HISC_COMPUTEMETHOD computemethod,
                 char *subname,
                 char *errmsg,
                 HISC_SUPERINFO *pinfo,
                 int* pnextlow,
                 const HISC_SAVESUPERTYPE savetype,
                 const char *savepath)
{
    int iRet = 0;
    assert(subname != NULL);
    assert(errmsg != NULL);
    assert(pinfo != NULL);
    assert(pnextlow != NULL);
    assert(*pnextlow >= 0);
    assert((savetype == HISC_SAVESUPER_NONE) || savepath != NULL);
    string sSubName = subname;
    if (sSubName == "all" || *pnextlow > 0)
    {
        (*pnextlow)++;
        if (*pnextlow <= pPrj->GetNumofLows())
        {
            sSubName = pPrj->GetSub(*pnextlow)->GetSubName();
            strcpy(subname, sSubName.data());
        }
        if (*pnextlow == pPrj->GetNumofLows())
            *pnextlow = -1;
    }
    int i = 0;
    for (i = 0; i < pPrj->GetNumofLows(); i++)
    {
        if (pPrj->GetSub(i + 1)->GetSubName() == sSubName)
            break;
    }
    if (i < pPrj->GetNumofLows())
    {
        string sSavePath;
        if (savetype != HISC_SAVESUPER_NONE)
            sSavePath = savepath;
        time_t tstart;
        time(&tstart);
        if (pPrj->GetSub(i + 1)->SynSuper(computemethod, *pinfo, savetype,
            sSavePath) < 0)
```

```
        {
            strcpy(errmsg, pPrj->GetErrMsg());
            iRet = pPrj->GetErrCode();
            pPrj->ClearErr();
        }
        time_t tend;
        time(&tend);
        pinfo->time = tend - tstart;
    }
    else
    {
        string sErr = "Unable to find the low level ";
        sErr += sSubName;
        strcpy(errmsg, sErr.data());
        iRet = HISC_NONEXISTED_LEVEL_NAME;
    }
    return iRet;
}
/**
 * DESCR:   synthesize high level supervisor
 * PARA:    computemethod: synthesize on reachable states or on coreachable
 *                         states (input)
 *          errmsg: returned errmsg (output)
 *          pinfo: returned supervisor infomation(output)
 *          savetype: how to save the supervisors (input)
 *          savepath: where to save the supervisor (input)
 * RETURN: 0: successful < 0: error happened (See errmsg.h)
 */
int syn_highsuper(HISC_COMPUTEMETHOD computemethod,
                  char *errmsg,
                  HISC_SUPERINFO *pinfo,
                  const HISC_SAVESUPERTYPE savetype,
                  const char *savepath)
{
    assert(errmsg != NULL);
    assert(pinfo != NULL);
    assert((savetype == HISC_SAVESUPER_NONE) || savepath != NULL);
    int iRet = 0;
    string sSavePath;
    if (savetype != HISC_SAVESUPER_NONE)
        sSavePath = savepath;
    time_t tstart;
    time(&tstart);
    pPrj->GetSub(0)->SynSuper(computemethod, *pinfo, savetype, sSavePath);
    strcpy(errmsg, pPrj->GetErrMsg());
    iRet = pPrj->GetErrCode();
    pPrj->ClearErr();
    time_t tend;
    time(&tend);
    pinfo->time = tend - tstart;
    return iRet;
}
/**
 * DESCR:   verify high level
 * PARA:    showtrace: show a trace to the bad state (not implemented)(input)
 *          errmsg: returned errmsg (output)
 *          pinfo: returned system infomation (output)
 *          saveproduct: whether to save the syn-product (input)
 *          savepath: where to save the syn-product (input)
 * RETURN:  successful < 0: error happened (See errmsg.h)
 * */
int verify_high(
            HISC_TRACETYPE showtrace,
            char *errmsg,
            HISC_SUPERINFO *pinfo,
            const HISC_SAVEPRODUCTTYPE saveproduct,
            const char *savepath)
{
    assert(errmsg != NULL);
    assert(pinfo != NULL);
    assert((saveproduct == HISC_NOTSAVEPRODUCT) || savepath != NULL);
    int iRet = 0;
    string sSavePath;
    if (saveproduct != HISC_NOTSAVEPRODUCT)
        sSavePath = savepath;
    time_t tstart;
```

```
    time(&tstart);
    if (pPrj->GetSub(0)->VeriSub(showtrace, *pinfo, saveproduct, sSavePath) < 0)
    {
        strcpy(errmsg, pPrj->GetErrMsg());
        iRet = pPrj->GetErrCode();
    }
    pPrj->ClearErr();
    time_t tend;
    time(&tend);
    pinfo->time = tend - tstart;
    return iRet;
}
/**
 * DESCR:   verify low level
 * PARA:    showtrace: show a trace to the bad state (not implemented) (input)
 *          subname: low level name ("all" means all the low levels) (input)
 *          errmsg: returned errmsg (output)
 *          pinfo: returned system infomation (output)
 *          pnextlow: next low level sub index(initially,it must be 0, mainly
 *                    used for "all") (input)
 *          saveproduct: whether to save syn-product (input)
 *          savepath: where to save syn-product (input)
 * RETURN: 0: successsful < 0: error happened (See errmsg.h)
 * */
int verify_low(
        HISC_TRACETYPE showtrace,
        char *subname,
        char *errmsg,
        HISC_SUPERINFO *pinfo,
        int* pnextlow,
        const HISC_SAVEPRODUCTTYPE saveproduct,
        const char *savepath)
{
    assert(subname != NULL);
    assert(errmsg != NULL);
    assert(pinfo != NULL);
    assert(pnextlow != NULL);
    assert(*pnextlow >= 0);
    assert((saveproduct == HISC_NOTSAVEPRODUCT) || savepath != NULL);
    int iRet = 0;
    string sSubName = subname;
    if (sSubName == "all" || *pnextlow > 0)
    {
        (*pnextlow)++;
        if (*pnextlow <= pPrj->GetNumofLows())
        {
            sSubName = pPrj->GetSub(*pnextlow)->GetSubName();
            strcpy(subname, sSubName.data());
        }
        if (*pnextlow == pPrj->GetNumofLows())
            *pnextlow = -1;
    }
    int i = 0;
    for (i = 0; i < pPrj->GetNumofLows(); i++)
    {
        if (pPrj->GetSub(i + 1)->GetSubName() == sSubName)
            break;
    }
    if (i < pPrj->GetNumofLows())
    {
        string sSavePath;
        if (saveproduct != HISC_NOTSAVEPRODUCT)
            sSavePath = savepath;
        time_t tstart;
        time(&tstart);
        if (pPrj->GetSub(i + 1)->VeriSub(showtrace, *pinfo, saveproduct,
            sSavePath) < 0)
        {
            strcpy(errmsg, pPrj->GetErrMsg());
            iRet = pPrj->GetErrCode();
            pPrj->ClearErr();
        }
        time_t tend;
        time(&tend);
        pinfo->time = tend - tstart;
    }
```

```
    else
    {
        string sErr = "Unable to find the low level ";
        sErr += sSubName;
        strcpy(errmsg, sErr.data());
        iRet = HISC_NONEXISTED_LEVEL_NAME;
    }
    return iRet;
}
```

```
/*************************************************************************
  FILE:  Project.h
  DESCR: Header file of Project.cpp (Process project file (.prj))
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*************************************************************************/
#ifndef _PROJECT_H_
#define _PROJECT_H_
#include <string>
#include <map>
#include "type.h"
#include <fstream>
#include "BddHisc.h"
using namespace std;
class CSub;
class CHighSub;
class CLowSub;
class CProject
{
public:
    CProject();
    virtual ~CProject();
    void InitPrj();
    void ClearPrj();
    int PrintPrj(ofstream& fout);
    int PrintPrjAll(string sFileName);
public:
    void LoadPrj(string vsPrjFile);
    int GenEventIndex(const EVENTSUB vEventSub, const int viSubIndex,
                      const unsigned short vusiLocalEventIndex);
    EVENTSUB GenEventInfo(const int viEventIndex, EVENTSUB & vEventSub,
                          int & viSubIndex,
                          unsigned short & vusiLocalEventIndex);
    int AddPrjEvent(const string & vsEventName, const int viEventIndex,
                        EVENTSUB &vEventSub, int & viSubIndex);
    int SearchPrjEvent(const string & vsEventName);
    int GetNumofLows() const {return m_iNumofLows;};
    CSub* GetSub(const int iSubIndex);
    void SetErr(const string & vsErrMsg, const int viErrCode);
    void ClearErr();
    const char *GetErrMsg() const {return m_sErrMsg.data();};
    int GetErrCode() const {return m_iErrCode;};
    INVEVENTS & GetInvAllEventsMap() {return m_InvAllEventsMap;};
private:
    string GetSubFileFromPrjFile(const string & vsPrjFile,
                                 const string &vsSubDir);
private:
    string m_sPrjName;  //Project name
    int m_iNumofLows;   //Number of low-level subsystems
    CHighSub *m_pHighSub; //High-level pointer
    CLowSub **m_pLowSub; //Pointers for low-levels
    EVENTS m_AllEventsMap; //The map containing all the events in this project
                           //(Event Name (key), Event global index)
    INVEVENTS m_InvAllEventsMap; //The map containing all the events in this
                                 //project (Event global index (key), Event Name)
    string m_sErrMsg; //Error msg during processing this project
    int m_iErrCode; //Error code during processing this project
};
#endif //_PROJECT_H_
```

```
/*************************************************************************
  FILE:  Project.cpp
  DESCR: Processing Project file (.prj)
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*************************************************************************/
#include "Project.h"
#include <fstream>
#include "errmsg.h"
#include "type.h"
#include "pubfunc.h"
#include "HighSub.h"
#include "LowSub.h"
#include <iostream>
#include <string>
#include <cassert>
using namespace std;
/**
 * DESCR:   Constructor
 * PARA:    None
 * RETURN:  None
 * ACCESS:  public
 */
CProject::CProject()
{
    InitPrj();
}
/**
 * DESCR:   Destructor
 * PARA:    None
 * RETURN:  None
 * ACCESS:  public
 */
CProject::~CProject()
{
    ClearPrj();
}
/**
 * DESCR:   Initialize data members
 * PARA:    None
 * RETURN:  None
 * ACCESS:  public
 */
void CProject::InitPrj()
{
    m_sPrjName.clear();
    m_iNumofLows = 0;
    m_pHighSub = NULL;
    m_pLowSub = NULL;
    m_AllEventsMap.clear();
    m_InvAllEventsMap.clear();
    m_iErrCode = 0;
    m_sErrMsg.clear();
}
/**
 * DESCR:   Release memory used by data members
 * PARA:    None
 * RETURN:  None
 * ACCESS:  public
 */
void CProject::ClearPrj()
{
    //can't change the order of deleting highsub and lowsub, see ~CSub()
    delete m_pHighSub;
    m_pHighSub = NULL;
    if (m_pLowSub != NULL)
    {
        for (int i = 0; i < m_iNumofLows; i++)
        {
            delete m_pLowSub[i];
            m_pLowSub[i] = NULL;
        }
    }
    delete[] m_pLowSub;
    m_pLowSub = NULL;
}
/**
 * DESCR:   Load a project file
 * PARA:    vsPrjFile: project file name with path
 * RETURN:  None
 * ACCESS:  public
 */
```

```
void CProject::LoadPrj(string vsPrjFile)
{
    ifstream fin;
    string sErr;
    try
    {
        string sPrjFile = str_trim(vsPrjFile);
        if (sPrjFile.length() <= 4)
        {
            SetErr("Invalid project file name!", HISC_BAD_PRJ_FILE);
            throw -1;
        }
        if (sPrjFile.substr(sPrjFile.length() - 4) != ".prj")
        {
            SetErr("Invalid project file name!", HISC_BAD_PRJ_FILE);
            throw -1;
        }
        fin.open(sPrjFile.data(), ifstream::in);
        //unable to find project file
        if (!fin)
        {
            SetErr("Unable to open the project file: " + sPrjFile,
                                HISC_BAD_PRJ_FILE);
            throw -1;
        }
        char scBuf[MAX_LINE_LENGTH];
        string sLine;
        int iField = -1; //0: SYSTEM 1:LOW 2:HIGH
        char *scFieldArr[] = {"SYSTEM", "LOW", "HIGH"};
        int iNumofLows = -1;
        while (fin.getline(scBuf,  MAX_LINE_LENGTH))
        {
            sLine = str_nocomment(scBuf);
            sLine = str_trim(sLine);
            if (sLine.empty())
                continue;
            //Field name
            if (sLine[0] == '[' && sLine[sLine.length() - 1] == ']')
            {
                sLine = sLine.substr(1, sLine.length() - 1);
                sLine = sLine.substr(0, sLine.length() - 1);
                sLine = str_upper(str_trim(sLine));
                iField++;
                if (iField > 2)
                {
                    SetErr("To many fields!", HISC_BAD_PRJ_FORMAT);
                    throw -1;
                }
                if (sLine != scFieldArr[iField])
                {
                    sErr = "Field" + sLine + "--name or order is wrong!";
                    SetErr(sErr, HISC_BAD_PRJ_FORMAT);
                    throw -1;
                }
            }
            //Field content
            else
            {
                switch (iField)
                {
                case 0:  //SYSTEM
                    if (m_sPrjName.length() == 0)
                        m_sPrjName = sLine;
                    else
                    {
                        SetErr("Project name can only be on one line!",
                               HISC_BAD_PRJ_FORMAT);
                        throw -1;
                    }
                    break;
                case 1: //LOW
                    if (iNumofLows < 0)
                    {
                        if (!IsInteger(sLine))
                        {
                            SetErr("No number of low sub subsystems.",
                                   HISC_BAD_PRJ_FORMAT);
                            throw -1;
                        }
```

```
                    m_iNumofLows = atoi(sLine.data());
                    if (m_iNumofLows <= 0)
                    {
                        sErr = "The system must have ";
                        sErr += "at least one low sub system.";
                        SetErr(sErr, HISC_BAD_PRJ_FORMAT);
                        throw -1;
                    }
                    iNumofLows++;
                    //assign memory space for Low Sub array
                    m_pLowSub = new (CLowSub *)[m_iNumofLows];
                    for (int i = 0; i < m_iNumofLows; i++)
                        m_pLowSub[i] = NULL;
                }
                else
                {
                    if (iNumofLows < m_iNumofLows)
                    {
                        string sLowFile =
                                    GetSubFileFromPrjFile(sPrjFile, sLine);
                        m_pLowSub[iNumofLows] =
                                    new CLowSub(sLowFile, iNumofLows + 1);
                        if (m_pLowSub[iNumofLows]->LoadSub() < 0)
                            throw -1;
                    }
                    else
                    {
                        SetErr("The number of low subs is not correct!",
                                HISC_BAD_PRJ_FORMAT);
                        throw -1;
                    }
                    iNumofLows++;
                }
                break;
            case 2: //HIGH
                if (m_pHighSub == NULL)
                {
                    string sHighFile =
                                GetSubFileFromPrjFile(sPrjFile, sLine);
                    m_pHighSub = new CHighSub(sHighFile, 0);
                    if (m_pHighSub->LoadSub() < 0)
                        throw -1;
                }
                else
                {
                    SetErr("Only one high sub is allowed.",
                           HISC_BAD_PRJ_FORMAT);
                    throw -1;
                }
                break;
            default:
                SetErr("Unknown bad project file format!",
                            HISC_BAD_PRJ_FORMAT);
                throw -1;
                break;
            }
        }
    } //while
    if (iField != 2)
    {
        SetErr("Too few fields.", HISC_BAD_PRJ_FORMAT);
        throw -1;
    }
    if (m_pHighSub == NULL)
    {
        SetErr("No high sub.", HISC_BAD_PRJ_FORMAT);
        throw -1;
    }
    if (m_iNumofLows <= 0)
    {
        SetErr("No low sub.", HISC_BAD_PRJ_FORMAT);
        throw -1;
    }
    if (iNumofLows < m_iNumofLows)
    {
        SetErr("Too few low subs.", HISC_BAD_PRJ_FORMAT);
        throw -1;
    }
    fin.close();
}
catch (int iError)
{
```

```
        if (fin.is_open())
            fin.close();
        return;
    }
    return;
}
/**
 * DESCR:   Set error msg and err code in this project
 * PARA:    vsvsErrMsg: Error message
 *          viErrCode: Error Code
 * RETURN:  None
 * ACCESS:  public
 */
void CProject::SetErr(const string & vsErrMsg, const int viErrCode)
{
    m_iErrCode = viErrCode;
    m_sErrMsg = vsErrMsg;
#ifdef  VERBOSE
    cout << "Error: " << m_sErrMsg << endl;
#endif
    return;
}
/**
 * DESCR:   Clear error msg and err code in this project
 * PARA:    None
 * RETURN:  None
 * ACCESS:  public
 */
void CProject::ClearErr()
{
    m_iErrCode = 0;
    m_sErrMsg.empty();
    return;
}
/*
 * DESCR:   Generate a sub file name with path (*.sub) from a prj file name
 *          with path (.prj) and a sub file name without path.
 *          ex: vsSubFile = "/home/roger/sim.prj", vsDES = vsSubDir = "low1",
 *              will return "/home/roger/sim/abc/low1.sub"
 * PARA:    vsPrjFile: prj file name with path
 *          vsSubDir: sub file name without path
 * RETURN:  Generated sub file name with path
 * ACCESS:  private
 */
string CProject::GetSubFileFromPrjFile(const string & vsPrjFile,
                                       const string &vsSubDir)
{
    assert(vsPrjFile.length() > 4);
    assert(vsPrjFile.substr(vsPrjFile.length() - 4) == ".prj");
    assert(vsSubDir.length() > 0);
    string sSubFile = vsSubDir + "/" + vsSubDir + ".sub";
    unsigned int iPos = vsPrjFile.find_last_of('/');
    if ( iPos == string::npos)
        return sSubFile;
    else
        return vsPrjFile.substr(0, iPos + 1) + sSubFile;
}
/**
 * DESCR:   Get level i  pointer.
 * PARA:    iSubIndex: level index (0: high-level, 1, 2,...: low-levels)
 * RETURN:  Level-i pointer
 * ACCESS:  public
 */
CSub* CProject::GetSub(const int iSubIndex)
{
    assert(iSubIndex <= m_iNumofLows);
    if (iSubIndex == 0)
    {
        assert(m_pHighSub != NULL);
        return m_pHighSub;
    }
    else
    {
        assert(m_pLowSub != NULL);
        assert(m_pLowSub[iSubIndex - 1] != NULL);
        return m_pLowSub[iSubIndex - 1];
    }
}
/**
 * DESCR:   Generate global event index from the event info in para
```

```
 * PARA:    vEventSub(H/R/A/L, the first 4 bits), (input)
 *          viSubIndex(Sub index, highsub = 0, low sub start from 1.
 *                     Next 12 bits), (input)
 *          vusiLocalEventIndex(local event index, odd: controllable,
 *                              even:uncontrollab. The rest 16 bits) (input)
 * RETURN:  Generated global event index
 * ACCESS:  public
 */
int CProject::GenEventIndex(const EVENTSUB vEventSub, const int viSubIndex,
                    const unsigned short vusiLocalEventIndex)
{
    assert(viSubIndex >= 0 && viSubIndex <= 4096);
    int iEventIndex = vEventSub;
    iEventIndex = iEventIndex << 28;
    int iSubIndex = 0;
    iSubIndex = viSubIndex;
    iSubIndex = iSubIndex << 16;
    iEventIndex += iSubIndex;
    iEventIndex += vusiLocalEventIndex;
    return iEventIndex;
}
/**
 * DESCR:   Generate event type(H/R/A/L), sub index and local event index from
 * global event index.
 * PARA:    viEventIndex: global event index (input)
 *          EventSub: H/R/A/L, see type.h (output)
 *          viSubIndex(Sub index, highsub = 0, low sub start from 1) (output)
 *          vusiLocalEventIndex(local event index, odd: controllable,
 *                              even:uncontrollable) (output)
 * RETURN:  Generated global event index
 * ACCESS:  public
 */
EVENTSUB CProject::GenEventInfo(const int viEventIndex, EVENTSUB & vEventSub,
                            int & viSubIndex,
                            unsigned short & vusiLocalEventIndex)
{
    int iEventSub = viEventIndex >> 28;
    assert(iEventSub <= 3);
    vEventSub = (EVENTSUB)iEventSub;
    viSubIndex = (viEventIndex & 0x0FFF0000) >> 16;
    vusiLocalEventIndex = (viEventIndex & 0x0000FFFF);
    return (EVENTSUB)iEventSub;
}
/*
 * DESCR:   Add an event to CProject event map
 *          If the event exists already exists in the map, the  it should have
 *          same global index;  Otherwise the event sets are not disjoint
 * PARA:    vsEventName: Event name(input)
 *          viEventIndex: global event index (input)
 *          cEventSub: Event type ('H", 'L', 'R', 'A')
 *                     (output, only for new events)
 *          cControllable: Controllable? ('Y', 'N')(output)(only for new events)
 * RETURN:  0: success
 *          <0 the event sets are not disjoint.
 * ACCESS:  public
 */
int CProject::AddPrjEvent(const string & vsEventName, const int viEventIndex,
                          EVENTSUB &vEventSub, int & viSubIndex)
{
    EVENTS::const_iterator citer;
    citer = m_AllEventsMap.find(vsEventName);
    if (citer != m_AllEventsMap.end())  //the event exists, check if the global
                                        //event index is same.
    {
        if (citer->second != viEventIndex)
        {
            vEventSub = (EVENTSUB)(citer->second >> 28);
            viSubIndex = (citer->second & 0x0FFF0000) >> 16;
            return -1;
        }
    }
    else  //the event does not exist
    {
        m_AllEventsMap[vsEventName] = viEventIndex;
        m_InvAllEventsMap[viEventIndex] = vsEventName;
    }
    return 0;
}
```

```
/*
 * DESCR:   Search an event by its name
 * PARA:    vsEventName: Event name(input)
 * RETURN:  >0: Gloable event index
 *          <0: not found
 * ACCESS:  public
 */
int CProject::SearchPrjEvent(const string & vsEventName)
{
    EVENTS::const_iterator citer;
    citer = m_AllEventsMap.find(vsEventName);
    if (citer != m_AllEventsMap.end())  //the event exists
        return citer->second;
    else  //the event does not exist
        return -1;
}
/**
 * DESCR:   Save prj info in memory to a file (for checking)
 * PARA:    fout: output file stream
 * RETURN:  0: sucess -1: fail
 * ACCESS:  public
 */
int CProject::PrintPrj(ofstream& fout)
{
    try
    {
        fout << "# Project File." << endl << endl;
        fout << "[SYSTEM]" << endl;
        fout << m_sPrjName << endl;
        fout << endl;
        fout << "[HIGH]" << endl;
        fout << m_pHighSub->GetSubName() << endl;
        fout << endl;
        fout << "[LOW]" << endl;
        fout << m_iNumofLows << endl;
        for (int i = 0; i < m_iNumofLows; i++)
        {
            fout << m_pLowSub[i]->GetSubName() << endl;
        }
        fout << "################################################" << endl;
    }
    catch(...)
    {
        return -1;
    }
    return 0;
}
/**
 * DESCR:   Save all the sub files in this project to a text file for checking
 * PARA:    sFileName: output file name
 * RETURN:  0: sucess -1: fail
 * ACCESS:  public
 */
int CProject::PrintPrjAll(string sFileName)
{
    ofstream fout;
    try
    {
        fout.open(sFileName.data());
        if (!fout)
            throw -1;
        PrintPrj(fout);
        if (m_pHighSub->PrintSubAll(fout) < 0)
            throw -1;
        for (int i = 0; i < m_iNumofLows; i++)
        {
            if (m_pLowSub[i]->PrintSubAll(fout) < 0)
                throw -1;
        }
        fout.close();
    }
    catch(...)
    {
        if (fout.is_open())
            fout.close();
        SetErr(sFileName + ":Unable to create the print file.",
                HISC_BAD_PRINT_FILE);
        return -1;
    }
    return 0;
}
```

```
/*************************************************************************
  FILE:  Sub.h
  DESCR: Header file for Sub*.cpp (subsystem processing file)
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*************************************************************************/
#ifndef _SUB_H_
#define _SUB_H_
#include <string>
#include <map>
#include "type.h"
#include "DES.h"
#include <fstream>
#include <fdd.h>
#include "BddHisc.h"
using namespace std;
class CSub
{
public: //public methods
    CSub(const string &  vsSubFile, int viSubIndex);
    virtual ~CSub();
    virtual unsigned short AddSubEvent(const string & vsEventName,
                                       const EVENTSUB vEventSub,
                                       const EVENTTYPE vEventType);
    virtual int PrintSub(ofstream & fout) = 0;
    virtual int PrintSubAll(ofstream& fout) = 0;
    virtual string SearchEventName(EVENTSUB k,unsigned short usiLocalIndex) = 0;
    virtual int LoadSub() = 0;
    virtual int SynSuper(const HISC_COMPUTEMETHOD computemethod,
                         HISC_SUPERINFO &superinfo,
                         const HISC_SAVESUPERTYPE savetype,
                         const string& savepath) = 0;
    virtual int VeriSub(const HISC_TRACETYPE showtrace,
                        HISC_SUPERINFO & superinfo,
                        const HISC_SAVEPRODUCTTYPE savetype,
                        const string& savepath) = 0;
public:  //access methods
    virtual string GetSubName() const {return m_sSubName;};
    virtual int GetSubIndex() const {return m_iSubIndex;};
    virtual int GetNumofDES() const
                {return m_iNumofPlants + m_iNumofSpecs + m_iNumofIntfs;};
    virtual unsigned short GetMaxUnCon(EVENTSUB EventSub)
                {return m_usiMaxUnCon[EventSub];};
    virtual unsigned short GetMaxCon(EVENTSUB EventSub)
                {return m_usiMaxCon[EventSub];};
private: //DES reorder related memebers
    int ** m_piCrossMatrix;
    int DESReorder_Sift();
    double TotalCross_Sift(double dOldCross, double dSwapCross,
                           int iCur, int iFlag);
    double cross(int i, int j);
    int DESReorder_Force();
    void UpdatePos();
    void InsertDES(int iCur, int iPos);
    double TotalCross_Force();
    double Force(int i);
    int InitialDESOrder();
protected:  //protected methods
    virtual string GetDESFileFromSubFile(const string & vsSubFile,
                   const string &vsDES);
    virtual int MakeBdd() = 0;
    virtual int InitBddFields();
    virtual int ClearBddFields();
    int DESReorder();
    //SaveSuper related methods
    int SaveSuper(const bdd & bddReach, const HISC_SAVESUPERTYPE savetype,
                  const string & savepath);
    int PrintStateSet(ofstream & fout, const bdd & bddStateSet,
                      STATES & statesMap, int viSetFlag);
    int PrintEvents(ofstream & fout);
    int PrintTextTrans(ofstream & fout, bdd & bddController,
                       EVENTSUB EventSub, unsigned short usiLocalIndex,
                       const bdd & bddReach, string sEventName,
                       STATES & statesMap);
    bdd SimplifyController(const bdd & bddController, EVENTSUB EventSub,
                           const unsigned short usiIndex);
protected:  //fields
```

```
    string m_sSubFile;   //this subsytem file name(".sub") with path.
    string m_sSubName;   //This subsystem name
    int m_iSubIndex;      //This subsystem index (High: 0 Low: 1,2,...)
    int m_iNumofPlants;   //Number of Plant DES
    int m_iNumofSpecs;    //Number of Specification DES
    int m_iNumofIntfs;    //Number of Interface DES
                          //(High: all interface DES; Low: 1)
    CDES **m_pDESArr;     //DES Array for all the DES in high or low levels.
                          //(High: including all interface DES,
                          //Low: only including 1 DES for this subsystem)
    LOCALEVENTS m_SubEventsMap; //save all the events map in this subsytems
                                //(name(key), local index(16 bits))
                                //just for compute local event index.
    unsigned short m_usiMaxCon[4]; //Max index of controllable events (1,3,...)
    unsigned short m_usiMaxUnCon[4];//Max index of uncontrollable events(2,4,..)
    /*BDD needed fields*/
    int m_iNumofBddNormVar; //Num of BDD normal variables in the sub.
    int *m_piDESOrderArr; //DES indices organized as clusters.
    int *m_piDESPosArr; //DES positions int the m_piDESOrderArr
    bdd m_bddInit;  //Initial state predicate
    bdd m_bddMarking;  //Marking states predicate
    bdd m_bddSuper; //The generated supervisor
    ////////////////////////////////////////////////////////////////
    //Transition predicates and its variable sets, variable pairs.
    //0: High level events
    //1: Request events
    //2: Answer events
    //3: Low level events
    ////////////////////////////////////////////////////////////////
    //Transition predicates
    bdd *m_pbdd_ConTrans[4];
    bdd *m_pbdd_UnConTrans[4];
    bdd *m_pbdd_UnConPlantTrans[4];
    //variable(DES index) set for transition predicates
    bdd *m_pbdd_ConVar[4];
    bdd *m_pbdd_ConVarPrim[4];
    bdd *m_pbdd_UnConVar[4];
    bdd *m_pbdd_UnConVarPrim[4];
    //plant part variables
    bdd *m_pbdd_UnConPlantVar[4];
    bdd *m_pbdd_UnConPlantVarPrim[4];
    bdd *m_pbdd_ConPhysicVar[4];   //for simplifying controller (note: physical)
    bdd *m_pbdd_ConPhysicVarPrim[4];//for simplifying controller (note:physical)
    //variable pairs(normal-prime)
    bddPair **m_pPair_Con[4];
    bddPair **m_pPair_UnCon[4];
    bddPair **m_pPair_ConPrim[4];
    bddPair **m_pPair_UnConPrim[4];
};
#endif //_SUB_H_
```

```
/*****************************************************************************
  FILE:  Sub.cpp
  DESCR: Processing subsystem file(.sub)
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*****************************************************************************/
#include "Sub.h"
#include <string>
#include "DES.h"
#include <map>
#include <cassert>
#include <fdd.h>
#include "pubfunc.h"
#include "errmsg.h"
#include "type.h"
#include "Project.h"
using namespace std;
extern CProject *pPrj;
/**
 * DESCR:   Constructor
 * PARA:    vsSubFile: subsystem file name with path (.sub)(input)
 *          viSubIndex: subsystem index (high: 0, low: 1,2,...)(input)
 * RETURN:  None
 * ACCESS:  public
 */
CSub::CSub(const string & vsSubFile, int viSubIndex)
{
    m_sSubFile = vsSubFile;
    m_sSubName.clear();
    m_iSubIndex = viSubIndex;
    m_iNumofPlants = -1;
    m_iNumofSpecs = -1;
    m_iNumofIntfs = -1;
    m_pDESArr = NULL;
    m_SubEventsMap.clear();
    for (int k = 0; k < 4; k++)
    {
        m_usiMaxCon[k] = 0xFFFF;
        m_usiMaxUnCon[k] = 0x0;
    }
    m_piDESOrderArr = NULL;
    m_piDESPosArr = NULL;
    InitBddFields();
}
/**
 * DESCR:   Destructor
 * PARA:    None
 * RETURN:  None
 * ACCESS:  public
 */
CSub::~CSub()
{
    if (m_pDESArr != NULL)
    {
        int iNumofDES = this->GetNumofDES();
        for (int i = 0; i < iNumofDES; i++)
        {
            if (m_pDESArr[i] != NULL)
            {
                if (m_pDESArr[i]->GetDESType() == INTERFACE_DES && m_iSubIndex == 0)
                    continue;
                else
                {
                    delete m_pDESArr[i];
                    m_pDESArr[i] = NULL;
                }
            }
        }
        delete[] m_pDESArr;
        m_pDESArr = NULL;
    }
    delete[] m_piDESOrderArr;
    m_piDESOrderArr = NULL;
    delete[] m_piDESPosArr;
    m_piDESPosArr = NULL;
    ClearBddFields();
}
/*
 * DESCR:   Initialize BDD related data members
 * PARA:    None
 * RETURN:  0
```

```
 * ACCESS:  protected
 */
int CSub::InitBddFields()
{
    m_iNumofBddNormVar = 0;
    m_bddInit = bddtrue;
    m_bddMarking = bddtrue;
    m_bddSuper = bddfalse;
    for (int k = 0; k < 4; k++)
    {
        m_pbdd_ConTrans[k] = NULL;
        m_pbdd_UnConTrans[k] = NULL;
        m_pbdd_UnConPlantTrans[k] = NULL;
        m_pbdd_ConVar[k] = NULL;
        m_pbdd_ConVarPrim[k] = NULL;
        m_pbdd_UnConVar[k] = NULL;
        m_pbdd_UnConVarPrim[k] = NULL;
        m_pbdd_UnConPlantVar[k] = NULL;
        m_pbdd_UnConPlantVarPrim[k] = NULL;
        m_pbdd_ConPhysicVar[k] = NULL;
        m_pbdd_ConPhysicVarPrim[k] = NULL;
        m_pPair_Con[k] = NULL;
        m_pPair_UnCon[k] = NULL;
        m_pPair_ConPrim[k] = NULL;
        m_pPair_UnConPrim[k] = NULL;
    }
    return 0;
}
/*
 * DESCR:   Release memory for BDD related data members
 * PARA:    None
 * RETURN:  0
 * ACCESS:  protected
 */
int CSub::ClearBddFields()
{
    for (int k = 0; k < 4; k++)
    {
        delete[] m_pbdd_ConTrans[k];
        m_pbdd_ConTrans[k] = NULL;
        delete[] m_pbdd_UnConTrans[k];
        m_pbdd_UnConTrans[k] = NULL;
        delete[] m_pbdd_UnConPlantTrans[k];
        m_pbdd_UnConPlantTrans[k] = NULL;
        delete[] m_pbdd_ConVar[k];
        m_pbdd_ConVar[k] = NULL;
        delete[] m_pbdd_UnConVar[k];
        m_pbdd_UnConVar[k] = NULL;
        delete[] m_pbdd_ConVarPrim[k];
        m_pbdd_ConVarPrim[k] = NULL;
        delete[] m_pbdd_UnConVarPrim[k];
        m_pbdd_UnConVarPrim[k] = NULL;
        delete[] m_pbdd_UnConPlantVar[k];
        m_pbdd_UnConPlantVar[k] = NULL;
        delete[] m_pbdd_UnConPlantVarPrim[k];
        m_pbdd_UnConPlantVarPrim[k] = NULL;
        delete[] m_pbdd_ConPhysicVar[k];
        m_pbdd_ConPhysicVar[k] = NULL;
        delete[] m_pbdd_ConPhysicVarPrim[k];
        m_pbdd_ConPhysicVarPrim[k] = NULL;
        if (m_pPair_UnCon[k] != NULL)
        {
            for (int i = 0; i < m_usiMaxUnCon[k]; i += 2)
            {
                if (m_pPair_UnCon[k][i/2] != NULL)
                {
                    bdd_freepair(m_pPair_UnCon[k][i/2]);
                    m_pPair_UnCon[k][i/2] = NULL;
                }
            }
            delete[] m_pPair_UnCon[k];
            m_pPair_UnCon[k] = NULL;
        }
        if (m_pPair_Con[k] != NULL)
        {
```

```
            for (int i = 1; i < (unsigned short)(m_usiMaxCon[k] + 1); i += 2)
            {
                if (m_pPair_Con[k][(i - 1)/2] != NULL)
                {
                    bdd_freepair(m_pPair_Con[k][(i - 1)/2]);
                    m_pPair_Con[k][(i - 1)/2] = NULL;
                }
            }
            delete[] m_pPair_Con[k];
            m_pPair_Con[k] = NULL;
        }
        if (m_pPair_UnConPrim[k] != NULL)
        {
            for (int i = 0; i < m_usiMaxUnCon[k]; i += 2)
            {
                if (m_pPair_UnConPrim[k][i/2] != NULL)
                {
                    bdd_freepair(m_pPair_UnConPrim[k][i/2]);
                    m_pPair_UnConPrim[k][i/2] = NULL;
                }
            }
            delete[] m_pPair_UnConPrim[k];
            m_pPair_UnConPrim[k] = NULL;
        }
        if (m_pPair_ConPrim[k] != NULL)
        {
            for (int i = 1; i < (unsigned short)(m_usiMaxCon[k] + 1); i += 2)
            {
                if (m_pPair_ConPrim[k][(i - 1)/2] != NULL)
                {
                    bdd_freepair(m_pPair_ConPrim[k][(i - 1)/2]);
                    m_pPair_ConPrim[k][(i - 1)/2] = NULL;
                }
            }
            delete[] m_pPair_ConPrim[k];
            m_pPair_ConPrim[k] = NULL;
        }
    }
    return 0;
}
/*
 * DESCR:   Generate a DES file name with path (*.hsc) from a sub file name
 *          with path (.sub) and a DES file name without path.
 *          ex: vsSubFile = "/home/roger/high.sub", vsDES = "AttchCase.hsc",
 *              will return "/home/roger/AttchCase.hsc"
 * PARA:    vsSubFile: sub file name with path
 *          vsDES: DES file name without path
 * RETURN:  Generated DES file name with path
 * ACCESS:  protected
 */
string CSub::GetDESFileFromSubFile(const string & vsSubFile,
                        const string &vsDES)
{
    assert(vsSubFile.length() > 4);
    assert(vsSubFile.substr(vsSubFile.length() - 4) == ".sub");
    assert(vsDES.length() > 0);
    string sDES = vsDES;
    if (sDES.length() > 4)
    {
        if (sDES.substr(sDES.length() - 4) == ".hsc")
        {
            sDES = sDES.substr(0, sDES.length() - 4);
        }
    }
    sDES += ".hsc";
    unsigned int iPos = vsSubFile.find_last_of('/');
    if ( iPos == string::npos)
        return sDES;
    else
        return vsSubFile.substr(0, iPos + 1) + sDES;
}
/**
 * DESCR:   Add events to the event Map of this sub. If the event already exits,
 *          return its index; Otherwise generate a new 16 bit unsigned index
 *          and return the index.
 * PARA:    vsEventName:    Event name
 *          vEventSub: Event Sub (H_EVENT/R_EVENT/A_EVENT/L_EVENT),
 *          vEventType: Controllable? (CON_EVENT, UNCON_EVENT)
```

```
 * RETURN:  >0: event index (odd: controllable  even: uncontrollable)
 *          0:  error
 * ACCESS:  public
 */
unsigned short CSub::AddSubEvent(const string & vsEventName,
                                 const EVENTSUB vEventSub,
                                 const EVENTTYPE vEventType)
{
    LOCALEVENTS::const_iterator citer;
    citer = m_SubEventsMap.find(vsEventName);
    if (citer != m_SubEventsMap.end())  //the event exists, return its index
        return citer->second;
    else  //the event does not exist, generate a new index.
    {
        if (vEventType == CON_EVENT)
        {
            m_usiMaxCon[vEventSub] += 2;
            m_SubEventsMap[vsEventName] = m_usiMaxCon[vEventSub];
            return m_usiMaxCon[vEventSub];
        }
        else
        {
            m_usiMaxUnCon[vEventSub] += 2;
            m_SubEventsMap[vsEventName] = m_usiMaxUnCon[vEventSub];
            return m_usiMaxUnCon[vEventSub];
        }
    }
    return 0;
}
```

```
/*************************************************************************
  FILE:  Sub1.cpp
  DESCR: Reordering the DES in the high-level or one low-level.
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*************************************************************************/
#include <cassert>
#include <vector>
#include "DES.h"
#include "pubfunc.h"
#include <cstdlib>
#include "Sub.h"
#include "Project.h"
/**
 * DESCR:   Main DES Reorder procedure
 * PARA:    None
 * RETURN:  0
 * ACCESS:  public
 */
int CSub::DESReorder()
{
    int iNumofDES = this->GetNumofDES();
    //compute the marix storing number of shared events between every two DES
    m_piCrossMatrix = new (int *)[iNumofDES];
    for (int i = 0; i < iNumofDES; i++)
        m_piCrossMatrix[i] = new int[iNumofDES];
    for (int i = 0; i < iNumofDES; i++)
        for (int j = 0; j < iNumofDES; j++)
            m_piCrossMatrix[i][j] =
                        NumofSharedEvents(m_pDESArr[i]->GetEventsArr(),
                                          m_pDESArr[i]->GetNumofEvents(),
                                          m_pDESArr[j]->GetEventsArr(),
                                          m_pDESArr[j]->GetNumofEvents());
    //Generate an initial order
    InitialDESOrder();
    UpdatePos();
    //Algorithm with force
    DESReorder_Force();
    UpdatePos();
    //sifting algorithm
    DESReorder_Sift();
    UpdatePos();
    //clear memory
    for (int i = 0; i < iNumofDES; i++)
    {
        delete[] m_piCrossMatrix[i];
        m_piCrossMatrix[i] = NULL;
    }
    delete[] m_piCrossMatrix;
    m_piCrossMatrix = NULL;
    //Order m_pDESArr according to the order of m_piDESOrderArr.
    CDES **pDESTmp = NULL;
    pDESTmp = new (CDES *)[this->GetNumofDES()];
    for (int i = 0; i < this->GetNumofDES(); i++)
    {
        pDESTmp[i] = m_pDESArr[m_piDESOrderArr[i]];
    }
    for (int i = 0; i < this->GetNumofDES(); i++)
    {
        m_pDESArr[i] = pDESTmp[i];
    }
    delete[] pDESTmp;
    return 0;
}
/*
 * DESCR:   Using sifting algorithm to reorder DES
 * PARA:    None
 * RETURN:  0
 * ACCESS:  private
 */
int CSub::DESReorder_Sift()
{
    int iNumofDES = this->GetNumofDES();
    bool bChanged = false;
    double dMinCross = 0.0;
    double dCurCross = 0.0;
    int *piCurOpt = new int[iNumofDES];
    int *piInit = new int[iNumofDES];
    int iTemp = 0;
```

```
int iCur = 0;
int iCount = 0;
double dOldCross = 0.0;
double dInitCross = 0.0;
double dSwapCross = 0.0;
//initialize optimal des order and loop initial order;
for (int j = 0; j < iNumofDES; j++)
{
    piCurOpt[j] = m_piDESOrderArr[j];
    piInit[j] = m_piDESOrderArr[j];
}
//initialize cross over value
dMinCross = TotalCross_Sift(0, 0, 0, 0);
dOldCross = dMinCross;
dInitCross = dMinCross;
//Initialize m_piDESPosArr
UpdatePos();
//Optimize the DES order
do
{
    iCount++;
    bChanged = false;
    for (int iDES = 0; iDES < iNumofDES; iDES++)
    {
        iCur = m_piDESPosArr[iDES];
        //move backward
        for (int i = iCur; i < iNumofDES - 1; i++)
        {
            //compute dSwapCross
            dSwapCross = TotalCross_Sift(0, 0, i, 1);
            //swap i, i+1
            iTemp = m_piDESOrderArr[i + 1];
            m_piDESOrderArr[i + 1] = m_piDESOrderArr[i];
            m_piDESOrderArr[i] = iTemp;
            //test if current order is better
            dCurCross = TotalCross_Sift(dOldCross, dSwapCross, i, 2);
            dOldCross = dCurCross;
            if (dCurCross - dMinCross < 0)
            {
                bChanged = true;
                dMinCross = dCurCross;
                for (int j = 0; j < iNumofDES; j++)
                    piCurOpt[j] = m_piDESOrderArr[j];
            }
        }
        //move forward
        for (int j = 0; j < iNumofDES; j++)
            m_piDESOrderArr[j] = piInit[j];
        dOldCross = dInitCross;
        for (int i = iCur; i > 0; i--)
        {
            //compute dSwapCross
            dSwapCross = TotalCross_Sift(0, 0, i - 1, 1);
            //swap i - 1, i
            iTemp = m_piDESOrderArr[i - 1];
            m_piDESOrderArr[i - 1] = m_piDESOrderArr[i];
            m_piDESOrderArr[i] = iTemp;
            //test if current order is better
            dCurCross = TotalCross_Sift(dOldCross, dSwapCross, i - 1, 2);
            dOldCross = dCurCross;
            if (dCurCross - dMinCross < 0)
            {
                bChanged = true;
                dMinCross = dCurCross;
                for (int j = 0; j < iNumofDES; j++)
                    piCurOpt[j] = m_piDESOrderArr[j];
            }
        }
        dInitCross = dMinCross;
        dOldCross = dMinCross;
        if (bChanged)
        {
            for (int j = 0; j < iNumofDES; j++)
            {
                m_piDESOrderArr[j] = piCurOpt[j];
                piInit[j] = m_piDESOrderArr[j];
            }
            UpdatePos();
```

```
            }
            else
            {
                for (int j = 0; j < iNumofDES; j++)
                    m_piDESOrderArr[j] = piInit[j];
            }
        }
    }while(bChanged == true );
    delete[] piCurOpt;
    piCurOpt = NULL;
    delete[] piInit;
    piInit = NULL;
    return 0;
}
/*
 * DESCR:   Compute total cross for sifting algorithm
 * PARA:    dOldCross:  old cross value
 *          dSwapCross: cross changed due to swapping
 *          iCur: current position
 *          iFlag: 0: completey compute total cross value
 *                 1: compute total cross based on the old cross and swapped DES
 *                    (much faster)
 * RETURN:  new cross value
 * ACCESS:  private
 */
double CSub::TotalCross_Sift(double dOldCross, double dSwapCross,
                             int iCur, int iFlag)
{
    double dCross = 0;
    if (iFlag == 0) //completely compute the cross
    {
        for (int i = 0; i < this->GetNumofDES(); i++)
        {
            for (int j = i + 2; j < this->GetNumofDES(); j++)
                dCross += cross(i, j);
        }
    }
    else if (iFlag == 1) //only compute iCur, iCur + 1
    {
        //iCur
        for (int i = 0; i < iCur - 1; i++)
            dCross += cross(i, iCur);
        for (int i = iCur + 2; i < this->GetNumofDES(); i++)
            dCross += cross(iCur, i);
        //iCur + 1
        for (int i = 0; i < (iCur + 1) - 1; i++)
            dCross += cross(i, iCur + 1);
        for (int i = (iCur + 1) + 2; i < this->GetNumofDES(); i++)
            dCross += cross(iCur + 1, i);
    }
    else //update
    {
        //iCur
        for (int i = 0; i < iCur - 1; i++)
            dCross += cross(i, iCur);
        for (int i = iCur + 2; i < this->GetNumofDES(); i++)
            dCross += cross(iCur, i);
        //iCur + 1
        for (int i = 0; i < (iCur + 1) - 1; i++)
            dCross += cross(i, iCur + 1);
        for (int i = (iCur + 1) + 2; i < this->GetNumofDES(); i++)
            dCross += cross(iCur + 1, i);
        dCross = dOldCross - dSwapCross + dCross;
    }
    return dCross;
}
/*
 * DESCR:   Compute the cross for DES i and DES j
 * PARA:    i,j: DES position index,
 * RETURN:  the cross for DES i and DES j
 * ACCESS:  private
 */
double CSub::cross(int i, int j)
{
    return sqrt((double)(m_piCrossMatrix[m_piDESOrderArr[i]]
                                        [m_piDESOrderArr[j]]) * (j - i - 1));
}
/*
 * DESCR:   Initialize a DES order for the sifting reorder algorithm
```

```
 *          (some ideas are from Zhonghua Zhong's STCT)
 * PARA:    None
 * RETURN:  0
 * ACCESS:  private
 */
int CSub::DESReorder_Force()
{
    int iNumofDES = this->GetNumofDES();
    int iCount = 0;
    //Optimize the DES order
    bool bChanged = false;
    double dMinCross = TotalCross_Force();
    double dCurCross = 0.0;
    do
    {
        iCount++;
        bChanged = false;
        int iOptPos = 0;
        int iDES = 0;
        for (iDES = 0; iDES < iNumofDES; iDES++)
        {
            int iPrePos = 0;
            int iNextPos = iNumofDES - 1;
            int iPos = m_piDESPosArr[iDES];
            iOptPos = iPos;
            int iNewPos = 0;
            while (true)
            {
                double dForce = Force(iPos);
                if (dForce < -0.05)
                {
                    iNextPos = iPos;
                    iNewPos = iPos - (((iPos - iPrePos) % 2 == 0)?
                        ((iPos - iPrePos) / 2):((iPos - iPrePos) / 2 + 1));
                    if (iNewPos <= iPrePos)
                        break;
                    InsertDES(iPos, iNewPos);
                    UpdatePos();
                    iPos = iNewPos;
                    dCurCross = TotalCross_Force();
                    if (dCurCross < dMinCross - 0.05)
                    {
                        iOptPos = iPos;
                        dMinCross = dCurCross;
                        bChanged = true;
                    }
                }
                else if (dForce > 0.05)
                {
                    iPrePos = iPos;
                    iNewPos = iPos + (((iNextPos - iPos) % 2 == 0)?
                        ((iNextPos - iPos) / 2):((iNextPos - iPos) / 2 + 1));
                    if (iNextPos <= iNewPos)
                        break;
                    InsertDES(iPos, iNewPos);
                    UpdatePos();
                    iPos = iNewPos;
                    dCurCross = TotalCross_Force();
                    if (dCurCross < dMinCross - 0.05)
                    {
                        iOptPos = iPos;
                        dMinCross = dCurCross;
                        bChanged = true;
                    }
                }
                else
                    break;
            }
            InsertDES(m_piDESPosArr[iDES], iOptPos);
            UpdatePos();
        }
    }while(bChanged == true);
    return 0;
}
/*
 * DESCR:   Update DES position in array m_piDESPosArr according the new order
 * PARA:    None
 * RETURN:  None
 * ACCESS:  private
 */
void CSub::UpdatePos()
{
```

```
    for (int i = 0 ; i < this->GetNumofDES(); i++)
        m_piDESPosArr[m_piDESOrderArr[i]] = i;
    return;
}
/*
 * DESCR:   Swap variables in m_piDESOrderArr for DESReorder_Force()
 * PARA:    iCur: current variable position
 *          iPos: destinate variable position
 * RETURN:  None
 * ACCESS:  private
 */
void CSub::InsertDES(int iCur, int iPos)
{
    int iDES = m_piDESOrderArr[iCur];
    if (iCur < iPos)
    {
        for (int i = iCur + 1; i <= iPos; i++)
            m_piDESOrderArr[i - 1] = m_piDESOrderArr[i];
        m_piDESOrderArr[iPos] = iDES;
    }
    else if (iCur > iPos)
    {
        for (int i = iCur - 1; i >= iPos; i--)
            m_piDESOrderArr[i + 1] = m_piDESOrderArr[i];
        m_piDESOrderArr[iPos] = iDES;
    }
    return;
}
/*
 * DESCR:   Compute total cross for DESReorder_Force()
 * PARA:    None
 * RETURN:  total cross
 * ACCESS:  private
 */
double CSub::TotalCross_Force()
{
    double dCross = 0;
    for (int i = 0; i < this->GetNumofDES(); i++)
    {
        for (int j = i + 2; j < this->GetNumofDES(); j++)
            dCross += cross(i, j);
    }
    return dCross;
}
/*
 * DESCR:   Decide to move DES_i left or right. (< 0 : move left; >0 move right)
 *          for DESReorder_Force()
 * PARA:    i: position in m_piDESOrderArr
 * RETURN:  returned force
 * ACCESS:  private
 */
double CSub::Force(int i)
{
    double dForce = 0;
    for (int j = 0; j < i - 1; j++)
        dForce += sqrt((double)m_piCrossMatrix[m_piDESOrderArr[i]]
                                              [m_piDESOrderArr[j]] * (j - i + 1));
    for (int j = i + 2; j < this->GetNumofDES(); j++)
        dForce += sqrt((double)m_piCrossMatrix[m_piDESOrderArr[i]]
                                              [m_piDESOrderArr[j]] * (j - i - 1));
    return dForce;
}
/*
 * DESCR:   Initialize a DES order
 * PARA:    None
 * RETURN:  0
 * ACCESS:  private
 */
int CSub::InitialDESOrder()
{
    int i = 0;
    int j = 0;
    int k = 0;
    int iNumofDES = this->GetNumofDES();
    //There is no DES at all
    if (iNumofDES <= 0)
        return 0;
    //Only one DES
    m_piDESOrderArr[0] = 0;
    if (iNumofDES <= 1)
        return 0;
```

```
    int iPos = 0;
    double dLeftCross = 0;
    double dRightCross = 0;
    double dNewCross = 0;
    double dOldCross = 0;
    vector<int> vecDESOrder;
    vector<int> vecShared;
    //two or more DES
    vecDESOrder.push_back(0);
    vecDESOrder.push_back(1);
    for (i = 2; i < iNumofDES; i++)
    {
        vecShared.clear();
        for (j = 0; j < i; j++)
            vecShared.push_back(m_piCrossMatrix[i][vecDESOrder[j]]);
        iPos = i;
        dOldCross = MAX_DOUBLE;
        for (j = i; j >= 0; j--)
        {
            dLeftCross = 0;
            dRightCross = 0;
            for (k = 0; k < j; k++)
            {
                dLeftCross += vecShared[k] * (j - k - 1);
            }
            for (k = j; k < i; k++)
            {
                dRightCross += vecShared[k] * (k - j);
            }
            dNewCross = dLeftCross + dRightCross;
            if (dNewCross == 0)
            {
                iPos = j;
                break;
            }
            else
            {
                if (dNewCross < dOldCross - 0.05)
                {
                    dOldCross = dNewCross;
                    iPos = j;
                }
            }
        }
        if (iPos == 0)
            vecDESOrder.insert(vecDESOrder.begin(), i);
        else if (iPos == i)
            vecDESOrder.push_back(i);
        else
        {
            vector<int>::iterator itr = vecDESOrder.begin();
            itr += iPos;
            vecDESOrder.insert(itr, i);
        }
    }
    assert((int)vecDESOrder.size() == this->GetNumofDES());
    for (i = 0; i < (int)vecDESOrder.size(); i++)
    {
        m_piDESOrderArr[i] = vecDESOrder[i];
    }
    return 0;
}
```

```
/*************************************************************************
  FILE:  Sub2.cpp
  DESCR: Save synthesized automata-based supervisor, or synthesized local
         control predicates, or the syn-product of a verified system.
         For the high-level or one low-level.
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*************************************************************************/
#include "LowSub.h"
#include "HighSub.h"
#include "Sub.h"
#include "Sub.h"
#include "pubfunc.h"
#include "type.h"
#include "fdd.h"
#include "Project.h"
#include "errmsg.h"
#include <string>
#include <cassert>
using namespace std;
extern CProject *pPrj;
/*
 * DESCR:   Save synthesized automata-based supervisor, or synthesized local
 *          control predicates, or the syn-product of a verified system.
 * PARA:    bddReach:   BDD including all the reachable states (input)
 *          savetype: which format to save (see type.h) (input)
 *          savepath: where to save the result (input)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  protected
 */
int CSub::SaveSuper(const bdd & bddReach, const HISC_SAVESUPERTYPE savetype,
                    const string & savepath)
{
    string sEventName;
    bdd bddController = bddfalse;
    bdd bddSimController = bddfalse;
    string sCurFile;
    ofstream fout;
    string sLine;
    string sSavePath;
    char scFileName[MAX_PATH];
    STATES statesMap;
    string sInitState;
    //save bdd variable file
    try
    {
        sSavePath = savepath;
        if (sSavePath.length() > 0)
        {
            if (sSavePath[sSavePath.length() - 1] != '/')
                sSavePath += "/";
        }
        //bdd Controller
        if (savetype == HISC_SAVESUPER_BDD || savetype == HISC_SAVESUPER_BOTH)
        {
            sCurFile = sSavePath + m_sSubName + "_var.txt";
            fout.open(sCurFile.data());
            if (!fout)
                throw -1;
            for (int i = 0; i < this->GetNumofDES(); i++)
            {
                sLine = m_pDESArr[i]->GetDESName() + " : ";
                int ivarnum = fdd_varnum(i * 2);
                for (int j = 0; j < ivarnum; j++)
                {
                    sLine += str_itos(fdd_vars(i * 2)[j]);
                    sLine += " ";
                }
                fout << sLine << endl;
                sLine.clear();
                for (int j = 0; j < m_pDESArr[i]->GetNumofStates(); j++)
                {
                    sLine = str_itos(j);
                    sLine += "  ";
                    sLine += m_pDESArr[i]->GetStateName(j);
                    fout << sLine << endl;
                }
                fout << endl;
            }
            fout.close();
        }
```

```
//automata file
if (savetype == HISC_SAVESUPER_AUTOMATA ||
                                savetype == HISC_SAVESUPER_BOTH)
{
    sCurFile = sSavePath + m_sSubName + "_sup.txt";
    fout.open(sCurFile.data());
    if (!fout)
        throw -1;
    //print the DES order in the state vector
    fout << "# OUTPUT ORDER: ";
    for (int i = 0; i < this->GetNumofDES(); i++)
    {
        fout << m_pDESArr[m_piDESPosArr[i]]->GetDESName() << " ";
    }
    fout << endl << endl;
    //put init state into statesMap
    //the index for initial state should be 0
    //(mainly for convieniently transforming it to TCT format later)
    if (PrintStateSet(fout, m_bddInit,  statesMap, 0) < 0)
        throw -1;
    STATES::const_iterator csmi = statesMap.begin();
    if (csmi != statesMap.end())
        sInitState = csmi->first;
    else
        sInitState.clear();
    //print [States] Field
    fout << "[States]" << endl;
    if (PrintStateSet(fout, bddReach, statesMap, 1) < 0)
        throw -1;
    //print [InitState] Field
    fout << endl << "[InitState]" << endl;
    if (!sInitState.empty())
        fout << "0 #" << sInitState << endl;
    //print [MarkingStates] Field
    fout << endl << "[MarkingStates]" << endl;
    if (PrintStateSet(fout, m_bddMarking, statesMap, 2) < 0)
        throw -1;
    //print [Events] Field
    fout << endl << "[Events]" << endl;
    if (PrintEvents(fout) < 0)
        throw -1;
    //print [Transtions] Fields
    fout << endl << "[Transitions]" << endl;
}
for (int k = 0; k < 4; k++)
{
    //UnControllable events
    for (unsigned short usi = 2; usi <= m_usiMaxUnCon[k]; usi += 2)
    {
        sEventName = this->SearchEventName((EVENTSUB)k, usi);
        //\Gamma(N_\sigma \and C)
        bddController = bdd_relprod(
            m_pbdd_UnConTrans[k][(usi - 2) / 2],
            bdd_replace(bddReach, m_pPair_UnCon[k][(usi - 2) / 2]),
            m_pbdd_UnConVarPrim[k][(usi - 2) / 2]) & bddReach;
        //print text transitions
        if (savetype == HISC_SAVESUPER_AUTOMATA ||
                                savetype == HISC_SAVESUPER_BOTH)
        {
            if (PrintTextTrans(fout, bddController, (EVENTSUB)k, usi,
                                bddReach, sEventName, statesMap) < 0)
                throw -1;
        }
    }
    //Controllable events
    for (unsigned short usi = 1;
                usi < (unsigned short) (m_usiMaxCon[k] + 1); usi += 2)
    {
        sEventName = this->SearchEventName((EVENTSUB)k, usi);
        bddController = bdd_relprod(
            m_pbdd_ConTrans[k][(usi - 1) / 2],
            bdd_replace(bddReach, m_pPair_Con[k][(usi - 1) / 2]),
            m_pbdd_ConVarPrim[k][(usi - 1) / 2]) & bddReach;
        bddSimController = SimplifyController(bddController,
                                                (EVENTSUB)k, usi);
        //high level: H, R events Low level: A, L events.
```

```
            if ((m_iSubIndex == 0 && k <= 1) || (m_iSubIndex != 0 && k > 1))
            {
                //print bdd controller
                if (savetype == HISC_SAVESUPER_BDD ||
                                        savetype == HISC_SAVESUPER_BOTH)
                {
                    //not simplified
                    sCurFile = sSavePath + m_sSubName + "_" +
                                                    sEventName + ".bdd";
                    strcpy(scFileName, sCurFile.data());
                    if (bdd_fnsave(scFileName, bddController) != 0)
                        throw -1;
                    sCurFile = sSavePath + m_sSubName + "_" +
                                                    sEventName + ".dot";
                    strcpy(scFileName, sCurFile.data());
                    if (bdd_fnprintdot(scFileName, bddController) != 0)
                        throw -1;
                    //triple-prime simplified
                    sCurFile = sSavePath + m_sSubName + "_" +
                                                    sEventName + "_sim.bdd";
                    strcpy(scFileName, sCurFile.data());
                    if (bdd_fnsave(scFileName, bddSimController) != 0)
                        throw -1;
                    sCurFile = sSavePath + m_sSubName + "_" +
                                                    sEventName + "_sim.dot";
                    strcpy(scFileName, sCurFile.data());
                    if (bdd_fnprintdot(scFileName, bddSimController) != 0)
                        throw -1;
                    //prime simplified
                    bddSimController = bdd_simplify(bddController,
                                                            m_bddSuper);
                    sCurFile = sSavePath + m_sSubName + "_" +
                                                    sEventName + "_re.bdd";
                    strcpy(scFileName, sCurFile.data());
                    if (bdd_fnsave(scFileName, bddSimController) != 0)
                        throw -1;
                    sCurFile = sSavePath + m_sSubName + "_" +
                                                    sEventName + "_re.dot";
                    strcpy(scFileName, sCurFile.data());
                    if (bdd_fnprintdot(scFileName, bddSimController) != 0)
                        throw -1;
                }
            }
            //print text transitions
            if (savetype == HISC_SAVESUPER_AUTOMATA ||
                                        savetype == HISC_SAVESUPER_BOTH)
            {
                if (PrintTextTrans(fout, bddController, (EVENTSUB)k, usi,
                                    bddReach, sEventName, statesMap) < 0)
                    throw -1;
            }
        }
    }
    if (savetype == HISC_SAVESUPER_AUTOMATA ||
                                    savetype == HISC_SAVESUPER_BOTH)
        fout.close();
}
catch(...)
{
    if (fout) fout.close();
    pPrj->SetErr("Unable to save the bdd controller or .",
                    HISC_BAD_SAVESUPER);
    return -1;
}
return 0;
}
/*
 * DESCR:   Print all the state vectors using state names
 * PARA:    fout: file stream (input)
 *          bddStateSet: BDD respresentation of the state set (input)
 *          statesMap: A STL Map to store (state name vector (key), integer
 *                     index)
 *          viSetFlat: 0: Initial state  1: All states 2: Marking States (input)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  protected
 */
int CSub::PrintStateSet(ofstream & fout, const bdd & bddStateSet,
                    STATES & statesMap, int viSetFlag)
{
```

```
int *statevec = NULL;
int iStateIndex = 0;
try
{
    string sLine;
    bdd bddTemp = bddfalse;
    bdd bddNormStateSet = bddtrue;
    string sInitState;
    bool bInitState = false;
    //restrict the prime variable to 0
    for (int i = 0; i < this->GetNumofDES(); i++)
        bddNormStateSet &= fdd_ithvar(i * 2 + 1, 0);
    bddNormStateSet &= bddStateSet;
    //save number of states
    if (viSetFlag != 0)
        fout << bdd_satcount(bddNormStateSet) << endl;
    //Initial state
    STATES::const_iterator csmi = statesMap.begin();
    if (csmi != statesMap.end())
        sInitState = csmi->first;
    else
        sInitState.clear();
    //print all the vectors
    statevec = fdd_scanallvar(bddNormStateSet);
    while ( statevec!= NULL)
    {
        sLine.clear();
        sLine = "<";
        for (int i = 0; i < this->GetNumofDES(); i++)
        {
            sLine += m_pDESArr[m_piDESPosArr[i]]->GetStateName(
                                statevec[m_piDESPosArr[i] * 2]) + ",";
        }
        sLine = sLine.substr(0, sLine.length() - 1);
        sLine += ">";
        iStateIndex++;
        //state index for initial state should be 0
        if (viSetFlag == 0)
        {
            iStateIndex = 0;
            statesMap[sLine] = iStateIndex;
        }
        else
        {
            //for marking states, should show the corresponding state index
            if (viSetFlag == 2)
                fout << statesMap[sLine] << " #" << sLine << endl;
            else  //all the states
            {
                if (bInitState)  //initial state alredy been printed
                {
                    statesMap[sLine] = iStateIndex;
                    fout << iStateIndex << " #" << sLine << endl;
                }
                else
                {
                    if (sLine != sInitState)
                    {
                        statesMap[sLine] = iStateIndex;
                        fout << iStateIndex << " #" << sLine << endl;
                    }
                    else
                    {
                        iStateIndex--;
                        bInitState = true;
                        fout << "0" << " #" << sLine << endl;
                    }
                }
            }
        }
        //remove the outputed state
        bddTemp = bddtrue;
        for (int i = 0; i < this->GetNumofDES(); i++)
            bddTemp &= fdd_ithvar(i * 2, statevec[i * 2]);
        bddNormStateSet = bddNormStateSet - bddTemp;
        free(statevec);
        statevec = NULL;
        statevec = fdd_scanallvar(bddNormStateSet);
    }
}
catch(...)
```

```
    {
        delete[] statevec;
        statevec = NULL;
        return -1;
    }
    return 0;
}
/*
 * DESCR:   Print all events from the pPrj->m_InvAllEventsMap
 * PARA:    fout: file stream (input)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  protected
 */
int CSub::PrintEvents(ofstream & fout)
{
    char cSub = '\0';
    char cCon = '\0';
    string sLine;
    try
    {
        INVEVENTS::const_iterator ci = pPrj->GetInvAllEventsMap().begin();
        for (; ci != pPrj->GetInvAllEventsMap().end(); ++ci)
        {
            if (((ci->first & 0x0FFF0000) >> 16 == m_iSubIndex) ||
               (((ci->first >> 28) == R_EVENT ||
               (ci->first >> 28) == A_EVENT) && m_iSubIndex == 0))
            {
                cSub = SubValueToLetter((EVENTSUB)(ci->first >> 28))[0];
                cCon = ci->first % 2 == 0 ? 'N':'Y';
                sLine = ci->second + "\t\t";
                sLine += cCon;
                sLine += "\t\t";
                sLine += cSub;
                fout << sLine << endl;
            }
        }
    }
    catch (...)
    {
        return -1;
    }
    return 0;
}
/*
 * DESCR:   Print all the transitions one by one
 * PARA:    fout: file stream (input)
 *          bddController: not simplified bdd control predicate for sEventName
 *          EventSub: 'H'/'R'/'A'/'L'
 *          usiLocalIndex: local index (in this sub)
 *          bddReach: BDD respresentation of reachable states in
 *                    synthesized automata-based supervisor or syn-product of
 *                    the verified system.
 *          sEventName: Event Name
 *          statesMap: state name and index map (index is for the output file)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  protected
 */
int CSub::PrintTextTrans(ofstream & fout, bdd & bddController,
                         EVENTSUB EventSub, unsigned short usiLocalIndex,
                         const bdd & bddReach, string sEventName,
                         STATES & statesMap)
{
    int *statevec1 = NULL;
    int *statevec2 = NULL;
    try
    {
        string sExit;
        string sEnt;
        bdd bddTemp = bddfalse;
        bdd bddNext = bddfalse;
        //extract each state from bddController
        statevec1 = fdd_scanallvar(bddController);
        while ( statevec1!= NULL)
        {
            sExit.clear();
            sExit = "<";
            for (int i = 0; i < this->GetNumofDES(); i++)
                sExit += m_pDESArr[m_piDESPosArr[i]]->GetStateName(
                                 statevec1[m_piDESPosArr[i] * 2]) + ",";
            sExit = sExit.substr(0, sExit.length() - 1);
            sExit += ">";
```

```
            bddTemp = bddtrue;
            for (int i = 0; i < this->GetNumofDES(); i++)
                bddTemp &= fdd_ithvar(i * 2, statevec1[i * 2]);
            bddController = bddController - bddTemp;
            free(statevec1);
            statevec1 = NULL;
            statevec1 = fdd_scanallvar(bddController);
            //Get the target state
            if (usiLocalIndex % 2 == 0)
                bddNext =
                    bdd_replace(
                        bdd_relprod(
                           m_pbdd_UnConTrans[EventSub][(usiLocalIndex - 2) / 2],
                           bddTemp,
                           m_pbdd_UnConVar[EventSub][(usiLocalIndex - 2) / 2]),
                        m_pPair_UnConPrim[EventSub][(usiLocalIndex - 2) / 2]) &
                    bddReach;
            else
                bddNext =
                    bdd_replace(
                        bdd_relprod(
                            m_pbdd_ConTrans[EventSub][(usiLocalIndex - 1) / 2],
                            bddTemp,
                            m_pbdd_ConVar[EventSub][(usiLocalIndex - 1) / 2]),
                        m_pPair_ConPrim[EventSub][(usiLocalIndex - 1) / 2]) &
                    bddReach;
            statevec2 = fdd_scanallvar(bddNext);
            if (statevec2 == NULL)
                throw -1;
            sEnt = "<";
            for (int i = 0; i < this->GetNumofDES(); i++)
                sEnt += m_pDESArr[m_piDESPosArr[i]]->GetStateName(
                                        statevec2[m_piDESPosArr[i] * 2]) + ",";
            sEnt = sEnt.substr(0, sEnt.length() - 1);
            sEnt += ">";
            free(statevec2);
            statevec2 = NULL;
            //print the transition
            fout << statesMap[sExit] << " <" << sEventName << "> " <<
                statesMap[sEnt] << endl;
        }
    }
    catch(...)
    {
        free(statevec1);
        statevec1 = NULL;
        free(statevec2);
        statevec2 = NULL;
        return -1;
    }
    return 0;
}
/*
 * DESCR:   Compute triple-prime simplified BDD control predicate for an event
 * PARA:    fout: file stream (input)
 *          bddController: BDD control predicate for event usiIndex
 *          EventSub: 'H'/'R'/'A'/'L'
 *          usiIndex: local index (in this sub)
 * RETURN:  triple-prime simplified BDD control predicate
 * ACCESS:  protected
 */
bdd CSub::SimplifyController(const bdd & bddController, EVENTSUB EventSub,
                             const unsigned short usiIndex)
{
    //event should be controllable
    assert(usiIndex % 2 == 1);
    bdd bddElig = bddfalse;
    bdd bddSpecElig = bddfalse;
    //\dHs'
    bddElig = bdd_exist(m_pbdd_ConTrans[EventSub][(usiIndex - 1) / 2],
                        m_pbdd_ConVarPrim[EventSub][(usiIndex - 1) / 2]);
    //spec part
    bddSpecElig = bdd_exist(bddElig,
                        m_pbdd_ConPhysicVar[EventSub][(usiIndex - 1) / 2]);
    return bddSpecElig & bdd_simplify(bddController, m_bddSuper & bddElig);
}
```

```
/*************************************************************************
  FILE:  HighSub.h
  DESCR: Header file for HighSub*.cpp (High-level processing files)
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*************************************************************************/
#ifndef _HSUB_H_
#define _HSUB_H_
#include <string>
#include "Sub.h"
#include "type.h"
#include <fdd.h>
using namespace std;
class CHighSub:public CSub
{
public:
    CHighSub(const string &  vsHighFile, int viSubIndex);
    virtual ~CHighSub();
public:
    virtual int PrintSub(ofstream& fout);
    virtual int PrintSubAll(ofstream & fout);
    virtual string SearchEventName(EVENTSUB EventSub,
                                           unsigned short usiLocalIndex);
    virtual int LoadSub();
    virtual int SynSuper(const HISC_COMPUTEMETHOD computemethod,
                         HISC_SUPERINFO &superinfo,
                         const HISC_SAVESUPERTYPE savetype,
                         const string& savepath);
    virtual int VeriSub(const HISC_TRACETYPE showtrace,
                        HISC_SUPERINFO & superinfo,
                        const HISC_SAVEPRODUCTTYPE savetype,
                        const string& savepath);
private:
    virtual int MakeBdd();
    int GenConBad(bdd &bddConBad);
    int VeriConBad(bdd &bddConBad, const bdd &bddReach, string & vsErr);
    int GenP3Bad(bdd &bddP3Bad);
    int VeriP3Bad(bdd &bddP3Bad, const bdd &bddReach, string & vsErr);
    int supcp(bdd & bddP);
    bdd cr(const bdd & bddPStart, const bdd & bddP, const int viEventSub,
           int & iErr);
    bdd r(const bdd &bddP, int &iErr);
    virtual int InitBddFields();
    virtual int ClearBddFields();
    void BadStateInfo(const bdd& bddBad, const int viErrCode,
                      const HISC_TRACETYPE showtrace,
                      const string &vsExtraInfo = "");
private:
    int *m_piUArr[2]; //Uncontrollable event index array, 0:Request  1:Answer
    int *m_piCArr[2]; //Controllable event index array, 0:Request  1:Answer
    bdd *m_pbdd_IVar; //Interface normal variables
    bdd *m_pbdd_IVarPrim; //Interface Prime Variables
    bdd *m_pbdd_UnConATrans; //transition predicate for answer events
                             //(for interface alone).
    bdd *m_pbdd_ConATrans; //transition predicate for answer events
                           //(only for interface alone).
};
#endif //_HSUB_H_
```

```
/**************************************************************************
  FILE:  HighSub.cpp
  DESCR: Reading high-level DES files and initialze all the BDDs
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
**************************************************************************/
#include "Sub.h"
#include "HighSub.h"
#include "LowSub.h"
#include <string>
#include "errmsg.h"
#include "type.h"
#include <fdd.h>
#include "DES.h"
#include "Project.h"
#include <fstream>
#include "pubfunc.h"
using namespace std;
extern CProject *pPrj;
/**
 * DESCR:   Constructor
 * PARA:    vsHighFile: subsystem file name with path (.sub)(input)
 *          viSubIndex: subsystem index (high: 0, low: 1,2,...)(input)
 * RETURN:  None
 * ACCESS:  public
 */
CHighSub::CHighSub(const string & vsHighFile, int viSubIndex):
CSub(vsHighFile, viSubIndex)
{
    m_iNumofIntfs = pPrj->GetNumofLows();
    m_piUArr[0] = NULL;
    m_piUArr[1] = NULL;
    m_piCArr[0] = NULL;
    m_piCArr[1] = NULL;
    InitBddFields();
}
/**
 * DESCR:   Destructor
 * PARA:    None
 * RETURN:  None
 * ACCESS:  public
 */
CHighSub::~CHighSub()
{
    delete[] m_piUArr[0];
    m_piUArr[0] = NULL;
    delete[] m_piUArr[1];
    m_piUArr[1] = NULL;
    delete[] m_piCArr[0];
    m_piCArr[0] = NULL;
    delete[] m_piCArr[1];
    m_piCArr[1] = NULL;
    ClearBddFields();
}
/*
 * DESCR:   Initialize BDD related data members (only those in HighSub.h)
 * PARA:    None
 * RETURN:  0
 * ACCESS:  private
 */
int CHighSub::InitBddFields()
{
    m_pbdd_IVar = NULL;
    m_pbdd_IVarPrim = NULL;
    m_pbdd_UnConATrans = NULL;
    m_pbdd_ConATrans = NULL;
    return 0;
}
/*
 * DESCR:   Release memory for BDD related data members(only those in Highsub.h)
 * PARA:    None
 * RETURN:  0
 * ACCESS:  private
 */
int CHighSub::ClearBddFields()
{
    delete[] m_pbdd_IVar;
    m_pbdd_IVar = NULL;
    delete[] m_pbdd_IVarPrim;
    m_pbdd_IVarPrim = NULL;
    delete[] m_pbdd_UnConATrans;
```

```
    m_pbdd_UnConATrans = NULL;
    delete[] m_pbdd_ConATrans;
    m_pbdd_ConATrans = NULL;
    return 0;
}
/**
 * DESCR:   Load high-level
 * PARA:    None
 * RETURN:  0 sucess <0 fail;
 * ACCESS:  public
 */
int CHighSub::LoadSub()
{
    ifstream fin;
    int iRet = 0;
    CDES *pDES = NULL;
    try
    {
        m_sSubFile = str_trim(m_sSubFile);
        if (m_sSubFile.length() <= 4)
        {
            pPrj->SetErr("Invalid file name: " + m_sSubFile,HISC_BAD_HIGH_FILE);
            throw -1;
        }
        if (m_sSubFile.substr(m_sSubFile.length() - 4) != ".sub")
        {
            pPrj->SetErr("Invalid file name: " + m_sSubFile,HISC_BAD_HIGH_FILE);
            throw -1;
        }
        fin.open(m_sSubFile.data(), ifstream::in);
        if (!fin) //unable to find high sub file
        {
            pPrj->SetErr("Unable to open file: " + m_sSubFile,
                         HISC_BAD_HIGH_FILE);
            throw -1;
        }
        m_sSubName = GetNameFromFile(m_sSubFile);
        char scBuf[MAX_LINE_LENGTH];
        string sLine;
        int iField = -1; //0: SYSTEM 1:PLANT 2:SPEC
        char *scFieldArr[] = {"SYSTEM", "PLANT", "SPEC"};
        string sDESFile;
        int iTmp = 0;
        int iNumofPlants = 0;
        int iNumofSpecs = 0;
        while (fin.getline(scBuf,  MAX_LINE_LENGTH))
        {
            sLine = str_nocomment(scBuf);
            sLine = str_trim(sLine);
            if (sLine.empty())
                continue;
            if (sLine[0] == '[' && sLine[sLine.length() - 1] == ']')
            {
                sLine = sLine.substr(1, sLine.length() - 1);
                sLine = sLine.substr(0, sLine.length() - 1);
                sLine = str_upper(str_trim(sLine));
                iField++;
                if (iField <= 2)
                {
                    if (sLine != scFieldArr[iField])
                    {
                        pPrj->SetErr(m_sSubName +
                                     ": Field name or order is wrong!",
                                     HISC_BAD_HIGH_FORMAT);
                        throw -1;
                    }
                    if (iField == 1)
                    {
                        //Check number of Plants and apply for memory space
                        if (m_iNumofPlants + m_iNumofSpecs <= 0)
                        {
                            pPrj->SetErr(m_sSubName +
                                ": Must have at least one DES in this level.",
                                HISC_BAD_HIGH_FORMAT);
                            throw -1;
                        }
                        if (m_iNumofPlants < 0 || m_iNumofSpecs < 0)
                        {
```

```
                pPrj->SetErr(m_sSubName +
            ": Must specify the number of plant DES and spec DES.",
                        HISC_BAD_HIGH_FORMAT);
                throw -1;
            }
            m_pDESArr = new (CDES *)[this->GetNumofDES()];
            for (int i = 0; i < this->GetNumofDES(); i++)
                m_pDESArr[i] = NULL;
            //Initialize m_piDESOrderArr
            m_piDESOrderArr = new int[this->GetNumofDES()];
            for (int i = 0; i < this->GetNumofDES(); i++)
                m_piDESOrderArr[i] = i;
            //Initialize m_piDESPosArr
            m_piDESPosArr = new int[this->GetNumofDES()];
            for (int i = 0; i < this->GetNumofDES(); i++)
                m_piDESPosArr[i] = i;
        }
    }
    else
    {
        pPrj->SetErr(m_sSubName + ": Too many fields!",
                            HISC_BAD_HIGH_FORMAT);
        throw -1;
    }
}
else
{
    switch (iField)
    {
    case 0:  //[SYSTEM]
        if (!IsInteger(sLine))
        {
            pPrj->SetErr(m_sSubName + ": Number of DES is absent!",
                            HISC_BAD_HIGH_FORMAT);
            throw -1;
        }
        iTmp = atoi(sLine.data());
        if (iTmp < 0)
        {
            pPrj->SetErr(m_sSubName +
                        ": Number of DES is less than zero!",
                        HISC_BAD_HIGH_FORMAT);
            throw -1;
        }
        if (m_iNumofPlants < 0)
            m_iNumofPlants = iTmp;
        else if (m_iNumofSpecs < 0)
            m_iNumofSpecs = iTmp;
        else
        {
            pPrj->SetErr(m_sSubName +
                    ": Too many lines in SYSTEM field",
                    HISC_BAD_HIGH_FORMAT);
            throw -1;
        }
        break;
    case 1: //[PLANT]
        sDESFile = GetDESFileFromSubFile(m_sSubFile, sLine);
        pDES = new CDES(this, sDESFile, PLANT_DES);
        if (pDES->LoadDES() < 0)
            throw -1;   //here LoadDES() will generate the err msg.
        else
        {
            iNumofPlants++;
            if (iNumofPlants > m_iNumofPlants)
            {
                pPrj->SetErr(m_sSubName + ": Too many Plant DESs",
                        HISC_BAD_HIGH_FORMAT);
                throw -1;
            }
            m_pDESArr[iNumofPlants - 1] = pDES;
            pDES = NULL;
        }
        break;
    case 2: //[SPEC]
        sDESFile = GetDESFileFromSubFile(m_sSubFile, sLine);
        pDES = new CDES(this, sDESFile, SPEC_DES);
        if (pDES->LoadDES() < 0)
            throw -1;   //here LoadDES() will generate the err msg.
        else
```

```
                {
                    iNumofSpecs++;
                    if (iNumofSpecs > m_iNumofSpecs)
                    {
                        pPrj->SetErr(m_sSubName + ": Too many spec DESs",
                                HISC_BAD_HIGH_FORMAT);
                        throw -1;
                    }
                    m_pDESArr[m_iNumofPlants + iNumofSpecs - 1] = pDES;
                    pDES = NULL;
                }
                break;
            default:
                pPrj->SetErr(m_sSubName + ": Unknown error.",
                        HISC_BAD_HIGH_FORMAT);
                throw -1;
                break;
            }
        }
    } //while
    if (iNumofPlants < m_iNumofPlants)
    {
        pPrj->SetErr(m_sSubName + ": Too few plant DESs",
                        HISC_BAD_HIGH_FORMAT);
        throw -1;
    }
    if (iNumofSpecs < m_iNumofSpecs)
    {
        pPrj->SetErr(m_sSubName + ": Too few spec DESs",
                        HISC_BAD_HIGH_FORMAT);
        throw -1;
    }
    fin.close();
    //Get all the interface DES
    for (int i = 0; i < m_iNumofIntfs; i++)
    {
        m_pDESArr[m_iNumofPlants + m_iNumofSpecs + i] =
                        ((CLowSub *)pPrj->GetSub(i + 1))->GetIntfDES();
    }
    //Compute the R/A events starting index in Transtion precidate arrary
    for (int i = 0; i < 2; i++)
    {
        m_piUArr[i] = new int[m_iNumofIntfs];
        m_piCArr[i] = new int[m_iNumofIntfs];
        m_piUArr[i][0] = 0;
        m_piCArr[i][0] = 0;
    }
    for (int i = 1; i < m_iNumofIntfs; i++)
    {
        m_piUArr[0][i] = m_piUArr[0][i-1] +
                                pPrj->GetSub(i)->GetMaxUnCon(R_EVENT) / 2;
        m_piUArr[1][i] = m_piUArr[1][i-1] +
                                pPrj->GetSub(i)->GetMaxUnCon(A_EVENT) / 2;
        m_usiMaxUnCon[R_EVENT] += pPrj->GetSub(i)->GetMaxUnCon(R_EVENT);
        m_usiMaxUnCon[A_EVENT] += pPrj->GetSub(i)->GetMaxUnCon(A_EVENT);
        m_piCArr[0][i] = m_piCArr[0][i-1] +
            ((unsigned short)(pPrj->GetSub(i)->GetMaxCon(R_EVENT) + 1)) / 2;
        m_piCArr[1][i] = m_piCArr[1][i-1] +
            ((unsigned short)(pPrj->GetSub(i)->GetMaxCon(A_EVENT) + 1)) / 2;
        m_usiMaxCon[R_EVENT] +=
                (unsigned short)(pPrj->GetSub(i)->GetMaxCon(R_EVENT) + 1);
        m_usiMaxCon[A_EVENT] +=
                (unsigned short)(pPrj->GetSub(i)->GetMaxCon(A_EVENT) + 1);
    }
    m_usiMaxUnCon[R_EVENT] +=
                        pPrj->GetSub(m_iNumofIntfs)->GetMaxUnCon(R_EVENT);
    m_usiMaxUnCon[A_EVENT] +=
                        pPrj->GetSub(m_iNumofIntfs)->GetMaxUnCon(A_EVENT);
    m_usiMaxCon[R_EVENT] +=
      (unsigned short)(pPrj->GetSub(m_iNumofIntfs)->GetMaxCon(R_EVENT) + 1);
    m_usiMaxCon[A_EVENT] +=
      (unsigned short)(pPrj->GetSub(m_iNumofIntfs)->GetMaxCon(A_EVENT) + 1);
    //DES Reorder
    this->DESReorder();
}
catch (int iError)
{
    if (pDES != NULL)
```

```
        {
            delete pDES;
            pDES = NULL;
        }
        if (fin.is_open())
            fin.close();
        iRet = iError;
    }
    return iRet;
}
/*
 * DESCR:   Initialize BDD data memebers
 * PARA:    None
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
 */
int CHighSub::MakeBdd()
{
    try
    {
        //Initialize the bdd node table and cache size.
        long long lNumofStates = 1;
        for (int i = 0; i < this->GetNumofDES(); i++)
        {
            lNumofStates *= m_pDESArr[i]->GetNumofStates();
            if (lNumofStates >= MAX_INT)
                break;
        }
        if (lNumofStates <= 10000)
            bdd_init(1000, 100);
        else if (lNumofStates <= 1000000)
            bdd_init(10000, 1000);
        else if (lNumofStates <= 10000000)
            bdd_init(100000, 10000);
        else
        {
            bdd_init(2000000, 1000000);
            //bdd_setcacheratio(16);
            bdd_setmaxincrease(1000000);
        }
        giNumofBddNodes = 0;
        bdd_gbc_hook(my_bdd_gbchandler);
        //define domain variables
        int *piDomainArr = new int[2];
        for (int i = 0; i < 2 * this->GetNumofDES(); i += 2)
        {
            piDomainArr[0] = m_pDESArr[i/2]->GetNumofStates();
            piDomainArr[1] = piDomainArr[0];
            fdd_extdomain(piDomainArr, 2);
        }
        delete[] piDomainArr;
        piDomainArr = NULL;
        //compute the number of bdd variables (only for normal variables)
        m_iNumofBddNormVar = 0;
        for (int i = 0; i < 2 * (this->GetNumofDES()); i = i + 2)
            m_iNumofBddNormVar += fdd_varnum(i);
        //set the first level block
        int iNumofBddVar = 0;
        int iVarNum = 0;
        bdd bddBlock = bddtrue;
        for (int i = 0; i < 2 * (this->GetNumofDES()); i += 2)
        {
            iVarNum = fdd_varnum(i);
            bddBlock = bddtrue;
            for (int j = 0; j < 2 * iVarNum; j++)
            {
                bddBlock &= bdd_ithvar(iNumofBddVar + j);
            }
            bdd_addvarblock(bddBlock, BDD_REORDER_FREE);
            iNumofBddVar += 2 * iVarNum;
        }
        //compute initial state predicate
        for (int i = 0; i < this->GetNumofDES(); i++)
            m_bddInit &= fdd_ithvar(i * 2, m_pDESArr[i]->GetInitState());
        //compute marking states predicate
        bdd bddTmp = bddfalse;
        for (int i = 0; i < this->GetNumofDES(); i++)
        {
            bddTmp = bddfalse;
            MARKINGLIST::const_iterator ci =
```

```
         (m_pDESArr[i]->GetMarkingList()).begin();
    for (int j = 0; j < m_pDESArr[i]->GetNumofMarkingStates(); j++)
    {
        bddTmp |= fdd_ithvar(i * 2, *ci);
        ++ci;
    }
    m_bddMarking &= bddTmp;
}
//Initialize interface variables predicates
m_pbdd_IVar = new bdd[m_iNumofIntfs];
m_pbdd_IVarPrim = new bdd[m_iNumofIntfs];
if (m_usiMaxUnCon[A_EVENT] != 0) //Answer events
    m_pbdd_UnConATrans = new bdd[m_usiMaxUnCon[A_EVENT]/2](bddfalse);
if (m_usiMaxCon[A_EVENT] != 0xFFFF) //Answer events
    m_pbdd_ConATrans = new bdd[m_usiMaxCon[A_EVENT] / 2 + 1](bddfalse);
//Compute transitions predicate
for (int k = 0; k < 3; k++)
{
    if (m_usiMaxCon[k] != 0xFFFF)
    {
        m_pbdd_ConTrans[k] = new bdd[m_usiMaxCon[k] / 2 + 1](bddfalse);
        m_pbdd_ConVar[k] = new bdd[m_usiMaxCon[k] / 2 + 1](bddfalse);
        m_pbdd_ConVarPrim[k] = new bdd[m_usiMaxCon[k]/ 2 + 1](bddfalse);
        m_pbdd_ConPhysicVar[k] =
                            new bdd[m_usiMaxCon[k] / 2 + 1](bddfalse);
        m_pbdd_ConPhysicVarPrim[k] =
                            new bdd[m_usiMaxCon[k] / 2 + 1](bddfalse);
        m_pPair_Con[k] = new (bddPair *)[m_usiMaxCon[k] / 2 + 1];
        for (int iPair = 0; iPair < m_usiMaxCon[k] / 2 + 1; iPair++)
            m_pPair_Con[k][iPair] = NULL;
        m_pPair_ConPrim[k] = new (bddPair *)[m_usiMaxCon[k] / 2 + 1];
        for (int iPair = 0; iPair < m_usiMaxCon[k] / 2 + 1; iPair++)
            m_pPair_ConPrim[k][iPair] = NULL;
    }
    if (m_usiMaxUnCon[k] != 0)
    {
        m_pbdd_UnConTrans[k] = new bdd[m_usiMaxUnCon[k]/2](bddfalse);
        m_pbdd_UnConPlantTrans[k] =
                               new bdd[m_usiMaxUnCon[k]/2](bddfalse);
        m_pbdd_UnConVar[k] = new bdd[m_usiMaxUnCon[k]/2](bddfalse);
        m_pbdd_UnConVarPrim[k] = new bdd[m_usiMaxUnCon[k]/2](bddfalse);
        m_pbdd_UnConPlantVar[k] = new bdd[m_usiMaxUnCon[k]/2](bddfalse);
        m_pbdd_UnConPlantVarPrim[k] =
                                 new bdd[m_usiMaxUnCon[k]/2](bddfalse);
        m_pPair_UnCon[k] = new (bddPair *)[m_usiMaxUnCon[k]/2];
        for (int iPair = 0; iPair < m_usiMaxUnCon[k]/2; iPair++)
            m_pPair_UnCon[k][iPair] = NULL;
        m_pPair_UnConPrim[k] = new (bddPair *)[m_usiMaxUnCon[k]/2];
        for (int iPair = 0; iPair < m_usiMaxUnCon[k]/2; iPair++)
            m_pPair_UnConPrim[k][iPair] = NULL;
    }
}
map<int, bdd> bddTmpTransMap;  //<event_index, transitions>
for (int i = 0, iIntf = 0; i < this->GetNumofDES(); i++)
{
    //before compute transition predicate for each DES, clear it.
    bddTmpTransMap.clear();
    for (int j = 0; j < m_pDESArr[i]->GetNumofEvents(); j++)
    {
        bddTmpTransMap[(m_pDESArr[i]->GetEventsArr())[j]] = bddfalse;
    }
    //compute transition predicate for each DES
    for (int j = 0; j < m_pDESArr[i]->GetNumofStates(); j++)
    {
        TRANS::const_iterator ci =
                            (*(m_pDESArr[i]->GetTrans() + j)).begin();
        for (; ci != (*(m_pDESArr[i]->GetTrans() + j)).end(); ++ci)
        {
            bddTmpTransMap[ci->first] |= fdd_ithvar(i * 2, j) &
                fdd_ithvar(i * 2 + 1, ci->second);
        }
    }
    int iIndex = 0;
    unsigned short usiIndex = 0;
    int iSub = 0;
    EVENTSUB EventSub;
```

```
//combine the current DES transition predicate to
//subsystem transition predicate
map<int, bdd>::const_iterator ciTmp = bddTmpTransMap.begin();
for (; ciTmp != bddTmpTransMap.end(); ++ciTmp)
{
    if (ciTmp->first % 2 == 0)  //uncontrollable, start from 2
    {
        //Compute index
        pPrj->GenEventInfo(ciTmp->first, EventSub, iSub, usiIndex);
        usiIndex = (usiIndex - 2) / 2;
        if (EventSub == R_EVENT || EventSub == A_EVENT)
            iIndex = usiIndex +
                            m_piUArr[(int)EventSub - 1][iSub - 1];
        else
            iIndex = usiIndex;
        //Compute transition predicates
        if (m_pbdd_UnConVar[EventSub][iIndex] == bddfalse)
        {
            m_pbdd_UnConTrans[EventSub][iIndex] = bddtrue;
            m_pbdd_UnConVar[EventSub][iIndex] = bddtrue;
            m_pbdd_UnConVarPrim[EventSub][iIndex] = bddtrue;
        }
        m_pbdd_UnConTrans[EventSub][iIndex] &= ciTmp->second;
        m_pbdd_UnConVar[EventSub][iIndex] &= fdd_ithset(i * 2);
        m_pbdd_UnConVarPrim[EventSub][iIndex] &=
                                            fdd_ithset(i * 2 + 1);
        //compute uncontrollable plant vars and varprimes
        if (m_pDESArr[i]->GetDESType() != SPEC_DES)
        {
            if (m_pbdd_UnConPlantVar[EventSub][iIndex] == bddfalse)
            {
                m_pbdd_UnConPlantTrans[EventSub][iIndex] = bddtrue;
                m_pbdd_UnConPlantVar[EventSub][iIndex] = bddtrue;
                m_pbdd_UnConPlantVarPrim[EventSub][iIndex]= bddtrue;
            }
            m_pbdd_UnConPlantTrans[EventSub][iIndex] &=
                                                ciTmp->second;
            m_pbdd_UnConPlantVar[EventSub][iIndex] &=
                                                fdd_ithset(i * 2);
            m_pbdd_UnConPlantVarPrim[EventSub][iIndex] &=
                                            fdd_ithset(i * 2 + 1);
        }
        //compute uncontrollable answer events transitions
        if (m_pDESArr[i]->GetDESType() == INTERFACE_DES &&
            EventSub == A_EVENT)
            m_pbdd_UnConATrans[iIndex] = ciTmp->second;
    }
    else  //controllable, start from 1
    {
        //Compute index
        pPrj->GenEventInfo(ciTmp->first, EventSub, iSub, usiIndex);
        usiIndex = (usiIndex - 1) / 2;
        if (EventSub == R_EVENT || EventSub == A_EVENT)
            iIndex = usiIndex +
                        m_piCArr[(int)EventSub - 1][iSub - 1];
        else
            iIndex = usiIndex;
        //Compute transition predicate
        if (m_pbdd_ConVar[EventSub][iIndex] == bddfalse)
        {
            m_pbdd_ConTrans[EventSub][iIndex] = bddtrue;
            m_pbdd_ConVar[EventSub][iIndex] = bddtrue;
            m_pbdd_ConVarPrim[EventSub][iIndex] = bddtrue;
        }
        m_pbdd_ConTrans[EventSub][iIndex] &= ciTmp->second;
        m_pbdd_ConVar[EventSub][iIndex] &= fdd_ithset(i * 2);
        m_pbdd_ConVarPrim[EventSub][iIndex] &=fdd_ithset(i * 2 + 1);
        //compute controllable actual plant vars and varprimes
        //note: can not use != SPEC_DES
        if (m_pDESArr[i]->GetDESType() == PLANT_DES)
        {
            if (m_pbdd_ConPhysicVar[EventSub][iIndex] == bddfalse)
            {
                m_pbdd_ConPhysicVar[EventSub][iIndex] = bddtrue;
                m_pbdd_ConPhysicVarPrim[EventSub][iIndex] = bddtrue;
            }
            m_pbdd_ConPhysicVar[EventSub][iIndex] &=
```

```
                                                    fdd_ithset(i * 2);
                        m_pbdd_ConPhysicVarPrim[EventSub][iIndex] &=
                                                   fdd_ithset(i * 2 + 1);
                    }
                    //compute controllable answer events transitions
                    if (m_pDESArr[i]->GetDESType() == INTERFACE_DES &&
                                                    EventSub == A_EVENT)
                        m_pbdd_ConATrans[iIndex] = ciTmp->second;
                }
            }
            if (m_pDESArr[i]->GetDESType() == INTERFACE_DES)
            {
                m_pbdd_IVar[iIntf] = fdd_ithset(i * 2);
                m_pbdd_IVarPrim[iIntf] = fdd_ithset(i * 2 + 1);
                iIntf++;
            }
        }
        //compute m_pPair_UnCon, m_pPair_Con
        for (int k = 0; k < 3; k++)
        {
            for (int j = 0; j < m_usiMaxUnCon[k]; j += 2)
            {
                m_pPair_UnCon[k][j/2] = bdd_newpair();
                SetBddPairs(m_pPair_UnCon[k][j/2], m_pbdd_UnConVar[k][j/2],
                            m_pbdd_UnConVarPrim[k][j/2]);
                m_pPair_UnConPrim[k][j/2] = bdd_newpair();
                SetBddPairs(m_pPair_UnConPrim[k][j/2],
                            m_pbdd_UnConVarPrim[k][j/2],
                            m_pbdd_UnConVar[k][j/2]);
            }
            for (int j = 1; j < (unsigned short)(m_usiMaxCon[k] + 1); j += 2)
            //m_usiMaxCon[k] might be 0xFFFF
            {
                m_pPair_Con[k][(j - 1) / 2] = bdd_newpair();
                SetBddPairs(m_pPair_Con[k][(j - 1) / 2],
                            m_pbdd_ConVar[k][(j - 1) / 2],
                            m_pbdd_ConVarPrim[k][(j - 1) / 2]);
                m_pPair_ConPrim[k][(j - 1) / 2] = bdd_newpair();
                SetBddPairs(m_pPair_ConPrim[k][(j - 1) / 2],
                            m_pbdd_ConVarPrim[k][(j - 1) / 2],
                            m_pbdd_ConVar[k][(j - 1) / 2]);
            }
        }
    }
    catch(...)
    {
        string sErr;
        sErr = "Error happens when initializing high level ";
        sErr += " BDD!";
        pPrj->SetErr(sErr, HISC_SYSTEM_INITBDD);
        return -1;
    }
    return 0;
}
```

```
/*****************************************************************************
  FILE:  HighSub1.cpp
  DESCR: Some misc functions for high-level
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*****************************************************************************/
#include "Sub.h"
#include "HighSub.h"
#include "LowSub.h"
#include <string>
#include "errmsg.h"
#include "type.h"
#include <fdd.h>
#include "DES.h"
#include "Project.h"
#include <fstream>
#include "pubfunc.h"
using namespace std;
extern CProject *pPrj;
/**
 * DESCR:   Save DES list of high-level in memory to a file (for checking)
 * PARA:    fout: output file stream
 * RETURN:  0: sucess -1: fail
 * ACCESS:  public
 */
int CHighSub::PrintSub(ofstream& fout)
{
    try
    {
        fout << "#Sub system: " <<  m_sSubName << endl;
        fout << endl;
        fout << "[SYSTEM]" << endl;
        fout << m_iNumofPlants << endl;
        fout << m_iNumofSpecs << endl;
        fout << endl;
        fout << "[PLANT]" << endl;
        for (int i = 0; i < m_iNumofPlants; i++)
        {
            for (int j = 0; j < this->GetNumofDES(); j++)
            {
                if (m_piDESOrderArr[j] == i)
                {
                    fout << m_pDESArr[j]->GetDESName() << endl;
                    break;
                }
            }
        }
        fout << "[SPEC]" << endl;
        for (int i = m_iNumofPlants; i < m_iNumofPlants + m_iNumofSpecs; i++)
        {
            for (int j = 0; j < this->GetNumofDES(); j++)
            {
                if (m_piDESOrderArr[j] == i)
                {
                    fout << m_pDESArr[j]->GetDESName() << endl;
                    break;
                }
            }
        }
        fout << "################################################" << endl;
    }
    catch(...)
    {
        return -1;
    }
    return 0;
}
/**
 * DESCR:   Save all the DES in high-level to a text file for checking
 * PARA:    fout: output file stream
 * RETURN:  0: sucess -1: fail
 * ACCESS:  public
 */
int CHighSub::PrintSubAll(ofstream & fout)
{
    try
    {
        if (PrintSub(fout) < 0)
            throw -1;
        for (int i = 0; i < m_iNumofPlants + m_iNumofSpecs; i++)
        {
```

```
            if (m_pDESArr[i]->PrintDES(fout) < 0)
                throw -1;
        }
    }
    catch(...)
    {
        return -1;
    }
    return 0;
}
/*
 * DESCR: Generate Bad state info during verfication
 *        Note: showtrace is not implemented, currently it is used for showing
 *              a blocking is a deadlock or livelock (very slow).
 * PARA:    bddBad: BDD for the set of bad states
 *          viErrCode: error code (see errmsg.h)
 *          showtrace: show a trace from the initial state to a bad state or not
 *                     (not implemented)
 *          vsExtraInfo: Extra errmsg.
 * RETURN:  None
 * ACCESS:  private
 */
void CHighSub::BadStateInfo(const bdd& bddBad, const int viErrCode,
                    const HISC_TRACETYPE showtrace, const string &vsExtraInfo)
{
    int *piBad = fdd_scanallvar(bddBad);
    string sErr;
    if (piBad != NULL)
    {
        if (viErrCode == HISC_VERI_HIGH_UNCON)
            sErr = GetSubName() +
                        ": Level-wise controllable checking failed state:\n< ";
        else if (viErrCode == HISC_VERI_HIGH_P3FAILED)
            sErr = GetSubName() +
              ": Interface consistent conditions P3 checking failed state:\n< ";
        else if (viErrCode == HISC_VERI_HIGH_BLOCKING)
        {
            if (showtrace == HISC_SHOW_TRACE)
                sErr = GetSubName() + ": Blocking state ";
            else
                sErr = GetSubName() + ": Blocking state: \n < ";
        }
        //for blocking state, try to find the deadlock state
        //if there is no deadlock state, only show one of the live lock states
        if (showtrace == HISC_SHOW_TRACE)
        {
            if (viErrCode == HISC_VERI_HIGH_BLOCKING)
            {
                bdd bddBlock = bddBad;
                bdd bddNext = bddtrue;
                bdd bddTemp = bddtrue;
                do
                {
                    bddTemp = bddtrue;
                    for (int i = 0; i < this->GetNumofDES(); i++)
                        bddTemp &= fdd_ithvar(i * 2, piBad[i * 2]);
                    bddNext = bddfalse;
                    for (int k = 0; k < 4; k++)
                    {
                        for (unsigned short usi = 2;
                                usi <= m_usiMaxUnCon[k] &&
                                    bddNext == bddfalse; usi += 2)
                            bddNext |=
                                bdd_replace(
                                    bdd_relprod(
                                        m_pbdd_UnConTrans[k][(usi - 2) / 2],
                                        bddTemp,
                                        m_pbdd_UnConVar[k][(usi - 2) / 2]),
                                    m_pPair_UnConPrim[k][(usi - 2) / 2]) &
                                bddBad;
                        for (unsigned short usi = 1;
                            usi < (unsigned short) (m_usiMaxCon[k] + 1) &&
                                bddNext == bddfalse; usi += 2)
                        {
                            bddNext |=
                                bdd_replace(
                                    bdd_relprod(
                                        m_pbdd_ConTrans[k][(usi - 1) / 2],
                                        bddTemp,
                                        m_pbdd_ConVar[k][(usi - 1) / 2]),
```

```
                                m_pPair_ConPrim[k][(usi - 1) / 2]) &
                            bddBad;
                    }
                }
                if (bddNext == bddfalse)  //this is a deadlock state
                {
                    sErr += "(dead lock):\n< ";
                    break;
                }
                else //not a deadlock state
                {
                    bddBlock = bddBlock - bddTemp;
                    free(piBad);
                    piBad = NULL;
                    piBad = fdd_scanallvar(bddBlock);
                }
            } while (piBad != NULL);
            if (piBad == NULL) //live lock
            {
                sErr += "(live lock):\n< ";
                piBad = fdd_scanallvar(bddBad);
            }
        }
    }
    if (piBad != NULL)
    {
        for (int i = 0; i < this->GetNumofDES(); i++)
        {
            sErr += m_pDESArr[m_piDESPosArr[i]]->GetDESName() + ":" +
    m_pDESArr[m_piDESPosArr[i]]->GetStateName(piBad[m_piDESPosArr[i] * 2]);
            if (i < this->GetNumofDES() -1)
                sErr += ", ";
        }
    }
    sErr += " >\n";
    sErr += vsExtraInfo;
    pPrj->SetErr(sErr, viErrCode);
    free(piBad);
    piBad = NULL;
  }
  return;
}
/**
 * DESCR: Search event name from high-level local event index.
 *        Mainly handle request events and answer events.
 * PARA:    k: R_EVENT/A_EVENT/H_EVENT/L_EVENT
 *          usiLocalIndex: high-level local event index.
 * RETURN:  event name
 * ACCESS:  public
 */
string CHighSub::SearchEventName(EVENTSUB k, unsigned short usiLocalIndex)
{
    int iEventIndex = 0;
    if (k == R_EVENT || k == A_EVENT)
    {
        int iSub = 0;
        unsigned short usiIndex = 0;
        if (usiLocalIndex % 2 == 0)
        {
            for (iSub = 0; iSub < m_iNumofIntfs - 1; iSub++)
            {
                if (m_piUArr[k - 1][iSub] <= (usiLocalIndex - 2) / 2 &&
                            m_piUArr[k - 1][iSub + 1] > (usiLocalIndex - 2) /2)
                    break;
            }
            usiIndex = usiLocalIndex - m_piUArr[k - 1][iSub] * 2;
        }
        else
        {
            for (iSub = 0; iSub < m_iNumofIntfs - 1; iSub++)
            {
                if (m_piCArr[k - 1][iSub] <= (usiLocalIndex - 1) / 2 &&
                            m_piCArr[k - 1][iSub + 1] > (usiLocalIndex - 1) /2)
                    break;
            }
            usiIndex = usiLocalIndex - m_piCArr[k - 1][iSub] * 2;
        }
        iEventIndex = pPrj->GenEventIndex((EVENTSUB)k, iSub + 1, usiIndex);
    }
```

```
    else
        iEventIndex = pPrj->GenEventIndex((EVENTSUB)k, m_iSubIndex,
                                                        usiLocalIndex);
    return (pPrj->GetInvAllEventsMap())[iEventIndex];
}
```

```
/*************************************************************************
  FILE:  HighSub2.cpp
  DESCR: Verification and synthesis for high-level
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*************************************************************************/
#include <fdd.h>
#include <string>
#include <iostream>
#include "errmsg.h"
#include "Project.h"
#include "HighSub.h"
#include "Sub.h"
#include "LowSub.h"
#include "DES.h"
#include "type.h"
#include "pubfunc.h"
#include <sys/time.h>
using namespace std;
extern CProject *pPrj;
/**
 * DESCR:   Verify whether a high level is level-wise controllable,
 *          nonblocking and  p3 satisfied or not.
 * PARA:    showtrace:show a trace to a bad state or not(not implemented)(input)
 *          superinfo: returned verification info (see BddHisc.h)(output)
 *          savetype: save syn-product or not (See BddHisc.h)(input)
 *          savepath: where to save syn-product(input)
 * RETURN:  0: sucess <0: fail
 * ACCESS:  public
 */
int CHighSub::VeriSub(const HISC_TRACETYPE showtrace,
                      HISC_SUPERINFO & superinfo,
                      const HISC_SAVEPRODUCTTYPE savetype,
                      const string& savepath)
{
    int iRet = 0;
    int iErr = 0;
    //Initialize the BDD data memebers
    CSub::InitBddFields();
    InitBddFields();
    bdd bddReach = bddfalse;
    string sErr;
    try
    {
        //Make transition bdds
        if (MakeBdd() < 0)
            throw -1;
        bdd bddBad = bddfalse;
        bdd bddCoreach = bddfalse;
        bdd bddBlocking = bddfalse;
        bddReach = r(bddtrue, iErr);
        if (iErr < 0)
            throw -1;
        m_bddMarking &= bddReach;
        //Initialize controllable bddBad
        bddBad = bddfalse;
        if (VeriConBad(bddBad, bddReach, sErr) < 0)
            throw -1;
        //check if any reachable states belong to bad states
        if (bddBad != bddfalse)
        {
            BadStateInfo(bddBad, HISC_VERI_HIGH_UNCON, showtrace, sErr);
            throw -2;
        }
        bddBad = bddfalse;
        //Initialize P3 bddBad
        if (VeriP3Bad(bddBad, bddReach, sErr) < 0)
            throw -1;
        //check if any reachable states belong to bad states
        if (bddBad != bddfalse)
        {
            BadStateInfo(bddBad, HISC_VERI_HIGH_P3FAILED, showtrace, sErr);
            throw -3;
        }
        bddBad = bddfalse;
        //Computing CR(not(bddBad))
        bddCoreach = cr(m_bddMarking, bddReach, ALL_EVENT, iErr);
        if (iErr != 0)
            throw -1;
        //check if the system is nonblocking
```

```
        bddBlocking = bddReach - bddCoreach;
        if (bddBlocking != bddfalse)
        {
            BadStateInfo(bddBlocking, HISC_VERI_HIGH_BLOCKING, showtrace);
            throw -4;
        }
        //final synchronous product;
        m_bddSuper = bddReach;
        //save supervisor
        superinfo.statesize = bdd_satcount(m_bddSuper)/pow((double)2,
                                                double(m_iNumofBddNormVar));
        superinfo.nodesize =  bdd_nodecount(m_bddSuper);
        if (savetype != HISC_NOTSAVEPRODUCT)
        {
            if (SaveSuper(m_bddSuper, HISC_SAVESUPER_AUTOMATA, savepath) < 0)
                throw -1;
        }
    }
    catch (int iResult)
    {
        if (iResult < -1)
        {
            superinfo.statesize = bdd_satcount(bddReach)/pow((double)2,
                                                double(m_iNumofBddNormVar));
            superinfo.nodesize =  bdd_nodecount(bddReach);
            if (savetype != HISC_NOTSAVEPRODUCT)
                SaveSuper(bddReach, HISC_SAVESUPER_AUTOMATA, savepath);
        }
        iRet = -1;
    }
    ClearBddFields();
    CSub::ClearBddFields();
    bdd_done();
    return iRet;
}
/**
 * DESCR:   Synthesize a supervisor for high-level. The superivsor states is
 *          saved in m_bddSuper.
 * PARA:    computemethod: first compute reachable states or not (See BddHisc.h)
 *                         (input)
 *          superinfo: returned synthesis info (see BddHisc.h)(output)
 *          savetype: how to save synthesized supervisor (See BddHisc.h)(input)
 *          savepath: where to save syn-product(input)
 * RETURN:  0: sucess <0: fail
 * ACCESS:  public
 */
int CHighSub::SynSuper(const HISC_COMPUTEMETHOD computemethod,
                       HISC_SUPERINFO &superinfo,
                       const HISC_SAVESUPERTYPE savetype,
                       const string& savepath)
{
    bool bFirstLoop = true;
    bdd bddK = bddfalse;
    bdd bddReach = bddfalse;
    bdd bddDead = bddfalse;
    int iRet = 0;
    int iErr = 0;
    //Initialize the BDD data memebers
    CSub::InitBddFields();
    InitBddFields();
    try
    {
        iRet = 0;
        //Make transition bdds
        if (MakeBdd() < 0)
            throw -1;
        //Initialize bddBad
        bdd bddBad = bddfalse;
        if (GenConBad(bddBad) < 0)
            throw -1;
        if (GenP3Bad(bddBad) < 0)
            throw -1;
        //always compute on reachable states, this is the one we used for AIP
        if (computemethod == HISC_ONREACHABLE)
        {
            bddReach = r(bddtrue, iErr);
            if (iErr < 0)
                throw -1;
            bddBad &= bddReach;
```

```
            //compute controllable, p3, nonblocking fixpoint
            do
            {
                bddK = bddBad;
                if (supcp(bddBad) < 0)
                    throw -1;
                bddBad &= bddReach;
                if (bddK == bddBad && bFirstLoop == false)
                    break;
                //Computing CR()
                bdd bddTemp = bddReach - bddBad;
                bddBad = bdd_not(cr(m_bddMarking, bddTemp, ALL_EVENT, iErr));
                if (iErr != 0)
                    throw -1;
                bddBad &= bddReach;
                bFirstLoop = false;
            } while (bddBad != bddK);
            m_bddSuper = r(bddReach - bddBad, iErr);
            if (iErr < 0)
                throw -1;
        }
        else  //first compute on coreachable states, then do reachable operation
        {
            //compute controllable, p3, nonblocking fixpoint
            do
            {
                bddK = bddBad;
                if (supcp(bddBad) < 0)
                    throw -1;
                if (bddK == bddBad && bFirstLoop == false)
                    break;
                //Computing CR(not(bddBad))
                bddBad = bdd_not(cr(m_bddMarking, bdd_not(bddBad),
                                                        ALL_EVENT, iErr));
                if (iErr != 0)
                    throw -1;
                bFirstLoop = false;
            } while (bddBad != bddK);
            m_bddSuper = r(bdd_not(bddBad), iErr);
            if (iErr < 0)
                throw -1;
        }
        m_bddMarking &= m_bddSuper;
        m_bddInit &= m_bddSuper;
        #ifdef PRINTDEBUG
        PrintBdd();
        #endif
        //save supervisor
        superinfo.statesize = bdd_satcount(m_bddSuper)/pow((double)2,
                                              double(m_iNumofBddNormVar));
        superinfo.nodesize =  bdd_nodecount(m_bddSuper);
        if (savetype != HISC_SAVESUPER_NONE)
        {
            if (SaveSuper(m_bddSuper, savetype, savepath) < 0)
                throw -1;
        }
    }
    catch( int iErr)
    {
        iRet = -1;
    }
    ClearBddFields();
    CSub::ClearBddFields();
    bdd_done();
    return iRet;
}
/**
 * DESCR:   Compute the initial bad states(Bad_H)(uncontorlalble event part)
 * PARA:    bddConBad: BDD containing all the bad states (output)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
 */
int CHighSub::GenConBad(bdd &bddConBad)
{
    try
    {
        bdd bddPlantTrans = bddfalse;
        bdd bddSpecTrans = bddfalse;
        for (int k = 0; k < 3; k++)
```

```
        {
            for (int i = 0; i < m_usiMaxUnCon[k]/ 2; i++)
            {
                //Get spec transition predicate
                bddSpecTrans = bdd_exist(m_pbdd_UnConTrans[k][i],
                                    m_pbdd_UnConPlantVar[k][i]);
                bddSpecTrans = bdd_exist(bddSpecTrans,
                                    m_pbdd_UnConPlantVarPrim[k][i]);
                //Compute illegal state predicate for each uncontrollable event
                bddConBad |=
                    bdd_exist(m_pbdd_UnConPlantTrans[k][i],
                              m_pbdd_UnConPlantVarPrim[k][i]) &
                    bdd_not(
                        bdd_exist(bddSpecTrans,
                                    bdd_exist(m_pbdd_UnConVarPrim[k][i],
                                              m_pbdd_UnConPlantVarPrim[k][i])));
            }
        }
    }
    catch(...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during generating controllable bad states.";
        pPrj->SetErr(sErr, HISC_HIGHERR_GENCONBAD);
        return -1;
    }
    return 0;
}
/**
 * DESCR:   Test if there are any bad states in the reachable states
 *          (Uncontorllable event part of Bad_H)
 * PARA:    bddConBad: BDD containing tested bad states(output).
 *                  Initially, bddBad should be bddfalse.
 *          bddReach: BDD containing all reachable states in high-level (input)
 *          vsErr: returned errmsg(output)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
 */
int CHighSub::VeriConBad(bdd &bddConBad, const bdd &bddReach, string & vsErr)
{
    vsErr.clear();
    try
    {
        bdd bddPlantTrans = bddfalse;
        bdd bddSpecTrans = bddfalse;
        for (int k = 0; k < 3; k++)
        {
            for (int i = 0; i < m_usiMaxUnCon[k]/ 2; i++)
            {
                //Get spec transition predicate
                bddSpecTrans = bdd_exist(m_pbdd_UnConTrans[k][i],
                                    m_pbdd_UnConPlantVar[k][i]);
                bddSpecTrans = bdd_exist(bddSpecTrans,
                                    m_pbdd_UnConPlantVarPrim[k][i]);
                //Compute illegal state predicate for each uncontrollable event
                bddConBad |=
                    bdd_exist(m_pbdd_UnConPlantTrans[k][i],
                              m_pbdd_UnConPlantVarPrim[k][i]) &
                    bdd_not(bdd_exist(bddSpecTrans,
                            bdd_exist(m_pbdd_UnConVarPrim[k][i],
                                      m_pbdd_UnConPlantVarPrim[k][i])));
                bddConBad &= bddReach;
                if (bddConBad != bddfalse)
                {
                    vsErr = "Causing uncontrollable event:";
                    vsErr += SearchEventName((EVENTSUB)k, (i + 1)* 2);
                    throw -1;
                }
            }
        }
    }
    catch(int)
    {
    }
    catch(...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during generating controllable bad states.";
```

```
        pPrj->SetErr(sErr, HISC_HIGHERR_GENCONBAD);
        return -1;
    }
    return 0;
}
/**
 * DESCR:   Compute the initial bad states(Bad_H)(answer event part)
 * PARA:    bddConBad: BDD containing all the bad states (output)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
 */
int CHighSub::GenP3Bad(bdd &bddP3Bad)
{
    try
    {
        bdd bddHighTrans = bddfalse;
        EVENTSUB EventSub;
        int iSub = 0;
        unsigned short usiLocalIndex = 0;
        int iIndex = 0;
        for (int iDES = 0, iIntf = 0; iDES < this->GetNumofDES(); iDES++)
        {
            if (m_pDESArr[iDES]->GetDESType() == INTERFACE_DES)
            {
                for (int i = 0; i < m_pDESArr[iDES]->GetNumofEvents(); i++)
                {
                    pPrj->GenEventInfo((m_pDESArr[iDES]->GetEventsArr())[i],
                                       EventSub, iSub, usiLocalIndex);
                    if (EventSub == A_EVENT)
                    {
                        if (usiLocalIndex % 2 == 0)
                        {
                            usiLocalIndex = (usiLocalIndex - 2) / 2;
                            iIndex = usiLocalIndex +
                                     m_piUArr[EventSub - 1][iSub - 1];
                            bddHighTrans = bdd_exist(
                                            m_pbdd_UnConTrans[A_EVENT][iIndex],
                                            m_pbdd_IVar[iIntf]);
                            bddHighTrans = bdd_exist(bddHighTrans,
                                                     m_pbdd_IVarPrim[iIntf]);
                            //Compute illegal predicate for each answer event
                            bddP3Bad |=
                                bdd_exist(m_pbdd_UnConATrans[iIndex],
                                          m_pbdd_IVarPrim[iIntf]) &
                                bdd_not(
                                    bdd_exist(bddHighTrans,
                                      bdd_exist(
                                        m_pbdd_UnConVarPrim[A_EVENT][iIndex],
                                        m_pbdd_IVarPrim[iIntf])));
                        }
                        else
                        {
                            usiLocalIndex = (usiLocalIndex - 1) / 2;
                            iIndex = usiLocalIndex +
                                     m_piCArr[EventSub - 1][iSub - 1];
                            bddHighTrans = bdd_exist(
                                            m_pbdd_ConTrans[A_EVENT][iIndex],
                                            m_pbdd_IVar[iIntf]);
                            bddHighTrans = bdd_exist(bddHighTrans,
                                            m_pbdd_IVarPrim[iIntf]);
                            //Compute illegal predicate for each answer event
                            bddP3Bad |=
                                bdd_exist(m_pbdd_ConATrans[iIndex],
                                          m_pbdd_IVarPrim[iIntf]) &
                                bdd_not(bdd_exist(bddHighTrans,
                                            bdd_exist(
                                             m_pbdd_ConVarPrim[A_EVENT][iIndex],
                                             m_pbdd_IVarPrim[iIntf])));
                        }
                    }
                }
                iIntf++;
            }
        }
    }
    catch(...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during generating point 3 bad states.";
```

```
        pPrj->SetErr(sErr, HISC_HIGHERR_GENP3BAD);
        return -1;
    }
    return 0;
}
/**
 * DESCR:   Test if there are any bad states in the reachable states
 *          (answer event part of Bad_H)
 * PARA:    bddP3Bad: BDD containing tested bad states(output).
 *                    Initially, bddBad should be bddfalse.
 *          bddReach: BDD containing all reachable states in high-level (input)
 *          vsErr: returned errmsg(output)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
 */
int CHighSub::VeriP3Bad(bdd &bddP3Bad, const bdd &bddReach, string &vsErr)
{
    vsErr.clear();
    try
    {
        bdd bddHighTrans = bddfalse;
        EVENTSUB EventSub;
        int iSub = 0;
        unsigned short usiLocalIndex = 0;
        int iIndex = 0;
        for (int iDES = 0, iIntf = 0; iDES < this->GetNumofDES(); iDES++)
        {
            if (m_pDESArr[iDES]->GetDESType() == INTERFACE_DES)
            {
                for (int i = 0; i < m_pDESArr[iDES]->GetNumofEvents(); i++)
                {
                    pPrj->GenEventInfo((m_pDESArr[iDES]->GetEventsArr())[i],
                                       EventSub, iSub, usiLocalIndex);
                    if (EventSub == A_EVENT)
                    {
                        if (usiLocalIndex % 2 == 0)
                        {
                            iIndex = (usiLocalIndex - 2) / 2 +
                                          m_piUArr[EventSub - 1][iSub - 1];
                            bddHighTrans = bdd_exist(
                                m_pbdd_UnConTrans[A_EVENT][iIndex],
                                m_pbdd_IVar[iIntf]);
                            bddHighTrans = bdd_exist(bddHighTrans,
                                                     m_pbdd_IVarPrim[iIntf]);
                            //Compute illegal predicate for each anwer event
                            bddP3Bad |=
                                bdd_exist(m_pbdd_UnConATrans[iIndex],
                                          m_pbdd_IVarPrim[iIntf]) &
                                bdd_not(bdd_exist(bddHighTrans,bdd_exist(
                                        m_pbdd_UnConVarPrim[A_EVENT][iIndex],
                                        m_pbdd_IVarPrim[iIntf])));
                            bddP3Bad &= bddReach;
                            if (bddP3Bad != bddfalse)
                            {
                                vsErr = "Causing answer event:";
                                vsErr += SearchEventName(A_EVENT,
                                          usiLocalIndex +
                                          m_piUArr[EventSub - 1][iSub - 1] * 2);
                                return 0;
                            }
                        }
                        else
                        {
                            iIndex = (usiLocalIndex - 1) / 2 +
                                          m_piCArr[EventSub - 1][iSub - 1];
                            bddHighTrans = bdd_exist(
                                          m_pbdd_ConTrans[A_EVENT][iIndex],
                                          m_pbdd_IVar[iIntf]);
                            bddHighTrans = bdd_exist(bddHighTrans,
                                          m_pbdd_IVarPrim[iIntf]);
                            //Compute illegal predicate for each answer event
                            bddP3Bad |= bdd_exist(m_pbdd_ConATrans[iIndex],
                                                  m_pbdd_IVarPrim[iIntf]) &
                                        bdd_not(bdd_exist(bddHighTrans,
                                            bdd_exist(
                                             m_pbdd_ConVarPrim[A_EVENT][iIndex],
                                             m_pbdd_IVarPrim[iIntf])));
                            bddP3Bad &= bddReach;
```

```
                            if (bddP3Bad != bddfalse)
                            {
                                vsErr = "Causing answer event:";
                                vsErr += SearchEventName(A_EVENT,
                                         usiLocalIndex +
                                         m_piCArr[EventSub - 1][iSub - 1] * 2);
                                throw -1;
                            }
                        }
                    }
                }
                iIntf++;
            }
        }
    }
    catch(int)
    {
    }
    catch(...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during generating point 3 bad states.";
        pPrj->SetErr(sErr, HISC_HIGHERR_GENP3BAD);
        return -1;
    }
    return 0;
}
/**
 * DESCR:   compute PHIC(P)
 * PARA:    bddP : BDD for predicate P. (input and output(=PHIC(P)))
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
 */
int CHighSub::supcp(bdd & bddP)
{
    bdd bddK1 = bddfalse;
    bdd bddK2 = bddfalse;
    int iEvent = 0;
    unsigned short usiIndex = 0;
    int iSub = 0;
    EVENTSUB EventSub;
    int iIndex = 0;
    try
    {
        while (bddP != bddK1)
        {
            bddK1 = bddP;
            for (int i = 0; i < this->GetNumofDES(); i++)
            {
                bddK2 = bddfalse;
                while (bddP != bddK2)
                {
                    bddK2 = bddP;
                    for (int j = 0; j < m_pDESArr[i]->GetNumofEvents(); j++)
                    {
                        iEvent = (m_pDESArr[i]->GetEventsArr())[j];
                        pPrj->GenEventInfo(iEvent, EventSub, iSub, usiIndex);
                        if ( iEvent % 2 == 0) //Uncontrollable events
                        {
                            usiIndex = (usiIndex - 2) / 2;
                            if (EventSub == R_EVENT || EventSub == A_EVENT)
                                iIndex = usiIndex +
                                         m_piUArr[(int)EventSub - 1][iSub - 1];
                            else
                                iIndex = usiIndex;
                            bddP |= bdd_appex(
                                    m_pbdd_UnConTrans[EventSub][iIndex],
                                    bdd_replace(bddK2,
                                                m_pPair_UnCon[EventSub][iIndex]),
                                    bddop_and,
                                    m_pbdd_UnConVarPrim[EventSub][iIndex]);
                        }
                        else if ( EventSub == A_EVENT )//should be controllable
                        {
                            usiIndex = (usiIndex - 1) / 2;
                            iIndex = usiIndex +
                                    m_piCArr[(int)EventSub - 1][iSub - 1];
                            bddP |=
                                bdd_appex(m_pbdd_ConTrans[EventSub][iIndex],
                                    bdd_replace(bddK2,
```

```
                                        m_pPair_Con[EventSub][iIndex]),
                                    bddop_and,
                                    m_pbdd_ConVarPrim[EventSub][iIndex]);
                            }
                        }
                    }
                }
            }
        }
    }
    catch (...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during computing low level PHIC(P).";
        pPrj->SetErr(sErr, HISC_HIGHERR_SUPCP);
        return -1;
    }
    return 0;
}
/**
 * DESCR:   compute CR(G_H, P', \Sigma', P)
 * PARA:    bddPStart: P' (input)
 *          bddP: P (input)
 *          viEventSub: \Sigma' (input) (0,1,2,3) <-> (H,R,A,L) ALL_EVENT<->All
 *          iErr: returned Errcode (0: success <0: fail)(output)
 * RETURN:  BDD for CR(G_H, P', \Sigma', P)
 * ACCESS:  private
 */
bdd CHighSub::cr(const bdd & bddPStart, const bdd & bddP,
                 const int viEventSub, int & iErr)
{
    try
    {
        bdd bddK = bddP & bddPStart;
        bdd bddK1 = bddfalse;
        bdd bddK2 = bddfalse;
        bdd bddKNew = bddfalse;
        int iEvent = 0;
        unsigned short usiIndex = 0;
        int iIndex = 0;
        EVENTSUB EventSub;
        int iSub = 0;
        while (bddK != bddK1)
        {
            bddK1 = bddK;
            for (int i = 0; i < this->GetNumofDES(); i++)
            {
                bddK2 = bddfalse;
                while (bddK != bddK2)
                {
                    bddKNew = bddK - bddK2;
                    bddK2 = bddK;
                    for (int j = 0; j < m_pDESArr[i]->GetNumofEvents(); j++)
                    {
                        iEvent = (m_pDESArr[i]->GetEventsArr())[j];
                        pPrj->GenEventInfo(iEvent, EventSub, iSub, usiIndex);
                        if (viEventSub == ALL_EVENT ||
                                          viEventSub == (int)EventSub)
                        {
                            if (iEvent % 2 == 0) //Uncontrollable
                            {
                                //Compute index
                                usiIndex = (usiIndex - 2) / 2;
                                if (EventSub == R_EVENT || EventSub == A_EVENT)
                                    iIndex = usiIndex +
                                        m_piUArr[(int)EventSub - 1][iSub - 1];
                                else
                                    iIndex = usiIndex;
                                //Compute bddK
                                bddK |=
                                    bdd_appex(
                                        m_pbdd_UnConTrans[EventSub][iIndex],
                                        bdd_replace(bddKNew,
                                          m_pPair_UnCon[EventSub][iIndex]),
                                          bddop_and,
                                          m_pbdd_UnConVarPrim[EventSub][iIndex])
                                        & bddP;
                            }
                            else
                            {
                                //Compute Index
```

```
                                usiIndex = (usiIndex - 1) / 2;
                                if (EventSub == R_EVENT || EventSub == A_EVENT)
                                    iIndex = usiIndex +
                                        m_piCArr[(int)EventSub - 1][iSub - 1];
                                else
                                    iIndex = usiIndex;
                                //Compute bddK
                                bddK |= bdd_appex(
                                    m_pbdd_ConTrans[EventSub][iIndex],
                                    bdd_replace(bddKNew,
                                                m_pPair_Con[EventSub][iIndex]),
                                    bddop_and,
                                    m_pbdd_ConVarPrim[EventSub][iIndex])
                                    & bddP;
                            }
                        }
                    }
                }
            }
        }
        return bddK;
    }
    catch (...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during computing coreachable.";
        pPrj->SetErr(sErr, HISC_HIGHERR_COREACH);
        iErr = -1;
        return bddfalse;
    }
}
/**
 * DESCR:   compute R(G_H, P)
 * PARA:    bddP: P (input)
 *          iErr: returned Errcode (0: success <0: fail)(output)
 * RETURN:  BDD for R(G_H, P)
 * ACCESS:  private
 */
bdd CHighSub::r(const bdd &bddP, int &iErr)
{
    try
    {
        bdd bddK = bddP & m_bddInit;
        bdd bddK1 = bddfalse;
        bdd bddK2 = bddfalse;
        bdd bddKNew = bddfalse;
        int iEvent = 0;
        int iIndex = 0;
        unsigned short usiIndex = 0;
        int iSub = 0;
        EVENTSUB EventSub;
        while (bddK != bddK1)
        {
            bddK1 = bddK;
            for (int i = 0; i < this->GetNumofDES(); i++)
            {
                bddK2 = bddfalse;
                while (bddK != bddK2)
                {
                    bddKNew = bddK-bddK2;
                    bddK2 = bddK;
                    for (int j = 0; j < m_pDESArr[i]->GetNumofEvents(); j++)
                    {
                        iEvent = (m_pDESArr[i]->GetEventsArr())[j];
                        pPrj->GenEventInfo(iEvent, EventSub, iSub, usiIndex);
                        if (iEvent % 2 == 0)
                        {
                            //Compute index
                            usiIndex = (usiIndex - 2) / 2;
                            if (EventSub == R_EVENT || EventSub == A_EVENT)
                                iIndex = usiIndex +
                                        m_piUArr[(int)EventSub - 1][iSub - 1];
                            else
                                iIndex = usiIndex;
                            //Compute bddK
                            bddK |= bdd_replace(
                                bdd_appex(m_pbdd_UnConTrans[EventSub][iIndex],
                                          bddKNew, bddop_and,
                                          m_pbdd_UnConVar[EventSub][iIndex]),
                                m_pPair_UnConPrim[EventSub][iIndex]) & bddP;
```

```
                        }
                        else
                        {
                            //Compute Index
                            usiIndex = (usiIndex - 1) / 2;
                            if (EventSub == R_EVENT || EventSub == A_EVENT)
                                iIndex = usiIndex +
                                         m_piCArr[(int)EventSub - 1][iSub - 1];
                            else
                                iIndex = usiIndex;
                            //Compute bddK
                            bddK |= bdd_replace(
                                bdd_appex(m_pbdd_ConTrans[EventSub][iIndex],
                                          bddKNew, bddop_and,
                                          m_pbdd_ConVar[EventSub][iIndex]),
                                m_pPair_ConPrim[EventSub][iIndex]) & bddP;
                        }
                    }
                }
            }
        }
        return bddK;
    }
    catch (...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during computing coreachable.";
        pPrj->SetErr(sErr, HISC_HIGHERR_REACH);
        iErr = -1;
        return bddfalse;
    }
}
```

```
/*************************************************************************
  FILE:  LowSub.h
  DESCR: Header file for LowSub*.cpp (Low-level processing files)
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*************************************************************************/
#ifndef _LSUB_H_
#define _LSUB_H_
#include <string>
#include <map>
#include "type.h"
#include "Sub.h"
#include "fdd.h"
using namespace std;
class CDES;
class CLowSub:public CSub
{
public:
    CLowSub(const string & vsLowFile, int viSubIndex);
    virtual ~CLowSub();
public:
    virtual int PrintSub(ofstream& fout);
    virtual int PrintSubAll(ofstream & fout);
    virtual string SearchEventName(EVENTSUB k, unsigned short usiLocalIndex);
    virtual int LoadSub();
    virtual int SynSuper(const HISC_COMPUTEMETHOD computemethod,
                         HISC_SUPERINFO &superinfo,
                         const HISC_SAVESUPERTYPE savetype,
                         const string& savepath);
    virtual int VeriSub(const HISC_TRACETYPE showtrace,
                         HISC_SUPERINFO & superinfo,
                         const HISC_SAVEPRODUCTTYPE savetype,
                         const string& savepath);
    CDES *GetIntfDES();
private:
    virtual int MakeBdd();
    virtual int InitBddFields();
    virtual int ClearBddFields();
    int CheckIntf();
    int SynPartSuper(const HISC_COMPUTEMETHOD computemethod,
                                        bdd & bddReach, bdd & bddBad);
    int GenConBad(bdd &bddConBad);
    int VeriConBad(bdd &bddConBad, const bdd &bddReach, string & vsErr);
    int GenP4Bad(bdd &bddP4Bad);
    int VeriP4Bad(bdd &bddP4Bad, const bdd &bddReach, string &vsErr);
    int supcp(bdd & bddP);
    bdd cr(const bdd & bddPStart, const bdd & bddP,
                                    const int viEventSub, int & iErr);
    bdd r(const bdd &bddP, int &iErr);
    bdd p5(const bdd& bddP, int &iErr);
    bdd p6(const bdd& bddP, int &iErr);
    void BadStateInfo(const bdd& bddBad, const int viErrCode,
            const HISC_TRACETYPE showtrace, const string &vsExtraInfo = "");
private:
    bdd *m_pbdd_RTrans[2]; //Request event transition predicate in the intf
                           //0: uncon 1: con
    bdd *m_pbdd_ATrans[2];  //Answer event transition predicate in the intf
                           //0: uncon 1: con
    bdd m_bddIVar; //normal interface variables
    bdd m_bddIVarPrim; //prime interface variables
    bdd m_bddIntfInit; //Initial state in the intface DES
    bdd m_bddIntfMarking; //Marker states in the interface DES
};
#endif //_LSUB_H_
```

```
/*************************************************************************
  FILE:  LowSub.cpp
  DESCR: Reading low-level DES files and initialze all the BDDs
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*************************************************************************/
#include "Sub.h"
#include "LowSub.h"
#include <string>
#include "DES.h"
#include "errmsg.h"
#include "type.h"
#include "pubfunc.h"
#include "Project.h"
#include <cassert>
#include <fstream>
using namespace std;
extern CProject *pPrj;
/**
 * DESCR:   Constructor
 * PARA:    vsLowFile: subsystem file name with path (.sub)(input)
 *          viSubIndex: subsystem index (high: 0, low: 1,2,...)(input)
 * RETURN:  None
 * ACCESS:  public
 */
CLowSub::CLowSub(const string & vsLowFile, int viSubIndex):
CSub(vsLowFile, viSubIndex)
{
    m_iNumofIntfs = 1;  //Number of Interface DES, defined in CSub
    InitBddFields();
}
/**
 * DESCR:   Destructor
 * PARA:    None
 * RETURN:  None
 * ACCESS:  public
 */
CLowSub::~CLowSub()
{
    ClearBddFields();
}
/*
 * DESCR:   Initialize BDD related data members (only those in LowSub.h)
 * PARA:    None
 * RETURN:  0
 * ACCESS:  private
 */
int CLowSub::InitBddFields()
{
    m_pbdd_RTrans[0] = NULL;
    m_pbdd_ATrans[0] = NULL;
    m_pbdd_RTrans[1] = NULL;
    m_pbdd_ATrans[1] = NULL;
    m_bddIVar = bddfalse;
    m_bddIVarPrim = bddfalse;
    m_bddIntfInit = bddfalse;
    m_bddIntfMarking = bddfalse;
    return 0;
}
/*
 * DESCR:   Release memory for BDD related data members(only those in Lowsub.h)
 * PARA:    None
 * RETURN:  0
 * ACCESS:  private
 */
int CLowSub::ClearBddFields()
{
    for (int m = 0; m < 2; m++)
    {
        delete[] m_pbdd_RTrans[m];
        m_pbdd_RTrans[m] = NULL;
        delete[] m_pbdd_ATrans[m];
        m_pbdd_ATrans[m] = NULL;
    }
    return 0;
}
/**
 * DESCR:   Load a low-level
 * PARA:    None
 * RETURN:  0 sucess <0 fail;
 * ACCESS:  public
 */
int CLowSub::LoadSub()
```

```
{
    ifstream fin;
    int iRet = 0;
    CDES *pDES = NULL;
    try
    {
        m_sSubFile = str_trim(m_sSubFile);
        if (m_sSubFile.length() <= 4)
        {
            pPrj->SetErr("Invalid file name: " + m_sSubFile, HISC_BAD_LOW_FILE);
            throw -1;
        }
        if (m_sSubFile.substr(m_sSubFile.length() - 4) != ".sub")
        {
            pPrj->SetErr("Invalid file name: " + m_sSubFile, HISC_BAD_LOW_FILE);
            throw -1;
        }
        fin.open(m_sSubFile.data(), ifstream::in);
        if (!fin) //unable to find low sub file
        {
            pPrj->SetErr("Unable to open file: " + m_sSubFile,
                         HISC_BAD_LOW_FILE);
            throw -1;
        }
        m_sSubName = GetNameFromFile(m_sSubFile);
        char scBuf[MAX_LINE_LENGTH];
        string sLine;
        int iField = -1; //0: SYSTEM 1:INTERFACE 2:PLANT 3:SPEC
        char *scFieldArr[] = {"SYSTEM", "INTERFACE", "PLANT", "SPEC"};
        string sDESFile;
        int iTmp = 0;
        int iNumofPlants = 0;
        int iNumofSpecs = 0;
        int iNumofIntfs = 0;
        while (fin.getline(scBuf,  MAX_LINE_LENGTH))
        {
            sLine = str_nocomment(scBuf);
            sLine = str_trim(sLine);
            if (sLine.empty())
                continue;
            if (sLine[0] == '[' && sLine[sLine.length() - 1] == ']')
            {
                sLine = sLine.substr(1, sLine.length() - 1);
                sLine = sLine.substr(0, sLine.length() - 1);
                sLine = str_upper(str_trim(sLine));
                iField++;
                if (iField <= 3)
                {
                    if (sLine != scFieldArr[iField])
                    {
                        pPrj->SetErr(m_sSubName +
                                     ": Field name or order is wrong!",
                                      HISC_BAD_LOW_FORMAT);
                        throw -1;
                    }
                    if (iField == 1)
                    {
                        //Check number of Plants and apply for memory space
                        if (m_iNumofPlants + m_iNumofSpecs <= 0)
                        {
                            pPrj->SetErr(m_sSubName +
                                    ": Must have at least one DES.",
                                    HISC_BAD_LOW_FORMAT);
                            throw -1;
                        }
                        if (m_iNumofPlants < 0 || m_iNumofSpecs < 0)
                        {
                            pPrj->SetErr(m_sSubName +
                             ": Must specify the number of plant DES and spec DES.",
                                     HISC_BAD_LOW_FORMAT);
                            throw -1;
                        }
                        m_pDESArr = new (CDES *)[this->GetNumofDES()];
                        for (int i = 0; i < this->GetNumofDES(); i++)
                            m_pDESArr[i] = NULL;
                        //Initialize m_piDESOrderArr
                        m_piDESOrderArr = new int[this->GetNumofDES()];
                        for (int i = 0; i < this->GetNumofDES(); i++)
```

```
                m_piDESOrderArr[i] = i;
            //Initialize m_piDESPosArr
            m_piDESPosArr = new int[this->GetNumofDES()];
            for (int i = 0; i < this->GetNumofDES(); i++)
                m_piDESPosArr[i] = i;
        }
    }
    else
    {
        pPrj->SetErr(m_sSubName + ": Too many fields!",
                HISC_BAD_LOW_FORMAT);
        throw -1;
    }
}
else
{
    switch (iField)
    {
    case 0:  //[SYSTEM]
        if (!IsInteger(sLine))
        {
            pPrj->SetErr(m_sSubName + ": Number of DES is absent!",
                        HISC_BAD_LOW_FORMAT);
            throw -1;
        }
        iTmp = atoi(sLine.data());
        if (iTmp < 0)
        {
            pPrj->SetErr(m_sSubName +
                         ": Number of DES is less than zero!",
                    HISC_BAD_LOW_FORMAT);
            throw -1;
        }
        if (m_iNumofPlants < 0)
            m_iNumofPlants = iTmp;
        else if (m_iNumofSpecs < 0)
            m_iNumofSpecs = iTmp;
        else
        {
            pPrj->SetErr(m_sSubName +
                    ": Too many lines in SYSTEM field",
                    HISC_BAD_LOW_FORMAT);
            throw -1;
        }
        break;
    case 1: //[INTERFACE]
        if (iNumofIntfs > 0)
        {
            pPrj->SetErr(m_sSubName +
            ": Only one interface DES is allowed or each low sub",
                HISC_BAD_LOW_FORMAT);
            throw -1;
        }
        sDESFile = GetDESFileFromSubFile(m_sSubFile, sLine);
        pDES = new CDES(this, sDESFile, INTERFACE_DES);
        if (pDES->LoadDES() < 0)
            throw -1;   //here LoadDES() will generate the err msg.
        else
        {
            iNumofIntfs++;
            m_pDESArr[0] = pDES;
            pDES = NULL;
        }
        break;
    case 2: //[PLANT]
        sDESFile = GetDESFileFromSubFile(m_sSubFile, sLine);
        pDES = new CDES(this, sDESFile, PLANT_DES);
        if (pDES->LoadDES() < 0)
            throw -1;   //here LoadDES() will generate the err msg.
        else
        {
            iNumofPlants++;
            if (iNumofPlants > m_iNumofPlants)
            {
                pPrj->SetErr(m_sSubName + ": Too many Plant DESs",
                        HISC_BAD_LOW_FORMAT);
                throw -1;
            }
            m_pDESArr[iNumofPlants] = pDES;
            pDES = NULL;
        }
```

```
                    break;
                case 3: //[SPEC]
                    sDESFile = GetDESFileFromSubFile(m_sSubFile, sLine);
                    pDES = new CDES(this, sDESFile, SPEC_DES);
                    if (pDES->LoadDES() < 0)
                        throw -1;   //here LoadDES() will generate the err msg.
                    else
                    {
                        iNumofSpecs++;
                        if (iNumofSpecs > m_iNumofSpecs)
                        {
                            pPrj->SetErr(m_sSubName + ": Too many spec DESs",
                                    HISC_BAD_LOW_FORMAT);
                            throw -1;
                        }
                        m_pDESArr[m_iNumofPlants + iNumofSpecs] = pDES;
                        pDES = NULL;
                    }
                    break;
                default:
                    pPrj->SetErr(m_sSubName + ": Unknown error.",
                                HISC_BAD_LOW_FORMAT);
                    throw -1;
                    break;
                }
            }
        } //while
        if (iNumofPlants < m_iNumofPlants)
        {
            pPrj->SetErr(m_sSubName + ": Too few plant DESs",
                        HISC_BAD_LOW_FORMAT);
            throw -1;
        }
        if (iNumofSpecs < m_iNumofSpecs)
        {
            pPrj->SetErr(m_sSubName + ": Too few spec DESs",
                    HISC_BAD_LOW_FORMAT);
            throw -1;
        }
        if (iNumofIntfs < m_iNumofIntfs)
        {
            pPrj->SetErr(m_sSubName + ": There must exists one interface DES.",
                    HISC_BAD_LOW_FORMAT);
            throw -1;
        }
        fin.close();
        this->DESReorder();
    }
    catch (int iError)
    {
        if (pDES != NULL)
        {
            delete pDES;
            pDES = NULL;
        }
        if (fin.is_open())
            fin.close();
        iRet = iError;
    }
    return iRet;
}
/*
 * DESCR:   Initialize BDD data memebers
 * PARA:    None
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
 */
int CLowSub::MakeBdd()
{
    try
    {
        //Initialize the bdd node table and cache size.
        long long lNumofStates = 1;
        for (int i = 0; i < this->GetNumofDES(); i++)
        {
            lNumofStates *= m_pDESArr[i]->GetNumofStates();
            if (lNumofStates >= MAX_INT)
                break;
        }
        if (lNumofStates <= 10000)
            bdd_init(1000, 100);
```

```
else if (lNumofStates <= 1000000)
    bdd_init(10000, 1000);
else if (lNumofStates <= 10000000)
    bdd_init(100000, 10000);
else
{
    bdd_init(2000000, 1000000);
    bdd_setmaxincrease(1000000);
}
giNumofBddNodes = 0;
bdd_gbc_hook(my_bdd_gbchandler);
//define domain variables
int *piDomainArr = new int[2];
for (int i = 0; i < 2 * this->GetNumofDES(); i += 2)
{
    piDomainArr[0] = m_pDESArr[i/2]->GetNumofStates();
    piDomainArr[1] = piDomainArr[0];
    fdd_extdomain(piDomainArr, 2);
}
delete[] piDomainArr;
piDomainArr = NULL;
//compute the number of bdd variables (only for normal variables)
m_iNumofBddNormVar = 0;
for (int i = 0; i < 2 * (this->GetNumofDES()); i = i + 2)
    m_iNumofBddNormVar += fdd_varnum(i);
//compute initial state predicate
for (int i = 0; i < this->GetNumofDES(); i++)
{
    m_bddInit &= fdd_ithvar(i * 2, m_pDESArr[i]->GetInitState());
    if (m_pDESArr[i]->GetDESType() == INTERFACE_DES)
        m_bddIntfInit = fdd_ithvar(i * 2, m_pDESArr[i]->GetInitState());
}
//set the first level block
int iNumofBddVar = 0;
int iVarNum = 0;
bdd bddBlock = bddtrue;
for (int i = 0; i < 2 * (this->GetNumofDES()); i += 2)
{
    iVarNum = fdd_varnum(i);
    bddBlock = bddtrue;
    for (int j = 0; j < 2 * iVarNum; j++)
    {
        bddBlock &= bdd_ithvar(iNumofBddVar + j);
    }
    bdd_addvarblock(bddBlock, BDD_REORDER_FREE);
    iNumofBddVar += 2 * iVarNum;
}
//compute marking states predicate
bdd bddTmp = bddfalse;
for (int i = 0; i < this->GetNumofDES(); i++)
{
    bddTmp = bddfalse;
    MARKINGLIST::const_iterator ci =
         (m_pDESArr[i]->GetMarkingList()).begin();
    for (int j = 0; j < m_pDESArr[i]->GetNumofMarkingStates(); j++)
    {
        bddTmp |= fdd_ithvar(i * 2, *ci);
        ++ci;
    }
    m_bddMarking &= bddTmp;
    if (m_pDESArr[i]->GetDESType() == INTERFACE_DES)
        m_bddIntfMarking = bddTmp;
}
//Initialize interface transitions predicates
if (m_usiMaxUnCon[R_EVENT] > 0)
    m_pbdd_RTrans[0] = new bdd[m_usiMaxUnCon[R_EVENT] / 2];
if (m_usiMaxUnCon[A_EVENT] > 0)
    m_pbdd_ATrans[0] = new bdd[m_usiMaxUnCon[A_EVENT]/2];
if ((unsigned short)(m_usiMaxCon[R_EVENT] + 1) > 0)
    m_pbdd_RTrans[1] = new bdd[(m_usiMaxCon[R_EVENT] + 1)/2];
if ((unsigned short)(m_usiMaxCon[A_EVENT] + 1) > 0)
    m_pbdd_ATrans[1] = new bdd[(m_usiMaxCon[A_EVENT] + 1)/2];
//Compute transitions predicate
for (int k = 1; k < 4; k++)
{
    if (m_usiMaxCon[k] != 0xFFFF)
    {
        m_pbdd_ConTrans[k] = new bdd[m_usiMaxCon[k] / 2 + 1](bddfalse);
```

```
        m_pbdd_ConVar[k] = new bdd[m_usiMaxCon[k] / 2 + 1](bddfalse);
        m_pbdd_ConVarPrim[k] =
                        new bdd[m_usiMaxCon[k] / 2 + 1](bddfalse);
        m_pbdd_ConPhysicVar[k] =
                        new bdd[m_usiMaxCon[k] / 2 + 1](bddfalse);
        m_pbdd_ConPhysicVarPrim[k] =
                        new bdd[m_usiMaxCon[k] / 2 + 1](bddfalse);
        m_pPair_Con[k] = new (bddPair *)[m_usiMaxCon[k] / 2 + 1];
        for (int iPair = 0; iPair < m_usiMaxCon[k] / 2 + 1; iPair++)
            m_pPair_Con[k][iPair] = NULL;
        m_pPair_ConPrim[k] = new (bddPair *)[m_usiMaxCon[k] / 2 + 1];
        for (int iPair = 0; iPair < m_usiMaxCon[k] / 2 + 1; iPair++)
            m_pPair_ConPrim[k][iPair] = NULL;
    }
    if (m_usiMaxUnCon[k] != 0)
    {
        m_pbdd_UnConTrans[k] = new bdd[m_usiMaxUnCon[k]/2](bddfalse);
        m_pbdd_UnConVar[k] = new bdd[m_usiMaxUnCon[k]/2](bddfalse);
        m_pbdd_UnConPlantTrans[k] =
                             new bdd[m_usiMaxUnCon[k]/2](bddfalse);
        m_pbdd_UnConVarPrim[k] = new bdd[m_usiMaxUnCon[k]/2](bddfalse);
        m_pbdd_UnConPlantVar[k] = new bdd[m_usiMaxUnCon[k]/2](bddfalse);
        m_pbdd_UnConPlantVarPrim[k] =
                             new bdd[m_usiMaxUnCon[k]/2](bddfalse);
        m_pPair_UnCon[k] = new (bddPair *)[m_usiMaxUnCon[k]/2];
        for (int iPair = 0; iPair < m_usiMaxUnCon[k]/2; iPair++)
            m_pPair_UnCon[k][iPair] = NULL;
        m_pPair_UnConPrim[k] = new (bddPair *)[m_usiMaxUnCon[k]/2];
        for (int iPair = 0; iPair < m_usiMaxUnCon[k]/2; iPair++)
            m_pPair_UnConPrim[k][iPair] = NULL;
    }
}
map<int, bdd> bddTmpTransMap;  //<event_index, transitions>
for (int i = 0; i < this->GetNumofDES(); i++)
{
    //before compute transition predicate for each DES, clear it.
    bddTmpTransMap.clear();
    for (int j = 0; j < m_pDESArr[i]->GetNumofEvents(); j++)
    {
        bddTmpTransMap[(m_pDESArr[i]->GetEventsArr())[j]] = bddfalse;
    }
    //compute transition predicate for each DES
    for (int j = 0; j < m_pDESArr[i]->GetNumofStates(); j++)
    {
        TRANS::const_iterator ci =
                            (*(m_pDESArr[i]->GetTrans() + j)).begin();
        for (; ci != (*(m_pDESArr[i]->GetTrans() + j)).end(); ++ci)
        {
            bddTmpTransMap[ci->first] |= fdd_ithvar(i * 2, j) &
                fdd_ithvar(i * 2 + 1, ci->second);
        }
    }
    //combine the current DES transition predicate to
    //subsystem transition predicate
    map<int, bdd>::const_iterator ciTmp = bddTmpTransMap.begin();
    for (; ciTmp != bddTmpTransMap.end(); ++ciTmp)
    {
        if (ciTmp->first % 2 == 0)  //uncontrollable, start from 2
        {
            int iEventSub = ciTmp->first >> 28;
            int iIndex = (ciTmp->first & 0x0000FFFF) / 2 - 1;
            if (m_pbdd_UnConVar[iEventSub][iIndex] == bddfalse)
            {
                m_pbdd_UnConTrans[iEventSub][iIndex] = bddtrue;
                m_pbdd_UnConVar[iEventSub][iIndex] = bddtrue;
                m_pbdd_UnConVarPrim[iEventSub][iIndex] = bddtrue;
            }
            m_pbdd_UnConTrans[iEventSub][iIndex] &= ciTmp->second;
            m_pbdd_UnConVar[iEventSub][iIndex] &= fdd_ithset(i * 2);
            m_pbdd_UnConVarPrim[iEventSub][iIndex] &=
                                            fdd_ithset(i * 2 + 1);
            //compute uncontrollable plant vars and varprimes
            if (m_pDESArr[i]->GetDESType() == PLANT_DES)
            {
                if (m_pbdd_UnConPlantVar[iEventSub][iIndex] == bddfalse)
                {
```

```
                m_pbdd_UnConPlantTrans[iEventSub][iIndex] = bddtrue;
                m_pbdd_UnConPlantVar[iEventSub][iIndex] = bddtrue;
                m_pbdd_UnConPlantVarPrim[iEventSub][iIndex] =
                                                        bddtrue;
            }
            m_pbdd_UnConPlantTrans[iEventSub][iIndex] &=
                                                    ciTmp->second;
            m_pbdd_UnConPlantVar[iEventSub][iIndex] &=
                                                fdd_ithset(i * 2);
            m_pbdd_UnConPlantVarPrim[iEventSub][iIndex] &=
                                            fdd_ithset(i * 2 + 1);
        }
        //compute interface Transitions
        if (m_pDESArr[i]->GetDESType() == INTERFACE_DES)
        {
            if ((EVENTSUB)iEventSub == R_EVENT)
                m_pbdd_RTrans[0][iIndex] = ciTmp->second;
            else
                m_pbdd_ATrans[0][iIndex] = ciTmp->second;
        }
    }
    else  //controllable
    {
        int iEventSub = ciTmp->first >> 28;
        int iIndex = ((ciTmp->first & 0x0000FFFF) - 1)/ 2;
        if (m_pbdd_ConVar[iEventSub][iIndex] == bddfalse)
        {
            m_pbdd_ConTrans[iEventSub][iIndex] = bddtrue;
            m_pbdd_ConVar[iEventSub][iIndex] = bddtrue;
            m_pbdd_ConVarPrim[iEventSub][iIndex] = bddtrue;
        }
        m_pbdd_ConTrans[iEventSub][iIndex] &= ciTmp->second;
        m_pbdd_ConVar[iEventSub][iIndex] &= fdd_ithset(i * 2);
        m_pbdd_ConVarPrim[iEventSub][iIndex] &=
                                            fdd_ithset(i * 2 + 1);
        //compute controllable physical plant vars and varprimes
        if (m_pDESArr[i]->GetDESType() == PLANT_DES)
        {
            if (m_pbdd_ConPhysicVar[iEventSub][iIndex] == bddfalse)
            {
                m_pbdd_ConPhysicVar[iEventSub][iIndex] = bddtrue;
                m_pbdd_ConPhysicVarPrim[iEventSub][iIndex]= bddtrue;
            }
            m_pbdd_ConPhysicVar[iEventSub][iIndex] &=
                                                fdd_ithset(i * 2);
            m_pbdd_ConPhysicVarPrim[iEventSub][iIndex] &=
                                            fdd_ithset(i * 2 + 1);
        }
        //compute interface Transitions
        if (m_pDESArr[i]->GetDESType() == INTERFACE_DES)
        {
            if ((EVENTSUB)iEventSub == R_EVENT)
                m_pbdd_RTrans[1][iIndex] = ciTmp->second;
            else
                m_pbdd_ATrans[1][iIndex] = ciTmp->second;
        }
    }
}
if (m_pDESArr[i]->GetDESType() == INTERFACE_DES)
{
    m_bddIVar = fdd_ithset(i * 2);
    m_bddIVarPrim = fdd_ithset(i * 2 + 1);
}
}
//compute m_pPair_UnCon, m_pPair_Con
for (int k = 1; k < 4; k++)
{
    for (int j = 0; j < m_usiMaxUnCon[k]; j += 2)
    {
        m_pPair_UnCon[k][j/2] = bdd_newpair();
        SetBddPairs(m_pPair_UnCon[k][j/2], m_pbdd_UnConVar[k][j/2],
                    m_pbdd_UnConVarPrim[k][j/2]);
        m_pPair_UnConPrim[k][j/2] = bdd_newpair();
        SetBddPairs(m_pPair_UnConPrim[k][j/2],
                    m_pbdd_UnConVarPrim[k][j/2],
                    m_pbdd_UnConVar[k][j/2]);
    }
```

```
            for (int j = 1; j < (unsigned short)(m_usiMaxCon[k] + 1); j += 2)
            {
                m_pPair_Con[k][(j - 1) / 2] = bdd_newpair();
                SetBddPairs(m_pPair_Con[k][(j - 1) / 2],
                                m_pbdd_ConVar[k][(j - 1) / 2],
                                m_pbdd_ConVarPrim[k][(j - 1) / 2]);
                m_pPair_ConPrim[k][(j - 1) / 2] = bdd_newpair();
                SetBddPairs(m_pPair_ConPrim[k][(j - 1) / 2],
                                m_pbdd_ConVarPrim[k][(j - 1) / 2],
                                m_pbdd_ConVar[k][(j - 1) / 2]);
            }
        }
    }
    catch(...)
    {
        string sErr;
        sErr = "Error happens when initializing low level ";
        sErr += this->GetSubIndex();
        sErr += " BDD!";
        pPrj->SetErr(sErr, HISC_SYSTEM_INITBDD);
        return -1;
    }
    return 0;
}
```

```
/*************************************************************************
  FILE:  LowSub1.cpp
  DESCR: Some misc functions for low-levels
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*************************************************************************/
#include "Sub.h"
#include "LowSub.h"
#include <string>
#include "DES.h"
#include "errmsg.h"
#include "type.h"
#include "pubfunc.h"
#include "Project.h"
#include <cassert>
#include <fstream>
#include <fdd.h>
using namespace std;
extern CProject *pPrj;
/**
 * DESCR:   Save DES list of low-levels in memory to a file (for checking)
 * PARA:    fout: output file stream
 * RETURN:  0: sucess -1: fail
 * ACCESS:  public
 */
int CLowSub::PrintSub(ofstream& fout)
{
    try
    {
        fout << "#Sub system: " <<  m_sSubName << endl;
        fout << endl;
        fout << "[SYSTEM]" << endl;
        fout << m_iNumofPlants << endl;
        fout << m_iNumofSpecs << endl;
        fout << endl;
        fout << "[INTERFACE]" << endl;
        for (int i = 0; i < this->GetNumofDES(); i++)
        {
            if (m_pDESArr[i]->GetDESType() == INTERFACE_DES)
            {
                fout << m_pDESArr[0]->GetDESName() << endl;
                break;
            }
        }
        fout << "[PLANT]" << endl;
        for (int i = 1; i < m_iNumofPlants + m_iNumofIntfs; i++)
        {
            for (int j = 0; j < this->GetNumofDES(); j++)
            {
                if (m_piDESOrderArr[j] == i)
                {
                    fout << m_pDESArr[j]->GetDESName() << endl;
                    break;
                }
            }
        }
        fout << "[SPEC]" << endl;
        for (int i = m_iNumofPlants + m_iNumofIntfs;
                i < this->GetNumofDES(); i++)
        {
            for (int j = 0; j < this->GetNumofDES(); j++)
            {
                if (m_piDESOrderArr[j] == i)
                {
                    fout << m_pDESArr[j]->GetDESName() << endl;
                    break;
                }
            }
        }
        fout << "################################################" << endl;
    }
    catch(...)
    {
        return -1;
    }
    return 0;
}
/**
 * DESCR:   Save all the DES in low-levels to a text file for checking
 * PARA:    fout: output file stream
 * RETURN:  0: sucess -1: fail
 * ACCESS:  public
```

```
 */
int CLowSub::PrintSubAll(ofstream & fout)
{
    try
    {
        if (PrintSub(fout) < 0)
            throw -1;
        for (int i = 0; i < this->GetNumofDES(); i++)
        {
            if (m_pDESArr[i]->PrintDES(fout) < 0)
                throw -1;
        }
    }
    catch(...)
    {
        return -1;
    }
    return 0;
}
/**
 * DESCR:   Get the interface DES pointer for this low-level,used by high-level.
 * PARA:    None
 * RETURN:  This low-level interface DES pointer
 * ACCESS:  public
 */
CDES * CLowSub::GetIntfDES()
{
    assert(m_pDESArr != NULL);
    for (int i = 0; i < this->GetNumofDES(); i++)
    {
        if (m_pDESArr[i]->GetDESType() == INTERFACE_DES)
            return m_pDESArr[i];
    }
    assert(false); //should never come here
    return m_pDESArr[0];
}
/*
 * DESCR: Generate Bad state info during verfication
 *        Note: showtrace is not implemented, currently it is used for showing
 *              a blocking is a deadlock or livelock (very slow).
 * PARA:    bddBad: BDD for the set of bad states
 *          viErrCode: error code (see errmsg.h)
 *          showtrace: show a trace from the initial state to a bad state or not
 *                     (not implemented)
 *          vsExtraInfo: Extra errmsg.
 * RETURN:  None
 * ACCESS:  private
 */
void CLowSub::BadStateInfo(const bdd& bddBad, const int viErrCode,
                    const HISC_TRACETYPE showtrace, const string &vsExtraInfo)
{
    int *piBad = fdd_scanallvar(bddBad);
    string sErr;
    if (piBad != NULL)
    {
        if (viErrCode == HISC_VERI_LOW_UNCON)
            sErr = GetSubName() +
                        ": Level-wise controllable checking failed state:\n< ";
        else if (viErrCode == HISC_VERI_LOW_BLOCKING)
        {
            if (showtrace == HISC_SHOW_TRACE)
                sErr = GetSubName() + ": Blocking state ";
            else
                sErr = GetSubName() + ": Blocking state: \n < ";
        }
        else if (viErrCode == HISC_VERI_LOW_P4FAILED)
            sErr = GetSubName() +
        ": Interface consistent conditions Point 4 checking failed state:\n< ";
        else if (viErrCode == HISC_VERI_LOW_P5FAILED)
            sErr = GetSubName() +
        ": Interface consistent conditions Point 5 checking failed state:\n< ";
        else if (viErrCode == HISC_VERI_LOW_P6FAILED)
            sErr = GetSubName() +
        ": Interface consistent conditions Point 6 checking failed state:\n< ";
        //for blocking state, try to find the deadlock state
        //if there is no deadlock state, only show one of the live lock states
        if (showtrace == HISC_SHOW_TRACE)
        {
            if (viErrCode == HISC_VERI_LOW_BLOCKING)
            {
                bdd bddBlock = bddBad;
```

```
                bdd bddNext = bddtrue;
                bdd bddTemp = bddtrue;
                do
                {
                    bddTemp = bddtrue;
                    for (int i = 0; i < this->GetNumofDES(); i++)
                        bddTemp &= fdd_ithvar(i * 2, piBad[i * 2]);
                    bddNext = bddfalse;
                    for (int k = 0; k < 4; k++)
                    {
                        for (unsigned short usi = 2;
                                usi <= m_usiMaxUnCon[k] && bddNext == bddfalse;
                                usi += 2)
                        {
                            bddNext |=
                                bdd_replace(
                                    bdd_relprod(
                                        m_pbdd_UnConTrans[k][(usi - 2) / 2],
                                        bddTemp,
                                        m_pbdd_UnConVar[k][(usi - 2) / 2]),
                                        m_pPair_UnConPrim[k][(usi - 2) / 2]) &
                                bddBad;
                        }
                        for (unsigned short usi = 1;
                            usi < (unsigned short) (m_usiMaxCon[k] + 1) &&
                            bddNext == bddfalse; usi += 2)
                        {
                            bddNext |=
                                bdd_replace(
                                    bdd_relprod(
                                        m_pbdd_ConTrans[k][(usi - 1) / 2],
                                        bddTemp,
                                        m_pbdd_ConVar[k][(usi - 1) / 2]),
                                        m_pPair_ConPrim[k][(usi - 1) / 2]) &
                                bddBad;
                        }
                    }
                    if (bddNext == bddfalse)  //this is a deadlock state
                    {
                        sErr += "(dead lock):\n< ";
                        break;
                    }
                    else //not a deadlock state
                    {
                        bddBlock = bddBlock - bddTemp;
                        free(piBad);
                        piBad = NULL;
                        piBad = fdd_scanallvar(bddBlock);
                    }
                } while (piBad != NULL);
                if (piBad == NULL) //live lock
                {
                    sErr += "(live lock):\n< ";
                    piBad = fdd_scanallvar(bddBad);
                }
            }
        }
        if (piBad != NULL)
        {
            for (int i = 0; i < this->GetNumofDES(); i++)
            {
                sErr += m_pDESArr[m_piDESPosArr[i]]->GetDESName() + ":" +
                        m_pDESArr[m_piDESPosArr[i]]->GetStateName(
                                        piBad[m_piDESPosArr[i] * 2]);
                if (i < this->GetNumofDES() -1)
                    sErr += ", ";
            }
        }
        sErr += " >\n";
        sErr += vsExtraInfo;
        pPrj->SetErr(sErr, viErrCode);
        free(piBad);
        piBad = NULL;
    }
    return;
}
/**
 * DESCR:   Search event name from this low-level local event index.
 * PARA:    k: R_EVENT/A_EVENT/H_EVENT/L_EVENT
```

```
 *          usiLocalIndex: this low-level local event index.
 * RETURN:  event name
 * ACCESS:  public
 */
string CLowSub::SearchEventName(EVENTSUB k, unsigned short usiLocalIndex)
{
    int iEventIndex = 0;
    iEventIndex = pPrj->GenEventIndex((EVENTSUB)k, m_iSubIndex, usiLocalIndex);
    return (pPrj->GetInvAllEventsMap())[iEventIndex];
}
```

```
/*****************************************************************************
  FILE:  LowSub2.cpp
  DESCR: Verify this low-level interface (Command-pair)
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*****************************************************************************/
#include "LowSub.h"
#include "Sub.h"
#include "pubfunc.h"
#include "type.h"
#include "fdd.h"
#include "Project.h"
#include "errmsg.h"
#include <string>
using namespace std;
extern CProject *pPrj;
/*
 * DESCR:   Verify if this low-level interface is a command-pair interface
 * PARA:    None
 * RETURN:  0: This low-level interface is a command-pair interface
 *          < 0: Not a command-pair interface
 * ACCESS: private
 */
int CLowSub::CheckIntf()
{
    int iRet = 0;
    string sErr = "Interface for low level ";
    sErr += this->GetSubName();
    sErr += ": ";
    bddPair *pPair = bdd_newpair();
    SetBddPairs(pPair, m_bddIVarPrim, m_bddIVar);
    bdd P_Ij = bddfalse;
    bdd P_Rj = bddfalse;
    bdd P_Aj = bddfalse;
    bdd P_RAj = bddfalse;
    bdd P_ARj = bddfalse;
    bdd P_Ijm = bddfalse;
    try
    {
        if (m_bddIntfInit == bddfalse)
        {
            pPrj->SetErr(sErr + "No reachable states.", HISC_BAD_INTERFACE);
            throw -1;
        }
        P_Ij = intf_r(bddtrue, iRet);
        if (iRet < 0)
            throw -1;
        P_Ijm = m_bddIntfMarking & P_Ij;
        for (int i = 1; i <= (unsigned short)(m_usiMaxCon[A_EVENT] + 1); i += 2)
            P_Rj |= bdd_exist(m_pbdd_ATrans[1][(i - 1)/2], m_bddIVar);
        for (int i = 2; i <= (unsigned short)(m_usiMaxUnCon[A_EVENT] + 1); i += 2)
            P_Rj |= bdd_exist(m_pbdd_ATrans[0][(i - 2)/2], m_bddIVar);
        P_Rj = bdd_replace(P_Rj, pPair);
        P_Rj &= P_Ij;
        P_Rj |= m_bddIntfInit;
        for (int i = 1; i <= (unsigned short)(m_usiMaxCon[R_EVENT] + 1); i += 2)
            P_Aj |= bdd_exist(m_pbdd_RTrans[1][(i - 1)/2], m_bddIVar);
        for (int i = 2; i <= (unsigned short)(m_usiMaxUnCon[R_EVENT]); i += 2)
            P_Aj |= bdd_exist(m_pbdd_RTrans[0][(i - 2)/2], m_bddIVar);
        P_Aj = bdd_replace(P_Aj, pPair);
        P_Aj &= P_Ij;
        for (int i = 1; i <= (unsigned short)(m_usiMaxCon[A_EVENT] + 1); i += 2)
            P_RAj |= bdd_exist(m_pbdd_ATrans[1][(i - 1)/2] & P_Rj, m_bddIVar);
        for (int i = 2; i <= (unsigned short)(m_usiMaxUnCon[A_EVENT] + 1); i += 2)
            P_RAj |= bdd_exist(m_pbdd_ATrans[0][(i - 2)/2] & P_Rj, m_bddIVar);
        for (int i = 1; i <= (unsigned short)(m_usiMaxCon[R_EVENT] + 1); i += 2)
            P_ARj |= bdd_exist(m_pbdd_RTrans[1][(i - 1)/2] & P_Aj, m_bddIVar);
        for (int i = 2; i <= (unsigned short)(m_usiMaxUnCon[R_EVENT]); i += 2)
            P_ARj |= bdd_exist(m_pbdd_RTrans[0][(i - 2)/2] & P_Aj, m_bddIVar);
        if (P_RAj != bddfalse || P_ARj != bddfalse || P_Rj != P_Ijm)
        {
            pPrj->SetErr(sErr + "Not a command-pair interface.", HISC_BAD_INTERFACE);
            throw -1;
        }
    }
    catch(int iErr)
    {
        iRet = iErr;
```

```
    }
    bdd_freepair(pPair);
    pPair = NULL;
    return iRet;
}
/*
 * DESCR:   Compute R(GIj, P)
 * PARA:    bddP: P (input)
            iErr: returned error code
 * RETURN:  R(GIj, P)
 * ACCESS: private
 */
bdd CLowSub::intf_r(const bdd &bddP, int &iErr)
{
    bdd bddK = bddP & m_bddIntfInit;
    bdd bddK1 = bddfalse;
    bdd bddKStep = bddfalse;
    bddPair *pPair = bdd_newpair();
    SetBddPairs(pPair, m_bddIVarPrim, m_bddIVar);
    iErr = 0;
    try
    {
        while (bddK != bddK1)
        {
            bddK1 = bddK;
            for (int i = 1; i <= (unsigned short)(m_usiMaxCon[R_EVENT] + 1); i += 2)
            {
                bddKStep = bdd_exist(m_pbdd_RTrans[1][(i - 1)/2] & bddK1, m_bddIVar);
                bddKStep = bdd_replace(bddKStep, pPair);
                bddK |= bddKStep;
            }
            for (int i = 2; i <= (unsigned short)(m_usiMaxUnCon[R_EVENT]); i += 2)
            {
                bddKStep = bdd_exist(m_pbdd_RTrans[0][(i - 2)/2] & bddK1, m_bddIVar);
                bddKStep = bdd_replace(bddKStep, pPair);
                bddK |= bddKStep;
            }
            for (int i = 1; i <= (unsigned short)(m_usiMaxCon[A_EVENT] + 1); i += 2)
            {
                bddKStep = bdd_exist(m_pbdd_ATrans[1][(i - 1)/2] & bddK1, m_bddIVar);
                bddKStep = bdd_replace(bddKStep, pPair);
                bddK |= bddKStep;
            }
            for (int i = 2; i <= (unsigned short)(m_usiMaxUnCon[A_EVENT] + 1); i += 2)
            {
                bddKStep = bdd_exist(m_pbdd_ATrans[0][(i - 2)/2] & bddK1, m_bddIVar);
                bddKStep = bdd_replace(bddKStep, pPair);
                bddK |= bddKStep;
            }
        }
    }
    catch(int)
    {
        string sErr = "Interface for low level ";
        sErr += this->GetSubName();
        sErr += ": Error during computing interface reachable states.";
        pPrj->SetErr(sErr, HISC_LOWERR_REACH);
        iErr = -1;
        bdd_freepair(pPair);
        pPair = NULL;
        return bddfalse;
    }
    bdd_freepair(pPair);
    pPair = NULL;
    return bddK;
}
```

```
/*****************************************************************************
  FILE:  LowSub3.cpp
  DESCR: Verification and synthesis for low-levels
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*****************************************************************************/
#include "LowSub.h"
#include "Sub.h"
#include "pubfunc.h"
#include "type.h"
#include "fdd.h"
#include "Project.h"
#include "errmsg.h"
#include <string>
#include <sys/time.h>
using namespace std;
extern CProject *pPrj;
/**
 * DESCR:   Verify whether a high level is level-wise controllable,
 *          nonblocking and  p4, p5 and p6 satisfied or not.
 * PARA:    showtrace:show a trace to a bad state or not(not implemented)(input)
 *          superinfo: returned verification info (see BddHisc.h)(output)
 *          savetype: save syn-product or not (See BddHisc.h)(input)
 *          savepath: where to save syn-product(input)
 * RETURN:  0: sucess <0: fail
 * ACCESS:  public
 */
int CLowSub::VeriSub(const HISC_TRACETYPE showtrace, HISC_SUPERINFO & superinfo,
                     const HISC_SAVEPRODUCTTYPE savetype, const string& savepath)
{
    int iRet = 0;
    int iErr = 0;
    //Initialize the BDD data memebers
    CSub::InitBddFields();
    InitBddFields();
    bdd bddReach = bddfalse;
    string sErr;
    try
    {
        //Make transition bdds
        if (MakeBdd() < 0)
            throw -1;
        //Check command pair interface
        if (CheckIntf() < 0)
            throw -1;
        bdd bddConBad = bddfalse;
        bdd bddP4Bad = bddfalse;
        bdd bddP5 = bddfalse;
        bdd bddP6 = bddfalse;
        bdd bddCoreach = bddfalse;
        //compute bddReach
        bddReach = r(bddtrue, iErr);
        if (iErr < 0)
            throw -1;
        m_bddMarking &= bddReach;
        //Initialize controllable bddBad
        bddConBad = bddfalse;
        if (VeriConBad(bddConBad, bddReach, sErr) < 0)
            throw -1;
        //check if any reachable states belong to bad states
        if (bddConBad != bddfalse)
        {
            BadStateInfo(bddConBad, HISC_VERI_LOW_UNCON, showtrace, sErr);
            throw -2;
        }
        //Initialize P4 bddBad
        if (VeriP4Bad(bddP4Bad, bddReach, sErr) < 0)
            throw -1;
        //check if any reachable states belong to bad states
        if (bddP4Bad != bddfalse)
        {
            BadStateInfo(bddP4Bad, HISC_VERI_LOW_P4FAILED, showtrace, sErr);
            throw -3;
        }
        bddCoreach = cr(m_bddMarking, bddReach, ALL_EVENT, iErr);
        if (iErr != 0)
            throw -1;
        //check if the system is nonblocking
        if (bddReach != bddCoreach)
        {
```

```
            BadStateInfo(bddReach - bddCoreach, HISC_VERI_LOW_BLOCKING, showtrace);
            throw -4;
        }
        bddP6 = p6(bddReach, iErr);
        if (iErr != 0)
            throw -1;
        if (bddP6 != bddReach)
        {
            BadStateInfo(bddReach - bddP6, HISC_VERI_LOW_P6FAILED, showtrace);
            throw -5;
        }
        //Computing P5()
        bddP5 = p5(bddReach, iErr);
        if (iErr != 0)
            throw -1;
        if (bddP5 != bddReach)
        {
            BadStateInfo(bddReach - bddP5, HISC_VERI_LOW_P5FAILED, showtrace);
            throw -6;
        }
        //final synchronous product;
        m_bddSuper = bddReach;
        //save syn-product
        superinfo.statesize = bdd_satcount(m_bddSuper)/pow((double)2, double(m_iNumofBddNormVar));
        superinfo.nodesize =  bdd_nodecount(m_bddSuper);
        if (savetype != HISC_NOTSAVEPRODUCT)
        {
            if (SaveSuper(m_bddSuper, HISC_SAVESUPER_AUTOMATA, savepath) < 0)
                throw -1;
        }
    }
    catch (int iResult)
    {
        if (iResult < -1)
        {
            superinfo.statesize = bdd_satcount(bddReach)/pow((double)2, double(m_iNumofBddNormVar));
            superinfo.nodesize =  bdd_nodecount(bddReach);
            if (savetype != HISC_NOTSAVEPRODUCT)
                SaveSuper(bddReach, HISC_SAVESUPER_AUTOMATA, savepath);
        }
        iRet = -1;
    }
    ClearBddFields();
    CSub::ClearBddFields();
    bdd_done();
    return iRet;
}
/**
 * DESCR:   Synthesize a supervisor for this low-level. The superivsor states is
 *          saved in m_bddSuper.
 * PARA:    computemethod: first compute reachable states or not (See BddHisc.h)
 *                         (input)
 *          superinfo: returned synthesis info (see BddHisc.h)(output)
 *          savetype: how to save synthesized supervisor (See BddHisc.h)(input)
 *          savepath: where to save syn-product(input)
 * RETURN:  0: sucess <0: fail
 * ACCESS:  public
 */
int CLowSub::SynSuper(const HISC_COMPUTEMETHOD computemethod,
                      HISC_SUPERINFO &superinfo,
                      const HISC_SAVESUPERTYPE savetype,
                      const string& savepath)
{
    bdd bddSuper = bddfalse;
    bdd bddK = bddfalse;
    bdd bddReach = bddfalse;
    int iRet = 0;
    int iErr = 0;
    //Initialize the BDD data memebers
    CSub::InitBddFields();
    InitBddFields();
    try
    {
        iRet = 0;
        //Make transition bdds
        if (MakeBdd() < 0)
            throw -1;
        //Check command pair interface
        if (CheckIntf() < 0)
```

```
            throw -1;
        //Initialize bddBad
        bdd bddBad = bddfalse;
        if (GenConBad(bddBad) < 0)
            throw -1;
        if (GenP4Bad(bddBad) < 0)
            throw -1;
        if (computemethod == HISC_ONREACHABLE)
        {
            bddReach = r(bddtrue, iErr);
            if (iErr < 0)
                throw -1;
            bddBad &= bddReach;
            do
            {
                bddK = bddBad;
                //controllable, p4, nonblocking
                if (SynPartSuper(computemethod, bddReach, bddBad) < 0)
                    throw -1;
                bddBad &= bddReach;
                //p6
                bddBad = bdd_not(p6(bddReach - bddBad, iErr));
                if (iErr < 0)
                    throw -1;
                bddBad &= bddReach;
                //p5
                bddBad = bdd_not(p5(bddReach - bddBad, iErr));
                if (iErr < 0)
                    throw -1;
                bddBad &= bddReach;
            } while (bddBad != bddK);
            m_bddSuper = r(bddReach - bddBad, iErr);
            if (iErr < 0)
                throw -1;
        }
        else
        {
            //synthesis
            do
            {
                bddK = bddBad;
                //controllable, p4, nonblocking
                if (SynPartSuper(computemethod, bddBad, bddBad) < 0)
                    throw -1;
                //p6
                bddBad = bdd_not(p6(bdd_not(bddBad), iErr));
                if (iErr < 0)
                    throw -1;
                //p5
                bddBad = bdd_not(p5(bdd_not(bddBad), iErr));
                if (iErr < 0)
                    throw -1;
            } while (bddBad != bddK);
            m_bddSuper = r(bdd_not(bddBad), iErr);
        }
        m_bddMarking &= m_bddSuper;
        m_bddInit &= m_bddSuper;
        //save supervisor
        superinfo.statesize = bdd_satcount(m_bddSuper)/pow((double)2,
                                           double(m_iNumofBddNormVar));
        superinfo.nodesize =  bdd_nodecount(m_bddSuper);
        if (savetype != HISC_SAVESUPER_NONE)
        {
            if (SaveSuper(m_bddSuper, savetype, savepath) < 0)
                throw -1;
        }
    }
    catch( int iErr)
    {
        iRet = -1;
    }
    ClearBddFields();
    CSub::ClearBddFields();
    bdd_done();
    return iRet;
}
/**
 * DESCR:   Does part of the sythesis work, i.e. controllable, p4, nonblocking
 * PARA:    computemethod: first compute reachable states or not (See BddHisc.h)
```

```
 *                          (input)
 *          bddReach: All the current reachable legal states
 *          bddBad: All the current bad states
 * RETURN:  0: sucess <0: fail
 * ACCESS:  private
 */
int CLowSub::SynPartSuper(const HISC_COMPUTEMETHOD computemethod,
                          bdd & bddReach, bdd & bddBad)
{
    bool bFirstLoop = true;
    bdd bddK = bddtrue;
    int iErr = 0;
    try
    {
        if (computemethod == HISC_ONREACHABLE)
        {
            //compute controllable, p4, nonblocking fixpoint
            do
            {
                bddK = bddBad;
                //Computing PLPC(bddBad)
                if (supcp(bddBad) < 0)
                    throw -1;
                bddBad &= bddReach;
                if (bddK == bddBad && bFirstLoop == false)
                    break;
                //Computing CR(G_Lj, not(bddBad))
                bdd bddTemp = bddReach - bddBad;
                bddBad = bdd_not(cr(m_bddMarking, bddTemp, ALL_EVENT, iErr));
                if (iErr != 0)
                    throw -1;
                bddBad &= bddReach;
                bFirstLoop = false;
            } while (bddBad != bddK);
        }
        else
        {
            //compute controllable, p4, nonblocking fixpoint
            do
            {
                bddK = bddBad;
                //Computing PLPC(bddBad)
                if (supcp(bddBad) < 0)
                    throw -1;
                if (bddK == bddBad && bFirstLoop == false)
                    break;
                //Computing CR(not(bddBad))
                bddBad = bdd_not(cr(m_bddMarking, bdd_not(bddBad),
                                    ALL_EVENT, iErr));
                if (iErr != 0)
                    throw -1;
                bFirstLoop = false;
            } while (bddBad != bddK);
        }
    }
    catch (int)
    {
        return -1;
    }
    return 0;
}
/**
 * DESCR:   Compute the initial bad states(Bad_{L_j})(uncontorlalble event part)
 * PARA:    bddConBad: BDD containing all the bad states (output)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
 */
int CLowSub::GenConBad(bdd &bddConBad)
{
    try
    {
        bdd bddPlantTrans = bddfalse;
        bdd bddSpecTrans = bddfalse;
        for (int k = 1; k < 4; k++)
        {
            for (int i = 0; i < m_usiMaxUnCon[k]/ 2; i++)
            {
                //Get spec transition predicate
                bddSpecTrans = bdd_exist(m_pbdd_UnConTrans[k][i],
                                         m_pbdd_UnConPlantVar[k][i]);
```

```
                bddSpecTrans = bdd_exist(bddSpecTrans,
                                m_pbdd_UnConPlantVarPrim[k][i]);
                //Compute illegal state predicate for each uncontrollable event
                bddConBad |= bdd_exist(m_pbdd_UnConPlantTrans[k][i],
                                m_pbdd_UnConPlantVarPrim[k][i]) &
                            bdd_not(bdd_exist(bddSpecTrans,
                                m_pbdd_UnConPlantVarPrim[k][i],
                                m_pbdd_UnConPlantVarPrim[k][i])));
            }
        }
    }
    catch(...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during generating controllable bad states.";
        pPrj->SetErr(sErr, HISC_LOWERR_GENCONBAD);
        return -1;
    }
    return 0;
}
/**
 * DESCR:   Test if there are any bad states in the reachable states
 *          (Uncontorllable event part of Bad_{L_j})
 * PARA:    bddConBad: BDD containing tested bad states(output).
 *                     Initially, bddBad should be bddfalse.
 *          bddReach: BDD containing all reachable states
 *                    in this low-level(input)
 *          vsErr: returned errmsg(output)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
 */
int CLowSub::VeriConBad(bdd &bddConBad, const bdd &bddReach, string & vsErr)
{
    try
    {
        bdd bddSpecTrans = bddfalse;
        for (int k = 1; k < 4; k++)
        {
            for (int i = 0; i < m_usiMaxUnCon[k]/ 2; i++)
            {
                //Get spec transition predicate
                bddSpecTrans = bdd_exist(m_pbdd_UnConTrans[k][i],
                                m_pbdd_UnConPlantVar[k][i]);
                bddSpecTrans = bdd_exist(bddSpecTrans,
                                m_pbdd_UnConPlantVarPrim[k][i]);
                //Compute illegal state predicate for each uncontrollable event
                bddConBad |= bdd_exist(m_pbdd_UnConPlantTrans[k][i],
                                    m_pbdd_UnConPlantVarPrim[k][i]) &
                                bdd_not(bdd_exist(bddSpecTrans,
                                        bdd_exist(m_pbdd_UnConVarPrim[k][i],
                                        m_pbdd_UnConPlantVarPrim[k][i])));
                bddConBad &= bddReach;
                if (bddConBad != bddfalse)
                {
                    vsErr = "Causing uncontrollable event:";
                    vsErr += SearchEventName((EVENTSUB)k, (i + 1) * 2);
                    throw -1;
                }
            }
        }
    }
    catch(int)
    {
    }
    catch(...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during generating controllable bad states.";
        pPrj->SetErr(sErr, HISC_LOWERR_GENCONBAD);
        return -1;
    }
    return 0;
}
/**
 * DESCR:   Compute the initial bad states(Bad_{L_j})(request event part)
 * PARA:    bddConBad: BDD containing all the bad states (output)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
```

```
 */
int CLowSub::GenP4Bad(bdd &bddP4Bad)
{
    try
    {
        bdd bddLowTrans = bddfalse;
        for (int i = 0; i < m_usiMaxUnCon[R_EVENT]/ 2; i++)
        {
            //Get low level sub transition predicate
            bddLowTrans = bdd_exist(m_pbdd_UnConTrans[R_EVENT][i], m_bddIVar);
            bddLowTrans = bdd_exist(bddLowTrans, m_bddIVarPrim);
            //Compute the illegal state predicate for each uncontrollable event
            bddP4Bad |= bdd_exist(m_pbdd_RTrans[0][i], m_bddIVarPrim) &
                        bdd_not(bdd_exist(bddLowTrans,
                            bdd_exist(m_pbdd_UnConVarPrim[R_EVENT][i],
                            m_bddIVarPrim)));
        }
        for (int i = 0; i < ((unsigned short)(m_usiMaxCon[R_EVENT] + 1))/ 2;i++)
        {
            //Get low level sub transition predicate
            bddLowTrans = bdd_exist(m_pbdd_ConTrans[R_EVENT][i], m_bddIVar);
            bddLowTrans = bdd_exist(bddLowTrans, m_bddIVarPrim);
            //Compute the illegal state predicate for each uncontrollable event
            bddP4Bad |= bdd_exist(m_pbdd_RTrans[1][i], m_bddIVarPrim) &
                        bdd_not(bdd_exist(bddLowTrans,
                            bdd_exist(m_pbdd_ConVarPrim[R_EVENT][i],
                            m_bddIVarPrim)));
        }
    }
    catch(...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during generating point 4 bad states.";
        pPrj->SetErr(sErr, HISC_LOWERR_GENP4BAD);
        return -1;
    }
    return 0;
}
/**
 * DESCR:   Test if there are any bad states in the reachable states
 *          (answer event part of Bad_{L_j})
 * PARA:    bddP4Bad: BDD containing tested bad states(output).
 *                    Initially, bddBad should be bddfalse.
 *          bddReach: BDD containing all reachable states
 *                    in this low-level (input)
 *          vsErr: returned errmsg(output)
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
 */
int CLowSub::VeriP4Bad(bdd &bddP4Bad, const bdd &bddReach, string &vsErr)
{
    try
    {
        bdd bddLowTrans = bddfalse;
        for (int i = 0; i < m_usiMaxUnCon[R_EVENT]/ 2; i++)
        {
            //Get low level sub transition predicate
            bddLowTrans = bdd_exist(m_pbdd_UnConTrans[R_EVENT][i], m_bddIVar);
            bddLowTrans = bdd_exist(bddLowTrans, m_bddIVarPrim);
            //Compute the illegal state predicate for each uncontrollable event
            bddP4Bad |= bdd_exist(m_pbdd_RTrans[0][i], m_bddIVarPrim) &
                        bdd_not(bdd_exist(bddLowTrans,
                            bdd_exist(m_pbdd_UnConVarPrim[R_EVENT][i],
                            m_bddIVarPrim)));
            bddP4Bad &= bddReach;
            if (bddP4Bad != bddfalse)
            {
                vsErr = "Causing request event:";
                vsErr += SearchEventName(R_EVENT, (i + 1) * 2);
                return 0;
            }
        }
        for (int i = 0; i < ((unsigned short)(m_usiMaxCon[R_EVENT] + 1))/ 2;i++)
        {
            //Get low level sub transition predicate
            bddLowTrans = bdd_exist(m_pbdd_ConTrans[R_EVENT][i], m_bddIVar);
            bddLowTrans = bdd_exist(bddLowTrans, m_bddIVarPrim);
            //Compute the illegal state predicate for each uncontrollable event
```

```
            bddP4Bad |= bdd_exist(m_pbdd_RTrans[1][i], m_bddIVarPrim) &
                        bdd_not(bdd_exist(bddLowTrans,
                            bdd_exist(m_pbdd_ConVarPrim[R_EVENT][i],
                            m_bddIVarPrim)));
            bddP4Bad &= bddReach;
            if (bddP4Bad != bddfalse)
            {
                vsErr = "Causing request event:";
                vsErr += SearchEventName(R_EVENT, i * 2 + 1);
                return 0;
            }
        }
    }
    catch(...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during generating point 4 bad states.";
        pPrj->SetErr(sErr, HISC_LOWERR_GENP4BAD);
        return -1;
    }
    return 0;
}
/**
 * DESCR:   compute PLPC(P)
 * PARA:    bddP : BDD for predicate P. (input and output(=PHIC(P)))
 * RETURN:  0: sucess -1: fail
 * ACCESS:  private
 */
int CLowSub::supcp(bdd & bddP)
{
    bdd bddK1 = bddfalse;
    bdd bddK2 = bddfalse;
    int iEvent = 0;
    int iIndex = 0;
    EVENTSUB EventSub;
    try
    {
        while (bddP != bddK1)
        {
            bddK1 = bddP;
            for (int i = 0; i < this->GetNumofDES(); i++)
            {
                bddK2 = bddfalse;
                while (bddP != bddK2)
                {
                    bddK2 = bddP;
                    for (int j = 0; j < m_pDESArr[i]->GetNumofEvents(); j++)
                    {
                        iEvent = (m_pDESArr[i]->GetEventsArr())[j];
                        EventSub = (EVENTSUB)(iEvent >> 28);
                        iIndex = iEvent & 0x0000FFFF;
                        if ( iEvent % 2 == 0)
                        {
                            iIndex = (iIndex - 2) / 2;
                            bddP |=
                                bdd_appex(m_pbdd_UnConTrans[EventSub][iIndex],
                                          bdd_replace(bddK2,
                                                m_pPair_UnCon[EventSub][iIndex]),
                                          bddop_and,
                                          m_pbdd_UnConVarPrim[EventSub][iIndex]);
                        }
                        else if ( EventSub == R_EVENT )
                        {
                            iIndex = (iIndex - 1) / 2;
                            bddP |=
                                bdd_appex(m_pbdd_ConTrans[EventSub][iIndex],
                                          bdd_replace(bddK2,
                                                m_pPair_Con[EventSub][iIndex]),
                                          bddop_and,
                                          m_pbdd_ConVarPrim[EventSub][iIndex]);
                        }
                    }
                }
            }
        }
    }
    catch (...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during computing PLPC(P).";
```

```
        pPrj->SetErr(sErr, HISC_LOWERR_SUPCP);
        return -1;
    }
    return 0;
}
/**
 * DESCR:   compute CR(G_{L_j}, P', \Sigma', P)
 * PARA:    bddPStart: P' (input)
 *          bddP: P (input)
 *          viEventSub: \Sigma' (input) (0,1,2,3) <-> (H,R,A,L) ALL_EVENT<->All
 *          iErr: returned Errcode (0: success <0: fail)(output)
 * RETURN:  BDD for CR(G_{L_j}, P', \Sigma', P)
 * ACCESS:  private
 */
bdd CLowSub::cr(const bdd & bddPStart, const bdd & bddP,
                const int viEventSub, int & iErr)
{
    try
    {
        bdd bddK = bddP & bddPStart;
        bdd bddK1 = bddfalse;
        bdd bddK2 = bddfalse;
        bdd bddKNew = bddfalse;
        int iEvent = 0;
        int iIndex = 0;
        EVENTSUB EventSub;
        while (bddK != bddK1)
        {
            bddK1 = bddK;
            for (int i = 0; i < this->GetNumofDES(); i++)
            {
                bddK2 = bddfalse;
                while (bddK != bddK2)
                {
                    bddKNew = bddK - bddK2;
                    bddK2 = bddK;
                    for (int j = 0; j < m_pDESArr[i]->GetNumofEvents(); j++)
                    {
                        iEvent = (m_pDESArr[i]->GetEventsArr())[j];
                        EventSub = (EVENTSUB)(iEvent >> 28);
                        if (viEventSub == ALL_EVENT ||
                            viEventSub == (int)EventSub)
                        {
                            iIndex = iEvent & 0x0000FFFF;
                            if (iEvent % 2 == 0)
                            {
                                iIndex = (iIndex - 2) / 2;
                                bddK |=
                                    bdd_appex(
                                        m_pbdd_UnConTrans[EventSub][iIndex],
                                        bdd_replace(bddKNew,
                                               m_pPair_UnCon[EventSub][iIndex]),
                                        bddop_and,
                                        m_pbdd_UnConVarPrim[EventSub][iIndex]) &
                                    bddP;
                            }
                            else
                            {
                                iIndex = (iIndex - 1) / 2;
                                bddK |=
                                    bdd_appex(
                                        m_pbdd_ConTrans[EventSub][iIndex],
                                        bdd_replace(bddKNew,
                                                m_pPair_Con[EventSub][iIndex]),
                                        bddop_and,
                                        m_pbdd_ConVarPrim[EventSub][iIndex]) &
                                    bddP;
                            }
                        }
                    }
                }
            }
        }
        return bddK;
    }
    catch (...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during computing coreachable predicate.";
        pPrj->SetErr(sErr, HISC_LOWERR_COREACH);
```

```
        iErr = -1;
        return bddfalse;
    }
}
/**
 * DESCR:   compute R(G_{L_j}, P)
 * PARA:    bddP: P (input)
 *          iErr: returned Errcode (0: success <0: fail)(output)
 * RETURN:  BDD for R(G_{L_j}, P)
 * ACCESS:  private
 */
bdd CLowSub::r(const bdd &bddP, int &iErr)
{
    try
    {
        bdd bddK = bddP & m_bddInit;
        bdd bddK1 = bddfalse;
        bdd bddK2 = bddfalse;
        bdd bddKNew = bddfalse;
        int iEvent = 0;
        int iIndex = 0;
        EVENTSUB EventSub;
        while (bddK != bddK1)
        {
            bddK1 = bddK;
            for (int i = 0; i < this->GetNumofDES(); i++)
            {
                bddK2 = bddfalse;
                while (bddK != bddK2)
                {
                    bddKNew = bddK-bddK2;
                    bddK2 = bddK;
                    for (int j = 0; j < m_pDESArr[i]->GetNumofEvents(); j++)
                    {
                        iEvent = (m_pDESArr[i]->GetEventsArr())[j];
                        EventSub = (EVENTSUB)(iEvent >> 28);
                        iIndex = iEvent & 0x0000FFFF;
                        if (iEvent % 2 == 0)
                        {
                            iIndex = (iIndex - 2) / 2;
                            bddK |= bdd_replace(
                                bdd_appex(m_pbdd_UnConTrans[EventSub][iIndex],
                                    bddKNew, bddop_and,
                                    m_pbdd_UnConVar[EventSub][iIndex]),
                                m_pPair_UnConPrim[EventSub][iIndex]) & bddP;
                        }
                        else
                        {
                            iIndex = (iIndex - 1) / 2;
                            bddK |= bdd_replace(
                                bdd_appex(m_pbdd_ConTrans[EventSub][iIndex],
                                    bddKNew, bddop_and,
                                    m_pbdd_ConVar[EventSub][iIndex]),
                                m_pPair_ConPrim[EventSub][iIndex]) & bddP;
                        }
                    }
                }
            }
        }
        return bddK;
    }
    catch (...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during computing coreachable.";
        pPrj->SetErr(sErr, HISC_LOWERR_REACH);
        iErr = -1;
        return bddfalse;
    }
}
/**
 * DESCR:   Compute interface consistent condition 5 operator \Gamma_{p5_j}(P).
 * PARA:    bddGood: P (input)
 *          iErr: returned Errcode (0: success <0: fail)(output)
 * RETURN:  BDD for predicate \Gamma_{p5_j}(P)
 * ACCESS:  private
 */
bdd CLowSub::p5(const bdd& bddGood, int &iErr)
{
    int iLocalErr = 0;
```

```
try
{
    bdd bddPBad5 = bddfalse;
    bdd bddPalpha = bddfalse;
    bdd bddPCRalpha = bddfalse;
    bdd bddPrho = bddfalse;
    bddPair *pPair = bdd_newpair();
    SetBddPairs(pPair, m_bddIVar, m_bddIVarPrim);
    //Uncontrollable answer events
    for (int i = 0; i < m_usiMaxUnCon[A_EVENT] / 2; i++)
    {
        bddPalpha =  bdd_relprod(m_pbdd_UnConTrans[A_EVENT][i],
                        bdd_replace(bddGood, m_pPair_UnCon[A_EVENT][i]),
                        m_pbdd_UnConVarPrim[A_EVENT][i]);
        bddPCRalpha = cr(bddPalpha, bddGood, (int)L_EVENT, iLocalErr);
        if (iLocalErr < 0)
            throw -1;
        for (int j = 0;
             j < ((unsigned short)(m_usiMaxCon[R_EVENT] + 1)) / 2; j++)
        {
            bddPrho = bdd_relprod(m_pbdd_ConTrans[R_EVENT][j],
                                  bdd_replace(bdd_not(bddPCRalpha),
                                  m_pPair_Con[R_EVENT][j]),
                                  m_pbdd_ConVarPrim[R_EVENT][j]);
            bddPrho &= bdd_relprod(m_pbdd_ConTrans[R_EVENT][j],
                                bdd_replace(bdd_exist(m_pbdd_ATrans[0][i],
                                m_bddIVarPrim), pPair),
                                m_pbdd_ConVarPrim[R_EVENT][j]);
            bddPBad5 |= bddPrho;
        }
        for (int j = 0; j < m_usiMaxUnCon[R_EVENT] / 2; j++)
        {
            bddPrho = bdd_relprod(m_pbdd_UnConTrans[R_EVENT][j],
                                  bdd_replace(bdd_not(bddPCRalpha),
                                  m_pPair_UnCon[R_EVENT][j]),
                                  m_pbdd_UnConVarPrim[R_EVENT][j]);
            bddPrho &= bdd_relprod(m_pbdd_UnConTrans[R_EVENT][j],
                                bdd_replace(bdd_exist(m_pbdd_ATrans[0][i],
                                m_bddIVarPrim), pPair),
                                m_pbdd_UnConVarPrim[R_EVENT][j]);
            bddPBad5 |= bddPrho;
        }
    }
    //Controllable answer events
    for (int i = 0;
         i < ((unsigned short)(m_usiMaxCon[A_EVENT] + 1)) / 2; i++)
    {
        bddPalpha =  bdd_relprod(m_pbdd_ConTrans[A_EVENT][i],
                        bdd_replace(bddGood, m_pPair_Con[A_EVENT][i]),
                        m_pbdd_ConVarPrim[A_EVENT][i]);
        bddPCRalpha = cr(bddPalpha, bddGood, (int)L_EVENT, iLocalErr);
        if (iLocalErr < 0)
            throw -1;
        for (int j = 0;
             j < ((unsigned short)(m_usiMaxCon[R_EVENT] + 1)) / 2; j++)
        {
            bddPrho = bdd_relprod(m_pbdd_ConTrans[R_EVENT][j],
                                  bdd_replace(bdd_not(bddPCRalpha),
                                  m_pPair_Con[R_EVENT][j]),
                                  m_pbdd_ConVarPrim[R_EVENT][j]);
            bddPrho &= bdd_relprod(m_pbdd_ConTrans[R_EVENT][j],
                                bdd_replace(bdd_exist(m_pbdd_ATrans[1][i],
                                m_bddIVarPrim), pPair),
                                m_pbdd_ConVarPrim[R_EVENT][j]);
            bddPBad5 |= bddPrho;
        }
        for (int j = 0; j < m_usiMaxUnCon[R_EVENT] / 2; j++)
        {
            bddPrho = bdd_relprod(m_pbdd_UnConTrans[R_EVENT][j],
                                  bdd_replace(bdd_not(bddPCRalpha),
                                  m_pPair_UnCon[R_EVENT][j]),
                                  m_pbdd_UnConVarPrim[R_EVENT][j]);
            bddPrho &= bdd_relprod(m_pbdd_UnConTrans[R_EVENT][j],
                                bdd_replace(bdd_exist(m_pbdd_ATrans[1][i],
                                m_bddIVarPrim), pPair),
```

```
                                    m_pbdd_UnConVarPrim[R_EVENT][j]);
                bddPBad5 |= bddPrho;
            }
        }
        bdd_freepair(pPair);
        pPair = NULL;
        return bddGood - bddPBad5;
    }
    catch (...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during computing Gamma_{p5_j}(P).";
        pPrj->SetErr(sErr, HISC_LOWERR_P5);
        iErr = -1;
        return bddfalse;
    }
}
/**
 * DESCR:   Compute interface consistent condition P6 operator \Gamma_{p6_j}(P).
 * PARA:    bddGood: P (input)
 *          iErr: returned Errcode (0: success <0: fail)(output)
 * RETURN:  BDD for predicate \Gamma_{p6_j}(P)
 * ACCESS:  private
 */
bdd CLowSub::p6(const bdd& bddP, int &iErr)
{
    try
    {
        int iErr = 0;
        bdd bddPBad6 = bddfalse;
        bddPBad6 = (m_bddIntfMarking & bddP) - cr(m_bddMarking, bddP,
                    (int)L_EVENT, iErr);
        if (iErr < 0)
            throw -1;
        return bddP - bddPBad6;
    }
    catch (...)
    {
        string sErr = this->GetSubName();
        sErr += ": Error during computing Gamma_{p6_j}(P).";
        pPrj->SetErr(sErr, HISC_LOWERR_P6);
        iErr = -1;
        return bddfalse;
    }
}
```

```
/**************************************************************************
  FILE:  DES.h
  DESCR: Header file of DES.cpp (Process DES file)
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
**************************************************************************/
#ifndef _DES_H_
#define _DES_H_
#include <string>
#include <map>
#include <list>
#include "type.h"
#include <fstream>
using namespace std;
class CSub;
class CDES
{
public: //Constructor and Destructor
    CDES(CSub *vpSub, const string &vsDESFile, const DESTYPE vDESType);
    virtual ~CDES();
public:
    int LoadDES();
    int PrintDES(ofstream & fout);
public:
    string GetDESName() const {return m_sDESName;};
    int * GetEventsArr() {return m_piEventsArr;};
    int GetNumofEvents() const {return m_DESEventsMap.size();};
    int GetNumofMarkingStates() const {return m_MarkingList.size();};
    MARKINGLIST & GetMarkingList() {return m_MarkingList;};
    int GetNumofStates() const { return m_iNumofStates;};
    int GetInitState() const {return m_iInitState;};
    map<int, int> *GetTrans() const {return m_pTransArr;};
    DESTYPE GetDESType() const {return m_DESType;};
    CSub*  GetSub() {return m_pSub;};
    string GetStateName(int iState) {return m_InvStatesMap[iState];};
private: //data memeber
    string m_sDESFile;   //DES file name with path
    string m_sDESName;   //DES name without path and file extension
    DESTYPE m_DESType;   //DES type
    int m_iNumofStates; //Number of States
    int m_iInitState;  //Initial state
    MARKINGLIST m_MarkingList; //Link list containing all marking states
    STATES m_StatesMap; //A STL Map for states (state name (key), state index)
    INVSTATES m_InvStatesMap; //A STL Map for states (state index (key),
                              //state name)(for printing)
    EVENTS m_DESEventsMap;  //A STL Map for events (event name (key),
                            //local event index). Used only for current DES
                            //(speed reason)
    INVEVENTS m_InvDESEventsMap; //A STL Map for events (localindex (key),
                                 //event name). Used only for current DES
                                 //(for printing)
    EVENTS m_UnusedEvents; //A STL Map for blocked events(name: key, index)
    int *m_piEventsArr;  //Save all the event indices ascendingly.
                         //used for find shared events between two DESes.
    TRANS *m_pTransArr; //Transiton Map array, indexed by event indices.
                        //TRANSMAP: first int: source state index
                        //          second int: target state index
    CSub *m_pSub; //which subsystem this DES belongs to
private: //internal function members
    int AddEvent(const string & vsEventName,
                 const char cEventSub,
                 const char cControllable);
    int AddTrans(const string & vsLine,
                 const string & vsExitState,
                 const int viExitState);
};
#endif //_DES_H_
```

```
/*************************************************************************
  FILE:  DES.cpp
  DESCR: Processing DES file
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*************************************************************************/
#include "DES.h"
#include "pubfunc.h"
#include <fstream>
#include <cstdlib>
#include "type.h"
#include "Project.h"
#include "errmsg.h"
#include <cassert>
#include "Sub.h"
#include "LowSub.h"
#include "HighSub.h"
using namespace std;
extern CProject *pPrj;
/**
 * DESCR:   Constructor
 * PARA:    vpSub:  which subsystem this DES belongs to (input)
 *          vsDESFile:  DES file name with path (input)
 *          vDESType:   DES Type (inpute)
 * RETURN: none
 * ACCESS: public
 */
CDES::CDES(CSub *vpSub, const string &vsDESFile, const DESTYPE vDESType)
{
    m_pSub = vpSub;
    m_sDESFile = vsDESFile;
    m_sDESName.clear();
    m_DESType = vDESType;
    m_iNumofStates = 0;
    m_iInitState = -1;
    m_MarkingList.clear();
    m_StatesMap.clear();
    m_InvStatesMap.clear();
    m_DESEventsMap.clear();
    m_UnusedEvents.clear();
    m_InvDESEventsMap.clear();
    m_piEventsArr = NULL;
    m_pTransArr = NULL;
}
/**
 * DESCR:   Destructor
 * PARA:    None
 * RETURN:  None
 * ACCESS:  public
 */
CDES::~CDES()
{
    delete[] m_pTransArr;
    m_pTransArr = NULL;
    delete[] m_piEventsArr;
    m_piEventsArr = NULL;
}
/**
 * DESCR:   Loading DES file
 * PARA:    None
 * RETURN:  0: sucess  -1: fail
 * ACCESS:  public
 */
int CDES::LoadDES()
{
    ifstream fin;
    int iRet = 0;
    string sErr;
    int i = 0;
    string sSubName = m_pSub->GetSubName();
    try
    {
        m_sDESFile = str_trim(m_sDESFile);
        if (m_sDESFile.length() <= 4)
        {
            pPrj->SetErr(sSubName + ": Invalid DES file name " + m_sDESFile,
                         HISC_BAD_DES_FILE);
            throw -1;
        }
        if (m_sDESFile.substr(m_sDESFile.length() - 4) != ".hsc")
        {
```

```
    pPrj->SetErr(sSubName + ": Invalid DES file name " + m_sDESFile,
                 HISC_BAD_DES_FILE);
    throw -1;
}
fin.open(m_sDESFile.data(), ios::in);
//unable to find DES file
if (!fin)
{
    pPrj->SetErr(sSubName + ": Unable to open the DES file " +
              m_sDESFile, HISC_BAD_DES_FILE);
    throw -1;
}
m_sDESName = GetNameFromFile(m_sDESFile);
string sDESLoc = sSubName + ":" + m_sDESName + ": ";
char scBuf[MAX_LINE_LENGTH];
string sLine;
int iField = -1; //0: States 1: InitState 2: MarkingStates
              //3: Events 4: Transitions
char *scFieldArr[] = {"STATES", "INITSTATE", "MARKINGSTATES",
                      "EVENTS", "TRANSITIONS"};
int iStatesFieldFlag = 0;  //1: just finised reading the [States] line,
                           //   so next line should be the num of states
                           //0: otherwise
int iTmpStateIndex = 0;
int iTmpEventIndex = 0;
char cEventSub = '\0';
char cControllable = '\0';
string sExitState;
int iExitState = -1;
while (fin.getline(scBuf,  MAX_LINE_LENGTH))
{
    sLine = str_nocomment(scBuf);
    sLine = str_trim(sLine);
    if (sLine.empty())
        continue;
    //Field line
    if (sLine[0] == '[' && sLine[sLine.length() - 1] == ']')
    {
        sLine = sLine.substr(1, sLine.length() - 1);
        sLine = sLine.substr(0, sLine.length() - 1);
        sLine = str_upper(str_trim(sLine));
        iField++;
        if (iField <= 4)
        {
            if (sLine != scFieldArr[iField])
            {
                pPrj->SetErr(sDESLoc +
                             "Field name or order is incorrect!",
                             HISC_BAD_DES_FORMAT);
                throw -1;
            }
            if (iField == 0)
            {
                iStatesFieldFlag = 1;
            }
        }
        else
        {
            pPrj->SetErr(sDESLoc + "Two many fields.",
                         HISC_BAD_DES_FORMAT);
            throw -1;
        }
    }
    else  //Data line
    {
        switch (iField)
        {
        case 0:  //States
            if (iStatesFieldFlag == 1) //num of states
            {
                if (atoi(sLine.data()) <= 0 ||
                   atoi(sLine.data()) > MAX_STATES_IN_ONE_COMPONENT_DES)
                {
                    pPrj->SetErr(sDESLoc + "Too few or too many states",
                                 HISC_BAD_DES_FORMAT);
                    throw -1;
                }
                //initialize the number of states
                m_iNumofStates = atoi(sLine.data());
```

```
            //initialize the transition arrary
            m_pTransArr = new TRANS[m_iNumofStates];
            iStatesFieldFlag = 0;
        }
        else
        {
            if (m_StatesMap.find(sLine) != m_StatesMap.end())
            {
                pPrj->SetErr(sDESLoc + "Duplicate state names--" +
                             sLine, HISC_BAD_DES_FORMAT);
                throw -1;
            }
            else if (sLine[0] == '(')
            {
                pPrj->SetErr(sDESLoc +
                    "The first letter of state names can not be (",
                    HISC_BAD_DES_FORMAT);
                throw -1;
            }
            else
            {
                m_StatesMap[sLine] = m_StatesMap.size() - 1;
                m_InvStatesMap[m_StatesMap.size() - 1] = sLine;
            }
        }
        break;
    case 1:  //InitState
        //-------------------------------------------------
        //Must specify the number of states
        if (m_iNumofStates == 0)
        {
            pPrj->SetErr(sDESLoc + "Number of states is absent.",
                         HISC_BAD_DES_FORMAT);
            throw -1;
        }
        //If there is no state names specified, generate state
        //names automatically.
        if (m_StatesMap.size() == 0)
        {
            for (i = 0; i < m_iNumofStates; i++)
            {
                m_StatesMap[str_itos(i)] = i;
                m_InvStatesMap[i] = str_itos(i);
            }
        }
        //if specify state names, the number of state names must be
        //equal to m_iNumofStates.
        if (((unsigned int)m_iNumofStates) != m_StatesMap.size())
        {
            pPrj->SetErr(sDESLoc + "States are incomplete.",
                    HISC_BAD_DES_FORMAT);
            throw -1;
        }
        //-------------------------------------------------------
        //Initial state name must be valid
        if (m_StatesMap.find(sLine) == m_StatesMap.end())
        {
            pPrj->SetErr(sDESLoc + "Initial state is not defined.",
                         HISC_BAD_DES_FORMAT);
            throw -1;
        }
        //only one initial state allowed
        if (m_iInitState != -1)
        {
            pPrj->SetErr(sDESLoc + "More than one initial states.",
                         HISC_BAD_DES_FORMAT);
            throw -1;
        }
        m_iInitState = m_StatesMap[sLine];
        break;
    case 2:  //MarkingStates
        if (m_StatesMap.find(sLine) == m_StatesMap.end())
        {
            pPrj->SetErr(sDESLoc + "Marking states do not exist.",
                         HISC_BAD_DES_FORMAT);
            throw -1;
        }
        iTmpStateIndex = m_StatesMap[sLine];
```

```
            for (MARKINGLIST::const_iterator ci = m_MarkingList.begin();
                ci != m_MarkingList.end(); ci++)
            {
                if (*ci == iTmpStateIndex)
                {
                    pPrj->SetErr(sDESLoc + "Duplicate marking states.",
                                 HISC_BAD_DES_FORMAT);
                    throw -1;
                }
            }
            m_MarkingList.push_back(iTmpStateIndex);
            break;
        case 3:  //Events
            //Get event type H/R/A/L
            if (sLine.length() < 5)
            {
                pPrj->SetErr(sDESLoc + "Incorrect event definition.",
                             HISC_BAD_DES_FORMAT);
                throw -1;
            }
            cEventSub = sLine[sLine.length() - 1];
            sLine = str_trim(sLine.substr(0, sLine.length() - 1));
            //Get controllable or not
            if (sLine.length() < 3)
            {
                pPrj->SetErr(sDESLoc + "Incorrect event definition.",
                             HISC_BAD_DES_FORMAT);
                throw -1;
            }
            cControllable = sLine[sLine.length() - 1];
            sLine = str_trim(sLine.substr(0, sLine.length() - 1));
            //Get event name
            if (sLine.empty())
            {
                pPrj->SetErr(sDESLoc + "Incorrect event definition.",
                             HISC_BAD_DES_FORMAT);
                throw -1;
            }
            if (cEventSub >= 'a')
                cEventSub -= 32;
            if (cControllable >= 'a')
                cControllable -= 32;
            iTmpEventIndex = AddEvent(sLine, cEventSub, cControllable);
            if (iTmpEventIndex < 0)
                throw -1;   //Errmsg generated by AddEvent
            m_DESEventsMap[sLine] = iTmpEventIndex;
            m_UnusedEvents[sLine] = iTmpEventIndex;
            m_InvDESEventsMap[iTmpEventIndex] = sLine;
            break;
        case 4:  //Transitions
            //check exiting state
            if (sLine[0] != '('
            {
                if (m_StatesMap.find(sLine) == m_StatesMap.end())
                {
                    pPrj->SetErr(sDESLoc + "Exiting state:" + sLine +
                        " in transitions does not exist",
                        HISC_BAD_DES_FORMAT);
                    throw -1;
                }
                iExitState = m_StatesMap[sLine];
                sExitState = sLine;
            }
            else  //Transitions
            {
                if (AddTrans(sLine, sExitState, iExitState) < 0)
                    throw -1;
            }
            break;
        default:
            pPrj->SetErr(sDESLoc + "Bad DES file format!",
                         HISC_BAD_DES_FORMAT);
            throw -1;
        }
    }
}
//No initial state defined
if (m_iInitState == -1)
{
```

```
            pPrj->SetErr(sDESLoc + "No initial state.", HISC_BAD_DES_FORMAT);
            throw -1;
        }
        //No marking states defined
        if (m_MarkingList.size() == 0)
        {
            pPrj->SetErr(sDESLoc + "No marking states", HISC_BAD_DES_FORMAT);
            throw -1;
        }
        //must have all the fields
        if (iField != 4)
        {
            pPrj->SetErr(sDESLoc + "Incomplete DES file.", HISC_BAD_DES_FORMAT);
            throw -1;
        }
        //Add event indices into m_piEventsArr;
        m_piEventsArr = new int[m_DESEventsMap.size()];
        i = 0;
        for (EVENTS::const_iterator ci = m_DESEventsMap.begin();
             ci != m_DESEventsMap.end(); ++ci)
        {
            m_piEventsArr[i] = ci->second;
            ++i;
        }
        qsort(m_piEventsArr, m_DESEventsMap.size(), sizeof(int), CompareInt);
        //unused events(blocked events)
        if (m_UnusedEvents.size() > 0)
        {
            string sWarn;
            sWarn = "\nWarning: ";
            sWarn += sDESLoc + "blocks events:\n";
            for (EVENTS::const_iterator ci = m_UnusedEvents.begin();
                    ci != m_UnusedEvents.end(); ++ci)
            {
                sWarn += ci->first;
                sWarn += "\n";
            }
            pPrj->SetErr(sWarn, HISC_WARN_BLOCKEVENTS);
        }
        fin.close();
    }
    catch (int iError)
    {
        if (fin.is_open())
            fin.close();
        iRet = iError;
    }
    return iRet;
}
/*
 * DESCR:   Add an event to CSub event map and CProject event map
 *          For CSub event map: If exists, return local index;
 *                              Otherwise create a new one.
 *          For CProject event map: If exists, must have same global index;
 *                                  Otherwise the event sets are not disjoint
 * PARA:    vsEventName: Event name(input)
 *          cEventSub: Event type ('H", 'L', 'R', 'A')(input)
 *          cControllable: Controllable? ('Y', 'N')(input)
 * RETURN:  >0 global event index
 *          <0 the event sets are not disjoint.
 * ACCESS:  Private
 */
int CDES::AddEvent(const string & vsEventName,
                   const char cEventSub,
                   const char cControllable)
{
    string sErr;
    int iTmpEventIndex = 0;
    int iTmpLocalEventIndex = 0;
    EVENTSUB vConflictEventSub = H_EVENT;
    int viConflictSubIndex = -1;
    string sDESLoc = m_pSub->GetSubName() + ": " + m_sDESName + ": ";
    //Interface DES
    if (m_DESType == INTERFACE_DES)
    {
        if (cEventSub != 'R' && cEventSub != 'A')
        {
            pPrj->SetErr(sDESLoc + "Interface DES can only inlcudes request" +
                         " or answer events.", HISC_BAD_DES_FORMAT);
```

```
        return -1;
    }
}
//Controllable or uncontrollable
if (cControllable != 'Y' && cControllable != 'N')
{
    pPrj->SetErr(sDESLoc + "Unknown event controllable type--" +vsEventName,
                 HISC_BAD_DES_FORMAT);
    return -1;
}
//already defined in current DES
if (m_DESEventsMap.find(vsEventName) != m_DESEventsMap.end())
{
    pPrj->SetErr(sDESLoc + "Duplicate events definition--" + vsEventName,
        HISC_BAD_DES_FORMAT);
    return -1;
}
//High sub DES
if (m_pSub->GetSubIndex() == 0)
{
    if (cEventSub == 'L')
    {
        pPrj->SetErr(sDESLoc + "High sub DES inlcudes a low sub event--" +
                     vsEventName, HISC_BAD_DES_FORMAT);
        return -1;
    }
    if (cEventSub == 'H')  //high sub event
    {
        //Compute local event index
        iTmpLocalEventIndex = m_pSub->AddSubEvent(vsEventName,
                                  SubLetterToValue(cEventSub),
                                  (cControllable == 'Y')?
                                  CON_EVENT:UNCON_EVENT);
        if ((cControllable == 'Y' && iTmpLocalEventIndex % 2 == 0) ||
           (cControllable == 'N' && iTmpLocalEventIndex % 2 == 1))
        {
            pPrj->SetErr(sDESLoc + "Event " + vsEventName +
                         " has inconsistent controllability definitions.",
                HISC_BAD_DES_FORMAT);
            return -1;
        }
        //Compute global event index
        iTmpEventIndex = pPrj->GenEventIndex(SubLetterToValue(cEventSub),
                                             m_pSub->GetSubIndex(),
                                             iTmpLocalEventIndex);
    }
    else  //interface events
    {
        iTmpEventIndex = pPrj->SearchPrjEvent(vsEventName);
        if (iTmpEventIndex < 0)
        {
            pPrj->SetErr(sDESLoc + "Event " + vsEventName +
                         " is not a valid Request/Answer Event.",
                         HISC_BAD_DES_FORMAT);
            return -1;
        }
        else
        {
            int iEventSub = iTmpEventIndex >> 28;
            int iSubIndex = (iTmpEventIndex & 0x0FFF0000) >> 16;
            if ((EVENTSUB)iEventSub != SubLetterToValue(cEventSub))
            {
                pPrj->SetErr(sDESLoc + "Event " + vsEventName +
                    " is not a " + SubValueToLetter((EVENTSUB)iEventSub) +
                    " Event in low level " +
                    pPrj->GetSub(iSubIndex)->GetSubName(),
                    HISC_BAD_DES_FORMAT);
                return -1;
            }
            else if(!(iTmpEventIndex % 2 == 0 && cControllable == 'N' ||
                    iTmpEventIndex % 2 == 1 && cControllable == 'Y'))
            {
                pPrj->SetErr(sDESLoc + "Event " + vsEventName +
                    " doesn't have same controllable property in low level "
                    + pPrj->GetSub(iSubIndex)->GetSubName(),
                    HISC_BAD_DES_FORMAT);
                return -1;
            }
            else
```

```
                return iTmpEventIndex;
            }
        }
    }
    else //Low sub DES
    {
        if (cEventSub == 'H')
        {
            pPrj->SetErr(sDESLoc + "Low sub DES inlcudes a high sub event--" +
                vsEventName, HISC_BAD_DES_FORMAT);
            return -1;
        }
        if (m_DESType == INTERFACE_DES || cEventSub == 'L')
        {
            //Compute local event index
            iTmpLocalEventIndex = m_pSub->AddSubEvent(vsEventName,
                     SubLetterToValue(cEventSub),
                     (cControllable == 'Y')? CON_EVENT:UNCON_EVENT);
            if ((cControllable == 'Y' && iTmpLocalEventIndex % 2 == 0) ||
               (cControllable == 'N' && iTmpLocalEventIndex % 2 == 1))
            {
                pPrj->SetErr(sDESLoc + "Event " + vsEventName +
                       " has inconsistent controllability definitions.",
                       HISC_BAD_DES_FORMAT);
                return -1;
            }
            //Compute global event index
            iTmpEventIndex = pPrj->GenEventIndex(SubLetterToValue(cEventSub),
                                                 m_pSub->GetSubIndex(),
                                                 iTmpLocalEventIndex);
        }
        else  //'R' and 'A' appeared in low level des, must be in interface des
        {
            iTmpEventIndex = pPrj->SearchPrjEvent(vsEventName);
            if (iTmpEventIndex < 0)
            {
                pPrj->SetErr(sDESLoc + "Event " + vsEventName +
                    " is not a valid Request/Answer Event.",
                    HISC_BAD_DES_FORMAT);
                return -1;
            }
            else
            {
                int iEventSub = iTmpEventIndex >> 28;
                int iSubIndex = (iTmpEventIndex & 0x0fff0000) >> 16;
                if (iSubIndex != m_pSub->GetSubIndex())
                {
                    pPrj->SetErr(sDESLoc + "Event " + vsEventName +
                        " is an invalid Request/Answer Event in interface DES.",
                        HISC_BAD_DES_FORMAT);
                    return -1;
                }
                else if ((EVENTSUB)iEventSub != SubLetterToValue(cEventSub))
                {
                    pPrj->SetErr(sDESLoc + "Event " + vsEventName +
                        " has a different event level in the interface DES.",
                        HISC_BAD_DES_FORMAT);
                    return -1;
                }
                else if(!(iTmpEventIndex % 2 == 0 && cControllable == 'N' ||
                        iTmpEventIndex % 2 == 1 && cControllable == 'Y'))
                {
                    pPrj->SetErr(sDESLoc + "Event " + vsEventName +
                     " has a different controllable property in interface DES.",
                     HISC_BAD_DES_FORMAT);
                    return -1;
                }
                else
                    return iTmpEventIndex;
            }
        }
    }
    //Add Event to pPrj->m_AllEventsMap
    if (pPrj->AddPrjEvent(vsEventName, iTmpEventIndex,
                vConflictEventSub, viConflictSubIndex) < 0)
    {
        sErr = "Event conflict--" +  m_pSub->GetSubName() + ":" +
                this->GetDESName() + ":" +
                vsEventName + " is also defined in sub " +
```

```
                pPrj->GetSub(viConflictSubIndex)->GetSubName() + " as a " +
                SubValueToLetter(vConflictEventSub) + " event";
        pPrj->SetErr(sErr, HISC_BAD_DES_FORMAT);
        iTmpEventIndex = -1;
    }
    return iTmpEventIndex;
}
/*
 * DESCR:   Add a transition to the m_pTransArr of the current DES.
 * PARA:    vsLine: a text line in [Transition] field(input)
 *          vsExitState: source state name of the transition(input)
 *          viExitState: source state index of the transition(input)
 * RETURN:  0: success -1: fail
 * ACCESS:  private
 */
int CDES::AddTrans(const string & vsLine,
                   const string & vsExitState,
                   const int viExitState)
{
    string sTrans = vsLine;
    string sEnterState;
    int iEnterStateIndex;
    string sTransEvent;
    int iTransEventIndex;
    unsigned int iSepLoc = string::npos;
    string sErrMsg;
    string sDESLoc = m_pSub->GetSubName() + ": " + m_sDESName + ": ";
    try
    {
        if (viExitState == -1)
        {
            pPrj->SetErr(sDESLoc + "No existing state for transitions",
                                        HISC_BAD_DES_FORMAT);
            throw -1;
        }
        //remove '(' and ')'
        sTrans = sTrans.substr(1);
        sTrans = sTrans.substr(0, sTrans.length() - 1);
        sTrans = str_trim(sTrans);
        //find sepration character '\t' or ' '
        iSepLoc = sTrans.find_last_of('\t');
        if (iSepLoc == string::npos)
            iSepLoc = sTrans.find_last_of(' ');
        if (iSepLoc == string::npos)
        {
            pPrj->SetErr(sDESLoc +
                        "No event or entering state for transition. (" +
                        sTrans + ")", HISC_BAD_DES_FORMAT);
            throw -1;
        }
        else
        {
            sEnterState = str_trim(sTrans.substr(iSepLoc + 1));
            sTransEvent = str_trim(sTrans.substr(0, iSepLoc));
        }
        //Check event in transitions
        if (m_DESEventsMap.find(sTransEvent) == m_DESEventsMap.end())
        {
            pPrj->SetErr(sDESLoc + "Event " + sTransEvent +
                    " in transitions does not exist.",
                    HISC_BAD_DES_FORMAT);
            throw -1;
        }
        iTransEventIndex = m_DESEventsMap[sTransEvent];
        m_UnusedEvents.erase(sTransEvent);
        //Check entering state
        if (m_StatesMap.find(sEnterState) == m_StatesMap.end())
        {
            pPrj->SetErr(sDESLoc + "State " + sEnterState +
                    " in transitions does not exist.",
                    HISC_BAD_DES_FORMAT);
            throw -1;
        }
        iEnterStateIndex = m_StatesMap[sEnterState];
        //Check determinacy
        if (m_pTransArr[viExitState].find(iTransEventIndex) !=
            m_pTransArr[viExitState].end())
```

```
        {
            pPrj->SetErr(sDESLoc + "ExitState:" + vsExitState +
                    " has nondeterministic transitions on event " + sTransEvent,
                    HISC_BAD_DES_FORMAT);
            throw -1;
        }
        m_pTransArr[viExitState][iTransEventIndex] = iEnterStateIndex;
    }
    catch(int)
    {
        return -1;
    }
    return 0;
}
/*
 * DESCR:   Print this DES in memory to a file (for checking)
 * PARA:    fout: file stream(input)
 * RETURN:  0: success -1: fail
 * ACCESS:  public
 */
int CDES::PrintDES(ofstream & fout)
{
    try
    {
        int i = 0;
        fout << endl << "#-------DES: " << m_sDESName << " ---------" << endl;
        fout << "[States]" << endl;
        fout << m_iNumofStates << endl;
        for (INVSTATES::const_iterator ci = m_InvStatesMap.begin();
             ci != m_InvStatesMap.end(); ++ci)
        {
            fout << ci->second << endl;
        }
        fout << endl;
        fout << "[InitState]" << endl;
        fout << m_InvStatesMap[m_iInitState] << endl;
        fout << endl;
        fout << "[MarkingStates]" << endl;
        for (MARKINGLIST::const_iterator ci = m_MarkingList.begin();
             ci != m_MarkingList.end(); ++ci)
        {
            fout << m_InvStatesMap[*ci] << endl;
        }
        fout << endl;
        fout << "[Events]" << endl;
        for (INVEVENTS::const_iterator ci = m_InvDESEventsMap.begin();
             ci != m_InvDESEventsMap.end(); ++ci)
        {
            if (ci->first % 2 == 0)  //uncontrollable
                fout << ci->second << "\t" << "N" << "\t" <<
                    SubValueToLetter((EVENTSUB)(ci->first >> 28)).at(0) << endl;
            else
                fout << ci->second << "\t" << "Y" << "\t" <<
                    SubValueToLetter((EVENTSUB)(ci->first >> 28)).at(0) << endl;
        }
        fout << endl;
        fout << "[Transitions]" << endl;
        if (m_pTransArr != NULL)
        {
            for (i = 0; i < m_iNumofStates; i++)
            {
                fout << m_InvStatesMap[i] << endl;
                for (TRANS::const_iterator ci = (m_pTransArr[i]).begin();
                     ci != (m_pTransArr[i]).end(); ++ci)
                {
                    fout << "(" << m_InvDESEventsMap[ci->first] << " "
                        << m_InvStatesMap[ci->second] << ")" << endl;
                }
            }
        }
        fout << "################################################" << endl;
    }
    catch(...)
    {
        return -1;
    }
    return 0;
}
```

```
/*************************************************************************
  FILE:  type.h
  DESCR: customized data types
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*************************************************************************/
#ifndef _TYPE_H_
#define _TYPE_H_
#include <map>
#include <list>
#include <string>
using namespace std;
#define MAX_LINE_LENGTH 513
#define COMMENT_CHAR '#'
#define MAX_STATES_IN_ONE_COMPONENT_DES 1024
#define MAX_DOUBLE 1.7e308
#define MAX_INT 2147483647
#define ALL_EVENT 99
#define MAX_PATH 255
enum DESTYPE {PLANT_DES = 0, SPEC_DES = 1, INTERFACE_DES = 2};
enum EVENTTYPE {CON_EVENT = 0, UNCON_EVENT = 1};
enum EVENTSUB {H_EVENT = 0, R_EVENT = 1, A_EVENT = 2, L_EVENT = 3};
typedef map<string, int> STATES; //state name, index
typedef map<int, string> INVSTATES; //state index, name
typedef map<string, int> EVENTS; //event name, global index
typedef map<int, string> INVEVENTS; //event global index, name
typedef map<string, unsigned short> LOCALEVENTS; //event name, level-wise index
typedef map<unsigned short, string> LOCALINVEVENTS;//event level-wise index,name
typedef list<int> MARKINGLIST; //link list to save all the marker states index
typedef map<int, int> TRANS; //source state index (key), target state index
#endif //_TYPE_H_
```

```
/*****************************************************************************
  FILE:  errmsg.h
  DESCR: constants for error code
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*****************************************************************************/
#ifndef ____ERRMSG_H____
#define ____ERRMSG_H____
#define HISC_BAD_PRJ_FILE -1
#define HISC_BAD_PRJ_FORMAT -2
#define HISC_BAD_HIGH_FILE -3
#define HISC_BAD_HIGH_FORMAT -4
#define HISC_BAD_LOW_FILE -5
#define HISC_BAD_LOW_FORMAT -6
#define HISC_BAD_DES_FILE -7
#define HISC_BAD_DES_FORMAT -8
#define HISC_SYSTEM_INITBDD -9
#define HISC_NONEXISTED_LEVEL_NAME -10
#define HISC_BAD_INTERFACE -11
#define HISC_LOWERR_GENCONBAD -20
#define HISC_LOWERR_GENP4BAD -21
#define HISC_LOWERR_SUPCP -22
#define HISC_LOWERR_COREACH -23
#define HISC_LOWERR_REACH -24
#define HISC_LOWERR_P5 -25
#define HISC_LOWERR_P6 -26
#define HISC_VERI_LOW_UNCON -201
#define HISC_VERI_LOW_BLOCKING -202
#define HISC_VERI_LOW_P4FAILED -203
#define HISC_VERI_LOW_P5FAILED -204
#define HISC_VERI_LOW_P6FAILED -205
#define HISC_HIGHERR_GENCONBAD -30
#define HISC_HIGHERR_GENP3BAD -31
#define HISC_HIGHERR_SUPCP -32
#define HISC_HIGHERR_COREACH -33
#define HISC_HIGHERR_REACH -34
#define HISC_VERI_HIGH_UNCON -101
#define HISC_VERI_HIGH_P3FAILED -102
#define HISC_VERI_HIGH_BLOCKING -103
#define HISC_BAD_SAVESUPER -97
#define HISC_BAD_PRINT_FILE -98
#define HISC_NOT_ENOUGH_MEMORY -99
#define HISC_WARN_BLOCKEVENTS -10000
#endif //____ERRMSG_H____
```

```
/*************************************************************************
  FILE:  pubfunc.h
  DESCR: Header file for pubfunc.cpp
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*************************************************************************/
#ifndef ____PUBFUNC_H____
#define ____PUBFUNC_H____
#include <string>
#include "type.h"
#include "errmsg.h"
#include <fdd.h>
using namespace std;
extern int giNumofBddNodes;
extern string str_trim(const string &str);
extern string str_upper(const string &str);
extern string str_lower(const string &str);
extern string str_itos(int iInt);
extern string str_ltos(long long lLong);
extern string str_nocomment(const string & str);
extern int scp_err(const string & sErr, const int iErrCode);
extern string GetNameFromFile(const string & vsFile);
extern EVENTSUB SubLetterToValue(char cEventSub);
extern string SubValueToLetter(EVENTSUB vEventSub);
extern int IsInteger(const string &str);
extern int CompareInt(const void* pa, const void* pb);
extern void bddPrintStats(const bddStat &stat);
extern void SetBddPairs(bddPair *pPair, const bdd & bddOld, const bdd & bddNew);
extern int NumofSharedEvents(const int * pEventsArr_a, const int viNumofEvents_a,
        const int * pEventsArr_b, const int viNumofEvents_b);
extern void my_bdd_gbchandler(int pre, bddGbcStat *s);
#endif //____PUBFUNC_H____
```

```
/*****************************************************************************
  FILE:  pubfunc.cpp
  DESCR: Containing some utility functions used in the program
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*****************************************************************************/
#include <string>
#include <iostream>
#include "type.h"
#include <stdio.h>
#include "pubfunc.h"
#include <cassert>
using namespace std;
int giNumofBddNodes = 0;
/**
 * DESCR:   Trim away all the prefix and suffix spaces or tabs of a string
 * PARA:    str: a string (input)
 * RETURN:  trimmed string
 * */
string str_trim(const string &str)
{
    string sTmp("");
    unsigned int i = 0;
    //trim off the prefix spaces
    for (i = 0; i < str.length(); i++)
    {
        if (str[i] != 32 && str[i] != 9)
            break;
    }
    if (i < str.length())
    {
        sTmp = str.substr(i);
    }
    else
    {
        return sTmp;
    }
    //trim off the suffix spaces
    for (i = sTmp.length() - 1; i >= 0; i--)
    {
        if (sTmp[i] != 32 && sTmp[i] != 9)
            break;
    }
    if (i >= 0)
    {
        sTmp = sTmp.substr(0, i + 1);
    }
    else
    {
        sTmp.clear();
    }
    return sTmp;
}
/**
 * DESCR:   convert all the letters in a string to uppercase
 * PARA:    str: a string (input)
 * RETURN:  converted string
 * */
string str_upper(const string &str)
{
    unsigned int i = 0;
    string sTmp(str);
    for (i = 0; i < str.length(); i++)
    {
        if (sTmp[i] >= 'a' & sTmp[i] <= 'z')
        {
            sTmp[i] = sTmp[i] - 32;
        }
    }
    return sTmp;
}
/**
 * DESCR:   convert all the letters in a string to lowercase
 * PARA:    str: a string (input)
 * RETURN:  converted string
 * */
string str_lower(const string &str)
{
    unsigned int i = 0;
```

```
    string sTmp(str);
    for (i = 0; i < str.length(); i++)
    {
        if (sTmp[i] >= 'A' & sTmp[i] <= 'Z')
        {
            sTmp[i] = sTmp[i] + 32;
        }
    }
    return sTmp;
}
/**
 * DESCR:   convert an integer to a string
 * PARA:    iInt: an integer
 * RETURN:  converted string
 * */
string str_itos(int iInt)
{
    char scTmp[65];
    string str;
    sprintf(scTmp, "%d", iInt);
    str = scTmp;
    return str;
}
/**
 * DESCR:   convert a long integer to a string
 * PARA:    iInt: a long integer
 * RETURN:  converted string
 * */
string str_ltos(long long lLong)
{
    char scTmp[65];
    string str;
    sprintf(scTmp, "%lld", lLong);
    str = scTmp;
    return str;
}
/**
 * DESCR:   trim off all the characters after a COMMENT_CHAR
 * PARA:    str : a string
 * RETURN:  processed string
 * */
string str_nocomment(const string & str)
{
    int i;
    int iLen = str.length();
    for (i = 0; i < iLen; i++)
    {
        if (str[i] == COMMENT_CHAR)
            break;
    }
    if (i < iLen)
        return str.substr(0, i);
    else
        return str;
}
/**
 * DESCR:   Get sub name or des name from a full path file name
 *          with extension ".sub"/".hsc"
 *          ex: vsFile = "/home/roger/m1.sub" will return "m1"
 * PARA:    vsFile: file name with path
 * RETURN:  sub name or des name
 * */
string GetNameFromFile(const string & vsFile)
{
    assert(vsFile.length() > 4);
    assert(vsFile.substr(vsFile.length() - 4) == ".sub" ||
           vsFile.substr(vsFile.length() - 4) == ".hsc");
    unsigned int iPos = vsFile.find_last_of('/');
    if ( iPos == string::npos)
    {
        return vsFile.substr(0, vsFile.length() - 4);
    }
    else
    {
        return vsFile.substr(iPos + 1, vsFile.length() - 4 - (iPos + 1));
    }
}
/**
 * DESCR:   Convert 'H', 'R', 'A', 'L' to corresponding EVENTSUB value
```

```
 * PARA:    cEventSub ('H', 'R', 'A', 'L')
 * RETURN:  corresponding EVENTSUB value
 * */
EVENTSUB SubLetterToValue(char cEventSub)
{
    assert(cEventSub == 'H' || cEventSub == 'R' ||
            cEventSub == 'A' || cEventSub == 'L');
    switch (cEventSub)
    {
    case 'H':
        return H_EVENT;
        break;
    case 'R':
        return R_EVENT;
        break;
    case 'A':
        return A_EVENT;
        break;
    case 'L':
        return L_EVENT;
        break;
    }
    return H_EVENT;
}
/**
 * DESCR:   Convert EVENTSUB value to corresponding name
 * PARA:    vEventSub: EVENTSUB value
 * RETURN:  corresponding name
 * */
string SubValueToLetter(EVENTSUB vEventSub)
{
    switch (vEventSub)
    {
    case H_EVENT:
        return "HIGH SUB";
        break;
    case R_EVENT:
        return "REQUEST";
        break;
    case A_EVENT:
        return "ANSWER";
        break;
    case L_EVENT:
        return "LOW SUB";
        break;
    }
    return "HIGH SUB";
}
/**
 * DESCR:   Test if a string could be converted to an integer
 * PARA:    str: a string
 * RETURN:  0: no 1: yes
 * */
int IsInteger(const string &str)
{
    if (str.length() == 0)
        return 0;
    for (unsigned int i = 0; i < str.length(); i++)
    {
        if (str[i] >= '0' && str[i] <= '9')
            continue;
        else
            return 0;
    }
    return 1;
}
/**
 * DESCR:   Compare two integers which are provided by two general pointers.
 *          qsort, bsearch will use this function
 * PARA:    pa, pb: general pointers pointing to two integers
 * RETURN:  1: a>b
 *          0: a=b
 *         -1: a<b
 * */
int CompareInt(const void* pa, const void* pb)
{
    int a = *((int *) pa);
    int b = *((int *) pb);
    if (a > b)
        return 1;
    else if (a < b)
        return -1;
    else
        return 0;
}
```

```
/**
 * DESCR:   To print the content of a bddStat variable.
 *          Original BDD package doesn't provide such a function.
 * PARA:    bddStat: see documents of Buddy package
 * RETURN:  None
 * */
void bddPrintStats(const bddStat &stat)
{
    cout << endl;
    cout << "--------------bddStat-----------------" << endl;
    cout << "Num of new produced nodes: " << stat.produced << endl;
    cout << "Num of allocated nodes: " << stat.nodenum << endl;
    cout << "Max num of user defined nodes: " << stat.maxnodenum << endl;
    cout << "Num of free nodes: " << stat.freenodes << endl;
    cout << "Min num of nodes after garbage collection: " << stat.minfreenodes
         << endl;
    cout << "Num of vars:" << stat.varnum << endl;
    cout << "Num of entries in the internal caches:" << stat.cachesize << endl;
    cout << "Num of garbage collections done until now:" << stat.gbcnum << endl;
    return;
}
/**
 * DESCR:   Set bddpairs based on two bdd variable sets.
 *          The original function bdd_setbddpair(...) is not
 *          as the document said.
 * PARA:    pPair: where to add bdd variable pairs
 *          bddOld: variable will be replaced
 *          bddNew: new variable
 * RETURN:  None
 * */
void SetBddPairs(bddPair *pPair, const bdd & bddOld, const bdd & bddNew)
{
    assert(pPair != NULL);
    int *vOld = NULL;
    int *vNew = NULL;
    int nOld = 0;
    int nNew = 0;
    bdd_scanset(bddOld, vOld, nOld);
    bdd_scanset(bddNew, vNew, nNew);
    assert(nOld == nNew);
    for (int i = 0; i < nOld; i++)
    {
        bdd_setpair(pPair, vOld[i], vNew[i]);
    }
    free(vOld);
    free(vNew);
    return;
}
/**
 * DESCR:   Compute the number of shared events between two DES
 * PARA:    pEventsArr_a:   Event array for DES a (global index, sorted)
 *          viNumofEvents_a: Number of events in array pEventsArr_a
 *          pEventsArr_b:   Event array for DES b (global index, sorted)
 *          viNumofEvents_b: Number of events in array pEventsArr_b
 * RETURN:  Number of shared events
 * */
int NumofSharedEvents(const int * pEventsArr_a, const int viNumofEvents_a,
        const int * pEventsArr_b, const int viNumofEvents_b)
{
    int iNum = 0;
    int i = 0;
    assert(pEventsArr_a != NULL);
    assert(pEventsArr_b != NULL);
    if (viNumofEvents_a <= viNumofEvents_b)
    {
        for (i = 0; i < viNumofEvents_a; i++)
        {
            if (bsearch(&(pEventsArr_a[i]), pEventsArr_b, viNumofEvents_b,
                        sizeof(int), CompareInt) != NULL)
            {
                iNum++;
            }
        }
    }
    else
    {
        for (i = 0; i < viNumofEvents_b; i++)
        {
            if (bsearch(&(pEventsArr_b[i]), pEventsArr_a, viNumofEvents_a,
                        sizeof(int), CompareInt) != NULL)
```

```
            {
                iNum++;
            }
        }
    }
    return iNum;
}
/**
 * DESCR:   Customized Garbage collection handler for this program
 * PARA:    see document of Buddy Package
 * RETURN:  None
 * */
void my_bdd_gbchandler(int pre, bddGbcStat *s)
{
   if (!pre)
   {
      if (s->nodes > giNumofBddNodes)
      {
          printf("Garbage collection #%d: %d nodes / %d free",
             s->num, s->nodes, s->freenodes);
          printf(" / %.1fs / %.1fs total\n",
             (float)s->time/(float)(CLOCKS_PER_SEC),
             (float)s->sumtime/(float)CLOCKS_PER_SEC);
          giNumofBddNodes = s->nodes;
      }
   }
   return;
}
```

```
/*************************************************************************
  FILE:  main.h
  DESCR: Header file for main.cpp
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*************************************************************************/
int getchoice(bool bPrjLoaded, const char *scPrjFile);
char getkeystroke(char *allowed_choices, int len);
int getchoice_savesup();
int getchoice_saveproduct();
int getchoice_tracetype();
```

```
/*************************************************************************
  FILE:  main.cpp
  DESCR: An example with very simple text interface to show how to use the
         HISC verification and synthesis library.
  AUTH:  Raoguang Song
  DATE:  (C) Jan, 2006
*************************************************************************/
#include "main.h"
#include <iostream>
#include <cstdio>
#include <string>
#include "../bddhisc/BddHisc.h"
#include <termios.h>
#include <unistd.h>
using namespace std;
#define MAX_PATH 256
#define MAX_ERRMSG_LENGTH 4096
int main()
{
    bool bPrjLoaded = false;
    char ch = '\0';
    char prjfile[MAX_PATH];
    char errmsg[MAX_ERRMSG_LENGTH];
    prjfile[0] = '\0';
    int iret = 0;
    char prjoutputfile[MAX_PATH];
    char subname[MAX_PATH];
    int nextlow = 0;
    char savepath[MAX_PATH];
    savepath[0] = '\0';
    HISC_SUPERINFO superinfo;
    HISC_SAVESUPERTYPE savesupertype;
    HISC_SAVEPRODUCTTYPE saveproducttype;
    HISC_TRACETYPE tracetype;
    int computetime = 0;
    init_hisc();
    while (ch != 'q' && ch != 'Q')
    {
        ch = getchoice(bPrjLoaded, prjfile);
        switch (ch)
        {
            case 'q':
            case 'Q':
                break;
            //Load a project
            case 'P':
            case 'p':
                cout << "Project name:";
                cin.getline(prjfile, MAX_PATH);
                iret = load_prj(prjfile, errmsg);
                if (iret < 0)
                {
                    if (iret > -10000) //error
                        bPrjLoaded = false;
                    else
                        bPrjLoaded = true; //waring
                }
                else
                    bPrjLoaded = true;
                break;
            //close the current project
            case 'c':
            case 'C':
                iret = close_prj(errmsg);
                bPrjLoaded = false;
                prjfile[0] = '\0';
                break;
            //File the current project
            case 'f':
            case 'F':
                cout << "file name:";
                cin.getline(prjoutputfile, MAX_PATH);
                iret = print_prj(prjoutputfile, errmsg);
                break;
            //Low Level synthesis
            case 'o':
            case 'O':
                cout << "low level name:";
                cin.getline(subname, MAX_PATH);
                cout << "Save the supervisor? (0:none  1:bdd-based " <<
                     "2:automata-based  3:both)" << endl;
                cout << "Warning: choose 2 or 3 may generate huge file."<< endl;
```

```
    cout << "Your choice:";
    savesupertype = (HISC_SAVESUPERTYPE)getchoice_savesup();
    if (savesupertype != HISC_SAVESUPER_NONE)
    {
        cout << "Path to save the supervisor:";
        cin.getline(savepath, MAX_PATH);
    }
    nextlow = 0;
    computetime = 0;
    do
    {
        superinfo.statesize = -1;
        superinfo.nodesize = -1;
        superinfo.time = 0;
        iret = syn_lowsuper(HISC_ONREACHABLE, subname, errmsg,
                            &superinfo, &nextlow,
                            savesupertype, savepath);
        cout << "Low Level Name: " << subname << endl;
        if (iret == 0)
        {
            cout << "Supervisor state size: " <<
                                superinfo.statesize << endl;
            cout << "Number of bdd nodes to store the supervisor: "
                                << superinfo.nodesize << endl;
            cout << "Computing time:" << superinfo.time <<
                                " seconds." << endl;
            computetime += superinfo.time;
        }
        else
        {
            cout << errmsg << endl;
            cout << "Press any key to continue...";
            iret = 0;
            errmsg[0] = '\0';
            getkeystroke(NULL, 0);
        }
    } while (nextlow > 0);
    cout << "Total computing time:" << computetime <<
                " seconds." << endl;
    break;
//Low Level verification
case 'l':
case 'L':
    cout << "low level name:";
    cin.getline(subname, MAX_PATH);
    cout << "Save the synchronous product" <<
            "(may generate huge file)(Y/N)? ";
    saveproducttype = (HISC_SAVEPRODUCTTYPE)getchoice_saveproduct();
    if (saveproducttype != HISC_NOTSAVEPRODUCT)
    {
        cout << "Path to save the synchronous product:";
        cin.getline(savepath, MAX_PATH);
    }
    cout << "Show the blocking type(may take long time)(Y/N)?";
    tracetype = (HISC_TRACETYPE)getchoice_tracetype();
    nextlow = 0;
    computetime = 0;
    do
    {
        superinfo.statesize = -1;
        superinfo.nodesize = -1;
        superinfo.time = 0;
        iret = verify_low(tracetype, subname, errmsg, &superinfo,
                    &nextlow, saveproducttype, savepath);
        cout << "Low Level Name: " << subname << endl;
        if (iret == 0)
            cout << "This low level has been verified succesfully!"
                 << endl;
        if (superinfo.statesize >= 0)
            cout << "State size of the synchronous product: " <<
                    superinfo.statesize << endl;
        if (superinfo.nodesize >= 0)
            cout << "Number of bdd nodes to store" <<
                " the synchronous product: " << superinfo.nodesize
                << endl;
        cout << "Computing time: " << superinfo.time <<
                " seconds." << endl;
        computetime += superinfo.time;
        if (iret < 0)
        {
```

```
                cout << errmsg << endl;
                cout << "Press any key to continue...";
                iret = 0;
                errmsg[0] = '\0';
                getkeystroke(NULL, 0);
            }
        } while (nextlow > 0);
        cout << "Total computing time:" << computetime << " seconds."
                << endl;
        break;
    //High Level synthesis
    case 'i':
    case 'I':
        cout << "Save the supervisor? (0:none  1:bdd-based  " <<
                "2:automata-based  3:both)" << endl;
        cout << "Warning: choose 2 or 3 may generate huge file." <<endl;
        cout << "Your choice:";
        savesupertype = (HISC_SAVESUPERTYPE)getchoice_savesup();
        if (savesupertype != HISC_SAVESUPER_NONE)
        {
            cout << "Path to save the supervisor:";
            cin.getline(savepath, MAX_PATH);
        }
        superinfo.statesize = -1;
        superinfo.nodesize = -1;
        superinfo.time = 0;
        iret = syn_highsuper(HISC_ONREACHABLE, errmsg, &superinfo,
                             savesupertype, savepath);
        if (iret == 0)
        {
            cout << "Supervisor state size: " <<
                    superinfo.statesize << endl;
            cout << "Number of bdd nodes to store the supervisor: " <<
                    superinfo.nodesize << endl;
            cout << "Computing time:" << superinfo.time <<
                    " seconds." << endl;
        }
        break;
    //High Level verification
    case 'h':
    case 'H':
        cout << "Save the synchronous product" <<
                "(may generate huge file)(Y/N)? ";
        saveproducttype = (HISC_SAVEPRODUCTTYPE)getchoice_saveproduct();
        if (saveproducttype != HISC_NOTSAVEPRODUCT)
        {
            cout << "Path to save the synchronous product:";
            cin.getline(savepath, MAX_PATH);
        }
        cout << "Show the blocking type(may take long time)(Y/N)?";
        tracetype = (HISC_TRACETYPE)getchoice_tracetype();
        superinfo.statesize = -1;
        superinfo.nodesize = -1;
        superinfo.time = 0;
        iret = verify_high(tracetype, errmsg, &superinfo,
                saveproducttype, savepath);
        if (iret == 0)
            cout << "The high level has been verified succesfully!"
                 << endl;
        if (superinfo.statesize >= 0)
            cout << "State size of the synchronous product: "
                << superinfo.statesize << endl;
        if (superinfo.nodesize >= 0)
            cout << "Number of bdd nodes to store" <<
            " the synchronous product: " << superinfo.nodesize << endl;
        cout << "Computing time: " << superinfo.time
            << " seconds." << endl;
        break;
    }
    if (iret < 0)
    {
        cout << errmsg << endl;
        cout << "Press any key to continue...";
        iret = 0;
        errmsg[0] = '\0';
        getkeystroke(NULL, 0);
    }
}
close_hisc();
```

```
    return 0;
}
int getchoice(bool bPrjLoaded, const char *prjfile)
{
    char allowed_choice[50];
    int numofchoice = 0;
    cout << endl << endl << endl << endl << endl;
    cout << "************************************************" << endl;
    cout << "  Bdd-based HISC Synthesis and Verification Tool " << endl;
    cout << "************************************************" << endl;
    if (!bPrjLoaded)
    {
        allowed_choice[0] = 'p';
        allowed_choice[1] = 'P';
        allowed_choice[2] = 'q';
        allowed_choice[3] = 'Q';
        numofchoice = 4;
        cout << "  P - Load a HISC project                       " << endl;
    }
    else
    {
        allowed_choice[0] = 'c';
        allowed_choice[1] = 'C';
        allowed_choice[2] = 'q';
        allowed_choice[3] = 'Q';
        allowed_choice[4] = 'h';
        allowed_choice[5] = 'H';
        allowed_choice[6] = 'i';
        allowed_choice[7] = 'I';
        allowed_choice[8] = 'F';
        allowed_choice[9] = 'f';
        allowed_choice[10] = 'O';
        allowed_choice[11] = 'o';
        allowed_choice[12] = 'l';
        allowed_choice[13] = 'L';
        numofchoice = 14;
        cout << "  H - High level verification                  " << endl;
        cout << "  I - High level synthesis                     " << endl;
        cout << "  L - Low Level verification                   " << endl;
        cout << "  O - Low Level synthesis                      " << endl;
        cout << "  F - File the current project                 " << endl;
        cout << "  C - Close the current project                " << endl;
    }
    cout << "  Q - Quit                                      " << endl;
    cout << "************************************************" << endl;
    if (bPrjLoaded)
    {
        cout << "Current Project: " << prjfile << endl;
    }
    cout << endl;
    cout << "Procedure desired:";
    return getkeystroke(allowed_choice, numofchoice);
}
char getkeystroke(char *allowed_choices, int len)
{
    char choice;
    struct termios initial_settings, new_settings;
    //cin.ignore(255, '\n');
    tcgetattr(fileno(stdin), &initial_settings);
    new_settings = initial_settings;
    new_settings.c_lflag &= ~ICANON;
    //new_settings.c_lflag &= ~ECHO;
    new_settings.c_cc[VMIN] = 1;
    new_settings.c_cc[VTIME] = 0;
    new_settings.c_lflag &= ~ISIG;
    tcsetattr(fileno(stdin), TCSANOW, &new_settings);
    if (len > 0)
    {
        do {
            choice = fgetc(stdin);
            int i;
            for (i = 0; i < len; i++)
            {
                if (choice == allowed_choices[i])
                    break;
            }
            if (i == len)
                choice = '\n';
        } while (choice == '\n' || choice == '\r');
    }
```

```
    else
        choice = fgetc(stdin);
    tcsetattr(fileno(stdin),TCSANOW, &initial_settings);
    cout << endl;
    return choice;
}
int getchoice_savesup()
{
    char allowed_choice[50];
    int numofchoice = 0;
    char choice;
    allowed_choice[0] = '0';
    allowed_choice[1] = '1';
    allowed_choice[2] = '2';
    allowed_choice[3] = '3';
    numofchoice = 4;
    choice = getkeystroke(allowed_choice, numofchoice);
    return choice - '0';
}
int getchoice_saveproduct()
{
    char allowed_choice[50];
    int numofchoice = 0;
    char choice;
    allowed_choice[0] = 'Y';
    allowed_choice[1] = 'y';
    allowed_choice[2] = 'N';
    allowed_choice[3] = 'n';
    numofchoice = 4;
    choice = getkeystroke(allowed_choice, numofchoice);
    if (choice == 'Y' || choice == 'y')
        return 1;
    else
        return 0;
}
int getchoice_tracetype()
{
    char allowed_choice[50];
    int numofchoice = 0;
    char choice;
    allowed_choice[0] = 'Y';
    allowed_choice[1] = 'y';
    allowed_choice[2] = 'N';
    allowed_choice[3] = 'n';
    numofchoice = 4;
    choice = getkeystroke(allowed_choice, numofchoice);
    if (choice == 'Y' || choice == 'y')
        return 1;
    else
        return 0;
}
```