Synthesis Method for Hierarchical Interface-based Supervisory Control

Version 1.01

By
Pengcheng Dai, B.Eng.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Master of Applied Science
Department of Computing and Software
McMaster University

MASTER OF APPLIED SCIENCE(2006) McMaster University
(Software Engineering) Hamilton, Ontario

TITLE: Synthesis Method for
Hierarchical Interface-based Supervisory Control

AUTHOR: Pengcheng Dai, B.Eng.(Tianjin University)

SUPERVISOR: Dr. Ryan J. Leduc

NUMBER OF PAGES: viii, 212

Abstract

Hierarchical Interface-based Supervisory Control (HISC) decomposes a discrete-event system (DES) into a high-level subsystem which communicates with $n \geq 1$ low-level subsystems, through separate interfaces which restrict the interaction of the subsystems. It provides a set of local conditions that can be used to verify global conditions such as nonblocking and controllability. As each clause of the definition can be verified using a single subsystem, the complete system model never needs to be stored in memory, offering potentially significant savings in computational resources.

Currently, a designer must create the supervisors for a HISC system himself, and then verify that they satisfy the HISC conditions. In this thesis, we develop a synthesis method that respects the HISC hierarchical structure. We replace the supervisor for each level by a corresponding specification DES. We then do a per level synthesis to construct for each level a maximally permissive supervisor that satisfies the corresponding HISC conditions.

We define a set of language based fixpoint operators and show that they compute the required level-wise supremal languages. We then present algorithms that implement the fixpoint operators. We present a complexity analysis for the algorithms and show that they potentially offer significant improvement over the monolithic approach.

A large manufacturing system example (estimated worst case state space on the order of 10^{22}) extended from the AIP example is discussed. A software tool for synthesis and verification of HISC systems using our approach was also developed.

Acknowledgments

I am full of gratitude to my supervisor, Dr. R.J. Leduc. With his enthusiasm, his inspiration, and his great efforts to explain complicated things clearly and simply, he helped to make this work fun for me. Throughout my masters study and research, he provided sound advice, good teaching and lots of good ideas. I would have been lost without him and this work can never be done without his full support and continuous encouragement.

Contents

List of Figures

Chapter 1

Introduction

In the area of Discrete-Event Systems (DES), two common tasks are to verify that a composite system, based on a Cartesian product of subsystems, is (i) nonblocking and (ii) controllable. The main obstacle to performing these tasks is the combinatorial explosion of the product state space.

The Hierarchical Interface-based Supervisory Control(HISC) framework was proposed by Leduc *et al.* in [34, 35, 36, 37, 33] to alleviate the state explosion problem. The HISC approach decomposes a system into a *high-level subsystem* which communicates with $n \geq 1$ parallel *low-level subsystems* through separate interfaces that restrict the interaction of the subsystems. It provides a set of local conditions that can be used to verify global conditions such as nonblocking and controllability. As each clause of the definition can be verified using a single subsystem, the complete system model never needs to be stored in memory, offering potentially significant savings in computational resources.

Currently, a designer must create the supervisors for a HISC system himself, and then verify that they satisfy the HISC conditions. If they do not, he must modify them until they do satisfy the conditions. For a complex system, it may be very non obvious how to achieve this. Also, the resulting supervisors may be more

restrictive than they need to be. In this thesis, we develop a synthesis method that respects the HISC hierarchical structure. We replace the supervisor for each level by a corresponding specification DES. We then do a per level synthesis to construct for each level a maximally permissive supervisor that satisfies the corresponding HISC conditions. We then develop a set of algorithms to implement these fixpoint operators. As the synthesis will be done on a per level basis, the complete system model never needs to be constructed. We thus expect to see similar savings in computation as in the HISC verification method. This savings should be even more pronounced as synthesis is an iterative process, thus typically requiring much more computation.

1.1 Research Review

Researchers in supervisory control have recently begun to advocate interface based architectural solutions to dealing with complexity [38, 39, 41, 22].[1] These approaches develop interfaces between components to provide structure that guarantees global properties such as controllability [39, 41, 22] or nonblocking [38, 39, 41]. The most significant feature that distinguishes the HISC approach from [22] is the results on nonblocking, although Endsley et al. later extended their work to include a form of deadlock detection in [23].

In [21] interface automata are used to model software components and verify their compatibility. This work has independently derived conditions for software component interface compatibility that are similar to the HISC interface consistency properties. In [21], automata representing component interfaces are directly composed to produce the interface of the new composite component and a refinement relation is developed to aid in refining a component interface specification into an implementation. There is no explicit concept of control, though implic-

[1] This literature review is based heavily upon the review in [36], with permission.

itly component inputs are considered uncontrollable and the component outputs are effectively controllable. In contrast HISC uses an interface automaton that mediates communication between the components in order to decompose the verification of global nonblocking and controllability into "local" checks on each of the components and their interface.

Related work by Fabian *et al.* [25, 26] applied object-oriented concepts in the design of DES control software, and extended supervisory control theory to the nondeterministic supervisors which that approach required. Later, Shayman *et al.* [64] introduced the concept of control and observation masks to encapsulate process logic. These approaches have two disadvantages relative to interface based supervisory control: (i) they do not address issues related to nonblocking and (ii) they require a more complex mathematical setting than the deterministic automata with synchronous product operator that is commonly employed in supervisory control theory. By using interface DES to regulate subsystem interaction, we are able to impose architecture without change to the standard DES setting.

One of the earliest and most useful methods designed to handle the combinatorial explosion of the product state space that results from systems composed of interacting subsystems is *modular control* [79, 20, 58, 67]. This method involves designing multiple supervisors as opposed to a centralized supervisor, each supervisor implementing a portion of the control specification. While the method scales well in practice for the verification of controllability (see e.g. [3, 43]), verifying nonblocking of the closed loop system is still a problem.

In *Decentralized control* [7, 45, 62, 63, 75, 80, 4], local supervisors, with only partial observations of the plant, are designed as a group to implement a global specification. While this is an effective method to design distributed controllers, it still requires the computation of the synchronous product of all of the plant subcomponents (the composite plant) and thus offers no computational savings over a centralized solution.

One way to improve the scalability of modular and decentralized schemes is to exploit the existing architecture of the system. In [73] the concept of a specification that is separable over the component subsystems is introduced and shown to be necessary and sufficient for a decentralized control scheme to exist that optimally meets the specification. The work does not consider nonblocking supervision. These results are extended to a more general architecture in [1] that deals with nonblocking by detecting potential blocking states locally and then backtracking globally to determine their reachability. The structure associated with the event sets of subsystems is exploited in [58] to obtain a reduction in complexity for the non-conflicting check of modular control. Similarly the standard controllability definition has been refined and localized in [2] to check on a per subplant basis only those uncontrollable events that can occur locally.

Another approach is embodied by *Vector DES* (VDES) [79, 17, 44] and *Petri Nets* (PN) [50, 86, 87]. These state based methods make use of the algebraic regularity inherent in certain systems. They are used when the state of the system can be represented as a vector of integers, whose components are incremented or decremented by events. These methods are primarily useful for systems with a high degree of regularity that lend themselves to vector representation. However, the VDES/PN models are not well adapted to the synthesis or verification of nonblocking controllers without first converting the models to automata by means of the reachability graph [70].

A promising approach is the development of a multi-level hierarchy. In order to aid in classification, we make a distinction between structural multi-level hierarchies with explicit mechanisms (modeling constructs) to facilitate hierarchy (e.g. [10, 28, 72, 47]) as opposed to aggregate (bottom up) multi-level hierarchies which we will discuss later. In structural multi-level hierarchies, plants and supervisors are modeled as multi-level structures similar to automata, except that certain states at a given level can be expanded into a more detailed lower level model.

Although [72] allowed a system to be represented hierarchically using Cartesian products (AND superstates) or disjoint unions (OR superstates), AND states had to be converted to OR states using the synchronous product before computations could be effectively performed. Similarly, [28] was restricted to using only OR states. Both approaches could verify controllability, but did not address nonblocking. Recently, these limitations have been overcome by Ma *et al.* [47, 48] who, with the use of *binary decision diagrams* (BDDs) [11], has been able to verify controllability and nonblocking for a system on the order of 10^{24} states.

The next approach of interest is the model aggregation methods [5, 13, 16, 18, 24, 55, 56, 65, 85, 74, 69]. In these approaches, aggregate models are derived from low level models by using either state-based or language-based aggregation methods. Although this approach can be effective in constructing high level models with reduced state spaces, they have some drawbacks:

- In hierarchical methods such as [85, 74, 55], there is no direct connection between control actions at the high level, and at lower levels. To create an implementation, a control action at the high-level may need to be "interpreted" as equivalent control action(s) at the low level.

- Aggregate models must be constructed sequentially from the bottom up, starting from the lowest level; thus a given level cannot be constructed and verified in parallel with the levels below it, making a distributed design process difficult.

- The DES methods provide necessary and sufficient conditions for checking controllability, and in many cases nonblocking, using the aggregate models. While this is desirable, it causes the individual levels to be tightly coupled; a change made to the lowest level may require that all aggregate models and results have to be re-evaluated. In contrast, the sufficient conditions of interface based supervisory control that we develop allow us to design

and verify levels independently, ensuring that a change to one level of the hierarchy will not impact the others. This independence comes at the cost of possible false negatives forcing an overly conservative design.

We also note the related work in hybrid systems of Moor *et al.* [51] who have developed a multi-level aggregation approach inspired by [85, 74]. This new approach is different as they use an input/output structure to represent both time and event driven system dynamics, allowing them to verify both controllability and nonblocking results.

In contrast to the majority of approaches which apply mathematical techniques to produce aggregate models of an existing system, our method of restricting component interaction to well defined interfaces provides a design heuristic to guarantee scalability by construction.

The last approach we discuss is the use of symbolic methods to represent the transition structures underlying DES [29]. Zhang *et al.* [83, 84] as well as Vahidi et al [71] have developed algorithms that use *integer decision diagrams* (an extension of BDDs) to verify centralized DES systems on the order of 10^{23} states. That work builds on results of symbolic model checking [12, 49] that have successfully used BDDs to handle systems of similar size.

While this thesis was being written, research work on using binary decision diagrams to verify HISC properties was carried on by Song [66] independently. This built upon the work in this thesis, allowing the HISC method to be applied to even larger systems.

Finally we note that the interface DES that support the HISC system architecture differ from the "interface processes" employed in compositional model checking [8]. In the latter, an interface process is an aggregate model that is used as a replacement for a particular subsystem to produce a reduced state model that facilitates verification. For example, let $\mathbf{P}_i, i = 1, 2$ be subsystem models and ψ

be the temporal logic formula of interest. In order to verify that $\mathbf{P}_1 \parallel \mathbf{P}_2 \models \psi$ by compositional model checking, $\mathbf{P}_2$ might be replaced by an aggregate "interface process" $\mathbf{A}_2$ such that if $\mathbf{P}_1 \parallel \mathbf{A}_2 \models \psi$ then $\mathbf{P}_1 \parallel \mathbf{P}_2 \models \psi$.

1.2 Thesis Overview

The thesis is organized as follows. Chapter 1 gives an introduction to the background of this work and outlines the structure of this thesis. Chapter 2 gives an introduction to the basics of discrete event system and the supervisory control theory.

Chapter 3 introduces the hierarchical interface-based supervisory control theory and definitions. We discuss a new set of definitions, first introduced in [40], that are more concise and easier to implement than the original ones. In Chapter 4, we prove that these new definitions are equivalent to the original definitions given in [34, 36, 37].

Chapter 5 defines the synthesis method for high and low level subsystems, and a set of fixpoint operators that implement the synthesis method. We prove they compute the required level-wise supremal languages.

In Chapter 6, we present our algorithms to verify the interface consistency properties, and implement the fixpoint operators for synthesis. For each algorithm we perform a complexity analysis.

In Chapter 7, We rework the AIP example from [33, 34, 40] and apply our software to it. Finally we conclude our work in Chapter 8.

Chapter 2

DES Overview

RW supervisory control theory [59, 77, 79] provides a framework to model and control the behavior of Discrete Event Systems, and it is the basis of this work. In this chapter, we will give a brief introduction of the theory, including concepts of languages, automata, supervisory control and a few common operators over DES such as meet, sync and supcon.

2.1 Language

Let *alphabet* Σ be a non-empty finite set of distinct symbols, such as α, β, and so on. A *string* is a finite symbol sequence over Σ, such as $\alpha\alpha\beta$. We denote the *empty string* (a string with no symbols) as ϵ and the set of all non-empty strings over Σ as Σ^+. We then extend this to include ϵ as below

$$\Sigma^* = \{\epsilon\} \cup \Sigma^+.$$

We say that L is a *language* over Σ if $L \subseteq \Sigma^*$, and we denote the set of all sublanguages of Σ^* as $Pwr(\Sigma^*)$ We then have that $(Pwr(\Sigma^*), \subseteq)$ is a *poset*, i.e., the relation $\subseteq$ is reflexive, transitive and antisymmetric on $Pwr(\Sigma^*)$. The operations $\cap$ and $\cup$ of any two elements in $Pwr(\Sigma^*)$ always exists, thus $(Pwr(\Sigma^*), \cap, \cup)$ is a

lattice. When there always exists a greatest lower bound and least upper bound for each subset in the lattice, we say the lattice is *complete*.

Let $(X, \leq)$ be a poset. We say a function $f : X \to X$ is *monotone* if

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

We say an element $x \in X$ is a *fixpoint* of f if $f(x) = x$. Further, we say x is the *greatest fixpoint* of f if

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

We will also use the notation $f^i(x)$, $i \in \{0, 1, 2, \ldots\}$, to mean i applications of f in a row with $f^0(x) := x$. i.e. $f^1(x) = f(x)$, $f^2(x) = f(f(x))$ and so on.

Let $t, s \in \Sigma^*$. We say that t is a *prefix* of s and write $t \leq s$, if $s = tu$ for some $u \in \Sigma^*$. We also say that t can be extended to s. We can now define the *extension* operator.

Definition 2.1.1 *For language $L \subseteq \Sigma^*$, we define the function $Ext_L : Pwr(\Sigma^*) \to Pwr(\Sigma^*)$, for arbitrary $K \in Pwr(\Sigma^*)$ as follows:*

$$Ext_L(K) := \{t \in L |\, s \leq t \text{ for some } s \in K\}$$

In essence, $\mathrm{Ext}_L(K)$ is the set of all strings in L that have prefixes in K. If we have $K \subseteq L$, we would then have $K \subseteq \mathrm{Ext}_L(K)$ as $s \leq s$.

The *prefix closure* of language L, denoted $\overline{L}$ is the language consisting of all prefixes of strings of L, and is defined as follows:

$$\overline{L} = \{t \in \Sigma^* |\, t \leq s \text{ for some } s \in L\}$$

We say that L is *closed* if $L = \overline{L}$.

An *equivalence* relation $E \subseteq X \times X$ is a binary relation over a non-empty set X, such that is satisfies:

$$\begin{aligned} Reflexive &: (\forall x \in X) xEx \\ Symmetric &: (\forall x, x' \in X) xEx' \Rightarrow x'Ex \\ Transitive &: (\forall x, x', x'' \in X) xEx' \wedge x'Ex'' \Rightarrow xEx'' \end{aligned}$$

For $x \in X$, the *coset* of x with respect to equivalence relation E, denoted by $[x]$, is the set of all elements in X that are equivalent to x:

$$[x] := \{x' \in X | x'Ex\}$$

Two such cosets $[x]$, $[y]$ are either identical or disjoint.

Let $L \subseteq \Sigma^*$ be an arbitrary language, and let $s, t \in \Sigma^*$. The *Nerode* equivalence relation over Σ on L is defined as:

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

We write $\|L\|$ as the cardinality of the set of all cosets of the Nerode equivalence relation on L. We say a language L is *regular* if $\|L\| < \infty$. All languages in this report are regular unless otherwise stated.

Example 2.1.2 *Let* $\Sigma = \{\alpha, \beta\}, L = \{\epsilon, \alpha\}$. *Then we have cosets* $\{\epsilon\}, \{\alpha\}, \{\alpha\{\alpha, \beta\}^+, \beta\{\alpha, \beta\}^*\}$, *thus we have* $\|L\| = 3$. $\Diamond$

2.2 Automata

For a regular language L, since we have finite number of Nerode cosets, we can use a *finite state machine* to represent this language. A *recognizer* over Σ is a 5-tuple

$$R = (X, \Sigma, \zeta, x_o, X_m)$$

in which X is the state set, x_o is the initial state, $X_m \subseteq X$ is the set of marker states, and $\zeta : X \times \Sigma \to X$ is the transition function. The function ζ is extended to

$$\zeta : X \times \Sigma^* \to X$$

in the standard way by induction on string length.

The language L recognized by R is defined to be

$$L := \{s \in \Sigma^* | \zeta(x_o, s) \in X_m\}$$

For string $s \in (\Sigma^* - \overline{L})$, $\zeta(x_o, s)$ leads to a dump state.

In a recognizer, we use a small circle to represent a state and we assign a name for each state such $s0$, $s1$, and so on. The initial state has a thicker border and all marker states are indicated by a gray filled circle. The dump state is marked with a + sign. A transition is indicated by an arrow from its source state leading to its target state, labelled by an event in Σ. If the arrow is labelled by multiple events, then it represents a transition for each event.

Example 2.2.1 *For language L given in example 2.1.2, the recognizer is shown in Figure 2.1. In this recognizer, state $s0$ is the initial state and states $s0$ and $s1$ are marker states. State $s2$ is the dump state.* ◊

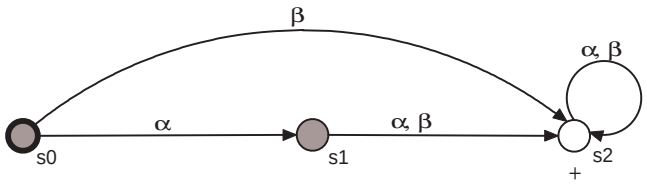

Figure 2.1: A Simple Recognizer

When every state in the recognizer corresponds to a unique Nerode equivalent class of L, we say such a recognizer is *canonical*.

Let $L \subseteq \Sigma^*$ be an arbitrary language. For a string $s \in L$, the $Elig_L()$ operator is defined to be the set of events that can immediately follow s in L:

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

We will represent DES using generators. A *generator* $\mathbf{G}$ is a 5-tuple

$$\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$$

where Σ is the event set, $Y \neq \emptyset$ is the state set, $y_o \in Y$ is the initial state, $Y_m \subseteq Y$ is set of marker states, and our transition function $\delta : Y \times \Sigma \to Y$ is a partial function. We use the notation $\delta(y, \sigma)!$ to mean that $\delta(y, \sigma)$ is defined. We extend δ to the partial function $\delta : Y \times \Sigma^* \to Y$ in the standard way.

For a generator $\mathbf{G}$, the language

$$L_m(\mathbf{G}) := \{s \in \Sigma^* \mid \delta(y_o, s)! \wedge \delta(y_o, s) \in Y_m\}$$

is called the *marked behavior* or *marked language* of $\mathbf{G}$.

The language

$$L(\mathbf{G}) := \{s \in \Sigma^* \mid \delta(y_o, s)!\}$$

is called the *closed behavior* of $\mathbf{G}$. $L(\mathbf{G})$ contains all strings that $\mathbf{G}$ can generate.

A state is *reachable* if there is a path from y_o to it. We say the state is *coreachable* if there is a path from it to any marked state. We say a generator $\mathbf{G}$ is *reachable* if all of its states are reachable. We say $\mathbf{G}$ is *coreachable* if all of its states are coreachable. We say $\mathbf{G}$ is *nonblocking* if every reachable state of $\mathbf{G}$ is coreachable. For a nonblocking generator, we have

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

If a generator is both reachable and coreachable, we say it's *trim*. A trim generator is always nonblocking, but the reverse doesn't always hold.

Let $\Sigma_1 \subseteq \Sigma$. We define a *natural projection* $P : \Sigma^* \to \Sigma_1^*$ according to

$$\begin{aligned} P(\epsilon) &= \epsilon \\ P(\sigma) &= \epsilon \qquad \text{if } \sigma \notin \Sigma_1 \\ P(\sigma) &= \sigma \qquad \text{if } \sigma \in \Sigma_1 \\ P(s\sigma) &= P(s)P(\sigma) \qquad s \in \Sigma^*, \sigma \in \Sigma \end{aligned}$$

For any string $s \in \Sigma^*$, $P(s)$ filters out all events that belong to $\Sigma - \Sigma_1$. The inverse image function of P is $P^{-1} : \mathrm{Pwr}(\Sigma_1^*) \to \mathrm{Pwr}(\Sigma^*)$. For $L \subseteq \Sigma_1^*$, we get

$$P^{-1}(L) := \{s \in \Sigma^* | \, P(s) \in L\}.$$

Let $P_i : \Sigma^* \to \Sigma_i^*, i = 1, 2$, be natural projections. Let $L_1 \subseteq \Sigma_1^*, L_2 \subseteq \Sigma_2^*$. We define the *synchronous product* of L_1, L_2, denoted $L_1 \| L_2$, according to

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

thus $s \in L_1 \| L_2$ iff $P_1(s) \in L_1 \wedge P_2(s) \in L_2$.

Example 2.2.2 *Let* $\Sigma = \{\alpha, \beta, \gamma\}$, $\Sigma_1 = \{\alpha, \beta\}$, $\Sigma_2 = \{\alpha, \gamma\}$, $P_i : \Sigma^* \to \Sigma_i^*, i = 1, 2$, *and* $L_1 = \{\alpha\beta\}$, $L_2 = \{\alpha\gamma\}$. *Then we have* $P_1^{-1}(L_1) = \{\gamma^*\alpha\gamma^*\beta\gamma^*\}, P_2^{-1}(L_2) = \{\beta^*\alpha\beta^*\gamma\beta^*\}$, *and we get* $L_1 \| L_2 = \{\alpha\beta\gamma, \alpha\gamma\beta\}$. $\Diamond$

When we synchronize two generators, we get a new generator which has the synchronized languages as its languages. For generators $\mathbf{G}_1 = (Y_1, \Sigma_1, \delta_1, y_{o,1}, Y_{m,1})$ and $\mathbf{G}_2 = (Y_2, \Sigma_2, \delta_2, y_{o,2}, Y_{m,2})$, the *synchronous product* of the two DES, denoted $\mathbf{G} = \mathbf{G}_1 \| \mathbf{G}_2$, is the reachable DES with the properties

$$\begin{aligned} L(\mathbf{G}) &= L(\mathbf{G}_1) \| L(\mathbf{G}_2) \\ L_m(\mathbf{G}) &= L_m(\mathbf{G}_1) \| L_m(\mathbf{G}_2) \end{aligned}$$

and with event set $\Sigma = \Sigma_1 \cup \Sigma_2$.

When two generators has the same event set, the synchronous product operation turns into the *meet* (intersection) operation. The TCT procedure **meet** implements the meet operation on two generators. Let $\mathbf{G} = \mathbf{meet}(\mathbf{G}_1, \mathbf{G}_2)$, we then get a reachable DES with the properties

$$\begin{aligned} L(\mathbf{G}) &= L(\mathbf{G}_1) \cap L(\mathbf{G}_2) \\ L_m(\mathbf{G}) &= L_m(\mathbf{G}_1) \cap L_m(\mathbf{G}_2) \end{aligned}$$

For two arbitrary languages L_1, L_2, we say they are *nonconflicting* if

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

Example 2.2.3 *Let* $\Sigma = \{\alpha, \beta, \gamma\}, L_1 = \{\alpha\beta\}, L_2 = \{\alpha\gamma\}$. *We have*

$$\begin{aligned} \overline{L_1 \cap L_2} &= \emptyset \\ \neq \overline{L_1} \cap \overline{L_2} &= \{\epsilon, \alpha, \alpha\beta\} \cap \{\epsilon, \alpha, \alpha\gamma\} = \{\epsilon, \alpha\} \end{aligned}$$

thus L_1 *and* L_2 *are conflicting.* $\Diamond$

2.3 Supervisory Control

In supervisory control theory, control action is achieved by disabling certain events to prevent some unwanted events from happening. Events in an alphabet Σ are divided into two categories,

$$\Sigma = \Sigma_c \dot{\cup} \Sigma_u$$

where the disjoint subsets Σ_c and Σ_u comprise respectively the *controllable events* and the *uncontrollable events*. A controllable event works like a switch which can be either turned on or off. An uncontrollable event can not be blocked from happening, i.e., if an uncontrollable event is defined at a certain state, there is no chance for us to stop it from happening if the current work flow reaches that state.

A *control pattern* is a subset of Σ that contains all uncontrollable events. We introduce the set of all control patterns:

$$\Gamma = \{\gamma | \gamma \in Pwr(\Sigma) \wedge \Sigma_u \subseteq \gamma\}.$$

A *supervisory control* for $\mathbf{G}$ is any map

$$V : L(\mathbf{G}) \rightarrow \Gamma$$

The pair $(\mathbf{G}, V)$ will be written as V/G, to suggest '$\mathbf{G}$ under the supervision of V'. The closed behavior of $V/\mathbf{G}$ is defined to be the language $L(V/\mathbf{G}) \subseteq L(\mathbf{G})$ described as follows:

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

The marked behavior of $V/\mathbf{G}$ is

$$L_m(V/\mathbf{G}) = L(V/\mathbf{G}) \cap L_m(\mathbf{G})$$

With respect to $\mathbf{G}$, we say that V is nonblocking if

$$\overline{L_m(V/\mathbf{G})} = L(V/\mathbf{G})$$

With $\mathbf{G}$ understood, we use the abbreviation NSC to refer to a nonblocking supervisory control.

A useful generalization of a NSC is to allow the supervisory control to also include marking. Let $M \subseteq L_m(\mathbf{G})$. We define a *marking nonblocking supervisory control* (MNSC) for the pair $(M, \mathbf{G})$ as a map $V : L(\mathbf{G}) \rightarrow \Gamma$ as before, with the difference that we now define the marked behavior of $V/\mathbf{G}$ as

$$L_m(V/\mathbf{G}) = L(V/\mathbf{G}) \cap M.$$

A language $K \subseteq \Sigma^*$ is said to be *controllable(with respect to* $\mathbf{G}$*)* if

$$(\forall s,t)s \in \overline{K} \ \wedge \ t \in \Sigma_u \ \wedge \ st \in L(\mathbf{G}) \Rightarrow st \in \overline{K}$$

For a more concise statement, we use the following notation. For $S \subseteq \Sigma^*$ and $\Sigma_o \subseteq \Sigma$, let $S\Sigma_o$ denote the set of strings of form $s\sigma$ with $s \in S$ and $\sigma \in \Sigma_o$. Then K is controllable iff

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

Let $\mathbf{G}$ be a plant DES defined over alphabet Σ_G, and $\mathbf{S}$ be a supervisor DES defined over alphabet Σ_S. We will use the following notation:

$$\begin{aligned}
\Sigma &:= \Sigma_G \ \cup \ \Sigma_S \\
P_G &: \Sigma^* \to \Sigma_G^* \\
P_S &: \Sigma^* \to \Sigma_S^* \\
L_G &:= P_G^{-1}(L(\mathbf{G})) \\
L_S &:= P_S^{-1}(L(\mathbf{S}))
\end{aligned}$$

We say that a supervisor $\mathbf{S}$ is controllable for $\mathbf{G}$ if

$$(\forall s \in L_G \cap L_S)\text{Elig}_{L_G}(s) \cap \Sigma_u \subseteq \text{Elig}_{L_S}(s) \tag{2.1}$$

We define the closed loop behavior, denoted CL, of our system to be

$$\text{CL} := \mathbf{S}||\mathbf{G}$$

This is essentially the expected behavior of the plant under the control of our supervisor.

It is clear that the empty set $\emptyset$, $L(\mathbf{G})$ and Σ^* are always controllable with respect to $\mathbf{G}$.

Let $K \subseteq L \subseteq \Sigma^*$. We say the language K is *L-Closed* if $K = \overline{K} \cap L$. The following two theorems are from [79].

Theorem 1 *Let $K \subseteq L_m(\mathbf{G}), K \neq \emptyset$. There exists a nonblocking supervisory control V for $\mathbf{G}$ such that $L_m(V/\mathbf{G}) = K$ iff*

1. *K is controllable with respect to $\mathbf{G}$, and*
2. *K is $L_m(\mathbf{G})$-closed.* $\Diamond$

We now give the counterpart to *Theorem 1* for MNSC.

Theorem 2 *Let $K \subseteq L_m(\mathbf{G}), K \neq \emptyset$. There exists a marking nonblocking supervisory control V for $(K, \mathbf{G})$ such that*

$$L_m(V/\mathbf{G}) = K$$

iff K is controllable with respect to $\mathbf{G}$. $\Diamond$

In this thesis, we will only be dealing with marking nonblocking supervisory controls.

In order to implement the supervision in the supervisory control framework, one way is to design the supervisor directly. In this approach, the designer needs to make sure the supervisor is controllable with respect to the plant and satisfies the control specifications. The other way to do this is based on constructing specification DES which are generally not controllable with respect to the plant, and use software to automatically compute a controllable supervisor from the given DES specifications and the plant model. Usually the specification DES are much easier to design than supervisors, since the designer only needs to focus on specifying what they want the system to do, instead of the details of how to make the system do what they want.

Let $\mathbf{G}$ be a plant DES with $\Sigma = \Sigma_c \cup \Sigma_u$ and $E \subseteq \Sigma^*$ be a specification language over Σ. We introduce the set of all sublanguages of E that are controllable with respect to $\mathbf{G}$:

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

It is shown in [79] that the the supremal element, denoted $\sup\mathcal{C}(E)$, always exists and equals :

$$\sup\mathcal{C}(E) = \bigcup_{K \in \mathcal{C}(E)} K$$

TCT provides a procedure **supcon** to calculate the supremal controllable element of a specification DES for given plant DES. It starts from the meet of the two DES and trims off non-controllable and blocking states until the result is controllable for the plant and nonblocking. Since we are dealing with regular languages and finite state machines only, this process is guaranteed to stop after finite number of iterations.

Chapter 3

HISC Introduction

For a detailed introduction to HISC, we refer you to [34]. In this chapter, we will introduce to you the primary definitions and notation, but will not go into depth about their development and interpretation. In this thesis, we use use the HISC definitions from [33], which are a slightly modified set of definitions from those of [34]. In Chapter 4, we explain our rationale for this, and we present proofs that these definitions are equivalent to the originals.

3.1 Basic Setting

In HISC there is a master-slave relationship.[1] A *high level subsystem* sends a command to a particular *low level subsystem,* which then performs the indicated task and returns an answer. Figure 3.1 shows conceptually the structure and information flow of the system in the special case when there is only a single low level system. Communication between the high level system and the low level system occurs in a serial fashion. A request from the high level is followed by an answer from the low level before the next request is issued to the low level

[1]Most of this chapter originally appeared as part of [40] and is reused with permission. Definition 3.4.2 and Lemma 1 are new.

subsystem. This style of interaction is enforced by an interface that mediates communication between the two subsystems. All system components, including the interface, are modeled as automata as shown in Fig. 3.2 where our *flat system* would be $\mathbf{G} := \mathbf{G}_H||\mathbf{G}_I||\mathbf{G}_L$. By flat system we mean the equivalent DES if we ignored the interface structure.

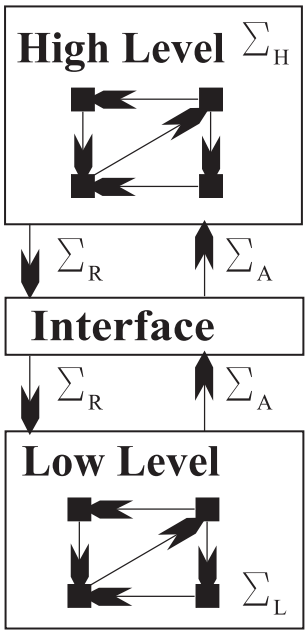

Figure 3.1: Interface Block Diagram.

In order to restrict information flow and decouple the subsystems, the event set Σ is split into four disjoint alphabets: Σ_H, Σ_L, Σ_R, and Σ_A. The events in Σ_H are *high level events* and the events in Σ_L *low level events* as these events appear only in the high level and low level models, $\mathbf{G}_H$ and $\mathbf{G}_L$ respectively. We then have $\mathbf{G}_H$ defined over $\Sigma_H \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A$ and $\mathbf{G}_L$ defined over $\Sigma_L \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A$.

3.2 Interfaces

As the *interface automaton* $\mathbf{G}_I$ is only concerned with communication between the two subsystems, it is defined over the events that are common to both levels of

the hierarchy, $\Sigma_R \dot{\cup} \Sigma_A$, which are collectively known as the set of *interface events*, denoted Σ_I. The events in Σ_R, called *request events*, represent commands sent from the high level subsystem to the low level subsystem. The events in Σ_A are *answer events* and represent the low level subsystem's responses to the request events. In order to enforce the serialization of requests and answers, we restrict the interface to the subclass of command-pair interfaces defined below.

Definition 3.2.1 *A DES* $\mathbf{G}_I = (X, \Sigma_R \dot{\cup} \Sigma_A, \xi, x_o, X_m)$ *is a* command-pair interface *if:*

(A) $L(\mathbf{G}_I) \subseteq \overline{(\Sigma_R.\Sigma_A)^*}$, *and*

(B) $L_m(\mathbf{G}_I) = (\Sigma_R.\Sigma_A)^* \cap L(\mathbf{G}_I)$

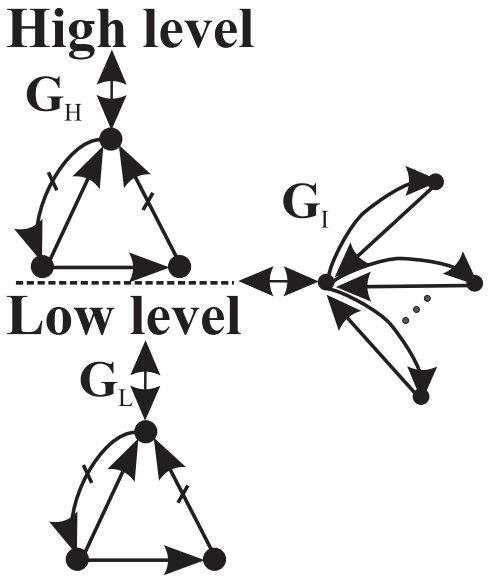

Figure 3.2: Two Tiered Structure of the System.

Condition (A) says that request events and answer events must alternate (i.e. serialization of requests) while condition (B) states that every answered request results in a marked state. An example command pair interface with $\Sigma_R := \{\rho_i | i = 1, 2, 3\}$ and $\Sigma_A := \{\alpha_i | i = 1, \ldots, 7\}$ is shown in Fig. 3.3.

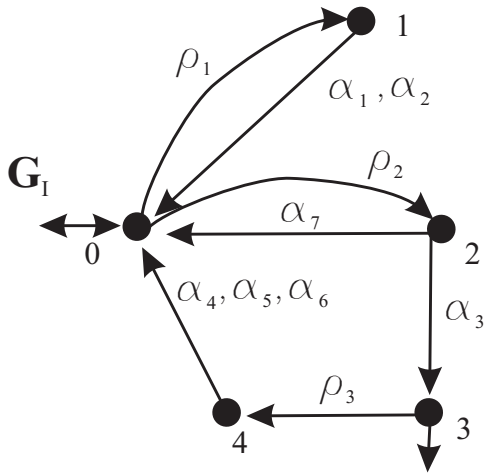

Figure 3.3: Example Interface.

3.3 Basic Notation

We now generalize the above "serial case" where there is a single low level system, to the *parallel* case where there are n low level subsystems. In this case we say that the system is an n^{th} *degree* parallel system. Figure 3.4 shows conceptually the structure and flow of information. The single high level subsystem, interacts with $n \geq 1$ independent low level subsystems, communicating with each low level subsystem in parallel through a separate interface.

As in the serial case, to restrict the flow of information at the interface, we partition the system alphabet into pairwise disjoint alphabets:

$$\Sigma := \Sigma_H \dot{\cup} \dot{\bigcup_{j=1,\ldots,n}} [\Sigma_{L_j} \dot{\cup} \Sigma_{R_j} \dot{\cup} \Sigma_{A_j}] \tag{3.1}$$

The high level subsystem is modeled by DES $\mathbf{G}_H$ (defined over event set $\Sigma_H \dot{\cup} (\dot{\cup}_{j \in \{1,\ldots,n\}} [\Sigma_{R_j} \dot{\cup} \Sigma_{A_j}])$). For $j \in \{1, \ldots, n\}$, the j^{th} low level subsystem is modeled by DES $\mathbf{G}_{L_j}$ (defined over event set $\Sigma_{L_j} \dot{\cup} \Sigma_{R_j} \dot{\cup} \Sigma_{A_j}$), and the j^{th} interface by DES $\mathbf{G}_{I_j}$ (defined over event set $\Sigma_{R_j} \dot{\cup} \Sigma_{A_j}$). The overall system has the structure shown in Fig. 3.5. Thus our flat system is $\mathbf{G} = \mathbf{G}_H || \mathbf{G}_{I_1} || \mathbf{G}_{L_1} || \ldots || \mathbf{G}_{I_n} || \mathbf{G}_{L_n}$.

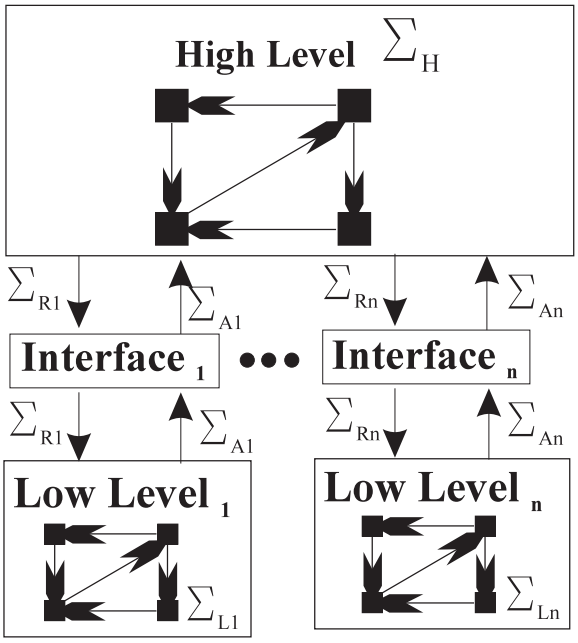

Figure 3.4: Parallel Interface Block Diagram.

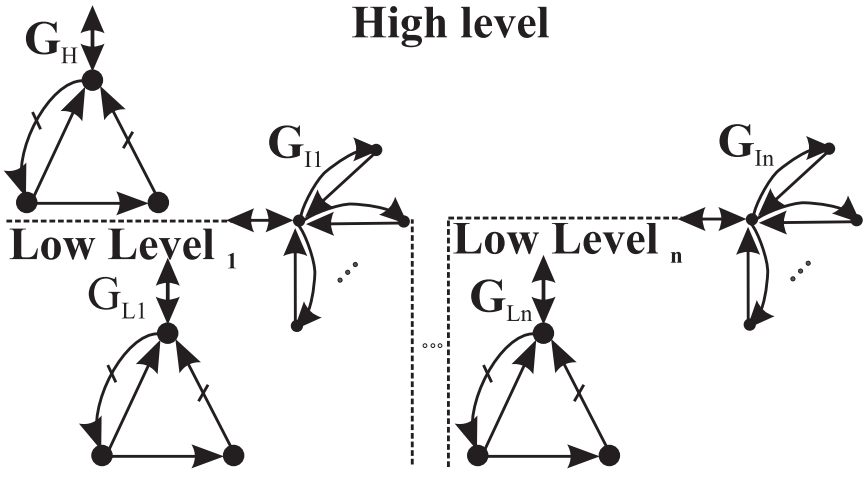

Figure 3.5: Two Tiered Structure of Parallel System

To simplify notation in our exposition, we bring in the following event sets, natural projections, and languages. For the remainder of this section, the index j has range $\{1, \ldots, n\}$.

$$\begin{aligned}
\Sigma_{I_j} &:= \Sigma_{R_j} \cup \Sigma_{A_j}, & P_{I_j} &: \Sigma^* \to \Sigma_{I_j}^* \\
\Sigma_{IL_j} &:= \Sigma_{L_j} \cup \Sigma_{I_j}, & P_{IL_j} &: \Sigma^* \to \Sigma_{IL_j}^* \\
\Sigma_{IH} &:= \Sigma_H \cup \bigcup_{k \in \{1,\ldots,n\}} \Sigma_{I_k} & P_{IH} &: \Sigma^* \to \Sigma_{IH}^* \\
\mathcal{H} &:= P_{IH}^{-1}(L(\mathbf{G}_H)), & \mathcal{H}_m &:= P_{IH}^{-1}(L_m(\mathbf{G}_H)) \subseteq \Sigma^* \\
\mathcal{L}_j &:= P_{IL_j}^{-1}(L(\mathbf{G}_{L_j})), & \mathcal{L}_{m_j} &:= P_{IL_j}^{-1}(L_m(\mathbf{G}_{L_j})) \subseteq \Sigma^* \\
\mathcal{I}_j &:= P_{I_j}^{-1}(L(\mathbf{G}_{I_j})), & \mathcal{I}_{m_j} &:= P_{I_j}^{-1}(L_m(\mathbf{G}_{I_j})) \subseteq \Sigma^*
\end{aligned}$$

3.4 Interface Consistency Definition

We now present the properties that the system must satisfy to ensure that it interacts with the interfaces correctly.

Definition 3.4.1 *The n^{th} degree ($n \geq 1$) parallel interface system composed of DES $\mathbf{G}_H, \mathbf{G}_{I_1}, \mathbf{G}_{L_1}, \ldots, \mathbf{G}_{I_n}, \mathbf{G}_{L_n}$, 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 Property

3.

$$(\forall s \in \mathcal{H} \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_k)\ Elig_{\mathcal{I}_j}(s)\ \cap\ \Sigma_{A_j} \subseteq Elig_{\mathcal{H} \cap \bigcap_{k \neq j} \mathcal{I}_k}(s)$$

Low Level Properties

4. $(\forall s \in \mathcal{L}_j \cap \mathcal{I}_j)\ Elig_{\mathcal{I}_j}(s)\ \cap\ \Sigma_{R_j} \subseteq Elig_{\mathcal{L}_j}(s)$

5. $(\forall s \in \Sigma^*.\Sigma_{R_j} \cap \mathcal{L}_j \cap \mathcal{I}_j)$

$$Elig_{\mathcal{L}_j \cap \mathcal{I}_j}(s\Sigma^*_{L_j})\ \cap\ \Sigma_{A_j} = Elig_{\mathcal{I}_j}(s)\ \cap\ \Sigma_{A_j} \quad where$$

$$Elig_{\mathcal{L}_j \cap \mathcal{I}_j}(s\Sigma^*_{L_j}) := \bigcup_{l \in \Sigma^*_{L_j}} Elig_{\mathcal{L}_j \cap \mathcal{I}_j}(sl)$$

6. $(\forall s \in \mathcal{L}_j \cap \mathcal{I}_j)$

$$s \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma^*_{L_j})\ sl \in \mathcal{L}_{m_j} \cap \mathcal{I}_{m_j}.$$

The first two properties assert that the system has the required basic architecture, with the high level and low level subsystems only sharing request and answer events and the interaction between the levels mediated by interfaces. This provides a form of information hiding as it restricts the high level subsystem from knowing (and directly affecting) internal details of the low level subsystems and vice versa.

The High level property (3) asserts that when $\mathbf{G}_H$ is synchronized with all of the other subsystem interfaces $\mathbf{G}_{I_k}, k \neq j$, it must always accept an answer event if the event is eligible in the interface $\mathbf{G}_{I_j}$. In other words, the high level subsystem is forbidden to assume more about when an answer event can occur than what is provided by the interface. Similarly, low level property (4) asserts that the low level subsystem ($\mathbf{G}_{L_j}$) must always accept a request event if the event is eligible in its interface $\mathbf{G}_{I_j}$. We note that both (3) and (4) can be computed using the standard algorithms for controllability.

Condition (5) states that immediately after a request event (some $\rho \in \Sigma_{R_j}$) has occurred, and before it is followed by any low level events in Σ_{L_j}, there exist one or more paths via strings in $\Sigma^*_{L_j}$ to each answer event that $\mathbf{G}_{I_j}$ says can follow the

request event. Finally, (6) asserts that every string marked by the interface $\mathbf{G}_{I_j}$ and accepted by the low level subsystem, can be extended by a low level string to a string marked by $\mathbf{G}_{L_j}$.

We now give an equivalent definition for interface consistency that will be useful later when we examine synthesis in the HISC setting. This new definition is only used to make the definitions for synthesis easier and clearer. The only thing that has changed is that point 5 is given in a new formulation. The new point 5 is not exactly equal to the old one, but when used in conjunction with point 4, it is equivalent to the original point 5.

Definition 3.4.2 *The n^{th} degree ($n \geq 1$) parallel interface system composed of DES $\mathbf{G}_H, \mathbf{G}_{I_1}, \mathbf{G}_{L_1}, \ldots, \mathbf{G}_{I_n}, \mathbf{G}_{L_n}$, 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 Property

3. $$(\forall s \in \mathcal{H} \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_k)\ Elig_{\mathcal{I}_j}(s)\ \cap\ \Sigma_{A_j} \subseteq Elig_{\mathcal{H} \cap \bigcap_{k \neq j} \mathcal{I}_k}(s)$$

Low Level Properties

4. $(\forall s \in \mathcal{L}_j \cap \mathcal{I}_j)\ Elig_{\mathcal{I}_j}(s)\ \cap\ \Sigma_{R_j} \subseteq Elig_{\mathcal{L}_j}(s)$
5. $(\forall s \in \mathcal{L}_j \cap \mathcal{I}_j)(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})$
   $s\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s\rho l\alpha \in \mathcal{L}_j \cap \mathcal{I}_j$

6. $(\forall s \in \mathcal{L}_j \cap \mathcal{I}_j)$
$s \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma_{L_j}^*)\ sl \in \mathcal{L}_{m_j} \cap \mathcal{I}_{m_j}$.

We now prove that Definition 3.4.1 and Definition 3.4.2 are equivalent.

Lemma 1 *The* n^{th} *degree* ($n \geq 1$) *parallel interface system composed of DES* $\mathbf{G}_H, \mathbf{G}_{I_1}, \mathbf{G}_{L_1}, \ldots, \mathbf{G}_{I_n}, \mathbf{G}_{L_n}$, *satisfies Definition 3.4.1 with respect to the alphabet partition given by (3.1), if and only if the system satisfies Definition 3.4.2 with respect to the alphabet partition given by (3.1).*

Proof We will show that Definition 3.4.1 $\Leftrightarrow$ Definition 3.4.2.

As the only difference between Definition 3.4.1 and Definition 3.4.2 is their respective point 5, it is sufficient to show that each definition implies that point 5 of the other definition is satisfied.

Let $j \in \{1, \ldots, n\}$.

(I) Show Definition 3.4.1 $\Rightarrow$ Definition 3.4.2.

We assume the parallel system satisfies Definition 3.4.1.

We will now show this implies the system satisfies point 5 of Definition 3.4.2.

To show that point 5 of Definition 3.4.2 is satisfied, we need to show:
$(\forall s \in \mathcal{L}_j \cap \mathcal{I}_j)(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\ \ s\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s\rho l\alpha \in \mathcal{L}_j \cap \mathcal{I}_j$

Let $s \in \mathcal{L}_j \cap \mathcal{I}_j$, $\rho \in \Sigma_{R_j}$, and $\alpha \in \Sigma_{A_j}$ **(1)**

Assume $s\rho\alpha \in \mathcal{I}_j$ **(2)**

We will now show this implies: $(\exists l \in \Sigma_{L_j}^*)\, s\rho l\alpha \in \mathcal{L}_j \cap \mathcal{I}_j$

As we wish to apply point 5 of Definition 3.4.1, we need to show that $s\rho \in \mathcal{L}_j$.

As we have $s\rho\alpha \in \mathcal{I}_j$ by **(2)**, we can conclude that $s\rho \in \mathcal{I}_j$ as $\mathcal{I}_j$ is closed.

$\Rightarrow \rho \in \text{Elig}_{\mathcal{I}_j}(s)$

$\Rightarrow s\rho \in \mathcal{L}_j$ by **(1)** and point 4 of Definition 3.4.1.

By point 5 of Definition 3.4.1, we can now conclude:

$\text{Elig}_{\mathcal{L}_j \cap \mathcal{I}_j}(s\rho\Sigma^*_{L_j}) \cap \Sigma_{A_j} = \text{Elig}_{\mathcal{I}_j}(s\rho) \cap \Sigma_{A_j}$

$\Rightarrow \alpha \in \text{Elig}_{\mathcal{L}_j \cap \mathcal{I}_j}(s\rho\Sigma^*_{L_j})$, by **(2)**.

$\Rightarrow (\exists l \in \Sigma^*_{L_j}) s\rho l\alpha \in \mathcal{L}_j \cap \mathcal{I}_j$, as required.

Part I complete.

(II) Show Definition 3.4.2 $\Rightarrow$ Definition 3.4.1.

We assume the parallel system satisfies Definition 3.4.2.

We will now show this implies the system satisfies point 5 of Definition 3.4.1.

To show that point 5 of Definition 3.4.1 is satisfied, we need to show:

$(\forall s \in \Sigma^*.\Sigma_{R_j} \cap \mathcal{L}_j \cap \mathcal{I}_j)\ \text{Elig}_{\mathcal{L}_j \cap \mathcal{I}_j}(s\Sigma^*_{L_j}) \cap \Sigma_{A_j} = \text{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{A_j}$

Let $s \in \Sigma^*.\Sigma_{R_j} \cap \mathcal{L}_j \cap \mathcal{I}_j$. **(3)**

We will now show that: $\text{Elig}_{\mathcal{L}_j \cap \mathcal{I}_j}(s\Sigma^*_{L_j}) \cap \Sigma_{A_j} = \text{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{A_j}$

As $\text{Elig}_{\mathcal{L}_j \cap \mathcal{I}_j}(s\Sigma^*_{L_j}) := \bigcup_{l \in \Sigma^*_{L_j}} \text{Elig}_{\mathcal{L}_j \cap \mathcal{I}_j}(sl)$, it is sufficient to show:

$$\bigcup_{l \in \Sigma^*_{L_j}} \text{Elig}_{\mathcal{L}_j \cap \mathcal{I}_j}(sl) \cap \Sigma_{A_j} = \text{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{A_j}$$

We will show: $\subseteq$ and $\supseteq$.

(II.a) We first show: $\bigcup_{l \in \Sigma^*_{L_j}} \text{Elig}_{\mathcal{L}_j \cap \mathcal{I}_j}(sl) \cap \Sigma_{A_j} \subseteq \text{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{A_j}$

Let $\alpha \in \bigcup_{l \in \Sigma^*_{L_j}} \text{Elig}_{\mathcal{L}_j \cap \mathcal{I}_j}(sl) \cap \Sigma_{A_j}$ **(4)**

$\Rightarrow (\exists l \in \Sigma^*_{L_j})\, sl\alpha \in \mathcal{L}_j \cap \mathcal{I}_j$

$\Rightarrow sl\alpha \in \mathcal{I}_j$

As $P_{I_j}(sl\alpha) = P_{I_j}(s\alpha)$ by definition of P_{I_j}, we can apply *Proposition 20 Point (e)* of [34] and conclude:

$s\alpha \in \mathcal{I}_j$

$\Rightarrow \alpha \in \text{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{A_j}$ by **(4)**, as required.

(II.b) Show: $\text{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{A_j} \subseteq \bigcup_{l \in \Sigma^*_{L_j}} \text{Elig}_{\mathcal{L}_j \cap \mathcal{I}_j}(sl) \cap \Sigma_{A_j}$

Let $\alpha \in \text{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{A_j}$ **(5)**

$\Rightarrow s\alpha \in \mathcal{I}_j$ **(6)**

We will now show this implies: $\alpha \in \bigcup_{l \in \Sigma^*_{L_j}} \text{Elig}_{\mathcal{L}_j \cap \mathcal{I}_j}(sl) \cap \Sigma_{A_j}$

We first note that $s \in \Sigma^*.\Sigma_{R_j} \cap \mathcal{L}_j \cap \mathcal{I}_j$ by **(3)** implies:

$(\exists s' \in \Sigma^*)(\rho \in \Sigma_{R_j})\ s'\rho = s$ **(7)**

$\Rightarrow s'\rho\alpha \in \mathcal{I}_j$ by **(6)**, and $s' \in \mathcal{L}_j \cap \mathcal{I}_j$ as $\mathcal{I}_j$ is closed.

We can now conclude by **(5)** and point 5 of Definition 3.4.2, that:

$(\exists l \in \Sigma^*_{L_j})\, s'\rho l\alpha \in \mathcal{L}_j \cap \mathcal{I}_j$

$\Rightarrow (\exists l \in \Sigma^*_{L_j})\, sl\alpha \in \mathcal{L}_j \cap \mathcal{I}_j$, by **(7)**.

$\Rightarrow \alpha \in \bigcup_{l \in \Sigma^*_{L_j}} \text{Elig}_{\mathcal{L}_j \cap \mathcal{I}_j}(sl) \cap \Sigma_{A_j}$, as required.

By **Part II.a** and **II.b**, we can conclude:

$\bigcup_{l \in \Sigma^*_{L_j}} \text{Elig}_{\mathcal{L}_j \cap \mathcal{I}_j}(sl) \cap \Sigma_{A_j} = \text{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{A_j}$.

Part II complete.

By **Part I** and **Part II**, we can conclude:

Definition 3.4.1 $\Leftrightarrow$ Definition 3.4.2

□

3.5 Local Conditions for Global Nonblocking of the System

We now provide the conditions that the subsystems and their interface(s) must satisfy in addition to the interface consistency properties, if the system **G** is to be nonblocking.

Definition 3.5.1 *The n^{th} degree ($n \geq 1$) parallel interface system composed of DES $\mathbf{G}_H, \mathbf{G}_{I_1}, \mathbf{G}_{L_1}, \ldots, \mathbf{G}_{I_n}, \mathbf{G}_{L_n}$, is said to be* level-wise nonblocking *if the following conditions are satisfied:*

(I) nonblocking at the high level*:*

$$\overline{\mathcal{H}_m \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_{m_k}} = \mathcal{H} \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_k$$

(II) nonblocking at the low level*: for all $j \in \{1, \ldots, n\}$,*

$$\overline{\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j}} = \mathcal{L}_j \cap \mathcal{I}_j$$

The above definition can be paraphrased as saying that for each component subsystem synchronized with its interface(s), every reachable state must have a path to a state that is marked by both the subsystem and its interface(s).

3.6 Local Conditions for Global Controllability of the System

The representation of the system given in Fig. 3.5 simplifies notation when verifying nonblocking by ignoring the distinction between plants and supervisors. For controllability, we need to split the subsystems into their plant and supervisor components, as shown in Fig. 3.6 for the serial case. To do this, we define the *high level plant* to be $\mathbf{G}_H^p$, and the *high level supervisor* to be $\mathbf{S}_H$ (both defined over event set Σ_{IH}). Similarly, the j^{th} *low level plant* and *supervisor* are $\mathbf{G}_{L_j}^p$ and $\mathbf{S}_{L_j}$ (defined over Σ_{IL_j}). The high level subsystem and the j^{th} low level subsystem are then $\mathbf{G}_H := \mathbf{G}_H^p || \mathbf{S}_H$ and $\mathbf{G}_{L_j} := \mathbf{G}_{L_j}^p || \mathbf{S}_{L_j}$, respectively.

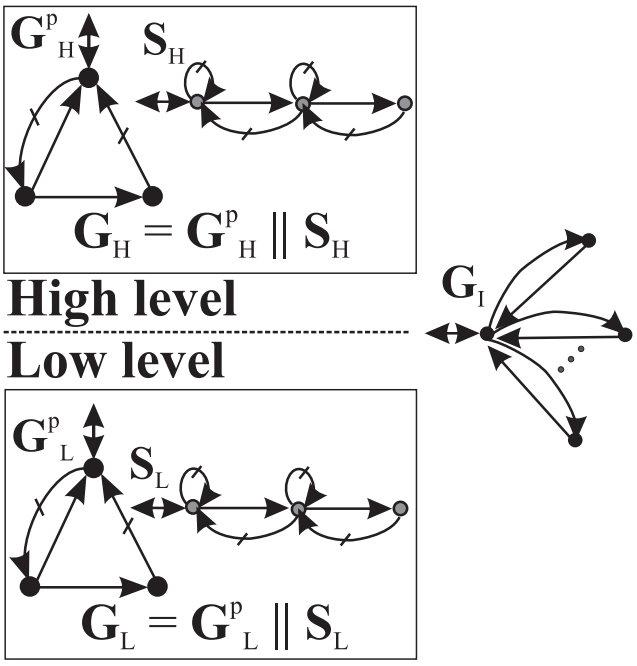

Figure 3.6: Plant and Supervisor Subplant Decomposition

We can now define our *flat supervisor* and *plant* as well as some useful languages

as follows:

$$\textbf{Plant} := \mathbf{G}^p_H||\mathbf{G}^p_{L_1}||\ldots||\mathbf{G}^p_{L_n} \qquad \textbf{Sup} := \mathbf{S}_H||\mathbf{S}_{L_1}||\ldots||\mathbf{S}_{L_n}||\mathbf{G}_{I_1}||\ldots||\mathbf{G}_{I_n}$$

$$\mathcal{H}^p := P^{-1}_{IH}(L(\mathbf{G}^p_H)), \qquad \mathcal{S}_H := P^{-1}_{IH}(L(\mathbf{S}_H)), \quad \subseteq \Sigma^*$$

$$\mathcal{L}^p_j := P^{-1}_{IL_j}(L(\mathbf{G}^p_{L_j})), \qquad \mathcal{S}_{L_j} := P^{-1}_{IL_j}(L(\mathbf{S}_{L_j})), \quad \subseteq \Sigma^*$$

For the controllability requirements at each level, we adopt the standard partition $\Sigma = \Sigma_u \dot{\cup} \Sigma_c$, splitting our alphabet into *uncontrollable* and *controllable events.* Note that this partition may, in general, be independent of the partition (3.1).

Definition 3.6.1 *The n^{th} degree ($n \geq 1$) parallel interface system composed of DES $\mathbf{G}^p_H$, $\mathbf{G}^p_{L_1}, \ldots, \mathbf{G}^p_{L_n}$, $\mathbf{S}_H$, $\mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$, is* level-wise controllable *with respect to the alphabet partition given by (3.1), if for all $j \in \{1, \ldots, n\}$ the following conditions hold:*

(I) The alphabet of $\mathbf{G}^p_H$ and $\mathbf{S}_H$ is Σ_{IH}, the alphabet of $\mathbf{G}^p_{L_j}$ and $\mathbf{S}_{L_j}$ is Σ_{IL_j}, and the alphabet of $\mathbf{G}_{I_j}$ is Σ_{I_j}

(II) $(\forall s \in \mathcal{L}^p_j \cap \mathcal{S}_{L_j} \cap \mathcal{I}_j) \quad Elig_{\mathcal{L}^p_j}(s) \cap \Sigma_u \subseteq Elig_{\mathcal{S}_{L_j} \cap \mathcal{I}_j}(s)$

(III) $(\forall s \in \mathcal{H}^p \cap [\cap_{k\in\{1,\ldots,n\}}\mathcal{I}_k] \cap \mathcal{S}_H) \qquad Elig_{\mathcal{H}^p \cap [\cap_{k\in\{1,\ldots,n\}}\mathcal{I}_k]}(s) \cap \Sigma_u \subseteq Elig_{\mathcal{S}_H}(s)$

The above definition states that the system is level-wise controllable if, for the given distributed supervisor, the high level supervisor is controllable for the high level plant combined with all of the interfaces (by III) and that each low level supervisor synchronized with the subsystem's interface is controllable for the subsystem's low level plant (by II).

Chapter 4

Equivalence of HISC Definitions

In the original Hierarchical Interface-based Supervisory Control definitions presented in [34], systems are divided into serial systems (single low level) and parallel systems ($n \geq 1$ low levels), where a serial system is a special case of a parallel system with degree $n = 1$. For each type of system, a set of interface conditions were developed, with the parallel system definitions building upon the serial system definitions.

In this thesis,[1] we use a slightly modified set of definitions, which no longer treats serial systems (referred to as the "serial case") and parallel systems (referred to as the "parallel case") differently. We present a single set of definitions (*interface consistency* from Section 3.4, *level-wise nonblocking* from Section 3.5, and *level-wise controllable* from Section 3.6) for the parallel case that is expressed directly in terms of the components of a parallel system. The new definitions introduced in this work are more concise and clear as a result. The new definitions also make it clear exactly what checks need to be performed, and on which component. In this chapter, we will show that these new definitions are equivalent to the original ones from [34].

[1] This chapter also appeared as part of [40], but is originally from this thesis.

In the following sections we will restate the original definitions for ease of reference, adding "(ORIG)" to the titles that are the same as our new definitions to prevent confusion. We will first present the interface consistency definitions, then nonblocking, and finally controllability. We will then show that each is equivalent to the corresponding new definition.

4.1 Useful Propositions

We start by introducing a few useful propositions for later proofs. However, we first need to introduce the following definition:

Definition 4.1.1 *For natural projection $P_o : \Sigma^* \to \Sigma_o^*$ for some $\Sigma_o \subseteq \Sigma$, we say that language $L \subseteq \Sigma^*$ is* P_o-invariant *if $P_o^{-1}(P_o(L)) = L$.*

In short, events in $\Sigma - \Sigma_o$ (Σ with the events of Σ_o removed) have no affect on membership in L. If L was the language generated by some DES $\mathbf{G}$, then these events would be selflooped at every state in $\mathbf{G}$.

Our first proposition is for a n^{th} degree ($n \geq 1$) parallel interface system composed of DES $\mathbf{G}_H, \mathbf{G}_{I_1}, \mathbf{G}_{L_1}, \ldots, \mathbf{G}_{I_n}, \mathbf{G}_{L_n}$ as defined in Section 3.3. We first need to define the natural projection, $P_j : \Sigma^* \to \Sigma'(j)^*$, where Σ is given by (3.1) in Section 3.3, $j \in \{1, \ldots, n\}$, and $\Sigma'(j) = \Sigma - \dot{\cup}_{k \in \{1, \ldots, (j-1), (j+1), \ldots, n\}} \Sigma_{L_k}$. The proposition below essentially states that the indicated languages are P_j-invariant, a property that will be useful in the following proofs. The result follows from the fact that the languages are inverse projections of closed and marked languages of DES whose alphabets are subsets of $\Sigma'(j)$.

Proposition 1 *With $\mathcal{H}$, $\mathcal{H}_m$, $\mathcal{I}_k$, $\mathcal{I}_{m_k} (k = \{1 \ldots n\})$, $\mathcal{L}_j$, and $\mathcal{L}_{m_j}$ as defined in Section 3.3, we have:*

(a) $\quad P_j^{-1}(P_j(\mathcal{H})) = \mathcal{H}$

(*b*) $P_j^{-1}(P_j(\mathcal{H}_m)) = \mathcal{H}_m$

(*c*) $P_j^{-1}(P_j(\mathcal{I}_k)) = \mathcal{I}_k, k = \{1 \ldots n\}$

(*d*) $P_j^{-1}(P_j(\mathcal{I}_{m_k})) = \mathcal{I}_{m_k}, k = \{1 \ldots n\}$

(*e*) $P_j^{-1}(P_j(\mathcal{L}_j)) = \mathcal{L}_j$

(*f*) $P_j^{-1}(P_j(\mathcal{L}_{m_j})) = \mathcal{L}_{m_j}$

Proof

Point (a): Show $P_j^{-1}(P_j(\mathcal{H})) = \mathcal{H}$

By definition, $\mathcal{H} = P_{IH}^{-1}(L(G_H))$. It is thus sufficient to show:

$$P_j^{-1} \cdot P_j \cdot P_{IH}^{-1}(L(G_H)) = P_{IH}^{-1}(L(G_H))$$

We then note that $P_{IH} : \Sigma^* \to \Sigma_{IH}^*$, and $\Sigma'(j) = \Sigma_{IH} \cup \Sigma_{L_j}$, by definition.

We thus have $\Sigma_{IH} \subseteq \Sigma'(j)$.

We can now apply *Proposition 6* from [34] by taking $\Sigma_b = \Sigma_{IH}$ and $\Sigma_a = \Sigma'(j)$, and conclude:

$$P_j^{-1} \cdot P_j \cdot P_{IH}^{-1} = P_{IH}^{-1}$$

It follows immediately that: $P_j^{-1} \cdot P_j \cdot P_{IH}^{-1}(L(G_H)) = P_{IH}^{-1}(L(G_H))$

Point (b)-(f):
Proofs are identical to **Point (a)** after appropriate substitutions.

□

Our next proposition develops some useful properties of P-invariant languages. **Point (a)** essentially says that removing events from $\Sigma - \Sigma_1$ does not affect membership in languages L_k defined in the proposition. **Point (b)** says that the natural projection P and set intersection commute for P-invariant languages. **Point (c)**

says that the intersection of P-invariant languages is also P-invariant. **Point (d)** provides a useful relationship between strings $P(s)$ and s for P-invariant languages.

Proposition 2 *Let* $\Sigma_1 \subseteq \Sigma$, *language* $L_k \subseteq \Sigma^*, k = 1, 2, \ldots, n$, *natural projection* $P : \Sigma^* \to \Sigma_1^*$. *If* $P^{-1}(P(L_k)) = L_k$, *then*

(a) $P(L_k) \subseteq L_k$;

(b) $P(L_1) \cap P(L_2) \cap \ldots \cap P(L_n) = P(L_1 \cap L_2 \cap \ldots \cap L_n)$;

(c) $P^{-1}(P(L_1 \cap L_2 \cap \ldots \cap L_n)) = L_1 \cap L_2 \cap \ldots \cap L_n$.

(d) $(\forall s \in \Sigma^*)\, P(s) \in P(L_k) \Rightarrow s \in L_k$

Proof Assume $P^{-1}(P(L_k)) = L_k$, $k = 1, 2, \ldots, n$. **(A.1)**

Point (a): Show $P(L_k) \subseteq L_k$.

Let $s \in P(L_k)$. Sufficient to show $s \in L_k$.

$s \in P(L_k) \Rightarrow s \in \Sigma_1^*$

$\Rightarrow P(s) = s$ thus $P(s) \in P(L_k)$.

$s \in P^{-1}(P(L_k))$, by definition of P^{-1}.

We then have $s \in P^{-1}(P(L_k)) = L_k$ (by **(A.1)**) as required.

Point (b): Show $P(L_1) \cap P(L_2) \cap \ldots \cap P(L_n) = P(L_1 \cap L_2 \cap \ldots \cap L_n)$

We need to show:

(b.I) $P(L_1) \cap P(L_2) \cap \ldots \cap P(L_n) \subseteq P(L_1 \cap L_2 \cap \ldots \cap L_n)$ and

(b.II) $P(L_1 \cap L_2 \cap \ldots \cap L_n) \subseteq P(L_1) \cap P(L_2) \cap \ldots \cap P(L_n)$.

(b.I) Show $P(L_1) \cap P(L_2) \cap \ldots \cap P(L_n) \subseteq P(L_1 \cap L_2 \cap \ldots \cap L_n)$.

Let $s \in P(L_1) \cap P(L_2) \cap \ldots \cap P(L_n)$. **(A.2)**

Must show implies $s \in P(L_1 \cap L_2 \cap \ldots \cap L_n)$

By **Point (a)**, **(A.1)**, and **(A.2)**, we have $s \in L_1 \cap L_2 \cap \ldots \cap L_n$

$\Rightarrow P(s) \in P(L_1 \cap L_2 \cap \ldots \cap L_n)$ **(A.3)**

Since $s \in P(L_1)$ (by **(A.2)**), we have $s \in \Sigma_1^*$ by definition of the natural projection P.

We then have $P(s) = s$, and thus $s \in P(L_1 \cap L_2 \cap \ldots \cap L_n)$ (by **(A.3)**), as required.

(b.II) Show $P(L_1 \cap L_2 \cap \ldots \cap L_n) \subseteq P(L_1) \cap P(L_2) \cap \ldots \cap P(L_n)$.

Let $s \in P(L_1 \cap L_2 \cap \ldots \cap L_n)$. **(A.4)**

Must show implies $s \in P(L_1) \cap P(L_2) \cap \ldots \cap P(L_n)$.

From **(A.4)**, we know: $(\exists s' \in L_1 \cap L_2 \cap \ldots \cap L_n)\, P(s') = s$ **(A.5)**

We next note that $s' \in L_k \Rightarrow P(s') \in P(L_k)$, $k = 1, \ldots, n$.

$\Rightarrow s = P(s') \in P(L_1) \cap P(L_2) \cap \ldots \cap P(L_n)$ (by **(A.5)**), as required.

By **Part (b.I)** and **Part (b.II)**, we can now conclude:

$$P(L_1) \cap P(L_2) \cap \ldots \cap P(L_n) = P(L_1 \cap L_2 \cap \ldots \cap L_n)$$

Point (c): Show $P^{-1}(P(L_1 \cap L_2 \cap \ldots \cap L_n)) = L_1 \cap L_2 \cap \ldots \cap L_n$

We need to show:

(c.I) $P^{-1}(P(L_1 \cap L_2 \cap \ldots \cap L_n)) \subseteq L_1 \cap L_2 \cap \ldots \cap L_n$ and
(c.I) $L_1 \cap L_2 \cap \ldots \cap L_n \subseteq P^{-1}(P(L_1 \cap L_2 \cap \ldots \cap L_n))$.

(c.I) Show $P^{-1}(P(L_1 \cap L_2 \cap \ldots \cap L_n)) \subseteq L_1 \cap L_2 \cap \ldots \cap L_n$.

Let $s \in P^{-1}(P(L_1 \cap L_2 \cap \ldots \cap L_n))$ **(A.6)**

Must show implies $s \in L_1 \cap L_2 \cap \ldots \cap L_n$

From **(A.6)** and definition of P^{-1}, we have:
$P(s) \in P(L_1 \cap L_2 \cap \ldots \cap L_n)$.

$\Rightarrow P(s) \in P(L_1) \cap P(L_2) \cap \ldots \cap P(L_n)$, by **Point (b)**.

$\Rightarrow s \in P^{-1}P(L_k)$, $k = 1, \ldots, n$.

$\Rightarrow s \in L_k$, $k = 1, \ldots, n$, by **(A.1)**.

$\Rightarrow s \in L_1 \cap L_2 \cap \ldots \cap L_n$, as required.

(c.II) Show $L_1 \cap L_2 \cap \ldots \cap L_n \subseteq P^{-1}(P(L_1 \cap L_2 \cap \ldots \cap L_n))$.

Let $s \in L_1 \cap L_2 \cap \ldots \cap L_n$ **(A.7)**

Must show implies $s \in P^{-1}(P(L_1 \cap L_2 \cap \ldots \cap L_n)$

From **(A.7)** and definition of P, we can conclude:
$P(s) \in P(L_1 \cap L_2 \cap \ldots \cap L_n)$

By definition of P^{-1}, we can conclude:
$s \in P^{-1}(P(L_1 \cap L_2 \cap \ldots \cap L_n))$

By **Part (c.I)** and **Part (c.II)**, we can now conclude:
$P^{-1}(P(L_1 \cap L_2 \cap \ldots \cap L_n)) = L_1 \cap L_2 \cap \ldots \cap L_n$

Point (d): Show $(\forall s \in \Sigma^*)\, P(s) \in P(L_k) \Rightarrow s \in L_k$.

Let $s \in \Sigma^*$ and assume $P(s) \in P(L_k)$.

$\Rightarrow s \in P^{-1}(P(L_k))$, by definition of P^{-1}.

$\Rightarrow s \in L_k$ as $P^{-1}(P(L_k)) = L_k$ by **(A.1)**.

□

The next proposition provides a useful result about the controllability definition. If, in the left side of the *iff* condition below, we equate Σ_b with the set of uncontrollable events, L_2 with the language of the plant, L_3 with the language of the supervisor, and $L_1 = L_2 \cap L_3$, we have the controllability definition. The proposition essentially says that if L_k is P-invariant, controllability is not affected by removing all events in $\Sigma - \Sigma_a$.

Proposition 3 *For alphabet Σ, with event sets $\Sigma_b \subseteq \Sigma_a \subseteq \Sigma$, languages $L_1, L_2, L_3 \subseteq \Sigma^*$, and natural projection $P : \Sigma^* \to \Sigma_a^*$, if $P^{-1}(P(L_k)) = L_k$, $k = 1, 2, 3$, then*

$(\forall s \in L_1)\ Elig_{L_2}(s) \cap \Sigma_b \subseteq Elig_{L_3}(s) \Leftrightarrow (\forall s' \in P(L_1))\ Elig_{P(L_2)}(s') \cap \Sigma_b \subseteq Elig_{P(L_3)}(s')$

Proof

Assume $P^{-1}(P(L_k)) = L_k,\ k = 1, 2, 3.$ **(A.1)**

By the definition of Elig() operator, it is sufficient to show:

$[(\forall s \in L_1)(\forall \sigma \in \Sigma_b)\ s\sigma \in L_2 \Rightarrow s\sigma \in L_3] \quad \Leftrightarrow$ **(A.2)**

$[(\forall s' \in P(L_1))(\forall \sigma' \in \Sigma_b)\ s'\sigma' \in P(L_2) \Rightarrow s'\sigma' \in P(L_3)]$ **(A.3)**

We must show: **I) (A.2)** $\Rightarrow$ **(A.3)** and **II) (A.3)** $\Rightarrow$ **(A.2)**.

(I) Show **(A.2)** $\Rightarrow$ **(A.3)**.

Assume **(A.2)**.

Let $s' \in P(L_1)$, $\sigma' \in \Sigma_b$, and assume $s'\sigma' \in P(L_2)$. **(A.4)**

Must show implies $s'\sigma' \in P(L_3)$.

By *Proposition 2(a)* and **(A.1)**, we have $P(L_1) \subseteq L_1$

$\Rightarrow s' \in L_1$, by **(A.4)**.

Similarly, we have $s'\sigma' \in L_2$.

$\Rightarrow s'\sigma' \in L_3$, as, $\sigma' \in \Sigma_b$, and by **(A.2)**

$\Rightarrow P(s'\sigma') \in P(L_3)$. **(A.5)**

By **(A.4)**, $s' \in P(L_1) \subseteq \Sigma_1^*$, thus $P(s') = s'$, by definition of P.

$\Rightarrow P(s'\sigma') = s'P(\sigma') = s'\sigma'$, by definition of P, and fact $\Sigma_b \subseteq \Sigma_a$.

$\Rightarrow s'\sigma \in P(L_3)$ (by **(A.5)**), as required.

Part (I) complete.

(II) Show **(A.3)** $\Rightarrow$ **(A.2)**.

Assume **(A.3)**.

Let $s \in L_1$, $\sigma \in \Sigma_b$, and assume $s\sigma \in L_2$. **(A.6)**

Must show implies $s\sigma \in L_3$.

From **(A.6)**, we have:

$P(s) \in P(L_1)$ and, $P(s\sigma) = P(s)\sigma \in P(L_2)$, by definition of P, and fact $\Sigma_b \subseteq \Sigma_a$.

$\Rightarrow P(s)\sigma \in P(L_3)$, by **(A.3)**, after taking $s' = P(s)$ and $\sigma' = \sigma$.

$\Rightarrow P(s\sigma) \in P(L_3)$, as $\sigma \in \Sigma_b$ and $\Sigma_b \subseteq \Sigma_a$.

$\Rightarrow s\sigma \in L_3$, by **(A.1)**, and *Proposition 2(d)*.

Part (II) complete.

By **Parts (I)** and **(II)**, we have **(A.2)** $\Leftrightarrow$ **(A.3)**, as required.

□

4.2 Interface Consistency

In the original interface consistency definition, we first defined the *serial interface consistency* definition for the serial system consisting of DES $\mathbf{G}'_H$, $\mathbf{G}_L$, and $\mathbf{G}_I$.[2] We then used the concept of *serial system extractions* to extend the serial definition to the parallel case. We will first define some notation for the serial case, restate the original definitions, and then finally we will show that the original interface consistency definition is equivalent to the new one.

We assume that the alphabet partition for a serial system is specified by $\Sigma' := \Sigma'_H \dot{\cup} \Sigma_L \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A$, and then we introduce the following event sets, natural projections, and useful languages:

$$\begin{aligned}
\Sigma_I &:= \Sigma_R \dot{\cup} \Sigma_A, & P_{IH'} &: \Sigma'^* \to \Sigma^*_{IH'} \\
\Sigma_{IH'} &:= \Sigma'_H \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A, & P_{IL} &: \Sigma'^* \to \Sigma^*_{IL} \\
\Sigma_{IL} &:= \Sigma_L \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A, & P_I &: \Sigma'^* \to \Sigma^*_I \\
\mathcal{H}' &:= P^{-1}_{IH'}(L(\mathbf{G}'_H)), & \mathcal{H}'_m &:= P^{-1}_{IH'}(L_m(\mathbf{G}'_H)) \subseteq \Sigma'^* \\
\mathcal{L} &:= P^{-1}_{IL}(L(\mathbf{G}_L)), & \mathcal{L}_m &:= P^{-1}_{IL}(L_m(\mathbf{G}_L)) \subseteq \Sigma'^* \\
\mathcal{I} &:= P^{-1}_{I}(L(\mathbf{G}_I)), & \mathcal{I}_m &:= P^{-1}_{I}(L_m(\mathbf{G}_I)) \subseteq \Sigma'^*
\end{aligned}$$

When a system contained only one low level (serial case), we used the serial interface consistency definition given below.

[2]Some primes ("′") have been added to avoid confusion with the definitions in Section 3.3.

Definition 4.2.1 *The system composed of DES G'_H, G_L and G_I, is* serial interface consistent *with respect to the alphabet partition $\Sigma' := \Sigma'_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$, if the following properties hold,*

Multi-level Properties

1. *The event set of G'_H is Σ'_{IH}, and the event set of G_L is Σ_{IL}.*
2. *G_I is a command-pair interface.*

High Level Properties

3. $(\forall s \in \mathcal{H}' \cap \mathcal{I})\ Elig_{\mathcal{I}}(s) \cap \Sigma_A \subseteq Elig_{\mathcal{H}'}(s)$

Low Level Properties

4. $(\forall s \in \mathcal{L} \cap \mathcal{I})\ Elig_{\mathcal{I}}(s) \cap \Sigma_R \subseteq Elig_{\mathcal{L}}(s)$
5. $(\forall s \in \Sigma'^*.\Sigma_R \cap \mathcal{L} \cap \mathcal{I})$
   $Elig_{\mathcal{L}\cap\mathcal{I}}(s\Sigma_L^*) \cap \Sigma_A = Elig_{\mathcal{I}}(s) \cap \Sigma_A$ *where*
   $$Elig_{\mathcal{L}\cap\mathcal{I}}(s\Sigma_L^*) := \bigcup_{l\in\Sigma_L^*} Elig_{\mathcal{L}\cap\mathcal{I}}(sl)$$
6. $(\forall s \in \mathcal{L} \cap \mathcal{I})$
   $s \in \mathcal{I}_m \Rightarrow (\exists l \in \Sigma_L^*)\ sl \in \mathcal{L}_m \cap \mathcal{I}_m$

It's clear that for $n = 1$ and after appropriate relabeling, the interface consistency definition (Definition 3.4.1) reduces to the serial interface consistency definition; thus any result (such as in [34]) using the serial interface consistency definition would be immediately satisfied by Definition 3.4.1, with $n = 1$.

For the general case ($n \geq 1$ low levels), we need to extend our serial case definitions to the parallel case. As the event set of each low level is disjoint from the event sets of the other low levels, we can consider the parallel interface system as n serial interface systems (referred to as *serial system extractions*) by choosing

one low level and ignoring the others. This is shown conceptually in Fig. 4.1. The full definition is given below.

Definition 4.2.2 *For the n^{th} degree ($n \geq 1$) parallel interface system composed of DES $G_H, G_{L_1}, \ldots, G_{L_n}, G_{I_1}, \ldots, G_{I_n}$, with alphabet partition given by (3.1), the j^{th}* serial system extraction *(subsystem form), denoted by* system(j)*, is composed of the following elements:*

$$\begin{aligned}
G'_H(j) &:= G_H||G_{I_1}||\ldots||G_{I_{(j-1)}}||G_{I_{(j+1)}}||\ldots||G_{I_n} \\
G_L(j) &:= G_{L_j}, \qquad G_I(j) := G_{I_j} \\
\Sigma'_H(j) &:= \dot{\cup}_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}}\Sigma_{I_k}\,\dot{\cup}\,\Sigma_H \\
\Sigma_L(j) &:= \Sigma_{L_j}\ , \quad \Sigma_R(j) := \Sigma_{R_j}\ , \quad \Sigma_A(j) := \Sigma_{A_j} \\
\Sigma'(j) &:= \Sigma'_H(j)\,\dot{\cup}\,\Sigma_L(j)\,\dot{\cup}\,\Sigma_R(j)\,\dot{\cup}\,\Sigma_A(j) \\
&= \Sigma\ -\ \dot{\cup}_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}}\Sigma_{L_k}
\end{aligned}$$

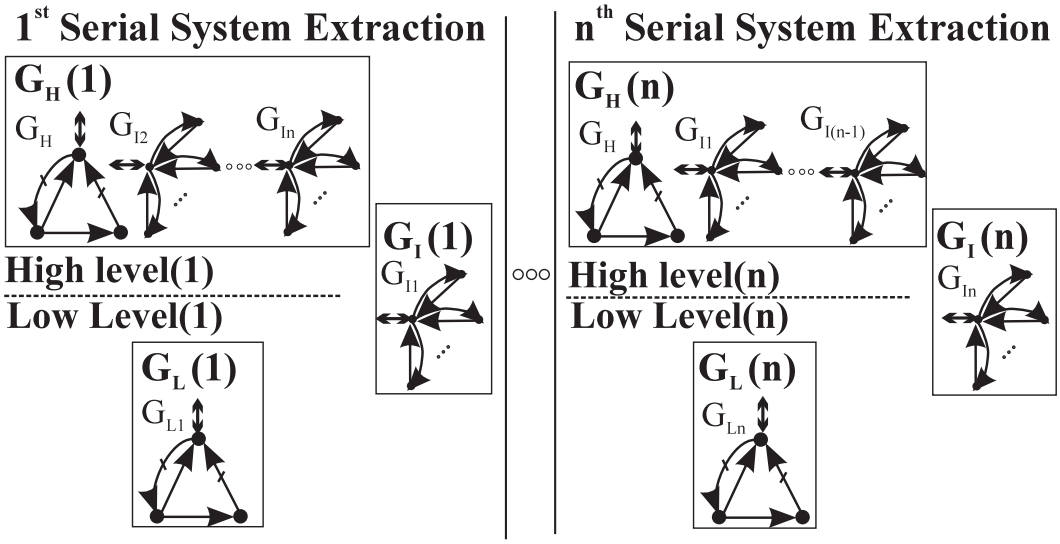

Figure 4.1: The Serial System Extractions

We are now ready to state the original interface consistency definition, for the parallel case.

Definition 4.2.3 *The n^{th} degree ($n \geq 1$) parallel interface system composed of DES G_H, $G_{L_1}, \ldots, G_{L_n}, G_{I_1}, \ldots, G_{I_n}$, is* interface consistent (ORIG) *with respect to alphabet partition given by (3.1), if for all $j \in \{1, \ldots, n\}$, the j^{th} serial system extraction of the system is serial interface consistent.*

Our next step is to introduce an intermediate form of the *interface consistency* definition, created from unrolling the interface consistency (ORIG) definition by applying the serial system extraction. This new form is easily obtainable from Definition 4.2.3 and has the same structure as Definition 3.4.1. This will make it easier to show that the two definitions are equivalent. To construct this new form of the definition, we will equate the components of a serial extraction system with the components of a serial system, and then interpret the notation of a serial interface system in the obvious way.

Definition 4.2.4 *The n^{th} degree ($n \geq 1$) parallel interface system composed of DES $\mathbf{G}_H, \mathbf{G}_{I_1}, \mathbf{G}_{L_1}, \ldots, \mathbf{G}_{I_n}, \mathbf{G}_{L_n}$, is* interface consistent (IntmORIG) *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 $\mathbf{G}'_H(j)$ 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 Property

3. $(\forall s \in \mathcal{H}'(j) \cap \mathcal{I}(j))\ Elig_{\mathcal{I}(j)}(s)\ \cap\ \Sigma_{A_j} \subseteq Elig_{\mathcal{H}'(j)}(s)$

Low Level Properties

4. $(\forall s \in \mathcal{L}(j) \cap \mathcal{I}(j))\ Elig_{\mathcal{I}(j)}(s)\ \cap\ \Sigma_{R_j} \subseteq Elig_{\mathcal{L}(j)}(s)$

5. $(\forall s \in \Sigma^*.\Sigma_{R_j} \cap \mathcal{L}(j) \cap \mathcal{I}(j))$

$$Elig_{\mathcal{L}(j)\cap\mathcal{I}(j)}(s\Sigma^*_{L_j}) \cap \Sigma_{A_j} = Elig_{\mathcal{I}(j)}(s) \cap \Sigma_{A_j} \quad where$$

$$Elig_{\mathcal{L}(j)\cap\mathcal{I}(j)}(s\Sigma^*_{L_j}) := \bigcup_{l\in\Sigma^*_{L_j}} Elig_{\mathcal{L}(j)\cap\mathcal{I}(j)}(sl)$$

6. $(\forall s \in \mathcal{L}(j) \cap \mathcal{I}(j))$

$$s \in \mathcal{I}_m(j) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ sl \in \mathcal{L}_m(j) \cap \mathcal{I}_m(j).$$

We will now show that the intermediate form is equivalent to the original form of the interface consistency definition.

Proposition 4 *The n^{th} degree ($n \geq 1$) parallel interface system composed of DES $\mathbf{G}_H$, $\mathbf{G}_{L_1}, \ldots, \mathbf{G}_{L_n}, \mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$, is interface consistent (IntmORIG) (Definition 4.2.4) with respect to alphabet partition given by (3.1), iff, the system is interface consistent (ORIG) (Definition 4.2.3) with respect to alphabet partition given by (3.1).*

Proof

We will prove this by starting with Definition 4.2.3 and converting it into the form of Definition 4.2.4.

If Definition 4.2.3 is satisfied, then for all $j \in \{1, \ldots, n\}$, the j^{th} *serial system extraction* (subsystem form), denoted by *system*(j), is serial interface consistent.

That means for all $j \in \{1, \ldots, n\}$, the following conditions are satisfied: **(A.1)**

1. The event set of $G'_H(j)$ is $\Sigma'_{IH}(j)$, and the event set of $G_L(j)$ is $\Sigma_{IL}(j)$.
2. $G_I(j)$ is a command-pair interface.
3. $(\forall s \in \mathcal{H}'(j) \cap \mathcal{I}(j))\ \text{Elig}_{\mathcal{I}(j)}(s) \cap \Sigma_A(j) \subseteq \text{Elig}_{\mathcal{H}'(j)}(s)$
4. $(\forall s \in \mathcal{L}(j) \cap \mathcal{I}(j))\ \text{Elig}_{\mathcal{I}(j)}(s) \cap \Sigma_R(j) \subseteq \text{Elig}_{\mathcal{L}(j)}(s)$

5. $(\forall s \in \Sigma'^*(j).\Sigma_R(j) \cap \mathcal{L}(j) \cap \mathcal{I}(j))$
   $\text{Elig}_{\mathcal{L}(j)\cap\mathcal{I}(j)}(s\Sigma_L^*(j)) \cap \Sigma_A(j) = \text{Elig}_{\mathcal{I}(j)}(s) \cap \Sigma_A(j)$ where

$$\text{Elig}_{\mathcal{L}(j)\cap\mathcal{I}(j)}(s\Sigma_L^*(j)) := \bigcup_{l\in\Sigma_L^*(j)} \text{Elig}_{\mathcal{L}(j)\cap\mathcal{I}(j)}(sl)$$

6. $(\forall s \in \mathcal{L}(j) \cap \mathcal{I}(j))$
   $s \in \mathcal{I}_m(j) \Rightarrow (\exists l \in \Sigma_L^*(j))\ sl \in \mathcal{L}_m(j) \cap \mathcal{I}_m(j)$

We next note the following facts:

- From Definition 4.2.2, we know that $G_L(j) = G_{L_j}$, $G_I(j) = G_{I_j}$, $\Sigma_L(j) = \Sigma_{L_j}$, $\Sigma_R(j) = \Sigma_{R_j}$, $\Sigma_A(j) = \Sigma_{A_j}$, and $\Sigma'(j) = \Sigma - \dot{\cup}_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}}\Sigma_{L_k}$ **(A.2)**

- From *Proposition 21* in [34], we know that that G_{I_j} is defined over event set Σ_{IL_j}. **(A.3)**

- From *Proposition 23* in [34], we know that $\Sigma'_{IH}(j) = \Sigma_{IH}$, and $\Sigma_{IL}(j) = \Sigma_{IL_j}$. **(A.4)**

- We note that $\Sigma^*.\Sigma_{R_j} \cap \mathcal{L}(j) \cap \mathcal{I}(j)) = \Sigma'^*(j).\Sigma_{R_j} \cap \mathcal{L}(j) \cap \mathcal{I}(j))$ as $\Sigma'^*(j).\Sigma_{R_j} \subseteq \Sigma^*.\Sigma_{R_j}$, $\mathcal{L}(j) \subseteq \Sigma'^*(j)$ thus $(\Sigma^*.\Sigma_{R_j} - \Sigma'^*(j).\Sigma_{R_j}) \cap \mathcal{L}(j) = \emptyset$. **(A.5)**

Now, substituting the results of **(A.2)** - **(A.5)** into **(A.1)**, we can conclude that, for all $j \in \{1, \ldots, n\}$, the following conditions are satisfied:

1. The event set of $\mathbf{G}'_H(j)$ 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.

3. $(\forall s \in \mathcal{H}'(j) \cap \mathcal{I}(j))\ \text{Elig}_{\mathcal{I}(j)}(s) \cap \Sigma_{A_j} \subseteq \text{Elig}_{\mathcal{H}'(j)}(s)$

4. $(\forall s \in \mathcal{L}(j) \cap \mathcal{I}(j))\ \text{Elig}_{\mathcal{I}(j)}(s) \cap \Sigma_{R_j} \subseteq \text{Elig}_{\mathcal{L}(j)}(s)$

5. $(\forall s \in \Sigma^*.\Sigma_{R_j} \cap \mathcal{L}(j) \cap \mathcal{I}(j))$
$\text{Elig}_{\mathcal{L}(j)\cap\mathcal{I}(j)}(s\Sigma^*_{L_j}) \cap \Sigma_{A_j} = \text{Elig}_{\mathcal{I}(j)}(s) \cap \Sigma_{A_j}$ where
$$\text{Elig}_{\mathcal{L}(j)\cap\mathcal{I}(j)}(s\Sigma^*_{L_j}) := \bigcup_{l\in\Sigma^*_{L_j}} \text{Elig}_{\mathcal{L}(j)\cap\mathcal{I}(j)}(sl)$$

6. $(\forall s \in \mathcal{L}(j) \cap \mathcal{I}(j))$
$s \in \mathcal{I}_m(j) \Rightarrow (\exists l \in \Sigma^*_{L_j})\ sl \in \mathcal{L}_m(j) \cap \mathcal{I}_m(j).$

which is exactly equal to Definition 4.2.4, as required.

□

We conclude this section by presenting our theorem that shows that our new definition of interface consistency is equivalent to the original.

Theorem 3 *The n^{th} degree ($n \geq 1$) parallel interface system composed of DES $\mathbf{G}_H, \mathbf{G}_{L_1}, \ldots, \mathbf{G}_{L_n}, \mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$, is interface consistent (Definition 3.4.1) with respect to alphabet partition given by (3.1), iff, the system is interface consistent (ORIG) (Definition 4.2.3) with respect to alphabet partition given by (3.1).*

Proof

We first note that as Definition 4.2.4 is equivalent to Definition 4.2.3 by *Proposition 4*, it is sufficient to show:

$$\text{Definition 3.4.1} \Leftrightarrow \text{Definition 4.2.4}$$

As the two definitions are almost exactly of the same form, we will prove this point by point, for each of the six points of the two definitions.

1. For **point (1)**, the two definitions are almost the same already, thus we only have to account for the difference.

   a) Assume Definition 4.2.4 is satisfied.

From *Proposition 21* in [34], we can immediately conclude that the event set of $\mathbf{G}_H$ is Σ_{IH}.

b) Assume Definition 3.4.1 is satisfied.

Must show this implies the event set of $\mathbf{G}'_H(j)$ is Σ_{IH}.

By definition, $\mathbf{G}'_H(j) = G_H||G_{I_1}||\ldots||G_{I_{(j-1)}}||G_{I_{(j+1)}}||\ldots||G_{I_n}$.

From **point (1)** of Definition 3.4.1, we know that the event set of $\mathbf{G}_H$ is Σ_{IH}. From the command-pair interface definition, and **point (1)** of Definition 3.4.1, we know that the event set of G_{I_k} is Σ_{I_k} $(k = 1, \ldots j-1, j+1, \ldots n)$.

We thus have the event set of $\mathbf{G}'_H(j)$ is:

$$\Sigma_{\mathbf{G}'_H(j)} = \cup_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}}\Sigma_{I_k} \cup \Sigma_{IH} = \Sigma_{IH}$$

2. **Point (2)** is automatic.

3. For **Point (3)**, we need to show:

(Definition 4.2.4) $(\forall s \in \mathcal{H}'(j) \cap \mathcal{I}(j))$ $\text{Elig}_{\mathcal{I}(j)}(s) \cap \Sigma_{A_j} \subseteq \text{Elig}_{\mathcal{H}'(j)}(s) \Leftrightarrow$ **(A.1)**

(Definition 3.4.1) $(\forall s \in \mathcal{H} \cap \bigcap_{k=1,\ldots,n}\mathcal{I}_k)$ $\text{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{A_j} \subseteq \text{Elig}_{\mathcal{H}\cap \bigcap_{k\neq j}\mathcal{I}_k}(s)$ **(A.2)**

We will start by massaging **(A.1)** into the correct form so that we can apply *Proposition 3*.

We first note that by *Proposition 1*, the languages $\mathcal{H}$ and $\mathcal{I}_k$ $(k = 1, \ldots, n)$ are P_j-invariant. That means that we can apply *Proposition 2* to them.

From *Proposition 23* of [34], we have:

$$\begin{aligned}
\mathcal{H}'(j) &= P_j(\mathcal{H}) \cap \bigcap_{k=1,\ldots,(j-1),(j+1),\ldots,n} P_j(\mathcal{I}_k) \\
&= P_j(\mathcal{H}) \cap \bigcap_{k\neq j} P_j(\mathcal{I}_k) \\
&= P_j(\mathcal{H} \cap \bigcap_{k\neq j} \mathcal{I}_k), \qquad \text{by } \textit{Proposition 2(b)} \qquad (\mathbf{A.3}) \\
\mathcal{I}(j) &= P_j(\mathcal{I}_j) \qquad (\mathbf{A.4}) \\
\mathcal{H}'(j) \cap \mathcal{I}(j)) &= P_j(\mathcal{H}) \cap \bigcap_{k\neq j} P_j(\mathcal{I}_k) \ \cap P_j(\mathcal{I}_j) \\
&= P_j(\mathcal{H} \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_k), \qquad \text{by } \textit{Proposition 2(b)} \qquad (\mathbf{A.5})
\end{aligned}$$

Substituting from **(A.3)**-**(A.5)** into **(A.1)**, we have:

$$(\forall s \in P_j(\mathcal{H} \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_k)\ \mathrm{Elig}_{P_j(\mathcal{I}_j)}(s) \cap \Sigma_{A_j} \subseteq \mathrm{Elig}_{P_j(\mathcal{H}\cap\bigcap_{k\neq j}\mathcal{I}_k)}(s) \qquad (\mathbf{A.6})$$

By *Proposition 1* and *Proposition 2(c)*, we have:

$P_j^{-1}(P_j(\mathcal{H} \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_k)) = \mathcal{H} \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_k$ and $P_j^{-1}(P_j(\mathcal{H} \cap \bigcap_{k\neq j} \mathcal{I}_k)) = \mathcal{H} \cap \bigcap_{k\neq j} \mathcal{I}_k$

In other words, languages $\mathcal{H} \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_k$ and $\mathcal{H} \cap \bigcap_{k\neq j} \mathcal{I}_k$ are P_j-invariant. **(A.7)**

We next note that, by *Proposition 1*, we have $P_j^{-1}(P_j(\mathcal{I}_j)) = \mathcal{I}_j$, and by definition, we have $\Sigma_{A_j} \subseteq \Sigma'^*(j)$.

We now take $\Sigma_b = \Sigma_{A_j}$, $\Sigma_a = \Sigma'^*(j)$, $P = P_j$, $L_1 = \mathcal{H} \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_k$, $L_2 = \mathcal{I}_j$, $L_3 = \mathcal{H} \cap \bigcap_{k\neq j} \mathcal{I}_k$, and we can conclude by *Proposition 3*, and **(A.7)** that:

$$\begin{aligned}
& (\forall s \in P_j(\mathcal{H} \cap \bigcap_{k\in\{1,\ldots,n\}} \mathcal{I}_k))\ \mathrm{Elig}_{P_j(\mathcal{I}_j)}(s) \cap \Sigma_{A_j} \subseteq \mathrm{Elig}_{P_j(\mathcal{H}\cap\bigcap_{k\neq j}\mathcal{I}_k)}(s) \\
\Leftrightarrow \quad & (\forall s \in \mathcal{H} \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_k)\ \mathrm{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{A_j} \subseteq \mathrm{Elig}_{\mathcal{H}\cap\bigcap_{k\neq j}\mathcal{I}_k}(s)
\end{aligned}$$

We can now conclude by **(A.6)**, that **(A.1)** $\Leftrightarrow$ **(A.2)**, as required.

4. For **Point (4)**, we need to show:

   (Definition 4.2.4) $(\forall s \in \mathcal{L}(j) \cap \mathcal{I}(j))\ \text{Elig}_{\mathcal{I}(j)}(s) \cap \Sigma_{R_j} \subseteq \text{Elig}_{\mathcal{L}(j)}(s)$ $\Leftrightarrow$ **(A.8)**

   (Definition 3.4.1) $(\forall s \in \mathcal{L}_j \cap \mathcal{I}_j)\ \text{Elig}_{\mathcal{I}_j}(s)\ \cap\ \Sigma_{R_j} \subseteq \text{Elig}_{\mathcal{L}_j}(s)$ **(A.9)**

   From *Proposition 23* of [34], we have $\mathcal{I}(j) = P_j(\mathcal{I}_j)$, and $\mathcal{L}(j) = P_j(\mathcal{L}_j)$. We also note that by *Proposition 1*, $\mathcal{L}_j$ and $\mathcal{I}_j$ are P_j-invariant. **(A.10)**

   $\Rightarrow P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j) = P_j(\mathcal{L}_j \cap \mathcal{I}_j))$, by *Proposition 2(b)*.

   Substituting this and **(A.10)** into **(A.8)**, we get:

   $$(\forall s \in P_j(\mathcal{L}_j \cap \mathcal{I}_j))\ \text{Elig}_{P_j(\mathcal{I}_j)}(s)\ \cap\ \Sigma_{R_j} \subseteq \text{Elig}_{P_j(\mathcal{L}_j)}(s) \qquad (\mathbf{A.11})$$

   By *Proposition 1* and *Proposition 2(c)*, we have:

   $$P_j^{-1}(P_j(\mathcal{L}_j \cap \mathcal{I}_j)) = \mathcal{L}_j \cap \mathcal{I}_j$$

   In other words, the language $\mathcal{L}_j \cap \mathcal{I}_j$ is P_j-invariant. **(A.12)**

   We next note that, by definition, $\Sigma_{R_j} \subseteq \Sigma'^*(j)$.

   We now take $\Sigma_b = \Sigma_{R_j}$, $\Sigma_a = \Sigma'^*(j)$, $L_1 = \mathcal{L}_j \cap \mathcal{I}_j$, $L_2 = \mathcal{I}_j$, $L_3 = \mathcal{L}_j$, and we can conclude by *Proposition 3*, **(A.10)**, and **(A.12)** that:

   $$\begin{aligned} & (\forall s \in P_j(\mathcal{L}_j \cap \mathcal{I}_j))\ \text{Elig}_{P_j(\mathcal{I}_j)}(s)\ \cap\ \Sigma_{R_j} \subseteq \text{Elig}_{P_j(\mathcal{L}_j)}(s) \\ \Leftrightarrow \quad & (\forall s \in \mathcal{L}_j \cap \mathcal{I}_j)\ \text{Elig}_{\mathcal{I}_j}(s)\ \cap\ \Sigma_{R_j}\ \subseteq \text{Elig}_{\mathcal{L}_j}(s) \end{aligned}$$

   We can now conclude by **(A.11)**, that **(A.8)** $\Leftrightarrow$ **(A.9)**, as required.

5. For **Point (5)**, we need to show:

   (Definition 4.2.4)
   $(\forall s \in \Sigma^*.\Sigma_{R_j} \cap \mathcal{L}(j) \cap \mathcal{I}(j))\, \text{Elig}_{\mathcal{L}(j) \cap \mathcal{I}(j)}(s\Sigma_{L_j}^*)\ \cap\ \Sigma_{A_j} = \text{Elig}_{\mathcal{I}(j)}(s)\ \cap\ \Sigma_{A_j}$

$\Leftrightarrow$ **(A.13)**

$$\text{where } \text{Elig}_{\mathcal{L}(j)\cap\mathcal{I}(j)}(s\Sigma^*_{L_j}) := \bigcup_{l\in\Sigma^*_{L_j}} \text{Elig}_{\mathcal{L}(j)\cap\mathcal{I}(j)}(sl)$$

(Definition 3.4.1) $(\forall s \in \Sigma^*.\Sigma_{R_j}\cap\mathcal{L}_j\cap\mathcal{I}_j)\,\text{Elig}_{\mathcal{L}_j\cap\mathcal{I}_j}(s\Sigma^*_{L_j})\cap\Sigma_{A_j} = \text{Elig}_{\mathcal{I}_j}(s)\cap\Sigma_{A_j}$ **(A.14)**

$$\text{where } \text{Elig}_{\mathcal{L}_j\cap\mathcal{I}_j}(s\Sigma^*_{L_j}) := \bigcup_{l\in\Sigma^*_{L_j}} \text{Elig}_{\mathcal{L}_j\cap\mathcal{I}_j}(sl)$$

From *Proposition 23* of [34], we have $\mathcal{I}(j) = P_j(\mathcal{I}_j)$, and $\mathcal{L}(j) = P_j(\mathcal{L}_j)$. We also note that by *Proposition 1*, $\mathcal{L}_j$ and $\mathcal{I}_j$ are P_j-invariant. **(A.15)**

Substituting into **(A.13)**, we get:

$(\forall s \in \Sigma^*.\Sigma_{R_j}\cap P_j(\mathcal{L}_j)\cap P_j(\mathcal{I}_j))$

$\text{Elig}_{P_j(\mathcal{L}_j)\cap P_j(\mathcal{I}_j)}(s\Sigma^*_{L_j}) \cap \Sigma_{A_j} = \text{Elig}_{P_j(\mathcal{I}_j)}(s) \cap \Sigma_{A_j}$

$$\text{where } \text{Elig}_{P_j(\mathcal{L}_j)\cap P_j(\mathcal{I}_j)}(s\Sigma^*_{L_j}) := \bigcup_{l\in\Sigma^*_{L_j}} \text{Elig}_{P_j(\mathcal{L}_j)\cap P_j(\mathcal{I}_j)}(sl)$$

By the definition of Elig() operator, it is sufficient to show:

$(\forall s \in \Sigma^*.\Sigma_{R_j}\cap P_j(\mathcal{L}_j)\cap P_j(\mathcal{I}_j))(\forall\alpha\in\Sigma_{A_j})$ **(A.16)**

$[s\Sigma^*_{L_j}\alpha\cap P_j(\mathcal{L}_j)\cap P_j(\mathcal{I}_j)\neq\emptyset \Leftrightarrow s\alpha\in P_j(\mathcal{I}_j)]$ $\Leftrightarrow$

$(\forall s' \in \Sigma^*.\Sigma_{R_j}\cap\mathcal{L}_j\cap\mathcal{I}_j)(\forall\alpha'\in\Sigma_{A_j})\,[s'\Sigma^*_{L_j}\alpha'\cap\mathcal{L}_j\cap\mathcal{I}_j\neq\emptyset \Leftrightarrow s'\alpha'\in\mathcal{I}_j]$

(A.17)

We first observe that: **(A.18)**

$(\forall l\in\Sigma^*_{L_j})\,P_j(l) = l$, as $\Sigma_{L_j}\subseteq\Sigma'(j)$.

$(\forall\rho\in\Sigma_{R_j})\,P_j(\rho) = \rho$, as $\Sigma_{R_j}\subseteq\Sigma'(j)$.

$(\forall\alpha\in\Sigma_{A_j})\,P_j(\alpha) = \alpha$, as $\Sigma_{A_j}\subseteq\Sigma'(j)$.

In order to prove **(A.16)** $\Leftrightarrow$ **(A.17)**, we need to show: **(I)** **(A.16)** $\Rightarrow$ **(A.17)** and **(II)** **(A.17)** $\Rightarrow$ **(A.16)**

(I) Show **(A.16)** $\Rightarrow$ **(A.17)**.

Assume **(A.16)**. Must show implies **(A.17)**.

Let $s' \in \Sigma^*.\Sigma_{R_j} \cap \mathcal{L}_j \cap \mathcal{I}_j$ and $\alpha' \in \Sigma_{A_j}$ **(A.19)**

Must show: **(I.a)** $s'\Sigma^*_{L_j}\alpha' \cap \mathcal{L}_j \cap \mathcal{I}_j \neq \emptyset \Rightarrow s'\alpha' \in \mathcal{I}_j$ and **(I.b)** $s'\alpha' \in \mathcal{I}_j \Rightarrow s'\Sigma^*_{L_j}\alpha' \cap \mathcal{L}_j \cap \mathcal{I}_j \neq \emptyset$.

We first note that as $s' \in \Sigma^*.\Sigma_{R_j} \cap \mathcal{L}_j \cap \mathcal{I}_j$ (by **(A.19)**), it follows that:

$P_j(s') \in P_j(\Sigma^*.\Sigma_{R_j}) \cap P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j)$

As $P_j(\Sigma^*.\Sigma_{R_j}) = \Sigma'^*(j).\Sigma_{R_j}$ (by **(A.18)** and definition of P_j) and $\Sigma'^*(j).\Sigma_{R_j} \subseteq \Sigma^*.\Sigma_{R_j}$ (by definition of $\Sigma'^*(j)$), we have:

$P_j(s') \in \Sigma^*.\Sigma_{R_j} \cap P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j)$ **(A.20)**

(I.a) Show $s'\Sigma^*_{L_j}\alpha' \cap \mathcal{L}_j \cap \mathcal{I}_j \neq \emptyset \Rightarrow s'\alpha' \in \mathcal{I}_j$

Assume $s'\Sigma^*_{L_j}\alpha' \cap \mathcal{L}_j \cap \mathcal{I}_j \neq \emptyset$ **(A.21)**

Must show implies $s'\alpha' \in \mathcal{I}_j$

From **(A.21)**, we can conclude:

$(\exists l \in \Sigma^*_{L_j})\, s'l\alpha' \in \mathcal{L}_j \cap \mathcal{I}_j$

$\Rightarrow P_j(s'l\alpha') \in P_j(\mathcal{L}_j \cap \mathcal{I}_j)$

$\Rightarrow P_j(s')l\alpha' \in P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j)$, by **(A.15)**, **(A.18)**, and *Proposition 2(b)*.

$\Rightarrow P_j(s')\Sigma^*_{L_j}\alpha' \cap P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j) \neq \emptyset$ **(A.22)**

By **(A.20)**, **(A.19)**, **(A.22)**, and taking $s = P_j(s')$, $\alpha = \alpha'$, we can apply **(A.16)** and conclude:

$P_j(s')\alpha' \in P_j(\mathcal{I}_j)$.

$\Rightarrow P_j(s'\alpha') \in P_j(\mathcal{I}_j)$, by **(A.18)**

$\Rightarrow s'\alpha' \in \mathcal{I}_j$, by **(A.15)**, and *Proposition 2(d)*.

Part (I.a) complete.

(I.b) Show $s'\alpha' \in \mathcal{I}_j \Rightarrow s'\Sigma_{L_j}^*\alpha' \cap \mathcal{L}_j \cap \mathcal{I}_j \neq \emptyset$.

Let $s'\alpha' \in \mathcal{I}_j$. **(A.23)**

Must show $(\exists l \in \Sigma_{L_j}^*)\, s'l\alpha' \in \mathcal{L}_j \cap \mathcal{I}_j$.

From **(A.23)**, we can conclude:

$P_j(s'\alpha') \in P_j(\mathcal{I}_j)$

$\Rightarrow P_j(s')\alpha' \in P_j(\mathcal{I}_j)$, by **(A.18)**. **(A.24)**

By **(A.20)**, **(A.19)**, **(A.24)**, and taking $s = P_j(s')$, $\alpha = \alpha'$, we can apply **(A.16)** and conclude:

$(\exists l \in \Sigma_{L_j}^*)\, P_j(s')l\alpha' \in P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j)$

$\Rightarrow P_j(s'l\alpha') \in P_j(\mathcal{L}_j \cap \mathcal{I}_j)$, by **(A.18)**, **(A.15)**, and *Proposition 2(b).*

$\Rightarrow s'l\alpha' \in \mathcal{L}_j \cap \mathcal{I}_j$, by **(A.15)**, and *Proposition 2(c), (d).*

Part (I.b) complete.

By **Parts (I.a)** and **(I.b)**, we have **(A.16)** $\Rightarrow$ **(A.17)**.

Part (I) complete.

(II) Show **(A.17)** $\Rightarrow$ **(A.16)**.

Assume **(A.17)**. Must show implies **(A.16)**.

Let $s \in \Sigma^*.\Sigma_{R_j} \cap P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j)$ and $\alpha \in \Sigma_{A_j}$. **(A.25)**

Must show: **(II.a)** $s\Sigma_{L_j}^*\alpha \cap P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j) \neq \emptyset \Rightarrow s\alpha \in P_j(\mathcal{I}_j)$ and **(II.b)** $s\alpha \in P_j(\mathcal{I}_j) \Rightarrow s\Sigma_{L_j}^*\alpha \cap P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j) \neq \emptyset$.

We first note that by **(A.25)**, we have:

$s \in P_j(\mathcal{L}_j)$ and $s \in P_j(\mathcal{I}_j)$

$\Rightarrow s \in \mathcal{L}_j \cap \mathcal{I}_j$ by **(A.15)** and *Proposition 2(a)*.

$\Rightarrow s \in \Sigma^*.\Sigma_{R_j} \cap \mathcal{L}_j \cap \mathcal{I}_j$, by **(A.25)**. **(A.26)**

(II.a) Show $s\Sigma^*_{L_j}\alpha \cap P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j) \neq \emptyset \Rightarrow s\alpha \in P_j(\mathcal{I}_j)$.

Assume $s\Sigma^*_{L_j}\alpha \cap P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j) \neq \emptyset$ **(A.27)**

Must show implies $s\alpha \in P_j(\mathcal{I}_j)$.

From **(A.27)**, we can conclude:

$(\exists l \in \Sigma^*_{L_j})\, sl\alpha \in P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j)$

$\Rightarrow sl\alpha \in \mathcal{L}_j \cap \mathcal{I}_j$, by **(A.15)** and *Proposition 2(a)*.

$\Rightarrow s\Sigma^*_{L_j}\alpha \cap \mathcal{L}_j \cap \mathcal{I}_j \neq \emptyset$ **(A.28)**

By **(A.25)**, **(A.26)**, **(A.28)**, and taking $s' = s$ and $\alpha' = \alpha$, we can apply **(A.17)** and conclude:

$s\alpha \in \mathcal{I}_j$

$\Rightarrow P_j(s\alpha) \in P_j(\mathcal{I}_j)$ **(A.29)**

We first note that by **(A.25)**, we have $s \in P_j(\mathcal{L}_j) \subseteq \Sigma'^*(j)$

$\Rightarrow P_j(s\alpha) = s\alpha$, by **(A.18)**, and definition of P_j.

$\Rightarrow s\alpha \in P_j(\mathcal{I}_j)$, by **(A.29)**.

Part (II.a) complete.

(II.b) Show $s\alpha \in P_j(\mathcal{I}_j) \Rightarrow s\Sigma^*_{L_j}\alpha \cap P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j) \neq \emptyset$.

Assume $s\alpha \in P_j(\mathcal{I}_j)$. **(A.30)**

Must show $(\exists l \in \Sigma^*_{L_j})\, sl\alpha \in P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j)$.

From **(A.15)**, **(A.30)**, and *Proposition 2(a)*, we can conclude: $s\alpha \in \mathcal{I}_j$ **(A.31)**

By **(A.26)**, **(A.25)**, **(A.31)**, and taking $s' = s$ and $\alpha' = \alpha$, we can apply **(A.17)** and conclude:

$(\exists l \in \Sigma^*_{L_j})\, sl\alpha \in \mathcal{L}_j \cap \mathcal{I}_j$ **(A.32)**

$\Rightarrow P_j(sl\alpha) \in P_j(\mathcal{L}_j \cap \mathcal{I}_j)$

$\Rightarrow P_j(sl\alpha) \in P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j)$, by **(A.15)** and *Proposition 2(b)*.

We next note that by **(A.25)**, we have $s \in P_j(\mathcal{L}_j) \subseteq \Sigma'^*(j)$

$\Rightarrow P_j(sl\alpha) = sl\alpha$, by **(A.18)**, **(A.32)**, and definition of P_j.

$\Rightarrow sl\alpha \in P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j)$, as required.

Part (II.b) complete.

By **Parts (II.a)** and **(II.b)**, we have **(A.17)** $\Rightarrow$ **(A.16)**.

Part (II) complete.

By **Parts (I)** and **Part (II)**, we have **(A.16)** $\Leftrightarrow$ **(A.17)**, as required.

6. For **Point (6)**, we need to show:

(Definition 4.2.4)

$(\forall s \in \mathcal{L}(j) \cap \mathcal{I}(j))\, s \in \mathcal{I}_m(j) \Rightarrow (\exists l \in \Sigma^*_{L_j})\, sl \in \mathcal{L}_m(j) \cap \mathcal{I}_m(j)$ $\Leftrightarrow$ **(A.33)**

(Definition 3.4.1) $(\forall s' \in \mathcal{L}_j \cap \mathcal{I}_j)\, s' \in \mathcal{I}_{m_j} \Rightarrow (\exists l' \in \Sigma^*_{L_j})\ s'l' \in \mathcal{L}_{m_j} \cap \mathcal{I}_{m_j}$ **(A.34)**

From *Proposition 23* of [34], we have $\mathcal{I}(j) = P_j(\mathcal{I}_j)$, $\mathcal{L}(j) = P_j(\mathcal{L}_j)$, $\mathcal{I}_m(j) = P_j(\mathcal{I}_{m_j})$, and $\mathcal{L}_m(j) = P_j(\mathcal{L}_{m_j})$. We also note that by *Proposition 1*, $\mathcal{L}_j$, $\mathcal{I}_j$, $\mathcal{L}_{m_j}$, and $\mathcal{I}_{m_j}$ are P_j-invariant. **(A.35)**

Substituting into **(A.33)**, we get:

$(\forall s \in P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j))\, s \in P_j(\mathcal{I}_{m_j}) \Rightarrow (\exists l \in \Sigma_{L_j}^*)\ sl \in P_j(\mathcal{L}_{m_j}) \cap P_j(\mathcal{I}_{m_j})$ **(A.36)**

It is thus sufficient to show **(A.36)** $\Leftrightarrow$ **(A.34)**.

To do this, we need to show: **(I)** **(A.36)** $\Rightarrow$ **(A.34)** and **(II)** **(A.34)** $\Rightarrow$ **(A.36)**.

(I) Show **(A.36)** $\Rightarrow$ **(A.34)**.

Assume **(A.36)**. Must show implies **(A.34)**.

Let $s' \in \mathcal{L}_j \cap \mathcal{I}_j$. Assume $s' \in \mathcal{I}_{m_j}$. **(A.37)**

We must show implies $(\exists l' \in \Sigma_{L_j}^*)\ s'l' \in \mathcal{L}_{m_j} \cap \mathcal{I}_{m_j}$.

From **(A.37)**, we can conclude:

$P_j(s') \in P_j(\mathcal{L}_j \cap \mathcal{I}_j)$ and $P_j(s') \in P_j(\mathcal{I}_{m_j})$

$\Rightarrow P_j(s') \in P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j))$ and $P_j(s') \in P_j(\mathcal{I}_{m_j})$ by **(A.35)**, and *Proposition 2(b)*.

We can now apply **(A.36)** by taking $s = P_j(s')$, and conclude:

$(\exists l' \in \Sigma_{L_j}^*)\ P_j(s')l' \in P_j(\mathcal{L}_{m_j}) \cap P_j(\mathcal{I}_{m_j})$.

As $\Sigma_{L_j}^* \subseteq \Sigma'(j)$, and by **(A.35)**, *Proposition 2(b)*, and definition of P_j, we can conclude:

$P_j(s'l') \in P_j(\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})$.

We can now conclude by **(A.35)**, and *Proposition 2(c), (d)*:

$s'l' \in \mathcal{L}_{m_j} \cap \mathcal{I}_{m_j}$, as required.

Part (I) complete.

(II) Show **(A.34)** $\Rightarrow$ **(A.36)**.

Assume **(A.34)**. Must show implies **(A.36)**.

Let $s \in P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j)$, and assume $s \in P_j(\mathcal{I}_{m_j})$. **(A.38)**

We must show implies $(\exists l \in \Sigma_{L_j}^*)\ sl \in P_j(\mathcal{L}_{m_j}) \cap P_j(\mathcal{I}_{m_j})$.

From **(A.38)**, we can conclude:

$s \in P_j(\mathcal{L}_j)$, $s \in P_j(\mathcal{I}_j)$, and $s \in P_j(\mathcal{I}_{m_j})$

$\Rightarrow s \in \mathcal{L}_j \cap \mathcal{I}_j$ and $s \in \mathcal{I}_{m_j}$, by **(A.35)**, and *Proposition 2(a)*.

We can now apply **(A.34)** by taking $s' = s$, and conclude:

$(\exists l \in \Sigma_{L_j}^*)\ sl \in \mathcal{L}_{m_j} \cap \mathcal{I}_{m_j}$

$\Rightarrow P_j(sl) \in P_j(\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})$

$\Rightarrow P_j(sl) \in P_j(\mathcal{L}_{m_j}) \cap P_j(\mathcal{I}_{m_j})$, by **(A.35)**, and *Proposition 2(b)*.

We next note that by **(A.38)**, we have $s \in P_j(\mathcal{L}_j) \subseteq \Sigma'^*(j)$.

We thus have $P_j(sl) = sl$ as $\Sigma_{L_j}^* \subseteq \Sigma'^*(j)$, and by definition of P_j.

$\Rightarrow sl \in P_j(\mathcal{L}_{m_j}) \cap P_j(\mathcal{I}_{m_j})$, as required.

Part (II) complete.

By **Parts (I)** and **(II)**, we have **(A.36)** $\Leftrightarrow$ **(A.34)**, as required.

We have now shown that all six points of the two definitions are equivalent, and we

can thus conclude that the parallel system satisfies Definition 3.4.1 *iff* it satisfies Definition 4.2.3.

□

4.3 Level-wise Nonblocking

In the original level-wise nonblocking definition, we first defined the *serial level-wise nonblocking* definition for the serial system consisting of DES $\mathbf{G}'_H$, $\mathbf{G}_L$, and $\mathbf{G}_I$. We then used the concept of serial system extractions (Definition 4.2.2) to extend the serial definition to the parallel case.

We now restate the serial level-wise nonblocking definition below. It's clear that for $n = 1$ and after appropriate relabeling, the level-wise nonblocking definition (Definition 3.5.1) reduces to the serial level-wise nonblocking definition; thus any result (such as in [34]) using the serial level-wise nonblocking definition would be immediately satisfied by Definition 3.5.1, with $n = 1$.

Definition 4.3.1 *The system composed of DES G'_H, G_L, and G_I, is said to be* serial level-wise nonblocking *with respect to the alphabet partition $\Sigma' := \Sigma'_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$, if the following conditions are satisfied:*

(I) $\overline{\mathcal{H}'_m \cap \mathcal{I}_m} = \mathcal{H}' \cap \mathcal{I}$

(II) $\overline{\mathcal{L}_m \cap \mathcal{I}_m} = \mathcal{L} \cap \mathcal{I}$

We now define the original level-wise nonblocking definition by extending the serial level-wise nonblocking definition to the parallel case, using Definition 4.2.2.

Definition 4.3.2 *The n^{th} degree ($n \geq 1$) parallel interface system composed of DES G_H, $G_{L_1}, \ldots, G_{L_n}, G_{I_1}, \ldots, G_{I_n}$, is* level-wise nonblocking (ORIG) *with respect to the alphabet partition given by (3.1), if for all $j \in \{1, \ldots, n\}$, the j^{th} serial system extraction of the system is serial level-wise nonblocking.*

Before we can prove the equivalence of our new level-wise nonblocking definition, we first need a useful corollary. In Section 4.2, we made extensive use of *Proposition 23* in [34]. However, this proposition required that the parallel system be interface consistent (ORIG). However, for the parts of *Proposition 23* in [34] we need for our next theorem, a weaker condition is sufficient. The restriction on the alphabet of the DES belonging to the parallel system that we use has always been implicit in the level-wise nonblocking definitions; we are only make it explicit so we can use it in the corollary below.

Corollary 1 *If the* n^{th} *degree* ($n \geq 1$) *parallel interface system composed of DES* $G_H, G_{L_1}, \ldots, G_{L_n}, G_{I_1}, \ldots, G_{I_n}$ *with alphabet partition given by (3.1), is as defined in Section 3.3 with respect to the alphabets of the given DES, then for the* j^{th} *serial system extraction,* system(j)*, the following is true:*

(i) The following event sets are: $\Sigma_I(j) = \Sigma_{I_j}$, $\Sigma_{IH'}(j) = \Sigma_{IH}$, *and* $\Sigma_{IL}(j) = \Sigma_{IL_j}$

(ii) The following inverse natural projections are: $P_{IH'}(j)^{-1} = P_j \cdot P_{IH}^{-1}$, $P_{IL}(j)^{-1} = P_j \cdot P_{IL_j}^{-1}$, *and* $P_I(j)^{-1} = P_j \cdot P_{I_j}^{-1}$

(iii) The indicated languages satisfy the following statements:

$$\begin{aligned}
\mathcal{H}'(j) &= P_j(\mathcal{H}) \cap [\cap_{k \in \{1,\ldots,(j-1),(j+1),\ldots,n\}} P_j(\mathcal{I}_k)] \\
\mathcal{H}'_m(j) &= P_j(\mathcal{H}_m) \cap [\cap_{k \in \{1,\ldots,(j-1),(j+1),\ldots,n\}} P_j(\mathcal{I}_{m_k})] \\
\mathcal{L}(j) &= P_j(\mathcal{L}_j) \\
\mathcal{L}_m(j) &= P_j(\mathcal{L}_{m_j}) \\
\mathcal{I}(j) &= P_j(\mathcal{I}_j) \\
\mathcal{I}_m(j) &= P_j(\mathcal{I}_{m_j})
\end{aligned}$$

Proof Results follow immediately from the proofs of the corresponding parts of *Proposition 23* in [34].

□

We conclude this section by presenting our theorem that shows that our new definition of level-wise nonblocking is equivalent to the original.

Theorem 4 *The n^{th} degree ($n \geq 1$) parallel interface system composed of DES $\mathbf{G}_H, \mathbf{G}_{L_1}, \ldots, \mathbf{G}_{L_n}, \mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$ as defined in Section 3.3 with respect to the alphabets of the given DES, is level-wise nonblocking (Definition 3.5.1) with respect to alphabet partition given by (3.1), iff, the system is level-wise nonblocking (ORIG) (Definition 4.3.2) with respect to alphabet partition given by (3.1).*

Proof

Assume that the n^{th} degree ($n \geq 1$) parallel interface system composed of DES $\mathbf{G}_H, \mathbf{G}_{L_1}, \ldots, \mathbf{G}_{L_n}, \mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$ is defined as in Section 3.3 with respect to the alphabets of the given DES. **(A.1)**

We start by converting Definition 4.3.2 into a more useful form.

If Definition 4.3.2 is satisfied, then for all $j \in \{1, \ldots, n\}$, the j^{th} serial system extraction (subsystem form), denoted by *system*(j), is serial level-wise nonblocking.

We thus have Definition 4.3.2 is equivalent to:

($\forall j \in \{1, \ldots, n\}$), *system*($j$) satisfies: **(A.2)**

(I) $\overline{\mathcal{H}'_m(j) \cap \mathcal{I}_m}(j) = \mathcal{H}'(j) \cap \mathcal{I}(j)$

(II) $\overline{\mathcal{L}_m(j) \cap \mathcal{I}_m}(j) = \mathcal{L}(j) \cap \mathcal{I}(j)$

We can now apply *Corollary 1* and conclude that $\mathcal{H}'(j) = P_j(\mathcal{H}) \cap [\cap_{k \in \{1, \ldots, (j-1), (j+1), \ldots, n\}} P_j(\mathcal{I}_k)]$, $\mathcal{H}'_m(j) = P_j(\mathcal{H}_m) \cap [\cap_{k \in \{1, \ldots, (j-1), (j+1), \ldots, n\}} P_j(\mathcal{I}_{m_k})]$, $\mathcal{L}(j) = P_j(\mathcal{L}_j)$, $\mathcal{L}_m(j) = P_j(\mathcal{L}_{m_j})$, $\mathcal{I}(j) = P_j(\mathcal{I}_j)$, and $\mathcal{I}_m(j) = P_j(\mathcal{I}_{m_j})$.

Substituting into **(A.2)** and simplifying, we find that Definition 4.3.2 is equivalent to: **(A.3)**

(I) for all $j \in \{1,\ldots,n\}$, $\overline{P_j(\mathcal{H}_m) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_{m_k})} = P_j(\mathcal{H}) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_k)$

(II) for all $j \in \{1,\ldots,n\}$, $\overline{P_j(\mathcal{L}_{m_j}) \cap P_j(\mathcal{I}_{m_j})} = P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j)$

In order to prove the system satisfies Definition 4.3.2 *iff* it satisfies Definition 3.5.1, it is thus sufficient to show **(A.3)** *iff* Definition 3.5.1.

As **(A.3)** and Definition 3.5.1 are of the same form, we will prove equivalence point by point.

We first note that for all $j \in \{1,\ldots,n\}$, the languages $\mathcal{H}$, $\mathcal{H}_m$, $\mathcal{I}_k$, $\mathcal{I}_{m_k}(k \in \{1\ldots n\})$, $\mathcal{L}_j$, and $\mathcal{L}_{m_j}$ are P_j-invariant, by *Proposition 1.* **(A.4)**

(I) For **Point I**, we need to show:

(A.3) $\forall j \in \{1,\ldots,n\}, \overline{P_j(\mathcal{H}_m) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_{m_k})} = P_j(\mathcal{H}) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_k)$

$\Leftrightarrow$ **(A.5)**

(Definition 3.5.1) $\overline{\mathcal{H}_m \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_{m_k}} = \mathcal{H} \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_k$ **(A.6)**

To do this, we need to show: **(I.a)** **(A.5)** $\Rightarrow$ **(A.6)** and **(I.b)** **(A.6)** $\Rightarrow$ **(A.5)**.

(I.a) Show **(A.5)** $\Rightarrow$ **(A.6)**.

Assume **(A.5)**. Must show implies **(A.6)**.

As $\overline{\mathcal{H}_m \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_{m_k}} \subseteq \mathcal{H} \cap \bigcap_{k\in\{1\ldots n\}} \mathcal{I}_k$ is automatic, we only need to show:
$\mathcal{H} \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_k \subseteq \overline{\mathcal{H}_m \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_{m_k}}$.

Let $s \in \mathcal{H} \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_k$ **(A.7)**

Must show implies $s \in \overline{\mathcal{H}_m \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_{m_k}}$.

Assume $j \in \{1,\ldots,n\}$. From **(A.7)**, we can conclude:
$P_j(s) \in P_j(\mathcal{H} \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_k)$

By **(A.4)** and *Proposition 2(b)*, we have:

$P_j(s) \in P_j(\mathcal{H}) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_k)$

By **(A.5)**, we can conclude:

$P_j(s) \in \overline{P_j(\mathcal{H}_m) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_{m_k})}$

$\Rightarrow (\exists s' \in \Sigma'(j)^*)\, P_j(s)s' \in P_j(\mathcal{H}_m) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_{m_k})$

As $s' \in \Sigma'(j)^*$, we thus have $P_j(s') = s'$, by definition of P_j.

$\Rightarrow P_j(ss') \in P_j(\mathcal{H}_m) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_{m_k})$, as P_j is catenative.

By **(A.4)** and *Proposition 2(b)*, we have:

$P_j(ss') \in P_j(\mathcal{H}_m \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_{m_k})$

$\Rightarrow ss' \in \mathcal{H}_m \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_{m_k}$, by **(A.4)**, and *Proposition 2(c),(d)*.

$\Rightarrow s \in \overline{\mathcal{H}_m \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_{m_k}}$, as required.

Part (I.a) complete.

(I.b) Show **(A.6)** $\Rightarrow$ **(A.5)**.

Assume **(A.6)**. Must show implies **(A.5)**.

Let $j \in \{1, \ldots, n\}$. As $\overline{P_j(\mathcal{H}_m) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_{m_k})} \subseteq P_j(\mathcal{H}) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_k)$ is automatic, we only need to show:

$P_j(\mathcal{H}) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_k) \subseteq \overline{P_j(\mathcal{H}_m) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_{m_k})}$

Let $s \in P_j(\mathcal{H}) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_k)$ **(A.8)**

Must show implies $s \in \overline{P_j(\mathcal{H}_m) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_{m_k})}$.

From **(A.4)**, **(A.8)**, and *Proposition 2(b)*, we can conclude:

$s \in P_j(\mathcal{H} \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_k)$

$\Rightarrow s \in \mathcal{H} \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_k$, by **(A.4)**, and *Proposition 2(c),(a).*

By **(A.6)**, we can conclude:
$s \in \overline{\mathcal{H}_m \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_{m_k}}$

$\Rightarrow (\exists s' \in \Sigma^*)\, ss' \in \mathcal{H}_m \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_{m_k}$

$\Rightarrow P_j(ss') \in P_j(\mathcal{H}_m \cap \bigcap_{k=1,\ldots,n} \mathcal{I}_{m_k})$

$\Rightarrow P_j(ss') \in P_j(\mathcal{H}_m) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_{m_k})$, by **(A.4)**, and *Proposition 2(b).*

$\Rightarrow P_j(s)P_j(s') \in P_j(\mathcal{H}_m) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_{m_k})$, as P_j is catenative.

As $s \in P_j(\mathcal{H}) \subseteq \Sigma'(j)^*$ (by **(A.8)**), we have $P_j(s) = s$ (by definition of P_j).

$\Rightarrow sP_j(s') \in P_j(\mathcal{H}_m) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_{m_k})$

$s \in \overline{P_j(\mathcal{H}_m) \cap \bigcap_{k=1,\ldots,n} P_j(\mathcal{I}_{m_k})}$, as required.

Part (I.b) complete.

By **Parts (I.a)** and **(I.b)**, we have **(A.5)** $\Leftrightarrow$ **(A.6)**, as required.

(II) For **Point II**, and letting $j \in \{1, \ldots, n\}$, we need to show:

(A.3) $\overline{P_j(\mathcal{L}_{m_j}) \cap P_j(\mathcal{I}_{m_j})} = P_j(\mathcal{L}_j) \cap P_j(\mathcal{I}_j)$ $\Leftrightarrow$ **(A.9)**

(Definition 3.5.1) $\overline{\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j}} = \mathcal{L}_j \cap \mathcal{I}_j$ **(A.10)**

The proof here is identical to **Part I**, after appropriate relabelling.

By **Parts I** and **II**, we can conclude **(A.3)** *iff* Definition 3.5.1, as required.

□

4.4 Level-wise Controllability

In the original level-wise controllability definition, we first defined the *serial level-wise controllability* definition for the serial system consisting of plant components $\mathbf{G}_H^{p\,\prime}$, $\mathbf{G}_L^p$, supervisors $\mathbf{S}'_H$, $\mathbf{S}_L$, and interface G_I. We then used the concept of serial system extractions (Definition 4.4.2 below) to extend the serial definition to the parallel case.

We assume that the alphabet partition for a serial system is specified by $\Sigma' := \Sigma'_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$, and then we introduce the following useful languages:

$$\begin{aligned} \mathcal{H}^{p\prime} &:= P_{IH}^{-1} L(\mathbf{G}_H^{p\,\prime}), \qquad \mathcal{S}'_H := P_{IH}^{-1} L(\mathbf{S}'_H), \qquad \subseteq \Sigma^* \\ \mathcal{L}^p &:= P_{IL}^{-1} L(\mathcal{G}_L), \qquad \mathcal{S}_L := P_{IL}^{-1} L(\mathcal{S}_L), \qquad \subseteq \Sigma^* \end{aligned}$$

We now restate the serial level-wise controllability definition below. It's clear that for $n = 1$ and after appropriate relabelling, the level-wise controllability definition (Definition 3.6.1) reduces to the serial level-wise controllability definition; thus any result (such as in [34]) using the serial level-wise controllability definition would be immediately satisfied by Definition 3.6.1, with $n = 1$.

Definition 4.4.1 *The system composed of plant components $\mathbf{G}_H^{p\,\prime}$, $\mathbf{G}_L^p$, supervisors $\mathbf{S}'_H$, $\mathbf{S}_L$, and interface $\mathbf{G}_I$, is said to be* serial level-wise controllable *with respect to the alphabet partition $\Sigma' := \Sigma'_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$, if the following conditions are satisfied:*

(I) The alphabet of $\mathbf{G}_H^{p\,\prime}$ and $\mathbf{S}'_H$ is Σ_{IH}, the alphabet of $\mathbf{G}_L^p$ and $\mathbf{S}_L$ is Σ_{IL}, and the alphabet of $\mathbf{G}_I$ is Σ_I.

(II) $(\forall s \in \mathcal{L}^p \cap \mathcal{S}_L \cap \mathcal{I})\ Elig_{\mathcal{L}^p}(s) \cap \Sigma_u \subseteq Elig_{\mathcal{S}_L \cap \mathcal{I}}(s)$

(III) $(\forall s \in \mathcal{H}^{p\prime} \cap \mathcal{I} \cap \mathcal{S}'_H)\ Elig_{\mathcal{H}^{p\prime} \cap \mathcal{I}}(s) \cap \Sigma_u \subseteq Elig_{\mathcal{S}'_H}(s)$

We now restate the general form of the serial system extraction needed for the controllability definition. We simply refer to the j^{th} serial system extraction, as the type of the parallel system (general form or subsystem form) will make clear which definition is intended.

Definition 4.4.2 *For the n^{th} degree ($n \geq 1$) parallel interface system composed of DES $\mathbf{G}_H^p$, $\mathbf{G}_{L_1}^p, \ldots, \mathbf{G}_{L_n}^p$, $\mathbf{S}_H$, $\mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$, with alphabet partition given by (3.1), the j^{th}* serial system extraction *(general form), denoted by* system(j)*, is composed of the following elements:*

$$\begin{aligned}
\mathbf{G}_H^{p\,\prime}(j) &:= \mathbf{G}_H^p||\mathbf{G}_{I_1}||\ldots||\mathbf{G}_{I_{(j-1)}}||\mathbf{G}_{I_{(j+1)}}||\ldots||\mathbf{G}_{I_n} \\
\mathbf{S}'_H(j) &:= \mathbf{S}_H, \quad \mathbf{G}_L^p(j) := \mathbf{G}_{L_j}^p, \quad \mathbf{S}_L(j) := \mathbf{S}_{L_j}, \quad \mathbf{G}_I(j) := \mathbf{G}_{I_j} \\
\Sigma'_H(j) &:= \dot{\cup}_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}}\Sigma_{I_k} \dot{\cup} \Sigma_H \\
\Sigma_L(j) &:= \Sigma_{L_j}, \quad \Sigma_R(j) := \Sigma_{R_j}, \quad \Sigma_A(j) := \Sigma_{A_j} \\
\Sigma'(j) &:= \Sigma'_H(j) \dot{\cup} \Sigma_L(j) \dot{\cup} \Sigma_R(j) \dot{\cup} \Sigma_A(j) \\
&= \Sigma - \dot{\cup}_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}}\Sigma_{L_k}
\end{aligned}$$

We are now ready to state the original level-wise controllability definition, for the parallel case.

Definition 4.4.3 *The n^{th} degree ($n \geq 1$) parallel interface system composed of DES $\mathbf{G}_H^p, \mathbf{G}_{L_1}^p, \ldots, \mathbf{G}_{L_n}^p$, $\mathbf{S}_H$, $\mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$, is* level-wise controllable (ORIG) *with respect to alphabet partition given by (3.1), if for all $j \in \{1, \ldots, n\}$, the j^{th} serial system extraction of the system is serial level-wise controllable.*

We now need to provide a counterpart to *Proposition 1* for languages of a general form system. For $j \in \{1, \ldots, n\}$, the proposition below essentially states that the indicated languages are P_j-invariant.

Proposition 5 *With $\mathcal{H}^p$, $\mathcal{S}_H$, $\mathcal{L}_j^p$, and $\mathcal{S}_{L_j}$ as defined in Section 3.6, we have:*

(a) $P_j^{-1}(P_j(\mathcal{H}^p)) = \mathcal{H}^p$

(b) $P_j^{-1}(P_j(\mathcal{S}_H) = \mathcal{S}_H$

(c) $P_j^{-1}(P_j(\mathcal{L}_j^p)) = \mathcal{L}_j^p$

(d) $P_j^{-1}(P_j(\mathcal{S}_{L_j})) = \mathcal{S}_{L_j}$

Proof

Point (a)-(d) Proofs are identical to **Point (a)** of *Proposition 1* after appropriate substitutions.

□

We conclude this section by presenting our theorem that shows that our new definition of level-wise controllability is equivalent to the original.

Theorem 5 *The n^{th} degree ($n \geq 1$) parallel interface system composed of DES $\mathbf{G}_H^p, \mathbf{G}_{L_1}^p, \ldots, \mathbf{G}_{L_n}^p$, $\mathbf{S}_H, \mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$, is level-wise controllable (Definition 3.6.1) with respect to alphabet partition given by (3.1), iff, the system is level-wise controllable (ORIG) (Definition 4.4.3) with respect to alphabet partition given by (3.1).*

Proof

We start by converting Definition 4.4.3 into a more useful form.

If Definition 4.4.3 is satisfied, then for all $j \in \{1, \ldots, n\}$, the j^{th} serial system extraction (general form), denoted by *system*(j), is serial level-wise controllable.

We thus have Definition 4.4.3 is equivalent to:

$(\forall j \in \{1, \ldots, n\})$, *system*$(j)$ satisfies: **(A.1)**

(I) The alphabet of $\mathbf{G}_H^{p\,\prime}(j)$ and $\mathbf{S}'_H(j)$ is $\Sigma_{IH}(j)$, the alphabet of $\mathbf{G}_L^p(j)$ and $\mathbf{S}_L(j)$ is $\Sigma_{IL}(j)$, and the alphabet of $\mathbf{G}_I(j)$ is $\Sigma_I(j)$.

(II) $(\forall s \in \mathcal{L}^p(j) \cap \mathcal{S}_L(j) \cap \mathcal{I}(j))\ \mathrm{Elig}_{\mathcal{L}^p(j)}(s) \cap \Sigma_u \subseteq \mathrm{Elig}_{\mathcal{S}_L(j)\cap\mathcal{I}(j)}(s)$

(III) $(\forall s \in \mathcal{H}^{p\prime}(j) \cap \mathcal{I}(j) \cap \mathcal{S}'_H(j))\ \mathrm{Elig}_{\mathcal{H}^{p\prime}(j)\cap\mathcal{I}(j)}(s) \cap \Sigma_u \subseteq \mathrm{Elig}_{\mathcal{S}'_H(j)}(s)$

It is thus sufficient to show that system satisfies **(A.1)** *iff* it satisfies Definition 3.6.1.

As **(A.1)** and Definition 3.6.1 are of the same form, we will prove equivalence point by point.

(I) For **Point I**, we must show:

(A.1) $(\forall j \in \{1, \ldots, n\})$ the alphabet of $\mathbf{G}_H^{p\prime}(j)$ and $\mathbf{S}'_H(j)$ is $\Sigma_{IH}(j)$, the alphabet of $\mathbf{G}_L^p(j)$ and $\mathbf{S}_L(j)$ is $\Sigma_{IL}(j)$, and the alphabet of $\mathbf{G}_I(j)$ is $\Sigma_I(j)$ **(A.2)**

$\Leftrightarrow$

(Defn. 3.6.1) $(\forall j \in \{1, \ldots, n\})$ 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} **(A.3)**

(I.a) Show **(A.2)** $\Rightarrow$ **(A.3)**

Assume **(A.2)**. Must show implies **(A.3)**.

This follows immediately from *Proposition 28* of [34].

(I.b) Show **(A.3)** $\Rightarrow$ **(A.2)**

Assume **(A.3)**. Must show implies **(A.2)**.

By definition, $\mathbf{G}_H^{p\prime}(j) = \mathbf{G}_H^p||\mathbf{G}_{I_1}||\ldots||\mathbf{G}_{I_{(j-1)}}||\mathbf{G}_{I_{(j+1)}}||\ldots||\mathbf{G}_{I_n}$, $\mathbf{S}'_H(j) = \mathbf{S}_H$, $\mathbf{G}_L^p(j) = \mathbf{G}_{L_j}^p$, $\mathbf{S}_L(j) = \mathbf{S}_{L_j}$, and $\mathbf{G}_I(j) := \mathbf{G}_{I_j}$. **(A.4)**

From *Proposition 30* of [34], we can conclude:

$\Sigma_{IH}(j) = \Sigma_{IH}$, $\Sigma_{IL}(j) = \Sigma_{IL_j}$, and $\Sigma_I(j) = \Sigma_{I_j}$.

Combining with **(A.4)** and substituting into **(A.2)**, we can conclude that it is sufficient to show that:

$(\forall j \in \{1,\ldots,n\})$ the alphabet of $\mathbf{G}_H^p||\mathbf{G}_{I_1}||\ldots||\mathbf{G}_{I_{(j-1)}}||\mathbf{G}_{I_{(j+1)}}||\ldots||\mathbf{G}_{I_n}$ 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} **(A.5)**

All of this follows immediately from **(A.3)**, except showing that the alphabet of $\mathbf{G}_H^{p\,\prime}(j) = \mathbf{G}_H^p||\mathbf{G}_{I_1}||\ \ldots||\mathbf{G}_{I_{(j-1)}}||\mathbf{G}_{I_{(j+1)}}||\ldots||\mathbf{G}_{I_n}$ is Σ_{IH}.

From the definition of the synchronous product and **(A.3)**, we can conclude that the alphabet of $\mathbf{G}_H^{p\,\prime}(j)$ is:

$\Sigma_{\mathbf{G}_H^{p\,\prime}(j)} = \cup_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}}\Sigma_{I_k}\ \cup \Sigma_{IH}\ =\ \Sigma_{IH}$

Part I.b complete.

By **Parts I.a** and **I.b**, we can conclude **(A.2)** $\Leftrightarrow$ **(A.3)**, as required.

(II) For **Point II**, and $j \in \{1,\ldots,n\}$, we must show:

(A.1) $(\forall s \in \mathcal{L}^p(j) \cap \mathcal{S}_L(j) \cap \mathcal{I}(j))\ \mathrm{Elig}_{\mathcal{L}^p(j)}(s) \cap \Sigma_u\ \subseteq\ \mathrm{Elig}_{\mathcal{S}_L(j)\cap\mathcal{I}(j)}(s)$ $\Leftrightarrow$ **(A.6)**

(Defn. 3.6.1) $(\forall s \in \mathcal{L}_j^p \cap \mathcal{S}_{L_j} \cap \mathcal{I}_j)\ \mathrm{Elig}_{\mathcal{L}_j^p}(s) \cap \Sigma_u\ \subseteq\ \mathrm{Elig}_{\mathcal{S}_{L_j}\cap\mathcal{I}_j}(s)$ **(A.7)**

We start by converting **(A.6)** into a more useful form.

From *Proposition 30* of [34], we can conclude:

$\mathcal{L}^p(j) = P_j(\mathcal{L}_j^p),\ \ \mathcal{S}_L(j) = P_j(\mathcal{S}_{L_j}),\ \ \mathcal{I}(j) = P_j(\mathcal{I}_j)$

Substituting into **(A.6)**, we find that **(A.6)** is equivalent to:

$(\forall s \in P_j(\mathcal{L}_j^p) \cap P_j(\mathcal{S}_{L_j}) \cap P_j(\mathcal{I}_j))\ \mathrm{Elig}_{P_j(\mathcal{L}_j^p)}(s)\cap\Sigma_u\ \subseteq\ \mathrm{Elig}_{P_j(\mathcal{S}_{L_j})\cap P_j(\mathcal{I}_j)}(s)$ **(A.8)**

It is thus sufficient to show **(A.8)** $\Leftrightarrow$ **(A.7)**.

We first note that, by *Propositions 1* and *5*, languages $\mathcal{L}_j^p$, $\mathcal{S}_{L_j}$, and $\mathcal{I}_j$ are P_j-invariant. **(A.9)**

$\Rightarrow \mathcal{L}_j^p \cap \mathcal{S}_{L_j} \cap \mathcal{I}_j$ and $\mathcal{S}_{L_j} \cap \mathcal{I}_j$ are P_j-invariant, by *Proposition 2(c)*. **(A.10)**

By **(A.9)** and *Proposition 2(b)*, we can also conclude:

$P_j(\mathcal{L}_j^p \cap \mathcal{S}_{L_j} \cap \mathcal{I}_j) = P_j(\mathcal{L}_j^p) \cap P_j(\mathcal{S}_{L_j}) \cap P_j(\mathcal{I}_j)$ and $P_j(\mathcal{S}_{L_j} \cap \mathcal{I}_j) = P_j(\mathcal{S}_{L_j}) \cap P_j(\mathcal{I}_j)$ **(A.11)**

We next note that **(A.7)** and **(A.8)** are almost in the correct form to apply *Proposition 3*. The only problem is that Σ_u is not necessarily a subset of $\Sigma'(j)$.

Claim 1:

$(\forall s \in P_j(\mathcal{L}_j^p) \cap P_j(\mathcal{S}_{L_j}) \cap P_j(\mathcal{I}_j))$ (†)

$\text{Elig}_{P_j(\mathcal{L}_j^p)}(s) \cap \Sigma_u \subseteq \text{Elig}_{P_j(\mathcal{S}_{L_j}) \cap P_j(\mathcal{I}_j)}(s) \Leftrightarrow \text{Elig}_{P_j(\mathcal{L}_j^p)}(s) \cap (\Sigma_u \cap \Sigma'(j)) \subseteq \text{Elig}_{P_j(\mathcal{S}_{L_j}) \cap P_j(\mathcal{I}_j)}(s)$

Let $s \in P_j(\mathcal{L}_j^p) \cap P_j(\mathcal{S}_{L_j}) \cap P_j(\mathcal{I}_j)$

We first note that $P_j(\mathcal{L}_j^p) \subseteq \Sigma'(j)$, by definition of P_j.

$\Rightarrow \text{Elig}_{P_j(\mathcal{L}_j^p)}(s) \subseteq \Sigma'(j)$, by definition of the Elig operator.

$\Rightarrow \text{Elig}_{P_j(\mathcal{L}_j^p)}(s) \cap \Sigma'(j) = \text{Elig}_{P_j(\mathcal{L}_j^p)}(s)$

The result follows immediately, thus (*dagger*) holds. **Claim 1** complete.

Claim 2:

$(\forall s \in \mathcal{L}_j^p \cap \mathcal{S}_{L_j} \cap \mathcal{I}_j)$ (‡)

$\text{Elig}_{\mathcal{L}_j^p}(s) \cap \Sigma_u \subseteq \text{Elig}_{\mathcal{S}_{L_j} \cap \mathcal{I}_j}(s) \Leftrightarrow \text{Elig}_{\mathcal{L}_j^p}(s) \cap (\Sigma_u \cap \Sigma'(j)) \subseteq \text{Elig}_{\mathcal{S}_{L_j} \cap \mathcal{I}_j}(s)$

Let $s \in \mathcal{L}_j^p \cap \mathcal{S}_{L_j} \cap \mathcal{I}_j$ **(A.12)**

(2.a) Show $\text{Elig}_{\mathcal{L}_j^p}(s) \cap \Sigma_u \subseteq \text{Elig}_{\mathcal{S}_{L_j} \cap \mathcal{I}_j}(s) \Rightarrow \text{Elig}_{\mathcal{L}_j^p}(s) \cap (\Sigma_u \cap \Sigma'(j)) \subseteq \text{Elig}_{\mathcal{S}_{L_j} \cap \mathcal{I}_j}(s)$

This is automatic as $\text{Elig}_{\mathcal{L}_j^p}(s) \cap (\Sigma_u \cap \Sigma'(j)) \subseteq \text{Elig}_{\mathcal{L}_j^p}(s) \cap \Sigma_u$

(2.b) Show $\text{Elig}_{\mathcal{L}_j^p}(s) \cap (\Sigma_u \cap \Sigma'(j)) \subseteq \text{Elig}_{\mathcal{S}_{L_j} \cap \mathcal{I}_j}(s) \Rightarrow \text{Elig}_{\mathcal{L}_j^p}(s) \cap \Sigma_u \subseteq \text{Elig}_{\mathcal{S}_{L_j} \cap \mathcal{I}_j}(s)$

Assume $\text{Elig}_{\mathcal{L}_j^p}(s) \cap (\Sigma_u \cap \Sigma'(j)) \subseteq \text{Elig}_{\mathcal{S}_{L_j} \cap \mathcal{I}_j}(s)$ **(A.13)**

Let $\sigma \in \text{Elig}_{\mathcal{L}_j^p}(s) \cap \Sigma_u$. **(A.14)**

Must show that implies $\sigma \in \text{Elig}_{\mathcal{S}_{L_j} \cap \mathcal{I}_j}(s)$.

Sufficient to show that $s\sigma \in \mathcal{S}_{L_j} \cap \mathcal{I}_j$, by definition of Elig operator.

From **(A.14)**, we have two possibilities: $\sigma \in \Sigma'(j) \cap \Sigma_u$ or $\sigma \in \Sigma_u - \Sigma'(j)$

For case $\sigma \in \Sigma'(j) \cap \Sigma_u$, the results follow immediately from **(A.13)**.

For case $\sigma \in \Sigma_u - \Sigma'(j)$, we start by noting that this implies that $P_j(\sigma) = \epsilon$, by definition of P_j **(A.15)**

We next note that $s \in \mathcal{S}_{L_j} \cap \mathcal{I}_j$ (by **(A.12)**) implies that:
$P_j(s) \in P_j(\mathcal{S}_{L_j} \cap \mathcal{I}_j)$

$\Rightarrow P_j(s)P_j(\sigma) \in P_j(\mathcal{S}_{L_j} \cap \mathcal{I}_j)$, by **(A.15)**.

$\Rightarrow P_j(s\sigma) \in P_j(\mathcal{S}_{L_j} \cap \mathcal{I}_j)$, as P_j is catenative.

$\Rightarrow s\sigma \in \mathcal{S}_{L_j} \cap \mathcal{I}_j$, by **(A.10)** and *Proposition 2(d).*

Part 2.b complete.

By **Parts 2.a** and **2.b**, we can conclude that (‡) holds. **Claim 2** complete.

By **(A.11)**, **Claims 1** and **2**, we see that to prove **(A.8)** $\Leftrightarrow$ **(A.7)**, it is sufficient to prove:

$(\forall s \in P_j(\mathcal{L}_j^p \cap \mathcal{S}_{L_j} \cap \mathcal{I}_j))\text{Elig}_{P_j(\mathcal{L}_j^p)}(s) \cap (\Sigma_u \cap \Sigma'(j)) \subseteq \text{Elig}_{P_j(\mathcal{S}_{L_j} \cap \mathcal{I}_j)}(s) \quad \Leftrightarrow$ **(A.16)**

$(\forall s \in \mathcal{L}_j^p \cap \mathcal{S}_{L_j} \cap \mathcal{I}_j)\ \text{Elig}_{\mathcal{L}_j^p}(s) \cap (\Sigma_u \cap \Sigma'(j)) \subseteq \text{Elig}_{\mathcal{S}_{L_j} \cap \mathcal{I}_j}(s)$ **(A.17)**

We can now take $\Sigma_a = \Sigma'(j)$, $\Sigma_b = \Sigma_u \cap \Sigma'(j)$, $P = P_j$, $L_1 = \mathcal{L}_j^p \cap \mathcal{S}_{L_j} \cap \mathcal{I}_j$, $L_2 = \mathcal{L}_j^p$, $L_3 = \mathcal{S}_{L_j} \cap \mathcal{I}_j$, and conclude by **(A.9)**, **(A.10)**, and *Proposition 3* that:
(A.16) $\Leftrightarrow$ **(A.17)**

Part II complete.

(III) For **Point III**, we must show:

(A.1) $(\forall s \in \mathcal{H}^{p\prime}(j) \cap \mathcal{I}(j) \cap \mathcal{S}'_H(j))\ \text{Elig}_{\mathcal{H}^{p\prime}(j) \cap \mathcal{I}(j)}(s) \cap \Sigma_u \subseteq \text{Elig}_{\mathcal{S}'_H(j)}(s) \quad \Leftrightarrow$ **(A.18)**

(Defn. 3.6.1) $(\forall s \in \mathcal{H}^p \cap [\cap_{k\in\{1,\ldots,n\}}\mathcal{I}_k] \cap \mathcal{S}_H)\,\text{Elig}_{\mathcal{H}^p \cap [\cap_{k\in\{1,\ldots,n\}}\mathcal{I}_k]}(s) \cap \Sigma_u \subseteq \text{Elig}_{\mathcal{S}_H}(s)$ **(A.19)**

We start by converting **(A.18)** into a more useful form.

From *Proposition 30* of [34], we can conclude:
$\mathcal{H}^{p\prime}(j) = P_j(\mathcal{H}^p) \cap [\cap_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}} P_j(\mathcal{I}_k)]$, $\mathcal{S}'_H(j) = P_j(\mathcal{S}_H)$, $\mathcal{I}(j) = P_j(\mathcal{I}_j)$

Substituting into **(A.18)**, we find that **(A.18)** is equivalent to:

$(\forall s \in P_j(\mathcal{H}^p) \cap [\cap_{k\in\{1,\ldots,n\}} P_j(\mathcal{I}_k)] \cap P_j(\mathcal{S}_H))$
$\text{Elig}_{P_j(\mathcal{H}^p) \cap [\cap_{k\in\{1,\ldots,n\}} P_j(\mathcal{I}_k)]}(s) \cap \Sigma_u \subseteq \text{Elig}_{P_j(\mathcal{S}_H)}(s)$ **(A.20)**

It is thus sufficient to show **(A.20)** $\Leftrightarrow$ **(A.19)**.

We first note that, by *Propositions 1* and *5*, languages $\mathcal{H}^p$, $\mathcal{S}_H$, and $\mathcal{I}_k$ $(k = 1, \ldots, n)$ are P_j-invariant. **(A.21)**

$\Rightarrow \mathcal{H}^p \cap [\cap_{k\in\{1,\ldots,n\}}\mathcal{I}_k] \cap \mathcal{S}_H$ and $\mathcal{H}^p \cap [\cap_{k\in\{1,\ldots,n\}}\mathcal{I}_k$ are P_j-invariant (*Proposition 2(c)*). **(A.22)**

By **(A.21)** and *Proposition 2(b)*, we can also conclude:

$P_j(\mathcal{H}^p \cap [\cap_{k\in\{1,\ldots,n\}}\mathcal{I}_k] \cap \mathcal{S}_H) = P_j(\mathcal{H}^p) \cap [\cap_{k\in\{1,\ldots,n\}} P_j(\mathcal{I}_k)] \cap P_j(\mathcal{S}_H)$ and $P_j(\mathcal{H}^p \cap [\cap_{k\in\{1,\ldots,n\}}\mathcal{I}_k)$

$= P_j(\mathcal{H}^p) \cap [\cap_{k\in\{1,\ldots,n\}} P_j(\mathcal{I}_k)]$ **(A.23)**

The remainder of the proof is identical to **Part II**, after suitable relabelling.

Part III complete.

We thus conclude by **Points I**, **II**, and **III**, that Definition 3.6.1 is equivalent to Definition 4.4.3, as required.

□

4.5 Main Nonblocking and Controllability Results

Now that we have shown that our new HISC definitions are equivalent to the original ones from [34], we can apply the results from [34] to systems that satisfy our new definitions.

We are now ready to present our nonblocking theorem for parallel interface systems. It basically states that if the system is level-wise nonblocking and interface consistent, then the flat system will be nonblocking.

Theorem 6 *If the n^{th} degree ($n \geq 1$) parallel interface system composed of DES $\mathbf{G}_H, \mathbf{G}_{I_1}, \mathbf{G}_{L_1}, \ldots, \mathbf{G}_{I_n}, \mathbf{G}_{L_n}$, is level-wise nonblocking (Definition 3.5.1) and interface consistent (Definition 3.4.1) with respect to the alphabet partition given by (3.1), then $\overline{L_m(\mathbf{G})} = L(\mathbf{G})$, where $\mathbf{G} = \mathbf{G}_H||\mathbf{G}_{I_1}||\mathbf{G}_{L_1}||\ldots||\mathbf{G}_{I_n}||\mathbf{G}_{L_n}$.*

Proof Results follow immediately from *Theorem 3* from [34], and *Theorems 3*, and *4*. □

We now present a sufficient condition for controllability of parallel interface systems. It states that if the system is level-wise controllable, then the flat supervisor is controllable for the flat plant.[3]

Theorem 7 *If the n^{th} degree ($n \geq 1$) parallel 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 controllable (Definition 3.6.1) with respect to the alphabet partition given by (3.1), then*

$$(\forall s \in L(\mathbf{Plant}) \cap L(\mathbf{Sup})) \quad Elig_{L(\mathbf{Plant})}(s) \cap \Sigma_u \subseteq Elig_{L(\mathbf{Sup})}(s)$$

Proof Results follow immediately from *Theorem 4* from [34], and *Theorem 5*. □

[3] At first glance, the controllability definition used below might seem slightly different than the one given in Section 2.3, but this can be easily reconciled by noting that for *Theorem 7*, $\Sigma_G = \Sigma_S = \Sigma$.

Chapter 5

HISC Synthesis Method

In Chapter 3, we describe a n^{th} degree ($n \geq 1$) interface system composed of plant DES, supervisor DES , and interface DES. For this system, we showed how the properties of interface consistency, level-wise nonblocking, and level-wise controllable could be used to verify that the flat system is nonblocking, and that the flat supervisor is controllable for the flat plant. However, if the system fails one of these conditions, we need a way to automatically modify the system so that it will satisfy all three of the above conditions. We need a synthesis method that will respect the HISC structure and provide a similar level of scalability.

5.1 Synthesis Setting

In Chapter 3, we referred to a system composed of plant DES $\mathbf{G}_H^p$, $\mathbf{G}_{L_1}^p, \ldots,$ $\mathbf{G}_{L_n}^p$, supervisor DES $\mathbf{S}_H$, $\mathbf{S}_{L_1}, \ldots, \mathbf{S}_{L_n}$, and interface DES $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$ as a n^{th} degree interface system. When we specify a n^{th} degree interface system and give supervisors (as opposed to specifications), we will refer to such a system as a n^{th} *degree supervisor interface system.*

For a n^{th} *degree supervisor interface system*, we assume that we are given a

supervisor for the high-level, and one for each of the n low-levels, and that we are verifying that the interface system satisfies the interface conditions. For synthesis, we will assume that we are instead given a specification for each component. Our goal will then be to synthesize a supervisor for each component that will satisfy the corresponding HISC conditions by design, and will be maximally permissive for its component.

For synthesis, we will replace supervisor $\mathbf{S}_H$ by specification DES $\mathbf{E}_H$ (defined over Σ_{IH}), and for $j \in \{1, \ldots, n\}$, we will replace supervisor $\mathbf{S}_{L_j}$ by specification DES $\mathbf{E}_{L_j}$ (defined over Σ_{IL_j}). We will refer to the system composed of plant DES $\mathbf{G}^p_H$, $\mathbf{G}^p_{L_1}, \ldots,$ $\mathbf{G}^p_{L_n}$, specification DES $\mathbf{E}_H$, $\mathbf{E}_{L_1}, \ldots, \mathbf{E}_{L_n}$, and interface DES $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$ as a n^{th} *degree specification interface system.*

As a starting point for synthesis, we need to make sure that our specification interface system meets certain basic requirements. These are portions of the HISC conditions that we will not be able to correct for as part of our synthesis procedure.

Definition 5.1.1 *The n^{th} degree specification interface system composed of plant DES $\mathbf{G}^p_H$, $\mathbf{G}^p_{L_1}, \ldots,$ $\mathbf{G}^p_{L_n}$, specification DES $\mathbf{E}_H$, $\mathbf{E}_{L_1}, \ldots, \mathbf{E}_{L_n}$, and interface DES $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$ is* HISC-valid *with respect to alphabet partition given by (3.1), if for all $j \in \{1, \ldots, n\}$, the following conditions are satisfied:*

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

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

For the rest of this chapter, we will use Φ to stand for the n^{th} degree HISC-valid specification interface system that respects the alphabet partition given by (3.1) and is composed of plant DES $\mathbf{G}^p_H$, $\mathbf{G}^p_{L_1}, \ldots,$ $\mathbf{G}^p_{L_n}$, specification DES $\mathbf{E}_H$, $\mathbf{E}_{L_1}, \ldots, \mathbf{E}_{L_n}$, and interface DES $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$, that we are considering. We will also take j to be an index in the range $\{1, \ldots, n\}$.

In Chapter 3, we introduced the languages $\mathcal{H}^p$, $\mathcal{L}^p_j$, $\mathcal{I}_j$, and $\mathcal{I}_{m_j}$. We now introduce a few more useful languages.

$$\mathcal{H}^p_m = P^{-1}_{IH}(L_m(\mathbf{G}^p_H))$$
$$\mathcal{E}_H = P^{-1}_{IH}(L(\mathbf{E}_H))$$
$$\mathcal{E}_{H_m} = P^{-1}_{IH}(L_m(\mathbf{E}_H))$$
$$\mathcal{L}^p_{m_j} = P^{-1}_{IL_j}(L_m(\mathbf{G}^p_{L_j}))$$
$$\mathcal{E}_{L_j} = P^{-1}_{IL_j}(L(\mathbf{E}_{L_j}))$$
$$\mathcal{E}_{L_{j,m}} = P^{-1}_{IL_j}(L_m(\mathbf{E}_{L_j}))$$

To simplify proofs in the following chapters, we define languages[1]

$$\mathcal{I} = \bigcap_{k\in\{1,\ldots,n\}} \mathcal{I}_k$$
$$\mathcal{I}_m = \bigcap_{k\in\{1,\ldots,n\}} \mathcal{I}_{m_k}$$

We will refer to the DES that represents the high level of Φ as:

$$\mathbf{G}_{\text{HL}} = \mathbf{G}^p_H||\mathbf{E}_H||\mathbf{G}_{I_1}||\ldots||\mathbf{G}_{I_n}$$

We can now define the languages for $\mathbf{G}_{\text{HL}}$ over Σ^* as follows:

$$\mathcal{Z}_H = P^{-1}_{IH}(L(\mathbf{G}_{\text{HL}})) = \mathcal{H}^p \cap \mathcal{E}_H \cap \mathcal{I}$$
$$\mathcal{Z}_{H_m} = P^{-1}_{IH}(L_m(\mathbf{G}_{\text{HL}})) = \mathcal{H}^p_m \cap \mathcal{E}_{H_m} \cap \mathcal{I}_m$$

We will refer to the DES that represents the j^{th} low level of Φ as:

$$\mathbf{G}_{\text{LL}_j} = \mathbf{G}^p_{L_j}||\mathbf{E}_{L_j}||\mathbf{G}_{I_j}$$

We can now define the languages for $\mathbf{G}_{\text{LL}_j}$ over Σ^* as follows:

$$\mathcal{Z}_{L_j} = P^{-1}_{IL_j}(L(\mathbf{G}_{\text{LL}_j})) = \mathcal{L}^p_j \cap \mathcal{E}_{L_j} \cap \mathcal{I}_j$$
$$\mathcal{Z}_{L_{j,m}} = P^{-1}_{IL_j}(L_m(\mathbf{G}_{\text{LL}_j})) = \mathcal{L}^p_{m_j} \cap \mathcal{E}_{L_{j,m}} \cap \mathcal{I}_{m_j}$$

[1]We also used $\mathcal{I}$ and $\mathcal{I}_m$ in Chapter 4 to represent languages in the serial case. This should not cause any confusion as when $n = 1$ (the serial case), they become equivalent.

5.2 High Level Synthesis

We start by examining how, give system Φ, we can synthesize a supervisor for the high level. Our first step is to capture the HISC properties that the supervisor's marked language must satisfy.

Definition 5.2.1 *Let $Z \subseteq \Sigma^*$. For system Φ, language Z is* high level interface controllable (HIC) *if for all $s \in \mathcal{H}^p \cap \mathcal{I} \cap \overline{Z}$, the following conditions are satisfied:*

1. $Elig_{\mathcal{H}^p \cap \mathcal{I}}(s) \cap \Sigma_u \subseteq Elig_{\overline{Z}}(s)$
2. $(\forall j \in \{1, \ldots, n\})\ Elig_{\mathcal{I}_j}(s) \cap \Sigma_{A_j} \subseteq Elig_{\mathcal{H}^p \cap \overline{Z} \cap \bigcap_{k \neq j} \mathcal{I}_k}(s)$

These conditions are essentially point 3 of Definition 3.6.1 and point 3 of Definition 3.4.2, where we have substituted $\overline{Z}$ for any reference of the high level supervisor's closed behavior, $\mathcal{S}_H$, and we have used the identity $\mathbf{G}_H := \mathbf{G}_H^p || \mathbf{S}_H$ for the high level subsystem.

For an arbitrary language $E \subseteq \Sigma^*$, we now define the set of all sublanguages of E that are high level interface controllable with respect to Φ as

$$\mathcal{C}_H(E) := \{Z \subseteq E |\ Z \text{ is HIC with respect to } \Phi\}$$

It is easy to see that $(\mathcal{C}_H(E), \subseteq)$ is a poset. We will now show that the set $\mathcal{C}_H(E)$ is nonempty, and that the supremum always exists.

Proposition 6 *Let $E \subseteq \Sigma^*$. For system Φ, $\mathcal{C}_H(E)$ is nonempty and is closed under arbitrary union. In particular, $\mathcal{C}_H(E)$ contains a (unique) supremal element that we will denote $sup\mathcal{C}_H(E)$.*

Proof

Let $E \subseteq \Sigma^*$.

We will break the proof into three parts: 1) show $\mathcal{C}_H(E)$ is nonempty, 2) show $\mathcal{C}_H(E)$ is closed under arbitrary union. 3) show $\mathcal{C}_H(E)$ contains a (unique) supremal element.

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

Clearly, $\emptyset \subseteq E$ and the empty set is HIC with respect to system Φ and is thus in $\mathcal{C}_H(E)$.

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

Let $Z_\beta \in \mathcal{C}_H(E)$ for all $\beta \in B$, where B is an index set. Let $Z = \cup\{Z_\beta | \, \beta \in B\}$.

Sufficient to show that $Z \in \mathcal{C}_H(E)$.

Clearly, $Z \subseteq E$. All we still need to show is that Z is HIC with respect to system Φ.

This means showing that for all $s \in \mathcal{H}^p \cap \mathcal{I} \cap \overline{Z}$, the following conditions are satisfied:

1. $\text{Elig}_{\mathcal{H}^p \cap \mathcal{I}}(s) \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z}}(s)$
2. $(\forall j \in \{1, \ldots, n\})\ \text{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{A_j} \subseteq \text{Elig}_{\mathcal{H}^p \cap \overline{Z} \cap \bigcap_{k \neq j} \mathcal{I}_k}(s)$

Let $s \in \mathcal{H}^p \cap \mathcal{I} \cap \overline{Z}$. **(1)**

We first note that this gives us $s \in \overline{Z}$

$\Rightarrow (\exists s' \in \Sigma^*)\, ss' \in Z$

$\Rightarrow (\exists \beta \in B)\, ss' \in Z_\beta$, by definition of Z.

$\Rightarrow s \in \overline{Z}_\beta$

We thus have: $s \in \mathcal{H}^p \cap \mathcal{I} \cap \overline{Z}_\beta$, by **(1)**. **(2)**

a) Show $\text{Elig}_{\mathcal{H}^p \cap \mathcal{I}}(s) \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z}}(s)$

Sufficient to show $(\forall \sigma \in \Sigma_u)\, s\sigma \in \mathcal{H}^p \cap \mathcal{I} \Rightarrow s\sigma \in \overline{Z}$

Let $\sigma \in \Sigma_u$. **(3)**

Assume $s\sigma \in \mathcal{H}^p \cap \mathcal{I}$ **(4)**

Will now show this implies $s\sigma \in \overline{Z}$.

We immediately have: $s \in \mathcal{H}^p \cap \mathcal{I} \cap \overline{Z}_\beta$, $\sigma \in \Sigma_u$, and $s\sigma \in \mathcal{H}^p \cap \mathcal{I}$ by **(2)**, **(3)**, and **(4)**.

As $Z_\beta \in \mathcal{C}_H(E)$ by definition and is thus HIC for Φ, we can conclude:
$s\sigma \in \overline{Z}_\beta$

$\Rightarrow (\exists s'' \in \Sigma^*)\, s\sigma s'' \in Z_\beta$

$\Rightarrow s\sigma s'' \in Z$, by definition of Z.

$\Rightarrow s\sigma \in \overline{Z}$, as required.

Part a complete.

b) Show $(\forall j \in \{1, \ldots, n\})\ \text{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{A_j} \subseteq \text{Elig}_{\mathcal{H}^p \cap \overline{Z} \cap \bigcap_{k \neq j} \mathcal{I}_k}(s)$ Let $j \in \{1, \ldots, n\}$.

Sufficient to show: $(\forall \alpha \in \Sigma_{A_j})\, s\alpha \in \mathcal{I}_j \Rightarrow s\alpha \in \mathcal{H}^p \cap \overline{Z} \cap \bigcap_{k \neq j} \mathcal{I}_k$

Let $\alpha \in \Sigma_{A_j}$ **(5)**

Assume $s\alpha \in \mathcal{I}_j$ **(6)**

Will now show implies $s\alpha \in \mathcal{H}^p \cap \overline{Z} \cap \bigcap_{k \neq j} \mathcal{I}_k$

We immediately have: $s \in \mathcal{H}^p \cap \mathcal{I} \cap \overline{Z}_\beta$, $\alpha \in \Sigma_{A_j}$, and $s\alpha \in \mathcal{I}_j$ by **(2)**, **(5)**, and **(6)**.

As $Z_\beta \in \mathcal{C}_H(E)$ by definition and is thus HIC for Φ, we can conclude:

$$s\alpha \in \mathcal{H}^p \cap \overline{Z}_\beta \cap \bigcap_{k \neq j} \mathcal{I}_k \qquad \textbf{(7)}$$

$\Rightarrow s\alpha \in \overline{Z}_\beta$

$\Rightarrow (\exists u \in \Sigma^*)\, s\alpha u \in Z_\beta$

$\Rightarrow s\alpha u \in Z$, by definition of Z.

$\Rightarrow s\alpha \in \overline{Z}$

Combining with **(7)**, we have $s\alpha \in \mathcal{H}^p \cap \overline{Z} \cap \bigcap_{k \neq j} \mathcal{I}_k$, as required.

Part b complete.

From Parts a and b, we can conclude that Z is HIC with respect to system Φ.

We can thus conclude that $Z \in \mathcal{C}_H(E)$, as required.

Part 2 complete.

3) Show $\mathcal{C}_H(E)$ contains a (unique) supremal element.

Sufficient to show that supremal element exists, as uniqueness would thus follow.

Let $\sup\mathcal{C}_H(E) = \cup\{Z |\, Z \in \mathcal{C}_H(E)\}$

Claim: $\sup\mathcal{C}_H(E)$ is the supremal element.

From Part 2, we have: $\sup\mathcal{C}_H(E) \in \mathcal{C}_H(E)$

Clearly, $(\forall Z \in \mathcal{C}_H(E))\, Z \subseteq \sup\mathcal{C}_H(E)$, thus $\sup\mathcal{C}_H(E)$ is an upper bound for $\mathcal{C}_H(E)$.

All that remains is to show:

$$(\forall Z' \in \mathcal{C}_H(E))\ ((\forall Z \in \mathcal{C}_H(E))\ Z \subseteq Z') \Rightarrow \sup\mathcal{C}_H(E) \subseteq Z'$$

Let $Z' \in \mathcal{C}_H(E)$.

Assume $(\forall Z \in \mathcal{C}_H(E))\ Z \subseteq Z'$ **(8)**

Must show implies $\sup\mathcal{C}_H(E) \subseteq Z'$

Let $s \in \sup\mathcal{C}_H(E)$. Must show implies $s \in Z'$.

$s \in \sup\mathcal{C}_H(E) \Rightarrow (\exists Z \in \mathcal{C}_H(E))\, s \in Z$, by definition of $\sup\mathcal{C}_H(E)$.

$\Rightarrow s \in Z'$, by **(8)**

We thus conclude that $\sup\mathcal{C}_H(E)$ is the supremal element.

Part 3 complete.

□

We now note that if we take language $E = \mathcal{Z}_{H_m}$, we can conclude that $\sup\mathcal{C}_H(\mathcal{Z}_{H_m}) = \sup\mathcal{C}_H(\mathcal{H}^p_m \cap \mathcal{E}_{H_m} \cap \mathcal{I}_m)$ exists. As $\sup\mathcal{C}_H(\mathcal{Z}_{H_m}) \subseteq \mathcal{Z}_{H_m}$ by definition, it follows that $\sup\mathcal{C}_H(\mathcal{Z}_{H_m}) \cap \mathcal{Z}_{H_m} = \sup\mathcal{C}_H(\mathcal{Z}_{H_m})$. This implies that $\overline{\sup\mathcal{C}_H(\mathcal{Z}_{H_m})} \subseteq \mathcal{Z}_H$ as $\mathcal{Z}_{H_m} \subseteq \mathcal{Z}_H$ and $\mathcal{Z}_H$ is closed. This means that if we take $\sup\mathcal{C}_H(\mathcal{Z}_{H_m})$ as the marked language of our high level supervisor, and $\overline{\sup\mathcal{C}_H(\mathcal{Z}_{H_m})}$ as the supervisor's closed behavior, then the supervisor will represent the closed loop behavior of the high level. It will thus follow that the high level will be nonblocking, and thus *point 1* of Definition 3.5.1 will automatically be satisfied.

5.2.1 High Level Fixpoint Operator

Now that we have shown that $\sup\mathcal{C}_H(\mathcal{Z}_{H_m})$ exists, we need a means to construct it. We will do so by defining a fixpoint operator Ω_H, and show that our supremal element is the greatest fixpoint of the operator. To do this, we need to first define functions Ω_{HNB} and Ω_{HIC}.

Definition 5.2.2 *For system Φ, we define the* high level nonblocking operator*, $\Omega_{HNB} : Pwr(\Sigma^*) \to Pwr(\Sigma^*)$, for arbitrary $Z \in Pwr(\Sigma^*)$ as follows:*

$$\Omega_{HNB}(Z) := Z \cap \mathcal{Z}_{H_m}$$

The way we will be using Ω_{HNB}, we would have $Z \subseteq \mathcal{Z}_H$ and closed, thus $\Omega_{\text{HNB}}(Z)$ would be the marked strings of the high level that remain in Z. Clearly, operator Ω_{HNB} is monotone.

Definition 5.2.3 *For system Φ, we define the* high level interface controllable operator*, $\Omega_{HIC} : Pwr(\Sigma^*) \to Pwr(\Sigma^*)$, for arbitrary $Z \in Pwr(\Sigma^*)$ as follows:*

$$\Omega_{HIC}(Z) := \overline{Z} - Ext_{\overline{Z}}(FailHIC(\overline{Z}))$$

where

$$FailHIC(\overline{Z}) := \{s \in \mathcal{H}^p \cap \mathcal{I} \cap \overline{Z} \mid \neg[Elig_{\mathcal{H}^p \cap \mathcal{I}}(s) \cap \Sigma_u \subseteq Elig_{\overline{Z}}(s)] \vee [(\exists j \in \{1, \ldots, n\}) \neg(Elig_{\mathcal{I}_j}(s) \cap \Sigma_{A_j} \subseteq Elig_{\mathcal{H}^p \cap \overline{Z} \cap \bigcap_{k \neq j} \mathcal{I}_k}(s))]\}$$

We first note that $\text{FailHIC}(\overline{Z}) \subseteq \overline{Z}$ and thus $\text{FailHIC}(\overline{Z}) \subseteq \text{Ext}_{\overline{Z}}(\text{FailHIC}(\overline{Z}))$ as $s \leq s$, for all $s \in \Sigma^*$. The way we will be using $\Omega_{\text{HIC}}(Z)$, we would have $Z \subseteq \mathcal{Z}_{H_m}$ and thus we would be removing from $\overline{Z}$ any string that has a prefix that would cause $\overline{Z}$ to fail the HIC definition. The reason we also remove the extensions of failing strings, is to ensure that we get a prefix closed language.

We first prove a proposition with a useful result.

Proposition 7 *Let $Z \in Pwr(\Sigma^*)$. For all $X \subseteq \overline{Z}$, it is true that*

$$\overline{\overline{Z} - Ext_{\overline{Z}}(X)} = \overline{Z} - Ext_{\overline{Z}}(X)$$

Proof

Let $(X \subseteq \overline{Z})$

As $\overline{\overline{Z} - \mathrm{Ext}_{\overline{Z}}(X)} \supseteq \overline{Z} - \mathrm{Ext}_{\overline{Z}}(X)$ is automatic, we only need to show $\subseteq$.

Let $s \in \overline{\overline{Z} - \mathrm{Ext}_{\overline{Z}}(X)}$ **(1)**

We will now show this implies: $s \in \overline{Z} - \mathrm{Ext}_{\overline{Z}}(X)$

From **(1)**, we have: $(\exists s' \in \Sigma^*) ss' \in \overline{Z} - \mathrm{Ext}_{\overline{Z}}(X)$

$\Rightarrow ss' \in \overline{Z} \wedge ss' \notin \mathrm{Ext}_{\overline{Z}}(X)$ **(2)**

$\Rightarrow ss' \notin \{t \in \overline{Z} \,|\, t' \leq t \text{ for some } t' \in X\}$, by definition of the Ext operator.

Clearly $ss' \notin \mathrm{Ext}_{\overline{Z}}(X) \Rightarrow s \notin \mathrm{Ext}_{\overline{Z}}(X)$ or we would have a contradiction.

$\Rightarrow s \in \overline{Z} \wedge s \notin \mathrm{Ext}_{\overline{Z}}(X)$ by **(2)** and fact that $\overline{Z}$ is closed.

$\Rightarrow s \in \overline{Z} - \mathrm{Ext}_{\overline{Z}}(X)$

□

Lemma 2 *Let $Z \in Pwr(\Sigma^*)$. For system Φ, the operator Ω_{HIC} always produces a prefix closed language. ie. $\Omega_{HIC}(Z) = \overline{\Omega_{HIC}(Z)}$*

Proof

We first note that by definition, we have: $\Omega_{\mathrm{HIC}}(Z) = \overline{Z} - \mathrm{Ext}_{\overline{Z}}(\mathrm{FailHIC}(\overline{Z}))$

It is thus sufficient to show that:

$\overline{\overline{Z} - \mathrm{Ext}_{\overline{Z}}(\mathrm{FailHIC}(\overline{Z}))} = \overline{Z} - \mathrm{Ext}_{\overline{Z}}(\mathrm{FailHIC}(\overline{Z}))$

We have FailHIC($\overline{Z}$) $\subseteq \overline{Z}$ by definition, so we can now apply Proposition 7 and conclude: $\overline{\overline{Z} - \text{Ext}_{\overline{Z}}(\text{FailHIC}(\overline{Z}))} = \overline{Z} - \text{Ext}_{\overline{Z}}(\text{FailHIC}(\overline{Z}))$

□

We now show that operator Ω_{HIC} is monotone.

Lemma 3 *For system Φ, the operator Ω_{HIC} is monotone. ie.*

$$(\forall Z, Z' \in Pwr(\Sigma^*))\ Z \subseteq Z' \Rightarrow \Omega_{HIC}(Z) \subseteq \Omega_{HIC}(Z')$$

Proof

Let $Z, Z' \in \text{Pwr}(\Sigma^*)$

Assume $Z \subseteq Z'$ **(1)**

Let $s \in \Omega_{\text{HIC}}(Z)$. **(2)**

We will now show this implies: $s \in \Omega_{\text{HIC}}(Z')$

By Definition of Ω_{HIC} operator, it is sufficient to show:
$s \in \overline{Z'} - \text{Ext}_{\overline{Z'}}(\text{FailHIC}(\overline{Z'}))$

From **(2)**, we have: $s \in \Omega_{\text{HIC}}(Z)$

$\Rightarrow s \in \overline{Z} - \text{Ext}_{\overline{Z}}(\text{FailHIC}(\overline{Z}))$, by definition of Ω_{HIC}.

$\Rightarrow s \in \overline{Z} \ \wedge \ s \notin \text{Ext}_{\overline{Z}}(\text{FailHIC}(\overline{Z}))$ **(3)**

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

$\Rightarrow s \in \overline{Z'}$ as $Z \subseteq Z'$ (by **(1)**), and fact prefix closure preserves ordering. **(4)**

All that remains now is to show that: $s \notin \text{Ext}_{\overline{Z'}}(\text{FailHIC}(\overline{Z'}))$

This means showing: $s \notin \{t \in \overline{Z'} \,|\, t' \leq t \text{ for some } t' \in \text{FailHIC}(\overline{Z'})\}$, by definition of the Ext operator.

Thus sufficient to show that: $(\forall s' \leq s)\, s' \notin \text{FailHIC}(\overline{Z'})$

Substituting for FailHIC($\overline{Z'}$), we see we must show:

$$(\forall s' \leq s)\, s' \notin \{t \in \mathcal{H}^p \cap \mathcal{I} \cap \overline{Z'} |\, \neg[\text{Elig}_{\mathcal{H}^p \cap \mathcal{I}}(t) \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z'}}(t)] \vee [(\exists j \in \{1, \ldots, n\})\neg(\text{Elig}_{\mathcal{I}_j}(t) \cap \Sigma_{A_j} \subseteq \text{Elig}_{\mathcal{H}^p \cap \overline{Z'} \cap \bigcap_{k \neq j} \mathcal{I}_k}(t))]\}$$

Which means it's sufficient to show:

$$(\forall s' \leq s)\, s' \in \mathcal{H}^p \cap \mathcal{I} \cap \overline{Z'} \Rightarrow [\text{Elig}_{\mathcal{H}^p \cap \mathcal{I}}(s') \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z'}}(s')] \wedge [(\forall j \in \{1, \ldots, n\})(\text{Elig}_{\mathcal{I}_j}(s') \cap \Sigma_{A_j} \subseteq \text{Elig}_{\mathcal{H}^p \cap \overline{Z'} \cap \bigcap_{k \neq j} \mathcal{I}_k}(s'))]$$

Let $s' \leq s$ **(6)**

Assume $s' \in \mathcal{H}^p \cap \mathcal{I} \cap \overline{Z'}$ **(7)**

We will now show this implies:

$$[\text{Elig}_{\mathcal{H}^p \cap \mathcal{I}}(s') \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z'}}(s')] \wedge [(\forall j \in \{1, \ldots, n\})(\text{Elig}_{\mathcal{I}_j}(s') \cap \Sigma_{A_j} \subseteq \text{Elig}_{\mathcal{H}^p \cap \overline{Z'} \cap \bigcap_{k \neq j} \mathcal{I}_k}(s'))] \quad \dagger$$

We next note that we have $s \notin \text{Ext}_{\overline{Z}}(\text{FailHIC}(\overline{Z}))$ by **(3)**.

$$\Rightarrow (\forall s'' \leq s)\, s'' \in \mathcal{H}^p \cap \mathcal{I} \cap \overline{Z} \Rightarrow [\text{Elig}_{\mathcal{H}^p \cap \mathcal{I}}(s'') \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z}}(s'')] \wedge [(\forall j \in \{1, \ldots, n\})(\text{Elig}_{\mathcal{I}_j}(s'') \cap \Sigma_{A_j} \subseteq \text{Elig}_{\mathcal{H}^p \cap \overline{Z} \cap \bigcap_{k \neq j} \mathcal{I}_k}(s''))] \quad \textbf{(8)}$$

We now note that as $s' \leq s$ by **(6)**, and $s \in \overline{Z}$ by **(3)**, it follows that $s' \in \overline{Z}$ as $\overline{Z}$ is closed.

$\Rightarrow s' \in \mathcal{H}^p \cap \mathcal{I} \cap \overline{Z}$, by **(7)**.

Using **(8)**, we can now conclude:

$$[\text{Elig}_{\mathcal{H}^p \cap \mathcal{I}}(s') \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z}}(s')] \wedge$$

$$[(\forall j \in \{1,\ldots,n\})(\text{Elig}_{\mathcal{I}_j}(s') \cap \Sigma_{A_j} \subseteq \text{Elig}_{\mathcal{H}^p \cap \overline{Z} \cap \bigcap_{k \neq j} \mathcal{I}_k}(s'))] \qquad \textbf{(9)}$$

We next note that we have $\overline{Z} \subseteq \overline{Z'}$, as $Z \subseteq Z'$ (by **(1)**) and fact prefix closure preserves ordering. **(10)**

We will now show that † is satisfied in two parts.

A) Show $\text{Elig}_{\mathcal{H}^p \cap \mathcal{I}}(s') \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z'}}(s')$

Sufficient to show: $(\forall \sigma \in \Sigma_u)\, s'\sigma \in \mathcal{H}^p \cap \mathcal{I} \Rightarrow s'\sigma \in \overline{Z'}$

Let $\sigma \in \Sigma_u$ and assume $s'\sigma \in \mathcal{H}^p \cap \mathcal{I}$.

$s'\sigma \in \overline{Z'}$ follows immediately from **(9)** and **(10)**.

B) Show $(\forall j \in \{1,\ldots,n\})(\text{Elig}_{\mathcal{I}_j}(s') \cap \Sigma_{A_j} \subseteq \text{Elig}_{\mathcal{H}^p \cap \overline{Z'} \cap \bigcap_{k \neq j} \mathcal{I}_k}(s'))$

Sufficient to show: $(\forall j \in \{1,\ldots,n\})(\forall \alpha \in \Sigma_{A_j})\, s'\alpha \in \mathcal{I}_j \Rightarrow s'\alpha \in \mathcal{H}^p \cap \overline{Z'} \cap \bigcap_{k \neq j} \mathcal{I}_k$

Let $j \in \{1,\ldots,n\}$, $\alpha \in \Sigma_{A_j}$, and assume $s'\alpha \in \mathcal{I}_j$.

$s'\alpha \in \mathcal{H}^p \cap \overline{Z'} \cap \bigcap_{k \neq j} \mathcal{I}_k$ follows immediately from **(9)** and **(10)**.

By **Part A** and **Part B**, we can now conclude that † is satisfied.

□

We now are ready to define our fixpoint operator Ω_H.

Definition 5.2.4 *For system Φ, we define the* high level fixpoint operator*, $\Omega_H : Pwr(\Sigma^*) \to Pwr(\Sigma^*)$, for arbitrary $Z \in Pwr(\Sigma^*)$ as follows:*

$$\Omega_H(Z) := \Omega_{HNB}(\Omega_{HIC}(Z))$$

As operators Ω_{HIC} and Ω_{HNB} are monotone, it is easy to show that Ω_H so defined is also monotone.

We next present two useful propositions before we give our main result for this section.

Proposition 8 *Let $Z, Z' \subseteq \Sigma^*$ be arbitrary languages. For system Φ, the follow properties are true:*

1. $Z \subseteq Z' \Rightarrow (\forall i \in \{0, 1, 2, \ldots\})\, \Omega_H^i(Z) \subseteq \Omega_H^i(Z')$
2. $\Omega_H(Z) = Z \;\Rightarrow\; Z \in \mathcal{C}_H(\mathcal{Z}_{H_m})$
3. *The sequence $\{\Omega_H^i(\mathcal{Z}_H),\, i = 0, 1, 2, \ldots\}$ is monotonically decreasing. ie.* $\Omega_H^{i+1}(\mathcal{Z}_H) \subseteq \Omega_H^i(\mathcal{Z}_H)$

Proof

1. Show $Z \subseteq Z' \Rightarrow (\forall i \in \{0, 1, 2, \ldots\})\, \Omega_H^i(Z) \subseteq \Omega_H^i(Z')$

Assume $Z \subseteq Z'$. **(1)**

We now present a proof by induction.

Base Case: $i = 0$

By definition, we get $\Omega_H^0(Z) = Z$ and $\Omega_H^0(Z') = Z'$.

$\Rightarrow \Omega_H^0(Z) \subseteq \Omega_H^0(Z')$ by **(1)**.

Inductive step: Let $i \in \{0, 1, 2, \ldots\}$

Assume $\Omega_H^i(Z) \subseteq \Omega_H^i(Z')$ **(2)**

We will now show this implies $\Omega_H^{i+1}(Z) \subseteq \Omega_H^{i+1}(Z')$

As Ω_H is monotone, it follows from **(2)** that:

$\Omega_H(\Omega_H^i(Z)) \subseteq \Omega_H(\Omega_H^i(Z'))$

$\Rightarrow \Omega_H^{i+1}(Z) \subseteq \Omega_H^{i+1}(Z')$, as required.

We can now conclude by induction that:

$(\forall i \in \{0, 1, 2, \ldots\})\, \Omega_H^i(Z) \subseteq \Omega_H^i(Z')$.

2. Show $\Omega_H(Z) = Z \Rightarrow Z \in \mathcal{C}_H(\mathcal{Z}_{H_m})$.

Assume $\Omega_H(Z) = Z$. **(3)**

We will show this implies $Z \in \mathcal{C}_H(\mathcal{Z}_{H_m})$

By definition of $\mathcal{C}_H$, it is sufficient to show that $Z \subseteq \mathcal{Z}_{H_m}$ and that Z is HIC with respect to Φ.

By **(3)** and the definition of Ω_H, we have: $Z = \Omega_{\text{HNB}}(\Omega_{\text{HIC}}(Z))$

$\Rightarrow Z = (\overline{Z} - \text{Ext}_{\overline{Z}}(\text{FailHIC}(\overline{Z}))) \cap \mathcal{Z}_{H_m}$

Which implies $Z \subseteq \mathcal{Z}_{H_m}$ and $Z \subseteq (\overline{Z} - \text{Ext}_{\overline{Z}}(\text{FailHIC}(\overline{Z})))$. **(4)**

All that remains is to show that Z is HIC with respect to Φ.

To do this, we first need to show that $\text{FailHIC}(\overline{Z}) = \emptyset$. We will do this using proof by contradiction:

Assume $\text{FailHIC}(\overline{Z}) \neq \emptyset$.

$\Rightarrow \exists s \in \text{FailHIC}(\overline{Z})$ **(5)**

As $\text{FailHIC}(\overline{Z}) \subseteq \overline{Z}$ by definition, we can conclude $s \in \overline{Z}$. **(6)**

$\Rightarrow (\exists s' \in \Sigma^*)\, ss' \in Z$ **(7)**

We can also conclude by **(5)** and the definition of the Ext operator that:

$s \in \text{Ext}_{\overline{Z}}(\text{FailHIC}(\overline{Z}))$

However, we have by **(7)** and **(4)** that:

$ss' \in \overline{Z} - \text{Ext}_{\overline{Z}}(\text{FailHIC}(\overline{Z}))$

$\Rightarrow ss' \notin \text{Ext}_{\overline{Z}}(\text{FailHIC}(\overline{Z})) \wedge ss' \in \overline{Z}$

$\Rightarrow (\forall s'' \in \text{FailHIC}(\overline{Z}))\, \neg(s'' \leq ss')$

Which contradicts **(5)**.

We thus conclude that $\text{FailHIC}(\overline{Z}) = \emptyset$.

$$\Rightarrow (\forall t \in \mathcal{H}^p \cap \mathcal{I} \cap \overline{Z})\, [\text{Elig}_{\mathcal{H}^p \cap \mathcal{I}}(t) \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z}}(t)] \wedge$$
$$[(\forall j \in \{1, \ldots, n\})(\text{Elig}_{\mathcal{I}_j}(t) \cap \Sigma_{A_j} \subseteq \text{Elig}_{\mathcal{H}^p \cap \overline{Z} \cap \bigcap_{k \neq j} \mathcal{I}_k}(t))]$$

Which implies by *Definition* 5.2.1 that Z is HIC with respect to Φ.

We thus have $Z \in \mathcal{C}_H(\mathcal{Z}_{H_m})$, as required.

3. Show $\Omega_H^{i+1}(\mathcal{Z}_H) \subseteq \Omega_H^i(\mathcal{Z}_H)$, for $i = 0, 1, 2, \ldots$.

We will first show that $\Omega_H^1(\mathcal{Z}_H) \subseteq \Omega_H^0(\mathcal{Z}_H)$, i.e., $\Omega_H(\mathcal{Z}_H) \subseteq \mathcal{Z}_H$.

By definition of Ω_H, we have:

$\Omega_H(\mathcal{Z}_H) = \Omega_{\text{HNB}}(\Omega_{\text{HIC}}(\mathcal{Z}_H)) = \Omega_{\text{HIC}}(\mathcal{Z}_H) \cap \mathcal{Z}_{H_m} \subseteq \mathcal{Z}_{H_m} \subseteq \mathcal{Z}_H$

We thus have $\Omega_H(\mathcal{Z}_H) \subseteq \mathcal{Z}_H$.

This means we can take $Z = \Omega_H(\mathcal{Z}_H)$, and $Z' = \mathcal{Z}_H$, and apply *point 1.*

We thus take $i \in \{0, 1, 2, \ldots\}$ and can conclude:

$\Omega_H^i(\Omega_H(\mathcal{Z}_H)) \subseteq \Omega_H^i(\mathcal{Z}_H)$

$\Rightarrow \Omega_H^{i+1}(\mathcal{Z}_H) \subseteq \Omega_H^i(\mathcal{Z}_H)$, as required.

□

Proposition 9 *For system* Φ*,* $sup\mathcal{C}_H(\mathcal{Z}_{H_m})$ *is the greatest fixpoint of* Ω_H*.*

Proof

To prove that $\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})$ is the greatest fixpoint of Ω_H, we need to show:

1. $\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m}) = \Omega_H(\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m}))$
2. $(\forall Z \in \text{Pwr}(\Sigma^*))\, Z = \Omega_H(Z) \Rightarrow Z \subseteq \text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})$

The second part follows from *Point 2* of *Proposition 8*. As every fixpoint is in $\mathcal{C}_H(\mathcal{Z}_{H_m})$, it follows that the fixpoint is $\subseteq \text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})$ since $\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})$ is the supremal element of $\mathcal{C}_H(\mathcal{Z}_{H_m})$.

All that is left to show is that $\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})$ is a fixpoint of Ω_H.

We first note that by definition of Ω_H we have:

$\Omega_H(\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})) = \Omega_{\text{HIC}}(\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})) \cap \mathcal{Z}_{H_m}$ **(1)**

By definition of Ω_{HIC} we have: **(2)**

$\Omega_H(\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})) = (\overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})} - \text{Ext}_{\overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})}}(\text{FailHIC}(\overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})}))) \cap \mathcal{Z}_{H_m}$

We now note that as $\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})$ is HIC with respect to Φ, by definition.

By *Definition* 5.2.1, it thus follows that: $\text{FailHIC}(\overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})}) = \emptyset$

$\Rightarrow \Omega_{\text{HIC}}(\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})) = \overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})} - \emptyset = \overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})}$, by definition of the Ext operator.

$\Rightarrow \Omega_H(\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})) = \overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})} \cap \mathcal{Z}_{H_m}$, by **(1)**. **(3)**

We are now ready to show $\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m}) = \Omega_H(\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m}))$.

(I) Show $\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m}) \subseteq \Omega_H(\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m}))$

By **(3)**, is its sufficient to show that $\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m}) \subseteq \overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})} \cap \mathcal{Z}_{H_m}$

We first note that $\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m}) \subseteq \overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})}$, by definition of prefix closure.

Also as $\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})$ is in $\mathcal{C}_H(\mathcal{Z}_{H_m})$, we have $\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m}) \subseteq \mathcal{Z}_{H_m}$.

$\Rightarrow \text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m}) \subseteq \overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})} \cap \mathcal{Z}_{H_m}$

Part (I) complete.

(II) Show $\Omega_H(\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})) \subseteq \text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})$

Let $s \in \Omega_H(\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m}))$. **(4)**

We will now show this implies $s \in \text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})$

From **(4)** and **(2)**, we can conclude that:

$$s \in (\overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})} - \text{Ext}_{\overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})}}(\text{FailHIC}(\overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})}))) \cap \mathcal{Z}_{H_m} \quad \textbf{(5)}$$

$$\Rightarrow [s \in \overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})}] \wedge [s \notin \text{Ext}_{\overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})}}(\text{FailHIC}(\overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})}))] \quad \textbf{(6)}$$

$\Rightarrow (\forall s' \leq s) s' \notin \text{FailHIC}(\overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})})$ by definition of the Ext operator.

$$\Rightarrow (\forall s' \in \overline{\{s\}})\, s' \notin \text{FailHIC}(\overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})}) \quad \textbf{(7)}$$

We next note that $s \in \overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})}$ (by **(6)**) implies that $s \in \mathcal{Z}_H$ as $\mathcal{Z}_H$ is closed and $\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m}) \subseteq \mathcal{Z}_{H_m} \subseteq \mathcal{Z}_H$, thus $\overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})} \subseteq \overline{\mathcal{Z}_{H_m}} \subseteq \overline{\mathcal{Z}_H}$ as prefix closure respects ordering.

We thus have $s \in \mathcal{H}^p \cap \mathcal{I}$ by definition of $\mathcal{Z}_H$.

$\Rightarrow s \in \mathcal{H}^p \cap \mathcal{I} \cap \overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})}$

$\Rightarrow (\forall s' \in \overline{\{s\}})\, s' \in \mathcal{H}^p \cap \mathcal{I} \cap \overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})}$ as all three languages are closed.

Combining with **(7)**, we can conclude that for all $s' \in \overline{\{s\}}$, the following is true:

1. $s' \in \mathcal{H}^p \cap \mathcal{I} \cap \overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})}$

2. $\text{Elig}_{\mathcal{H}^p \cap \mathcal{I}}(s') \cap \Sigma_u \subseteq \text{Elig}_{\overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})}}(s')$ **(8)**

3. $(\forall j \in \{1, \ldots, n\})\ \text{Elig}_{\mathcal{I}_j}(s') \cap \Sigma_{A_j} \subseteq \text{Elig}_{\mathcal{H}^p \cap \overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})} \cap \bigcap_{k \neq j} \mathcal{I}_k}(s')$ **(9)**

Let $Z = \text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m}) \cup \{s\}$ **(10)**

We will now show that Z is in $\mathcal{C}_H(\mathcal{Z}_{H_m})$, which will imply $Z \subseteq \text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})$, giving us the needed result.

We first note that by **(10)**, we have:

$\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m}) \subseteq Z$

$\Rightarrow \overline{\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m})} \subseteq \overline{Z}$, as prefix closure preserves ordering. **(11)**

We next note that we have $\text{sup}\mathcal{C}_H(\mathcal{Z}_{H_m}) \subseteq \mathcal{Z}_{H_m}$ by definition and by **(5)**, we have $s \in \mathcal{Z}_{H_m}$

We thus have $Z \subseteq \mathcal{Z}_{H_m}$.

To show that Z is in $\mathcal{C}_H(\mathcal{Z}_{H_m})$, all that now remains is to demonstrate that Z is HIC with respect to system Φ.

Let $t \in \mathcal{H}^p \cap \mathcal{I} \cap \overline{Z}$ **(12)**

We will now show that the following conditions are satisfied:

1. $\text{Elig}_{\mathcal{H}^p \cap \mathcal{I}}(t) \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z}}(t)$

2. $(\forall j \in \{1, \ldots, n\})\ \text{Elig}_{\mathcal{I}_j}(t) \cap \Sigma_{A_j} \subseteq \text{Elig}_{\mathcal{H}^p \cap \overline{Z} \cap \bigcap_{k \neq j} \mathcal{I}_k}(t)$

1) Show $\text{Elig}_{\mathcal{H}^p \cap \mathcal{I}}(t) \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z}}(t)$

Let $\sigma \in \Sigma_u$, and $t\sigma \in \mathcal{H}^p \cap \mathcal{I}$.

Sufficient to show implies $t\sigma \in \overline{Z}$.

If $t \in \overline{Z} - \overline{\{s\}}$, we have $t \in \overline{\sup\mathcal{C}_H(\mathcal{Z}_{H_m})}$.

As $\sup\mathcal{C}_H(\mathcal{Z}_{H_m})$ is HIC for Φ, it follows that $t\sigma \in \overline{\sup\mathcal{C}_H(\mathcal{Z}_{H_m})}$.

$\Rightarrow t\sigma \in \overline{Z}$, by **(11)**.

If $t \in \overline{\{s\}}$, it follows directly from **(8)** and **(11)**.

2) Show $(\forall j \in \{1, \ldots, n\})\ \mathrm{Elig}_{\mathcal{I}_j}(t) \cap \Sigma_{A_j} \subseteq \mathrm{Elig}_{\mathcal{H}^p \cap \overline{Z} \cap \bigcap_{k \neq j} \mathcal{I}_k}(t)$

Let $j \in \{1, \ldots, n\}$, and $\alpha \in \Sigma_{A_j}$.

Assume $t\alpha \in \mathcal{I}_j$.

Sufficient to show implies $t\alpha \in \mathcal{H}^p \cap \overline{Z} \cap \bigcap_{k \neq j} \mathcal{I}_k$

If $t \in \overline{Z} - \overline{\{s\}}$, we have $t \in \overline{\sup\mathcal{C}_H(\mathcal{Z}_{H_m})}$.

As $\sup\mathcal{C}_H(\mathcal{Z}_{H_m})$ is HIC for Φ, it follows that $t\alpha \in \mathcal{H}^p \cap \overline{\sup\mathcal{C}_H(\mathcal{Z}_{H_m})} \cap \bigcap_{k \neq j} \mathcal{I}_k$

$\Rightarrow t\sigma \in \mathcal{H}^p \cap \overline{Z} \cap \bigcap_{k \neq j} \mathcal{I}_k$, by **(11)**.

If $t \in \overline{\{s\}}$, it follows directly from **(9)** and **(11)**.

By 1) and 2), Z is HIC with respect to system Φ .

$\Rightarrow Z \subseteq \sup\mathcal{C}_H(\mathcal{Z}_{H_m})$, as $\sup\mathcal{C}_H(\mathcal{Z}_{H_m})$ is the supremal element for $\mathcal{C}_H(\mathcal{Z}_{H_m})$

$\Rightarrow s \in \sup\mathcal{C}_H(\mathcal{Z}_{H_m})$ (by **(10)**) , as required.

Part (II) complete.

By (I) and (II), we get $\sup\mathcal{C}_H(\mathcal{Z}_{H_m}) = \Omega_H(\sup\mathcal{C}_H(\mathcal{Z}_{H_m}))$ as required.

We thus conclude that $\sup\mathcal{C}_H(\mathcal{Z}_{H_m})$ is the greatest fixpoint point of Ω_H.

□

We will now show that if $\Omega_H(\mathcal{Z}_H)$ reaches a fixpoint after a finite number of steps, then that fixpoint is our supremal element. In Chapter 6, we will give an automata based algorithm that implements $\Omega_H(\mathcal{Z}_H)$. As the algorithm operates by removing one or more states of $\mathbf{G}_{\text{HL}}$ which is assumed to have a finite state space, we know it will complete in a finite number of steps (ie. it must stop when we have no more states left to remove).

Theorem 8 *For system Φ, if there exists $i \in \{0, 1, 2, \ldots\}$ such that $\Omega^i_H(\mathcal{Z}_H)$ is a fixpoint, then $\Omega^i_H(\mathcal{Z}_H) = sup\mathcal{C}_H(\mathcal{Z}_{H_m})$.*

Proof

Assume $\exists i \in \{0, 1, 2, \ldots\}$, such that $\Omega_H(\Omega^i_H(\mathcal{Z}_H)) = \Omega^i_H(\mathcal{Z}_H)$ **(1)**

We first note that we have: $\sup\mathcal{C}_H(\mathcal{Z}_{H_m}) \subseteq \mathcal{Z}_{H_m} \subseteq \mathcal{Z}_H$

This allows us to apply *Point 1* of *Proposition 8* and conclude:

$\Omega^i_H(\sup\mathcal{C}_H(\mathcal{Z}_{H_m})) \subseteq \Omega^i_H(\mathcal{Z}_H)$ **(3)**

By *Proposition 9*, we know that $\sup\mathcal{C}_H(\mathcal{Z}_{H_m})$ is the greatest fixpoint of Ω_H. **(4)**

$\Rightarrow \Omega^i_H(\sup\mathcal{C}_H(\mathcal{Z}_{H_m})) = \sup\mathcal{C}_H(\mathcal{Z}_{H_m})$

Combine this with **(3)**, and we can conclude:

$\sup\mathcal{C}_H(\mathcal{Z}_{H_m}) \subseteq \Omega^i_H(\mathcal{Z}_H)$ **(5)**

As $\sup\mathcal{C}_H(\mathcal{Z}_{H_m})$ is the greatest fixpoint of Ω_H (by **(4)**) and $\Omega^i_H(\mathcal{Z}_H)$ is a fixpoint, it thus follows: $\Omega^i_H(\mathcal{Z}_H) \subseteq \sup\mathcal{C}_H(\mathcal{Z}_{H_m})$

By **(5)**, we thus have $\Omega^i_H(\mathcal{Z}_H) = \sup\mathcal{C}_H(\mathcal{Z}_{H_m})$ as required.

□

We now show that we can use $\sup\mathcal{C}_H(\mathcal{Z}_{H_m})$ for our high level supervisor and satisfy the relevant interface conditions. We will use $S_{H_m} \subseteq \Sigma^*$ to stand for the marked language of the high level supervisor.

Corollary 2 *For system Φ, if there exists $i \in \{0, 1, 2, \ldots\}$ such that $\Omega^i_H(\mathcal{Z}_H)$ is a fixpoint, then system Φ with $S_{H_m} = \Omega^i_H(\mathcal{Z}_H)$ and $S_H = \overline{S_{H_m}}$ satisfies Point 3 of Definition 3.4.2, Point I of Definition 3.5.1 and Point III of Definition 3.6.1.*

Proof

Assume $\exists i \in \{0, 1, 2, \ldots\}$, such that $\Omega_H(\Omega^i_H(\mathcal{Z}_H)) = \Omega^i_H(\mathcal{Z}_H)$. **(1)**

Let $S_{H_m} = \Omega^i_H(\mathcal{Z}_H)$ and $S_H = \overline{S_{H_m}}$.

By Theorem 8, $S_{H_m} = \sup\mathcal{C}_H(\mathcal{Z}_{H_m})$ is HIC with respect to Φ. **(2)**

By Definition 5.2.1 and using the fact that $\overline{S_{H_m}} = S_H$, we have for all s in $\mathcal{H}^p \cap S_H \cap \mathcal{I}$

1. $\text{Elig}_{\mathcal{H}^p \cap \mathcal{I}}(s) \cap \Sigma_u \subseteq \text{Elig}_{S_H}(s)$ **(3)**
2. $(\forall j \in \{1, \ldots, n\})\ \text{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{A_j} \subseteq \text{Elig}_{\mathcal{H}^p \cap S_H \cap \bigcap_{k \neq j} \mathcal{I}_k}(s)$ **(4)**

We note that Point (III) of Definition 3.6.1 follows immediately from **(3)**.

Using the fact that $\mathcal{H} = \mathcal{H}^p \cap S_H$, we can substitute into **(4)** and get for all s in $\mathcal{H} \cap \mathcal{I}$

$$(\forall j \in \{1, \ldots, n\})\ \text{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{A_j} \subseteq \text{Elig}_{\mathcal{H} \cap \bigcap_{k \neq j} \mathcal{I}_k}(s)$$

Point 3 of Definition 3.4.2 immediately follows.

All that remains is to show that Point I of Definition 3.5.1 is satisfied.

This means showing that $\overline{\mathcal{H}_m \cap \mathcal{I}_m} = \mathcal{H} \cap \mathcal{I}$.

By **(2)**, we have $S_{H_m} = \sup\mathcal{C}_H(\mathcal{Z}_{H_m})$.

$\Rightarrow S_{H_m} \subseteq \mathcal{Z}_{H_m}$, as $\sup\mathcal{C}_H(\mathcal{Z}_{H_m}) \subseteq \mathcal{Z}_{H_m}$ by definition. **(5)**

$\Rightarrow S_{H_m} \subseteq \mathcal{Z}_H$, as $\mathcal{Z}_{H_m} \subseteq \mathcal{Z}_H$

$\Rightarrow \overline{S_{H_m}} \subseteq \mathcal{Z}_H$, as $\mathcal{Z}_H$ is closed and prefix closure preserves ordering.

$\Rightarrow S_H \subseteq \mathcal{Z}_H$, by definition of S_H. **(6)**

Substituting for $\mathcal{Z}_{H_m}$ in **(5)**, we get $S_{H_m} \subseteq \mathcal{H}_m^p \cap \mathcal{E}_{H_m} \cap \mathcal{I}_m$. **(7)**

Substituting for $\mathcal{Z}_H$ in **(6)**, we get $S_H \subseteq \mathcal{H}^p \cap \mathcal{E}_H \cap \mathcal{I}$. **(8)**

Using the fact that $\mathcal{H}_m = \mathcal{H}_m^p \cap S_{H_m}$, we get $\mathcal{H}_m \cap \mathcal{I}_m = \mathcal{H}_m^p \cap S_{H_m} \cap \mathcal{I}_m$.

$\Rightarrow \mathcal{H}_m \cap \mathcal{I}_m = S_{H_m}$, by **(7)**. **(9)**

Using the fact that $\mathcal{H} = \mathcal{H}^p \cap S_H$, we get $\mathcal{H} \cap \mathcal{I} = \mathcal{H}^p \cap S_H \cap \mathcal{I}$.

$\Rightarrow \mathcal{H} \cap \mathcal{I} = S_H$, by **(8)**.

As $S_H = \overline{S_{H_m}}$, by definition, it follows from **(9)** that $\overline{\mathcal{H}_m \cap \mathcal{I}_m} = \mathcal{H} \cap \mathcal{I}$, as required.

□

5.3 Low Level Synthesis

We now exam how, given system Φ, we can synthesize a supervisor for the j^{th} low level. Our first step is to capture the HISC properties that the supervisor's marked

language must satisfy.

Definition 5.3.1 *Let $Z \subseteq \Sigma^*$. For system Φ, language Z is j^{th}* low level interface controllable (LICj) *if for all $s \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{Z}$, the following conditions are satisfied:*

1. $Elig_{\mathcal{L}_j^p}(s) \cap \Sigma_u \subseteq Elig_{\overline{Z} \cap \mathcal{I}_j}(s)$
2. $Elig_{\mathcal{I}_j}(s) \cap \Sigma_{R_j} \subseteq Elig_{\mathcal{L}_j^p \cap \overline{Z}}(s)$
3. $(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})$
   $s\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z} \cap \mathcal{I}_j$
4. $s \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, sl \in \mathcal{L}_{m_j}^p \cap Z \cap \mathcal{I}_{m_j}$

These conditions are essentially point 2 of Definition 3.6.1, and points 4-6 of Definition 3.4.2, where we have substituted $\overline{Z}$ for any reference of the j^{th} low level supervisor's closed behavior ($\mathcal{S}_{L_j}$), Z for any reference of the supervisor's marked language, and we have used the identity $\mathbf{G}_{L_j} := \mathbf{G}_{L_j}^p || \mathbf{S}_{L_j}$ for the j^{th} low level subsystem.

For an arbitrary language $E \subseteq \Sigma^*$, we now define the set of all sublanguages of E that are j^{th} low level interface controllable with respect to Φ as

$$\mathcal{C}_{L_j}(E) := \{Z \subseteq E |\, Z \text{ is LICj with respect to } \Phi\}$$

It is easy to see that $(\mathcal{C}_{L_j}(E), \subseteq)$ is a poset. We will now show that the set $\mathcal{C}_{L_j}(E)$ is nonempty, and that the supremum always exists.

Proposition 10 *Let $E \subseteq \Sigma^*$. For system Φ, $\mathcal{C}_{L_j}(E)$ is nonempty and is closed under arbitrary union. In particular, $\mathcal{C}_{L_j}(E)$ contains a (unique) supremal element that we will denote $sup\mathcal{C}_{L_j}(E)$.*

Proof

Let $E \subseteq \Sigma^*$.

We will break the proof into three parts: 1) show $\mathcal{C}_{L_j}(E)$ is nonempty, 2) show $\mathcal{C}_{L_j}(E)$ is closed under arbitrary union. 3) show $\mathcal{C}_{L_j}(E)$ contains a (unique) supremal element.

1) Show $\mathcal{C}_{L_j}(E)$ is nonempty.

Clearly, $\emptyset \subseteq E$ and the empty set is LICj with respect to system Φ and is thus in $\mathcal{C}_{L_j}(E)$.

2) Show $\mathcal{C}_{L_j}(E)$ is closed under arbitrary union.

Let $Z_\beta \in \mathcal{C}_{L_j}(E)$ for all $\beta \in B$, where B is an index set. Let $Z = \cup\{Z_\beta | \, \beta \in B\}$.

Clearly $Z_\beta \subseteq Z$ for each $\beta \in B$.

$\Rightarrow (\forall \beta \in B)\, \overline{Z_\beta} \subseteq \overline{Z}$ as prefix closure preserves ordering. **(1)**

Sufficient to show that $Z \in \mathcal{C}_{L_j}(E)$.

Clearly, $Z \subseteq E$. All we still need to show is that Z is LICj with respect to system Φ.

This means showing that for all $s \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{Z}$, the following conditions are satisfied:

1. $\text{Elig}_{\mathcal{L}_j^p}(s) \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z} \cap \mathcal{I}_j}(s)$
2. $\text{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{R_j} \subseteq \text{Elig}_{\mathcal{L}_j^p \cap \overline{Z}}(s)$
3. $(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})$
   $s\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z} \cap \mathcal{I}_j$
4. $s \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, sl \in \mathcal{L}_{m_j}^p \cap Z \cap \mathcal{I}_{m_j}$

Let $s \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{Z}$. **(2)**

We first note that this gives us $s \in \overline{Z}$

$\Rightarrow (\exists s' \in \Sigma^*)\, ss' \in Z$

$\Rightarrow (\exists \beta \in B)\, ss' \in Z_\beta$, by definition of Z.

$\Rightarrow s \in \overline{Z}_\beta$

We thus have: $s \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{Z}_\beta$, by **(2)**. **(3)**

a) Show $\text{Elig}_{\mathcal{L}_j^p}(s) \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z} \cap \mathcal{I}_j}(s)$

Sufficient to show $(\forall \sigma \in \Sigma_u)\, s\sigma \in \mathcal{L}_j^p \Rightarrow s\sigma \in \overline{Z} \cap \mathcal{I}_j$

Let $\sigma \in \Sigma_u$. **(4)**

Assume $s\sigma \in \mathcal{L}_j^p$ **(5)**

Will now show this implies $s\sigma \in \overline{Z} \cap \mathcal{I}_j$.

We immediately have: $s \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{Z}_\beta$, $\sigma \in \Sigma_u$, and $s\sigma \in \mathcal{L}_j^p$ by **(3)**, **(4)**, and **(5)**.

As $Z_\beta \in \mathcal{C}_{L_j}(E)$ by definition and is thus LICj for Φ, we can conclude:
$s\sigma \in \overline{Z}_\beta \cap \mathcal{I}_j$

$\Rightarrow s\sigma \in \overline{Z} \cap \mathcal{I}_j$ (by**(1)**), as required.

Part a complete.

b) Show $\text{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{R_j} \subseteq \text{Elig}_{\mathcal{L}_j^p \cap \overline{Z}}(s)$

Sufficient to show $(\forall \rho \in \Sigma_{R_j})\, s\rho \in \mathcal{I}_j \Rightarrow s\rho \in \mathcal{L}_j^p \cap \overline{Z}$

Let $\rho \in \Sigma_{R_j}$. **(6)**

Assume $s\rho \in \mathcal{I}_j$ **(7)**

Will now show this implies $s\rho \in \mathcal{L}_j^p \cap \overline{Z}$.

We immediately have: $s \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{Z}_\beta$, $\rho \in \Sigma_{R_j}$, and $s\rho \in \mathcal{I}_j$ by **(3)**, **(6)**, and **(7)**.

As $Z_\beta \in \mathcal{C}_{L_j}(E)$ by definition and is thus LICj for Φ, we can conclude:
$s\sigma \in \mathcal{L}_j^p \cap \overline{Z}_\beta$

$\Rightarrow s\sigma \in \mathcal{L}_j^p \cap \overline{Z}$ (by**(1)**), as required.

Part b complete.

c) Show $(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\ s\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z} \cap \mathcal{I}_j$

Let $\rho \in \Sigma_{R_j}$, $\alpha \in \Sigma_{A_j}$. **(8)**

Assume $s\rho\alpha \in \mathcal{I}_j$ **(9)**

We will now show this implies $(\exists l \in \Sigma_{L_j}^*)\, s\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z} \cap \mathcal{I}_j$

We immediately have: $s \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{Z}_\beta$, $\rho \in \Sigma_{R_j}$, $\alpha \in \Sigma_{A_j}$, and $s\rho\alpha \in \mathcal{I}_j$ by **(3)**, **(8)**, and **(9)**.

As $Z_\beta \in \mathcal{C}_{L_j}(E)$ by definition and is thus LICj for Φ, we can conclude:
$(\exists l \in \Sigma_{L_j}^*)\, s\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z}_\beta \cap \mathcal{I}_j$

$\Rightarrow s\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z} \cap \mathcal{I}_j$ (by**(1)**), as required.

Part c complete.

d) Show $s \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, sl \in \mathcal{L}_{m_j}^p \cap Z \cap \mathcal{I}_{m_j}$

Assume $s \in \mathcal{I}_{m_j}$ **(10)**

We will now show this implies $(\exists l \in \Sigma_{L_j}^*)\, sl \in \mathcal{L}_{m_j}^p \cap Z \cap \mathcal{I}_{m_j}$

We immediately have: $s \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{Z}_\beta$ and $s \in \mathcal{I}_{m_j}$ by **(3)** and **(10)**.

As $Z_\beta \in \mathcal{C}_{L_j}(E)$ by definition and is thus LICj for Φ, we can conclude:

$(\exists l \in \Sigma^*_{L_j})\ sl \in \mathcal{L}^p_{m_j} \cap Z_\beta \cap \mathcal{I}_{m_j}$

$\Rightarrow sl \in \mathcal{L}^p_{m_j} \cap Z \cap \mathcal{I}_{m_j}$ (by definition of Z), as required.

Part d complete.

From Parts a, b, c and d, we can conclude that Z is LICj with respect to system Φ.

We can thus conclude that $Z \in \mathcal{C}_{L_j}(E)$, as required.

Part 2 complete.

3) Show $\mathcal{C}_{L_j}(E)$ contains a (unique) supremal element.

Sufficient to show that supremal element exists, as uniqueness would thus follow.

Let $\sup\mathcal{C}_{L_j}(E) = \cup\{Z |\ Z \in \mathcal{C}_{L_j}(E)\}$

Claim: $\sup\mathcal{C}_{L_j}(E)$ is the supremal element.

From Part 2, we have: $\sup\mathcal{C}_{L_j}(E) \in \mathcal{C}_{L_j}(E)$

Clearly, $(\forall Z \in \mathcal{C}_{L_j}(E))\, Z \subseteq \sup\mathcal{C}_{L_j}(E)$, thus $\sup\mathcal{C}_{L_j}(E)$ is an upper bound for $\mathcal{C}_{L_j}(E)$.

All that remains is to show:

$$(\forall Z' \in \mathcal{C}_{L_j}(E))\ ((\forall Z \in \mathcal{C}_{L_j}(E))\ Z \subseteq Z') \Rightarrow \sup\mathcal{C}_{L_j}(E) \subseteq Z'$$

Let $Z' \in \mathcal{C}_{L_j}(E)$.

Assume $(\forall Z \in \mathcal{C}_{L_j}(E))\ Z \subseteq Z'$ **(11)**

Must show implies $\sup\mathcal{C}_{L_j}(E) \subseteq Z'$

Let $s \in \sup\mathcal{C}_{L_j}(E)$. Must show implies $s \in Z'$.

$s \in \text{sup}\mathcal{C}_{L_j}(E) \Rightarrow (\exists Z \in \mathcal{C}_{L_j}(E))\, s \in Z$, by definition of $\text{sup}\mathcal{C}_{L_j}(E)$.

$\Rightarrow s \in Z'$, by **(11)**

We thus conclude that $\text{sup}\mathcal{C}_{L_j}(E)$ is the supremal element.

Part 3 complete.

□

We now note that if we take language $E = \mathcal{Z}_{L_{j,m}}$, we can conclude that $\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) = \text{sup}\mathcal{C}_{L_j}(\mathcal{L}^p_{m_j} \cap \mathcal{E}_{L_{j,m}} \cap \mathcal{I}_{m_j})$ exists. As $\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) \subseteq \mathcal{Z}_{L_{j,m}}$ by definition, it follows that $\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) \cap \mathcal{Z}_{L_{j,m}} = \text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$. This implies that $\overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})} \subseteq \mathcal{Z}_{L_j}$ as $\mathcal{Z}_{L_{j,m}} \subseteq \mathcal{Z}_{L_j}$, and $\mathcal{Z}_{L_j}$ is closed. This means that if we take $\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ as the marked language of our j^{th} low level supervisor, and $\overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}$ as the supervisor's closed behavior, then the supervisor will represent the closed loop behavior of the j^{th} low level. It will thus follow that the j^{th} low level will be nonblocking, and thus *point 2* of Definition 3.5.1 will automatically be satisfied for this j.

5.3.1 The j^{th} Low Level Fixpoint Operator

Now that we have shown that $\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ exists, we need a means to construct it. We will do so by defining a fixpoint operator Ω_{L_j}, and show that our supremal element is the greatest fixpoint of the operator. To do this, we need to first define functions Ω_{LNB_j} and Ω_{LIC_j}.

Definition 5.3.2 *For system Φ, we define the j^{th}* low level nonblocking operator, *$\Omega_{LNB_j} : Pwr(\Sigma^*) \rightarrow Pwr(\Sigma^*)$, for arbitrary $Z \in Pwr(\Sigma^*)$ as follows:*

$$\Omega_{LNB_j}(Z) := Z \cap \mathcal{Z}_{L_{j,m}}$$

The way we will be using Ω_{LNB_j}, we would have $Z \subseteq \mathcal{Z}_{L_j}$ and closed, thus $\Omega_{\text{LNB}_j}(Z)$ would be the marked strings of the j^{th} low level that remain in Z. Clearly, operator

Ω_{LNB_j} is monotone.

Definition 5.3.3 *For system Φ, we define the j^{th}* low level interface controllable operator, *$\Omega_{LIC_j} : Pwr(\Sigma^*) \rightarrow Pwr(\Sigma^*)$, for arbitrary $Z \in Pwr(\Sigma^*)$ as follows:*

$$\Omega_{LIC_j}(Z) := \overline{Z} - Ext_{\overline{Z}}(FailLIC_j(\overline{Z}))$$

where

$$\begin{aligned} FailLIC_j(\overline{Z}) := \{ & s \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{Z} \mid \neg[Elig_{\mathcal{L}_j^p}(s) \cap \Sigma_u \subseteq Elig_{\overline{Z} \cap \mathcal{I}_j}(s)] \\ & \vee \neg[Elig_{\mathcal{I}_j}(s) \cap \Sigma_{R_j} \subseteq Elig_{\mathcal{L}_j^p \cap \overline{Z}}(s)] \\ & \vee \neg[(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\, s\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z} \cap \mathcal{I}_j] \\ & \vee \neg[s \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, sl \in \mathcal{L}_{m_j}^p \cap Z \cap \mathcal{I}_{m_j}]\} \end{aligned}$$

We first note that $\text{FailLIC}_j(\overline{Z}) \subseteq \overline{Z}$ and thus $\text{FailLIC}_j(\overline{Z}) \subseteq \text{Ext}_{\overline{Z}}(\text{FailLIC}_j(\overline{Z}))$ as $s \leq s$, for all $s \in \Sigma^*$. The way we will be using $\Omega_{\text{LIC}_j}(Z)$, we would have $Z \subseteq \mathcal{Z}_{L_{j,m}}$ and thus we would be removing from $\overline{Z}$ any string that has a prefix that would cause $\overline{Z}$ to fail the LICj definition. The reason we also remove the extensions of failing strings, is to ensure that we get a prefix closed language.

Lemma 4 *Let $Z \in Pwr(\Sigma^*)$. For system Φ, the operator Ω_{LIC_j} always produces a prefix closed language. ie. $\Omega_{LIC_j}(Z) = \overline{\Omega_{LIC_j}(Z)}$*

Proof

We first note that by definition, we have: $\Omega_{\text{LIC}_j}(Z) = \overline{Z} - \text{Ext}_{\overline{Z}}(\text{FailLIC}_j(\overline{Z}))$

It is thus sufficient to show that:

$\overline{\overline{Z} - \text{Ext}_{\overline{Z}}(\text{FailLIC}_j(\overline{Z}))} = \overline{Z} - \text{Ext}_{\overline{Z}}(\text{FailLIC}_j(\overline{Z}))$

We have $\text{FailLIC}_j(\overline{Z}) \subseteq \overline{Z}$ by definition, so we can now apply Proposition 7 and conclude: $\overline{\overline{Z} - \text{Ext}_{\overline{Z}}(\text{FailLIC}_j(\overline{Z}))} = \overline{Z} - \text{Ext}_{\overline{Z}}(\text{FailLIC}_j(\overline{Z}))$

□

We now show that operator Ω_{LIC_j} is monotone.

Lemma 5 *For system Φ, the operator Ω_{LIC_j} is monotone. ie.*

$$(\forall Z, Z' \in Pwr(\Sigma^*))\, Z \subseteq Z' \Rightarrow \Omega_{LIC_j}(Z) \subseteq \Omega_{LIC_j}(Z')$$

Proof

Let $Z, Z' \in \mathrm{Pwr}(\Sigma^*)$

Assume $Z \subseteq Z'$ **(1)**

Let $s \in \Omega_{\mathrm{LIC}_j}(Z)$. **(2)**

We will now show this implies: $s \in \Omega_{\mathrm{LIC}_j}(Z')$.

By Definition of Ω_{LIC_j} operator, it is sufficient to show:
$s \in \overline{Z'} - \mathrm{Ext}_{\overline{Z'}}(\mathrm{FailLIC}_j(\overline{Z'}))$

From **(2)**, we have: $s \in \Omega_{\mathrm{LIC}_j}(Z)$

$\Rightarrow s \in \overline{Z} - \mathrm{Ext}_{\overline{Z}}(\mathrm{FailLIC}_j(\overline{Z}))$, by definition of Ω_{LIC_j}.

$\Rightarrow s \in \overline{Z} \ \wedge \ s \notin \mathrm{Ext}_{\overline{Z}}(\mathrm{FailLIC}_j(\overline{Z}))$ **(3)**

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

$\Rightarrow s \in \overline{Z'}$ as $Z \subseteq Z'$ (by **(1)**), and fact prefix closure preserves ordering. **(4)**

All that remains now is to show that: $s \notin \mathrm{Ext}_{\overline{Z'}}(\mathrm{FailLIC}_j(\overline{Z'}))$

This means showing: $s \notin \{t \in \overline{Z'} \,|\, t' \leq t \text{ for some } t' \in \mathrm{FailLIC}_j(\overline{Z'})\}$, by definition of the Ext operator.

Thus sufficient to show that: $(\forall s' \leq s)\, s' \notin \mathrm{FailLIC}_j(\overline{Z'})$

Substituting for FailHIC($\overline{Z'}$), we see we must show:

$$(\forall s' \leq s)\, s' \notin \{t \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{Z'} \mid \neg[\mathrm{Elig}_{\mathcal{L}_j^p}(t) \cap \Sigma_u \subseteq \mathrm{Elig}_{\overline{Z'} \cap \mathcal{I}_j}(t)]$$
$$\vee\ \neg[\mathrm{Elig}_{\mathcal{I}_j}(t) \cap \Sigma_{R_j} \subseteq \mathrm{Elig}_{\mathcal{L}_j^p \cap \overline{Z'}}(t)]$$
$$\vee\ \neg[(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\, t\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, t\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z'} \cap \mathcal{I}_j]$$
$$\vee\ \neg[t \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, tl \in \mathcal{L}_{m_j}^p \cap Z' \cap \mathcal{I}_{m_j}]\}$$

Which means it's sufficient to show:

$$(\forall s' \leq s)\, s' \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{Z'} \Rightarrow [\mathrm{Elig}_{\mathcal{L}_j^p}(s') \cap \Sigma_u \subseteq \mathrm{Elig}_{\overline{Z'} \cap \mathcal{I}_j}(s')]$$
$$\wedge\ [\mathrm{Elig}_{\mathcal{I}_j}(s') \cap \Sigma_{R_j} \subseteq \mathrm{Elig}_{\mathcal{L}_j^p \cap \overline{Z'}}(s')]$$
$$\wedge\ [(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\, s'\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s'\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z'} \cap \mathcal{I}_j]$$
$$\wedge\ [s' \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s'l \in \mathcal{L}_{m_j}^p \cap Z' \cap \mathcal{I}_{m_j}]$$

Let $s' \leq s$ **(6)**

Assume $s' \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{Z'}$ **(7)**

We will now show this implies:

$$[\mathrm{Elig}_{\mathcal{L}_j^p}(s') \cap \Sigma_u \subseteq \mathrm{Elig}_{\overline{Z'} \cap \mathcal{I}_j}(s')] \wedge [\mathrm{Elig}_{\mathcal{I}_j}(s') \cap \Sigma_{R_j} \subseteq \mathrm{Elig}_{\mathcal{L}_j^p \cap \overline{Z'}}(s')]$$
$$\wedge\ [(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\, s'\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s'\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z'} \cap \mathcal{I}_j]$$
$$\wedge\ [s' \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s'l \in \mathcal{L}_{m_j}^p \cap Z' \cap \mathcal{I}_{m_j}] \qquad \dagger$$

We next note that we have $s \notin \mathrm{Ext}_{\overline{Z}}(\mathrm{FailLIC}_j(\overline{Z}))$ by **(3)**.

$$\Rightarrow (\forall s'' \leq s)\, s'' \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{Z} \Rightarrow [\mathrm{Elig}_{\mathcal{L}_j^p}(s'') \cap \Sigma_u \subseteq \mathrm{Elig}_{\overline{Z} \cap \mathcal{I}_j}(s'')]$$
$$\wedge\ [\mathrm{Elig}_{\mathcal{I}_j}(s'') \cap \Sigma_{R_j} \subseteq \mathrm{Elig}_{\mathcal{L}_j^p \cap \overline{Z}}(s'')]$$
$$\wedge\ [(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\, s''\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s''\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z} \cap \mathcal{I}_j]$$
$$\wedge\ [s'' \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s''l \in \mathcal{L}_{m_j}^p \cap Z \cap \mathcal{I}_{m_j}] \qquad \textbf{(8)}$$

We now note that as $s' \leq s$ by **(6)**, and $s \in \overline{Z}$ by **(3)**, it follows that $s' \in \overline{Z}$ as $\overline{Z}$ is closed.

$\Rightarrow s' \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{Z}$, by **(7)**.

Using **(8)**, we can now conclude:

$$[\text{Elig}_{\mathcal{L}_j^p}(s') \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z} \cap \mathcal{I}_j}(s'')] \wedge [\text{Elig}_{\mathcal{I}_j}(s') \cap \Sigma_{R_j} \subseteq \text{Elig}_{\mathcal{L}_j^p \cap \overline{Z}}(s')]$$
$$\wedge [(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\, s'\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s'\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z} \cap \mathcal{I}_j]$$
$$\wedge [s' \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s'l \in \mathcal{L}_{m_j}^p \cap Z \cap \mathcal{I}_{m_j}] \quad \textbf{(9)}$$

We next note that we have $\overline{Z} \subseteq \overline{Z'}$, as $Z \subseteq Z'$ (by **(1)**) and fact prefix closure preserves ordering. **(10)**

We will now show that † is satisfied in four parts.

A) Show $\text{Elig}_{\mathcal{L}_j^p}(s') \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z'} \cap \mathcal{I}_j}(s')$

Sufficient to show: $(\forall \sigma \in \Sigma_u)\, s'\sigma \in \mathcal{L}_j^p \Rightarrow s\sigma \in \overline{Z'} \cap \mathcal{I}_j$

Let $\sigma \in \Sigma_u$ and assume $s'\sigma \in \mathcal{L}_j^p$.

$s'\sigma \in \overline{Z'} \cap \mathcal{I}_j$ follows immediately from **(9)** and **(10)**.

B) Show $\text{Elig}_{\mathcal{I}_j}(s') \cap \Sigma_{R_j} \subseteq \text{Elig}_{\mathcal{L}_j^p \cap \overline{Z'}}(s')$

Sufficient to show: $(\forall \rho \in \Sigma_{R_j})\, s'\rho \in \mathcal{I}_j \Rightarrow s'\rho \in \mathcal{L}_j^p \cap \overline{Z'}$

Let $\rho \in \Sigma_{R_j}$ and assume $s'\rho \in \mathcal{I}_j$.

$s'\rho \in \mathcal{L}_j^p \cap \overline{Z'}$ follows immediately from **(9)** and **(10)**.

C) Show $(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\, s'\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s'\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z'} \cap \mathcal{I}_j$

Let $\rho \in \Sigma_{R_j}$, $\alpha \in \Sigma_{A_j}$ and assume $s'\rho\alpha \in \mathcal{I}_j$

$(\exists l \in \Sigma_{L_j}^*)\, s'\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z'} \cap \mathcal{I}_j$ follows immediately from **(9)** and **(10)**.

D) Show $s' \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s'l \in \mathcal{L}_{m_j}^p \cap Z' \cap \mathcal{I}_{m_j}$

Assume $s' \in \mathcal{I}_{m_j}$

$(\exists l \in \Sigma^*_{L_j})\ s'l \in \mathcal{L}^p_{m_j} \cap Z' \cap \mathcal{I}_{m_j}$ follows immediately from **(1)** and **(9)**.

By **Parts A-D**, we can now conclude that † is satisfied.

□

We now are ready to define our fixpoint operator Ω_{L_j}.

Definition 5.3.4 *For system Φ, we define the j^{th}* low level fixpoint operator*, $\Omega_{L_j} : Pwr(\Sigma^*) \rightarrow Pwr(\Sigma^*)$, for arbitrary $Z \in Pwr(\Sigma^*)$ as follows:*

$$\Omega_{L_j}(Z) := \Omega_{LNB_j}(\Omega_{LIC_j}(Z))$$

As operators Ω_{LIC_j} and Ω_{LNB_j} are monotone, it is easy to show that Ω_{L_j} so defined is also monotone.

We next present two useful propositions before we give our main result for this section.

Proposition 11 *Let $Z, Z' \subseteq \Sigma^*$ be arbitrary languages. For system Φ, the follow properties are true:*

1. $Z \subseteq Z' \Rightarrow (\forall i \in \{0, 1, 2, \ldots\})\, \Omega^i_{L_j}(Z) \subseteq \Omega^i_{L_j}(Z')$
2. $\Omega_{L_j}(Z) = Z \ \Rightarrow\ Z \in \mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$
3. *The sequence $\{\Omega^i_{L_j}(\mathcal{Z}_{L_j}),\ i = 0, 1, 2, \ldots\}$ is monotonically decreasing. ie.* $\Omega^{i+1}_{L_j}(\mathcal{Z}_{L_j}) \subseteq \Omega^i_{L_j}(\mathcal{Z}_{L_j})$

Proof

1. Show $Z \subseteq Z' \Rightarrow (\forall i \in \{0, 1, 2, \ldots\})\, \Omega^i_{L_j}(Z) \subseteq \Omega^i_{L_j}(Z')$

Proof identical to the proof in part 1 of Proposition 8, after relabelling.

2. Show $\Omega_{L_j}(Z) = Z \ \Rightarrow\ Z \in \mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$

Assume $\Omega_{L_j}(Z) = Z$. **(3)**

We will show this implies $Z \in \mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$

By definition of $\mathcal{C}_{L_j}$, it is sufficient to show that $Z \subseteq \mathcal{Z}_{L_{j,m}}$ and that Z is LICj with respect to Φ.

By **(3)** and the definition of Ω_{L_j}, we have: $Z = \Omega_{\mathrm{LNB}_j}(\Omega_{\mathrm{LIC}_j}(Z))$

$\Rightarrow Z = [\overline{Z} - \mathrm{Ext}_{\overline{Z}}(\mathrm{FailLIC}_j(\overline{Z}))] \cap \mathcal{Z}_{L_{j,m}}$

Which implies $Z \subseteq \mathcal{Z}_{L_{j,m}}$ and $Z \subseteq [\overline{Z} - \mathrm{Ext}_{\overline{Z}}(\mathrm{FailLIC}_j(\overline{Z}))]$. **(4)**

All that remains is to show that Z is LICj with respect to Φ.

To do this, we first need to show that $\mathrm{FailLIC}_j(\overline{Z}) = \emptyset$. We will do this using proof by contradiction:

Assume $\mathrm{FailLIC}_j(\overline{Z}) \neq \emptyset$.

$\Rightarrow \exists s \in \mathrm{FailLIC}_j(\overline{Z})$ **(5)**

As $\mathrm{FailLIC}_j(\overline{Z}) \subseteq \overline{Z}$ by definition, we can conclude $s \in \overline{Z}$. **(6)**

$\Rightarrow (\exists s' \in \Sigma^*)\, ss' \in Z$ **(7)**

We can also conclude by **(5)** and the definition of the Ext operator that:
$s \in \mathrm{Ext}_{\overline{Z}}(\mathrm{FailLIC}_j(\overline{Z}))$

However, we have by **(7)** and **(4)** that:
$ss' \in \overline{Z} - \mathrm{Ext}_{\overline{Z}}(\mathrm{FailLIC}_j(\overline{Z}))$

$\Rightarrow ss' \notin \mathrm{Ext}_{\overline{Z}}(\mathrm{FailLIC}_j(\overline{Z})) \wedge ss' \in \overline{Z}$

$\Rightarrow (\forall s'' \in \mathrm{FailLIC}_j(\overline{Z}))\, \neg(s'' \leq ss')$

Which contradicts **(5)**.

We thus conclude that $\text{FailLIC}_j(\overline{Z}) = \emptyset$.

$$\Rightarrow (\forall t \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{Z})\,[\text{Elig}_{\mathcal{L}_j^p}(t) \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z}\cap\mathcal{I}_j}(t)]$$
$$\wedge\ [\text{Elig}_{\mathcal{I}_j}(t) \cap \Sigma_{R_j} \subseteq \text{Elig}_{\mathcal{L}_j^p\cap\overline{Z}}(t)]$$
$$\wedge\ [(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\, t\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, t\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z} \cap \mathcal{I}_j]$$
$$\wedge\ [t \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, tl \in \mathcal{L}_{m_j}^p \cap Z \cap \mathcal{I}_{m_j}]$$

Which implies by *Definition* 5.3.1 that Z is LICj with respect to Φ.

We thus have $Z \in \mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$, as required.

3. Show $\Omega_{L_j}^{i+1}(\mathcal{Z}_{L_j}) \subseteq \Omega_{L_j}^{i}(\mathcal{Z}_{L_j})$, for $i = 0, 1, 2, \ldots$.

We will first show that $\Omega_{L_j}^{1}(\mathcal{Z}_{L_j}) \subseteq \Omega_{L_j}^{0}(\mathcal{Z}_{L_j})$, i.e., $\Omega_{L_j}(\mathcal{Z}_{L_j}) \subseteq \mathcal{Z}_{L_j}$.

By definition of Ω_{L_j}, we have:

$$\Omega_{L_j}(\mathcal{Z}_{L_j}) = \Omega_{\text{LNB}_j}(\Omega_{\text{LIC}_j}(\mathcal{Z}_{L_j})) = \Omega_{\text{LIC}_j}(\mathcal{Z}_{L_j}) \cap \mathcal{Z}_{L_{j,m}} \subseteq \mathcal{Z}_{L_{j,m}} \subseteq \mathcal{Z}_{L_j}$$

We thus have $\Omega_{L_j}(\mathcal{Z}_{L_j}) \subseteq \mathcal{Z}_{L_j}$.

This means we can take $Z = \Omega_{L_j}(\mathcal{Z}_{L_j})$, and $Z' = \mathcal{Z}_{L_j}$, and apply *point 1.*

We thus take $i \in \{0, 1, 2, \ldots\}$ and can conclude:

$$\Omega_{L_j}^{i}(\Omega_{L_j}(\mathcal{Z}_{L_j})) \subseteq \Omega_{L_j}^{i}(\mathcal{Z}_{L_j})$$

$\Rightarrow \Omega_{L_j}^{i+1}(\mathcal{Z}_{L_j}) \subseteq \Omega_{L_j}^{i}(\mathcal{Z}_{L_j})$, as required.

□

Proposition 12 *For system Φ, $\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ is the greatest fixpoint of Ω_{L_j}.*

Proof

To prove that $\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ is the greatest fixpoint of Ω_{L_j}, we need to show:

1. $\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) = \Omega_{L_j}(\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}))$

2. $(\forall Z \in \mathrm{Pwr}(\Sigma^*))\, Z = \Omega_{L_j}(Z) \Rightarrow Z \subseteq \mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$

The second part follows from *Point 2* of *Proposition 11*. As every fixpoint is in $\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$, it follows that the fixpoint is $\subseteq \mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ since $\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ is the supremal element of $\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$.

All that is left to show is that $\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ is a fixpoint of Ω_{L_j}.

We first note that by definition of Ω_{L_j} we have:

$\Omega_{L_j}(\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})) = \Omega_{\mathrm{LIC}_j}(\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})) \cap \mathcal{Z}_{L_{j,m}}$ **(1)**

By definition of Ω_{HIC} we have: **(2)**

$\Omega_{L_j}(\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})) = [\overline{\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})} - \mathrm{Ext}_{\overline{\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}}(\mathrm{FailLIC}_j(\overline{\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}))] \cap \mathcal{Z}_{L_{j,m}}$

We now note that as $\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ is LICj with respect to Φ, by definition.

By *Definition* 5.3.1, it thus follows that: $\mathrm{FailLIC}_j(\overline{\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}) = \emptyset$

$\Rightarrow \Omega_{\mathrm{LIC}_j}(\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})) = \overline{\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})} - \emptyset = \overline{\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}$, by definition of the Ext operator.

$\Rightarrow \Omega_{L_j}(\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})) = \overline{\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})} \cap \mathcal{Z}_{L_{j,m}}$, by **(1)**. **(3)**

We are now ready to show $\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) = \Omega_{L_j}(\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}))$.

(I) Show $\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) \subseteq \Omega_{L_j}(\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}))$

By **(3)**, is its sufficient to show that $\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) \subseteq \overline{\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})} \cap \mathcal{Z}_{L_{j,m}}$

We first note that $\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) \subseteq \overline{\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}$, by definition of prefix closure.

Also as $\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ is in $\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$, we have $\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) \subseteq \mathcal{Z}_{L_{j,m}}$.

$\Rightarrow \mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) \subseteq \overline{\mathrm{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})} \cap \mathcal{Z}_{L_{j,m}}$

Part (I) complete.

(II) Show $\Omega_{L_j}(\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})) \subseteq \text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$

Let $s \in \Omega_{L_j}(\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}))$. **(4)**

We will now show this implies $s \in \text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$

From **(4)** and **(2)**, we can conclude that:

$s \in [\overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})} - \text{Ext}_{\overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}}(\text{FailLIC}_j(\overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}))] \cap \mathcal{Z}_{L_{j,m}}$ **(5)**

$\Rightarrow [s \in \overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}] \wedge [s \notin \text{Ext}_{\overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}}(\text{FailLIC}_j(\overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}))]$ **(6)**

$\Rightarrow (\forall s' \leq s) s' \notin \text{FailLIC}_j(\overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})})$ by definition of the Ext operator.

$\Rightarrow (\forall s' \in \overline{\{s\}})\, s' \notin \text{FailLIC}_j(\overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})})$ **(7)**

We next note that $s \in \overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}$ (by **(6)**) implies that $s \in \mathcal{Z}_{L_j}$ as $\mathcal{Z}_{L_j}$ is closed and $\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) \subseteq \mathcal{Z}_{L_{j,m}} \subseteq \mathcal{Z}_{L_j}$, thus $\overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})} \subseteq \overline{\mathcal{Z}_{L_{j,m}}} \subseteq \overline{\mathcal{Z}_{L_j}}$ as prefix closure respects ordering.

We thus have $s \in \mathcal{L}_j^p \cap \mathcal{I}_j$ by definition of $\mathcal{Z}_{L_j}$.

$\Rightarrow s \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}$

$\Rightarrow (\forall s' \in \overline{\{s\}})\, s' \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}$, as all three languages are closed.

Combining with **(7)**, we can conclude that for all $s' \in \overline{\{s\}}$, the following: **(8)**

1. $s' \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}$
2. $\text{Elig}_{\mathcal{L}_j^p}(s') \cap \Sigma_u \subseteq \text{Elig}_{\overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})} \cap \mathcal{I}_j}(s')$
3. $\text{Elig}_{\mathcal{I}_j}(s') \cap \Sigma_{R_j} \subseteq \text{Elig}_{\mathcal{L}_j^p \cap \overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}}(s')$
4. $(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\, s'\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s'\rho l\alpha \in \mathcal{L}_j^p \cap \overline{\text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})} \cap \mathcal{I}_j$
5. $s' \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, s'l \in \mathcal{L}_{m_j}^p \cap \text{sup}\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) \cap \mathcal{I}_{m_j}$

Let $Z = \sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) \cup \{s\}$ **(9)**

We will now show that Z is in $\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$, which will imply $Z \subseteq \sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$, giving us the needed result.

We first note that by **(9)**, we have:

$\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) \subseteq Z$

$\Rightarrow \overline{\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})} \subseteq \overline{Z}$, as prefix closure preserves ordering. **(10)**

We next note that we have $\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) \subseteq \mathcal{Z}_{L_{j,m}}$ by definition, and by **(5)** we have $s \in \mathcal{Z}_{L_{j,m}}$

We thus have $Z \subseteq \mathcal{Z}_{L_{j,m}}$.

To show that Z is in $\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$, all that now remains is to demonstrate that Z is LICj with respect to system Φ.

Let $t \in \mathcal{L}_j^p \cap \mathcal{I}_j \cap \overline{Z}$ **(11)**

We will now show that the following conditions are satisfied:

1. $\text{Elig}_{\mathcal{L}_j^p}(t) \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z}\cap\mathcal{I}_j}(t)$
2. $\text{Elig}_{\mathcal{I}_j}(t) \cap \Sigma_{R_j} \subseteq \text{Elig}_{\mathcal{L}_j^p\cap\overline{Z}}(t)$
3. $(\forall\rho \in \Sigma_{R_j})(\forall\alpha \in \Sigma_{A_j})\, t\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, t\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z} \cap \mathcal{I}_j$
4. $t \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, tl \in \mathcal{L}_{m_j}^p \cap Z \cap \mathcal{I}_{m_j}$

1) Show $\text{Elig}_{\mathcal{L}_j^p}(t) \cap \Sigma_u \subseteq \text{Elig}_{\overline{Z}\cap\mathcal{I}_j}(t)$

Let $\sigma \in \Sigma_u$, and $t\sigma \in \mathcal{L}_j^p$.

Sufficient to show implies $t\sigma \in \overline{Z} \cap \mathcal{I}_j$.

If $t \in \overline{Z} - \overline{\{s\}}$, we have $t \in \overline{\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}$.

As $\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ is LICj for Φ, it follows that $t\sigma \in \overline{\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})} \cap \mathcal{I}_j$.

$\Rightarrow t\sigma \in \overline{Z} \cap \mathcal{I}_j$, by **(10)**.

If $t \in \overline{\{s\}}$, it follows directly from **(8)** and **(10)**.

2) Show $\text{Elig}_{\mathcal{I}_j}(t) \cap \Sigma_{R_j} \subseteq \text{Elig}_{\mathcal{L}_j^p \cap \overline{Z}}(t)$

Let $\rho \in \Sigma_{R_j}$ and $t\rho \in \mathcal{I}_j$

Sufficient to show implies $t\rho \in \mathcal{L}_j^p \cap \overline{Z}$.

If $t \in \overline{Z} - \overline{\{s\}}$, we have $t \in \overline{\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}$.

As $\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ is LICj for Φ, it follows that $t\rho \in \mathcal{L}_j^p \cap \overline{\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}$.

$\Rightarrow t\rho \in \mathcal{L}_j^p \cap \overline{Z}$, by **(10)**.

If $t \in \overline{\{s\}}$, it follows directly from **(8)** and **(10)**.

3) Show $(\forall\rho \in \Sigma_{R_j})(\forall\alpha \in \Sigma_{A_j})\, t\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, t\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z} \cap \mathcal{I}_j$

Let $\rho \in \Sigma_{R_j}$, $\alpha \in \Sigma_{A_j}$, and $t\rho\alpha \in \mathcal{I}_j$.

We will now show this implies $(\exists l \in \Sigma_{L_j}^*)\, t\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z} \cap \mathcal{I}_j$

If $t \in \overline{Z} - \overline{\{s\}}$, we have $t \in \overline{\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}$.

As $\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ is LICj for Φ, it follows that:
$(\exists l \in \Sigma_{L_j}^*)\, t\rho l\alpha \in \mathcal{L}_j^p \cap \overline{\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})} \cap \mathcal{I}_j$.

$\Rightarrow t\rho l\alpha \in \mathcal{L}_j^p \cap \overline{Z} \cap \mathcal{I}_j$, by **(10)**.

If $t \in \overline{\{s\}}$, it follows directly from **(8)** and **(10)**.

4) Show $t \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma_{L_j}^*)\, tl \in \mathcal{L}_{m_j}^p \cap Z \cap \mathcal{I}_{m_j}$

Assume $t \in \mathcal{I}_{m_j}$.

We will now show this implies $(\exists l \in \Sigma^*_{L_j})\ tl \in \mathcal{L}^p_{m_j} \cap Z \cap \mathcal{I}_{m_j}$

If $t \in \overline{Z} - \overline{\{s\}}$, we have $t \in \overline{\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})}$.

As $\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ is LICj for Φ, it follows that:
$(\exists l \in \Sigma^*_{L_j})\ tl \in \mathcal{L}^p_{m_j} \cap \sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) \cap \mathcal{I}_{m_j}$.

$\Rightarrow tl \in \mathcal{L}^p_{m_j} \cap Z \cap \mathcal{I}_{m_j}$, by **(9)**.

If $t \in \overline{\{s\}}$, it follows directly from **(8)** and **(10)**.

We can now conclude by *points 1-4* that Z is LICj with respect to system Φ.

$\Rightarrow Z \subseteq \sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$, as $\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ is the supremal element for $\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$

$\Rightarrow s \in \sup\mathcal{C}_H(\mathcal{Z}_{H_m})$ (by **(9)**) , as required.

Part (II) complete.

By (I) and (II), we get $\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) = \Omega_{L_j}(\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}))$ as required.

We thus conclude that $\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ is the greatest fixpoint of Ω_{L_j}.

□

We will now show that if $\Omega_{L_j}(\mathcal{Z}_{L_j})$ reaches a fixpoint after a finite number of steps, then that fixpoint is our supremal element. In Chapter 6, we will give an automata based algorithm that implements $\Omega_{L_j}(\mathcal{Z}_{L_j})$. As the algorithm operates by removing one or more states of $\mathbf{G}_{\mathrm{LL}_j}$ which is assumed to have a finite state space, we know it will complete in a finite number of steps (ie. it must stop when we have no more states left to remove).

Theorem 9 *For system Φ, if there exists $i \in \{0, 1, 2, \ldots\}$ such that $\Omega^i_{L_j}(\mathcal{Z}_{L_j})$ is a fixpoint, then $\Omega^i_{L_j}(\mathcal{Z}_{L_j}) = sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$.*

Proof

Assume $\exists i \in \{0, 1, 2, \ldots\}$, such that $\Omega_{L_j}(\Omega^i_{L_j}(\mathcal{Z}_{L_j})) = \Omega^i_{L_j}(\mathcal{Z}_{L_j})$ **(1)**

We first note that we have: $\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) \subseteq \mathcal{Z}_{L_{j,m}} \subseteq \mathcal{Z}_{L_j}$

This allows us to apply *Point 1* of *Proposition 11* and conclude:

$\Omega^i_{L_j}(\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})) \subseteq \Omega^i_{L_j}(\mathcal{Z}_{L_j})$ **(3)**

By *Proposition 12*, we know that $\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ is the greatest fixpoint of Ω_{L_j}. **(4)**

$\Rightarrow \Omega^i_{L_j}(\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})) = \sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$

Combine this with **(3)**, and we can conclude:

$\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) \subseteq \Omega^i_{L_j}(\mathcal{Z}_{L_j})$ **(5)**

As $\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ is the greatest fixpoint of Ω_{L_j} (by **(4)**) and $\Omega^i_{L_j}(\mathcal{Z}_{L_j})$ is a fixpoint, it thus follows: $\Omega^i_{L_j}(\mathcal{Z}_{L_j}) \subseteq \sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$

By **(5)**, we thus have $\Omega^i_{L_j}(\mathcal{Z}_{L_j}) = \sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ as required.

□

We now show that we can use $\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ for our j^{th} low level supervisor and satisfy the relevant interface conditions. We will use $S_{L_{j,m}} \subseteq \Sigma^*$ to stand for the marked language of the j^{th} low level supervisor.

Corollary 3 *For system Φ, if there exists $i \in \{0, 1, 2, \ldots\}$ such that $\Omega^i_{L_j}(\mathcal{Z}_{L_j})$ is a fixpoint, then system Φ with $S_{L_{j,m}} = \Omega^i_{L_j}(\mathcal{Z}_{L_j})$ and $S_{L_j} = \overline{S_{L_{j,m}}}$ satisfies Points 4, 5, and 6 of Definition 3.4.2, Point II of Definition 3.5.1 and Point II of Definition 3.6.1.*

Proof

Assume $\exists i \in \{0, 1, 2, \ldots\}$, such that $\Omega_{L_j}(\Omega^i_{L_j}(\mathcal{Z}_{L_j})) = \Omega^i_{L_j}(\mathcal{Z}_{L_j})$. **(1)**

Let $S_{L_{j,m}} = \Omega^i_{L_j}(\mathcal{Z}_{L_j})$ and $S_{L_j} = \overline{S_{L_{j,m}}}$.

By Theorem 9, $S_{L_{j,m}} = \sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ is LIC_j with respect to Φ. **(2)**

By Definition 5.3.1 and using the fact that $S_{L_j} = \overline{S_{L_{j,m}}}$, we have for all $s \in \mathcal{L}^p_j \cap \mathcal{I}_j \cap S_{L_j}$

1. $\text{Elig}_{\mathcal{L}^p_j}(s) \cap \Sigma_u \subseteq \text{Elig}_{S_{L_j} \cap \mathcal{I}_j}(s)$ **(3)**
2. $\text{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{R_j} \subseteq \text{Elig}_{\mathcal{L}^p_j \cap S_{L_j}}(s)$ **(4)**
3. $(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})$
   $s\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma^*_{L_j})\, s\rho l\alpha \in \mathcal{L}^p_j \cap S_{L_j} \cap \mathcal{I}_j$ **(5)**
4. $s \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma^*_{L_j})\, sl \in \mathcal{L}^p_{m_j} \cap S_{L_{j,m}} \cap \mathcal{I}_{m_j}$ **(6)**

We immediately note that Point II of Definition 3.6.1 follows immediately from **(3)**.

We next note that we can use the fact that $\mathcal{L}_j = \mathcal{L}^p_j \cap S_{L_j}$, and $\mathcal{L}_{m_j} = \mathcal{L}^p_{m_j} \cap S_{L_{j,m}}$ to rewrite **(3)-(6)** as for all $s \in \mathcal{L}_j \cap \mathcal{I}_j$

1. $\text{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{R_j} \subseteq \text{Elig}_{\mathcal{L}_j}(s)$ **(7)**
2. $(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})$
   $s\rho\alpha \in \mathcal{I}_j \Rightarrow (\exists l \in \Sigma^*_{L_j})\, s\rho l\alpha \in \mathcal{L}_j \cap \mathcal{I}_j$ **(8)**
3. $s \in \mathcal{I}_{m_j} \Rightarrow (\exists l \in \Sigma^*_{L_j})\, sl \in \mathcal{L}_{m_j} \cap \mathcal{I}_{m_j}$ **(9)**

We now note that Points 4, 5, and 6 of Definition 3.4.2 follow immediately from **(7)-(9)**, respectively.

All that remains is to show that Point I of Definition 3.5.1 is satisfied.

This means showing that $\overline{\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j}} = \mathcal{L}_j \cap \mathcal{I}_j$

By **(2)**, we have $S_{L_{j,m}} = \sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$.

$\Rightarrow S_{L_{j,m}} \subseteq \mathcal{Z}_{L_{j,m}}$, as $\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}}) \subseteq \mathcal{Z}_{L_{j,m}}$ by definition. **(10)**

$\Rightarrow S_{L_{j,m}} \subseteq \mathcal{Z}_{L_j}$, as $\mathcal{Z}_{L_{j,m}} \subseteq \mathcal{Z}_{L_j}$

$\Rightarrow \overline{S_{L_{j,m}}} \subseteq \mathcal{Z}_{L_j}$, as $\mathcal{Z}_{L_j}$ is closed and prefix closure preserves ordering.

$\Rightarrow S_{L_j} \subseteq \mathcal{Z}_{L_j}$, by definition of S_H. **(11)**

Substituting for $\mathcal{Z}_{L_{j,m}}$ in **(10)**, we get $S_{L_{j,m}} \subseteq \mathcal{L}^p_{m_j} \cap \mathcal{E}_{L_{j,m}} \cap \mathcal{I}_{m_j}$. **(12)**

Substituting for $\mathcal{Z}_{L_j}$ in **(11)**, we get $S_{L_j} \subseteq \mathcal{L}^p_j \cap \mathcal{E}_{L_j} \cap \mathcal{I}_j$. **(13)**

Using the fact that $\mathcal{L}_{m_j} = \mathcal{L}^p_{m_j} \cap S_{L_{j,m}}$, we get $\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j} = \mathcal{L}^p_{m_j} \cap S_{L_{j,m}} \cap \mathcal{I}_{m_j}$.

$\Rightarrow \mathcal{L}_{m_j} \cap \mathcal{I}_{m_j} = S_{L_{j,m}}$, by **(12)**. **(14)**

Using the fact that $\mathcal{L}_j = \mathcal{L}^p_j \cap S_{L_j}$, we get $\mathcal{L}_j \cap \mathcal{I}_j = \mathcal{L}^p_j \cap S_{L_j} \cap \mathcal{I}_j$.

$\Rightarrow \mathcal{L}_j \cap \mathcal{I}_j = S_{L_j}$, by **(13)**.

As $S_{L_j} = \overline{S_{L_{j,m}}}$, by definition, it follows from **(9)** that $\overline{\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j}} = \mathcal{L}_j \cap \mathcal{I}_j$, as required.

□

We have now shown that $\sup\mathcal{C}_H(\mathcal{Z}_{H_m})$ exists and for all $j \in \{1, \ldots, n\}$, $\sup\mathcal{C}_{L_j}(\mathcal{Z}_{L_{j,m}})$ exists. We have also given fixpoint operators for each that allow us to construct them. In Chapter 6, we will present out automata based algorithms that implements our fixpoint operators. We will then tie everything together and present our overall synthesis results for system Φ.

Chapter 6

Algorithms

Our goal is to construct a supervisor for the high level, and one for each low level based on a set of specifications for each level, such that the supervisor will satisfy the corresponding HISC conditions by design, and will be maximally permissive for its level.

We will first give a few common data structures and algorithms used in this chapter and then present our algorithms. We give an algorithm to verify whether a given interface is a command-pair interface or not, an algorithm to check that a given parallel system satisfies the interface consistency condition, and finally, a set of algorithms to construct a HIC supervisor for the high level, and a LICj supervisor for each low level. We first give pseudo code to present the algorithms and then we provide a time complexity analysis.

6.1 Common Data Structures and Algorithms

Before bringing in algorithms for the interface system, we first discuss a few data structures and algorithms that will be commonly used in the algorithms given in the following sections.

6.1.1 DES

The data structure of a DES is designed to handle all elements of a DES, with access functions provided for the data members. The DES data structure has the following member structures:

- *states* : all states of the DES. *states* can be implemented as an array or linked list. Since we are building a DES with the number of states unknown at the beginning, a linked list is used. A traversal of the whole state space takes linear time.
- *marker_states* : all marker states of the DES. It is implemented as a linked list in our algorithms.
- *initial_state* : the initial state of the DES. Since the initial state is important, we have a pointer to it for fast access.
- *events* : A linked list off all events that belong to the event set of the DES.

A state consists of:

- *index* : integer starting from 1 as the unique key to identify a state. Since each state has a unique integer index, it's very easy to construct an array of states and get fast access to each state and their properties.
- *trans* : transition list, including all transitions starting from this state. This is a linked list. A *transition* consists of an event and a state, and is always associated with its source state (the state that transition starts from).
- *inverseTrans* : inverse transition list, including all transitions ending with the state. It consists of the event, and the source state. This is designed to provide fast lookup for synthesis and while checking properties such as nonblocking. It is implemented as a linked list.

An *event* consists of a unique index, starting from 1. An event has a property *event type* that carries the value A or R if the event is an answer or request event, or value N otherwise. Checking the event type of an event takes constant time. An event also has a boolean flag $isControl$ which is true if an event is controllable. As a result, checking if an event is in Σ_u or Σ_c takes constant time.

Transitions are frequently accessed to determine if an event is defined at a state when doing a synthesis or checking conditions. For fast access, we create a *transition matrix*. The matrix is a three dimension array, where a location is determined by an index for the DES, its states and the possible events. We can determine if a DES has a transition at a given state for a given event by checking the corresponding location in the array. If the value stored is zero, then the transition is undefined. Otherwise, the index of the target state (the state the transition takes us to) is stored at this location; thus getting the next state of a transition or checking whether a transition is defined takes constant time.

6.1.2 Functions

In this section we list functions that are used in our algorithms and access functions to our data structures.

- $\lfloor x \rfloor$

  The function $floor(x)$ ($\lfloor x \rfloor$) gets the largest integer that's smaller than the given number x. The function takes constant time.

- *pop*

  Is similar to a stack 'pop' function. In our algorithms we used it on our pending list and some other linked lists.[1] It takes an element from the list as output and removes that element from the list. The function takes constant time.

[1] the pending list used in our algorithms is defined in Section 6.1.3.

- *push*

  Is similar to a stack 'push' function. It appends an element to the given list. The function takes constant time.

- *addState*

  A DES object can call the *addState* function to add a state to its state list. It also adds the state to the marker_states list if the state is marked. Both state list and marker_states are linked lists and this function takes constant time.

- $addInverseTrans$

  Given a source state and an event, the function adds a new inverse transition to the calling state's inverse transition list ($inverseTrans$). This means there exists a transition from the source state, labeled by the event, leading to the calling state. Since $inverseTrans$ is a linked list structure, this function takes constant time.

- $removeInverseTrans$

  It removes a transition from a state's $inverseTrans$ list. This function takes $O(n_\Sigma n_X)$, where n_Σ is the number of events of the DES, and n_X is the number of states. We note that $n_\Sigma n_X$ is actually the upper bound for the number of reverse transitions for the entire DES (ie. if we added up the reverse transitions for each state, they can't exceed this number for a deterministic DES). Where it is possible that we can have this many reverse transitions at a given state, it would mean there would be none at all the other states! In particular, if we were looping through all n_X states, and examining all reverse transitions, it would appear we would have $n_\Sigma n_X^2$ steps, when in actually this can not exceed $n_\Sigma n_X$.

- $addTrans$

  Given a source state and an event, add a new transition for the calling state's

transition list ($trans$). This means there exists a transition from the calling state, labeled by the event, leading to the source state. The transition list is a linked list structure, this function takes constant time.

- $removeTrans$

  Same as $removeInverseTrans$, this function takes linear time, $O(n_\Sigma)$.

6.1.3 Pending and Found List

When applying a synthesis algorithm to a subsystem consisting of m DES, we start from the synchronous product of the m DES, then trim off states that don't satisfy certain properties such as controllability.

Let n_i be the state size of the i^{th} component DES, $i = 1, 2, \ldots m$; let N_X be the upper bound of all n_i (ie. $n_i \leq N_X$ for each i). When we do a synchronous product, the state space is worst case exponential in the size of an individual DES (i.e. $O(N_X^m)$).

While constructing the synchronous product or verifying certain properties, we often need to maintain a pending list. This list contains items remaining to be processed. We also usually need to maintain a found list which contains items already added to the pending list. Note: the found list contains items already encountered, but they may not still been in the pending list if they have already been processed.

The pending list operates like a work pool. We we need a new item to process, we take it from this list. When we encounter a new item which is not in the found list, we add it to the pool and to our found list. The order of the items in the pending list doesn't matter to the final result, so we can thus use either a stack or a queue as an implementation. In either case, adding or removing items from the pending list can be done in constant time.

We next discuss the found list. When constructing the synchronous product, we are essentially doing a traversal of all reachable states in the cross product of the DES. When we encounter a state, which is a tuple of m component states from our m DES, we need to search the found list to check whether we have encountered the tuple already. If the tuple hasn't been "found" yet, we add it to our found list, and define a state variable for the tuple. Operations like insert and search are thus frequently required. To provide good performance in these operations and to store the tuple information efficiently, we adapt the *trie* structure to store the found list. A trie is a multi-branch tree with certain properties that we will define below. See Figure 6.1 for an example.

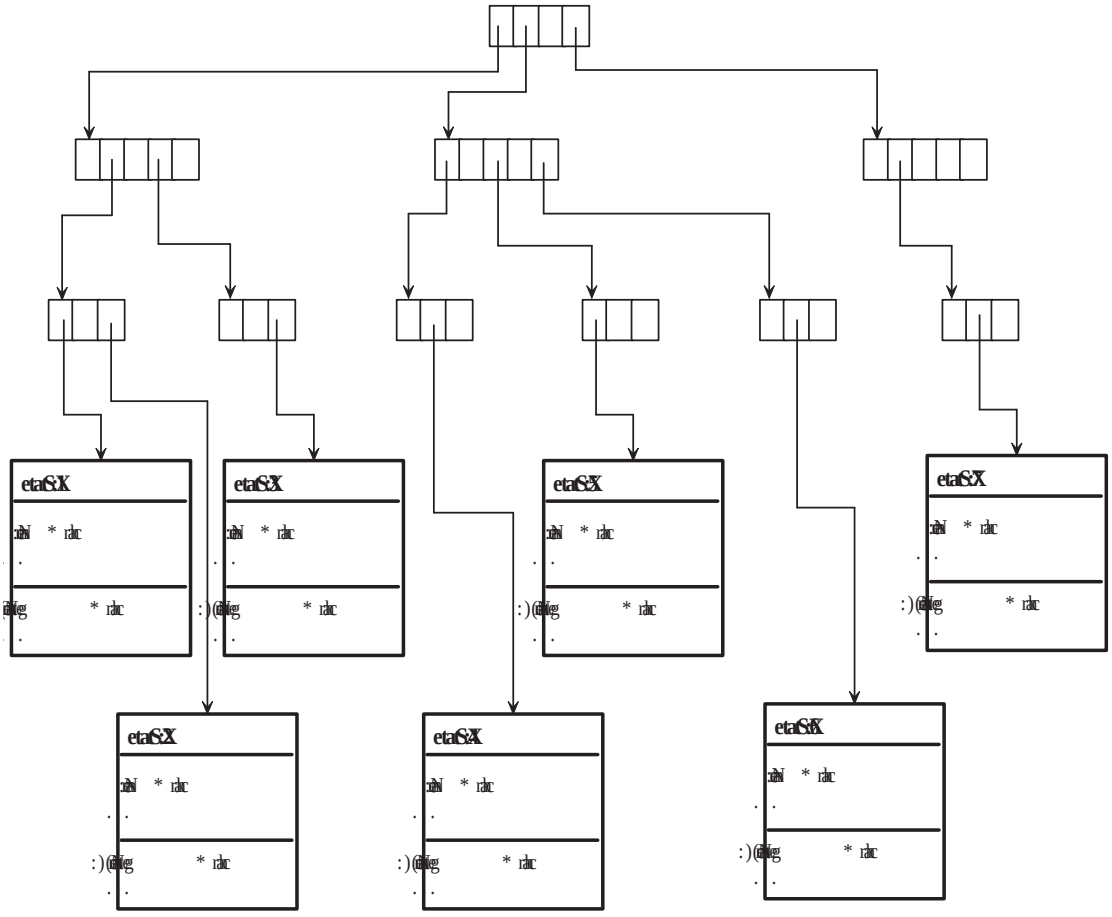

Figure 6.1: Trie Illustration

Since a state in a synchronous product is based on the component states from

the m component DES, we represent it as a m-tuple such as $(x_1, x_2, ..., x_m)$. When we have m component DES, we will use a trie of height m. Each level represents a DES, with DES 1 represented by the root node, DES 2 by the level below the root node, and so on. In other words, the nodes at level $i-1$ represent DES i.

The nodes at a level i of the trie consist of an array of pointers. The size of the array is n_{i+1}, the state size of DES $i+1$. This means that such a node can have n_{i+1} children. For nodes at level other than $m-1$, the elements of the array point to the next level of the trie, or contain the NULL pointer depending on which tuples have been stored already. We will make this clearer in a moment by discussing an example. For nodes at level $m-1$, the array elements contain either pointers to state variables, or the NULL pointer depending on which tuples have been stored in the trie already.

For example, assume we have a subsystem with $m = 3$ DES: $\mathbf{G_1}$, $\mathbf{G_2}$ and $\mathbf{G_3}$. We will also assume they have state spaces of size 4 (ie. states 1, 2, 3, 4, and 5), 5 and 3, respectively. Figure 6.1 contains a trie that could correspond to such a system. Say we encountered state tuple $(1, 2, 2)$ and we wanted to determine if it was already present in the trie. We would first check position one of the array at the root node. If it is the NULL pointer, then that means no tuples with a "1" in the first position have yet been added to the trie. In Figure 6.1, it so happens that we have a pointer at position one that leads us to a node at level one. We next check position two of this node, and again find a pointer to the next level. We follow the pointer to the node at level two, and check position to of the array. If the tuple has already been added, we will find a pointer to the state variable that represents the tuple. For our example, we find the NULL pointer meaning the tuple is not present in the trie. Given a state in the synchronous product, we can thus look it up in the trie in $O(m)$ steps, where m is the number of DES in the subsystem.

If a tuple has not already been added to the trie, we allocate the missing nodes

and a new state structure, and set the pointers appropriately. This means worst case allocating memory for m items, setting one pointer at level 0, followed by setting $\Sigma_{k=2,3,\ldots,m} n_k \leq (m-1)N_X)$ pointers (including initializing unused ones to NULL). Adding a state is thus $O(m+1+(m-1)N_X) = O(m(1+N_X)+1-N_X)$. If we take N_X as a constant, we get $O(m)$.

The trie structure used for states has a fixed height. We also used a variable height version of the trie structure to keep track of events. For a detailed discussion of variable height trie structures, see [32].

6.1.4 Disjoint Union

When checking command-pair interface properties and other conditions, we need to frequently verify that two or more sets are disjoint. We also often have to verify whether a set is equal to the disjoint union of two or more sets. In the format

$$S = S_1 \dot{\cup} S_2$$

We need to check two properties

1. $S = S_1 \cup S_2$
2. $S_1 \cap S_2 = \emptyset$

For fast processing, we store the sets as arrays. We thus need to make sure there are no duplicate elements in the array. In the following algorithms, we will use integer elements for demonstration purposes. In order to check whether two sets are disjoint or not, we can simply put the two sets together and sort the results. If the two sets are disjoint, there should be no duplicate elements in the resulting set. We first present the uMergeSort algorithm in Listing 6.1. The algorithm is a variation of Merge-Sort [19].

To present uMergeSort, we first must present the algorithm for uMerge, which uMergeSort uses. Algorithm uMerge takes four parameters: an input array A, start and end indexes p and r, and a middle index q. Array A is a place holder for the input and output arrays. The algorithm takes two sub-arrays from A (i.e. $A[p..q]$ and $A[(q+1)..r]$) and then merges the two sub-arrays into one array, with the elements in ascending order. We use two array variables, L and R, to hold the two sub-arrays in the algorithm. We also will use the notation ∞ to represent a number that is bigger than any possible value in the array.

Algorithm uMergeSort takes three parameters: A as the input array,as well as b and e as the indices of the first and last elements in the array.

Listing 6.1: Modified Merge sort

```
1  bool uMerge(A, p, q, r)
2  begin
3      n1 ← q−p+1;
4      n2 ← r−q;
5      L[1..n1] ← A[p..q];
6      L[n1+1] ← ∞;
7      R[1..n2] ← A[(q+1)..r];
8      R[n2+1] ← ∞;
9      i ← 1;
10     j ← 1;
11     for k ← p to r do
12         if L[i] = R[j] then
13             return false;
14         else if L[i] < R[j] then
15             A[k] ← L[i++];
16         else
17             A[k] ← R[j++];
18         end if
```

```
19      end for
20      return true;
21  end
22
23  bool    uMergeSort  (A, b, e)
24  begin
25      if b<e then
26          m ← ⌊(b+e)/2⌋;
27          if uMergeSort(A, b, m) = false or
28              uMergeSort(A, m+1, e) = false or
29              uMerge(A, b, m, e) = false then
30              return false;
31          end if
32      end if
33      return true;
34  end
```

We modified Merge-Sort such that when two items are found equal, the algorithm will immediately terminate and return false. The running time of uMerge-Sort is same as Merge-Sort, i.e., $O(n\log n)$, where n is the total number of elements in the input array.

The disjoint union checking algorithm is given in Listing 6.2. It takes as input three sets A_1, A_2 and S and it checks if S is equal to the disjoint union of A_1 and A_2. We use the array variable A_0 as temporary storage.

Listing 6.2: Verify disjoint union

```
1  bool DisjointUnion (A1, A2, S)
2  begin
3      n1 ← size of A1;
4      n2 ← size of A2;
5      n3 ← size of S;
```

```
6       if n3 ≠ n1 + n2 then
7           return false;
8       end if
9       A0[1..n1] ←A1[1..n1];
10      A0[n1+1..n3] ←A2[1..n2];
11      if uMergeSort(A0, 1, n3) = false or
12          uMergeSort(S, 1, n3) = false then
13          return false;
14      end if
15      for i ← 1 to n3 do
16          if A0[i] ≠ S[i] then
17              return false;
18      end for
19      return true;
20  end
```

We first check to make sure that the size of S is equal to the sum of the sizes of the two member sets (ie. $|S| = |A_1| + |A_2|$: lines 3-8). We the copy the two member sets into a larger set A_0 (lines 9-10). This takes linear time. Next, we call Algorithm 6.1 to check whether the two member sets have duplicate elements (line 11). We then Perform the same check on S (line 12). These checks each take $O(n_3 \log n_3)$. We then traverse the two sorted sets and check whether each corresponding element is equal (lines 15-18). This takes linear time. The running time of this algorithm is thus dominated by uMergeSort, which is $O(n_3 \log n_3)$.

6.1.5 Language vs. States

The synthesis process starts with creating a cross product of the component DES. Based on this new DES, our algorithms trim off states that represent strings that don't meet our requirements, such as controllability and interface consistency.

In Chapter 5, we presented a set of language based fixpoint operators to construct our supremal languages. The algorithms we present in this chapter construct DES that represent these supremal languages, but they operate by removing states, instead of strings. In this section, we will show the equivalence of removing states to removing strings that fail our language based definition.

Let DES $\mathbf{G_i} := (Q_i, \Sigma, \delta_i, q_{oi}, Q_{mi})$, $i = 1, 2$. A *cross product* of DES $\mathbf{G_1}, \mathbf{G_2}$ is defined as $\mathbf{G_1} \times \mathbf{G_2} := (Q, \Sigma, \delta, q_o, Q_m)$, where $Q = Q_1 \times Q_2, \delta = \delta_1 \times \delta_2, q_o = (q_{o1}, q_{o2})$, and $Q_m = Q_{m1} \times Q_{m2}$, with

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

whenever $\delta_1(q_1, \sigma)!$ and $\delta_2(q_2, \sigma)!$. The *meet* of the two DES is the reachable sub-DES of $\mathbf{G_1} \times \mathbf{G_2}$ ([79]).

We now extend the cross product definition to multiple DES. Given DES $\mathbf{G_i} := (Q_i, \Sigma, \delta_i, q_{oi}, Q_{mi})$, $i = 1, 2, \ldots, n$; a *cross product* of DES $\mathbf{G_1}, \mathbf{G_2}, \ldots, \mathbf{G_n}$ is defined as $\mathbf{G_1} \times \mathbf{G_2} \times \ldots \times \mathbf{G_n} := (Q, \Sigma, \delta, q_o, Q_m)$, where

$$\begin{aligned} Q &= Q_1 \times Q_2 \times \ldots \times Q_n \\ \delta &= \delta_1 \times \delta_2 \times \ldots \times \delta_2 \\ q_o &= (q_{o1}, q_{o2}), \ldots, q_{on}) \\ Q_m &= Q_{m1} \times Q_{m2} \times \ldots \times Q_{mn} \end{aligned}$$

with

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

whenever for all $i = 1, 2, \ldots, n$, $\delta_i(q_i, \sigma)!$. Again, the *meet* of the n DES is the reachable sub-DES of $\mathbf{G_1} \times \mathbf{G_2} \times \ldots \times \mathbf{G_n}$.

In the HISC definitions, we used the synchronous product. The main difference between the *meet* and the synchronous product is that all DES combined in the *meet* must have the same event set, where in the synchronous product, they each

can be defined over a different event set. As it's easier to work with the meet, we can add appropriate selfloops to each DES so that they are then defined over a common event set.

We do this as follows. Let DES $\mathbf{G_i} = (Q_i, \Sigma_i, \delta_i, q_{oi}, Q_{mi})$, $\Sigma = \bigcup_{j=1,\ldots,n} \Sigma_j$, $P_i : \Sigma^* \rightarrow \Sigma_i^*$, $i = 1, 2, \ldots n$. We then define new DES $\mathbf{G'_i} = selfloop(\mathbf{G_i}, \Sigma - \Sigma_i)$. We thus have $L(\mathbf{G'_i}) = P_i^{-1}L(\mathbf{G_i})$ and $L_m(\mathbf{G'_i}) = P_i^{-1}L_m(\mathbf{G_i})$. It then follows that $L(meet(\mathbf{G'_1}, \mathbf{G'_2}, ...\mathbf{G'_n})) = L(\mathbf{G_1}\|\mathbf{G_2}\|...\|\mathbf{G_n})$ and $L_m(meet(\mathbf{G'_1}, \mathbf{G'_2}, ...\mathbf{G'_n})) = L_m(\mathbf{G_1}\|\mathbf{G_2}\|...\|\mathbf{G_n})$.

We will now present a useful relationship between the states of $\mathbf{G}$, and the Nerode equivalence relations for the individual $\mathbf{G_i}$. The proposition below states that for any two strings that go to the same state in $\mathbf{G}$, then these two strings also lead to the same state in each of the component DES, and thus they belong to the same nerode equivalent classes of the component DES's closed and marked languages.

Proposition 13 *Let DES* $\mathbf{G_i} := (Q_i, \Sigma, \delta_i, q_{oi}, Q_{mi})$ *(*$i = 1, 2$*) and* $\mathbf{G} = \mathbf{G_1} \times \mathbf{G_2} \times \ldots \times \mathbf{G_n} = (Q, \Sigma, \delta, q_o, Q_m)$. *It then follows:*

$$(\forall i \in \{1, 2, \ldots n\})(\forall s, t \in \Sigma^*)\ \delta(q_o, s) = \delta(q_o, t) \ \Rightarrow\ s \equiv_{L(\mathbf{G_i})} t \ \wedge\ s \equiv_{L_m(\mathbf{G_i})} t$$

$\Diamond$

Proof

Let $s, t \in \Sigma^*$, and $i \in \{1, 2, \ldots n\}$.

Assume $\delta(q_o, s) = \delta(q_o, t)$. **(1)**

We will now show this implies $s \equiv_{L(\mathbf{G_i})} t$ and $s \equiv_{L_m(\mathbf{G_i})} t$.

From **(1)** we have

$$(\delta_1 \times \delta_2 \times \ldots \times \delta_n)((q_{o1}, q_{o2}, \ldots, q_{on}), s) = (\delta_1 \times \delta_2 \times \ldots \times \delta_n)((q_{o1}, q_{o2}, \ldots, q_{on}), t)$$

$\Rightarrow (\delta_1(q_{o1}, s), \delta_2(q_{o2}, s), \ldots, \delta_n(q_{on}, s)) = (\delta_1(q_{o1}, s), \delta_2(q_{o2}, s), \ldots, \delta_n(q_{on}, t))$

$\Rightarrow \delta_i(q_{oi}, s) = \delta_i(q_{oi}, t)$.

$\Rightarrow s \equiv_{L(\mathbf{G_i})} t \wedge s \equiv_{L_m(\mathbf{G_i})} t$ (from [79]), as required.

□

We now present an analogous result for the synchronous product of n DES. Let DES $\mathbf{G_i} = (Q_i, \Sigma_i, \delta_i, q_{oi}, Q_{mi})$, $\Sigma = \bigcup_{j=1,\ldots,n} \Sigma_j$, $P_i : \Sigma^* \rightarrow \Sigma_i^*$, $i = 1, 2, \ldots n$. Define $\mathbf{G} = \mathbf{G_1} \| \mathbf{G_2} \| ... \| \mathbf{G_n} = (Q, \Sigma, \delta, q_o, Q_m)$, $\mathcal{L}_i := P_i^{-1} L(\mathbf{G_i})$, and $\mathcal{L}_{m,i} := P_i^{-1} L_m(\mathbf{G_i})$.

Proposition 14 *Let* $\mathbf{G_i}$ *(*$i = 1, 2, \ldots, n$*),* $\mathbf{G}$*,* $\mathcal{L}_i$*, and* $\mathcal{L}_{m,i}$ *be fined as above. It then follows:*

$$(\forall i \in \{1, 2, \ldots, n\})(\forall s, t \in \Sigma^*)\ \delta(q_o, s) = \delta(q_o, t) \ \Rightarrow\ s \equiv_{\mathcal{L}_i} t \ \wedge\ s \equiv_{\mathcal{L}_{m,i}} t$$

◊

Proof

Let $s, t \in \Sigma^*$, and $i \in \{1, 2, \ldots, n\}$.

Assume $\delta(q_o, s) = \delta(q_o, t)$. **(1)**

We will now show this implies $s \equiv_{\mathcal{L}_i} t \ \wedge\ s \equiv_{\mathcal{L}_{m,i}} t$.

We now define new DES $\mathbf{G'_j} = selfloop(\mathbf{G_j}, \Sigma - \Sigma_j)$ $(j = 1, 2, \ldots n)$.

We thus have $L(\mathbf{G'_j}) = P_j^{-1} L(\mathbf{G_j}) = \mathcal{L}_j$ and $L_m(\mathbf{G'_j}) = P_j^{-1} L_m(\mathbf{G_j}) = \mathcal{L}_{m,j}$, for all $j \in \{1, 2, \ldots n\}$. **(2)**

We next define $\mathbf{G'} = \mathbf{G'_1} \times \mathbf{G'_2} \times \ldots \times \mathbf{G'_n} = (Q', \Sigma, \delta', q_o, Q'_m)$

We note that as $L(\mathbf{G'}) = L(\mathbf{G})$, $L_m(\mathbf{G'}) = L_m(\mathbf{G})$, and the fact that the $\mathbf{G'_j}$ are constructed by simply adding selfloops to the original DES, it follows from **(1)**

that $\delta'(q_o, s) = \delta'(q_o, t)$.

We can now apply *Proposition 13* to $\mathbf{G}'$ and the $\mathbf{G}'_\mathbf{j}$ $(j = 1, 2, \ldots n)$ and conclude:

$s \equiv_{L(\mathbf{G_i}')} t \ \wedge \ s \equiv_{L_m(\mathbf{G_i}')} t$

$s \equiv_{\mathcal{L}_i} t \ \wedge \ s \equiv_{\mathcal{L}_{m,i}} t$, by **(2)**.

□

We will now use these propositions to prove some useful results related to the HISC conditions. We first introduce some notation that we will need. Let $I_1 \subseteq \{1, 2, \ldots n\}$ and $I_2 \subseteq \{1, 2, \ldots n\}$ be nonempty index sets for our n DES. Define $\mathcal{L}_{I_1} = \cap_{j \in I_1} \mathcal{L}_j$ and $\mathcal{L}_{I_2} = \cap_{k \in I_2} \mathcal{L}_k$.

Proposition 15 *Let* $\mathbf{G} = \mathbf{G_1} \| \mathbf{G_2} \| ... \| \mathbf{G_n} = (Q, \Sigma, \delta, q_o, Q_m)$, $\Sigma_a \subseteq \Sigma$ *and* I_1 *and* I_2 *be nonempty index sets for our* n *DES. It thus follows that for all* $s, t \in \Sigma^*$, *if* $\delta(q_o, s) = \delta(q_o, t)$ *then*

$$Elig_{\mathcal{L}_{I_1}}(s) \cap \Sigma_a \not\subseteq Elig_{\mathcal{L}_{I_2}}(s) \quad \Leftrightarrow \quad Elig_{\mathcal{L}_{I_1}}(t) \cap \Sigma_a \not\subseteq Elig_{\mathcal{L}_{I_2}}(t)$$

◊

Proof

Let $s, t \in \Sigma^*$.

Assume $\delta(q_o, s) = \delta(q_o, t)$ **(1)**

We will now show that this implies

$$\mathrm{Elig}_{\mathcal{L}_{I_1}}(s) \cap \Sigma_a \not\subseteq \mathrm{Elig}_{\mathcal{L}_{I_2}}(s) \quad \Leftrightarrow \quad \mathrm{Elig}_{\mathcal{L}_{I_1}}(t) \cap \Sigma_a \not\subseteq \mathrm{Elig}_{\mathcal{L}_{I_2}}(t)$$

We first note that as s and t are arbitrary and the condition to be proven is symmetric, it is sufficient to prove

$$\mathrm{Elig}_{\mathcal{L}_{I_1}}(s) \cap \Sigma_a \not\subseteq \mathrm{Elig}_{\mathcal{L}_{I_2}}(s) \quad \Rightarrow \quad \mathrm{Elig}_{\mathcal{L}_{I_1}}(t) \cap \Sigma_a \not\subseteq \mathrm{Elig}_{\mathcal{L}_{I_2}}(t)$$

We thus assume $\text{Elig}_{\mathcal{L}_{I_1}}(s) \cap \Sigma_a \not\subseteq \text{Elig}_{\mathcal{L}_{I_2}}(s)$. **(2)**

We will now show this implies $\text{Elig}_{\mathcal{L}_{I_1}}(t) \cap \Sigma_a \not\subseteq \text{Elig}_{\mathcal{L}_{I_2}}(t)$. **(3)**

To prove **(3)**, it is sufficient to show:

$(\exists \sigma \in \Sigma_a)\, t\sigma \in \mathcal{L}_{I_1} \wedge t\sigma \notin \mathcal{L}_{I_2}$

From **(2)**, we can conclude: $(\exists \sigma \in \Sigma_a)\, s\sigma \in \mathcal{L}_{I_1} \wedge s\sigma \notin \mathcal{L}_{I_2}$ **(4)**

We next note that by **(1)** we can apply *Proposition 14* and can conclude:

$(\forall i \in (I_1 \cup I_2))s \equiv_{\mathcal{L}_i} t$

Combining with **(4)**, we can thus conclude:

$t\sigma \in \cap_{j \in I_1} \mathcal{L}_j = \mathcal{L}_{I_1}$ and $t\sigma \notin \cap_{k \in I_2} \mathcal{L}_k = \mathcal{L}_{I_2}$

$\Rightarrow t\sigma \in \mathcal{L}_{I_1} \wedge t\sigma \notin \mathcal{L}_{I_2}$, as required.

□

We now note that if we choose I_1, I_2, and Σ_a appropriately, then *Proposition 15* can be applied to the level-wise controllability definition, as well as points 3 and 4 of the interface consistency definition. Essentially the proposition states that if a string fails such a property, then all the strings that lead to the same state in the synchronous product will fail the same property, thus we need to remove the state. This is consistent with how our state based algorithms work. We also note that removing a state not only removes all strings that reach this state from the initial state, but also removes all defined strings that can leave this state. In other words, if we remove state $q \in Q$ of DES $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$, we remove from $L(\mathbf{G})$ the strings $L_q := \{s \in L(\mathbf{G}) | \, \delta(q_o, s) = q\}$ as well as all strings that have prefixes in L_q (ie. $\text{Ext}_{L(\mathbf{G})}(L_q)$). This is consistent with how we defined our language based fixpoint operators in Chapter 5.

We next present a nonblocking result. To show nonblocking for a DES $\mathbf{G}$, we would need to show $L(\mathbf{G}) = \overline{L_m(\mathbf{G})}$. If $\mathbf{G}$ was blocking, then we would have

$L(\mathbf{G}) \not\subseteq \overline{L_m(\mathbf{G})}$. We will now show that if a string was in $L(\mathbf{G})$ but not in $\overline{L_m(\mathbf{G})}$, then all the strings that lead to the same state will also fail this condition, thus we need to remove the state. Clearly this result can be applied to make the high level, or a given low level nonblocking.

Proposition 16 *Let* $\mathbf{G} = (Q, \Sigma, \delta, q_o, Q_m)$. *It thus follows that for all* $s, t \in \Sigma^*$, *if* $\delta(q_o, s) = \delta(q_o, t)$ *then* $s \notin \overline{L_m(\mathbf{G})} \Leftrightarrow t \notin \overline{L_m(\mathbf{G})}$ $\Diamond$

Proof

Let $s, t \in \Sigma^*$.

Assume $\delta(q_o, s) = \delta(q_o, t)$ **(1)**

We will now show that this implies $s \notin \overline{L_m(\mathbf{G})} \Leftrightarrow t \notin \overline{L_m(\mathbf{G})}$

We first note that as s and t are arbitrary and the condition to be proven is symmetric, it is sufficient to prove: $s \notin \overline{L_m(\mathbf{G})} \Rightarrow t \notin \overline{L_m(\mathbf{G})}$

Assume $s \notin \overline{L_m(\mathbf{G})}$ **(2)**

We will now show this implies $t \notin \overline{L_m(\mathbf{G})}$.

We next note that we know from [79] that $\delta(q_o, s) = \delta(q_o, t)$ implies that $s \equiv_{L_m(\mathbf{G})} t$ **(3)**

From **(2)**, we can conclude: $(\forall u \in \Sigma^*)\, su \notin L_m(\mathbf{G})$

$\Rightarrow (\forall u \in \Sigma^*)\, tu \notin L_m(\mathbf{G})$, by **(3)**.

$\Rightarrow t \notin \overline{L_m(\mathbf{G})}$, as required.

□

We now present a set of propositions relevant to the high level interface controllable (HIC) definition, the j^{th} low level interface controllable (LICj) definition,

as well as for nonblocking.

Let Φ stand for the n^{th} degree HISC-valid specification interface system that respects the alphabet partition given by (3.1) and is composed of plant DES $\mathbf{G}_H^p$, $\mathbf{G}_{L_1}^p, \ldots, \mathbf{G}_{L_n}^p$, specification DES $\mathbf{E}_H$, $\mathbf{E}_{L_1}, \ldots, \mathbf{E}_{L_n}$, and interface DES $\mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$, that we are considering. We will also take j to be an index in the range $\{1, \ldots, n\}$. We will also make use of the related natural projections and languages defined in Section 5.1. We thus have $\mathbf{G}_{\text{HL}} = \mathbf{G}_H^p||\mathbf{E}_H||\mathbf{G}_{I_1}|| \ldots ||\mathbf{G}_{I_n}$ and $\mathbf{G}_{\text{LL}_j} = \mathbf{G}_{L_j}^p||\mathbf{E}_{L_j}||\mathbf{G}_{I_j}$. We now need to define DES $\mathbf{G}'_{\text{HL}}$ to be DES $\mathbf{G}_{\text{HL}}$ with events $\Sigma - \Sigma_{IH}$ selflooped at every state. This is to extend the event set of $\mathbf{G}_{\text{HL}}$ to Σ, so that it will be compatible with the languages used in the HIC definition (i.e. $L(\mathbf{G}'_{\text{HL}}) = \mathcal{Z}_H$). Similarly, we need to define DES $\mathbf{G}'_{\text{LL}_j}$ to be DES $\mathbf{G}_{\text{LL}_j}$ with events $\Sigma - \Sigma_{IL_j}$ selflooped at every state, thus $L(\mathbf{G}'_{\text{LL}_j}) = \mathcal{Z}_{L_j}$.

We first present a proposition for the HIC definition.

Proposition 17 *For system Φ, let $\mathbf{G} := \mathbf{G}'_{HL} = (Q, \Sigma, \delta, q_o, Q_m)$. It follows that for all $s, t \in L(\mathbf{G}'_{HL})$, if $\delta(q_o, s) = \delta(q_o, t)$ then*

1. $Elig_{\mathcal{H}^p \cap \mathcal{I}}(s) \cap \Sigma_u \not\subseteq Elig_{L(\mathbf{G}'_{HL})}(s) \Leftrightarrow Elig_{\mathcal{H}^p \cap \mathcal{I}}(t) \cap \Sigma_u \not\subseteq Elig_{L(\mathbf{G}'_{HL})}(t)$
2. $(\forall j \in \{1, \ldots, n\})\ Elig_{\mathcal{I}_j}(s) \cap \Sigma_{A_j} \not\subseteq Elig_{\mathcal{H}^p \cap L(\mathbf{G}'_{HL}) \cap \bigcap_{k \neq j} \mathcal{I}_k}(s) \Leftrightarrow$
$Elig_{\mathcal{I}_j}(t) \cap \Sigma_{A_j} \not\subseteq Elig_{\mathcal{H}^p \cap L(\mathbf{G}'_{HL}) \cap \bigcap_{k \neq j} \mathcal{I}_k}(t)$

$\Diamond$

Proof

Let $s, t \in L(\mathbf{G}'_{\text{HL}})$.

Assume $\delta(q_o, s) = \delta(q_o, t)$.

1. Show $\text{Elig}_{\mathcal{H}^p \cap \mathcal{I}}(s) \cap \Sigma_u \not\subseteq \text{Elig}_{L(\mathbf{G}'_{\text{HL}})}(s) \Leftrightarrow \text{Elig}_{\mathcal{H}^p \cap \mathcal{I}}(t) \cap \Sigma_u \not\subseteq \text{Elig}_{L(\mathbf{G}'_{\text{HL}})}(t)$

This follows from *Proposition 15* when we take $\Sigma_a = \Sigma_u$, set index I_1 to represent $\mathbf{G}_H^p$, $\mathbf{G}_{I_1}$, ..., $\mathbf{G}_{I_n}$, and set index I_2 to represent all of the DES used to construct DES $\mathbf{G}_{\text{HL}}$.

2. Show $(\forall j \in \{1, \ldots, n\})\ \text{Elig}_{\mathcal{I}_j}(s)\ \cap\ \Sigma_{A_j} \not\subseteq \text{Elig}_{\mathcal{H}^p \cap L(\mathbf{G}'_{\text{HL}}) \cap \bigcap_{k \neq j} \mathcal{I}_k}(s) \Leftrightarrow$

$$\text{Elig}_{\mathcal{I}_j}(t)\ \cap\ \Sigma_{A_j} \not\subseteq \text{Elig}_{\mathcal{H}^p \cap L(\mathbf{G}'_{\text{HL}}) \cap \bigcap_{k \neq j} \mathcal{I}_k}(t)$$

Let $j \in \{1, \ldots, n\}$.

The result follows from *Proposition 15* when we take $\Sigma_a = \Sigma_{A_j}$, set index I_1 to represent $\mathbf{G}_{I_j}$, and set index I_2 to represent all of the DES used to construct DES $\mathbf{G}_{\text{HL}}$. With respect to the definition of I_2, we use the fact that $L(\mathbf{G}'_{\text{HL}}) = \mathcal{H}^p \cap L(\mathbf{G}'_{\text{HL}}) \cap \bigcap_{k \neq j} \mathcal{I}_k$.

□

We now present a proposition for the LICj definition.

Proposition 18 *For system Φ, let $\mathbf{G} := \mathbf{G}'_{LL_j} = (Q, \Sigma, \delta, q_o, Q_m)$. It follows that for all $s, t \in L(\mathbf{G}'_{LL_j})$, if $\delta(q_o, s) = \delta(q_o, t)$ then*

1. $Elig_{\mathcal{L}_j^p}(s) \cap \Sigma_u\ \not\subseteq\ Elig_{L(\mathbf{G}'_{LL_j}) \cap \mathcal{I}_j}(s) \Leftrightarrow Elig_{\mathcal{L}_j^p}(t) \cap \Sigma_u\ \not\subseteq\ Elig_{L(\mathbf{G}'_{LL_j}) \cap \mathcal{I}_j}(t)$

2. $Elig_{\mathcal{I}_j}(s)\ \cap\ \Sigma_{R_j} \not\subseteq Elig_{\mathcal{L}_j^p \cap L(\mathbf{G}'_{LL_j})}(s) \Leftrightarrow Elig_{\mathcal{I}_j}(t)\ \cap\ \Sigma_{R_j} \not\subseteq Elig_{\mathcal{L}_j^p \cap L(\mathbf{G}'_{LL_j})}(t)$

3. $(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\ [s\rho\alpha \in \mathcal{I}_j] \wedge [(\forall l \in \Sigma_{L_j}^*)\, s\rho l\alpha \notin \mathcal{L}_j^p \cap L(\mathbf{G}'_{LL_j}) \cap \mathcal{I}_j] \Leftrightarrow$
$[t\rho\alpha \in \mathcal{I}_j] \wedge [(\forall l \in \Sigma_{L_j}^*)\, t\rho l\alpha \notin \mathcal{L}_j^p \cap L(\mathbf{G}'_{LL_j}) \cap \mathcal{I}_j]$

4. $[s \in \mathcal{I}_{m_j}] \wedge [(\forall l \in \Sigma_{L_j}^*)\ sl \notin \mathcal{L}_{m_j}^p \cap L_m(\mathbf{G}'_{LL_j}) \cap \mathcal{I}_{m_j}] \Leftrightarrow$
$[t \in \mathcal{I}_{m_j}] \wedge [(\forall l \in \Sigma_{L_j}^*)\ tl \notin \mathcal{L}_{m_j}^p \cap L_m(\mathbf{G}'_{LL_j}) \cap \mathcal{I}_{m_j}]$

◊

Proof

Let $s, t \in L(\mathbf{G}'_{\mathrm{LL}_j})$.

Assume $\delta(q_o, s) = \delta(q_o, t)$. **(1)**

Using **(1)**, we can apply *Proposition 14* and conclude: **(2)**

$$\begin{aligned} s \equiv_{\mathcal{L}_j^p} t \quad &\wedge \quad s \equiv_{\mathcal{L}_{m_j}^p} t \\ s \equiv_{\mathcal{E}_{L_j}} t \quad &\wedge \quad s \equiv_{\mathcal{E}_{L_{j,m}}} t \\ s \equiv_{\mathcal{I}_j} t \quad &\wedge \quad s \equiv_{\mathcal{I}_{m_j}} t \end{aligned}$$

1. Show $\mathrm{Elig}_{\mathcal{L}_j^p}(s) \cap \Sigma_u \not\subseteq \mathrm{Elig}_{L(\mathbf{G}'_{\mathrm{LL}_j}) \cap \mathcal{I}_j}(s) \Leftrightarrow \mathrm{Elig}_{\mathcal{L}_j^p}(t) \cap \Sigma_u \not\subseteq \mathrm{Elig}_{L(\mathbf{G}'_{\mathrm{LL}_j}) \cap \mathcal{I}_j}(t)$

This follows from *Proposition 15* when we take $\Sigma_a = \Sigma_u$, set index I_1 to represent $\mathbf{G}_{L_j}^p$, and set index I_2 to represent all of the DES used to construct DES $\mathbf{G}_{\mathrm{LL}_j}$.

2. Show $\mathrm{Elig}_{\mathcal{I}_j}(s) \cap \Sigma_{R_j} \not\subseteq \mathrm{Elig}_{\mathcal{L}_j^p \cap L(\mathbf{G}'_{\mathrm{LL}_j})}(s) \Leftrightarrow \mathrm{Elig}_{\mathcal{I}_j}(t) \cap \Sigma_{R_j} \not\subseteq \mathrm{Elig}_{\mathcal{L}_j^p \cap L(\mathbf{G}'_{\mathrm{LL}_j})}(t)$

This follows from *Proposition 15* when we take $\Sigma_a = \Sigma_{R_j}$, set index I_1 to represent $\mathbf{G}_{I_j}$, and set index I_2 to represent all of the DES used to construct DES $\mathbf{G}_{\mathrm{LL}_j}$.

3. Show $(\forall \rho \in \Sigma_{R_j})(\forall \alpha \in \Sigma_{A_j})\ [s\rho\alpha \in \mathcal{I}_j] \wedge [(\forall l \in \Sigma_{L_j}^*)\, s\rho l\alpha \notin \mathcal{L}_j^p \cap L(\mathbf{G}'_{\mathrm{LL}_j}) \cap \mathcal{I}_j] \Leftrightarrow$
$[t\rho\alpha \in \mathcal{I}_j] \wedge [(\forall l \in \Sigma_{L_j}^*)\, t\rho l\alpha \notin \mathcal{L}_j^p \cap L(\mathbf{G}'_{\mathrm{LL}_j}) \cap \mathcal{I}_j]$

Let $\rho \in \Sigma_{R_j}$, and $\alpha \in \Sigma_{A_j}$.

We first note that as s and t are arbitrary and the condition to be proven is symmetric, it is sufficient to prove:

$$[s\rho\alpha \in \mathcal{I}_j] \wedge [(\forall l \in \Sigma_{L_j}^*)\, s\rho l\alpha \notin \mathcal{L}_j^p \cap L(\mathbf{G}'_{\mathrm{LL}_j}) \cap \mathcal{I}_j] \Rightarrow [t\rho\alpha \in \mathcal{I}_j] \wedge [(\forall l \in \Sigma_{L_j}^*)\, t\rho l\alpha \notin \mathcal{L}_j^p \cap L(\mathbf{G}'_{\mathrm{LL}_j}) \cap \mathcal{I}_j]$$

Assume $[s\rho\alpha \in \mathcal{I}_j] \wedge [(\forall l \in \Sigma^*_{L_j})\, s\rho l\alpha \notin \mathcal{L}^p_j \cap L(\mathbf{G}'_{\mathrm{LL}_j}) \cap \mathcal{I}_j]$. **(3)**

We will now show this implies

$$[t\rho\alpha \in \mathcal{I}_j] \wedge [(\forall l \in \Sigma^*_{L_j})\, t\rho l\alpha \notin \mathcal{L}^p_j \cap L(\mathbf{G}'_{\mathrm{LL}_j}) \cap \mathcal{I}_j]$$

From **(2)**, **(3)**, and fact that by definition $L(\mathbf{G}'_{\mathrm{LL}_j}) = \mathcal{L}^p_j \cap \mathcal{E}_{L_j} \cap \mathcal{I}_j$, we can conclude

$$[t\rho\alpha \in \mathcal{I}_j] \;\wedge\; [(\forall l \in \Sigma^*_{L_j})\, t\rho l\alpha \notin \mathcal{L}^p_j \cap \mathcal{E}_{L_j} \cap \mathcal{I}_j$$

$\Rightarrow [t\rho\alpha \in \mathcal{I}_j] \wedge [(\forall l \in \Sigma^*_{L_j})\, t\rho l\alpha \notin \mathcal{L}^p_j \cap L(\mathbf{G}'_{\mathrm{LL}_j}) \cap \mathcal{I}_j]$, as required.

4. Show $[s \in \mathcal{I}_{m_j}] \wedge [(\forall l \in \Sigma^*_{L_j})\; sl \notin \mathcal{L}^p_{m_j} \cap L_m(\mathbf{G}'_{\mathrm{LL}_j}) \cap \mathcal{I}_{m_j}] \Leftrightarrow$
$[t \in \mathcal{I}_{m_j}] \wedge [(\forall l \in \Sigma^*_{L_j})\; tl \notin \mathcal{L}^p_{m_j} \cap L_m(\mathbf{G}'_{\mathrm{LL}_j}) \cap \mathcal{I}_{m_j}]$

We first note that as s and t are arbitrary and the condition to be proven is symmetric, it is sufficient to prove:

$[s \in \mathcal{I}_{m_j}] \wedge [(\forall l \in \Sigma^*_{L_j})\; sl \notin \mathcal{L}^p_{m_j} \cap L_m(\mathbf{G}'_{\mathrm{LL}_j}) \cap \mathcal{I}_{m_j}] \Rightarrow$
$[t \in \mathcal{I}_{m_j}] \wedge [(\forall l \in \Sigma^*_{L_j})\; tl \notin \mathcal{L}^p_{m_j} \cap L_m(\mathbf{G}'_{\mathrm{LL}_j}) \cap \mathcal{I}_{m_j}]$

Assume $[s \in \mathcal{I}_{m_j}] \wedge [(\forall l \in \Sigma^*_{L_j})\; sl \notin \mathcal{L}^p_{m_j} \cap L_m(\mathbf{G}'_{\mathrm{LL}_j}) \cap \mathcal{I}_{m_j}]$ **(4)**

We will now show this implies

$$[t \in \mathcal{I}_{m_j}] \wedge [(\forall l \in \Sigma^*_{L_j})\; tl \notin \mathcal{L}^p_{m_j} \cap L_m(\mathbf{G}'_{\mathrm{LL}_j}) \cap \mathcal{I}_{m_j}]$$

From **(2)**, **(4)**, and fact that by definition $L_m(\mathbf{G}'_{\mathrm{LL}_j}) = \mathcal{L}^p_{m_j} \cap \mathcal{E}_{L_{j,m}} \cap \mathcal{I}_{m_j}$, we can conclude: $[t \in \mathcal{I}_{m_j}] \wedge [(\forall l \in \Sigma^*_{L_j})\; tl \notin \mathcal{L}^p_{m_j} \cap \mathcal{E}_{L_{j,m}} \cap \mathcal{I}_{m_j}]$.

$\Rightarrow [t \in \mathcal{I}_{m_j}] \wedge [(\forall l \in \Sigma^*_{L_j})\; tl \notin \mathcal{L}^p_{m_j} \cap L_m(\mathbf{G}'_{\mathrm{LL}_j}) \cap \mathcal{I}_{m_j}]$, as required.

□

6.2 Verify Command-pair Interfaces

A DES must satisfy the properties given in Definition 3.2.1 to be a command-pair interface. As this definition is given in terms of the languages $L(\mathbf{G_I})$ and $L_m(\mathbf{G_I})$, we would prefer a state-based definition that would be easier to verify using automata. We present below such a definition, and then show that it is equivalent to Definition 3.2.1.

Definition 6.2.1 *A DES* $\mathbf{G_I} = (X, \Sigma_R \dot{\cup} \Sigma_A, \xi, x_0, X_m)$*, with* $X_{rch} \subseteq X$ *its set of reachable states and* $X_{rm} \subseteq X_m$ *its set of reachable marked states, is a command-pair interface if the following properties are satisfied:*

1. $x_0 \in X_{rm}$
2. $(\forall x \in X_{rm})(\forall \sigma \in \Sigma_R \dot{\cup} \Sigma_A)\, \xi(x, \sigma)! \Rightarrow \sigma \in \Sigma_R \wedge \xi(x, \sigma) \in X_{rch} - X_{rm}$
3. $(\forall x \in X_{rch} - X_{rm})(\forall \sigma \in \Sigma_R \dot{\cup} \Sigma_A)\, \xi(x, \sigma)! \Rightarrow \sigma \in \Sigma_A \wedge \xi(x, \sigma) \in X_{rm}$ $\Diamond$

We now show that our new definition for command-pair interfaces is is equivalent to the original definition given in Chapter 3.

Proposition 19 *Definition 3.2.1 and 6.2.1 are equivalent.* $\Diamond$

Proof

Let $\mathbf{G_I} = (X, \Sigma_R \dot{\cup} \Sigma_A, \xi, x_0, X_m)$, and let $X_{\text{rch}} \subseteq X$ be its set of reachable states and $X_{rm} \subseteq X_m$ its set of reachable marked states.

We will now show that $\mathbf{G_I}$ satisfies Definition 3.2.1 if and only if it satisfies Definition 6.2.1.

I) Assume $\mathbf{G_I}$ satisfies Definition 6.2.1.

We will now show that this implies $\mathbf{G_I}$ satisfies Definition 3.2.1.

To do this we need to show that $\mathbf{G_I}$ satisfies *points A* and *B* of Definition 3.2.1.

I.A) To show *point A*, we need to show: $L(\mathbf{G}_I) \subseteq \overline{(\Sigma_R.\Sigma_A)^*}$

This essentially says that a string in $L(\mathbf{G}_I)$ can be either the empty string, ϵ, or it must start with a request event and then alternate answer and then request events from then on.

This is equivalent to saying: †

$$(\forall s \in L(\mathbf{G}_I))(\exists k \in \{0,1,2,3,\ldots\})(\exists \sigma_1, \sigma_2, \ldots, \sigma_k \in \Sigma_R \dot{\cup} \Sigma_A)\, \sigma_1\sigma_2\ldots\sigma_k = s$$

where $k = 0$ means $s = \epsilon$, and for $1 \geq i \geq k$

$$\sigma_i \in \Sigma_R \text{ if } i \text{ is an odd number, and } \sigma_i \in \Sigma_A \text{ if } i \text{ is an even number}$$

From *point 1* of Definition 6.2.1, we know that x_0 is a marked state, and from *point 2* we know that at all reachable marked states only request events are allowed, and they always take us to a non marked state. From *point 3*, we know that at all reachable non marked state, only answer events are allowed, and they always take us to a marked state.

Clearly, this implies that for a string $t \in L(\mathbf{G}_I)$, either $t = \epsilon$ or t starts with a request event, and then alternates answer-request event from then on. In other words, $L(\mathbf{G}_I)$ satisfies †.

I.B) To show *point B*, we need to show: $L_m(\mathbf{G}_I) = (\Sigma_R.\Sigma_A)^* \cap L(\mathbf{G}_I)$

We thus need to show $L_m(\mathbf{G}_I) \subseteq (\Sigma_R.\Sigma_A)^* \cap L(\mathbf{G}_I)$ and $(\Sigma_R.\Sigma_A)^* \cap L(\mathbf{G}_I) \subseteq L_m(\mathbf{G}_I)$.

I.B.i) Assume $s \in (\Sigma_R.\Sigma_A)^* \cap L(\mathbf{G}_I)$.

Must show this implies $s \in L_m(\mathbf{G}_I)$

We first note that $s \in (\Sigma_R.\Sigma_A)^*$ implies $s = \epsilon$ or $s \in (\Sigma_R \dot{\cup} \Sigma_A)^*.\Sigma_A$

From *point 1* of Definition 6.2.1, we know that x_0 is a marked state, thus $\epsilon \in L_m(\mathbf{G}_I)$.

If $s \in (\Sigma_R \dot{\cup} \Sigma_A)^*.\Sigma_A$ we can conclude:
$(\exists s' \in (\Sigma_R \dot{\cup} \Sigma_A)^*)(\exists \alpha \in \Sigma_A)\, s'\alpha = s$

As $s \in L(\mathbf{G}_I))$, it follows that $s' \in L(\mathbf{G}_I))$ as $L(\mathbf{G}_I))$ is closed.

We thus have $\xi(x_0, s') \in X_{\text{rch}}$ and $\xi(x_0, s'\alpha) \in X_{\text{rch}}$.

By *point 2* of Definition 6.2.1, we can thus conclude $\xi(x_0, s') \in X_{\text{rch}} - X_{rm}$ as answer events are not permitted at states in X_{rm}.

We can thus conclude by *point 3* that $\xi(x_0, s'\alpha) \in X_{rm}$.

We thus have $s \in L_m(\mathbf{G}_I)$ (as $s = s'\alpha$), as required.

I.B.ii) Assume $s \in L_m(\mathbf{G}_I)$

As $L_m(\mathbf{G}_I) \subseteq L(\mathbf{G}_I)$, we immediately have $s \in L(\mathbf{G}_I)$.

It is thus sufficient to show $s \in (\Sigma_R.\Sigma_A)^*$

From **Part I.A**, we have $L(\mathbf{G}_I) \subseteq \overline{(\Sigma_R.\Sigma_A)^*}$.

As $L_m(\mathbf{G}_I) \subseteq L(\mathbf{G}_I)$, we thus have $L_m(\mathbf{G}_I) \subseteq \overline{(\Sigma_R.\Sigma_A)^*}$.

In other words, strings in $L_m(\mathbf{G}_I)$ can be either the empty string, or it must start with a request event and then alternate answer and then request events from then on.

If $s = \epsilon$, we would have $s \in (\Sigma_R.\Sigma_A)^*$.

We now examine the case that s starts with a request event and then alternates answer and then request events.

This means that it is sufficient to show that $s \in (\Sigma_R \dot{\cup} \Sigma_A)^*.\Sigma_A$.

As $s \in L_m(\mathbf{G}_I)$, it follows that $\xi(x_0, s) \in X_{rm}$.

From *points 2* and *3* of Definition 6.2.1, it follows that $\xi(x_0, s)$ can only be reached by an answer event transition, thus $s \in (\Sigma_R \dot{\cup} \Sigma_A)^*.\Sigma_A$, as required.

From **Part I.B.i** and **I.B.ii**, we have $L_m(\mathbf{G}_I) = (\Sigma_R.\Sigma_A)^* \cap L(\mathbf{G}_I)$, as required.

From **Part I.A** and **I.B**, we have that $\mathbf{G_I}$ satisfies Definition 3.2.1.

II) Assume $\mathbf{G_I}$ satisfies Definition 3.2.1.

We will now show that this implies $\mathbf{G_I}$ satisfies Definition 6.2.1.

We first not that by *point A* of Definition 3.2.1, we know that $L(\mathbf{G}_I)$ satisfies †. **(1)**

II.1) Show that $x_0 \in X_{rm}$

By *point B* of Definition 3.2.1, we have $\epsilon \in L_m(\mathbf{G}_I)$. It immediately follows that $x_0 \in X_{rm}$ from the definition of $L_m(\mathbf{G}_I)$.

II.2) Show that $(\forall x \in X_{rm})(\forall \sigma \in \Sigma_R \dot{\cup} \Sigma_A)\, \xi(x, \sigma)! \Rightarrow \sigma \in \Sigma_R \wedge \xi(x, \sigma) \in X_{\text{rch}} - X_{rm}$

Let $x \in X_{rm}$ and $\sigma \in \Sigma_R \dot{\cup} \Sigma_A$.

Assume $\xi(x, \sigma)!$. Must show implies $\sigma \in \Sigma_R \wedge \xi(x, \sigma) \in X_{\text{rch}} - X_{rm}$

From **Part II.1**, we know we have two choices: $x = x_0$ or $x \neq x_0$

If $x = x_0$, we know that $\sigma \in \Sigma_R$ as every string in $L(\mathbf{G}_I)$ must start with a request event, by **(1)**.

By *point B* of Definition 3.2.1, we can conclude that $\sigma \notin L_m(\mathbf{G}_I)$.

We can thus conclude that $\xi(x, \sigma) \in X_{\text{rch}} - X_{rm}$.

We now consider the case $x \neq x_0$.

As x is reachable by assumption, we can conclude $\exists s \in L(\mathbf{G}_I)$ such that $\xi(x_0, s) = x$.

From **(1)**, we can conclude that s starts with a request event, then alternates answer then request event.

As $x \in X_{rm}$ by assumption, we thus have $s \in L_m(\mathbf{G}_I)$.

$\Rightarrow s \in (\Sigma_R \dot{\cup} \Sigma_A)^*.\Sigma_A$, by *point B* of Definition 3.2.1 and fact $x \neq x_0$.

By **(1)**, we can thus conclude $\sigma \in \Sigma_R$. **(2)**

As $\xi(x, \sigma)!$ by assumption and fact x is reachable, we thus have $s\sigma \in L(\mathbf{G}_I)$.

$\Rightarrow s\sigma \notin L_m(\mathbf{G}_I)$ by **(2)** and *point B* of Definition 3.2.1.

$x \in X_{\text{rch}} - X_{rm}$, as required.

II.3) Show that $(\forall x \in X_{\text{rch}} - X_{rm})(\forall \sigma \in \Sigma_R \dot{\cup} \Sigma_A)\, \xi(x, \sigma)! \Rightarrow \sigma \in \Sigma_A \wedge \xi(x, \sigma) \in X_{rm}$

Let $x \in X_{\text{rch}} - X_{rm}$ and $\sigma \in \Sigma_R \dot{\cup} \Sigma_A$.

Assume $\xi(x, \sigma)!$. Must show implies $\sigma \in \Sigma_A \wedge \xi(x, \sigma) \in X_{rm}$

As x is reachable, we can conclude $\exists s \in L(\mathbf{G}_I)$ such that $\xi(x_0, \sigma) = x$.

As $x \in X_{\text{rch}} - X_{rm}$, we know that $s \notin L_m(\mathbf{G}_I)$. **(3)**

This also allows us to conclude $x \neq x_0$ by **Part II.1**.

We can thus conclude by **(1)** that s begins with a request event and then alternates answer then request event. **(4)**

From **(3)** and *point B* of Definition 3.2.1, we can conclude $s \notin (\Sigma_R \dot{\cup} \Sigma_A)^*.\Sigma_A$.

$\Rightarrow s \in (\Sigma_R \dot{\cup} \Sigma_A)^*.\Sigma_R$, by **(4)**.

$\Rightarrow \sigma \in \Sigma_A$, by **(4)**. **(5)**

As $\xi(x, \sigma)!$ by assumption and fact x is reachable, we thus have $s\sigma \in L(\mathbf{G}_I)$.

$\Rightarrow s\sigma \in L_m(\mathbf{G}_I)$, by **(5)** and *points A B* of Definition 3.2.1.

$\Rightarrow \xi(x, \sigma) \in X_{rm}$, as required.

From **Parts II.1-3**, we have that $\mathbf{G_I}$ satisfies Definition 6.2.1.

From **Parts I** and **II**, we can now conclude that $\mathbf{G_I}$ satisfies Definition 3.2.1 if and only if it satisfies Definition 6.2.1.

□

We will now present an algorithm that checks whether a given DES is a command pair interface according to Definition 6.2.1. All the properties will be verified in one traversal over the DES.

When an interface DES is designed and read into memory, we require that every state starting from the initial state, is given an index number starting from 1, which can be used as a keyword to identify a unique state. We reserve 0 for later use to mark an undefined state or a state with special meaning. Similarly, we index events starting from 1.

Given a DES $\mathbf{G_I} = (X, \Sigma_I, \xi, x_0, X_m)$ and event sets Σ_R, Σ_A, we need to check that $\mathbf{G_I}$ satisfies the properties listed in Definition 6.2.1.

Listing 6.3: Checking Command-pair interface

```
1  bool CommandPair ( G_I )
2  begin
3    if DisjointUnion(Σ_R, Σ_A, Σ_I)=false then
4        return false;
5    end if
6    if G_I = EMPTY then
7        return false;
8    end if
9    if x_0 ∉ X_m then
10       return false;                    //point 1
11   end if
12   X_Found ← {x_0};
13   X_Pend ← {x_0};
14   while X_Pend ≠ ∅ do
15       x ← pop X_Pend;
16       if x ∈ X_m then
17         for each σ ∈ Σ_I do
18           if ξ(x, σ)! then
19             x' ← ξ(x, σ);
20             if σ ∉ Σ_R OR x' ∈ X_m then
21               return false;          //point 2
22             else
23               if x' ∉ X_Found then
24                 push X_Found, x';
25                 push X_Pend , x';
26               end if
27             end if
```

```
28          end if
29        end for
30      else                                  //x ∉ X_m
31        for each σ ∈ Σ_I do
32          if ξ(x, σ)! then
33            x' ← ξ(x, σ);
34            if σ ∉ Σ_A OR x' ∉ X_m then
35              return false;                 //point 3
36            else
37              if x' ∉ X_Found then
38                push X_Found, x';
39                push X_Pend , x';
40              end if
41            end if
42          end if
43        end for
44      end if
45    end while
46    return true;
47  end
```

Let $n_x = |X|, n_s = |\Sigma_I|$ be the number of states and events. Lines 3-5 calls Algorithm 6.2 to check if Σ_I is a disjoint union of Σ_R and Σ_A, it takes $O(n_s \cdot \log n_s)$. Lines 6-8 checks if the DES is empty. This takes constant time. Lines 9-11 make sure that the initial state is marked. Lines 12-13 initialize the following state sets: X_{FOUND}: states found/processed; X_{Pend}: a set of states waiting to be processed. The initialization takes constant time.

Lines 17-29 check that, for any given state in X_{rm}, only request events occur and they lead to states in $X_{\text{rch}} - X_{rm}$ (Point 2). Similarly Lines 31-43 check Point 3.

The *while* block, lines 14-45, runs n_x times worst case. The for blocks in lines 17-29 and lines 31-43 run n_s times. Checking event type (such as on line 20) takes constant time. Since we know the size of the state space, we can use an array to store flags to mark whether a state is found or not. Checking of whether a state is found or not, as done on line 23, thus takes constant time. Operations (Push and pop) on X_{Pend} take constant time. So the *while* block, Line 14-45, takes $O(n_x \cdot n_s)$ running time. We conclude that the running time for Algorithm 6.3 is $O(n_s \cdot \log n_s + n_x \cdot n_s)$.

6.3 Level-wise Nonblocking and Controllable

Since our synthesis method constructs supervisors for each subsystem and makes sure the resulting supervisor is nonblocking and controllable, we don't need to verify these two properties for our supervisors.

In case a verification is required, normal verification tools such as TCT can be used. Verification is much simpler than synthesis since whenever a state or string fails the desired property, the whole process can be stopped and the state and property that failed can be then be returned to the user. Rudie [61] studied the complexity of supcon for systems with one plant and one spec (ie. a flat system), and concluded the time complexity for such a supcon algorithm to be $O(sn_pn_l+s_ue_pe_l)$, in which s and s_u represent numbers of events and uncontrollable events, n_p and n_l represent the state sizes of the plant and specification, e_p and e_l represent numbers of transitions in plant and specification. As $e_p \leq sn_p$ and $e_l \leq sn_l$, we can rewrite the complexity as $O(sn_pn_l + s_us^2n_pn_l)$. Interested readers are referred to [61] for verification algorithms for nonblocking and controllable.

6.4 Verify Interface Consistency

For a given n^{th} degree ($n \geq 1$) hierarchical interface based system composed of DES $\mathbf{G}_H, \mathbf{G}_{L_1}, \ldots, \mathbf{G}_{L_n}, \mathbf{G}_{I_1}, \ldots, \mathbf{G}_{I_n}$, we need to verify that all the interface consistency conditions in Definition 3.4.2 are satisfied.

First we will verify that the system respects the alphabet partition given by Equation 3.1. Then we will check the listed six interface consistency properties.

6.4.1 Alphabet Partition

We need to verify that

$$\Sigma \;=\; \Sigma_H \,\dot\cup \bigdot_{k\,=\,1,\ldots,n}(\Sigma_{L_k} \,\dot\cup\, \Sigma_{R_k} \,\dot\cup\, \Sigma_{A_k}).$$

By definition of disjoint union, we have $n_s = |\Sigma| = |\Sigma_H| + \Sigma_{k=1,\ldots,n}(|\Sigma_{L_k}| + |\Sigma_{R_k}| + |\Sigma_{A_k}|)$. Here we use $|\Sigma|$ to mean the number of elements in the set Σ.

To check that the two sets are disjoint, we can use a variation of Algorithm 6.2. Essentially, we would remove the S set from the algorithm, set $n_3 = |A_1| + |A_2|$, remove lines 4-8, the uMergeSort call for set S, and lines 15-18, and return the A_0 array (we will need this shortly). The complexity would still be $O(n_3 \log n_3)$. If we use n_s as an upper bound for n_3, we see that the comparison for any two of our sets would be $O(n_s \log n_s)$.

Since we have one high level with event set Σ_H and n low levels each having three event sets (Σ_{L_j}, Σ_{R_j} and Σ_{A_j}) associated with them, we thus have $3n + 1$ event sets in total. To directly verify that all sets are pairwise disjoint, we could compare every two combinations out of the $3n + 1$ possible sets to see if they are disjoint. This would require $C_2^{3n+1} = \frac{9n^2+3n}{2}$ comparisons. [2] However, we can actual check this condition using a maximum of $3n$ comparisons.

[2] $C_m^n := \frac{n!}{m!(n-m)!}$

Imagine a complete binary tree structure, with each event set as a leaf node. [3] We then apply our comparison algorithm that we discussed above to every two sets who share the same parent node. After comparing the two sets, we then replace their parent node with the resulting set from the disjoint union (array A_0 discussed above). If we have an odd number of sets at a given level, we move the rightmost node that represents a set, up one level to replace its parent node.

We repeat this process from bottom up until we have replaced the root node. If any of these steps we find the two sets we are comparing are not disjoint, the process terminates and we conclude that the alphabet partition does not satisfy the requirement, and thus the system is not interface consistent.

We will need to call our comparison algorithm at most $3n$ times. Clearly, the n_3 value for any comparison will be less than n_s. We thus have time complexity $O(3n) \cdot O(n_s \log n_s) = O(n \cdot n_s \log n_s)$.

In the remainder of this chapter we will assume that the disjoint union property has already been verified and we will use $\cup$ (normally set union) instead of $\dot{\cup}$ in our discussions.

6.4.2 Multi-level Properties

Multi-level property Point 1 can be verified by comparing event set of $\mathbf{G_H}$ and Σ_{IH}, and event set of $\mathbf{G_{L_j}}$ and Σ_{IL_j}, for $j = 1 \ldots n$. We need do $n+1$ comparisons in total.

Let n_s be the size of alphabet Σ and let n_D be an upper bound for the number of component DES that make up $\mathbf{G_H}$ and the $\mathbf{G_{L_j}}$. We assume that Σ_{IH}, and each Σ_{IL_j} are represented by a Boolean array of size n_s. As each event has an index number greater than or equal to one, we can specify whether an event in Σ belongs

[3] A complete binary tree is a binary tree which has at every level except possibly the lowest, completely filled. At the lowest level, all nodes must be as far left as possible.

to a given set by marking the corresponding entry as *true*. We can thus test set membership for a given event in constant time.

To determine the event set used by $\mathbf{G_H}$, we first initialize each entry in its array to false. This has time complexity $O(n_s)$. We then process the event set of each component DES and mark each corresponding array entry as *true*. This process is $O(n_s n_D)$. We can now check that every entry of this array is set to *true* if and only if the corresponding entry of the array for Σ_{IH} is set to *true* which will tell us if the sets are equal. This process is $O(n_s)$. Our entire process is thus $O(2n_s + n_s n_D) = O(n_s n_D)$. We can use a similar approach to check the n low levels. We thus find the total time complexity for Point 1 to be $O(nn_s n_D)$.

Multi-level property Point 2 requires that all interface DES are command-pair interface. We can verify this by applying Algorithm 6.3 (Section 6.2) to each $\mathbf{G_{I_j}}$, for $j = 1 \ldots n$. Let n_x be the largest state size of of all the interfaces. As Algorithm 6.3 runs in $O(n_s \log n_s + n_x \cdot n_s)$ time, we can thus check Point 2 in $O(nn_s \log n_s + nn_x \cdot n_s)$.

6.4.3 High Level Property

The high level property (Point 3 of Definition 3.4.2) is similar to the controllability definition. If we take $\mathbf{G_{I_j}}$ as the plant DES, $\mathbf{G}_H||\mathbf{G}_{I_1}||\mathbf{G}_{I_{j-1}}||\mathbf{G}_{I_{j+1}}||\mathbf{G}_{I_n}$, $\Sigma_u = \Sigma_{A_j}$ as the supervisor DES and $\Sigma_c = \Sigma - \Sigma_{A_j}$, we can then apply a normal controllability check for each $j \in \{1, 2, \ldots, n\}$. We also note that the synchronous product of the plant and the supervisor is the same in each case, so we can then do only one synchronization but check for each version of the controllability property at each reachable state.

When carrying out our check, we store pointers to the component DES in an array and put them in the following order: plant, specification and then interface DES. Let n_{HP} and n_{HS} be the number of plant and specification DES in the high

level, respectively. When checking the high level property, we need to use all the plant and specification DES from the high level, plus all n interface DES. The total number of DES involved is thus

$$m_H = n_{HP} + n_{HS} + n.$$

The algorithm to check Point 3 is given below in Listing 6.4.

Listing 6.4: Interface Consistency Pt 3 Check

```
1  bool Pt3Check()
2  begin
3    Σ_IH ←Σ_H ∪ ⋃_{k=1..n}(Σ_{A_k} ∪ Σ_{R_k});
4    n_HP ← number of high level plant DES;
5    n_HS ← number of high level specification DES;
6    m_H ← n_HP + n_HS + n;
7    for k← 1 to m_H do
8      Fill transition matrix (DES k):δ_H(k, x, σ);
9    end for
10   s_0 ← < x_{1_0}, x_{2_0}, ..., x_{m_H0} >;      //Tuple of initial states from all DES
11   pending ← {s_0};
12   found ← {s_0};
13   while pending ≠ ∅ do
14     s = < x_1, x_2, ..., x_{m_H} > ← extract element from pending;
15     for each σ ∈Σ_IH do
16       undefined ← false;
17       for i ← 1 to m_H do
18         if δ_H(i, x_i, σ) ! then
19           x_i' ←δ_H(i, x_i, σ) ;
20         else
21           undefined ← true;
22           break;
23         end if
```

```
24        end for
25        if undefined then
26          if i ≤ n_HP + n_HS then
27            for j ← n_HP + n_HS + 1 to m_H do
28              if δ_H(j, x_j, σ) ! and σ ∈Σ_{A_j} then
29                return false;
30              end if
31            end for
32          end if
33        else
34          s' ← < x_1‘, x_2‘, ..., x_{m_H}‘ > ;
35          if s' ∉ found then
36            pending ← pending ∪ {s'};
37            found← found ∪ {s'};
38          end if
39        end if
40      end for
41    end while
42    return true;                    //pt 3 check pass
43  end
```

Let $n_{sH} = |\Sigma_{IH}|$. Let n_H be the number of states of the largest DES (in terms of state size) among the m_H DES. Let n_x be state size of the synchronous product of $\mathbf{G_H}, \mathbf{G_{I_1}}, \ldots, \mathbf{G_{I_n}}$.

In Line 3, we simply copy the indicated event sets into a large array, which takes linear time. Lines 7-9 constructs the transition table for each component DES. As transition lists are stored as linked lists in DES, we need to construct the table in able to quickly determine if a transition is defined at a given state, and where the transition takes us. We construct the array for a given DES as follows: For each state of the DES, we initialize the entry for each event in Σ_{IH} to be to that

state (ie. set it to a selfloop). This is to account for events not in the event set of the DES. This effectively converts the synchronous product to the corresponding *meet* operation. We then loop through every event in the event set of the DES and set that entry to zero which indicates no transition with that event label defined as the state indexes start at one. We now loop through the transition list (stored as a linked list) for the that state. Since we assume the DES is deterministic, we can have maximum n_{sH} transitions at a given state. For each one we find, we set the corresponding array index to the indicated next state. As we do this for every state of each DES, this operation is thus $O(m_H \cdot n_H \cdot n_{sH})$.

The *while* block, lines 13-41, goes over the state space of the synchronous product DES, which is worst case $n_x \leq n_H^{m_H}$ states. The for loop in lines 15-40 runs n_{sH} times, while the two parallel for loops in lines 17-24 and lines 27-31 each run $O(m_H)$ times. Access to the trie representing the *found* set is also $O(m_H)$. The running time for this algorithm is thus $O(m_H \cdot n_H \cdot n_{sH}) + O(n_H^{m_H} \cdot n_{sH} \cdot m_H)$ which is dominated by the $O(n_H^{m_H} \cdot n_{sH} \cdot m_H)$ term.

6.4.4 Low Level Properties

Point 4, 5 and 6 are all low level properties, which allows us to check them all together and reduce the traversals required over a given low level subsystem.

We store pointers to the component DES in an array and organize them in the order: plant, specification and then interface DES. Let n_{L_iP} and n_{L_iS} be number of plant and specification DES in the i^{th} low level ($i \in \{1, 2, \ldots, n\}$), respectively. For the i^{th} low level, the total number of DES involved in the interface consistent check is the number of plants and specifications plus one interface, thus

$$m_{L_i} = n_{L_iP} + n_{L_iS} + 1.$$

We will use the data structures listed below in our algorithms to check the low level properties.

- *R-Reachable* contains states from the synchronous product for a given low level which can be reached by a request event, as well as the corresponding component state belonging to the interface for that low level. We use this list to check property 5 by calling function searchAnswer given in Algorithm 6.6 for each state (from the synchronous product) in R-Reachable.

- *5-Consistent* contains states from R-Reachable that we have already called function searchAnswer. This is to prevent duplicate searches when a given state can be reached by request events from multiple other states.

- *I-Marked* contains states that are not marked in the synchronous product of a given low level, but their corresponding interface component state is marked in the interface for that low level. We use this list to verify Point 6.

Listing 6.5: Interface Consistency Pt 4, 5 and 6 Check

```
1  bool LowICCheck()
2  begin
3    for i=1 to n do                    //Go over all n low levels
4      Σ_{IL_i} ← Σ_{L_i} ∪ Σ_{R_i} ∪ Σ_{A_i} ;
5      n_{L_iP} ← number of i^{th} low level plant DES;
6      n_{L_iS} ← number of i^{th} low level specification DES;
7      m_{L_i} ← n_{L_iP} + n_{L_iS} + 1;
8      for k← 1 to m_{L_i} do
9        Fill transition matrix (DES k): δ_i(k, x, σ);
10     end for
11     s_0 ← < x_{1_0}, x_{2_0}, ..., x_{m_{Li0}} > ; //Tuple of initial states from all DES
12     pending ← {s_0};
13     found ← {s_0};
14     R–Reachable ← ∅;
15     5–Consistent ← ∅;
16     I–Marked ← ∅;
```

```
17    markedStates ← ∅;
18    //begin pt 4 check
19    while pending ≠ ∅ do
20      s = < x_1, x_2, ..., x_{m_Li} > ← pop pending;
21      for each σ ∈ Σ_{IL_i} do
22        undefined ← false;
23        marked ← true ;
24        for j ← 1 to m_{L_i} do    // j represents a given component
25          if δ_i(j, x_j, σ) ! then    // DES at low level i
26            x_j' ← δ_i(j, x_j, σ) ;
27            if not x_j'.marked then
28              marked ← false;
29            end if
30          else
31            undefined ← true;
32            break;
33          end if
34        end for
35        if undefined and σ ∈Σ_{R_i} and    δ_i(m_Li, x_{m_Li}, σ)  ! then
36        //defined in the interface
37            return Pt 4 fails;
38        else
39          s' ← < x_1‘, x_2‘, ..., x_{m_Li}‘ > ;
40          if s' ∉ found then
41            if marked then
42              s'.marked ← true;
43              markedStates ← markedStates ∪ {s'}
44            end if
45            push pending, s';
46            push found, s';
47            if x_{m_Li}'.marked and not s'.marked then
```

```
48              I–Marked ← I–Marked ∪ {s'};
49            end if  //save for pt 6 check
50          end if
51          if σ ∈Σ_{R_i} then
52            R–Reachable ← R–Reachable ∪ {(s', x_{m_{Li}}')};
53          end if
54        end if
55      end for
56    end while
57    //end pt 4 check
58    //begin pt 5 check
59    while R–Reachable ≠ ∅ do
60      (s, x) ← extract item from R–Reachable;
61      if s ∉ 5–Consistent then
62        answers ← answer events defined at x in interface;
63        visited[] ← false; //Array of bool, shows states visited
64        if SEARCHANSWER(s, answers, visited, i) then
65          5–Consistent ← 5–Consistent ∪ {s};
66        else
67          return pt 5 fails;
68        end if
69      end if
70    end while
71    //end pt 5 check
72    //begin pt 6 check
73    if I–Marked ≠ ∅ then
74      pending ← markedStates;
75      found ← pending;
76      while pending ≠ ∅ do
77        s ← extract element from pending;
78        for each inverse transition (t, σ) of s do
```

```
79            if σ ∈Σ_{L_i} ∧ t ∉ found then
80              found ← found ∪ {t};
81              pending ← pending ∪ {t};
82            end if
83          end for
84        end while
85        if not I–Marked ⊆ found then
86          return Pt 6 fails;
87        end if
88      end if
89      //end pt 6 check
90    end for //End of loop from Line 3
91 end
```

We first give function SEARCHANSWER and analyze its complexity and then give the complexity analysis for Algorithm 6.5.

Listing 6.6: explore DES for indicated answer events from s

```
1 bool SEARCHANSWER (s, answers, visited, i)
2 //Check if all answer events defined at x_I are reachable from s
3 begin
4   if answers = ∅ then
5     return true;
6   end if
7   visited(s) ← true;
8   for each transition (s, σ) of s, leading to some state t do
9     if σ ∈ answers then
10      answers ← answers − {σ};
11    else if σ ∈Σ_{L_i} then
12      if not visited(t) then
13        if SEARCHANSWER (t, answers, visited, i) then;
14          return true;
```

```
15          end if
16        end if
17      end if
18    end for
19    if answers = ∅ then
20      return true;
21    else
22      return false;
23    end if
24  end
```

Let $n_{sL_i} = |\Sigma_{IL_i}|$. Let n_{L_i} be the number of states of the largest largest DES (in terms of state size) among the m_{L_i} DES of the i^{th} low level ($i \in \{1, 2, \ldots, n\}$). Lines 4-6 of Listing 6.6 terminates searching if we have found a path to all the required answer events. The for loop in lines 8-18 searches all transitions from the given state for answer events in the *answers* set. Whenever it finds an element in the set, it removes it from the set. It also extends the search to states that can be reached from the current state by low level events (lines 11-17). Line 12 ensures each state of the DES is visited no more than once. As we can remove an event from the *answers* set in line 10 without immediately checking if the set is now empty, it's possible we could exit the for loop with the *answers* set empty. We thus check for this in lines 19-21. The worst case scenario of this algorithm is that it needs to traverse all the transitions in the synchronous product, which is $O(n_{sL_i} \cdot n_{L_i}^{m_{L_i}})$.

We now analyze the time complexity of Algorithm 6.5. Since there are n low levels, we need to check Points 4, 5, 6 n times (Line 3-88). We now show the run time for the i^{th} low level. We have m_{L_i} component DES for the i^{th} low level, and each DES has no more than n_{L_i} states.

Line 4 copies the indicated event sets into a large array. This step takes linear

time $O(n_{sL_i})$. Similar to algorithm PT3CHECK in listing 6.4, lines 8-10 construct a transition matrix for all component DES in the i^{th} low level. The matrix is constructed in $O(m_{L_i} \cdot n_{L_i} \cdot n_{sL_i})$ time.

We now examine the complexity of checking Point 4 (lines 18-57). The *while* block (lines 19-56) goes over the state space of the synchronous product, which is worst case $n_{L_i}^{m_{L_i}}$. The for loop in lines 21-55 repeats n_{sL_i} times to check every event in low level i. This for loop also contains the for loop in lines 24-34 which goes over every DES, and thus runs m_{L_i} times. In parallel to the inner for loop, we also access the found variable (lines 40 and 46) which is implemented as a trie data structure. This access is $O(m_{L_i})$. Putting things together, we find that the outer for loop (21-55) is $O(n_{sL_i} \cdot m_{L_i})$. Combining this with the while loop, and we find that checking Point 4 check, takes $O(n_{sL_i} \cdot m_{L_i} \cdot n_{L_i}^{m_{L_i}})$. We note that checking Point 4 dominates line 4 and lines 8-10, and can thus be covered by the value for the Point 4 check.

We now consider the Point 5 check (lines 58-71). In the worst case, R-Reachable can contain an entry for every transition in the synchronous product. The *while* loop (lines 59-70) thus runs $O(n_{sL_i} n_{L_i}^{m_{L_i}})$. However, the if statement at line 61 ensures that the core loop only gets executed $O(n_{L_i}^{m_{L_i}})$ times. If fact, it's as if we have a *while* loop that executes $O(n_{sL_i} n_{L_i}^{m_{L_i}})$ times performing a constant number of operations each iteration, followed by a *while* loop that executes $O(n_{L_i}^{m_{L_i}})$ times and executes lines 62-68 each iteration. Computations inside the second *while* loop are additive and dominated by the SEARCHANSWER function, which takes $O(n_{sL_i} \cdot n_{L_i}^{m_{L_i}})$. So the Point 5 check thus takes $O(n_{sL_i} n_{L_i}^{m_{L_i}}) + O(n_{L_i}^{m_{L_i}}) \cdot O(n_{sL_i} \cdot n_{L_i}^{m_{L_i}}) = O(n_{sL_i} \cdot n_{L_i}^{2m_{L_i}})$.

The Point 6 check (lines 72-88) is similar to the normal *coreachability* check. The main difference being that we only traverse inverse transitions that are labelled by events in Σ_{L_i}. The *while* loop runs worst case once per state in the synchronous product, thus $O(n_{L_i}^{m_{L_i}})$ times. Each iteration, it processes all inverse transitions for

the current state being examined. The *for* loop at lines 78-83 thus runs $O(n_{sL_i} \cdot n_{L_i}^{m_{L_i}})$ times. The *while* loop thus appears to take $O(n_{sL_i} \cdot n_{L_i}^{2m_{L_i}})$ time. However, we note that the *while* loop is in effect processing each inverse transition in the synchronous product once. As each inverse transition corresponds to a unique transition, we can thus have maximum $O(n_{sL_i} \cdot n_{L_i}^{m_{L_i}})$ unique inverse transitions in a deterministic DES. We thus conclude that the *while* loop executes $O(n_{sL_i} \cdot n_{L_i}^{m_{L_i}})$ times.

After searching all transitions, we check whether all states in I-Marked were reached in our search (lines 85-87). This takes $O(n_{L_i}^{m_{L_i}})$ steps. Running time for the Point 6 check is dominated by the *while* loop, which is $O(n_{sL_i} \cdot n_{L_i}^{m_{L_i}})$.

Adding the running time together for the construction of the transition matrices and the checking of interface consistent properties Point 4, 5 and 6, and then repeating n times, we have the running time for Algorithm 6.5 as $O(n) \cdot (O(n_{sL_i} \cdot m_{L_i} \cdot n_{L_i}^{m_{L_i}}) + O(n_{sL_i} \cdot n_{L_i}^{2m_{L_i}}) + O(n_{sL_i} \cdot n_{L_i}^{m_{L_i}})) = O(n \cdot (n_{sL_i} \cdot m_{L_i} \cdot n_{L_i}^{m_{L_i}} + n_{sL_i} \cdot n_{L_i}^{2m_{L_i}}))$. As typically we have $n_{L_i}^{m_{L_i}} \gg m_{L_i}$, we can normally simplify this to $O(n \cdot n_{sL_i} \cdot n_{L_i}^{2m_{L_i}})$.

6.5 Interface Consistent Synthesis

When applying HISC synthesis algorithms to generate supervisors from given plants, specifications and interfaces, we want to make sure that the generated supervisors satisfy the interface consistency, level-wise nonblocking, and level-wise controllability definitions. We verify that the system is HISC-valid when we load the component DES into memory. This provides us with the starting point for our synthesis. For the high level subsystem, we want to make sure that it satisfies Point 3 of the interface consistency definition as well as Point I of the level-wise nonblocking definition and Point III of the level-wise controllability definition. For the low levels, we need to make sure that Points 4, 5 and 6 of the interface consistency definition as well as Point II of the level-wise nonblocking definition and

Point II of the level-wise controllability definition. By applying our algorithms on the given HISC structured system, we will produce a set supervisors such that the system, with the specifications replaced by these supervisors, is interface consistent level-wise nonblocking and level-wise controllable.

6.5.1 High Level Interface Consistent Supcon

The objective of the traditional supcon algorithm ([77, 61]) is to construct the maximally permissive controllable sublanguage of a given specification language E, with respect to the plant G. The objective of HISC synthesis is to create level-wise maximally permissive controllable sublanguages of the given specifications, while making sure that the resulting system satisfies the interface properties.

The high level synthesis algorithm is based on the synchronous product operation. It trims off states that violate the high level portion of the level-wise controllability, nonblocking and interface consistency definitions. Based on the synchronous product of the plant, specification DES, and interfaces, we trim off three kinds of state: uncontrollable, non-interface consistent and blocking states. Since trimming off any single state can possibly change the other properties, we need to keep trimming until all three properties are satisfied in the resulting DES. At that point no more states need to be trimmed off and we have reached a fixpoint for the synthesis algorithm.

In this section we list the iSupconHigh algorithm which implements the high level fixpoint operator Ω_H. iSupconHigh calls two functions trimState and trimDesHigh. trimState trims off a state in a controllable, high level interface consistent fashion, This means that when asked to trim a given state s, it will recursively trim s and any state s' that connects to s if s' fails the controllability property or Point 3 of the interface consistency property. Since the index of states in our DES start from 1, we can mark the trimmed states with index 0,

and then clean the trimmed states at the end of the synthesis algorithm. Function TRIMDESHIGH will typically do the bulk of the required work. It returns true if there is no need to trim another state. Definitions of these functions will follow. Also in the following algorithm, we will use pseudo code *addState* (add a state to the result DES. Also assigns state a unique index with value greater than one), $addTrans$ (add a transition to a state), $addInverseTrans$ (add an inverse transition to a state), $addToMarkerState$(adds state to list of marker states), $removeMarkerState$ (removes state from list of marker states), *removeState* (removes state from DES). All three add a node to a linked list and take constant time. We use *states* and *marker_states* to represent linked lists of states and marker states for a DES. A traversal of either lists takes linear time.

When running our algorithms, we store pointers to the component DES in an array and put them in the following order: plant, specification and then interface DES. Similar to algorithm PT3CHECK, we define n_{HP} and n_{HS} to be the number of plant and specification DES in the high level, respectively. We will use a total number of

$$m_H = n_{HP} + n_{HS} + n$$

DES in the high level synthesis, where n represents number of low levels and thus the number of interfaces.

Listing 6.7: High Level Synthesis

```
1 DES iSupconHigh()
2 begin
3   Σ_IH ←Σ_H ∪ ⋃_{k=1..n}(Σ_{A_k} ∪ Σ_{R_k});
4   n_HP ← number of high level plant DES;
5   n_HS ← number of high level specification DES;
6   m_H ← n_HP + n_HS + n;
7   for k← 1 to m_H do
8     Fill transition matrix (DES k):δ_H(k, x, σ);
9   end for
```

```
10  x0 ← < x1_0, x2_0, ..., xmH0 > ;      //Tuple of initial states from all DES
11  pending ← {x0};
12  found ← {x0};
13  DES resultDES;
14  resultDES.addInitialState(x0);
15  while pending ≠ ∅ do
16    x =< x1, x2, ..., xmH > ← extract element from pending;
17    for each σ ∈ ΣIH do
18      undefined ← false;
19      marked ← true;
20      for i ← 1 to mH do
21        if δH(i, xi, σ) ! then
22          xi' ←δH(i, xi, σ) ;
23          if xi'.marked = false then
24            marked ← false;
25          end if
26        else
27          undefined ← true;
28          break;
29        end if
30      end for
31      if undefined then
32        untrimmed ← true;
33        if i ≤ nHP + nHS then
34          for j ← nHP + nHS + 1 to mH do
35            if δH(j, xj, σ) ! and σ ∈ ΣAj then
36              //fail high level property
37              TRIMSTATE(x);
38                  untrimmed ← false;
39              break;
40            end if
```

```
41          end for
42        end if
43        if i > n_HP and i ≤ n_HP + n_HS and σ ∈ Σ_u
44                and untrimmed then
45          // as i > n_HP we know the σ transition is
46          // defined in all of the plants
47          specBlocked ← true;
48          for j ← n_HP + n_HS + 1 to m_H do
49            if not δ_H(j, x_j, σ) ! then
50              specBlocked ← false;
51              break;
52            end if
53          end for
54          if specBlocked then     //uncontrollable event defined for
55                                  //all plants, and interfaces
56            trimState(x);
57          end if
58        end if
59      else    // not undefined
60        x' ← < x_1', x_2', ..., x_{m_H}' > ;
61        if x' ∉ found then
62          if marked then
63            x'.marked ← true;
64            resultDES.addToMarkerState(x');
65          end if
66          resultDES.addState(x');
67          pending ← pending ∪ {x'};
68          found ← found ∪ {x'};
69        end if
70        if x'.index > 0 then
71          x.addTrans(x', σ);
```

```
72            x'.addInverseTrans(x, σ);
73          end if
74        end if
75      end for
76    end while
77    reach[] ← false;
78    reach[x_0] ← true;
79    pending ← {x_0};
80    found ← {x_0};
81    while pending ≠ ∅ do
82      x ← extract element from pending;
83      for each t ∈ x.trans do
84        x' ← t.state;
85        if x' ∉ found then
86          pending ← pending ∪ {x'};
87          found ← found ∪ {x'};
88          reach[x'] ← true;
89        end if
90      end for
91    end while
92    while TRIMDESHIGH(resultDES, reach) do
93      ;  //keep trimming until no changes happen
94    end while
95    for each x ∈ resultDES.states do
96      if not reach[x] then
97        resultDES.removeState(x);
98      end if
99    end for
100   return resultDES;
101 end
```

We now briefly discuss how the iSupconHigh algorithm works. We defer the complexity analysis until after we have presented and analyzed the trimState trimDesHigh algorithms.

In lines 15-75, we construct a possibly restricted synchronous product for the high level. As we process each state, we check to see if it violates Point 3 of the interface consistency definition, as well as Point III of the level-wise controllability definition. We do this by examining each event that does not have a transition defined at that state. We check Point 3 on lines 33-42. If $i \leq n_{HP} + n_{HS}$, we know that a plant or spec DES does not allow the event to occur at the current state. We then check each interface to see if the event is allowed by the interface, and if so, whether the event belongs to the interfaces set of answer events. If both true, then Point 3 fails and we call trimState to remove the state.

In lines 43-58, we check to make sure the state satisfies Point III of the level-wise controllability definition. If $i > n_{HP}$ and $i \leq n_{HP} + n_{HS}$ and the event is uncontrollable, then we know that a specification is trying to disable an uncontrollable event. As $i > n_{HP}$, we know that the event can occur in all of the plant components at this state. We then check to see if any interfaces prevents the event. If they all allow it, then the state fails Point III and we call trimState to remove the state.

After we have determined that an event has a transition defined at our current state, we process it in lines 59-73. Lines 61-68 ensure we only process each state once. Lines 69-72 ensure that the state we have reached has not already been found and trimmed from the DES before we add transitions to/from the state.

As it is possible that our DES may contain some unreachable states, we do a reachability check (lines 77-90) and store the results in variable *reach*. We pass *reach* into function trimDesHigh which will update the *reach* if it needs to trim off any states.

Now that the synchronous product has been constructed, we call function TRIMDESHIGH to make the result DES nonblocking. This function call mainly implements the Ω_{HNB} operator, but it also trims away states in a controllable and Point 3 consistent fashion. This is equivalent to make the DES nonblocking, followed by a pass to make sure the DES is still controllable and satisfies Point 3 of the interface consistency definition. We loop (lines 92-94) until calls to TRIMDESHIGH cause no change to our DES, meaning we have hit a fixpoint.

Now that we have reached a fixpoint, we remove from the DES all states that are unreachable.

In the following algorithms, a state x has a transition list *trans* and an inverse transition list *inverseTrans*. For some state x', we use pseudo code $x.removeTrans(x')$ to remove any transitions from x to x'. Similarly, we use $x.removeInverseTrans(x')$ to remove any inverse transitions that correspond to transitions from x to x'. We start by presenting TRIMSTATE.

Listing 6.8: TrimState

```
1  void TRIMSTATE(State x)
2  begin
3    if x.index = 0 then
4      return;
5    end if
6    x.index ← 0;      //mark the state as trimmed
7    for each t ∈ x.trans do
8      x1 ← t.state;
9      x1.removeInverseTrans(x);
10   end for
11   for each t ∈ x.inverseTrans do
12     x1 ← t.state;
13     e ← t.event;
14     x1.removeTrans(x);
```

```
15      if e ∈ Σ_u or e ∈ Σ_A then
16          trimState(x_1);
17      end if
18    end for
19  end
```

trimState2 given in Algorithm 6.9 is a variation of trimState. The only difference between these two is that Algorithm 6.9 is used in the final stage of the synthesis procedure, thus it adds checking for reachability. As Algorithm 6.8 is used during the creation of the synchronous product, it's possible that a state may only temporarily become unreachable and thus it should not be removed at that point.

Listing 6.9: Trim State With Reachable Check

```
1  void trimState2(x, reach)
2  begin
3    if x.index = 0 then
4      return;
5    end if
6    x.index ← 0;      //mark the state as trimmed
7    reach[x] ← false;
8    for each t ∈ x.trans do
9      x_1 ← t.state;
10     x_1.removeInverseTrans(x);
11     if x_1.inverseTrans = ∅ then    //x_1 no longer reachable
12       trimState2(x_1);
13     end if
14   end for
15   for each t ∈ x.inverseTrans do
16     x_1 ← t.state;
17     e ← t.event;
```

```
18      x1.removeTrans(x);
19      if e ∈ Σu or e ∈ ΣA then
20        trimState2(x1);
21      end if
22    end for
23  end
```

We now give the trimDesHigh algorithm, followed by a complexity analysis for algorithms 6.7- 6.10. We use an array *visited* to mark coreachable states, and at the end of the algorithm we trim off all non coreachable states to make the DES nonblocking. We use *states* and *marker_states* to represent linked lists of states and marker states for a DES. A traversal of either list takes linear time. Below, we refer to the **EMPTY** DES. This is a placeholder representing any DES with no states. This means that the closed behavior and the marked language of the **EMPTY** DES are both empty.

Listing 6.10: High Level DES Trim Function

```
1  bool trimDesHigh(des, reach)
2  begin
3    changed ← false;
4    if des = EMPTY then
5      return changed;
6    end if
7    pending ← ∅;
8    visited[] ← false;
9    for each x ∈ des.marker_states do
10     if reach[x] then
11       pending ← pending ∪ {x};
12       visited[x] = true;
13     else
14       des.removeMarkerState(x);
```

```
15      end if
16    end for
17    if pending = ∅ then
18      // means no reachable marker states
19      des ← EMPTY;
20      return changed;
21    end if
22    while pending ≠ ∅ do
23      x ← extract element from pending;
24      for each r ∈ x.inverseTrans do
25        x1 ← r.state;
26        if not visited[x1] then
27          visited[x1] = true;
28          pending ← pending ∪ {x1};
29        end if
30      end for
31    end while
32    for each x ∈ des.states do
33      if not visited[x] then
34        TRIMSTATE2 (x, reach);
35        changed ← true;
36      end if
37    end for
38    return changed;
39  end
```

As defined earlier, m_H is the number of high level DES (plants + specs) plus number of interfaces (n), $n_{sH} = |\Sigma_{IH}|$, and n_H is the number of states of the largest DES (in terms of state size) among the m_H DES. Let n_x be state size of the synchronous product of $\mathbf{G_H}, \mathbf{G_{I_1}}, \ldots, \mathbf{G_{I_n}}$. We can see that the synchronous product has worst case $n_x \leq n_H^{m_H}$ states.

Algorithm 6.8 and 6.9 check a state's transitions, which in the worst case have n_{sH} transitions and $n_H^{m_H} n_{sH}$ inverse transitions. When trimming one state, the algorithm may also go to another state and start trimming that state off by looping over the other state's transition and inverse transition lists and so on. The algorithm would appear to be $O(n_{sH} \cdot n_H^{2m_H})$ due to the number of possible reverse transitions at each state. However, as each reverse transition corresponds to a unique transition, there can be a maximum of $n_H^{m_H} n_{sH}$ such transitions in a deterministic DES. As this algorithm can be seen as looping over each all reverse transitions in the DES, it is thus actually $O(n_{sH} \cdot n_H^{m_H})$.

In Algorithm 6.10, initializing the *visited* array takes $O(n_H^{m_H})$. The *for* loop in lines 9-16 also takes $O(n_H^{m_H})$. We note that as we are looping through the linked list *marker_states*, we do not have to search to remove the current marked state, thus its removal can be done in constant time.

The *while* loop in lines 22-31 runs $O(n_H^{m_H})$ times. Within the *while* loop, the *for* loop in lines 24-30 runs $O(n_{sH} n_H^{m_H})$ times. This would appear that the *while* loop is $O(n_{sH} n_H^{2m_H})$, but as discussed above, a deterministic DES can have a maximum of $n_H^{m_H} n_{sH}$ reverse transitions so the *while* loop is actually $O(n_{sH} n_H^{m_H})$.

We now note that the *for* loop in lines 32-37 runs $O(n_H^{m_H})$ times. Within the *for* loop, function TRIMSTATE2 runs $O(n_{sH} \cdot n_H^{m_H})$. It would appear that the *for* loop is $O(n_{sH} \cdot n_H^{2m_H})$. However, as a state can only be trimmed once, implemented by lines 3-6 in both Algorithm 6.8 and 6.9, it is actually $O(n_{sH} \cdot n_H^{m_H})$. Putting everything together, we see that Algorithm 6.10 is $2 \cdot O(n_H^{m_H}) + O(n_{sH} \cdot n_H^{m_H}) + O(n_{sH} \cdot n_H^{m_H}) = O(n_{sH} \cdot n_H^{m_H})$.

Now we investigate time complexity for Algorithm 6.7. Similar to Algorithm 6.4, line 3 takes $O(n_{sH})$ time , and lines 7-9 takes $O(m_H \cdot n_H \cdot n_{sH})$ time. The *while* block (lines 15-76) goes over the state space of the result DES, and thus loops $O(n_H^{m_H})$ times.

Within the *while* loop, the *for* loop in lines 17-75 runs n_{sH} times, and the three parallel inner *for* loops in lines 20-30, lines 34-41, and lines 48-53 each run $O(m_H)$ times. Also in parallel to the inner *for* loops are accesses to the *found* variable on lines 37, and 56, and a call to TRIMSTATE on line 56. Accessing the *found* variable is $O(m_H)$, and TRIMSTATE runs $O(n_{sH} \cdot n_H^{m_H})$. Within the *for* loop on lines 34-41, we have a call to TRIMSTATE on line 37. The complexity for this *for* loop is $O(m_H) + O(n_{sH} \cdot n_H^{m_H}) = O(m_H + n_{sH} \cdot n_H^{m_H})$ as TRIMSTATE can be called at most once.

Putting everything together, the *for* loop in lines 17-75 appears to be $O(n_{sH}) \cdot (O(m_H) + O(n_{sH} \cdot n_H^{m_H})) = O(n_{sH}(m_H + n_{sH} \cdot n_H^{m_H}))$. As a state can be trimmed only once, the complexity is actually $O(n_{sH} \cdot m_H + n_{sH} \cdot n_H^{m_H})$.

We now see that the *while* loop (lines 15-76) appears to be $O(n_H^{m_H}) \cdot O(n_{sH} \cdot m_H + n_{sH} \cdot n_H^{m_H}) = O(n_{sH} \cdot m_H \cdot n_H^{m_H} + n_{sH} \cdot n_H^{2m_H})$. Again, as a state can only be removed once, we can argue as above that the *while* loop is actually $O(n_{sH} \cdot m_H \cdot n_H^{m_H} + n_{sH} \cdot n_H^{m_H}) = O(n_{sH} \cdot m_H \cdot n_H^{m_H})$.

We next note that the array *reach* (line 77) can be initialized in $O(n_H^{m_H})$ time. The variable *found* can be now implemented as an array, and thus be accessed (lines 85 and 87) in constant time. The *while* loop in lines 81-91 runs in $O(n_{sH} \cdot n_H^{m_H})$ time.

The *while* block in lines 92-94 calls Algorithm 6.10, where each call takes $O(n_{sH} \cdot n_H^{m_H})$ time. The worst case is that Algorithm 6.10 only trims one state each time and finally all states need to be trimmed off. The while block will thus run $O(n_H^{m_H})$ times, making the total run time $O(n_H^{m_H}) \cdot O(n_{sH} \cdot n_H^{m_H}) = O(n_{sH} \cdot n_H^{2m_H})$. However, the function TRIMDESHIGH is typically only called a constant number of times [4]. Therefore the running time for the *while* block will typically be $O(n_{sH} \cdot n_H^{m_H})$. Finally, the *for* loop in lines 95-99 runs $O(n_H^{m_H})$ times.

[4] In a standard supcon algorithm, there is a similar trim function. Rudie [61] shows that a standard supcon algorithm normally calls the trim function once. And if there are multiple calls, the number of calls is effectively constant compared to the number of states of the DES.

We are now ready to determine the complexity for all of Algorithm 6.7. Putting the parallel sections together, we see that Algorithm 6.7 runs in $O(n_{sH})+O(m_H \cdot n_H \cdot n_{sH})+O(n_{sH} \cdot m_H \cdot n_H^{m_H})+O(n_{sH} \cdot n_H^{2m_H})+O(n_H^{m_H}) = O(n_{sH} \cdot m_H \cdot n_H^{m_H} + n_{sH} \cdot n_H^{2m_H})$ as typically $m_H \cdot n_H \ll n_H^{m_H}$. However, as discussed above, the $O(n_{sH} \cdot n_H^{2m_H})$ term typically turns out to be $O(n_{sH} \cdot n_H^{m_H})$. We would thus expect the algorithm to behave as $O(n_{sH} \cdot m_H \cdot n_H^{m_H})$.

6.5.2 Low Level Interface Consistent Synthesis

For the k^{th} low level of an interface system, $k \in \{1, 2, \ldots, n\}$, we would like to compute the k^{th} low level level-wise maximally permissive supervisor that satisfies the interface properties. For the low levels, we need to make sure that Points 4, 5 and 6 of the interface consistency definition as well as Point II of the level-wise nonblocking definition and Point II of the level-wise controllability definition.

In this section we list the iSupconLow algorithm which implements the k^{th} low level fixpoint operator Ω_{L_k}. We will use the same notations as defined for LowIC-Check in Section 6.4.4. Similar to high level's trimState and trimState2, we use slightly modified versions trimStateLow and trimStateLow2 to remove unwanted states.

Listing 6.11: Low Level Synthesis

```
1 bool iSupconLow ()
2 begin
3   for i=1 to n do   // (A), go over all n low levels
4     DES resultDESi
5     ΣILi ← ΣLi ∪ ΣRi ∪ ΣAi ;
6     nLiP ← number of i^th low level plant DES;
7     nLiS ← number of i^th low level specification DES;
8     mLi ← nLiP + nLiS + 1;
9     for k← 1 to mLi do
```

```
10      Fill transition matrix (DES k): δ_i(k, x, σ);
11    end for
12    s_0 ← < x_{1_0}, x_{2_0}, ..., x_{m_{Li0}} >; //Tuple of initial states from all DES
13    pending ← {s_0};
14    found ← {s_0};
15    R–Reachable ← ∅;
16    I–Marked ← ∅;
17    //begin pt 4 check
18    while pending ≠ ∅ do
19      s = < x_1, x_2, ..., x_{m_Li} > ← pop pending;
20      for each σ ∈ Σ_{IL_i}
21        undefined ← false;
22        marked ← true;
23        for j ← 1 to m_{L_i}
24          if δ_i(j, x_j, σ) ! then
25            x_j' ← δ_i(j, x_j, σ) ;
26            if not x_j'.marked then
27              marked ← false;
28            end if
29          else
30            undefined ← true;
31            break;
32          end if
33        end for
34        if undefined and [(σ ∈Σ_{R_i} and δ_i(m_Li, x_{m_Li}, σ)!)
35                or (j > n_{L_iP} and σ ∈ Σ_u)] then
36          trimStateLow (s, i);
37        else
38          s' ← < x_1', x_2', ..., x_{m_Li}' > ;
39          if s' ∉ found then
40            if marked then
```

```
41              s'.marked ← true;
42              resultDESi.addToMarkerState(s');
43            end if
44            resultDESi.addState(s');
45            push pending, s';
46            push found, s';
47            if x_mLi'.marked and not s'.marked then
48              I–Marked ← I–Marked ∪ {s'};
49            end if  //save for pt 6 check
50          end if
51          if σ ∈Σ_Ri then
52            R–Reachable ← R–Reachable ∪ {(s', x_mLi')};
53          end if
54          if s'.index > 0 then
55            s.addTrans(s', σ);
56            s'.addInverseTrans(s, σ);
57          end if
58        end if
59      end for
60    end while
61    reach[] ← false;
62    reach[s_0] ← true;
63    pending ← {s_0};
64    found ← {s_0};
65    while pending ≠ ∅ do
66      s ← extract element from pending;
67      for each t ∈ s.trans do
68        s' ← t.state;
69        if s' ∉ found then
70          pending ← pending ∪ {s'};
71          found ← found ∪ {s'};
```

```
72              reach[s'] ← true;
73            end if
74          end for
75        end while
76        while trimDesLow(resultDESi, R–Reachable, I–Marked,
77                         i   , reach) do
78            ;  //keep trimming until no changes happen
79        end while
80        for each s ∈ resultDESi.states do
81          if not reach[s] then
82            resultDESi.removeState(s);
83          end if
84        end for
85      end for
86      return resultDESj, j = 1,2, .. , n;
87    end
```

We now briefly discuss how the iSupconLow algorithm works. Much is similar to the iSupconHigh algorithm, so we will only discuss the new stuff. We defer the complexity analysis until after we have presented and analyzed the trimStateLow trimDesLow algorithms.

In lines 34-36, we check whether a state violates Point 4 of the interface consistency definition, and Point II of the level-wise controllability definition. If the state fails either condition, we trim the state away from the result DES.

During the synchronization, we record a few sets (*R-Reachable* and *I-Marked*) and keep them for checking Point 5 and 6.

Finally after the synchronous product is constructed, we call function trimDesLow to make the result DES nonblocking and to ensure it satisfies Points 5 and 6 of the interface consistency definition. It also trims state away in a con-

trollable and Point 4 consistent fashion.

Listing 6.12: Low Level Trim State

```
1  void trimStateLow (x, i)
2  begin
3    if x.index = 0 then
4      return;
5    end if
6    x.index ← 0;      //mark the state as trimmed
7    for each t ∈ x.trans do
8      x_1 ← t.state;
9      x_1.removeInverseTrans(x);
10   end for
11   for each t ∈ x.inverseTrans do
12     x_1 ← t.state;
13     e ← t.event;
14     x_1.removeTrans(x);
15     if e ∈ Σ_u or e ∈ Σ_{R_i} then
16       trimStateLow (x_1, i);
17     end if
18   end for
19 end
```

trimStateLow2 given in Algorithm 6.13 is a variation of trimStateLow. The only difference between these two is that Algorithm 6.13 is used in the final stage of the synthesis procedure, thus it adds checking for reachability. As Algorithm 6.12 is used during the creation of the synchronous product, it's possible that a state may only temporarily become unreachable and thus it should not be removed at that point.

Listing 6.13: Low level Trim State With Reachable Check

```
1  void trimStateLow2(x, i, reach)
```

```
2 begin
3   if x.index = 0 then
4     return;
5   end if
6   x.index ← 0;     //mark the state as trimmed
7   reach[x] ← false;
8   for each t ∈ x.trans do
9     x1 ← t.state;
10    x1.removeInverseTrans(x);
11    if x1.inverseTrans = ∅ then    //x1 no longer reachable
12      trimStateLow2(x1, i, reach);
13    end if
14  end for
15  for each t ∈ x.inverseTrans do
16    x1 ← t.state;
17    e ← t.event;
18    x1.removeTrans(x);
19    if e ∈ Σu or e ∈ΣRi then
20      trimStateLow2(x1, i, reach);
21    end if
22  end for
23 end
```

We now give the trimDesLow algorithm, followed by a complexity analysis for algorithms 6.11-6.14. We use *states* and *marker_states* to represent linked lists of states and marker states for a DES. A traversal of either list takes linear time. Below, we refer to the **EMPTY** DES. This is a placeholder representing any DES with no states. This means that the closed behavior and the marked language of the **EMPTY** DES are both empty. We also note that we use in the algorithm below the function searchAnswer which was given earlier in Listing 6.5.

Listing 6.14: Low Level DES Trim Function

```
1 bool trimDesLow(des, R–Reachable, I–Marked, i , reach)
2 begin
3   changed ← false;
4   if des = EMPTY then
5     return changed;
6   end if
7   remain[] ← false;
8   pending ← ∅;
9   for each s ∈ des.marker_states do
10    if reach[s] then
11      pending ← pending ∪ {s};
12      remain[s] ← true;
13    else
14      des.removeMarkerState(s);
15    end if
16  end for
17  if pending = ∅ then
18    // means no reachable marker states
19    des ← EMPTY;
20    return changed;
21  end if
22  5–Consistent ← ∅
23  while pending ≠ ∅ do     //check coreachable
24    s ← pop pending;
25    for each r ∈ s.inverseTrans do
26      s' ← r.state;
27      if not remain[s'] then
28        remain[s'] ← true;
29        push pending, s';
30      end if
```

```
31    end for
32  end while
33  //check Point 5
34  for (s, x) ∈ R–Reachable do
35    if (not reach[s]) or (not remain[s]) then
36       R–Reachable ← R–Reachable − {(s, x)};
37    else
38      if s ∉ 5–Consistent then
39        answers ← answer events defined at x in interface;
40        visited[] ← false;
41        if searchAnswer(s, answers, visited, i) then
42          5–Consistent ← 5–Consistent ∪ {s};
43        else
44          //mark all states for trimming that reach s via
45          // a request event
46          for t ∈ s.inverseTrans do
47            if t.event ∈ Σ_Ri then
48                remain[t.state] ← false;
49            end if
50          end for
51        end if
52      end if
53    end if
54  end for
55  //check Point 6
56  if I–Marked ≠ ∅ then
57    // the next two lines imply that pending and found are
58    // assigned a copy of des.marker_states
59    pending ← des.marker_states;
60    found ← des.marker_states;
61    while pending ≠ ∅ do
```

```
62        s ← extract element from pending;
63        for each (s', σ) ∈ s.inverseTrans do
64          if σ ∈Σ_{L_i} ∧ s' ∉ found then
65            found ← found ∪ {s'};
66            push pending, {s'};
67          end if
68        end for
69      end while
70      for each x ∈ I–Marked do
71        if not reach[x] then
72          I–Marked ← I–Marked – {x};
73        else if x ∉ found then
74          remain[x] ← false;
75          I–Marked ← I–Marked – {x};
76        end if
77      end for
78    end if
79    for each x ∈ des.states do
80      if not remain[x] then
81        trimStateLow2(x, i, reach);
82        changed ← true;
83      end if
84    end for
85    return changed;
86  end
```

We now give a complexity analysis for algorithms 6.11-6.14. We will leave Algorithm 6.11 for last.

For low level k, $k = 1, 2, \ldots n$, let $n_{sk} = |\Sigma_{IL_k}|$, and n_k be the number of states of the largest DES (in terms of state size) among the m_{L_k} DES. We note that the state size of the synchronous product is $O(n_k^{m_{L_k}})$.

We first note that Algorithms 6.12 and 6.13 are almost identical to Algorithms 6.8 and 6.9, and thus take $O(n_{sk} \cdot n_k^{m_{L_k}})$ time when called for the k^{th} low level.

We now examine Algorithm 6.14 for processing low low level k. On line 7, we initialize the array *remain* which takes $O(n_k^{m_{L_k}})$ time. We use the array *remain* to mark whether a state should stay in the final result. The for loop on lines 9-16 also takes $O(n_k^{m_{L_k}})$ time.

The *while* loop in lines 23-32 is similar to the while loop on lines 22-31 in Algorithm 6.10, and is thus $O(n_{sk} n_k^{m_{L_k}})$.

We now consider the for loop in lines 34-54 that checks Point 5. In the worst case, R-Reachable can contain an entry for every transition in the synchronous product. The for loop thus runs $O(n_{sk} n_k^{m_{L_k}})$. However, the if statement at line 38 ensures that the core loop only gets executed $O(n_k^{m_{L_k}})$ times. If fact, it's as if we have a for loop that executes $O(n_{sk} n_k^{m_{L_k}})$ times performing a constant number of operations each iteration, followed by a for loop that executes $O(n_k^{m_{L_k}})$ times and executes lines 39-52 each iteration. Computations inside the second for loop are additive and dominated by the SEARCHANSWER function and the for loop on lines 46-50, both taking $O(n_{sk} \cdot n_k^{m_{L_k}})$ time. We thus have that the Point 5 check takes $O(n_{sk} n_k^{m_{L_k}}) + O(n_k^{m_{L_k}}) \cdot O(n_{sk} \cdot n_k^{m_{L_k}}) = O(n_{sk} \cdot n_k^{2m_{L_k}})$.

We next note that the initialization of the arrays on lines 59-60 each take $O(n_k^{m_{Lk}})$ time. We now examine the *while* loop in lines 61-69. As this loop is similar to the while loop in lines 76-84 in Algorithm 6.5, we know it executes $O(n_{sk} \cdot n_k^{m_{L_k}})$ times. It's followed by a for loop in lines 70-77 which runs $O(n_k^{m_{L_k}})$ times.

Finally, we note that the for loop in lines 83-88 is similar to the for loop in lines 32-37 of Algorithm 6.10 and thus runs in $O(n_{sk} \cdot n_k^{m_{L_k}})$ time.

As all the other steps take either constant or linear time, they are covered by the parts we analyzed. Put the analysis together for Algorithm 6.14, it takes

$O(n_k^{m_{L_k}}) + O(n_{sk} \cdot n_k^{m_{L_k}}) + O(n_{sk} \cdot n_k^{2m_{L_k}}) = O(n_{sk} \cdot n_k^{2m_{L_k}})$.

Now we investigate time complexity for Algorithm 6.11. The for loop (Line 3-85) runs n times, once for each low level. We will now analyze the rest for low level k.

Similar to Algorithm 6.4, line 5 takes $O(n_{sk})$ time , and lines 9-11 takes $O(m_{L_k} \cdot n_k \cdot n_{sk})$ time. We now examine the *while* loop in lines 18-60. The *while* loop goes over the state space of the synchronous product, which is worst case $n_k^{m_{L_k}}$. The for loop in lines 20-59 repeats n_{sk} times to check every event in low level k. This for loop also contains the for loop in lines 23-33 which goes over every DES, and thus runs m_{L_k} times. In parallel to the inner for loop, we also access the $found$ variable (lines 39 and 46) which is implemented as a trie data structure. This access is $O(m_{L_k})$. Also parallel to the inner for loop is a call to TRIMSTATELOW in line 36. Putting things together, we find that the outer for loop (20-59) is $O(n_{sk} \cdot m_{L_k} + n_{sk}^2 \cdot n_k^{m_{L_k}})$. However, as TRIMSTATELOW ensures that a state can only be trimmed at most once, we actually get $O(n_{sk} \cdot m_{L_k} + n_{sk} \cdot n_k^{m_{L_k}})$. Combining this with the while loop, we get $O(n_{sk} \cdot m_{L_k} \cdot n_k^{m_{L_k}} + n_{sk} \cdot n_k^{2m_{L_k}})$. Again, as TRIMSTATELOW ensures that a state can only be trimmed at most once, we get $O(n_{sk} \cdot m_{L_k} \cdot n_k^{m_{L_k}} + n_{sk} \cdot n_k^{m_{L_k}}) = O(n_{sk} \cdot m_{L_k} \cdot n_k^{m_{L_k}})$.

We next note that the array *reach* (line 61) can be initialized in $O(n_k^{m_{L_k}})$ time. The variable $found$ can now be implemented as an array, and thus be accessed (lines 69 and 71) in constant time. The *while* loop in lines 65-75 runs in $O(n_{sk} \cdot n_k^{m_{L_k}})$ time.

The *while* block in lines 76-79 calls the Algorithm 6.14. Each call of this algorithm takes $O(n_{sk} \cdot n_k^{2m_{L_k}})$ time. The worst case is that Algorithm 6.14 only trims one state each time and finally all states need to be trimmed off. The while block will thus run $O(n_k^{m_{L_k}})$ times, making the total run time $O(n_k^{m_{L_k}}) \cdot O(n_{sk} \cdot n_k^{2m_{L_k}}) = O(n_{sk} \cdot n_k^{3m_{L_k}})$. Similar to analysis for Algorithm 6.7, the number of calls to this function is effectively constant, thus this block would normally run in

$O(n_{sk} \cdot n_k^{2m_{Lk}})$ time. Finally, the *for* loop in lines 80-85 runs $O(n_k^{m_{L_k}})$ times.

Putting the parallel sections together, the run time for Algorithm 6.11 for one low level is $O(n_{sk}) + O(m_{L_k} \cdot n_k \cdot n_{sk}) + O(n_{sk} \cdot m_{L_k} \cdot n_k^{m_{L_k}}) + O(n_k^{m_{L_k}}) + (n_{sk} \cdot n_k^{m_{L_k}}) + O(n_{sk} \cdot n_k^{3m_{L_k}}) + O(n_k^{m_{L_k}}) = O(n_{sk} \cdot m_{L_k} \cdot n_k^{m_{L_k}} + n_{sk} \cdot n_k^{3m_{L_k}})$. As we have n low levels, we thus have $O(n(n_{sk} \cdot m_{L_k} \cdot n_k^{m_{L_k}} + n_{sk} \cdot n_k^{3m_{L_k}}))$. As typically $m_{L_k} \cdot n_k^{m_{L_k}} \ll n_k^{2m_{L_k}}$, we can simplify this to $O(n \cdot n_{sk} \cdot n_k^{3m_{L_k}})$. As discussed above, the $O(n_{sk} \cdot n_k^{3m_{L_k}})$ term typically turns out to be $O(n_{sk} \cdot n_k^{2m_{L_k}})$. We would thus expect the algorithm to behave as $O(n \cdot n_{sk} \cdot n_k^{2m_{L_k}})$.

6.5.3 Summary

It is shown in [27] that the synthesis for a system constructed from multiple plant and specification DES is a NP-hard problem. This means there is unlikely any algorithm that solves this problem in a polynomial time. The complexity is exponential to the number of component DES involved in the synthesis.

As we can see from our analysis, we would expect our algorithms take $O(n_{sH} \cdot m_H \cdot n_H^{m_H})$ and $O(n \cdot n_{sk} \cdot n_L^{2m_{Lk}})$ to generate our high and low level interface consistent supervisors.

Let N_H denote the size of the state space of $\mathbf{G}_H^p || \mathbf{E}_H$, while N_I and N_L are upper bounds for the state space size of $\mathbf{G}_{I_j}$ and $\mathbf{G}_{L_j}^p || \mathbf{E}_{L_j}$ $(j = 1, \ldots, n)$, respectively. As was discussed in [37], the limiting factor for a flat system would typically be $N_H N_L^n$, and $N_H N_I^n$ for the HISC method as it grows in the number of low levels. We would expect our method to offer significant improvement as long as $N_I \ll N_L$.

Of course, this increased scalability comes with a price: a more restrictive architecture and thus the possible loss of global maximal permissiveness. In other words, if we instead modeled the system as a flat system and did a normal synthesis operation such as using the TCT *supcon* algorithm [79], we might be able to get a larger closed loop behavior than we got with the HISC synthesis. However, we feel

that the tradeoff is worthwhile due to the increased scalability and the behavior encapsulation provided by the HISC method.

Chapter 7

AIP Example

To demonstrate the utility of our method, we apply it to a large manufacturing system, the Atelier Inter-établissement de Productique (AIP) as described in [9, 14], and later investigated by Leduc et al. [34, 35, 36, 37, 33] using the HISC method and then by Ma et al. [47, 48] using state tree structure and binary decision diagrams.

In this chapter we first introduce the system structure of the AIP. Then we describe our modifications to the AIP example which are based on Leduc et al. [34, 35, 36, 37, 33]. Finally, we apply our synthesis method to the example, and discuss the result.

In this chapter, a few figures are borrowed from [34, 35] with the author's permission. We will indicate in the figure's caption when this happens.

7.1 Introduction

The AIP is an automated manufacturing system consisting of a central conveyor loop (CL) and four external conveyor loops (EL). There are three assembly stations (AS) that process incoming pallets and four transfer units (TU) that transfer

pallets between central and external loops. An I/O station puts raw pallets into external loop 4 and takes away processed pallets from it.

The AIP system structure is shown in Figure 7.1. Figure 7.2 shows the hierarchical structure of AIP system.

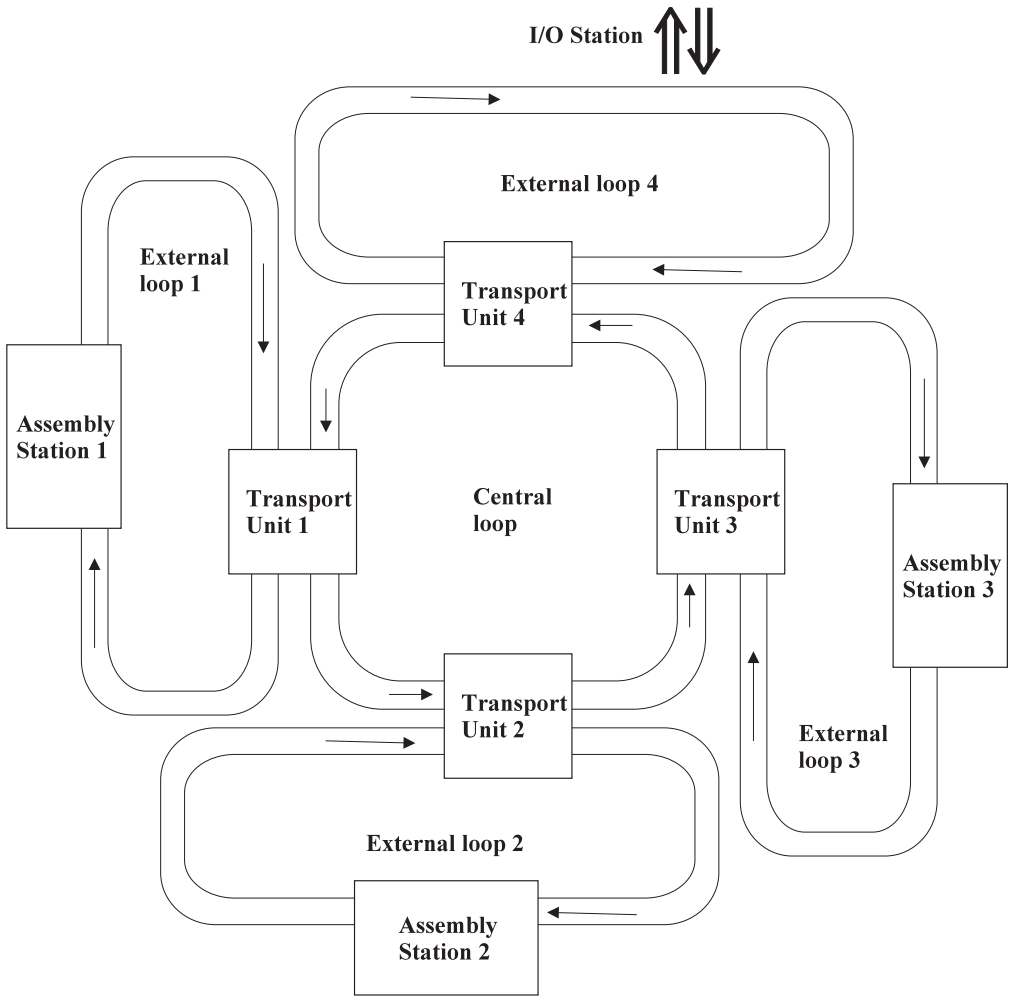

Figure 7.1: AIP System Structure ([34])

A transfer unit checks whether a pallet needs to be transferred or not and if it does, then performs the transfer between the central loop and the transfer unit's external loop.

Assembly stations monitor pallets on their corresponding external conveyor loops. If the pallets need to be processed, the assembly station processes it and

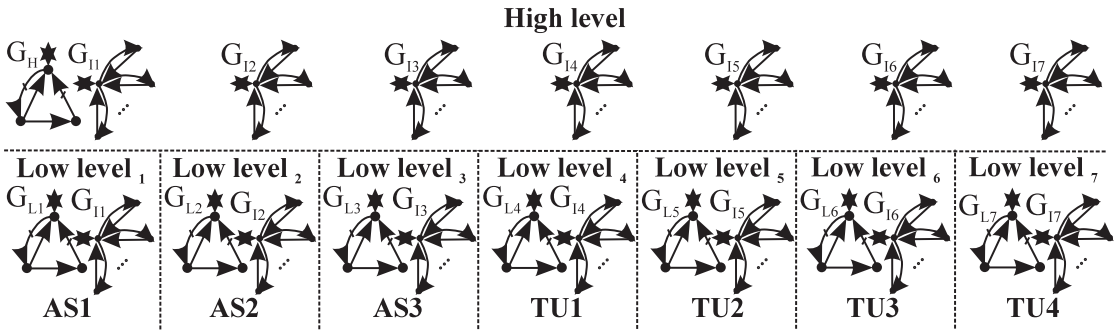

Figure 7.2: Hierarchical Structure of AIP([34])

then puts it back on its external loop. The pallet then waits to be transferred to the central loop. The three assembly stations can perform different assembly actions. Station 1 (AS1) can do task1A and task1B. Station 2 (AS2) can do task2A and task2B. Station 3 (AS3) acts like a master station which can perform all four tasks and works as a backup machine when AS1 or AS2 break down.

Figure 7.3 and 7.4 show the layout of an assembly station and a transfer unit, respectively.

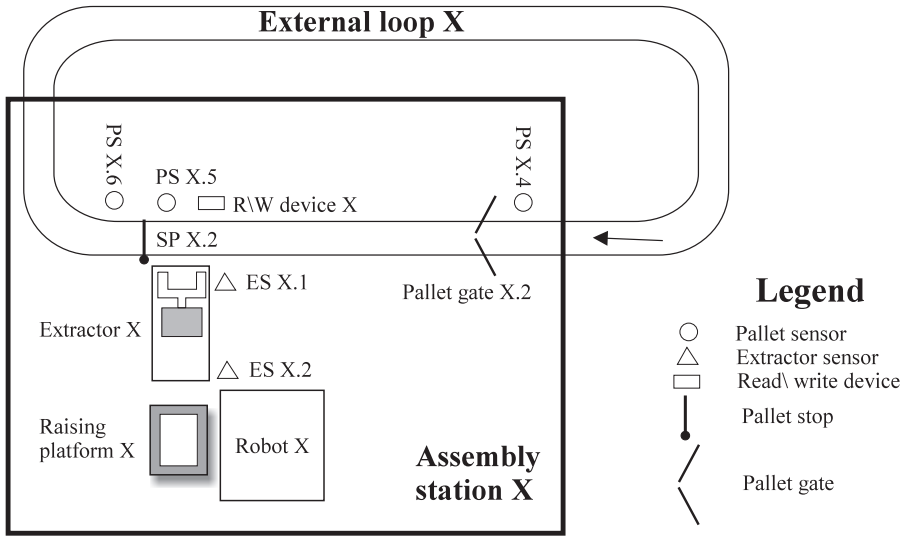

Figure 7.3: Assembly Station Layout ([34])

The AIP system can process two types of pallets: type 1 pallets and type 2 pallets. We assume that the system is initially empty and two types of pallets are

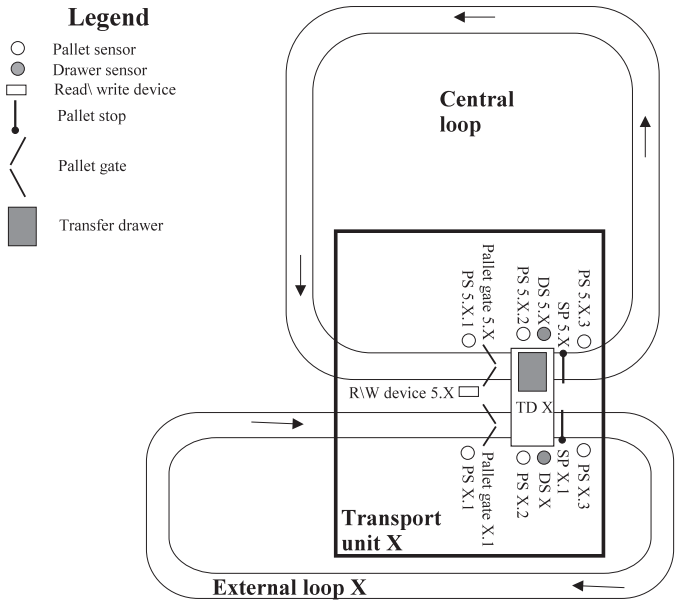

Figure 7.4: Transfer Unit Layout ([34])

fed into the system alternately in the order type 1, type 2, type 1, ...

The system needs to achieve the follow control specifications:

- Type 1 pallets are processed in the order task1A, task1B, task2A, task2B Type 2 pallet is processed in the order task2B, task2A, task1B, task1A. Pallets may only leave the system after all four steps have been completed.
- Pallets exiting the system must alternate in type, starting with type 1.
- Only one pallet is permitted in external loop 1 or 2 at a time.
- Assembly stations can only process one pallet at a time.
- When AS1 or AS2 are down, pallets will be routed to AS3. When the station is repaired, the system will return to normal.
- When an assembly error occurs, the unfinished pallet will be routed to AS3 for maintenance and then sent back to the original station to undergo the assembly operation again.

We first applied our algorithm on the original AIP example as appeared in [35, 36, 37, 33]. The result was supervisor with 3,306,240 states, constructed in 3 minutes 24 seconds, using 885MB of memory. As expected, no states were trimmed off during the synthesis process since the given set of specifications are themselves supervisors that were designed to meet the HISC requirements.

In next section we will modify the AIP example by slightly changing the control specifications so that we will need to do synthesis.

7.2 Modifying the AIP

In this chapter, we use the AIP model in Leduc et al. [34, 35, 36, 37, 33] as our starting point. In addition to the control specifications given in Section 7.1, we have added two new design requirements:

- Restrict capacity of external loop 3 to three pallets.
- For AS1 and AS2, if three consecutive errors happen, the assembly station is suspected to be broken and a repair procedure will be invoked.

We use italic font for uncontrollable events and regular font for controllable events. We use the event partition listed in [34] with one small change. As part of our redesign, we removed DES **DetWhichStnUp** and thus had to remove event *DetStnsUp* from Σ_H. In the DES in our diagrams, the initial state has a thick border and marker states are represented by filled gray circles.

7.2.1 High Level

The high level contains 6 plant DES and 7 specification DES, as shown in Figure 7.5. The synchronous product of these 6 plant DES is the high level plant $\mathbf{G}_H^p$, and

the synchronous product of all 7 specification DES is the high level specification DES $\mathbf{E}_H$.

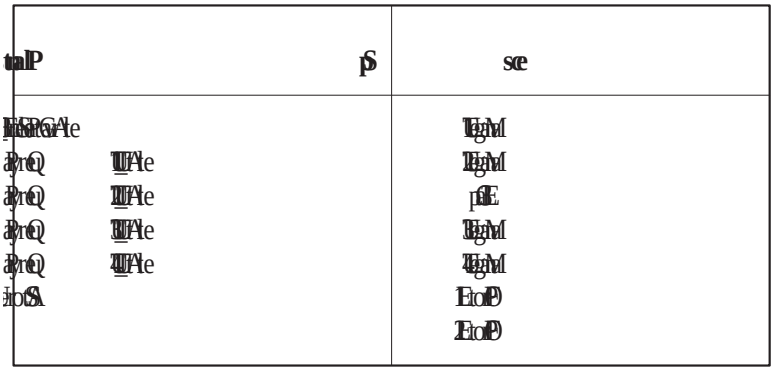

Figure 7.5: High Level DES List

We first discuss the plant DES. **PalletArvGateSenEL_2_AS3**, a shown in Figure 7.6, states that a pallet can not be processed by AS3 until a pallet has arrived at its gate. **QueryPalletAtTU.i** in Figure 7.7, where i in {'TU1', 'TU2', 'TU3', 'TU4'}, are a set of DES that check whether a pallet is ready to be transferred between the center loop and a specific external loop.

Figure 7.6: PalletArvGate-SenEL_2_AS3 ([34])

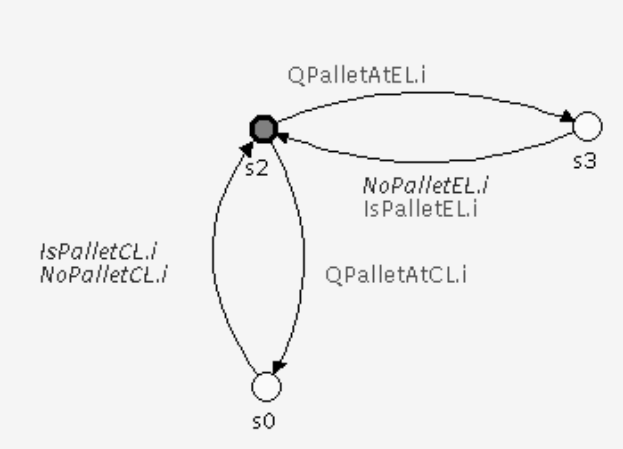

Figure 7.7: QueryPalletAtTU.i ([34])

DES **ASStoreUpState**, shown in Figure 7.8, stores the breakdown status of AS1 and AS2. It encodes the status in events which are used by TU3 to determine

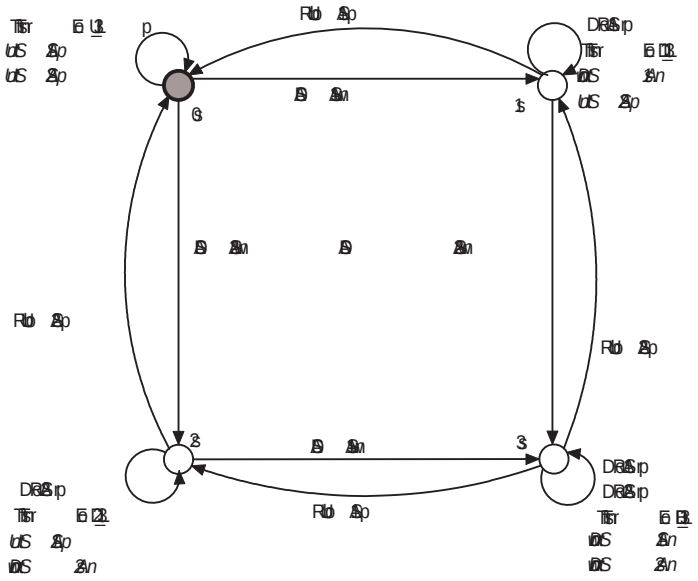

Figure 7.8: ASStoreUpState

if a pallet should be processed by AS3. For example, the event *TrnsfToEL3_1D* contains the information that AS1 is down and AS2 is currently up.

We now discuss the specification DES for the high level. DES **ManageTU1** and **ManageTU2**, shown in Figure 7.9 and Figure 7.10, differ from the original ones in [34] by removing the QStnUp.i events ('i' stands for either AS1 or AS2) as these events are no longer needed. These supervisors control the transfer of pallets between the center loop and the indicated external loop.

We update DES **ManageTU3**, shown in Figure 7.11, by removing the *DetStnsUp* event which was previously used to signal DES **DetWhichStnUp**. In the AIP example from [34, 35, 36, 37, 33], **ManageTU3** used supervisor **DetWhichStnUp** to determine whether AS1 and AS2 are up, and then encode this information as an event. This task is now performed by plant component **ASStoreUpState**, so DES **DetWhichStnUp** is no longer required. The DES **HndlComEventsAS** was also removed from the system (it was present in [34, 35, 36, 37, 33]) as it is no longer needed as its task was to arbitrate between **DetWhichStnUp** and **ManageTU1** and **ManageTU2** with respect to access

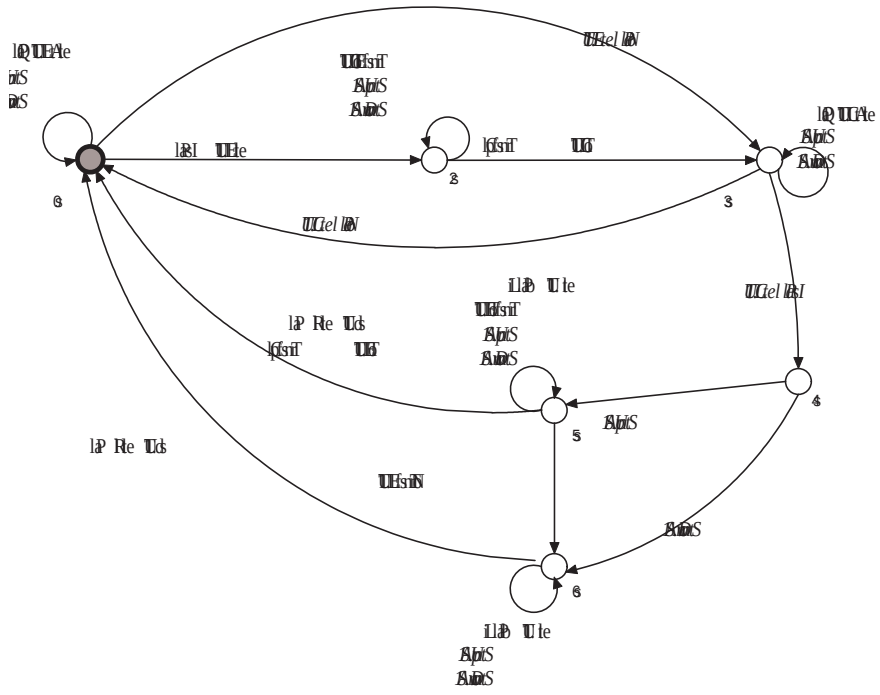

Figure 7.9: ManageTU1

to common controllable events.

For the new requirement that only three pallets are permitted at a time in external loop 3, we added DES **EL3Cap**, shown in Figure 7.12. The remaining supervisors DES **ManageTU4**, **OFProtEL1** and **OFProtEL2**, are unchanged. Readers are referred to [34] for more details.

7.2.2 Low Levels

We implemented the second new specification by modifying interfaces for low levels AS1 and AS2, shown in Figure 7.13. In the diagram, 'i' can take the value 'AS1' or 'AS2'. After three consecutive errors, the new interface forces a repair operation.

To accommodate the new interface, we had to modify plant components **Robot.AS1** and **Robot.AS2**, shown in Figures 7.16 and 7.17. In the original model, a robot repair could only be initiated after a timeout during processing oc-

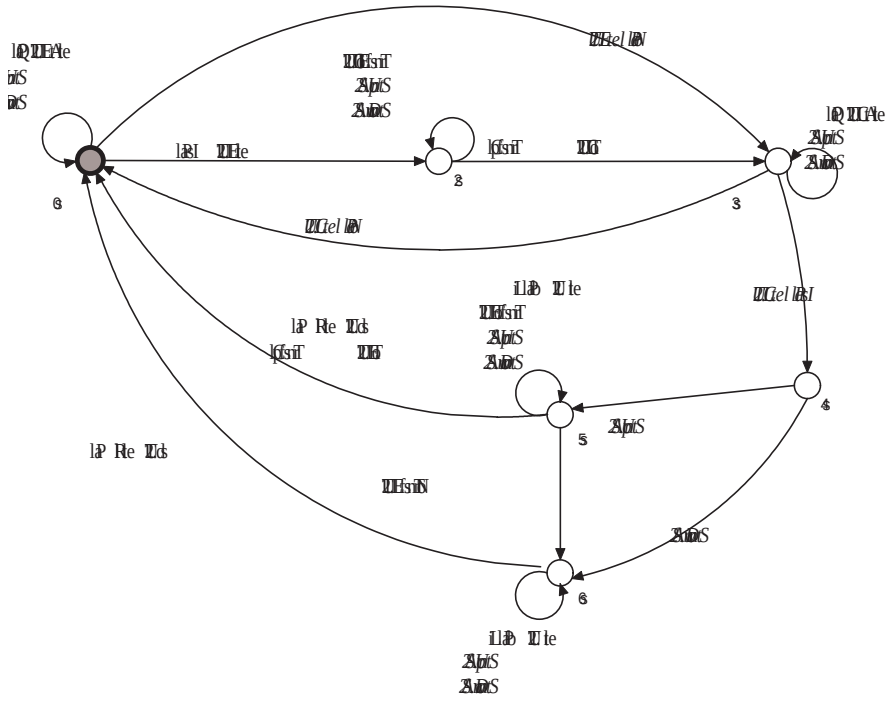

Figure 7.10: ManageTU2

curred. We had to add the functionality that a repair could occur while the robot was in its initial state. We then had to make corresponding changes to supervisors **DoRobotTasks.AS1**, **DoRobotTasks.AS2**, shown in Figures 7.14, 7.15. For the remainder DES at the low levels, readers are referred to [34, 35] .

7.2.3 Results

We now apply our software to our version of the AIP example, and determine that the system is HISC-valid. We next apply our iSupconHigh algorithm and get a high level interface consistent supervisor.We then apply our iSupconLow algorithm and get seven low level interface consistent supervisors, one for each low level. Since our algorithms build interface consistent, level-wise nonblocking and level-wise controllable supervisors, we can apply Theorem 6 and Theorem 7 and conclude that the flat system is nonblocking and the flat supervisor is controllable for the flat plant. Running on a Redhat Linux 9 computer with a 2.4 GHz Xeon

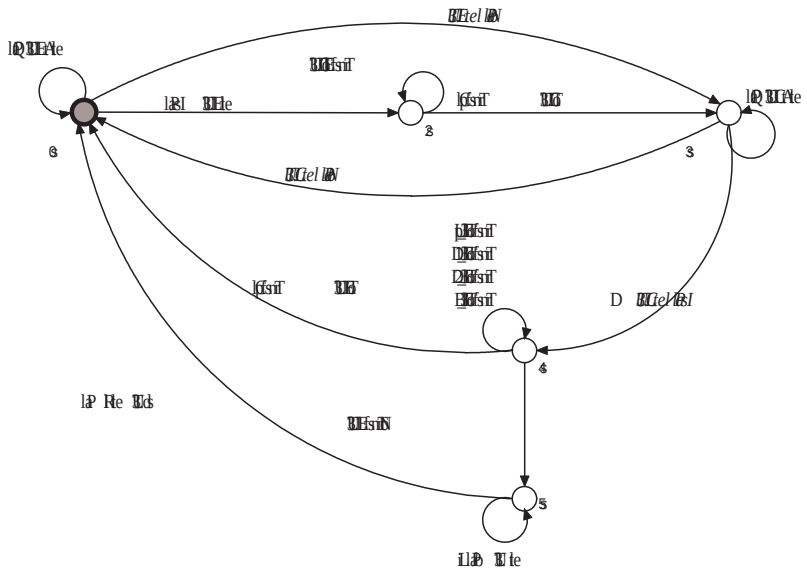

Figure 7.11: ManageTU3

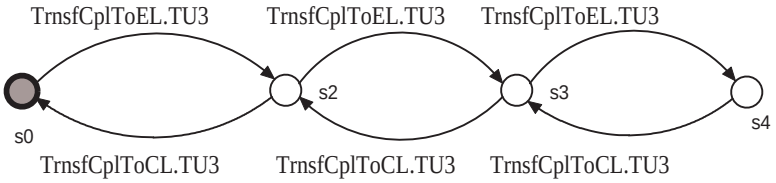

Figure 7.12: EL3Cap

CPU and 4G memory, the program finishes in 6 minutes and 2 seconds and uses 2GB memory. Detailed results are shown in Table 7.1. It shows the size of the various subsystem automata used in the AIP calculations. First, the size of the state space of each component without being synchronized with their respective interfaces (Standalone) is given and then state space size when synchronized with their interface DES ($\mathbf{G}_H$ is synchronized with all seven interfaces). Next, the size of the interfaces for the high level and each low level. We then give the number of states trimmed during synthesis, followed by the run time for each component.

In Section 6.5.3, we discussed that the limiting factor for a monolithic algorithm would be $N_H N_L^n$ and similarly $N_H N_I^n$ for the HISC method. In the equations, N_H denotes the size of the state space of $\mathbf{G}_H$, while N_I and N_L are the bounds for $\mathbf{G}_{I_j}$

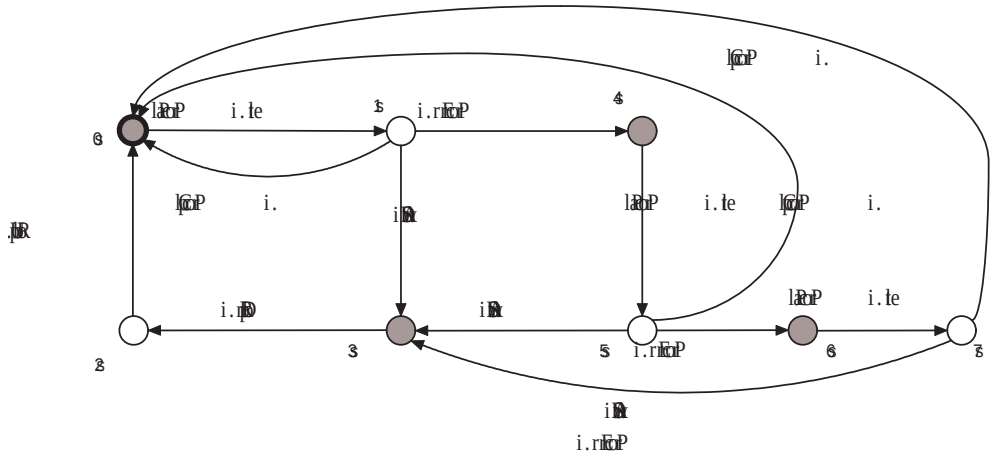

Figure 7.13: Interface for AS1 and AS2

and $\mathbf{G}_{L_j}$ $(j = 1, \ldots, n)$, respectively. If we substitute actual data from Table 7.1, $N_L^n = (353)^2(203)(98)^2(204)(152) = 7.53 \times 10^{15}$ and $N_I^n = (9)^2(2)(4)^4 = 41,471$. This is a potential savings of 11 orders of magnitude!

Table 7.1: AIP Results

	States			States	Computing
Subsystem	**Standalone**	**|| with $\mathbf{G}_{I_j}$**	**Size of $\mathbf{G}_{I_j}$**	Trimmed	Time
$\mathbf{G}_H$	793,800	6,634,800	41,472	349,200	6 min 1 sec
AS1	1,732	353	9	116	$<$ 1 sec
AS2	1,732	353	9	116	$<$ 1 sec
AS3	1,178	203	2	0	$<$ 1 sec
TU1	98	98	4	0	$<$ 1 sec
TU2	98	98	4	0	$<$ 1 sec
TU3	204	204	4	0	$<$ 1 sec
TU4	152	152	4	0	$<$ 1 sec

The modified AIP example has a worst case state space of 2.25×10^{22}. This was estimated by multiplying the size of the state space of the high level and all of the low levels together. It is quite likely that the actual system is much smaller. We also tried to construct a synchronous product of our entire system to see how long that would take and what the state space size would be, but the program

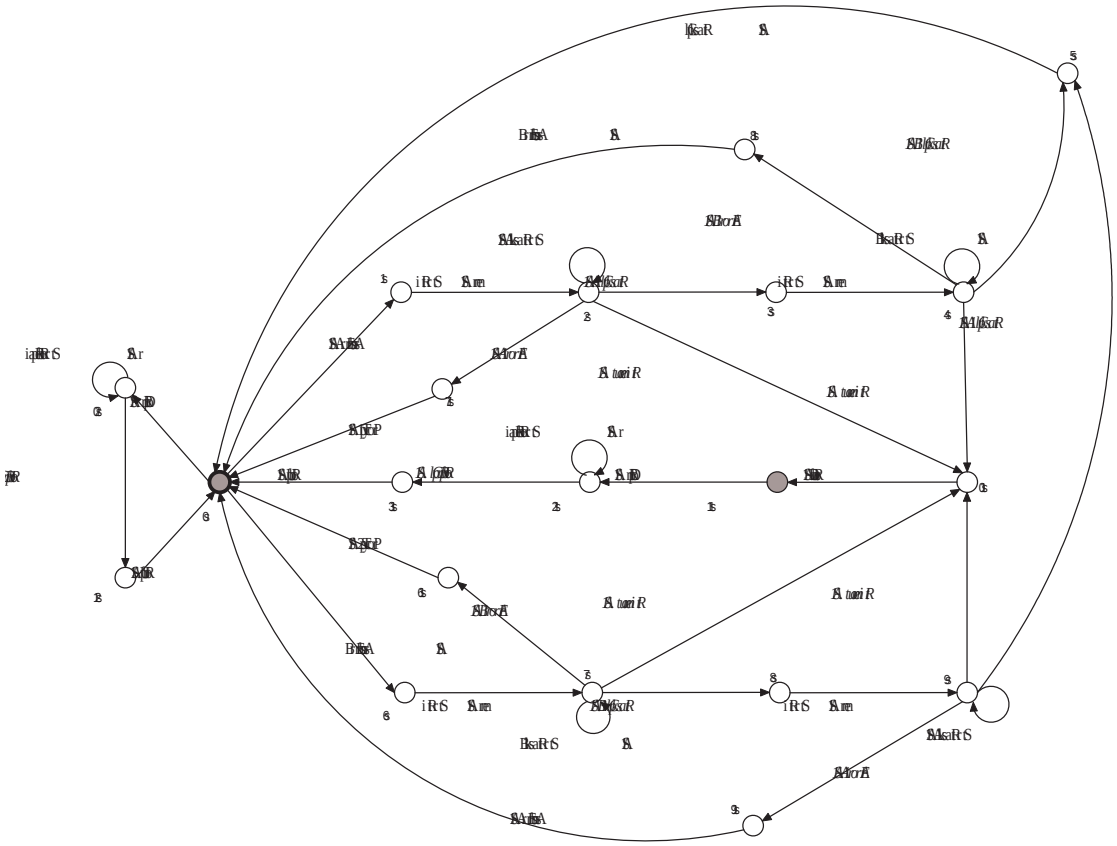

Figure 7.14: DoRobotTasks.AS1

crashed after using up all available memory.

It is interesting to note that typically we can not perform a flat synthesis, such as the *supcon* algorithm from TCT [79], to an HISC system and expect to receive a meaningful result. Let's consider the high level for the AIP. If we examine the specification in **EL3Cap** Figure 7.12, we see that it forbids answer event *TrnsfCpltToCL.TU3* from occurring at state *s0*. The high level synthesis is not permitted to disable an answer event, so it will correctly disable the request event *TrnsfELtoCL.TU3* that leads to a state that the answer event is defined. This is because it knows that the answer event is an abstraction of a corresponding low level task, and it is the low level task that we actually wish to prevent. However if we did a flat synthesis, *supcon* would just disable *TrnsfCpltToCL.TU3* as it's a controllable event, but allow request event *TrnsfELtoCL.TU3* to occur. This would

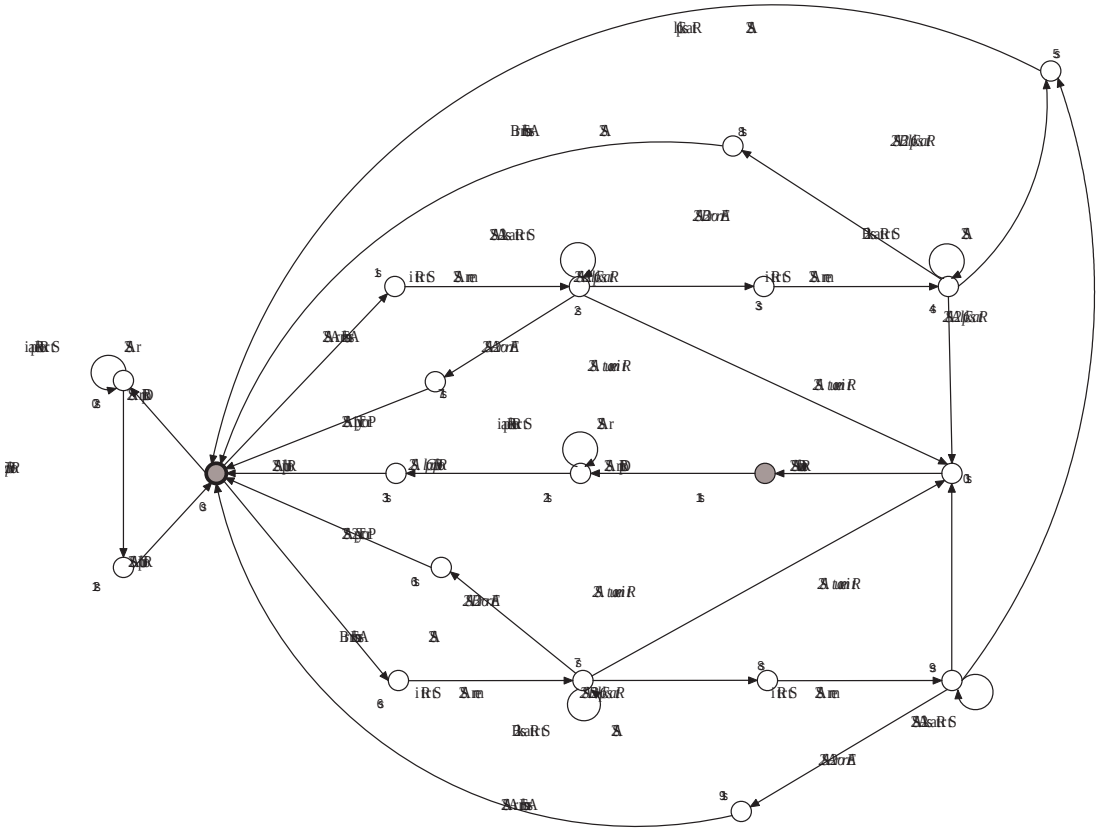

Figure 7.15: DoRobotTasks.AS2

have the effect of allowing the low level task to occur, but only disabling the low level from reporting that the task has completed! Definitely not the desired result. The reason that a normal flat synthesis would likely not produce a meaningful result is because it does not understand the hierarchy of an HISC system, or the abstractions that it employs.

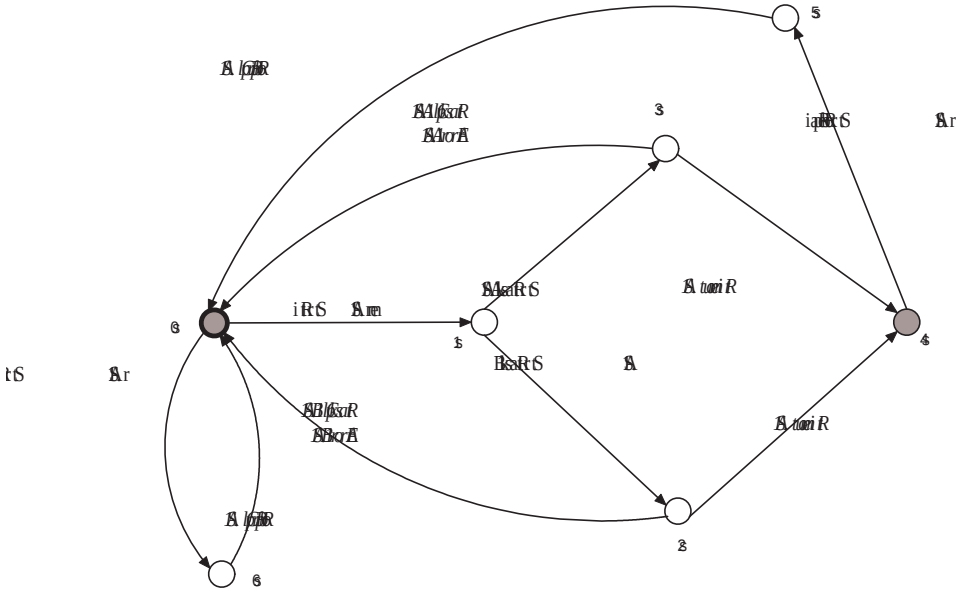

Figure 7.16: Robot.AS1

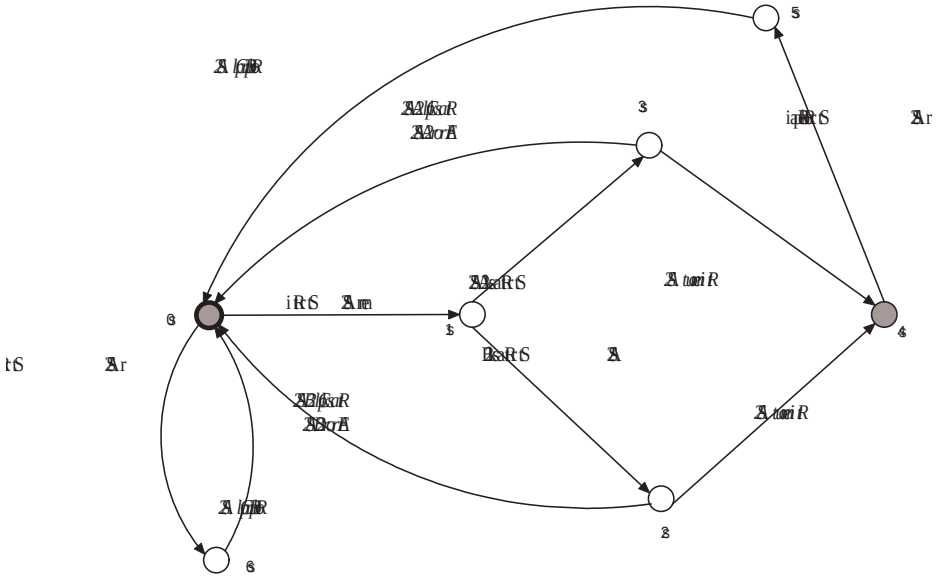

Figure 7.17: Robot.AS2

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis we developed a synthesis method for the Hierarchical Interface-based Supervisory Control (HISC) system that does a per level synthesis to construct for each level a maximally permissive supervisor that satisfies the corresponding HISC conditions.

We then defined a set of language based fixpoint operators and showed that they compute the required level-wise supremal languages. We then presented algorithms that implement the fixpoint operators. Next, we performed a complexity analysis for the algorithms. As the synthesis is done on a per level basis, the complete system model never needs to be stored in memory, offering potentially significant savings in computational resources. In fact, as long as the state size of the interfaces are much smaller than the state size of the corresponding low levels, we should see significant reduction in complexity.

Finally we developed software to implement the algorithms. We modified the AIP example from [34, 35, 36, 37, 33], and applied our software to it. We demonstrated that we were able to handle this complex example, where a monolithic

approach failed.

8.2 Future Work

The HISC framework is still relatively new, and there are many areas of interest to be explored. Some of these are:

- The current HISC model accommodates only two levels: one high level and one or more low level subsystems. In general, as we add more low levels to the system, the high level becomes increasingly more complex. Similarly, a given low could be potentially very large. It would be useful to extend the HISC method to allow for a multi-level hierarchy.
- Currently, only the high level can communicate directly with the low levels. It would be useful if the low levels were able to communicate directly with each other, and request other low levels to perform tasks for them. This would be useful for modeling a system composed of independent autonomous agents, for example.
- Although our synthesis method simplified the work for designers, there are not many choices for design tools. Currently, our software takes text files as input and can output in either text or TCT [79] data file format. A graphical interface for entering DES and the system hierarchy would be very helpful for designers.
- In the current HISC structure, communication between levels occurs via request and answer events. When the high level sends a request event to a particular low level, it can not send additional information to the low level until it receives an answer event back from it. It would be useful to able to send information in between the request and answer events. A possible application would be an abort signal if the task was no longer needed.

Bibliography

[1] Sharif Abdelwahed, *Interacting discrete event systems: Modeling, verification and supervisory control*, Ph.D. thesis, Dept. of Elec. and Comp. Eng., University of Toronto, 2002.

[2] Knut Åkesson, Hugo Flordal, and Martin Fabian, *Exploiting modularity for synthesis and verification of supervisors*, Proc. of the IFAC World Congress on Automatic Control (Barcelona, Spain), 2002.

[3] N. Alsop, *Formal techniques for the procedural control of industrial processes*, Ph.D. thesis, Department of Chemical Engineering and Chemical Technology, Imperial College of Science, Technology and Medicine, London, 1996.

[4] P. Apkarian and D. Noll, *Decentralized supervision of petri nets*, IEEE Trans. Automatic Control **51** (2006), no. 2, 376– 381.

[5] Adnan Aziz, Vigyan Singhal, and Gitanjali M. Swamy, *Minimizing interacting finite state machines: A compositional approach to language containment*, Proc. of IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors (Cambridge, Massachusetts), Oct 1994, pp. 255–261.

[6] G. Barrett and S. Lafortune, *Decentralized supervisory control with communicating controllers*, IEEE Transactions on Automatic Control **45** (2000), no. 9, 1620–1638.

[7] George Barrett and Stephane Lafortune, *Decentralized supervisory control with communicating controllers*, IEEE Trans. Automatic Control **45** (2000), no. 9, 1620–1638.

[8] Sergey Berezin, Sérgio Campos, and Edmund M. Clarke, *Compositional reasoning in model checking*, COMPOS'97, LNCS, vol. 1536, Springer-Verlag, 1998, pp. 81–102.

[9] Bertil Brandin and François Charbonnier, *The supervisory control of the automated manufacturing system of the AIP*, Proc. Rensselaer's 1994 Fourth International Conference on Computer Integrated Manufacturing and Automation Technology (Troy), Oct 1994, pp. 319–324.

[10] Y. Brave and M. Heymann, *Control of discrete event systems modeled as hierarchical state machines*, IEEE Trans. on Automatic Control **38** (1993), no. 12, 1803–1819.

[11] R. E. Bryant, *Graph-based algorithms for boolean function manipulation*, IEEE Trans. Comput. **C-35** (1986), no. 8, 677–691.

[12] J.R. Burch, Edmund M. Clarke, and K.L. McMillan, *Symbolic model checking:* 10^{20} *states and beyond*, Information and Computation **98** (1992), 142–170.

[13] P.E. Caines and Y.J. Wei, *The hierarchical lattices of a finite machine*, Systems Control Letters **25** (1995), 257–263.

[14] F. Charbonnier, *Commande par supervision des systèmes à événements discrets: application à un site expérimental l'Atelier Inter-établissement de Productique*, Tech. report, Laboratoire d'Automatique de Grenoble, Grenoble, France, 1994.

[15] H. Chen and H.M. Hanisch, *Model aggregation for hierachichal control synthesis of discrete event systems*, Proc. 39th Conf. of Desision Control (Sydney, Australia), Dec. 2000, pp. 418–423.

[16] Haoxun Chen and Hans-Michael Hanisch, *Model aggregation for hierarchical control synthesis of discrete event systems*, Proc. 39th Conf. Decision Contr. (Sydney, Australia), December 2000, pp. 418–423.

[17] S.-L. Chen, *Control of discrete-event systems of vector and mixed structural type*, Ph.D. thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, ONT, 1996.

[18] Yi-Liang Chen and Feng Lin, *Hierarchical modeling and abstraction of discrete event systems using finite state machines with parameters*, Proc. 40th Conf. Decision Contr. (Orlando, USA), December 2001, pp. 4110–4115.

[19] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Cliff Stein, *Introduction to algorithms*, MIT Press and McGraw-Hill, 2001.

[20] M. Courvoisier, M.Combacau, and A. de Bonneval, *Control and monitoring of large discrete event systems: a generic approach*, Proc. of ISIE 93 (Budapest), 1993, pp. 571–576.

[21] Luca de Alfaro and Thomas A. Henzinger, *Interface automata*, Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering, ACM Press, 2001, pp. 109–120.

[22] E. W. Endsley, M. R. Lucas, and D. M. Tilbury, *Modular design and verification of logic control for reconfigurable machining systems*, [ONLINE]. Available: http://www-personal.engin.umich.edu/~tilbury/papers.html, Oct. 2000.

[23] E. W. Endsley and D. M. Tilbury, *Modular verification of modular finite state machines*, Proc. 43th Conf. Decision Contr. (Atlantis, Paradise Island, Bahamas), vol. 1, December 2004, pp. 972–979.

[24] Jose M. Eyzell and Jose E.R. Cury, *Exploiting symmetry in the synthesis of supervisors for discrete event systems*, Proc. of American Control Conference (Philadelphia, USA), June 1998, pp. 244–248.

[25] M. Fabian, *On object-oriented non-deterministic supervisory control*, Ph.D. thesis, Chalmers Univ. of Tech., Goteborg, Sweden, 1995.

[26] M. Fabian and B. Lennartson, *Petri nets and control synthesis; an object oriented approach*, Proc. of I.M.S. (Vienna, Austria), June 1994, pp. 365–370.

[27] P. Gohari-M. and W.M. Wonham, *On the complexity of supervisory control design in the RW framework.*, IEEE Trans. on Systems, Man and Cybernetics; Part B: Cybernetics. (Special Issue on Discrete Systems and Control) (2000), 643–652.

[28] Peyman Gohari-Moghadam, *A linguistic framework for controlled hierarchical DES*, Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ont, 1998.

[29] Johan Gunnarsson, *Symbolic methods and tools for discrete event dynamic systems*, Ph.D. thesis, Department of Electrical Engineering at Linköping University, Sweden, 1997.

[30] Qian Pu Ken, *Modeling and control of discrete-event systems with hierarchical abstraction*, Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, ONT, 2000.

[31] P. Kozak and W.M. Wonham, *Fully decentralized solutions of supervisory control problems*, Automatic Control, IEEE Transactions on **40** (1995), 2094 – 2097.

[32] Robert Kruse, Bruce Leung, and Clovis Tondo, *Data structures and program design in c*, Prentice Hall, Inc., 1997.

[33] R. Leduc, M. Lawford, and P. Dai, *Hierarchical interface-based supervisory control of a flexible manufacturing system*, Accepted to IEEE Trans. on Control Systems Technology, Dec. 2005.

[34] R. J. Leduc, *Hierarchical interface-based supervisory control*, Ph.D. thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ont., 2002, [ONLINE] Available: http://www.cas.mcmaster.ca/~leduc.

[35] ———, *Hierarchical interface-based supervisory control: Command-pair interfaces (see extended version)*, Proc. of the Third International DCDIS Conference on Engineering Applications and Computational Algorithms (Guelph, Ontario, Canada), May 15-18 2003, [ONLINE] Available: http://www.cas.mcmaster.ca/~leduc, pp. 323–329.

[36] R. J. Leduc, B. A. Brandin, M. Lawford, and W. M. Wonham, *Hierarchical interface-based supervisory control, part I: Serial case*, IEEE Trans. Automatic Control **50** (2005), no. 9, 1322–1335.

[37] R. J. Leduc, M. Lawford, and W. M. Wonham, *Hierarchical interface-based supervisory control, part II: Parallel case*, IEEE Trans. Automatic Control **50** (2005), no. 9, 1336–1348.

[38] R.J. Leduc, B.A. Brandin, and W. Murray Wonham, *Hierarchical interface-based non-blocking verification*, Proceedings of the Canadian Conference on Electrical and Computer Engineering, May 2000, pp. 1–6.

[39] R.J. Leduc, B.A. Brandin, W. Murray Wonham, and M. Lawford, *Hierarchical interface-based supervisory control: Serial case*, Proc. of 40th Conf. Decision Contr. (Orlando, USA), December 2001, pp. 4116–4121.

[40] R.J. Leduc, M. Lawford, and P. Dai, *Hierarchical interface-based supervisory control of a flexible manufacturing system*, Tech. Report No. 32, Soft-

ware Quality Research Laboratory, Dept. of Computing and Software, McMaster University, Hamilton, ON, Canada, Dec. 2005, [ONLINE] Available: http://www.cas.mcmaster.ca/sqrl/sqrl_reports.html.

[41] R.J. Leduc, W. Murray Wonham, and M. Lawford, *Hierarchical interface-based supervisory control: Parallel case*, Proc. of 39th Annual Allerton Conference on Comm., Contr., and Comp., Oct 2001, pp. 386–395.

[42] R.J Leduc, W. Murray Wonham, and M. Lawford, *Hierarchical interface based supervisory control: Bi-level systems*, Tech. Report No. 0103, Systems Control Group, University of Toronto, Toronto, ON, Canada, Nov 2001.

[43] Ryan 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.

[44] Y. Li, *Control of vector discrete-event systems*, Ph.D. thesis, Department of Electrical Engineering, University of Toronto, Toronto, ONT, 1991.

[45] F. Lin and W.M. Wonham, *Decentralized control and coordination of discrete-event systems with partial observations*, Proc. 27th IEEE Conf. Decision Contr., Dec 1988, pp. 1125–1130.

[46] ———, *Decentralized control and coordination of discrete-event systems with partial observations*, IEEE Transactions on Automatic Control **35(122)** (1990), 1330–1337.

[47] C. Ma and W. Murray Wonham, *Control of state tree structures*, Proc. 11th Mediterranean Conference on Control and Automation, June 2003, Paper T4-005 (6pp.).

[48] Chuan Ma, *Nonblocking supervisory control of state tree structures*, Ph.D. thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, ONT, 2004.

[49] K.L. McMillan, *Symbolic model checking*, Kluwer, 1992.

[50] John O. Moody and Panos J. Antsaklis, *Supervisory control of discrete event systems using Petri nets*, Kluwer Academic Publishers, 1998.

[51] T. Moor, J. Raisch, and J.M. Davoren, *Admissibility criteria for a hierarchical design of hybrid control systems*, Proc. IFAC Conference on Analysis and Design of Hybrid Systems (Saint-Malo, France), June 2003, pp. 389–394.

[52] D. L. Parnas, P. C. Clements, and D. M. Weiss, *The modular structure of complex systems*, ACM SIGSOFT Software Engineering Notes , Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, March 1984.

[53] D.L. Parnas, *On the criteria to be used in decomposing system into modules.*, Communications of the ACM **15** (1972), no. 12, 1053–1058.

[54] ———, *A technique for software module specification with examples.*, Communications of the ACM **15** (1972), no. 5, 330 – 336.

[55] 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.

[56] Robin G. Qiu and Sanjay B. Joshi, *A structured adaptive supervisory control methodology for modeling the control of a discrete event manufacturing system*, IEEE Trans. Systems, Man, and Cybernetics, Part A **29** (1999), no. 6, 573–586.

[57] M. Queiroz and J. Cury, *Modular supervisory control of large scale discrete event systems*, Proceedings of WODES 2000 (Ghent, Belgium), Aug. 2000, pp. 103–110.

[58] M.H. de Queiroz and J.E.R. Cury, *Modular supervisory control of large scale discrete event systems*, Proceedings of WODES 2000 (Ghent, Belgium), Aug 2000, pp. 103–110.

[59] P. Ramadge and W. Wonham, *Supervisory control of a class of discrete-event processes*, SIAM J. Control Optim **25** (1987), no. 1, 206–230.

[60] P.J. Ramadge, *Control and supervision of discrete event processes*, Ph.D. thesis, Department of Electrical Engineering, University of Toronto, 1983.

[61] Karen Rudie, *Software for the control of discrete event systems: A complexity study*, Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, ONT, 1988.

[62] Karen Rudie and Jan C. Willems, *The computational complexity of decentralized discrete-event control problems*, IEEE Trans. Automatic Control **440** (1995), no. 7, 1313–1319.

[63] Karen Rudie and W.M. Wonham, *Think globally, act locally: Decentralized control*, IEEE Transactions on Automatic Control (1992), 1692–1708.

[64] M.A. Shayman and R. Kumar, *Process objects/masked composition: an object-oriented approach for modeling and control of discrete-event systems*, IEEE Trans. Automatic Control **44** (1999), no. 10, 1864–1869.

[65] Gang Shen and Peter E. Caines, *Hierarchically accelerated dynamic programming for finite-state machines*, IEEE Trans. Automatic Control **47** (2002), no. 2, 271–283.

[66] Raoguang Song, *Symbolic synthesis and verification of hierarchical interface-based supervisory control*, Master's thesis, Department of Computing and Software, McMaster University, Hamilton, ONT, March 2006.

[67] G. Stremersch and R.K. Boel, *Decomposition of the supervisory control problem for Petri nets under preservation of maximal permissiveness*, IEEE Trans. Automatic Control **46** (2001), no. 9, 1490–1496.

[68] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen, *The structure and value of modularity in software design*, Proceedings of the 7th international conference on Software engineering, vol. 26, September 2001.

[69] C. Torrico and J. Cury, *Hierarchical supervisory control of discrete-event systems based on state aggregation*, Proceedings of Fifteenth Triennial World Congress of the International Federation of Automatic Control (Barcelona, Spain), Jul. 2002.

[70] 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** (2002), 192–208.

[71] Arash Vahidi, Bengt Lennartson, and Martin Fabian, *Efficient analysis of large discrete-event systems with binary decision diagrams*, Proc. of the 44th IEEE Conference on Decision and Control and European Control Conference 2005 (Seville, Spain), 2005, pp. 2751–2756.

[72] 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.

[73] Y. Willner and M. Heymann, *Supervisory control of concurrent discrete-event systems*, International Journal of Control **54** (1991), no. 5, 1143–1169.

[74] K.C. Wong, *Discrete-event control architecture: An algebraic approach*, Ph.D. thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, ONT, 1994.

[75] K.C. Wong and J.H. van Schuppen, *Decentralized supervisory control of discrete event systems with communication*, Proc. of WODES 1996 (Edinburgh, UK), Aug 1996, pp. 284–289.

[76] ———, *Decentralized supervisory control of discrete event systems with communication*, Proceedings of WODES 1996 (Edinburgh, UK), Aug. 1996, pp. 284–289.

[77] W. Wonham and P. Ramadge, *On the supremal controllable sublanguage of a given language*, SIAM J. Control Optim **25** (1987), no. 3, 637–659.

[78] ———, *Modular supervisory control of discrete event systems*, Mathematics of Control, Signal and Systems **1** (1988), no. 1, 13–30.

[79] W. Murray Wonham, *Supervisory control of discrete-event systems*, Department of Electrical and Computer Engineering, University of Toronto, July 2005, Monograph and TCT software can be downloaded at http://www.control.toronto.edu/DES/.

[80] T. Yoo and S. Lafortune, *A general architecture for decentralized supervisory control of discrete-event systems*, Proc. of WODES 2000 (Ghent, Belgium), Aug 2000, pp. 111–118.

[81] T. Yoo and S. Lafortune, *A general architecture for decentralized supervisory control of discrete-event systems*, Proceedings of WODES 2000 (Ghent, Belgium), Aug. 2000, pp. 111–118.

[82] Hashtrudi Zad, R.H. Kwong, and W.M. Wonham, *Supremum operators and computation of supremal elements in system theory*, Decision and Control,

1997., Proceedings of the 36th IEEE Conference on, vol. 3, Dec 1997, pp. 2946 – 2951.

[83] Z.H. Zhang, *Smart TCT: an efficient algorithm for supervisory control design.*, Master's thesis, Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, Ont, 2001.

[84] Z.H. Zhang and W. Murray Wonham, *STCT: an efficient algorithm for supervisory control design*, Proc. of SCODES 2001 (INRIA, Paris), July 2001, pp. 82–93.

[85] H. Zhong and W.M. Wonham, *On the consistency of hiearchical supervision in discrete-event systems*, IEEE Transactions on Automatic Control (1990), 1125–1134.

[86] Meng Chu Zhou, David T. Wang, and Israel Mayk, *Using Petri nets for object-oriented design of command and control systems*, International Journal of Intelligent Control and Systems **2** (1998), no. 2, 287–300.

[87] MengChu Zhou and Frank DiCesare, *Petri net synthesis for discrete event control of manufacturing systems*, Kluwer Academic Publishers, 1993.