Hierarchical Interface-based Supervisory Control

Version 1.3.2

by

Ryan James Leduc

A thesis submitted in conformity with the requirements
for the Degree of Doctor of Philosophy,
Graduate Department of Electrical and Computer Engineering,
University of Toronto

Title: Hierarchical Interface-based Supervisory Control

Name: Ryan James Leduc

Degree: Doctor of Philosophy

Year of Convocation: 2002

Department: Electrical and Computer Engineering, University of Toronto

Abstract

In this thesis we present a hierarchical method that decomposes a system into a *high level subsystem* which communicates with $n \geq 1$ parallel *low level subsystems* through separate interfaces, which restrict the interaction of the subsystems. We first define the setting for the serial case ($n = 1$), and then generalise it for $n \geq 1$. We present a definition for an interface, and define a set of interface consistency properties that can be used to verify if a discrete-event system (DES) is nonblocking and controllable. Each clause of the definition can be verified using a single subsystem; thus the complete system model never needs to be constructed, offering significant savings in computational effort. Additionally, the development of clean interfaces facilitates re-use of the component subsystems.

We next provide a set of algorithms for evaluating these properties, and show that the algorithm's time complexity for evaluating a system is $\mathbf{O}(m^2)$, where $m = n + 1$ is the total number of subsystems.

Finally, we present the application of the method to a model of the Atelier Inter-établissement de Productique (AIP), a large (7×10^{21} possible states), highly automated, manufacturing system.

Acknowledgments

I would like to gratefully thank Prof W.M. Wonham for his guidance, support, and patience; for believing in this project and making it possible.

I would like to gratefully acknowledge the support of all my friends and family. Without their help, support and interest, I would never have made it through this project sane. I would like to especially thank Sherif Abdelwahed, Chris Derventzis, Peyman Gohari, Dr. Alireza Langari, Dr. Mark Lawford, Steven Postma, and Dr. Kai Wong, for their support and many interesting discussions.

I'd also like to thank Dr. Bertil Brandin for giving me the opportunity to be a guest scientist at Siemens Corporate Research in Munich Germany where the initial ideas for this thesis were developed, in discussions with Dr. Brandin. Thanks to Dr. Robi Malik for his early review of the material, and his helpful suggestions.

Contents

List of Tables

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. Although many methods have been developed to deal with this problem, large-scale systems are still problematic, particularly for verification of nonblocking.

1.1 Digital Logic Circuits

For inspiration, we first turn to digital logic circuits [56]. Complexity is routinely managed by engineers who design microcomputer processors, containing millions of transistors, for today's personal computers. These circuits are hierarchical in nature, and are designed by using the concept of interfaces to limit the interaction between different levels of the hierarchy. A complex system is designed by first creating basic components, with an interface that encapsulates the behaviour of the component, provides an abstract model of the component's operation, and provides a well defined method of interacting with the component. These components are combined together to create a new, more complex, component, with its own interface. At each step in the process, a component is treated as a black box, and the designer only utilizes the component's interface. At no time is the designer allowed to interact with the inner workings of a component or to "look below" the interface. This keeps the level of complexity at each level manageable.

To illustrate the utility of interfaces for reducing complexity, we look at a simple example.

Figure 1.1 demonstrates many of the principal issues involved. On the left of the diagram, the low level model and the interface representation of a 2 input, 2-bit multiplexer (mux) are shown. The low level model shows a great deal of detail (internal structure, internal signals) that is not needed to operate the circuit. The circuit is then encapsulated by specifying an interface. Only the events shown in the interface model can be seen or directly controlled. Internal signals are completely hidden by the interface. The designer is not permitted to interact with any signals not contained within the interface block, and thus the block is completely portable and reusable. Also, the internal implementation could be changed, but would not affect users of the interface.

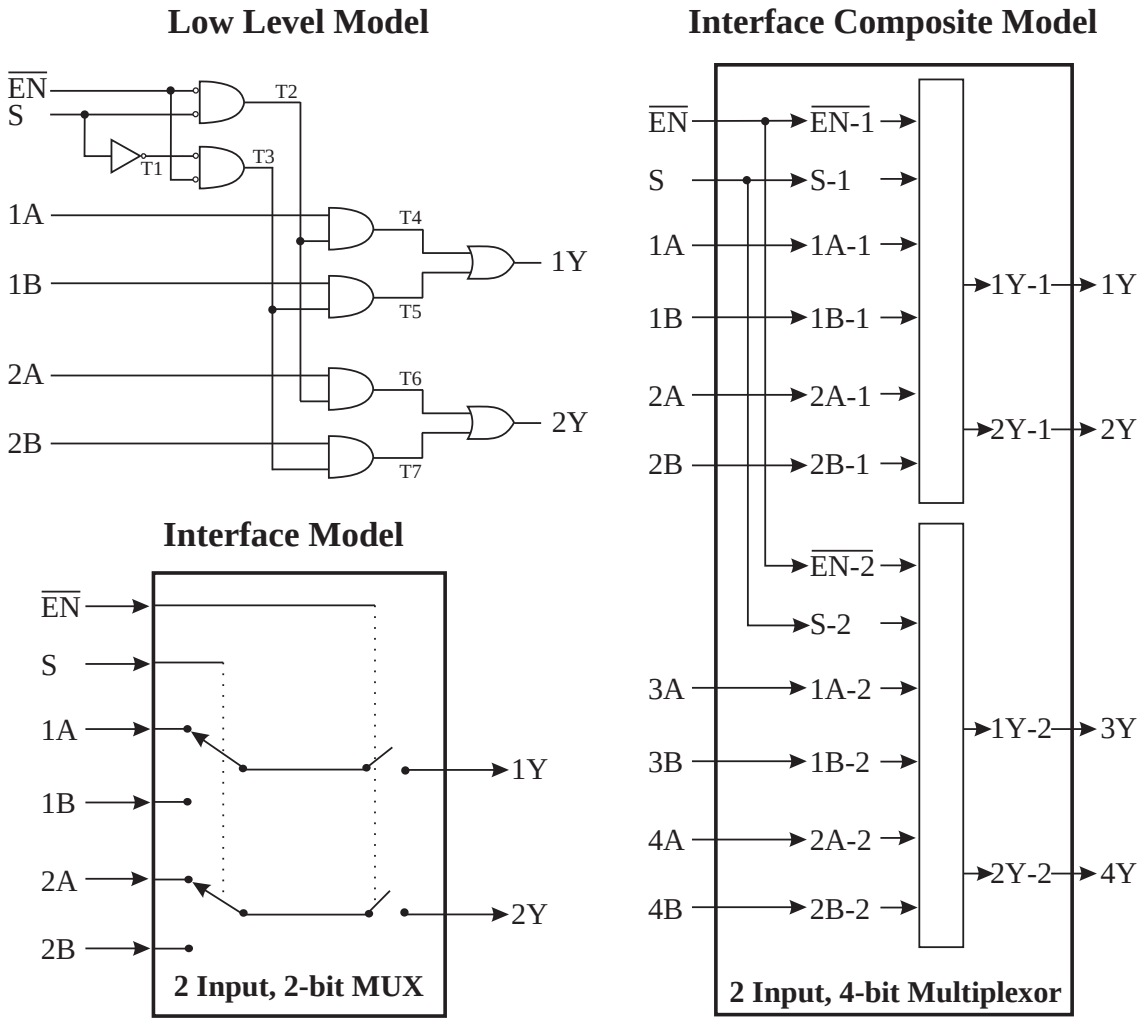


Figure 1.1: Digital Hardware Interface Example

Now that an interface has been designed, it is used as a building block to create more

complex circuits. The right half of Figure 1.1 shows a 2-input, 4-bit mux designed using the interface model of the 2-bit mux. Because the circuit is encapsulated, it can be safely reused. The high level model allows the circuit to be quickly, and easily designed. Also, note how the original interface events in the 2-bit mux (enable, selection and data signals) were all carefully chosen to encapsulate the functionality of the circuit. Low level signals are conspicuously absent.

Next, we note how all levels of the hierarchy are inherently concurrent. Signals at the high level can be changing while signals on one or both of the low levels (the 2 input, 2-bit mux blocks) are changing. The circuit would not be as useful if the high level froze while the low level circuits were operating, or if it could not interact with each low level independently.

Although it is true that the 4-bit mux could be designed using the low level model, and would be more efficient in terms of timing delay and number of gates used, it would require significantly more resources (manipulates 15 blocks as opposed to 2). For large circuits, using low level models would not be practical or efficient as the level of detail would be prohibitive. As can be seen by this example, an interface is an effective means to reduce complexity.

1.2 Software Programs

We now consider software programs, and we see similar ideas at work. To deal with the complexity of large scale systems, the software engineering community has long advocated the decomposition of software into modules (components) that interact via well defined interfaces (e.g., [26, 44, 43, 45]). This approach is referred to as "information hiding" and a well known application is object oriented programming languages [6, 64].

This approach has many advantages. Some of the more important ones are:

1. It limits complexity by hiding unnecessary detail behind interfaces.

2. The method promotes independent development as once the interface to a module is defined, the module and the modules that call it can be designed separately, relying only on the interface specification to ensure interoperability.

3. By encapsulating the behaviour of a module, the interface decouples modules from

each other. As a module is not permitted to know the inner details or interact with the internals of the modules it uses, we can change the implementation of these modules without affecting the module(s) that uses it. This provides a high degree of changeability.

4. The interface provides us with a concise, and meaningful definition of the module. In contrast to DES approaches such as *hierarchical supervisory control* [16, 58, 67], an interface (which can be thought of as an abstract model for the module) is carefully designed before the module (low level) is written. The module is then designed so that it accurately implements the interface. In *hierarchical supervisory control*, the low level is designed first, and then the abstract model is constructed from it.

5. It provides a high degree of comprehensibility. Because information is localized in modules and unnecessary details are hidden by the interface, it is much easier to understand a given module; one only needs to understand the details of the module, and the definitions of the interfaces of the modules it uses.

6. The method provides us with a well defined hierarchical structure. We have a hierarchical structure if we can define a relation between the modules that is a partial ordering. The relation that is usually used is "uses." Module A is at a higher level than module B if module A "uses" module B. This structure is important as it guarantees that we can remove the upper levels of our hierarchy, and what is left can be reused in another application.

1.3 Thesis Overview

Our goal in this thesis is to develop an approach similar to information hiding for DES, and that has the above properties. The method will develop well defined interfaces between components to provide the structure to allow local checks to guarantee global properties such as controllability or nonblocking. However, rather than use the relation "uses" to define our hierarchy, we will instead use the relation "gives work to" (ie. component A is at a higher level than component B if component A "gives work to" component B.) as this relation is more commonly used with interacting, processes (components).

We next note that the supervisory control community has recently begun to advocate

an approach along these lines [30, 21]. These approaches develop interfaces between components to provide the structure to guarantee global properties such as controllability [21] or nonblocking [30] (an early paper based on this thesis).

In this thesis, we present an interface-based hierarchical method, referred to as hierarchical interface-based supervisory control (HISC), to verify if a system is nonblocking and controllable.[1] We describe the application of our method to bi-level systems where the system is split into a *high level subsystem* which interacts with $n \geq 1$ parallel *low level subsystems* via separate interface DES, which regulates the subsystems' interaction.

In the remainder of this thesis, we first describe the setting for the serial case (ie. $n = 1$ and thus only one *low level*). We present a definition for an interface, and define a set of (local) consistency properties that can be used to verify if a discrete-event system is globally nonblocking and controllable. We then discuss a small example to illustrate the approach. Finally, we provide algorithms to verify the serial case properties, and perform a complexity analysis.

We then extend our definitions to the general case of $n \geq 1$ *low level subsystems*, referred to as the parallel case. We then extend our earlier example to illustrate this setting. Finally, we provide algorithms to verify the parallel case properties, and perform a complexity analysis.

We next present the application of the HISC method to a model of the Atelier Inter-établissement de Productique (AIP), a large (7×10^{21} possible states), highly automated, manufacturing system. We first describe the model of the AIP system, then our interfaced based supervisor design, and finally we verify that the system satisfies the interface requirements to show that the system is globally nonblocking and controllable. We close with a discussion of future work, and conclusions.

1.4 Literature Review

In this work, our goal is to develop a scalable method that can handle the combinatorial explosion of the product statespace. We will now review some of the previous approaches to solving this problem.

One of the earliest and most useful methods is *modular control* [20, 48, 55, 60]. This

[1] Part of this information is also available in the research papers [29, 31, 33].

method involves designing multiple supervisors as opposed to a centralized supervisor, each supervisor implementing a portion of the control specification. Although [1, 34] achieved excellent results for verifying controllability, verifying nonblocking was still a problem.

Another approach is *Vector DES* [60, 14, 15, 35] and *Petri Nets* [41, 68, 69]. This is a state based method that makes use of the algebraic regularity inherent in certain systems. It is used when the important details of the system can be expressed as a vector of integers, and events add or subtract these totals. Unfortunately, this method is primarily useful for systems with a high degree of regularity that lends itself to a vector representation.

The next approach is *decentralized control* [5, 36, 52, 53, 59, 63]. In this method, local supervisors were designed with only partial observations of the plant. These supervisors were designed as a group to implement a global specification. While this is a great way of designing distributed controllers, it still requires the computation of the composite plant and thus offers no computational savings over a centralized approach.

A very promising approach is the development of a multi-level hierarchy. [8, 23, 37, 38, 57]. 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. [8, 23, 37, 38, 57]) as opposed to aggregate (bottom up) multi-level hierarchies which we will discuss later. In structural multi-level hierarchies, plants and supervisors are modelled as multi-level structures similar to automata, except certain states at a given level can be expanded into a more detailed lower level model. Although [57] 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, [23] was restricted to using only OR states. Both approaches could verify controllability, but did not address nonblocking.

The next approach of interest is the model aggregation methods [2, 4, 11, 13, 16, 22, 25, 27, 46, 47, 54, 58, 62, 67]. 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 statespaces, they have some drawbacks:

- In hierarchical methods such as [67, 58, 46], 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 can't 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.

The last approach we discuss is the recent work of Zhang et al [65, 66] who have recently developed algorithms that use Integer Decision Diagrams (an extension of Binary Decision Diagrams (BDD:[9])) to verify centralized DES systems on the order of 10^{23} states. This builds upon the work by the model checking/temporal logic community [3, 19, 17, 18, 10, 28, 39, 40, 42, 49] who have successfully used BDDs to handle systems of similar size. This work doesn't represent a hierarchical approach, but a more efficient way to represent DES and verify properties. This means it should be possible to use it to compliment a hierarchical method.

Chapter 2

Discrete-Event Systems Preliminaries

Ramadge-Wonham supervisory control (RW theory:[50], [61], and [60]) provides a theoretical framework for the control of systems that are discrete in space and time. These systems are modelled as automata that generate a formal language of discrete events. These systems are customarily referred to as discrete event systems (DES). The DES are event-driven and may be non-deterministic[1]. The DES do not model when or why an event occurs, just the possible strings of events that the plant can generate. The events are considered to occur in an interleaving fashion. For a detailed discussion of Discrete-event Systems, please refer to [60]. Below, we present a summary of the terminology that we will be using in this thesis.

2.1 Generators

The DES automaton is represented as a 5-tuple as shown below.

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

where Y is the state set (at most countable); Σ is a finite set of event labels (also referred to as the alphabet); δ is the transition function; $y_o \in Y$ is the initial state and $Y_m \subseteq Y$ is the subset of marker states. We will also use the notation $\Sigma_{\mathbf{G}}$ as a shorthand for the event set that DES $\mathbf{G}$ is defined over. This is an easy way to refer to the alphabet given in the

[1]Capable of choosing between two possible next states by chance or unmodelled system dynamics.

5-tuple definition of G, particularly in situations when it is not explicitly stated.[2] For DES $\mathbf{G}$ above, $\Sigma_{\mathbf{G}} = \Sigma$.

The transition function $\delta : Y \,\mathrm{x}\, \Sigma \to Y$ is a partial function and is only defined for a subset of Σ at a given $y \in Y$. The notation $\delta(y, \sigma)!$ indicates that δ is defined for σ at state y.

We want to extend δ to operate on strings in Σ^*, where $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$ where ϵ is the empty string and Σ^+ is the set of all sequences of symbols $\sigma_1\sigma_2\sigma_3 \ldots \sigma_k$, $k \geq 1$ and $\sigma_i \in \Sigma$, $i = 1, 2, \ldots, k$. We now recursively extend δ to the partial function $\delta : Y \,\mathrm{x}\, \Sigma^* \to Y$ by applying the following rules for arbitrary $y \in Y$, $s \in \Sigma^*$ and $\sigma \in \Sigma$:

$$\begin{aligned} \delta(y, \epsilon) &= y \\ \delta(y, s\sigma) &= \delta(\delta(y, s), \sigma) \end{aligned}$$

as long as $y' := \delta(y, s)!$ and $\delta(y', \sigma)!$.

An example of a DES plant is given in Figure 2.1. Here, the plant is composed of two automata, **mach1** and **mach2**. The composite plant model is obtained by taking the synchronous product (defined below) of **mach1** and **mach2**. In the diagram, the entering arrow at state *0* of DES **mach1** indicates that this is the initial state. The exiting arrow indicates that this is a marked state. A transition or event in a DES $\mathbf{G}$ is a triple (y, σ, y') where $y, y' \in Y$, $\sigma \in \Sigma$, and $y' = \delta(y, \sigma)$. An example from DES **mach1** is the event $(0, \alpha_1, 1)$. We say an event $\sigma \in \Sigma$ is *eligible* in DES $\mathbf{G}$ at state $y \in Y$ if $\delta(y, \sigma)!$. For example, event α_1 is eligible at state **I1** in DES **mach1**.

For DES $\mathbf{G}$, the language generated, called the closed behaviour of $\mathbf{G}$, is denoted by $L(\mathbf{G})$, and is defined as follows:

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

The marked behaviour of $\mathbf{G}$, $L_m(\mathbf{G})$, is defined as follows:

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

Clearly, $\emptyset \subseteq L_m(\mathbf{G}) \subseteq L(\mathbf{G})$ and $\epsilon \in L(\mathbf{G})$ as long as $\mathbf{G} \neq \mathbf{EMPTY}$ where **EMPTY** is

[2] For example, in the case of the DES created by the synchronous product operator.

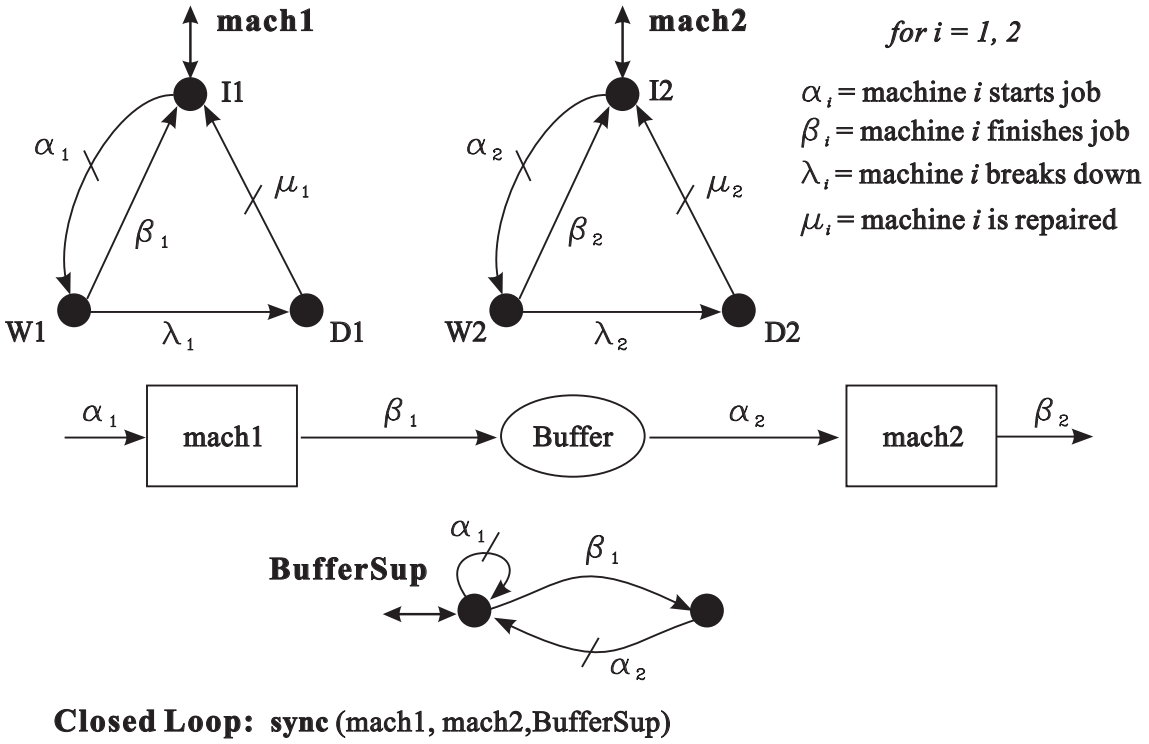


Figure 2.1: Simple Factory Example

the DES with an empty state set.

Next, the reachable state subset of DES $\mathbf{G}$, denoted Y_r, is defined to be:

$$Y_r := \{y \in Y |\ (\exists s \in \Sigma^*)\, \delta(y_o, s) = y\}$$

A DES $\mathbf{G}$ is reachable if $Y_r = Y$. We will always assume $\mathbf{G}$ is reachable as unreachable states don't affect $L(\mathbf{G})$ or $L_m(\mathbf{G})$, and an equivalent reachable DES can always be constructed.

Moving on, the *coreachable* state subset of DES $\mathbf{G}$, denoted Y_{cr}, is defined to be:

$$Y_{cr} := \{y \in Y |\ (\exists s \in \Sigma^*)\, \delta(y, s) \in Y_m\}$$

A DES $\mathbf{G}$ is coreachable if $Y_{cr} = Y$. If a DES is both reachable and coreachable, we say the DES is *trim.*

Finally, we discuss the nonconflicting property. In the definition below, the bar over the languages is the **prefix closure** operator discussed in Section 2.2. We say two languages

$L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Sigma^*$ are *nonconflicting* if:

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

We say that DES G_1 and G_2 are nonconflicting if their marked languages, $L_m(G_1)$ and $L_m(G_2)$, are nonconflicting.

2.2 Operations

In this section, we discuss some useful operations for languages and automata. We start with the **cat** operator. The catenation of strings is defined as follows: $\mathbf{cat} : \Sigma^* \text{ x } \Sigma^* \rightarrow \Sigma^*$ where:

$$\mathbf{cat}(\epsilon, s) = \mathbf{cat}(s, \epsilon) = s, \quad s \in \Sigma^*$$

$$\mathbf{cat}(s, t) = st \quad s, t \in \Sigma^+$$

This leads us to the **prefix closure** operator. A string $t \in \Sigma^*$ is a prefix of $s \in \Sigma^*$ if $s = \mathbf{cat}(t, u)$, for some $u \in \Sigma^*$. The relation "t is a prefix of s" is expressed as $t \leq s$. The prefix closure of a language $L \subseteq \Sigma^*$ is defined as:

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

We say L is closed if $L = \overline{L}$. Clearly as the name indicates, $L(\mathbf{G})$ - the closed behaviour of DES $\mathbf{G}$, is closed.

The natural projection is defined with respect to the subset of a larger alphabet. Let $\Sigma_o \subseteq \Sigma^*$. We define the natural projection $\mathrm{P}_o : \Sigma^* \rightarrow \Sigma_o^*$ as follows:

$$\begin{aligned}
\mathrm{P}_o(\epsilon) &= \epsilon \\
\mathrm{P}_o(\sigma) &= \begin{cases} \epsilon \text{ if } \sigma \notin \Sigma_o \\ \sigma \text{ if } \sigma \in \Sigma_o \end{cases} \\
\mathrm{P}_o(s\sigma) &= \mathrm{P}_o(s)\ \mathrm{P}_o(\sigma) \quad \text{where } s \in \Sigma^*, \sigma \in \Sigma
\end{aligned}$$

Clearly, the natural projection is catenative (i.e. $\mathrm{P}_o(ss') = \mathrm{P}_o(s)\mathrm{P}_o(s')$, where $s, s' \in \Sigma^*$).

Also useful is the inverse image map of the natural projection, given below for set $L \subseteq \Sigma_o^*$:

$$P_o^{-1}(L) \quad := \quad \cup_{s \in L} \{s' \in \Sigma^* | \, P_o(s') = s\}$$

This brings us to the synchronous product of two languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$ ($\Sigma = \Sigma_1 \cup \Sigma_2$), defined using the natural projection. Let P_i be the natural projection of Σ^* onto Σ_i^*, $i = 1, 2$. The synchronous product of L_1 and L_2 is defined to be:

$$L_1 || L_2 = \mathrm{P}_1^{-1}(L_1) \cap \mathrm{P}_2^{-1}(L_2)$$

where P_i^{-1}, $i = 1, 2$, is the inverse image map of the natural projections P_i.

We can now define the $||_s$ operator for DES. For DES $\mathbf{G}_1 = (Y_1, \Sigma_1, \delta_1, y_{o_1}, Y_{m_1})$ and $\mathbf{G}_2 = (Y_2, \ \Sigma_2, \delta_2, y_{o_2}, Y_{m_2})$, the synchronous product is defined to be a reachable DES $\mathbf{G} = \mathbf{G}_1 ||_s \mathbf{G}_2 = (Y, \Sigma, \delta, y_o, Y_m)$ with the properties:

$$L_m(\mathbf{G}) = L_m(\mathbf{G_1}) || L_m(\mathbf{G_2}), \quad L(\mathbf{G}) = L(\mathbf{G_1}) || L(\mathbf{G_2}), \quad \Sigma = \Sigma_1 \cup \Sigma_2$$

where the natural projections P_i, $i = 1, 2$, are as defined above. The $||_s$ operator is essentially the same as the CTCT **sync** operator (see [60]), except that the $||_s$ operator requires that the alphabets of DES G_i be specified explicitly; the **sync** operator takes Σ_i to be the events that appear in G_i. Requiring the alphabets to be explicitly specified, ensures that $||_s$ is associative.

A special case of the synchronous product operator is the **meet** operator. It is equivalent to the $||_s$ when both DES have the same event set. For DES $\mathbf{G}_1$ and $\mathbf{G}_2$ with $\Sigma_1 = \Sigma_2$, then $\mathbf{G} = \mathbf{meet}(\mathbf{G}_1, \mathbf{G}_2)$ is a reachable DES with the properties:

$$L_m(\mathbf{G}) = L_m(\mathbf{G_1}) \cap L_m(\mathbf{G_2}), \quad L(\mathbf{G}) = L(\mathbf{G_1}) \cap L(\mathbf{G_2})$$

Finally, we define the eligibility operator. For a language $L \subseteq \Sigma^*$ and a string $s \in \Sigma^*$, the operator $\mathrm{Elig}_L : \Sigma^* \to \mathrm{Pwr}(\Sigma)$ is defined as below. The notation $\mathrm{Pwr}(\Sigma)$ represents the power set of Σ (the set containing all sets that are $\subseteq \Sigma$).

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

2.3 Nonblocking and Controllability

For DES, the two main properties we want to check are nonblocking and controllability.

Nonblocking: A DES $\mathbf{G}$ is said to be nonblocking if the following is true:[3]

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

Nonblocking means that every string in $L(\mathbf{G})$ can be completed to a marked string. This means that the DES can always return to a marked state. This is a method to check if the DES will deadlock.

To control the plant, we define a supervisor. Supervisors monitor the events generated by the plant, and disable events according to some control law. The supervisors are represented as automata and defined as below:

$$\mathbf{S} = (X, \Sigma_S, \xi, x_o, X_m)$$

In [60], the closed loop behaviour of a plant $\mathbf{G}_1 = (Y_1, \Sigma_1, \delta_1, y_{o_1})$ under the control of supervisor $\mathbf{S}$ is achieved using the **meet** operator as below (assuming that $\Sigma_S = \Sigma_1$):

$$\mathbf{Closed_{meet}}(\mathbf{G}_1, \mathbf{S}) = \mathbf{meet}(\mathbf{G}_1, \mathbf{S})$$

From a practical point of view, using the **meet** operator can make specifying supervisors tedious and error prone; particularly when the event set is large, and the supervisor is specified as several modular supervisors that are then combined using the **meet** operator. As a supervisor must have the same event set as the plant, any event that the supervisor is indifferent to (doesn't care if it's enabled or not) must be selflooped (ie. as event α_1 is at the initial state of DES **BufferSup** in Figure 2.1) at every state. This creates a lot of clutter, and introduces the potential for error if a selfloop is missed.

Instead, we will use the synchronous product operator to specify the closed loop behaviour. When we specify a supervisor, we need only include the events it's concerned with in its event set. Particularly when using a graphical editor to specify and display the supervisor, this simplifies things. We thus define the closed loop behaviour of a plant $\mathbf{G}_1$

[3] An equivalent condition is that every reachable state of $\mathbf{G}$ is coreachable.

under the control of supervisor **S** as follows:

$$\mathbf{Closed}(\mathbf{G}_1, \mathbf{S}) = \mathbf{G}_1 ||_s \mathbf{S}$$

An oddity of using the synchronous product operator to specify the closed loop behaviour is that an event could be in the supervisor, but not in the plant. This may seem unintuitive, but it is useful for events that are not part of the original plant, but are artificially added to aid in the synchronization of supervisors.[4] One could create a new plant DES with such events selflooped at the initial state, but this would make things more cluttered, and possibly increase memory usage in DES analysis software. By using the synchronous product, the result is as if we created this new plant DES automatically.

We now introduce the concept of controllability. Controllability is a way to check if the behaviour restrictions specified by a supervisor are achievable. For the alphabet of interest, we partition it into two disjoint sets: Σ_u and Σ_c. These are the sets of *uncontrollable* and *controllable events,* respectively. Controllable events are events that a supervisor can disable, and thus prevent from occurring. Uncontrollable events can't be disabled. Informally, a supervisor is controllable for a given plant if the plant can't leave the behaviour specified by the supervisor by means of an uncontrollable event.

We now present a formal definition for controllability. We first give the more standard definition with respect to the **meet** operator, again assuming that $\Sigma_S = \Sigma_1$.

Controllability (meet): A supervisor **S** is *controllable* for a plant $\mathbf{G}_1$ if:

$$L(\mathbf{S})\Sigma_u \cap L(\mathbf{G}_1) \subseteq L(\mathbf{S})$$

We will now present the version with respect to the synchronous product operator. First we need to define the event set Σ, the natural projections P_1 and P_S, and languages L_{G_1} and L_S as below:

$$\Sigma := \Sigma_1 \cup \Sigma_S$$

$$P_1 : \Sigma^* \rightarrow \Sigma_1^*$$

[4] An example would be when one supervisor needs to wait until the other reaches a particular state, but there doesn't exist already an event that would signal this uniquely.

$$P_S : \Sigma^* \rightarrow \Sigma_S^*$$

$$L_{G_1} := P_1^{-1} L(\mathbf{G}_1)$$

$$L_S := P_S^{-1} L(\mathbf{S})$$

Controllability ($||_s$): A supervisor $\mathbf{S}$ is *controllable* for a plant $\mathbf{G}_1$ if:

$$L_S \Sigma_u \cap L_{G_1} \subseteq L_S$$

In this thesis, whenever we refer to controllability, we will be referring to the version with respect to synchronous product operator. We will actually use the following equivalent definition:

Alternative Controllability Definition ($||_s$): A supervisor $\mathbf{S}$ is controllable for a plant $\mathbf{G}_1$ if:

$$(\forall s \in L_{G_1} \cap L_S)\ \mathrm{Elig}_{L_{G_1}}(s)\ \cap\ \Sigma_u \subseteq \mathrm{Elig}_{L_S}(s)$$

Modular supervisors are implemented by taking the conjunction of two or more supervisors. We define the conjunction of two supervisors $\mathbf{S_1}$ and $\mathbf{S_2}$ (expressed as $\mathbf{S_1} \wedge \mathbf{S_2}$) as follows:

$$\mathbf{S_1} \wedge \mathbf{S_2} = \mathbf{S_1} ||_s \mathbf{S_2}$$

Returning to the example in Figure 2.1, our plant is $G = \mathbf{mach1}||_s\mathbf{mach2}$ and our supervisor is $S = \mathbf{BufferSup}$. Our event partition can be determined from the diagram by noting that transitions with a bar across them (such as α_1 and α_2) indicate that these are controllable events. Finally, our closed loop system is $\mathbf{Closed(S,G)} = (\mathbf{mach1}||_s\mathbf{mach2})||_s\mathbf{BufferSup} = \mathbf{mach1}||_s\mathbf{mach2}||_s\mathbf{BufferSup}$.

2.4 Nonconflicting Discussion

For our interface-based hierarchical method, we will want each component to be able to operate concurrently with every other component. To achieve this, we will use the synchronous product as our means to combine components. We thus need to develop a set of conditions that can be evaluated on a per component basis that will ensure that if we check nonblocking on a per component basis, we will be guaranteed that when we synchronize the

components together, the result will be nonblocking.

One might naturally think that the concept of nonconflicting might be used. Nonconflicting can be used as a basis of a check to insure that if two trim DES are nonconflicting, then their synchronous product is nonblocking. However, the work involved in verifying that two DES are nonconflicting is about the same as constructing the synchronous product of the two DES, and checking that it's nonblocking. Also, this is a pair-wise check. If we change one of the DES, we would have to reverify that the two DES are still nonconflicting. We desire a component-wise check such that $m \geq 2$ components can be verified separately, such that if we change one component, we only have to recheck the conditions for that component.

Next, we note that if a set of DES are pair-wise nonconflicting, it is not guaranteed that the synchronous product of the group is nonblocking. For example, consider the three trim DES in Figure 2.2. Each DES is nonconflicting with the other two DES, but the synchronous product of all three is not nonblocking.

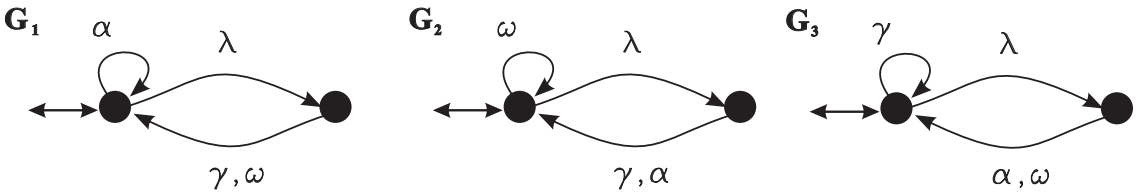


Figure 2.2: Nonconflict Example

It is the author's intuition that the concept of nonconflicting is too strong for our application as it is a necessary and sufficient condition for nonblocking. In other words, it probably requires too much information to be evaluated, and will thus tie the levels of a hierarchy too close together to offer significant savings. We require a sufficient condition that will give us more flexibility in the information that our abstraction can discard.

2.5 Verifying General Properties

In this work, we focus on verifying the controllable and nonblocking properties. These can in turn be used to verify more general properties about a system. For instance we can verify some properties unique to a system by using non-blocking. Say we wanted to determine

for a given system if an event α is always followed by two β events. We could add a DES where only the initial state is marked, and event α takes the DES to an unmarked state. The DES only returns to the initial state after the string $\beta\beta$ occurs. If the resulting system is nonblocking, then we know the property is satisfied.

We can use controllability to determine if a system is failsafe, and to test for fault tolerance. To test if a system is failsafe, we can explicitly model failure events as uncontrollable events, and then specify a supervisor that forbids "undesirable" events (events we don't want to be able to occur after a failure) from occurring when a specific failure event occurs. The failure events should not appear in other supervisors in the system, and the "undesirable" events should be uncontrollable. If the supervisor is controllable, then the system is failsafe for these failures.

Fault tolerance can be tested in a similar manner. Again, we explicitly model failure events and require that supervisors don't contain these events. If the supervisors are controllable, then the system is fault tolerant with respect to these failures.

2.6 Related Propositions

In this section, we present some propositions that will be used in later chapters. The propositions will refer to alphabets Σ_1, Σ_2, and $\Sigma := \Sigma_1 \cup \Sigma_2$, languages $L_1, L_1' \subseteq \Sigma_1^*$, and $L_2, L_2' \subseteq \Sigma_2^*$, and natural projections $P_i : \Sigma^* \to \Sigma_i^*$, where $i = 1, 2$. Finally, we apply these propositions to the synchronous product of DES $\mathbf{G}_1 = (Y_1, \Sigma_1, \delta_1, y_{o_1}, Y_{m_1})$ and $\mathbf{G}_2 = (Y_2, \Sigma_2, \delta_2, y_{o_2}, Y_{m_2})$.

The first two propositions are useful for working with the languages of a DES created by the synchronous product operator. The first proposition essentially says that the inverse natural projection of a closed language is also closed. The second proposition essentially says that the set of prefix closed subsets of Σ^* are closed under intersection.

Proposition 1 *If L_1 is closed, then $P_1^{-1}(L_1)$ is closed.*

Proof:

Assume L_1 is closed. **(1)**

We will now show this implies that $P_1^{-1}(L_1)$ is closed.

This means showing that $P_1^{-1}(L_1) = \overline{P_1^{-1}(L_1)}$

It is sufficient to show that $P_1^{-1}(L_1) \subseteq \overline{P_1^{-1}(L_1)}$ and $\overline{P_1^{-1}(L_1)} \subseteq P_1^{-1}(L_1)$. As $P_1^{-1}(L_1) \subseteq \overline{P_1^{-1}(L_1)}$ is automatic, all that remains to show is $\overline{P_1^{-1}(L_1)} \subseteq P_1^{-1}(L_1)$.

Let $s \in \overline{P_1^{-1}(L_1)}$ **(2)**

We will now show this implies that $s \in P_1^{-1}(L_1)$

We first note that $s \in \overline{P_1^{-1}(L_1)}$ implies that $(\exists s' \in \Sigma^*)\, ss' \in P_1^{-1}(L_1)$

$\Rightarrow P_1(ss') = P_1(s)P_1(s') \in L_1$

$\Rightarrow P_1(s) \in \overline{L_1}$

$\Rightarrow P_1(s) \in L_1$ by **(1)**

$\Rightarrow s \in P_1^{-1}(L_1)$, as required.

We thus have $\overline{P_1^{-1}(L_1)} \subseteq P_1^{-1}(L_1)$, and thus $P_1^{-1}(L_1) = \overline{P_1^{-1}(L_1)}$.

We thus conclude that $P_1^{-1}(L_1)$ is closed.

QED

Proposition 2 *If L_1 and L_2 are closed, then $L_1 \cap L_2$ is closed.*

Proof:

Assume L_1 and L_2 are closed. **(1)**

We will now show this implies that $L_1 \cap L_2$ is closed.

This means showing that $L_1 \cap L_2 = \overline{L_1 \cap L_2}$

It is sufficient to show that $L_1 \cap L_2 \subseteq \overline{L_1 \cap L_2}$ and $\overline{L_1 \cap L_2} \subseteq L_1 \cap L_2$. As $L_1 \cap L_2 \subseteq \overline{L_1 \cap L_2}$ is automatic, all that remains to show is $\overline{L_1 \cap L_2} \subseteq L_1 \cap L_2$.

Let $s \in \overline{L_1 \cap L_2}$ **(2)**

We will now show this implies that $s \in L_1 \cap L_2$

By **(2)**, we can conclude $(\exists s' \in \Sigma^*)\, ss' \in L_1 \cap L_2$

$\Rightarrow s \in \overline{L_1} \cap \overline{L_2}$

$\Rightarrow s \in L_1 \cap L_2$, by **(1)**

We thus have $\overline{L_1 \cap L_2} \subseteq L_1 \cap L_2$, and thus $L_1 \cap L_2 = \overline{L_1 \cap L_2}$, as required.

We thus conclude that $L_1 \cap L_2$ is closed.

QED

We now present a corollary that combines the above two propositions to get a similar result for the || operator.

Corollary 1 *If L_1 and L_2 are closed, then $L_1||L_2$ is closed.*

Proof:

Assume L_1 and L_2 are closed. **(1)**

We will now show this implies that $L_1||L_2$ is closed.

We first note that $L_1||L_2 = \mathrm{P}_1^{-1}(L_1) \cap \mathrm{P}_2^{-1}(L_2)$

From **(1)** and applying **Proposition 1**, we can conclude that languages $\mathrm{P}_1^{-1}(L_1)$ and $\mathrm{P}_2^{-1}(L_2)$ are closed.

We can now apply **Proposition 2** and conclude that $\mathrm{P}_1^{-1}(L_1) \cap \mathrm{P}_2^{-1}(L_2)$, as required.

QED

The next proposition says that the inverse natural projection respects subset ordering.

Proposition 3 *If $L_1 \subseteq L_1'$, then $P_1^{-1}(L_1) \subseteq P_1^{-1}(L_1')$*

Proof:

Assume $L_1 \subseteq L_1'$. **(1)**

We will now show this implies that $P_1^{-1}(L_1) \subseteq P_1^{-1}(L_1')$.

Let $s \in P_1^{-1}(L_1)$ **(2)**

We will now show this implies that $s \in P_1^{-1}(L_1')$.

By **(2)**, we have $P_1(s) \in L_1$

$\Rightarrow P_1(s) \in L_1'$, by **(1)**.

$\Rightarrow s \in P_1^{-1}(L_1')$, as required.

We can thus conclude $P_1^{-1}(L_1) \subseteq P_1^{-1}(L_1')$

QED

The last language proposition says that the synchronous product operator respects subset ordering. This is useful for working with the languages of a DES created by the synchronous product operator.

Proposition 4 *If $L_1 \subseteq L_1'$ and $L_2 \subseteq L_2'$, then $L_1||L_2 \subseteq L_1'||L_2'$*

Proof:

Assume $L_1 \subseteq L_1'$ and $L_2 \subseteq L_2'$. **(1)**

We will now show this implies that $L_1||L_2 \subseteq L_1'||L_2'$

Let $s \in L_1||L_2$ **(2)**

We will now show this implies that $s \in L_1'||L_2'$

From **(2)**, we have $s \in P_1^{-1}(L_1) \cap P_2^{-1}(L_2)$ **(3)**

From **(1)**, we can apply **Proposition 3** twice and conclude $P_1^{-1}(L_1) \subseteq P_1^{-1}(L_1')$ and $P_2^{-1}(L_2) \subseteq P_2^{-1}(L_2')$ **(4)**

Combining with **(3)**, we can now conclude $s \in P_1^{-1}(L_1') \cap P_2^{-1}(L_2')$

$\Rightarrow s \in L_1'||L_2'$ by the definition of the synchronous product operator. We thus have $L_1||L_2 \subseteq L_1'||L_2'$, as required.

QED

Next, we apply the above propositions to the synchronous product of DES.

Proposition 5 *If $G = G_1||_s G_2$, then language $L(G)$ is closed and $L_m(G) \subseteq L(G)$*

Proof:

Assume $G = G_1||_s G_2$. **(1)**

We will now show this implies that language $L(G)$ is closed and $L_m(G) \subseteq L(G)$.

By definition of the $||_s$ operator, we know that $L(G) = L(\mathbf{G_1})||L(\mathbf{G_2})$ and that $L_m(\mathbf{G}) = L_m(\mathbf{G_1})||L_m(\mathbf{G_2})$ **(2)**

We now note that languages $L(G_1)$, and $L(G_2)$ are closed by the definition of the closed behavior of a DES.

We can immediately apply **Corollary 1** and conclude that $L(G)$ is closed.

From the definition of the closed behavior and the marked language of a DES, we can conclude that $L_m(\mathbf{G_1}) \subseteq L(\mathbf{G_1})$ and $L_m(\mathbf{G_2}) \subseteq L(\mathbf{G_2})$.

We can now apply **Proposition 4** and conclude:

$$L_m(\mathbf{G_1})||L_m(\mathbf{G_2}) = \ L_m(G) \subseteq L(G) \ = L(\mathbf{G_1})||L(\mathbf{G_2})$$

QED

For our last proposition and its accompanying corollary, we need to introduce some different notation to avoid confusion with later notation. We will be using alphabets $\Sigma_a, \Sigma_b \subseteq \Sigma$ and natural projections $P_k : \Sigma^* \to \Sigma_k^*$, where $k = a, b$.

The following proposition provides a useful relationship for natural projections when the language each is projecting onto has the given relationship. When we examine the parallel case, we will see many instances of this relationship.

Proposition 6 *If* $\Sigma_b \subseteq \Sigma_a$ *then* $P_a^{-1} \cdot P_a \cdot P_b^{-1} = P_b^{-1}$

Proof:

Assume $\Sigma_b \subseteq \Sigma_a$. **(1)**

We will now show this implies $P_a^{-1} \cdot P_a \cdot P_b^{-1} = P_b^{-1}$

Let $s \in \Sigma_b^*$. Sufficient to show $P_b^{-1}(s) = P_a^{-1} \cdot P_a \cdot P_b^{-1}(s)$

This means showing: **I)** $P_b^{-1}(s) \subseteq P_a^{-1} \cdot P_a \cdot P_b^{-1}(s)$ and **II)** $P_a^{-1} \cdot P_a \cdot P_b^{-1}(s) \subseteq P_b^{-1}(s)$

I) Show $P_b^{-1}(s) \subseteq P_a^{-1} \cdot P_a \cdot P_b^{-1}(s)$

Let $s' \in P_b^{-1}(s)$. Will show implies $s' \in P_a^{-1} \cdot P_a \cdot P_b^{-1}(s)$.

$\Rightarrow \ P_a(s') \in P_a \cdot P_b^{-1}(s)$

$\Rightarrow \ P_a^{-1} \cdot P_a(s') \subseteq P_a^{-1} \cdot P_a \cdot P_b^{-1}(s)$

Clearly $s' \in P_a^{-1} \cdot P_a(s')$, as $P_a^{-1} \cdot P_a(s') := \{s'' \in \Sigma^* \mid P_a(s'') = P_a(s')\}$.

$\Rightarrow \ s' \in P_a^{-1} \cdot P_a \cdot P_b^{-1}(s)$, as required.

Case I complete.

II) Show $P_a^{-1} \cdot P_a \cdot P_b^{-1}(s) \subseteq P_b^{-1}(s)$

Let $s' \in P_a^{-1} \cdot P_a \cdot P_b^{-1}(s) := \cup_{u \in P_a \cdot P_b^{-1}(s)} \{v \in \Sigma^* \mid P_a(v) = u\}$. **(2)**

Will show implies $s' \in P_b^{-1}(s)$. Sufficient to show $P_b(s') = s$.

From **(2)**, we have $P_a(s') \in P_a \cdot P_b^{-1}(s)$

$\Rightarrow$ $(\exists s'' \in \Sigma^*)$ s.t. $s'' \in P_b^{-1}(s) \wedge (P_a(s'') = P_a(s'))$ **(3)**

As $s'' \in P_b^{-1}(s)$, we have $P_b(s'') = s$

Since $\Sigma_b \subseteq \Sigma_a$ (from **(1)**), we can conclude: $(\forall t \in \Sigma^*)\, P_b(t) = P_b \cdot P_a(t)$ **(4)**

From this we can conclude $P_b \cdot P_a(s'') = P_b(s'') = s$

From **(3)**, we have $P_a(s'') = P_a(s')$. We can thus conclude $P_b \cdot P_a(s') = P_b \cdot P_a(s'') = s$

From **(4)**, we can conclude $P_b(s') = P_b \cdot P_a(s') = s$

$\Rightarrow$ $s' \in P_b^{-1}(s)$, as required.

Case II complete.

By **Case I** and **Case II**, we have $P_b^{-1}(s) = P_a^{-1} \cdot P_a \cdot P_b^{-1}(s)$ and thus conclude $P_a^{-1} \cdot P_a \cdot P_b^{-1} = P_b^{-1}$

QED

The corollary below applies **Proposition 6** to provide a useful result for strings. We will be using this corollary extensively when we examine the parallel case.

Corollary 2 *If* $\Sigma_b \subseteq \Sigma_a$ *and* $L_b \subseteq \Sigma_b^*$ *then* $(\forall s \in \Sigma^*)\ P_a(s) \in P_a \cdot P_b^{-1}(L_b) \Rightarrow s \in P_b^{-1}(L_b)$

Proof:

Assume $\Sigma_b \subseteq \Sigma_a$ and $L_b \subseteq \Sigma_b^*$. **(1)**

Let $s \in \Sigma^*$ and $P_a(s) \in P_a \cdot P_b^{-1}(L_b)$. **(2)**

We will now show this implies that $s \in P_b^{-1}(L_b)$

We start by noting that **(2)** implies that $s \in P_a^{-1} \cdot P_a \cdot P_b^{-1}(L_b)$

From **Proposition 6**, we can conclude $P_a^{-1} \cdot P_a \cdot P_b^{-1} = P_b^{-1}$, by **(1)**.

$s \in P_b^{-1}(L_b)$ follows automatically.

QED

Chapter 3

Serial Case: Nonblocking

With the serial case of *hierarchical interface-based supervisory control,* what we are proposing is essentially a master-slave system, where a *high level subsystem* sends a command to a *low level subsystem,* which then performs the indicated task and sends back a reply. Figure 3.1 shows conceptually the structure of the system.

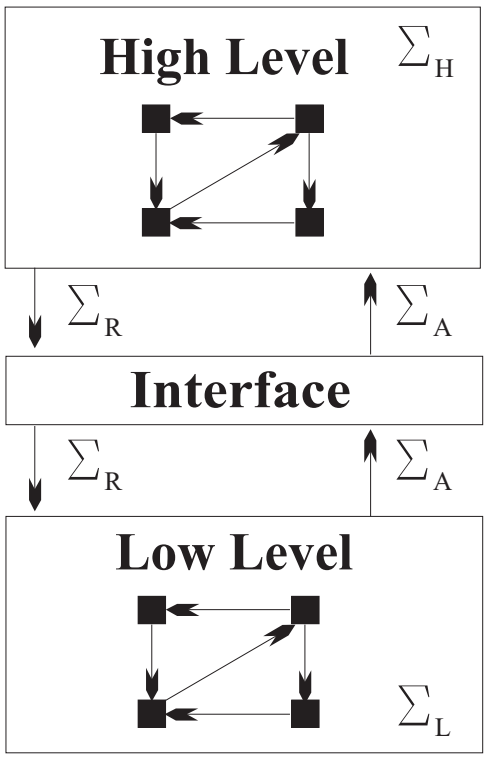


Figure 3.1: Interface Block Diagram.

To allow the system to be designed/maintained/verified on a component-wise basis, we impose an interface between the two subsystems that limits their interaction and knowledge of each other. The goal is to be able to work with each subsystem individually, requiring no information about the other subsystem beyond that provided by the interface.

To capture the restriction of the flow of information imposed by the *interface,* we split the alphabet of the system (Σ) into four mutually disjoint alphabets: Σ_H, Σ_L, Σ_R, and Σ_A. The events in Σ_H are called *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, respectively.

The alphabets Σ_R and Σ_A are called collectively *interface events.* These events are common to both levels of the hierarchy and represent communication between the two subsystems. More specifically, the events in Σ_R are called *request events* and represent commands sent from the *high level subsystem* to the *low level subsystem.* The events in Σ_A are called *answer events* and represent the low level's responses to the *request events* (high-level commands). Figure 3.1 shows conceptually the flow of information in our setting.

In the remainder of the chapter, we will first present a definition for an interface, followed by a set of local consistency and nonblocking requirements that the interface and subsystems must satisfy in order to guarantee global nonblocking. We then provide several supporting propositions, followed by the serial nonblocking theorem.

3.1 Notation and Definitions

In this section, we present a definition for interface, and some notation that will be useful in the following proofs.

3.1.1 Interface Definition

In this section we, will present two interface definitions: *star interfaces* and *command-pair interfaces.* As we will see later, *star interfaces* are a special case of the more general *command-pair interfaces.*

We start by describing a *star interface* as it has a regular structure and is thus easy to construct. To define a *star interface,* the designer selects a set of *request events*, and then for each *request event*, the designer defines a set of *answer events.* In essence, the designer defines a map $\mathbf{Answer} : \Sigma_R \to \mathrm{Pwr}(\Sigma_A)$. For $\rho \in \Sigma_R$, $\mathbf{Answer}(\rho)$ is the set of possible

answers (referred to as the *answer set*) the *low level subsystem* could provide after receiving request ρ. For consistency, we add the constraints given below. **Point 1** states that the *low level subsystem* must provide at least one response for each request it receives, and **point 2** states that Σ_A does not contain any unused events.

1. $(\forall \rho \in \Sigma_R)\, \mathbf{Answer}(\rho) \neq \emptyset$
2. $\Sigma_A = \cup_{\rho \in \Sigma_R} \mathbf{Answer}(\rho)$

In Figure 3.2, we see how a *star interface,* with $n = |\Sigma_R|$ $(n \geq 0)$, is expressed as a DES. The required structure for a *star interface* is given by DES G_I. Analysing DES G_I, we see that the DES has the following properties:

- The initial state is the only marked state.
- *Request events* are the only events defined at the initial state.
- Each *request event* starts at the initial state, and ends at a state other than the initial state.
- *Answer events* are not defined at the initial state
- At least one *answer event* transition can always follow a *request event* transition.

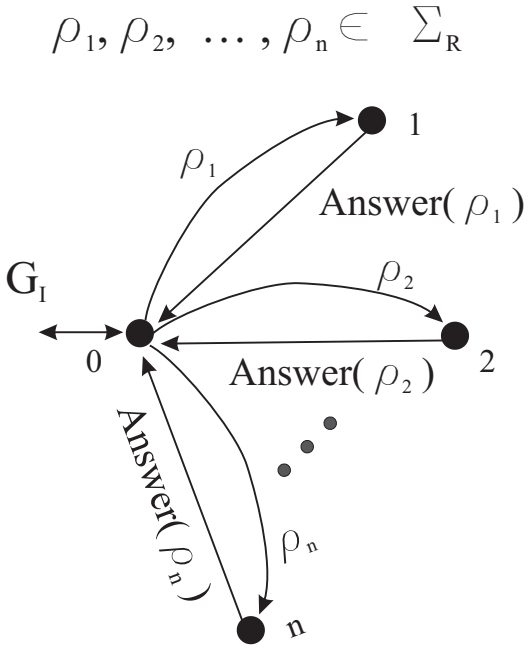


Figure 3.2: Interface Specification.

To allow for a minimal DES, we permit distinct *request events* to have the same next state. For example, if *request events* ρ_1 and ρ_2 in Figure 3.2 had the same *answer sets* (i.e. $\mathbf{Answer}(\rho_1) = \mathbf{Answer}(\rho_2)$), then their next states can be merged. In our example, this would mean states 1 and 2 of G_I would be combined. Finally, we require that the event set of G_I be set to $\Sigma_R \dot{\cup} \Sigma_A$ but we place no restrictions on whether a *request* or *answer event* is controllable or uncontrollable.

We now define *command-pair interfaces. Command-pair interfaces* were designed as a generalisation of *star interfaces. Star interfaces* were designed first as they were more intuitive, and then the key properties were identified and collected into the *command-pair interface* definition. A key difference is that the "star" shape is no longer required. A *command-pair interface* will still always has a *request event* followed by an *answer event,* but it can now contain additional state information. For example, in Figure 3.2 all possible *request events* are defined at the initial state. When an *answer event* has occurred, it always returns the *star interface* to the initial state, and thus the same choice of potential *request events*. With a *command-pair interface* we can have a DES structure as illustrated in Figure 3.3. *Request events* ρ_1 and ρ_2 might represent the regular behaviour of the system, while α_3 and ρ_3 represent breakdown and repair of the system. A *command-pair interface* allows the flexibility of only having the repair event eligible after a breakdown.

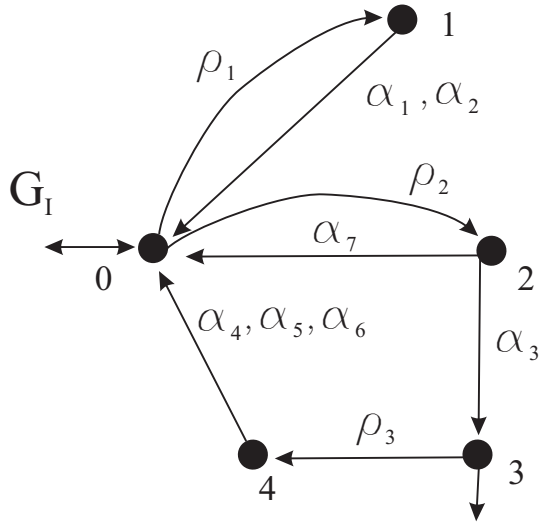


Figure 3.3: Example *Command-pair Interface.*

For the remainder of this work, when we refer to an *interface* we will mean explicitly a

command-pair interface, and we will use the two synonymously. We define a *command-pair interface* as below:

Definition: A DES $G_I = (X, \Sigma_I, \xi, x_o, X_m)$ is a *command-pair interface* if the following conditions are satisfied:

(A) $\Sigma_I = \Sigma_R \dot{\cup} \Sigma_A$

(B) $(\forall s \in L(G_I))(\forall \rho \in \Sigma_R)\ s\rho \in L(G_I) \Rightarrow s \in L_m(G_I)$

(C) $(\forall s \in L_m(G_I))(\forall \sigma \in \Sigma_I)\ s\sigma \in L(G_I) \Rightarrow \sigma \notin \Sigma_A$

(D) $L_m(G_I) = \{\epsilon\} \cup (\Sigma_I^*.\Sigma_A \cap L(G_I))$

(E) $L(G_I) \subseteq \overline{(\Sigma_R.\Sigma_A)^*}$

The first point, **point A**, says that G_I's event set is restricted to *request* and *answer events.* It also states that the two sets are disjoint. **Point B** then states that *request event* transitions are only defined at marked states. **Point C** states that there are no *answer events* defined at marked states. **Point D** says that the marked language of G_I consists of the empty string, and strings that end in an *answer event.* Finally, **Point E** says that in the language of G_I, a *request event* always occurs first and then *request* and *answer events* alternate.

We will now prove that *star interfaces* are a special case of *command-pair interfaces.* This will allow us to prove our theorems for the more general *command-pair interfaces,* but use the simpler *star interfaces* when they are sufficient.[1]

Proposition 7 *If DES* $G_I = (X, \Sigma_I, \xi, x_o, X_m)$ *is a* star interface, *then* G_I *is a* command-pair interface.

Proof:

Assume that G_I is a *star interface.*

We will now show this implies it's a *command-pair interface* by showing that G_I satisfies **points A-E** of the *command-pair interface* definition.

Point A: Show $\Sigma_I = \Sigma_R \dot{\cup} \Sigma_A$

[1]We will actually use *star interfaces* exclusively in our algorithms and examples as the *command-pair interface* definition was only just developed and there wasn't time to update our software and examples.

This is automatic from the *star interface* definition.

Point B: Show $(\forall s \in L(G_I))(\forall \rho \in \Sigma_R)\ s\rho \in L(G_I) \Rightarrow s \in L_m(G_I)$

The results follow immediately from observing Figure 3.2 and noting that *request event* transitions are only defined at the initial state, which is marked.

Point C: Show $(\forall s \in L_m(G_I))(\forall \sigma \in \Sigma_I)\ s\sigma \in L(G_I) \Rightarrow \sigma \notin \Sigma_A$

The results follow immediately from observing Figure 3.2 and noting that *answer event* transitions are not defined at the initial state, which is the only marked state.

Point D: Show $L_m(G_I) = \{\epsilon\} \cup (\Sigma_I^*.\Sigma_A \cap L(G_I))$

It is sufficient to show **1)** $L_m(G_I) \subseteq \{\epsilon\} \cup (\Sigma_I^*.\Sigma_A \cap L(G_I))$ and **2)** $L_m(G_I) \supseteq \{\epsilon\} \cup (\Sigma_I^*.\Sigma_A \cap L(G_I))$

Case 1) From Figure 3.2, we can see that $L_m(G_I)$ includes the empty string, as the initial state is marked and that G_I is not the **EMPTY** DES. We also see that the only transitions ending at the initial state (the only marked state) are for *answer events.* The results follow immediately.

Case 2) From above, we have that $L_m(G_I)$ includes the empty string. From Figure 3.2, we can see that every *answer event* transition ends at the initial state, which is marked. The results follow immediately.

By **case 1** and **2**, we thus have $L_m(G_I) = \{\epsilon\} \cup (\Sigma_I^*.\Sigma_A \cap L(G_I))$

Point E: Show $L(G_I) \subseteq \overline{(\Sigma_R.\Sigma_A)^*}$

From Figure 3.2, we can see that $L(G_I)$ contains the empty string (as G_I is not the **EMPTY** DES), and strings that start with a *request event.* We further see that *request* and *answer events* then alternate. The results follow immediately.

QED

3.1.2 Terminology and Notation

We now present some terminology and notation that will be useful in simplifying proofs. For our setting, we assume the *high level subsystem* is modelled by DES G_H (defined over event set $\Sigma_H \cup \Sigma_R \cup \Sigma_A$), the *low level subsystem* by DES G_L (defined over event set $\Sigma_L \cup \Sigma_R \cup \Sigma_A$), and the interface by DES G_I. Also, the term *high level* will mean the DES $G_H||_s G_I$, and the term *low level* the DES $G_L||_s G_I$. The overall structure of the system is displayed in Figure 3.4.

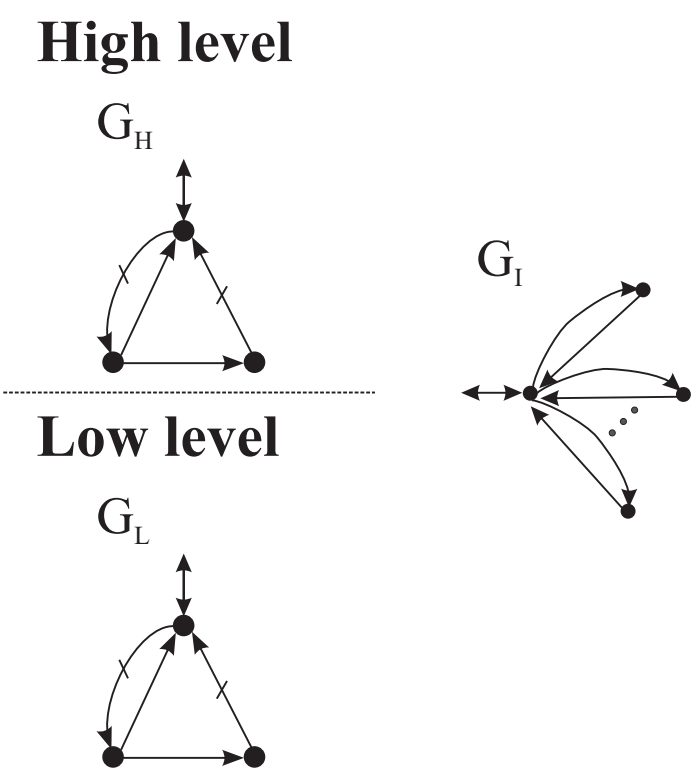


Figure 3.4: Two Tiered Structure of Serial System.

We next assume that the alphabet partition is specified by $\Sigma := \Sigma_H \dot{\cup} \Sigma_L \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A$ and define the *flat system* as below. By *flat system* we refer to the equivalent DES that would represent our system if we ignored the interface structure.

$$G = G_H ||_s G_L ||_s G_I$$

As we will often be referring to different groupings of events, we define the following subsets:

$$\begin{aligned} \Sigma_I &:= \Sigma_R \dot{\cup} \Sigma_A && \textit{Interface Events} \\ \Sigma_{IH} &:= \Sigma_H \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A && \textit{Interface \& High Level Events} \\ \Sigma_{IL} &:= \Sigma_L \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A && \textit{Interface \& Low Level Events} \end{aligned}$$

To work with languages defined over subsets of Σ, we define the following natural projec-

tions:

$$\begin{aligned} P_{IH} : \Sigma^* \rightarrow \quad & \Sigma_{IH}^* \\ P_{IL} : \Sigma^* \rightarrow \quad & \Sigma_{IL}^* \\ P_I : \Sigma^* \rightarrow \quad & \Sigma_I^* \end{aligned}$$

As we want to express the languages of *flat system* in terms of their components, we need to define the following languages:

$$\begin{aligned} \mathcal{H} := P_{IH}^{-1}(L(G_H)), \quad & \mathcal{H}_m := P_{IH}^{-1}(L_m(G_H)) \quad \subseteq \Sigma^* \\ \mathcal{L} := P_{IL}^{-1}(L(G_L)), \quad & \mathcal{L}_m := P_{IL}^{-1}(L_m(G_L)) \quad \subseteq \Sigma^* \\ \mathcal{I} := P_I^{-1}(L(G_I)), \quad & \mathcal{I}_m := P_I^{-1}(L_m(G_I)) \quad \subseteq \Sigma^* \end{aligned}$$

We can now represent the closed behaviour of our *flat system* as follows:

$$\begin{aligned} L(G) := \quad & L(G_H||_s G_L||_s G_I) \\ = \quad & P_{IH}^{-1}(L(G_H)) \cap P_{IL}^{-1}(L(G_L)) \cap P_I^{-1}(L(G_I)) \\ = \quad & \mathcal{H} \cap \mathcal{L} \cap \mathcal{I} \end{aligned}$$

Similarly, the marked language of our *flat system* is:

$$L_m(G) = \mathcal{H}_m \cap \mathcal{L}_m \cap \mathcal{I}_m$$

This allows us to present the proposition below which collects together several similar propositions. As it will be common in the proofs in this report to show that membership in languages such as $\mathcal{H}$ are dependent only on events in specific subsets (for $\mathcal{H}$, events in subset Σ_{IH}), this proposition will be very useful.

Proposition 8

(a) $(\forall s, s' \in \Sigma^*)\ s \in \mathcal{H}$ *and* $P_{IH}(s) = P_{IH}(s') \Rightarrow s' \in \mathcal{H}$

(b) $(\forall s, s' \in \Sigma^*)\ s \in \mathcal{H}_m$ *and* $P_{IH}(s) = P_{IH}(s') \Rightarrow s' \in \mathcal{H}_m$

(c) $(\forall s, s' \in \Sigma^*)\ s \in \mathcal{L}$ *and* $P_{IL}(s) = P_{IL}(s') \Rightarrow s' \in \mathcal{L}$

(d) $(\forall s, s' \in \Sigma^*)\ s \in \mathcal{L}_m$ *and* $P_{IL}(s) = P_{IL}(s') \Rightarrow s' \in \mathcal{L}_m$

(e) $(\forall s, s' \in \Sigma^*)\ s \in \mathcal{I}$ *and* $P_I(s) = P_I(s') \Rightarrow s' \in \mathcal{I}$

(f) $(\forall s, s' \in \Sigma^*)\ s \in \mathcal{I}_m$ *and* $P_I(s) = P_I(s') \Rightarrow s' \in \mathcal{I}_m$

Proof:

Point a:

Let $s, s' \in \Sigma^*$, $s \in \mathcal{H}$ and $P_{IH}(s) = P_{IH}(s')$ **(1)**

$s \in \mathcal{H} \Rightarrow s \in P_{IH}^{-1}(L(G_H))$, by definition of $\mathcal{H}$.

$\Rightarrow P_{IH}(s) \in L(G_H)$

$\Rightarrow P_{IH}(s') \in L(G_H)$, by **(1)**.

$\Rightarrow s' \in P_{IH}^{-1}(L(G_H)) = \mathcal{H}$, as required.

Points b-f:

Identical to the proof of **point a** above, after substitution.

QED

3.2 Serial Interface Consistent and Nonblocking

In this section, we present the interface properties that our system must satisfy to ensure that it interacts with the *interface* correctly as well as the nonblocking requirements each level must satisfy. Together they provide a set of local conditions that can be evaluated using at most one level of our hierarchy at a time.

Our first definition is the *serial level-wise nonblocking* definition. It requires that each level be individually nonblocking.

Serial Level-wise Nonblocking: The system composed of DES G_H, G_L, and G_I, is said to be *serial level-wise nonblocking* if the following conditions are satisfied:

(I) $\overline{\mathcal{H}_m \cap \mathcal{I}_m} = \mathcal{H} \cap \mathcal{I}$ *Nonblocking at the high level*

(II) $\overline{\mathcal{L}_m \cap \mathcal{I}_m} = \mathcal{L} \cap \mathcal{I}$ *Nonblocking at the low level*

We now present the *serial interface consistent* definition. It defines the interface properties that our system must satisfy to ensure that it interacts with the *interface* correctly.

It limits the information each level can have about the other, and what assumptions they can make about each other.

Serial Interface Consistent: 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 are satisfied:

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. $(\forall s \in \mathcal{H} \cap \mathcal{I})\ \text{Elig}_{\mathcal{I}}(s) \cap \Sigma_A \subseteq \text{Elig}_{\mathcal{H}}(s)$ *High level task completion agreement*

Low Level Properties

4. $(\forall s \in \mathcal{L} \cap \mathcal{I})\ \text{Elig}_{\mathcal{I}}(s) \cap \Sigma_R \subseteq \text{Elig}_{\mathcal{L}}(s)$ *Low level task request agreement*
5. $(\forall s \in \Sigma^*.\Sigma_R \cap \mathcal{L} \cap \mathcal{I})$
   $\text{Elig}_{\mathcal{L}\cap\mathcal{I}}(s\Sigma_L^*) \cap \Sigma_A = \text{Elig}_{\mathcal{I}}(s) \cap \Sigma_A$ *Low level task completion agreement*
   where $\text{Elig}_{\mathcal{L}\cap\mathcal{I}}(s\Sigma_L^*) := \cup_{l\in\Sigma_L^*} \text{Elig}_{\mathcal{L}\cap\mathcal{I}}(sl)$
6. $(\forall s \in \mathcal{L} \cap \mathcal{I})$
   $s \in \mathcal{I}_m \Rightarrow (\exists l \in \Sigma_L^*)\ sl \in \mathcal{L}_m \cap \mathcal{I}_m$ *Low level marking agreement*

We will now give a brief discussion of the meaning of each property.

Property 0: The first Property is inherent in the definition of the alphabet partition, $\Sigma := \Sigma_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$. It states that the four alphabets are pairwise disjoint.

Property 1: This property asserts that the high and low levels can only share *request* and *answer events.* This is an "information hiding" statement. It restricts the *high level subsystem* from knowing (and directly affecting) internal details about the *low level subsystem* (ie to be able to view/disable *low level events*) and vice versa.

Property 2: This property states that DES G_I satisfies the definition of a *command-pair interface.*

Property 3: This property asserts that the *high level subsystem* (G_H) must always accept an *answer event* if the event is eligible in the *interface.* If the *answer event* is not eligible in the *interface,* the *high level subsystem* is not required to accept it. This is equivalent to a controllability condition where the *interface* is taken to be the plant, the *high level subsystem* the supervisor, and *answer events* to be the uncontrollable events. 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.*

Property 4: This property asserts that the *low level subsystem* (G_L) must always accept a *request event* if the event is eligible in the *interface.* If the *request event* is not eligible in the *interface,* the *low level subsystem* is not required to accept it. This is equivalent to a controllability condition where the *interface* is taken to be the plant, the *low level subsystem* the supervisor, and *request events* to be the uncontrollable events. In other words, the *low level subsystem* is forbidden to assume more about when an *request event* can occur than what is provided by the *interface.*

Property 5: This property asserts that immediately after a *request event* (some $\rho \in \Sigma_R$) has occurred (and before it is followed by any *low level events*), there exists one or more paths via strings in Σ_L^* to each *answer event* (ie all $\alpha \in \mathbf{Answer}(\rho)$, assuming that we are dealing with a *star interface*) that can follow the *request event.* A given path may only lead to one of the possible *answer events,* but a path to each one must exist. However, as soon as a single *low level event* has occurred, one or more *answer events* may no longer be reachable (ie the *low level* may no longer be able to provide that particular *answer event*).

For example, assume our *low level* represents a machine that accepts the *request events* **startJob** and **startRepair**. *Request event* **startJob** can be followed by *answer events* **jobCompleted** or **machineDown**. *Request event* **startRepair** can be followed by *answer event* **repairCompleted** (this information would be embodied in $L(G_I)$). Immediately after *request event* **startJob** has occurred, there must be one or more *low level event* sequences, accepted by the *low level subsystem*, that lead to *answer events* **jobCompleted** and **machineDown**. For example, sequence **taskAComplete taskBComplete** would bring the *low level subsystem* into a state in which it would accept *answer event* **jobCompleted** but event **machineDown**

would no longer be possible. Similarly, sequence **taskAComplete machineFailure** would bring the *low level subsystem* into a state that it would accept *answer event* **machineDown** but event **jobCompleted** would no longer be possible. What's important to note here is that both *answer events* **jobCompleted** and **machineDown** were initially reachable after **startJob** occurred. Which *answer event* was allowed to occur was determined afterwards solely by *low level events.* Also, after one or more *low level events* had occurred, there was no guarantee that both *answer events* were still reachable.

One could summarize the purpose of this condition as to guarantee "honest advertising." If the *interface* asserts that a given *answer event* can follow a given *request event,* this must always be true at least immediately after the *request event* occurs, and then the *low level subsystem* is allowed through the occurrence of *low level events* to decide which *answer event* actually occurs. In our above example, our *interface* advertises that either *answer event* **jobCompleted** or **machineDown** can follow *request event* **startJob**. If the *low level subsystem* was in a down state such that only **machineDown** was possible, and *request event* **startJob** occurred, this condition would be violated. If the *high level* could only reach a marked state by the eventual occurrence of event **jobCompleted**, it would be deceived by the "false advertising" of the *interface* into believing that this could eventually be satisfied by repeatedly issuing **startJob** commands. Of course, a human designer would see the fallacy of this, but an automatic synthesis algorithm would be deceived.

Property 6: This property asserts that every string marked by the *interface* and accepted by the *low level subsystem*, can be extended by a low level string to a string marked by the *low level* (both G_I and G_L). In other words, once the *low level subsystem* has returned an *answer event*, it can always return to a marked state via a low level string (ie some $s \in \Sigma_L^*$).

From the point of the view of the *high level subsystem*, this property says that if one can bring the *high level subsystem* and the *interface* to a marked state, then one can bring the *low level subsystem* to a marked state via a low level string which would be ignored by the *high level* (ie they would stay in a marked state).

We are now ready to state the proposition below which establishes useful properties for

often used languages.

Proposition 9 *If the system composed of DES G_H, G_L, and G_I is* serial interface consistent *with respect to the alphabet partition $\Sigma := \Sigma_H \dot{\cup} \Sigma_L \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A$, then the following is true:*

(i) Languages $\mathcal{H}$, $\mathcal{L}$, and $\mathcal{I}$ are closed.

(ii) $\mathcal{H}_m \subseteq \mathcal{H}$, $\mathcal{L}_m \subseteq \mathcal{L}$, and $\mathcal{I}_m \subseteq \mathcal{I}$

Proof: See page 42.

Now that we have presented the *serial interface consistent* definition, we present the *serial interface strict marking* condition, which is a restriction of the *serial interface consistent* definition. As we will see later, this restriction is useful as it implies **Property 6** of the *serial interface consistent* definition, but is less expensive to evaluate.

Serial Interface Strict Marking: The system composed of DES G_H, G_L and G_I, is *serial interface strict marking* with respect to the alphabet partition $\Sigma := \Sigma_H \dot{\cup} \Sigma_L \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A$, if:

$(\forall s \in \mathcal{L} \cap \mathcal{I})\ s \in \mathcal{I}_m \Rightarrow\ s \in \mathcal{L}_m \cap \mathcal{I}_m$

The above statement could be summarized by saying that if we can bring the *interface* to a marked state, we are guaranteed to also have brought the *low level subsystem* to a marked state. The above statement is equivalent to **Property 6** of the *serial interface consistent* definition above, with string l restricted to the empty string.

We will now prove that we can use the *serial interface strict marking* condition to replace **Property 6** of the *serial interface consistent* definition.

Proposition 10 *If the system composed of DES G_H, G_L and G_I, satisfies* **properties 1-5** *of the* serial interface consistent *definition and is* serial interface strict marking *with respect to the alphabet partition $\Sigma := \Sigma_H \dot{\cup} \Sigma_L \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A$, then the system is* serial interface consistent.

Proof:

Assume that the system satisfies **properties 1-5** of the *serial interface consistent* definition

and is *serial interface strict marking.*

We now show this implies the system is *serial interface consistent.*

From **(1)**, we immediately have the system satisfying the first 5 properties of the *serial interface consistent* definition.

All that remains is to show that the system satisfies **Property 6** of the definition. This means showing:

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

Let $s \in \mathcal{L} \cap \mathcal{I}$, and $s \in \mathcal{L}_m$

We will now show this implies $(\exists l \in \Sigma_L^*)\ sl \in \mathcal{L}_m \cap \mathcal{I}_m$

From **(1)**, we have that the system is *serial interface strict marking.*

We can thus conclude: $s \in \mathcal{L}_m \cap \mathcal{I}_m$

We thus take $l = \epsilon$ and we have $sl \in \mathcal{L}_m \cap \mathcal{I}_m$, as required.

We thus have the system satisfying **properties 1-6,** and is thus *serial interface consistent.*

QED

3.3 Serial Nonblocking Propositions and Theorem

We will now present **Propositions 11-15**, followed by our main result for this chapter, **Theorem 1**. The following propositions perform two tasks: they break down the main theorem into a more manageable size, as well as provide useful results that will be re-used for the parallel case.

3.3.1 Low Level Nonblocking Proposition

Our first proposition is the *low level nonblocking proposition.* It asserts that a string s accepted by the system, can always be extended to a string accepted by the system, and marked by the *low level.* In other words, the *low level* is not dependent on high level events to reach a marked state.

Proposition 11 *If the system composed of DES G_H, G_L, and G_I is* serial level-wise

nonblocking *and* serial interface consistent *with respect to the alphabet partition* $\Sigma := \Sigma_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$, *then*

$(\forall s \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I})$

$\quad (\exists l \in \Sigma_{IL}^*)$ *s.t.* $(sl \in \mathcal{H} \cap \mathcal{L}_m \cap \mathcal{I}_m)$

Proof: See page 43.

3.3.2 Low Level Linkage Proposition

Our next proposition is the *low level linkage proposition.* It asserts that a string s accepted by the *system* and marked by the *high level,* implies that s can be extended by a low level string l such that sl is marked by the *system.* In other words, if you can get the *high level* to a marked state, you can always bring the low level to a marked state by a string containing events the *high level* is indifferent to.

Proposition 12 *If the system composed of DES* G_H, G_L, *and* G_I *is* serial level-wise nonblocking *and* serial interface consistent *with respect to the alphabet partition* $\Sigma := \Sigma_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$, *then*

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

Proof:

Assume system is *serial level-wise nonblocking* and *serial interface consistent.* **(1)**

Let $s \in \mathcal{L} \cap \mathcal{H}_m \cap \mathcal{I}_m$ **(2)**

We will now show that we can construct a string $l \in \Sigma_L^*$ such that $sl \in \mathcal{H}_m \cap \mathcal{L}_m \cap \mathcal{I}_m$

From **(2)**, we have $s \in \mathcal{L} \cap \mathcal{I}_m$ and thus $s \in \mathcal{L} \cap \mathcal{I}$.

From **(1)**, we can apply **Point 6** of the *serial interface consistent* definition and conclude:

$(\exists l \in \Sigma_L^*)$ s.t. $sl \in \mathcal{L}_m \cap \mathcal{I}_m$ **(3)**

As $l \in \Sigma_L^*$, we have $P_{IH}(s) = P_{IH}(sl)$. From **(2)**, we have $s \in \mathcal{H}_m$. We can now apply **Proposition 8**, **point b**, and conclude:

$sl \in \mathcal{H}_m$

Combining with **(3)**, have $l \in \Sigma_L^*$, and $sl \in \mathcal{H}_m \cap \mathcal{L}_m \cap \mathcal{I}_m$, as required.

QED

3.3.3 Event Agreement Propositions

We group the next three propositions together, as each builds upon the previous one.

One-step Event Agreement Proposition

Our first proposition is the one-step event agreement proposition. For this proposition, we are given a string s accepted by the *system* and a string h of the form $\Sigma_H^*.\Sigma_R.\Sigma_H^*.\Sigma_A$. This means that h contains exactly one request event and one *answer event* in the given order and that h may or may not contain *high level events* before or directly after the *request event*. The proposition asserts that if string h extends string s such that sh is acceptable to the *high level*, then a string u can be constructed such that u has a high level image equal to h, and that su is acceptable to the system and marked by the *interface*. In other words, we can use string h as a basis to construct string u by adding low level events so that the *low level subsystem* will accept the request and *answer event* contained in h. As these events are common to both levels, they must agree on their occurrence.

Proposition 13 *If the system composed of DES G_H, G_L, and G_I is* serial level-wise nonblocking *and* serial interface consistent *with respect to the alphabet partition* $\Sigma := \Sigma_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$*, then*

$$(\forall s \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I})(\forall h \in \Sigma_H^*.\Sigma_R.\Sigma_H^*.\Sigma_A)$$
$$sh \in \mathcal{H} \cap \mathcal{I} \Rightarrow (\exists u \in \Sigma^*)\ s.t.\ (su \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m) \wedge (P_{IH}(u) = h)$$

Proof: See page 45.

Inductive Event Agreement Proposition

Our next proposition is the inductive event agreement proposition. This proposition is different from **Proposition 13** as **Proposition 13** only handled the case that the string h contains exactly one *answer event* (i.e. only one command-pair), while this proposition allows h to contain one or more *answer events* (i.e. multiple command-pairs). It uses **Proposition 13** in an inductive proof to handle an arbitrary number of *answer events* in string h.

Proposition 14 *If the system composed of DES G_H, G_L, and G_I is* serial level-wise nonblocking *and* serial interface consistent *with respect to the alphabet partition* $\Sigma := \Sigma_H \dot{\cup} \Sigma_L \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A$, *then*

$(\forall s \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m)(\forall h \in \Sigma_{IH}^*.\Sigma_A)$

$sh \in \mathcal{H} \cap \mathcal{I} \Rightarrow (\exists u \in \Sigma^*)\ (P_{IH}(u) = h) \wedge (su \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m)$

Proof: See page 47.

General Event Agreement Proposition

The last proposition of the three is the general event agreement proposition. This proposition is more general than **Proposition 14** as it handles the case that string h doesn't contain *answer events* or doesn't end in an *answer event.* It makes use of **Proposition 14** to handle the other cases.

Proposition 15 *If the system composed of DES G_H, G_L, and G_I is* serial level-wise nonblocking *and* serial interface consistent *with respect to the alphabet partition* $\Sigma := \Sigma_H \dot{\cup} \Sigma_L \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A$, *then*

$(\forall s \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m)(\forall h \in \Sigma_{IH}^*)$

$sh \in \mathcal{H}_m \cap \mathcal{I}_m \Rightarrow (\exists u \in \Sigma^*)\ s.t.\ (su \in \mathcal{H}_m \cap \mathcal{L}_m \cap \mathcal{I}_m) \wedge (P_{IH}(u) = h) \wedge (P_I(u) \in \{\epsilon\} \cup \Sigma_R.\Sigma_I^*)$

Proof: See page 50.

3.3.4 Serial Nonblocking Theorem

We now present our main result for this chapter, the *serial interface nonblocking theorem.* In essence the theorem says that if the *high level* and *low level* are individually nonblocking, and the system is *serial interface consistent,* then the nonblocking property will be preserved by the synchronous product operation. As the *serial level-wise nonblocking* and *serial interface consistent* definitions can be evaluated by examining only one level of our system at a time, we now have a means to verify nonblocking of our system using local checks.

Theorem 1 *If the system composed of DES G_H, G_L, and G_I is* serial level-wise nonblocking *and* serial interface consistent *with respect to the alphabet partition* $\Sigma := \Sigma_H \dot{\cup} \Sigma_L \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A$, *then*

$L(G) = \overline{L}_m(G)$, *where* $G = G_H||_s G_L||_s G_I$

Proof:

Assume system is *serial level-wise nonblocking* and *serial interface consistent.* **(1)**

As $\overline{L_m(G)} \subseteq L(G)$ is automatic, it suffices to show $L(G) \subseteq \overline{L_m(G)}$

Let $s \in L(G) = \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}$ **(2)**

We will now show this implies $s \in \overline{L_m(G)}$

It is sufficient to show: $(\exists u \in \Sigma^*)\ su \in L_m(G) = \mathcal{H}_m \cap \mathcal{L}_m \cap \mathcal{I}_m$

Our first step is to show that we can construct a string l, accepted by the high level, that will bring the low level to a marked state.

We can achieve this immediately by noting that $s \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}$ and **(1)** allows us to exploit **Proposition 11**. We thus conclude:

$$(\exists l \in \Sigma_{IL}^*) \text{ s.t. } sl \in \mathcal{H} \cap \mathcal{L}_m \cap \mathcal{I}_m \quad \textbf{(3)}$$

We will now show that we can extend string sl to a string in $\mathcal{H}_m \cap \mathcal{L}_m \cap \mathcal{I}_m$. To do this, we will use **Proposition 15**. To apply the proposition, we must first construct a string $h \in \Sigma_{IH}^*$ with the property $slh \in \mathcal{H}_m \cap \mathcal{I}_m$

We first note that we have $sl \in \mathcal{H} \cap \mathcal{I}$ from **(3)**. This allows us to apply **Point I** of the *level-wise nonblocking* definition (nonblocking at the high level), and conclude:.

$$(\exists h' \in \Sigma^*) \text{ s.t. } slh' \in \mathcal{H}_m \cap \mathcal{I}_m \quad \textbf{(4)}$$

We next note that:

$$\begin{aligned} P_{IH}(slh') &= P_{IH}(sl)P_{IH}(h') \\ &= P_{IH}(sl)P_{IH}(P_{IH}(h')) \text{ as the natural projection is idempotent.} \\ &= P_{IH}(slP_{IH}(h')) \end{aligned} \quad \textbf{(5)}$$

We can now apply **Proposition 8**, **point b**, and conclude $slP_{IH}(h') \in \mathcal{H}_m$ **(6)**

As $\Sigma_I \subseteq \Sigma_{IH}$, we can conclude $P_I(slh') = P_I(slP_{IH}(h'))$ (by **(5)**).

We can now apply **Proposition 8**, **point f**, and conclude $slP_{IH}(h') \in \mathcal{I}_m$

Combining with **(6)**, we can conclude $slP_{IH}(h') \in \mathcal{H}_m \cap \mathcal{I}_m$

We thus take $h = P_{IH}(h')$ and we have:

$h \in \Sigma_{IH}^*$ and $slh \in \mathcal{H}_m \cap \mathcal{I}_m$ **(7)**

We next note we have $sl \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m$, by **(3)**. We can now apply **Proposition 15**, taking sl to be string s in that proposition, and conclude:

$(\exists u' \in \Sigma^*)$ s.t. $slu' \in \mathcal{H}_m \cap \mathcal{L}_m \cap \mathcal{I}_m$

We then take $u = lu'$ and we have $su \in \mathcal{H}_m \cap \mathcal{L}_m \cap \mathcal{I}_m = L_m(G)$, as required.

QED

3.4 Proofs of Selected Propositions

In order to make this work more readable, the proofs of some propositions in this chapter were not given as the propositions were introduced. They will now be presented in the following sections.

3.4.1 Proof of Proposition 9

Proof for **Proposition 9** on page 36: If the system composed of DES G_H, G_L, and G_I is *serial interface consistent* with respect to the alphabet partition $\Sigma := \Sigma_H \dot{\cup} \Sigma_L \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A$, then the following is true:

(i) Languages $\mathcal{H}$, $\mathcal{L}$, and $\mathcal{I}$ are closed.

(ii) $\mathcal{H}_m \subseteq \mathcal{H}$, $\mathcal{L}_m \subseteq \mathcal{L}$, and $\mathcal{I}_m \subseteq \mathcal{I}$

Proof:

Assume system is *serial interface consistent.* **(1)**

Will now show this implies that **Points i** and **ii** are satisfied.

We first note that by **(1)**, the system is *serial interface consistent.* By **Points 1** and 2 of this definition, we can conclude:

$L(G_H), L_m(G_H) \subseteq \Sigma_{IH}^*$, $L(G_L), L_m(G_L) \subseteq \Sigma_{IL}^*$, and $L(G_I), L_m(G_I) \subseteq \Sigma_I^*$.

This tells us that languages $\mathcal{H} = P_{IH}^{-1}(L(G_H))$, $\mathcal{L} = P_{IL}^{-1}(L(G_L))$, $\mathcal{I} = P_I^{-1}(L(G_I))$, $\mathcal{H}_m = P_{IH}^{-1}(L_m(G_H))$, $\mathcal{L}_m = P_{IL}^{-1}(L_m(G_L))$, and $\mathcal{I}_m = P_I^{-1}(L_m(G_I))$ are defined.

Point i: Show that Languages $\mathcal{H}$, $\mathcal{L}$, and $\mathcal{I}$ are closed.

We now note that languages $L(G_H)$, $L(G_L)$, and $L(G_I)$ are closed by the definition of the closed behaviour of a DES.

We can now apply **Proposition 1** repeatedly and conclude that $\mathcal{H}$, $\mathcal{L}$, and $\mathcal{I}$ are closed, as required.

Point ii: Show that $\mathcal{H}_m \subseteq \mathcal{H}$, $\mathcal{L}_m \subseteq \mathcal{L}$, and $\mathcal{I}_m \subseteq \mathcal{I}$.

From the definition of the closed behaviour and the marked language of a DES, we can conclude that:

$$L_m(G_H) \subseteq L(G_H),\ L_m(G_L) \subseteq L(G_L), \text{ and } L_m(G_I) \subseteq L(G_I).$$

Applying **Proposition 3** repeatedly, we can conclude:

$$\begin{aligned} P_{IH}^{-1}(L_m(G_H)) = \mathcal{H}_m &\quad\subseteq\quad \mathcal{H} = P_{IH}^{-1}(L(G_H)) \\ P_{IL}^{-1}(L_m(G_L)) = \mathcal{L}_m &\quad\subseteq\quad \mathcal{L} = P_{IL}^{-1}(L(G_L)) \\ P_{I}^{-1}(L_m(G_I)) = \mathcal{I}_m &\quad\subseteq\quad \mathcal{I} = P_{I}^{-1}(L(G_I)) \end{aligned}$$

QED

3.4.2 Proof of Proposition 11

Proof for **Proposition 11** on page 37: *If the system composed of DES G_H, G_L, and G_I is* serial level-wise nonblocking *and* serial interface consistent *with respect to the alphabet partition $\Sigma := \Sigma_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$, then*

$(\forall s \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I})$

$\quad (\exists l \in \Sigma_{IL}^*)$ *s.t.* $(sl \in \mathcal{H} \cap \mathcal{L}_m \cap \mathcal{I}_m)$

Proof:

Assume system is *serial level-wise nonblocking* and *serial interface consistent.* **(1)**

Let $s \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}$ **(2)**

Will now show this implies $(\exists l \in \Sigma_{IL}^*)$ s.t. $(sl \in \mathcal{H} \cap \mathcal{L}_m \cap \mathcal{I}_m)$

To do this, we will construct a suitable string l. We start by applying **Point II** of the *serial level-wise nonblocking* definition (by **(1)**) to conclude:

$$(\exists s' \in \Sigma^*)\ ss' \in \mathcal{L}_m \cap \mathcal{I}_m \qquad \textbf{(3)}$$

As we require a string in Σ_{IL}^*, we take $l' = P_{IL}(s') \in \Sigma_{IL}^*$. We will now show $sl' \in \mathcal{L}_m \cap \mathcal{I}_m$

From the definition of the natural projection, we have $P_{IL}(P_{IL}(s')) = P_{IL}(s')$

$\Rightarrow P_{IL}(ss') = P_{IL}(s)P_{IL}(s') = P_{IL}(s)P_{IL}(P_{IL}(s')) = P_{IL}(sP_{IL}(s')) = P_{IL}(sl')$, as the natural projection
is catenative.

As $\Sigma_I \subseteq \Sigma_{IL}$, $P_I(s') = P_I(P_{IL}(s))$. We can thus argue as above and conclude: $P_I(ss') = P_I(sl')$

We can now use **(3)**, and apply **Proposition 8**, **points d** and **f**, and conclude:

$$sl' \in \mathcal{L}_m \cap \mathcal{I}_m \qquad \textbf{(4)}$$

We now note that if $sl' \in \mathcal{H}$, we can take $l = l'$ and we have the desired result. We can thus, with no loss in generality, assume:

$$sl' \notin \mathcal{H} \qquad \textbf{(5)}$$

We will now show this implies that string sl' is not accepted by $\mathcal{H}$ due to a *request event*. We will then use this fact to construct a suitable string l.

We first observe that $s \in \mathcal{H}$, $sl' \notin \mathcal{H}$, and $l' \in \Sigma_{IL}^*$ implies:
$(\exists l'' \in \Sigma_{IL}^*)(\exists \sigma \in \Sigma_{IL})$ s.t. $(l''\sigma \leq l') \wedge (sl'' \in \mathcal{H}) \wedge (sl''\sigma \notin \mathcal{H})$ **(6)**

We note the following points, which will be used later in the proof:

- $sl'' \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}$ by the facts $sl'' \in \mathcal{H}$, $l'' < l'$, $sl' \in \mathcal{L}_m \cap \mathcal{I}_m$, and the fact $\mathcal{L}$ and $\mathcal{I}$ are closed languages. **(7)**
- $sl''\sigma \in \mathcal{I}$ by **(4)**, **(6)**, and fact $\mathcal{I}$ is closed. **(8)**

We now show that **(6)** implies $\sigma \in \Sigma_R$. We know:

- $\sigma \notin \Sigma_L$ as $\sigma \in \Sigma_L$ would imply $P_{IH}(sl''\sigma) = P_{IH}(sl'')$ Since $sl'' \in \mathcal{H}$, **Proposition 8**, **point a**, would then imply $sl''\sigma \in \mathcal{H}$, which would contradict **(6)**.
- $\sigma \notin \Sigma_A$ as $\sigma \in \Sigma_A$, $sl''\sigma \in \mathcal{I}$ (by **(8)**) and **point 3** of the *serial interface consistent* definition would imply $sl''\sigma \in \mathcal{H}$, which would contradict **(6)**.

As $\sigma \in \Sigma_{IL} = \Sigma_R \dot{\cup} \Sigma_A \dot{\cup} \Sigma_L$, we can thus conclude $\sigma \in \Sigma_R$ by process of elimination.

We now show that $\sigma \in \Sigma_R$ implies $sl'' \in \mathcal{I}_m$. This will allow us to use **Point 6** of the *serial interface consistent* definition to extend sl'' to a string marked by the *low level.*

From **Point 2** of the *serial interface consistent* definition, we have that DES G_I is a *command-pair interface.*

As $\sigma \in \Sigma_R$ and $sl''\sigma \in \mathcal{I}$ (from **(9)**), we can conclude: $P_I(sl''\sigma) = P_I(sl'')\sigma \in L(G_I)$

We can thus conclude by **Point B** of the *command-pair interface* definition that $P_I(sl'') \in L_m(G_I)$

$\Rightarrow sl'' \in \mathcal{I}_m$

From **(7)**, we have $sl'' \in \mathcal{L} \cap \mathcal{I}$ so we can now apply **Point 6** of the *serial interface consistent* definition and conclude:
$(\exists l''' \in \Sigma_L^*)$ s.t. $sl''l''' \in \mathcal{L}_m \cap \mathcal{I}_m$

As $l'' \in \Sigma_{IL}^*$ from **(6)**, we have $l''l''' \in \Sigma_{IL}^*$

We take $l = l''l'''$ and immediately have $sl \in \mathcal{L}_m \cap \mathcal{I}_m$ and $l \in \Sigma_{IL}^*$. All that remains is to show $sl \in \mathcal{H}$

From **(6)**, we have $sl'' \in \mathcal{H}$. As $l''' \in \Sigma_L^*$, we have $P_{IH}(sl'') = P_{IH}(sl''l''') = P_{IH}(sl)$

We can thus apply **Proposition 8**, **point a**, and conclude $sl \in \mathcal{H}$, as required.

QED

3.4.3 Proof of Proposition 13

Proof for **Proposition 13** on page 39: *If the system composed of DES G_H, G_L, and G_I is* serial level-wise nonblocking *and* serial interface consistent *with respect to the alphabet partition $\Sigma := \Sigma_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$, then*

$(\forall s \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I})(\forall h \in \Sigma_H^*.\Sigma_R.\Sigma_H^*.\Sigma_A)$
$sh \in \mathcal{H} \cap \mathcal{I} \Rightarrow (\exists u \in \Sigma^*)$ *s.t.* $(su \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m) \wedge (P_{IH}(u) = h)$

Proof:

Assume system is *serial level-wise nonblocking* and *serial interface consistent.* **(1)**

Let $s \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}$, $h \in \Sigma_H^*.\Sigma_R.\Sigma_H^*.\Sigma_A$, and $sh \in \mathcal{H} \cap \mathcal{I}$ **(2)**

We will now show this implies we can construct a string u with the desired properties.

We first note that $h \in \Sigma_H^*.\Sigma_R.\Sigma_H^*.\Sigma_A$ implies:
$(\exists h' \in \Sigma_H^*)(\rho \in \Sigma_R)(h'' \in \Sigma_H^*)(\alpha \in \Sigma_A)$ s.t. $h'\rho h''\alpha = h$ **(3)**

We will now show that we can construct a string $l \in \Sigma_L^*$ such that $h'\rho l h''\alpha \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m$. We will also show that $P_{IH}(h'\rho l h''\alpha) = h$.

Our approach will be to show that $sh'\rho \in \mathcal{L} \cap \mathcal{I}$ and $\alpha \in \text{Elig}_{\mathcal{I}}(sh'\rho)$. We will then use **Point 5** of the *serial interface consistent* definition to construct a suitable string l.

We next note that $sh \in \mathcal{H} \cap \mathcal{I}$ (by **(2)**), and that $h'\rho \leq h$ (by **(3)**). As $\mathcal{H}$ and $\mathcal{I}$ are closed languages, we can now conclude:

$$sh'\rho \in \mathcal{H} \cap \mathcal{I} \qquad \textbf{(4)}$$

As $h' \in \Sigma_H^*$ (by **(3)**), we have $P_{IL}(sh') = P_{IL}(s)$. As $s \in \mathcal{L}$ (by **(2)**), we can now apply **Proposition 8**, **point c**, and conclude:

$$sh' \in \mathcal{L} \qquad \textbf{(5)}$$

We now have $sh' \in \mathcal{L} \cap \mathcal{I}$ and $\rho \in \text{Elig}_{\mathcal{I}}(sh')$ (by **(4)**).

This allows us to apply **Point 4** of the *serial interface consistent* definition (by **(1)**) and conclude:

$\rho \in \text{Elig}_{\mathcal{L}}(sh')$ and thus $sh'\rho \in \mathcal{L}$

$\Rightarrow sh'\rho \in \mathcal{L} \cap \mathcal{I}$, by **(4)**.

As $h'' \in \Sigma_H^*$ by **(3)**, we can conclude $P_I(sh'\rho h''\alpha) = P_I(sh'\rho)P_I(h'')P_I(\alpha) = P_I(sh'\rho\alpha)$

From **(2)** and **(3)**, we have $sh'\rho h''\alpha \in \mathcal{I}$. We can now apply **Proposition 8**, **point e**, and conclude:

$sh'\rho\alpha \in \mathcal{I}$

$\Rightarrow \alpha \in \text{Elig}_{\mathcal{I}}(sh'\rho)$.

We can now apply **Point 5** of the *interface consistency properties* and conclude:

$$(\exists l \in \Sigma_L^*) \text{ s.t. } \alpha \in \text{Elig}_{\mathcal{L}\cap\mathcal{I}}(sh'\rho l). \qquad \textbf{(6)}$$

$\Rightarrow sh'\rho l\alpha \in \mathcal{L} \cap \mathcal{I}$

As $h'' \in \Sigma_H^*$ by **(3)**, we can conclude $P_{IL}(sh'\rho l\alpha) = P_{IL}(sh'\rho l h''\alpha)$ and $P_I(sh'\rho l\alpha) = P_I(sh'\rho l h''\alpha)$. We can now apply **Proposition 8**, **points c** and **e**, and conclude:

$$sh'\rho l h''\alpha \in \mathcal{L} \cap \mathcal{I} \qquad \textbf{(7)}$$

We next note that DES G_I is a *command-pair interface* by **(1)**.

As $\alpha \in \Sigma_A$ (by **(3)**), we can now conclude:

$$P_I(sh'\rho lh''\alpha) \in \Sigma_I^*.\Sigma_A \cap L(G_I)$$

$\Rightarrow P_I(sh'\rho lh''\alpha) \in L_m(G_I)$ by **point D** of the *command-pair interface* definition.

$\Rightarrow sh'\rho lh''\alpha \in \mathcal{I}_m$ **(8)**

From **(2)** and **(3)**, we have $sh'\rho h''\alpha \in \mathcal{H}$. As $l \in \Sigma_L^*$ (by **(6)**), we can conclude:

$$P_{IH}(sh'\rho h''\alpha) = P_{IH}(sh'\rho lh''\alpha). \quad \textbf{(9)}$$

We can now apply **Proposition 8**, **point a**, and conclude:

$$sh'\rho lh''\alpha \in \mathcal{H}$$

Combining with **(7)**, **(8)** and **(9)**, we have $sh'\rho lh''\alpha \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m$ and $P_{IH}(h'\rho lh''\alpha) = P_{IH}(h'\rho h''\alpha) = h$

We take $u = h'\rho lh''\alpha$, and the proof is complete.

QED

3.4.4 Proof of Proposition 14

Proof for **Proposition 14** on page 40: *If the system composed of DES G_H, G_L, and G_I is* serial level-wise nonblocking *and* serial interface consistent *with respect to the alphabet partition $\Sigma := \Sigma_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$, then*

$(\forall s \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m)(\forall h \in \Sigma_{IH}^*.\Sigma_A)$

$sh \in \mathcal{H} \cap \mathcal{I} \Rightarrow (\exists u \in \Sigma^*)\ (P_{IH}(u) = h) \wedge (su \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m)$

Proof:

Assume system is *serial level-wise nonblocking* and *serial interface consistent.* **(1)**

Let $s \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m$, $h \in \Sigma_{IH}^*.\Sigma_A$, and $sh \in \mathcal{H} \cap \mathcal{I}$ **(2)**

We will now show this implies we can construct a string u with the desired properties.

Our approach will be to break string h into substrings containing pairs of *request* and *answer events*. We will then construct u iteratively, using these substrings.

We first let n be the number of *answer events* in string h ($n \geq 1$ as $h \in \Sigma_{IH}^*.\Sigma_A$). This implies:

$(\exists h_1, h_2, \ldots, h_n \in (\Sigma_H \cup \Sigma_R)^*.\Sigma_A)$ s.t. $h_1 h_2 \ldots h_n = h$ **(3)**

We have thus broken h into n strings each containing one answer event at the end of the string.
From **(2)**, we immediately have: $sh_1 h_2 \ldots h_n \in \mathcal{H} \cap \mathcal{I}$ **(4)**

Using an inductive proof, we will now show:
$(\exists u_0, u_1, \ldots, u_n \in \Sigma^*)$ s.t. $(su_0 u_1 \ldots u_n \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m) \wedge (P_{IH}(u_0 u_1 \ldots u_n) = h_0 h_1 \ldots h_n)$, where $h_0 := \epsilon$

Claim to be proven:

For $k \in \{0, 1, \ldots, n\}$, there exists $u_0, u_1, \ldots, u_k \in \Sigma^*$ such that the following are true: **(5)**

(a) $P_{IH}(u_0 u_1 \ldots u_k) = h_0 h_1 \ldots h_k$

(b) $su_0 u_1 \ldots u_k \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m$

We will first prove the initial case ($k = 0$), and then the general case of $k \in \{1, \ldots, n\}$. We can then conclude by induction that the claim has been proven.

Initial Case: $k = 0$

We take $u_0 = \epsilon$ and we immediately have $P_{IH}(u_0) = \epsilon = h_0$, and thus **Property a** of **(5)** is satisfied.

We have $su_0 \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m$ as $su_0 = s$ and $s \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m$ (by **(2)**), and thus **Property b** of **(5)** is satisfied.

Initial case complete.

Inductive Step:

Let $k \in \{1, \ldots, n\}$. Assume $\exists u_0, u_1, \ldots, u_{k-1} \in \Sigma^*$ and that they satisfy **Properties a** and **b** of **(5)** when $k-1$ is substituted for k. **(6)**

We will show that this implies $\exists u_k \in \Sigma^*$ that satisfies **Properties a** and **b** of **(5)**.

Our approach will be to apply **Proposition 13**. To do this, our first step is to show that
$su_0 u_1 \ldots u_{k-1} h_k \in \mathcal{H} \cap \mathcal{I}$

We first note that $sh_0 h_1 \ldots h_k \in \mathcal{H} \cap \mathcal{I}$ by **(4)**, and the facts that $h_0 = \epsilon$ and $\mathcal{H}$ and $\mathcal{I}$ are closed
languages. **(7)**

From **(6)**, we have $P_{IH}(u_0u_1 \dots u_{k-1}) = h_0h_1 \dots h_{k-1}$. From **(3)** and fact $h_0 = \epsilon$, we have $P_{IH}(h_0h_1 \dots h_k) = h_0h_1 \dots h_k$. We can thus conclude:

$$\begin{aligned} P_{IH}(su_0u_1 \dots u_{k-1}h_k) &= P_{IH}(s)P_{IH}(u_0u_1 \dots u_{k-1})P_{IH}(h_k) \\ &= P_{IH}(s)h_0h_1 \dots h_{k-1}P_{IH}(h_k) \\ &= P_{IH}(sh_0h_1 \dots h_k) \qquad \textbf{(8)} \end{aligned}$$

With **(7)**, we can now apply **Proposition 8**, **point a**, and conclude $su_0u_1 \dots u_{k-1}h_k \in \mathcal{H}$ **(9)**

As $\Sigma_I \subseteq \Sigma_{IH}$, we can conclude by **(8)** that:

$$P_I(su_0u_1 \dots u_{k-1}h_k) = P_I(sh_0h_1 \dots h_k)$$

With **(7)**, we can now apply **Proposition 8**, **point e**, and conclude $su_0u_1 \dots u_{k-1}h_k \in \mathcal{I}$

Combining with **(9)**, we have $su_0u_1 \dots u_{k-1}h_k \in \mathcal{H} \cap \mathcal{I}$ **(10)**

We will now show that $h_k \in \Sigma_H^*.\Sigma_R.\Sigma_H^*.\Sigma_A$. It is sufficient to show that $P_I(h_k) \in \Sigma_R.\Sigma_A$.

We first note that $P_I(su_0u_1 \dots u_{k-1}) \in L_m(G_I)$ as $su_0u_1 \dots u_{k-1} \in \mathcal{I}_m$, by **(6)**. **(11)**

Similarly by **(10)**, we have $P_I(su_0u_1 \dots u_{k-1}h_k) \in L(G_I)$. **(12)**

We now note that $h_k \in (\Sigma_H \cup \Sigma_R)^*.\Sigma_A$ (by **(3)**) implies that $P_I(h_k) \in \Sigma_R^*.\Sigma_A$

We next note that DES G_I is a *command-pair interface* by **(1)**.

From **(11)**, we can conclude $P_I(su_0u_1 \dots u_{k-1}) \in \{\epsilon\} \cup (\Sigma_I^*.\Sigma_A \cap L(G_I))$ by **point D** of the *command-pair interface* definition. **(13)**

This implies that either $P_I(su_0u_1 \dots u_{k-1}) = \epsilon$ or $P_I(su_0u_1 \dots u_{k-1})$ ends in an *answer event.*

In either case, **point E** of the *command-pair interface* definition implies that $P_I(su_0u_1 \dots u_{k-1})$ can only be extended to a string in $L(G_I)$ by a *request event.* **(14)**

From **(12)**, we have $P_I(su_0u_1 \dots u_{k-1}h_k) = P_I(su_0u_1 \dots u_{k-1})P_I(h_k) \in L(G_I)$

$\Rightarrow P_I(h_k) \in \Sigma_R.\Sigma_R^*.\Sigma_A$ since $P_I(h_k) \in \Sigma_R^*.\Sigma_A$ and therefore must contain an *answer event.* By **(14)**, we know that the *answer event* must be preceded by at least one *request event.*

From **point E** of the *command-pair interface* definition, we know that, in $L(G_I)$, a *request event* must be followed by an *answer event,* before another *request event* can occur.
$\Rightarrow P_I(h_k) \in \Sigma_R.\Sigma_A$

$\Rightarrow h_k \in \Sigma_H^*.\Sigma_R.\Sigma_H^*.\Sigma_A$

We may now apply **Proposition 13** by taking $su_0u_1 \dots u_{k-1}$ to be string s, and h_k to be string h in that proposition. We thus conclude:
$(\exists u' \in \Sigma^*)$ s.t. $(su_0u_1 \dots u_{k-1}u' \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m) \ \wedge \ (P_{IH}(u') = h_k)$

We can also conclude $P_{IH}(u_0u_1 \dots u_{k-1}u') = h_0h_1 \dots h_k$ by **(6)** and the fact P_{IH} is catenative.

We now take $u_k = u'$ and we have $u_k \in \Sigma^*$ and we have it satisfying **Properties a** and **b** of **(5)**.

Inductive step complete.

We have now proven the **initial case** and the **inductive step**. We now conclude that the **Claim** is true, by induction.

We thus take $k = n$ and have $u_0, u_1, \dots, u_n \in \Sigma^*$, $su_0u_1 \dots u_n \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m$, and $P_{IH}(u_0u_1 \dots u_n) = h_0h_1 \dots h_n = h$

We thus take $u = u_0u_1 \dots u_n$, and the proof is complete.

QED

3.4.5 Proof of Proposition 15

Proof for **Proposition 15** on page 40: *If the system composed of DES G_H, G_L, and G_I is* serial level-wise nonblocking *and* serial interface consistent *with respect to the alphabet partition $\Sigma := \Sigma_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$, then*

$(\forall s \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m)(\forall h \in \Sigma_{IH}^*)$

$sh \in \mathcal{H}_m \cap \mathcal{I}_m \Rightarrow (\exists u \in \Sigma^*)$ *s.t.* $(su \in \mathcal{H}_m \cap \mathcal{L}_m \cap \mathcal{I}_m) \wedge (P_{IH}(u) = h) \wedge (P_I(u) \in \{\epsilon\} \cup \Sigma_R.\Sigma_I^*)$

Proof:

Assume system is *serial level-wise nonblocking* and *serial interface consistent.* **(1)**

Let $s \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I}_m$, $h \in \Sigma_{IH}^*$, and $sh \in \mathcal{H}_m \cap \mathcal{I}_m$ **(2)**

We will now show this implies we can construct a string u with the desired properties.

We have two cases to examine:
I) $h \in \Sigma_H^*$ (string h does not contain any *interface events*)
II) $h \notin \Sigma_H^*$ (string h contains one or more *interface events*)

Case I) $h \in \Sigma_H^*$

From **(2)**, we have $s \in \mathcal{L}$. As $h \in \Sigma_H^*$, we can conclude $P_{IL}(s) = P_{IL}(sh)$

We can thus apply **Proposition 8**, **point c**, and conclude $sh \in \mathcal{L}$

Combining with **(2)**, we thus have $sh \in \mathcal{L} \cap \mathcal{H}_m \cap \mathcal{I}_m$

We can now apply **Proposition 12** and conclude:

$$(\exists l \in \Sigma_L^*)\ shl \in \mathcal{H}_m \cap \mathcal{L}_m \cap \mathcal{I}_m$$

We take $u = hl$ and we immediately have $su \in \mathcal{H}_m \cap \mathcal{L}_m \cap \mathcal{I}_m$, $P_{IH}(u) = h$, and $P_I(u) = \epsilon \in \{\epsilon\} \cup \Sigma_R.\Sigma_I^*$

Case I complete.

Case II) $h \notin \Sigma_H^*$

This implies $P_I(h) \neq \epsilon$ **(3)**

As string h contains one or more *interface events*, events the two levels share, we must construct a string so that they will both accept the interface events, and arrive in a marked state together.

Our approach will be first to apply **Proposition 14** to enable us to construct a string u such that $su \in \mathcal{L} \cap \mathcal{H}_m \cap \mathcal{I}_m$. We will then use **Proposition 12** to show that $su \in \mathcal{H}_m \cap \mathcal{L}_m \cap \mathcal{I}_m$.

Our first step will to be to construct a string $h' \leq h$, $h' \in \Sigma_{IH}^*.\Sigma_A$, so that we can apply **Proposition 14** .

To construct a suitable h', we will start by showing that $P_I(h) \in \Sigma_R.\Sigma_I^*.\Sigma_A$

We first note that DES G_I is a *command-pair interface* by **(1)**.

From **(2)**, we have $s \in \mathcal{I}_m$

$\Rightarrow P_I(s) \in L_m(G_I)$

$\Rightarrow P_I(s) \in \{\epsilon\} \cup (\Sigma_I^*.\Sigma_A \cap L(G_I))$ by **point D** of the *command-pair interface* definition. **(4)**

This implies that either $P_I(s) = \epsilon$ or $P_I(s)$ ends in an *answer event.*

In either case, **point E** of the definition implies that $P_I(s)$ can only be extended to a string in $L(G_I)$ by a *request event.* **(5)**

Now, from **(2)** we also have $sh \in \mathcal{I}_m$.

$\Rightarrow P_I(sh) = P_I(s)P_I(h) \in L_m(G_I)$

As $P_I(h) \neq \epsilon$ (from **(3)**), we can conclude by **(5)** that $P_I(h) \in \Sigma_R.\Sigma_I^*$ **(6)**

As $P_I(s)P_I(h) \in L_m(G_I)$, we can conclude by **point D** of the *command-pair interface* definition that $P_I(s)P_I(h) \in \Sigma_I^*.\Sigma_A$

Combining with **(6)**, we can conclude $P_I(h) \in \Sigma_R.\Sigma_I^*.\Sigma_A$ **(7)**

$\Rightarrow h \in P_I^{-1}(\Sigma_R.\Sigma_I^*.\Sigma_A)$

$\Rightarrow h \in \Sigma_H^*.\Sigma_R.\Sigma_{IH}^*.\Sigma_A.\Sigma_H^*$ as $h \in \Sigma_{IH}^*$ **(8)**

$\Rightarrow (\exists h' \in \Sigma_{IH}^*.\Sigma_A)(h'' \in \Sigma_H^*)$ s.t. $h'h'' = h$ **(9)**

We can now conclude $sh' \in \mathcal{H} \cap \mathcal{I}$ as $sh \in \mathcal{H}_m \cap \mathcal{I}_m$, by **(2)** and the fact that $\mathcal{H}$ and $\mathcal{I}$ are closed languages.

We can now apply **Proposition 14** by taking h' to be string h in that proposition. We can now conclude:

$(\exists u' \in \Sigma^*)$ s.t. $(P_{IH}(u') = h') \wedge (su' \in \mathcal{H} \cap \mathcal{L} \cap \mathcal{I})$ **(10)**

We note for future use that $P_{IH}(u') = h'$ and the fact that $\Sigma_I \subseteq \Sigma_{IH}$ implies that $P_I(u') = P_I(h')$. From **(8)**, we have $h \in \Sigma_H^*.\Sigma_R.\Sigma_{IH}^*.\Sigma_A.\Sigma_H^*$. **(11)**

We can thus conclude by **(9)** that $P_I(u') \in \Sigma_R.\Sigma_I^*$ **(12)**

We will now show that $su'h'' \in \mathcal{L} \cap \mathcal{H}_m \cap \mathcal{I}_m$ so that we can apply **Proposition 12** .

From **(10)**, we have $su' \in \mathcal{L}$. From **(9)**, we have $h'' \in \Sigma_H^*$. We can thus conclude:

$P_{IL}(su'h'') = P_{IL}(su')$

We can now apply **Proposition 8**, **point c**, and conclude $su'h'' \in \mathcal{L}$ **(13)**

From **(2)**, we have $sh \in \mathcal{H}_m \cap \mathcal{I}_m$. From **(9)**, we can conclude:

$sh'h'' \in \mathcal{H}_m \cap \mathcal{I}_m$ **(14)**

As $P_{IH}(u') = h'$ (from **(10)**), we have:

$P_{IH}(sh'h'') = P_{IH}(s)P_{IH}(h')P_{IH}(h'') = P_{IH}(s)P_{IH}(u')P_{IH}(h'') = P_{IH}(su'h'')$.

We can now apply **Proposition 8**, **point b**, and conclude $su'h'' \in \mathcal{H}_m$ **(15)**

From **(11)**, we have $P_I(u') = P_I(h')$. We can argue as above and conclude $P_I(sh'h'') = P_I(su'h'')$

We can now apply **Proposition 8**, **point f**, and conclude $su'h'' \in \mathcal{I}_m$

Combining with **(13)** and**(15)**, we can conclude:

$su'h'' \in \mathcal{L} \cap \mathcal{H}_m \cap \mathcal{I}_m$

We can now apply **Proposition 12** by taking $su'h''$ to be string s in that proposition. We thus conclude:
$(\exists l \in \Sigma_L^*)$ s.t. $su'h''l \in \mathcal{H}_m \cap \mathcal{L}_m \cap \mathcal{I}_m$ **(16)**

We thus take $u = u'h''l$ and we have $P_{IH}(u) = P_{IH}(u'h''l) = h'h''\epsilon = h$ and $su \in \mathcal{H}_m \cap \mathcal{L}_m \cap \mathcal{I}_m$.

From **(12)**, we have $P_I(u') \in \Sigma_R.\Sigma_I^*$. We thus have:

$P_I(u) = P_I(u'h''l) = P_I(u') \in \Sigma_R.\Sigma_I^*$, as $h'' \in \Sigma_H^*$ (by **(9)**) and $l \in \Sigma_L^*$ (by **(16)**)

Case II is now complete.

By **Cases I** and **II**, we now have constructed a string $u \in \Sigma^*$ such that $P_I(u) \in \{\epsilon\} \cup \Sigma_R.\Sigma_I^*$, $P_{IH}(u) = h$ and $su \in \mathcal{H}_m \cap \mathcal{L}_m \cap \mathcal{I}_m$, as required.

QED

Chapter 4

Serial Case: Controllability

Now that we have discussed nonblocking in the serial interface setting, we now consider controllability. In the remainder of this chapter, we will define our setting and notation and then present some supporting propositions, followed by the serial controllability theorem.

4.1 Definitions and Notation

We now present some definitions and notation that will be useful in simplifying proofs. When we discussed nonblocking, we were only concerned with the *high* and *low level subsystems,* ignoring distinctions between plants and supervisors. For controllability, we need to split the subsystems into their plant and supervisor components. We will do so as shown in Figure 4.1. We define the *high level plant* to be DES $\mathcal{G}_H$, and the *high level supervisor* to be $\mathcal{S}_H$ (both defined over event set Σ_{IH}). Similarly, the *low level plant* and *supervisor* are $\mathcal{G}_L$ and $\mathcal{S}_L$ (defined over event set Σ_{IL}). To be consistent with our definitions in Chapter 3, we define the following identities for the *high* and *low level subsystems* as follows:

$$G_H := \mathcal{G}_H||_s\mathcal{S}_H \qquad\qquad G_L := \mathcal{G}_L||_s\mathcal{S}_L$$

We now have two ways to describe our system for the serial case, depending on the level of detail required. We will call the original method described in Chapter 3 in terms of an *interface* and *high* and *low subsystems*, the *serial subsystem based form.* This form is useful as it simplifies nonblocking definitions and proofs. We call the above method, given in terms of an interface and plants and supervisors, the *serial general form* as the *serial*

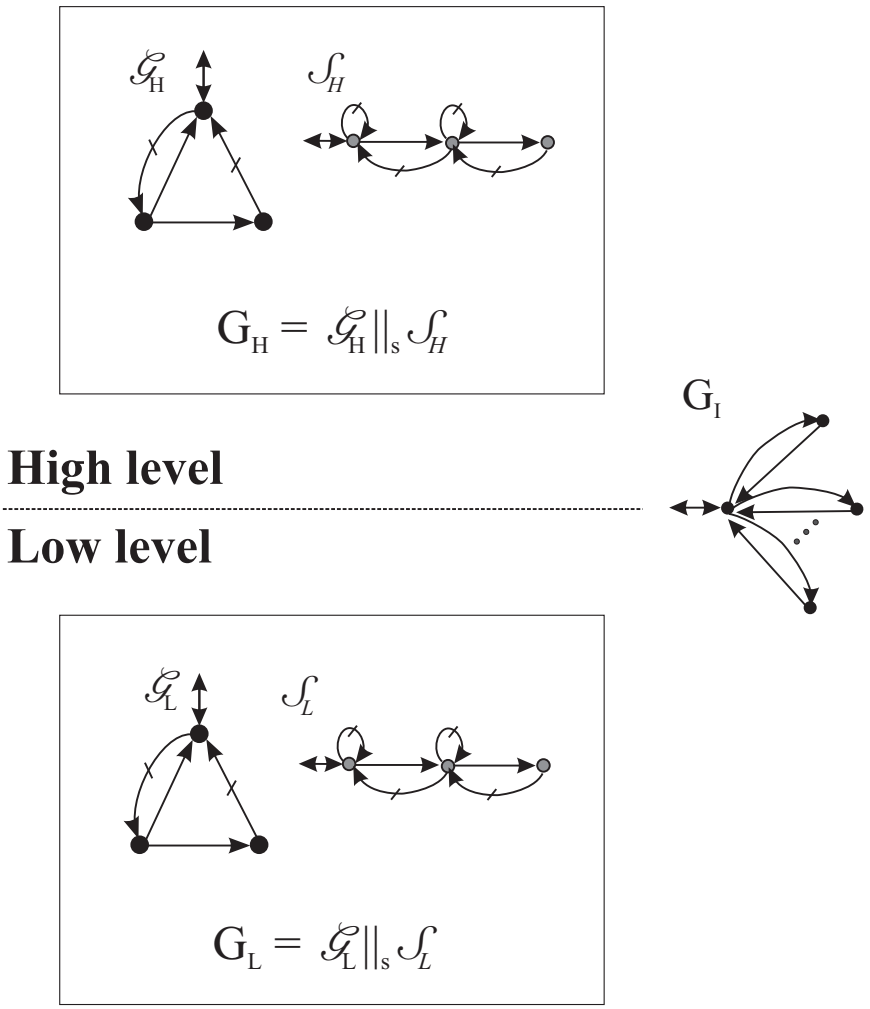


Figure 4.1: Plant and Supervisor Subplant Decomposition

subsystem based form can be recovered by applying the above identities for G_H and G_L. When we refer terms applicable to both forms (e.g. the *high level*), we will simply state the term, allowing the type of the system to make our meaning clear.

Our next step is to define the *flat supervisor* and *plant* for our system. By *flat supervisor* and *plant*, we refer to the equivalent DES supervisor and plant that would represent our system if we ignored the interface structure. They are defined as follows:

$$\textbf{Plant} := \mathcal{G}_H||_s\mathcal{G}_L \qquad\qquad \textbf{Sup} := \mathcal{S}_H||_s\mathcal{S}_L||_sG_I$$

In the above definition, we have taken the interface, G_I, as a supervisor. This is to

recognise the fact that G_I is a specification of services that the *low level* is to provide to the *high level.* As such, low level supervisors are usually required to implement these services. By treating G_I as a supervisor, we can verify to make sure that the desired pattern of *request* and *answer events* is achievable.

We next want to express the languages of **Plant** and **Sup** in terms of their components. To do this, we need to first define the following useful languages:

$$\mathbf{H} := P_{IH}^{-1}L(\mathcal{G}_H), \qquad \mathbf{H}_\mathcal{S} := P_{IH}^{-1}L(\mathcal{S}_H), \qquad \subseteq \Sigma^*$$
$$\mathbf{L} := P_{IL}^{-1}L(\mathcal{G}_L), \qquad \mathbf{L}_\mathcal{S} := P_{IL}^{-1}L(\mathcal{S}_L), \qquad \subseteq \Sigma^*$$

We can now express the languages of **Plant** and **Sup** as follows:

$$L(\mathbf{Plant}) = \mathbf{H} \cap \mathbf{L} \qquad\qquad L(\mathbf{Sup}) = \mathbf{H}_\mathcal{S} \cap \mathbf{L}_\mathcal{S} \cap \mathcal{I}$$

This allows us to present the proposition below that collects together several similar propositions. As it will be common in the proofs in this report to show that membership in languages such as **H** is dependent only on events in specific subsets (for **H**, events in subset Σ_{IH}), this proposition will be very useful.

Proposition 16

(a) $(\forall s, s' \in \Sigma^*)\ s \in \mathbf{H}$ *and* $P_{IH}(s) = P_{IH}(s') \Rightarrow s' \in \mathbf{H}$

(b) $(\forall s, s' \in \Sigma^*)\ s \in \mathbf{H}_\mathcal{S}$ *and* $P_{IH}(s) = P_{IH}(s') \Rightarrow s' \in \mathbf{H}_\mathcal{S}$

(c) $(\forall s, s' \in \Sigma^*)\ s \in \mathbf{L}$ *and* $P_{IL}(s) = P_{IL}(s') \Rightarrow s' \in \mathbf{L}$

(d) $(\forall s, s' \in \Sigma^*)\ s \in \mathbf{L}_\mathcal{S}$ *and* $P_{IL}(s) = P_{IL}(s') \Rightarrow s' \in \mathbf{L}_\mathcal{S}$

Proof:

Points a-d:

Identical to the proof of **point a** of **Proposition 8**, after substitution.

QED

4.2 Serial Level-wise Controllability

The goal in this chapter is to develop a means of verifying, using only local checks, that our system's *flat supervisor* is controllable for the *flat plant.* To do this, we will use the *serial level-wise controllable* definition.

Serial Level-wise Controllable: The system composed of plant components $\mathcal{G}_H$, $\mathcal{G}_L$, supervisors $\mathcal{S}_H$, $\mathcal{S}_L$, and interface 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 $\mathcal{G}_H$ and $\mathcal{S}_H$ is Σ_{IH}, the alphabet of $\mathcal{G}_L$ and $\mathcal{S}_L$ is Σ_{IL}, and the alphabet of G_I is Σ_I

(II) $(\forall s \in \mathbf{L} \cap \mathbf{L}_{\mathcal{S}} \cap \mathcal{I}) \quad \text{Elig}_{\mathbf{L}}(s) \cap \Sigma_u \subseteq \text{Elig}_{\mathbf{L}_{\mathcal{S}} \cap \mathcal{I}}(s)$ *Controllability at the low level*

(III) $(\forall s \in \mathbf{H} \cap \mathcal{I} \cap \mathbf{H}_{\mathcal{S}}) \quad \text{Elig}_{\mathbf{H} \cap \mathcal{I}}(s) \cap \Sigma_u \subseteq \text{Elig}_{\mathbf{H}_{\mathcal{S}}}(s)$ *Controllability at the high level*

To summarise the definition, **Point I** simply states that the plants, supervisors, and interface have the indicated event sets. This is in essence restricting the control actions allowable by the supervisors to their specified alphabets. For example, this implies that $\mathcal{S}_H$ is forbidden to disable any low level events.

The next point states that the *interface* and $\mathcal{S}_L$ are together controllable for the low level plant $\mathcal{G}_L$. In other words, we are treating the *low level* as a self contained system and performing a standard controllability test for the modular supervisor $\mathcal{S}_L \wedge G_I$ with respect to the plant $\mathcal{G}_L$.

The last point states that supervisor $\mathcal{S}_H$ is controllable for the high level plant $\mathcal{G}_H$, when it is already under the control of the *interface.* In other words, we are treating the *high level* as a self contained system and performing a standard controllability test for supervisor $\mathcal{S}_H$ with respect to the composite plant $\mathcal{G}_H||_s G_I$. By treating the *interface* as a plant at the *high level*, we allow the high level supervisor, $\mathcal{S}_H$, to be more flexible as the *interface* may have more information about when interface events are eligible than the high level plant.

We are now ready to state the proposition below which establishes useful properties for often used languages.

Proposition 17 *If the system composed of plant components $\mathcal{G}_H$, $\mathcal{G}_L$, supervisors $\mathcal{S}_H$, $\mathcal{S}_L$, and interface G_I, is* serial level-wise controllable *with respect to the alphabet partition $\Sigma := \Sigma_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$, then languages* $\mathbf{H}$, $\mathbf{H}_\mathcal{S}$, $\mathbf{L}$, $\mathbf{L}_\mathcal{S}$, $\mathcal{I}$, $L(\mathbf{Plant})$, *and* $L(\mathbf{Sup})$ *are closed.*

Proof: See page 61.

4.3 Propositions and Theorem

We are now ready to present our main results for this chapter. We will first present two supporting propositions, followed by our serial case controllability theorem.

4.3.1 Low level Controllability Proposition

Our first proposition asserts that if the system is *serial level-wise controllable*, then G_I and $\mathcal{S}_L$ are together controllable for the system's *flat plant.*

Proposition 18 *If the system composed of plant components $\mathcal{G}_H$, $\mathcal{G}_L$, supervisors $\mathcal{S}_H$, $\mathcal{S}_L$, and interface G_I, is* serial level-wise controllable *with respect to the alphabet partition $\Sigma := \Sigma_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$, then:*

$$(\forall s \in L(\mathbf{Plant}) \cap \mathbf{L}_\mathcal{S} \cap \mathcal{I}) \quad Elig_{L(\mathbf{Plant})}(s) \cap \Sigma_u \subseteq Elig_{\mathbf{L}_\mathcal{S} \cap \mathcal{I}}(s)$$

where $\mathbf{Plant} = \mathcal{G}_H ||_s \mathcal{G}_L$

Proof: See page 62.

4.3.2 High Level Controllability Proposition

The last proposition asserts that if the system is *serial level-wise controllable*, then $\mathcal{S}_H$ is controllable for our *flat plant* when it is already under the control of the *interface.*

Proposition 19 *If the system composed of plant components $\mathcal{G}_H$, $\mathcal{G}_L$, supervisors $\mathcal{S}_H$, $\mathcal{S}_L$, and interface G_I, is* serial level-wise controllable *with respect to the alphabet partition $\Sigma := \Sigma_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$, then:*

$$(\forall s \in L(\mathbf{Plant}) \cap \mathcal{I} \cap \mathbf{H}_{\mathcal{S}}) \quad Elig_{L(\mathbf{Plant})\cap\mathcal{I}}(s) \cap \Sigma_u \subseteq Elig_{\mathbf{H}_{\mathcal{S}}}(s)$$

where $\mathbf{Plant} = \mathcal{G}_H||_s\mathcal{G}_L$

Proof: See page 62.

4.3.3 Serial Controllability Theorem

We now present our main result for this chapter, the *serial controllability theorem.* In essence, this theorem asserts that if the system is *serial level-wise controllable*, then controllability can be checked for each level separately in order to determine that the system's *flat supervisor* is controllable for the system's *flat plant.* As the *serial level-wise controllable* definition can be evaluated by examining only one level of our system at a time, we now have a means to verify controllability of our system using local checks.

Theorem 2 *If the system composed of plant components* $\mathcal{G}_H$*,* $\mathcal{G}_L$*, supervisors* $\mathcal{S}_H$*,* $\mathcal{S}_L$*, and interface* G_I*, is* serial level-wise controllable *with respect to the alphabet partition* $\Sigma := \Sigma_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$*, then:*

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

where $\mathbf{Plant} = \mathcal{G}_H||_s\mathcal{G}_L$ *and* $\mathbf{Sup} = \mathcal{S}_H||_s\mathcal{S}_L||_sG_I$*.*

Proof:

Assume system is *serial level-wise controllable* **(1)**

Let $s \in L(\mathbf{Plant})\cap L(\mathbf{Sup})$ and $\sigma \in \text{Elig}_{L(\mathbf{Plant})}(s)\cap\Sigma_u$ **(2)**

Will now show this implies $\sigma \in \text{Elig}_{L(\mathbf{Sup})}(s)$

From **(2)**, we have $s \in L(\mathbf{Plant})\cap L(\mathbf{Sup}) = \mathbf{H}\cap\mathbf{L}\cap\mathbf{H}_{\mathcal{S}}\cap\mathbf{L}_{\mathcal{S}}\cap\mathcal{I}$ **(3)**

We also have $s\sigma \in L(\mathbf{Plant}) = \mathbf{H} \cap \mathbf{L}$ **(4)**

As $L(\mathbf{Sup}) = \mathbf{H}_{\mathcal{S}} \cap \mathbf{L}_{\mathcal{S}} \cap \mathcal{I}$, it is sufficient to show that $s\sigma \in \mathbf{H}_{\mathcal{S}} \cap \mathbf{L}_{\mathcal{S}} \cap \mathcal{I}$

From **(3)**, we have $s \in \mathbf{H} \cap \mathbf{L} \cap \mathbf{L}_{\mathcal{S}} \cap \mathcal{I} = L(\mathbf{Plant}) \cap \mathbf{L}_{\mathcal{S}} \cap \mathcal{I}$

From **(2)**, we have $\sigma \in \text{Elig}_{L(\mathbf{Plant})}(s) \cap \Sigma_u$. We can thus conclude by **Proposition 18**

that $\sigma \in \text{Elig}_{\mathbf{L}_\mathcal{S} \cap \mathcal{I}}(s)$

$\Rightarrow s\sigma \in \mathbf{L}_\mathcal{S} \cap \mathcal{I}$ **(5)**

All that remains is to show that $s\sigma \in \mathbf{H}_\mathcal{S}$

Using **(4)** and **(5)**, we have $s\sigma \in \mathbf{H} \cap \mathbf{L} \cap \mathcal{I}$

$\Rightarrow s\sigma \in L(\textbf{Plant}) \cap \mathcal{I}$

$\Rightarrow \sigma \in \text{Elig}_{L(\textbf{Plant}) \cap \mathcal{I}}(s) \cap \Sigma_u$

From **(5)**, we have $s \in L(\textbf{Plant}) \cap \mathcal{I} \cap \mathbf{H}_\mathcal{S}$

We can thus conclude by **Proposition 19** that $\sigma \in \text{Elig}_{\mathbf{H}_\mathcal{S}}(s)$

$\Rightarrow s\sigma \in \mathbf{H}_\mathcal{S}$

Combining with **(5)**, we have $s\sigma \in \mathbf{H}_\mathcal{S} \cap \mathbf{L}_\mathcal{S} \cap \mathcal{I}$, as required.

QED

4.3.4 Software Tool

To aid in investigating *hierarchical interface-based supervisory control,* we have developed software routines to verify that a system satisfies the conditions below. The routines were developed by Leduc during his collaboration with Siemens Corporate Research and they use the algorithms presented in Chapter 6. The routines are an experimental add-on to Siemen's Valid software program.

- DES G_I satisfies the *star interface* definition.
- *Serial level-wise nonblocking*
- *Serial level-wise controllable*
- *Serial interface consistent,* using the *serial interface strict marking* condition to check for **Property 6**.

4.4 Proofs of Selected Propositions

In order to make this work more readable, the proofs of some propositions in this chapter were not given as the propositions were introduced. They will now be presented in the

following sections.

4.4.1 Proof of Proposition 17

Proof for **Proposition 17** on page 58: If the system composed of plant components $\mathcal{G}_H$, $\mathcal{G}_L$, supervisors $\mathcal{S}_H$, $\mathcal{S}_L$, and interface G_I, is *serial level-wise controllable* with respect to the alphabet partition $\Sigma := \Sigma_H \dot{\cup} \Sigma_L \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A$, then languages $\mathbf{H}$, $\mathbf{H}_\mathcal{S}$, $\mathbf{L}$, $\mathbf{L}_\mathcal{S}$, $\mathcal{I}$, $L(\mathbf{Plant})$, and $L(\mathbf{Sup})$ are closed.

Proof:

Assume system is *serial level-wise controllable.* **(1)**

Will now show this implies that the indicated languages are closed.

We first note that by **(1)**, the system is *serial level-wise controllable.* By **Point 1** of this definition, we can conclude:

$$L(\mathcal{G}_H), L(\mathcal{S}_H) \subseteq \Sigma_{IH}^*,\ L(\mathcal{G}_L), L(\mathcal{S}_L) \subseteq \Sigma_{IL}^*, \text{ and } L(G_I) \subseteq \Sigma_I^*.$$

This tells us that languages $\mathbf{H} = P_{IH}^{-1}(L(\mathcal{G}_H))$, $\mathbf{H}_\mathcal{S} = P_{IH}^{-1}L(\mathcal{S}_H)$, $\mathbf{L} = P_{IL}^{-1}(L(\mathcal{G}_L))$, $\mathbf{L}_\mathcal{S} = P_{IL}^{-1}L(\mathcal{S}_L)$, $\mathcal{I} = P_I^{-1}(L(G_I))$ are defined.

We will start by showing that languages $\mathbf{H}$, $\mathbf{H}_\mathcal{S}$, $\mathbf{L}$, $\mathbf{L}_\mathcal{S}$, and $\mathcal{I}$ are closed.

We now note that languages $L(\mathcal{G}_H)$, $L(\mathcal{S}_H)$, $L(\mathcal{G}_L)$, $L(\mathcal{S}_L)$, and $L(G_I)$ are closed by the definition of the closed behaviour of a DES.

We can now apply **Proposition 1** repeatedly and conclude that $\mathbf{H}$, $\mathbf{H}_\mathcal{S}$, $\mathbf{L}$, $\mathbf{L}_\mathcal{S}$, and $\mathcal{I}$ are closed, as required. **(2)**

We will now show that languages $L(\mathbf{Plant})$, and $L(\mathbf{Sup})$ are closed.

We now note that $L(\mathbf{Plant}) = \mathbf{H} \cap \mathbf{L}$ and $L(\mathbf{Sup}) = \mathbf{H}_\mathcal{S} \cap \mathbf{L}_\mathcal{S} \cap \mathcal{I}$.

Combining with **(2)**, we can now apply **Proposition 2** repeatedly and conclude that $L(\mathbf{Plant})$, and $L(\mathbf{Sup})$ are closed, as required.

QED

4.4.2 Proof of Proposition 18

Proof for **Proposition 18** on page 58: *If the system composed of plant components $\mathcal{G}_H$, $\mathcal{G}_L$, supervisors $\mathcal{S}_H$, $\mathcal{S}_L$, and interface G_I, is* serial level-wise controllable *with respect to the alphabet partition $\Sigma := \Sigma_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$, then:*

$$(\forall s \in L(\mathbf{Plant}) \cap \mathbf{L}_\mathcal{S} \cap \mathcal{I}) \quad \mathit{Elig}_{L(\mathbf{Plant})}(s) \cap \Sigma_u \subseteq \mathit{Elig}_{\mathbf{L}_\mathcal{S} \cap \mathcal{I}}(s)$$

where $\mathbf{Plant} = \mathcal{G}_H ||_s \mathcal{G}_L$

Proof:

Assume system is *serial level-wise controllable* **(1)**

Let $s \in L(\mathbf{Plant}) \cap \mathbf{L}_\mathcal{S} \cap \mathcal{I}$ and $\sigma \in \mathrm{Elig}_{L(\mathbf{Plant})}(s) \cap \Sigma_u$ **(2)**

Will now show this implies $\sigma \in \mathrm{Elig}_{\mathbf{L}_\mathcal{S} \cap \mathcal{I}}(s)$

From **(2)**, we have $s, s\sigma \in L(\mathbf{Plant}) = \mathbf{H} \cap \mathbf{L}$

Also using **(2)**, we can now conclude $s \in \mathbf{L} \cap \mathbf{L}_\mathcal{S} \cap \mathcal{I}$ and $\sigma \in \mathrm{Elig}_{\mathbf{L}}(s) \cap \Sigma_u$

Using **(1)**, we can conclude by **Point II** of the *serial level-wise controllable* definition that $\sigma \in \mathrm{Elig}_{\mathbf{L}_\mathcal{S} \cap \mathcal{I}}(s)$, as required.

QED

4.4.3 Proof of Proposition 19

Proof for **Proposition 19** on page 58: *If the system composed of plant components $\mathcal{G}_H$, $\mathcal{G}_L$, supervisors $\mathcal{S}_H$, $\mathcal{S}_L$, and interface G_I, is* serial level-wise controllable *with respect to the alphabet partition $\Sigma := \Sigma_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$, then:*

$$(\forall s \in L(\mathbf{Plant}) \cap \mathcal{I} \cap \mathbf{H}_\mathcal{S}) \quad \mathit{Elig}_{L(\mathbf{Plant}) \cap \mathcal{I}}(s) \cap \Sigma_u \subseteq \mathit{Elig}_{\mathbf{H}_\mathcal{S}}(s)$$

Proof:

Assume system is *serial level-wise controllable* **(1)**

Let $s \in L(\mathbf{Plant}) \cap \mathcal{I} \cap \mathbf{H}_\mathcal{S}$ and $\sigma \in \mathrm{Elig}_{L(\mathbf{Plant}) \cap \mathcal{I}}(s) \cap \Sigma_u$ **(2)**

Will now show this implies $\sigma \in \text{Elig}_{\mathbf{H}_\mathcal{S}}(s)$

From **(2)**, we have $s, s\sigma \in L(\mathbf{Plant}) \cap \mathcal{I} = \mathbf{H} \cap \mathbf{L} \cap \mathcal{I}$

Also using **(2)**, we can now conclude $s \in \mathbf{H} \cap \mathcal{I} \cap \mathbf{H}_\mathcal{S}$ and $\sigma \in \text{Elig}_{\mathbf{H} \cap \mathcal{I}} \cap \Sigma_u$

Using **(1)**, we can conclude by **Point III** of the *serial level-wise controllable* definition that $\sigma \in \text{Elig}_{\mathbf{H}_\mathcal{S}}(s)$, as required.

QED

Chapter 5

Simple Manufacturing Example

We now present a simple manufacturing example to illustrate the method for the serial case. The example presented was inspired in part by the examples given in Wang [57], and in Brandin [7]. Table 5.1 defines abbreviations used for the event labels.

Abbrev.	Meaning	Abbrev.	Meaning
pt	part (item)	str	start
cmpl	complete	attch	attach
fin	finish	ent	enter
rlse	release	lv	leave
pol	polish	recog	recognize
arr	arrive		

Table 5.1: Abbreviations Used in Event Labels

In the following sections, we will describe our problem setting, and then present the original plant components. We will then assign them to a particular level of our hierarchy, augmenting if necessary the low level plant models so that they work better with an interface. We will then define the interface, supervisors, and finally we will present the complete system. We will conclude by demonstrating that the *flat system* is nonblocking and that the *flat supervisor* is controllable for the *flat plant.*

5.1 Description of Manufacturing Unit

As shown in Figure 5.1, the manufacturing unit is composed of three cells connected by a conveyor belt. In front of each cell, is a part acquisition unit that automatically stops a part and holds it until it is given a release command. Parts enter the system at the far left

and exit at the far right. After the item exits the conveyor system, it goes to a packaging machine.

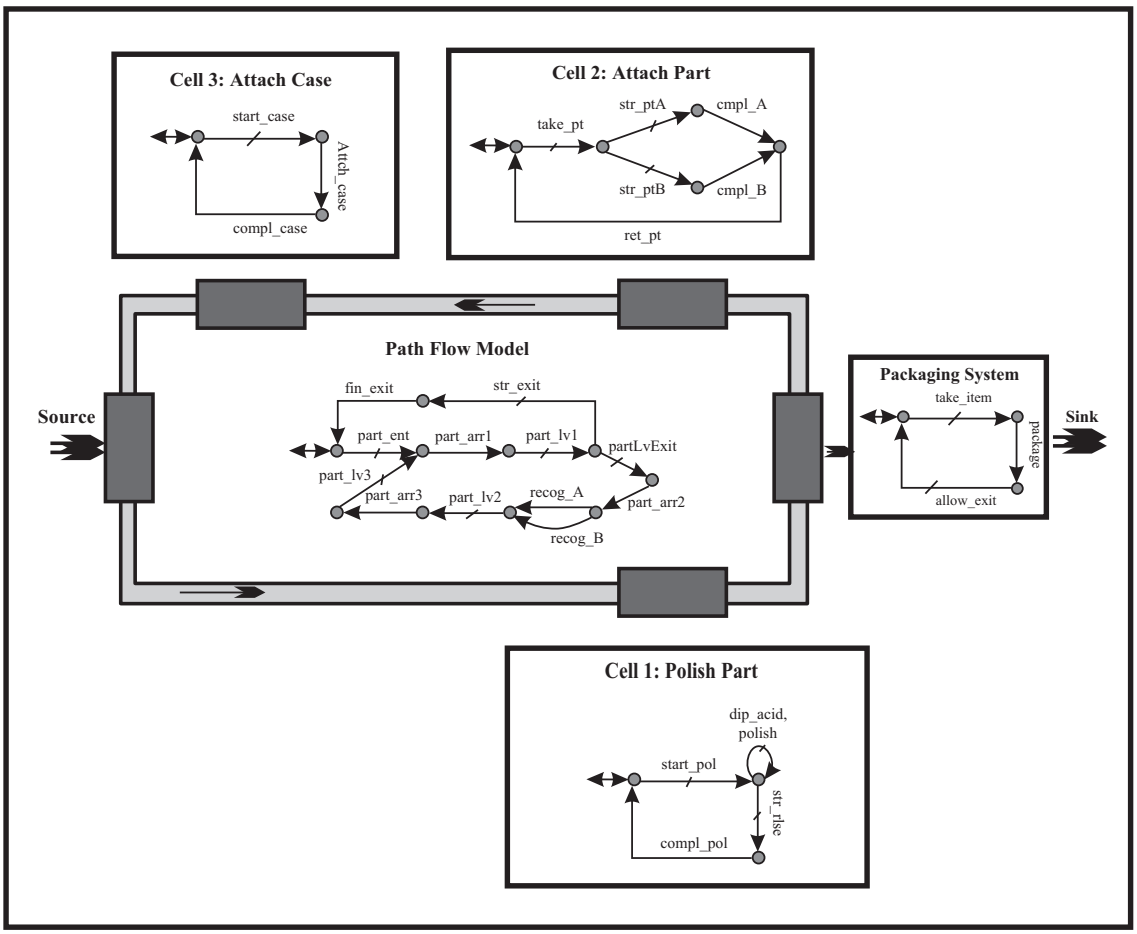


Figure 5.1: Block Diagram of Plant

The diagram shows a flat view of the plant (the supervisors will be added later). We see the plant models for cell one (polishes part), cell two (attaches part of type A or type B to the assembly of what's being built), cell three (attach case to assembly), and the path flow model that show how parts enter the system, travel around the belt, and finally leave the system. Of note in the path flow model are the events *recog_A,* and *recog_B*. The acquisition unit in front of cell two is capable of recognising if a part is of type A or type B. On the far right, we see the model for the packaging system.

5.1.1 Defining Infrastructure

The first step in the process is to decide which plant models belong to the *high level subsystem,* and which to the *low level subsystem.* The division we have chosen can be seen in Figure 5.2.

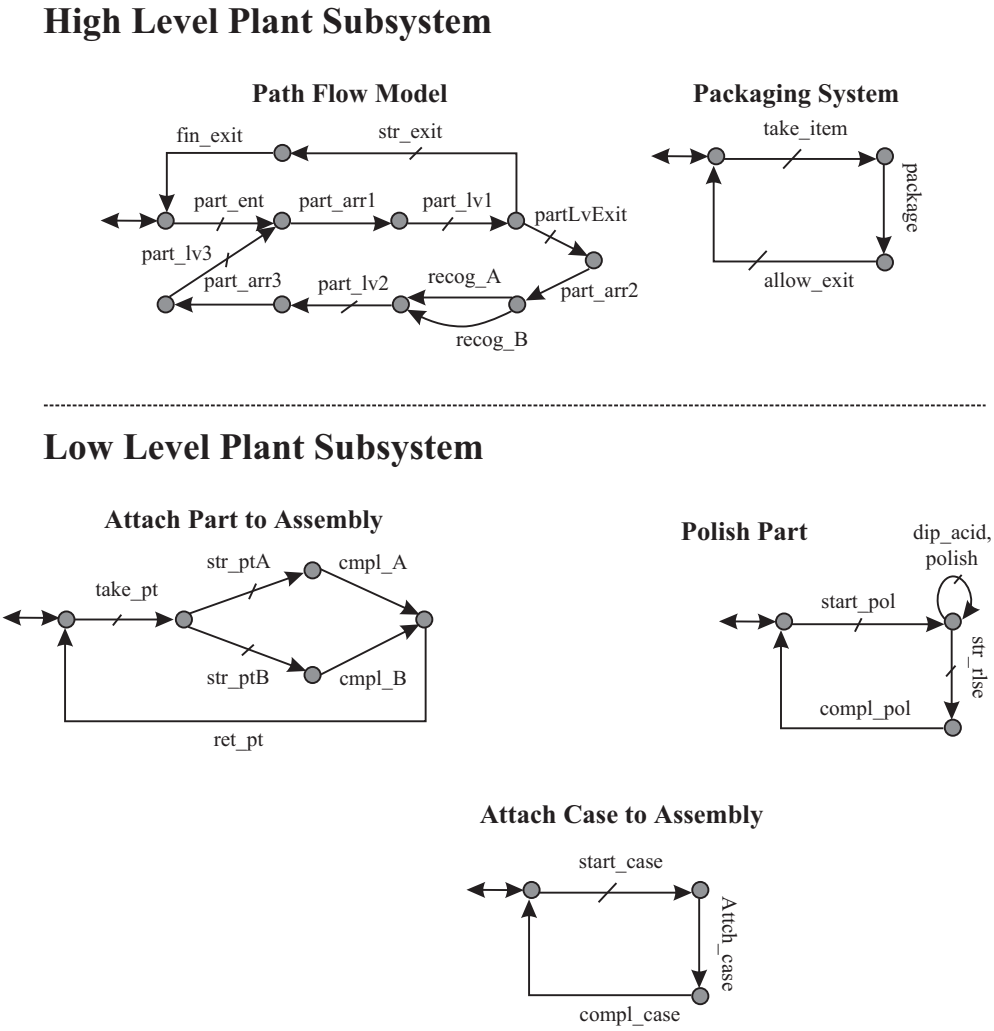


Figure 5.2: Original Plant

We now note that the model for cell two is not well suited to being accessed through an interface. It requires that the decision to attach part A or part B be made after event *take_pt* occurs. To make this functionality available to the upper level, we augment the model by adding the DES **Define New Events** shown in Figure 5.3. The new request

events (*attch_ptA* and *attch_ptB*) will provide the high level with an easy selection method while the new finish events (*finA_attch* and *finB_attch*) will inform the high level of the completion of their respective tasks.

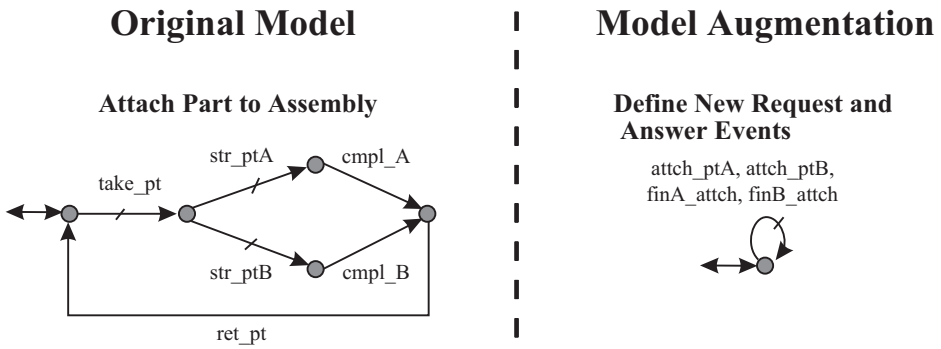


Figure 5.3: Augmenting Low Level Plant

We are now ready to define our *interface.* Figure 5.4 shows the interface DES, G_I. From the diagram, we can see which events are *request events* and which events are *answer events.*

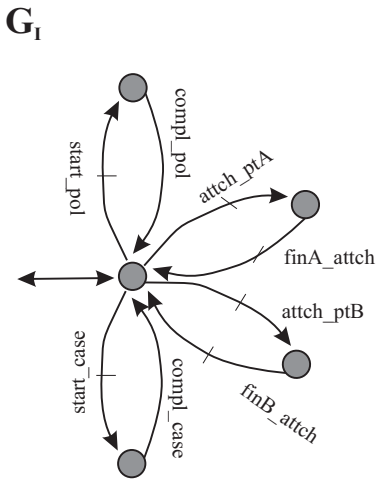


Figure 5.4: Interface Definition

We next define the alphabet partition $\Sigma := \Sigma_H \dot{\cup} \Sigma_L \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A$ as follows:

$$\Sigma_R = \{start\_pol,\ attch\_ptA,\ attch\_ptB,\ start\_case\}$$

$$\Sigma_A = \{comp\_pol,\ finA\_attch,\ finB\_attch,\ compl\_case\}$$

$$
\begin{aligned}
\Sigma_H &= \{part\_ent,\ part\_arr1,\ part\_lv1,\ partLvExit,\\
&\quad str\_exit,\ fin\_exit,\ part\_arr2,\ recog\_A,\ recog\_B,\\
&\quad part\_lv2,\ part\_arr3,\ part\_lv3,\ take\_item,\\
&\quad allow\_exit,\ package\}\\
\Sigma_L &= \{take\_pt,\ str\_ptA,\ str\_ptB,\ compl\_A,\ compl\_B,\\
&\quad ret\_pt,\ dip\_acid,\ polish,\ str\_rlse,\ attch\_case\}
\end{aligned}
$$

5.2 Designing Supervisors

Now that we have defined our *interface,* we are ready to design the low level supervisors that will provide the functionality for the *request events,* and give meaning to the *answer events.* The idea is for the *low level* to offer well-defined "services" to the *high level.*

We start with cell one. Here we want the sequence *dip_acid-polish* to be repeated twice, after a *start_pol* event occurs. The supervisor is shown in Figure 5.5, and is labelled **Polishing Sequence**. For cell two, we have to provide supervisors so that the cell reacts appropriately when events *attch_ptA* and *attch_ptB* occur. We also must guarantee that *answer events finA_attch* and *finB_attch* only occur when they have the appropriate meaning. The DES **Affix Part** in Figure 5.5 shows how this is done. Finally, we do nothing for cell three as it is so simple, its functionality being already present.

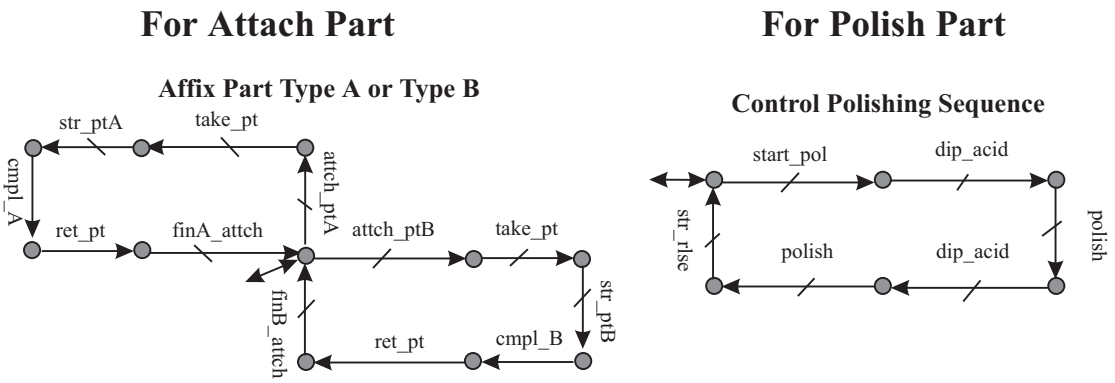


Figure 5.5: Supervisors to Support Interface

Now that the low level functionality is taken care of, we will design high level supervisors

that use the *interface.* Figure 5.6 shows a supervisor (**Sequence Tasks**) that allows a part to visit each cell, executes the appropriate command for the cell and part type, and then allows the part to leave the conveyor system. The figure also shows a supervisor (**Exit Buffer**) that implements a two item buffer for the packaging system. Finally, we note that the above supervisors were designed by hand, but we could have also employed synthesis methods.

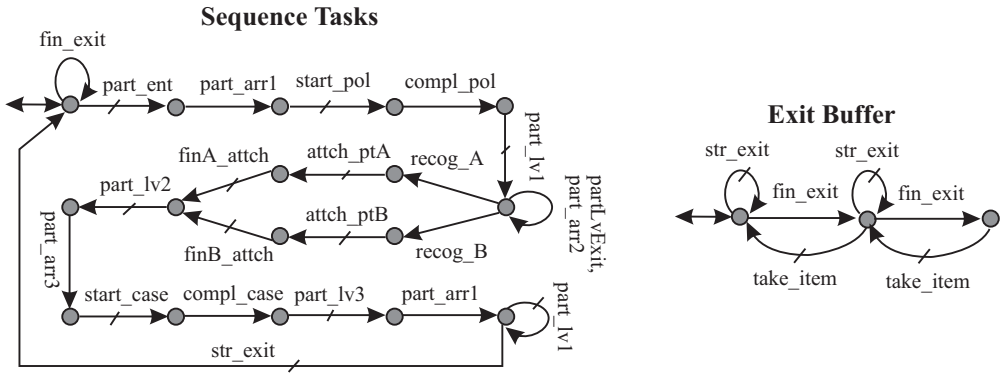


Figure 5.6: High Level Supervisors

5.3 The Final System

With the system components defined, it is time to put them together. Figure 5.7 shows our *high level subsystem*, plant, and supervisor, DES G_H, $\mathcal{G}_H$, and $\mathcal{S}_H$. We also have our *low level subsystem*, plant, and supervisor, DES G_L, $\mathcal{G}_L$, and $\mathcal{S}_L$. They are defined to be the synchronous product of the indicated automata.

We now want to determine whether the *flat system* is nonblocking. For this, we used our software tool to verify that the system is *serial interface consistent*, and *serial level-wise nonblocking.* We can thus conclude by **Theorem 1** that the *flat system* is nonblocking.

Next, we want to show that the *flat supervisor* is controllable for the *flat plant.* For this, we used our software tool to verify that the system is *serial level-wise controllable.* We can thus conclude by **Theorem 2** that the *flat supervisor* is controllable for the *flat plant.*

5.4 Concurrency of Subsystems

Before concluding this example, we comment on the inherent concurrency of the *high* and *low levels.* Unlike state expansion methods such as Wang [57] and Gohari [23] that expand

a high level state into a group of low level states, the interface method is based on the *synchronous product*, limiting information flow, and a set of consistency rules. In general, there is no one-to-one association between a high level state and a set of low level states. This allows the *high level* to remain active while the *low level* is active, and thus operate concurrently. In the cited state expansion methods, the high level state would remain fixed while the low level becomes active.

This concurrency can be seen in the current example, by noting that once the event *fin_exit* has occurred, the string shown below is then possible.

part_ent part_arr1 start_pol dip_acid take_item polish

The string clearly shows how the *high level* event *take_item* can occur in the middle of a sequence of *low level* events, thus demonstrating that both levels are active.

5.5 Design of Supervisors in Thesis

For the examples in this thesis, all supervisors are designed for their level as modular supervisors. The supervisors are designed by hand to meet the given specifications, and then verified that they are locally controllable and they don't cause the local plant to block (ie. for the serial case, they satisfy their portion of the *serial level-wise nonblocking* and *serial level-wise controllable* definitions). If they are not, they are modified until they are controllable and nonblocking. If a subsystem fails to satisfy its portion of the interface properties, then it is modified until it does satisfy them. The details of this process for specific examples will not be given for reasons of brevity; only the final results are presented.

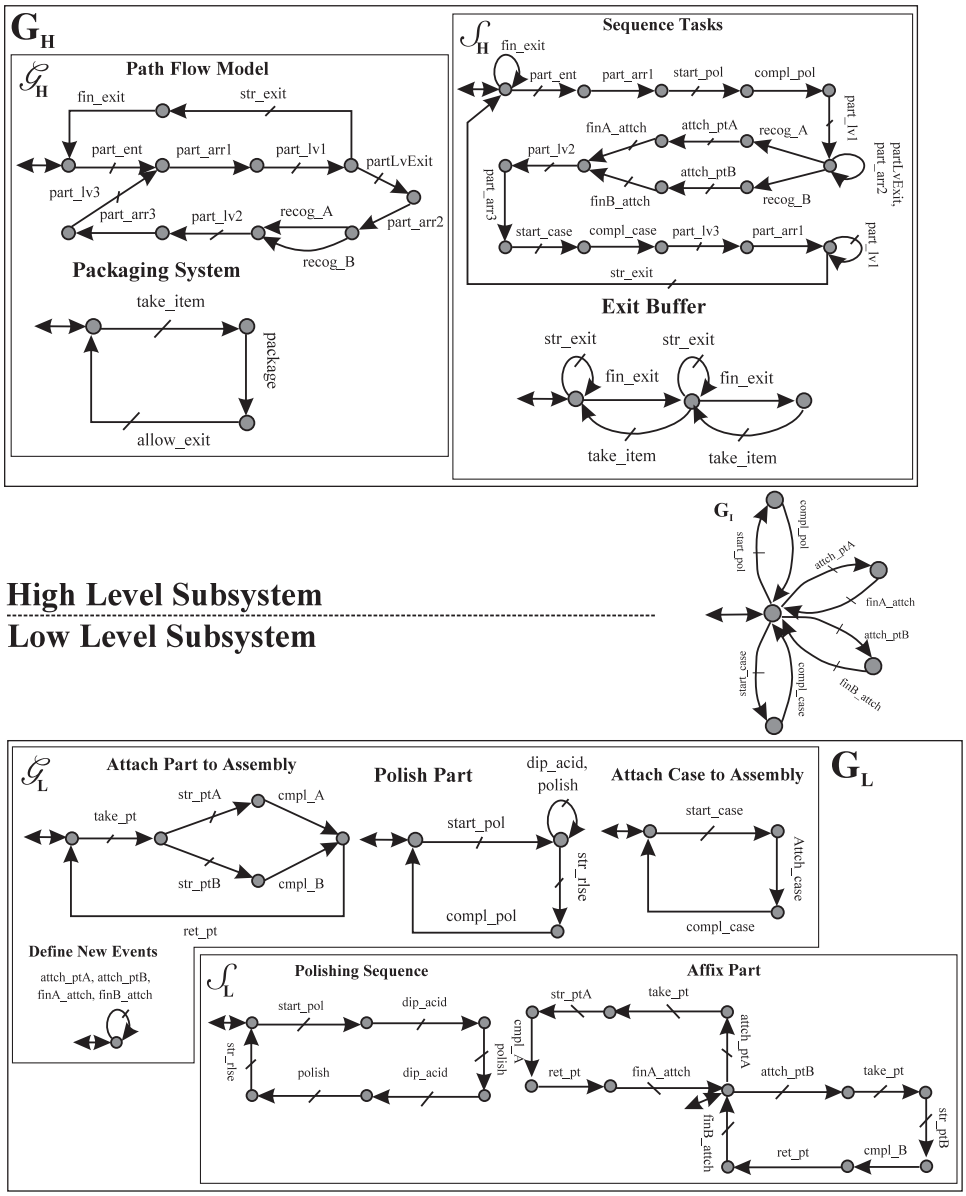


Figure 5.7: Complete System Definition

Chapter 6

Serial Case Algorithms

In this chapter, we provide evaluation methods for definitions *star interface, serial level-wise nonblocking, serial level-wise controllable,* and *serial interface consistent.* Our purpose here is to provide enough details that the definitions can be evaluated, but skipping over aspects that are straightforward or have been investigated elsewhere (e.g. controllability algorithms). We will be discussing a naive "proof of concept" algorithm (horribly inefficient but easy to construct based on existing algorithms), and we will leave the investigation of detailed efficient algorithms for later work. The reason for this is partly time constraints but primarily because the strength of our method doesn't depend on the individual algorithms but on the ability to decompose our system into subsystems and perform local checks. For *serial interface systems,* breaking the system into two subsystems allows the "combinatorial explosion" to work for us. Finally, we present a complexity analysis for verifying a *serial interface system.*

6.1 Preliminary Definitions

Before we can define our algorithms, we need to more formally define the DES to be evaluated. For a *serial subsystem based system,* we need to define the DES below. We will not need specific definitions for a *serial general form system.* We will assume that we will be given G_H, G_L, $\mathcal{G}_H$, $\mathcal{S}_H$, $\mathcal{G}_L$, $\mathcal{S}_L$, Σ_H, Σ_L, Σ_R, Σ_A, Σ_u, Σ_c, and the map **Answer** : $\Sigma_R \rightarrow \text{Pwr}(\Sigma_A)$ (see definition on page 25), but will have to construct G_I.

Also, we assume that all DES are deterministic and have finite state and event sets.

$$\begin{aligned} G_H &:= (Y_H, \Sigma_{G_H}, \delta_H, y_{H_o}, Y_{H_m}) \\ G_L &:= (Y_L, \Sigma_{G_L}, \delta_L, y_{L_o}, Y_{L_m}) \\ G_I &:= (X, \Sigma_{G_I}, \xi, x_o, X_m) \end{aligned}$$

We next define our system's event set as

$$\Sigma = \Sigma_H \dot{\cup} \Sigma_L \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A$$

We now define a useful operator. In Section 2.1, we defined the transition function $\delta : Y \times \Sigma^* \to Y$ for a given DES $\mathbf{G} = (Y, \Sigma, \delta, y_o, Y_m)$. In this chapter, we will need to use the inverse transition function $\delta^{-1} : Y \times \Sigma^* \to \mathbf{Pwr}\ (Y)$. For $s \in \Sigma^*$ and $y \in Y$ we have:

$$\delta^{-1}(y, s) := \{y' \in Y \mid \delta(y', s) = y\}$$

6.2 Evaluating *Star Interfaces*

Our approach for evaluating *star interfaces* is to construct our *star interface, G_I*, correct by design. We start by defining $\Sigma_{G_I} = \Sigma_R \cup \Sigma_A$ and specifying that the state set X is a subset of $\mathcal{N} = \{0, 1, \ldots\}$.

We next define the initial state to be $x_o = 0$, and the marked states to be $X_m = \{x_o\}$.

We now define the one step transition function $\xi' : X \times \Sigma_{G_I} \to X$ and the state set X iteratively. We start by initializing ξ' to be undefined for all $(x, \sigma) \in X \times \Sigma_{G_I}$, and the state set to be $X = \{x_o\}$. We also define the variable *count* and initialize it to be *count = 1.*

We now evaluate:

for ρ **in** Σ_R
    $x' :=$ *count* ; *// assign x' next unused state*

$count := count + 1$;

$X := X \cup \{x'\}$;

$\xi'(x_o, \rho) := x'$; *// define transition for ρ to state in X_A*

// Set answer event *transitions to return to initial state.*

for α **in Answer**(ρ)

$\xi'(x', \alpha) := x_o$;

end for

end for

We have now defined the state set X. We define the partial function $\xi : X \times \Sigma^*_{G_I} \to X$ by extending the one step transition function ξ' in the standard way.

All that remains is to verify the two points below. As this is straightforward, we will not present a specific algorithm.

1. $(\forall \rho \in \Sigma_R)$ **Answer**$(\rho) \neq \emptyset$
2. $\Sigma_A = \cup_{\rho \in \Sigma_R}$ **Answer**(ρ)

6.3 Evaluating *Serial Interface Consistent*

We now evaluate the *serial interface consistent* definition. This requires evaluating **Points 1-6** of the definition, as well as the implied **Point 0** ($\Sigma = \Sigma_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$). We will evaluate **Points 5** and **6** simultaneously as their evaluations fit together nicely.

6.3.1 Point 0

In the *serial interface consistent* definition, the event set is defined to be $\Sigma := \Sigma_H \dot\cup \Sigma_L \dot\cup \Sigma_R \dot\cup \Sigma_A$. This contains two implicit assumptions. The first is that $\Sigma = \Sigma_H \cup \Sigma_L \cup \Sigma_R \cup \Sigma_A$. From the definitions in Section 6.1, this is automatic.

The second implicit assumption is that the four event sets are pairwise disjoint. This means checking the three points below. As this is straightforward, we won't present a specific algorithm.

1. $\Sigma_R \cap \Sigma_A = \emptyset$

2. $\Sigma_H \cap \Sigma_L = \emptyset$

3. $(\Sigma_R \cup \Sigma_A) \cap (\Sigma_H \cup \Sigma_L) = \emptyset$

6.3.2 Point 1

To check **Point 1** of the interface properties, we simply check that the $\Sigma_{G_H} = \Sigma_{IH}$ and that $\Sigma_{G_L} = \Sigma_{IL}$.

6.3.3 Point 2

To verify that G_I is a *command-pair interface,* we will check that G_I satisfies the more restrictive (see **Proposition 7**) *star interface* definition. To check that G_I is a *star interface*, we will use the method described in Section 6.2. The reason we are only checking the *star interface* definition is that the *command-pair interface* was only just devised, and time didn't permit developing an algorithm for it and including it in our software tool.

6.3.4 Point 3

This property can be evaluated using normal controllability algorithms, after defining a few parameters. We define **ThePlant** $= G_I$, $TheSpec = G_H$, $\Sigma_u = \Sigma_A$, and $\Sigma_c = \Sigma - \Sigma_A$. We then check that **TheSpec** is controllable for **ThePlant** using algorithms like CTCT's **condat** function (see [60]).

6.3.5 Point 4

This property can be evaluated using normal controllability algorithms, after defining a few parameters. We define **ThePlant** $= G_I$, $TheSpec = G_L$, $\Sigma_u = \Sigma_R$, and $\Sigma_c = \Sigma - \Sigma_R$. We then check that **TheSpec** is controllable for **ThePlant** using algorithms like CTCT's **condat** function (see [60]).

6.3.6 Points 5 and 6

We now present an algorithm for verifying **Point 5** and **Points 6** simultaneously. To verify **Point 6**, we will first attempt to verify that the system satisfies the more restrictive (see

Proposition 10) *serial interface strict marking* condition and if that fails, the actual **Point 6** property.

Before we can present the algorithm, we need to construct a DES to represent the *low level*. As we wish to avoid repeating existing algorithms, we will assume we already have the DES $G_{IL} = G_L||_s G_I = (Y_{IL} \subseteq Y_L \times X, \Sigma_{G_{IL}}, \delta_{IL}, y_{IL_o}, Y_{IL_m})$. This can be created by using the CTCT **sync** operator (see [60]). We also assume that the inverse transition function δ_{IL}^{-1} was defined as part of the **sync** algorithm. Our algorithm will start from this point.

We will present our algorithm in two parts. **Part I** of the algorithm will verify if **Point 5** and the *serial interface strict marking* definition are satisfied. If the system is not *serial interface strict marking*, a list of states from DES G_{IL} whose X component is marked by G_I, but the Y_L component is not marked by G_L, is created. This list, equivalent to $Y_{IL} \cap [(Y_L - Y_{L_m}) \times X_m]$ will be used as a starting point for **Part II** of the algorithm.

Before we present **Parts I** and **II** of the algorithm in the sections below, we first need to define some variables that they will require.

Y_{ck_mk}**:** This is the set of states of G_{IL} that are in $Y_{IL} \cap [(Y_L - Y_{L_m}) \times X_m]$. These states represent strings that cause the system to not be *serial interface strict marking*.

Y_{fnd}**:** This is the set of states of G_{IL} reached in a given phase of the algorithm. Its exact meaning will be dependent on the point in the algorithm it is being used.

Y_{pend}**:** This is the list of states of G_{IL} that are pending. They are the set of states the algorithm has found, but not yet processed. Its exact meaning will be dependent on the point in the algorithm it is being used.

$\Sigma_{\neg fnd}$**:** This set represent answer events in $\mathbf{Answer}(\rho)$ for some $\rho \in \Sigma_R$ that have not yet been reached by the search while evaluating **Point 5**.

Part I: Check Point 5 and *Serial Interface Strict Marking* Definition

The first part of our algorithm checks each state of the *low level* to see if its X component is marked by G_I. These states represent the only places where *request events* are defined as well as strings marked by G_I. If a given state $y \in Y_{IL}$ is marked by G_I, the state is checked to see if it is also marked by G_{IL}. If state y is in Y_{IL_m}, then the state's Y_L component is

marked by G_L (i.e. a string leading to this state must be in $\mathcal{L}_m \cap \mathcal{I}_m$). If state y is not marked by G_{IL}, then this means the system is not *serial interface strict marking*. This state is added to list Y_{ck_mk} to be checked in **Part II**.

Our next step is to check that we can find a path from each *request event* (leaving state y) to each *answer event* that can follow that *request event*. If this test fails, then **Point 5** is not satisfied and we stop immediately.

After we have checked every state, we then check if list Y_{ck_mk} is empty. If it is, then every state in G_{IL} marked by G_I is also, marked by G_L, and thus the system is *serial interface strict marking*. If the list is non-empty, it is passed to **Part II** and we check to see if **Point 6** of the *serial interface consistent* definition is satisfied.

We now present the algorithm for **Part I**. We start by initializing variable Y_{ck_mk} to be $Y_{ck\_mk} := \emptyset$.

// Check each state in G_{IL}
for y **in** Y_{IL}
  if $(y \in Y_L \times X_m)$ **then** *// Process state if marked by interface* (G_I)
    if $(y \notin Y_{IL_m})$ **then**
      // If reach here, then system is not serial interface strict marking
      $Y_{ck\_mk} := Y_{ck\_mk} \cup \{y\}$;
    end if
    // Process each request event. *This part checks* **Point 5**
    for ρ **in** Σ_R
      if $(\delta_{IL}(y, \rho)!)$ **then**
        // Search starts after request event *occurs*
        $Y_{fnd} := \{\delta_{IL}(y, \rho)\}$;
        $Y_{pend} := \{\delta_{IL}(y, \rho)\}$;
        // List of answer events *that we must be able to reach after* ρ *has occurred*
        $\Sigma_{\neg fnd} :=$ **Answer**(ρ) ;
        // Loop until no more states to process or found all $\alpha \in$ **Answer**(ρ)
        while $(Y_{pend} \neq \emptyset$ **and** $\Sigma_{\neg fnd} \neq \emptyset)$ **do**
          select $y' \in Y_{pend}$;
          $Y_{pend} := Y_{pend} - \{y'\}$;

```
            // Determine next states
            for σ in Σ_IL
               if ( δ_IL(y', σ)! ) then
                  y'' := δ_IL(y', σ);
                  // Terminate branches ending in answer events
                  if (σ ∈ Σ_A) then
                     if (σ ∈ Σ_¬fnd) then
                        // Found new event in Answer(ρ). Remove from list
                        Σ_¬fnd := Σ_¬fnd − {σ};
                     end if
                  else if ((σ ∈ Σ_L) ∧ (y'' ∉ Y_fnd)) then
                     // Mark state as found and as pending to be explored
                     Y_fnd := Y_fnd ∪ {y''};
                     Y_pend := Y_pend ∪ {y''};
                  end if
                  if (Σ_¬fnd = ∅) then
                     // Have found path to all events in Answer(ρ) so exit loop
                     exit for loop;
                  end if
               end if
            end for
         end while
         // Determine if we exited while loop because we reached all required events,
         // or ran out of states to examine
         if (Σ_¬fnd ≠ ∅) then
            return "point 5 fails";
         end if
      end if
   end for
end if
end for
```

// If reach here, then **Point 5** *holds*

if $(Y_{ck\_mk} = \emptyset)$ **then**

return *"points 5 and 6 pass"*;

end if

// If reached here, **Point 5** *holds, but the system is not* serial interface strict marking.

// We need to evaluate **Part II** *of the algorithm*

Part II: Check Point 6 of *Serial Interface Consistent* Definition

The second part of our algorithm is evaluated if **Part I** determines that the system is not *serial interface strict marking.* **Part II** does a reverse reachability check (i.e. uses the inverse transition function) using only events $\sigma \in \Sigma_L$. It starts at the states marked by G_{IL} (i.e. states representing strings in $\mathcal{L}_m \cap \mathcal{I}_m$) and does a reverse traversal to try to find a path to each state in Y_{ck_mk}, the state set constructed during **Part I** of the algorithm. This state set represents states in $Y_{IL} \cap [(Y_L - Y_{L_m}) \times X_m]$. If each state in Y_{ck_mk} is reachable from a state in Y_{IL_m} using only $\sigma \in \Sigma_L$, then **Point 6** of *serial interface consistent* definition is satisfied.

We now present the algorithm for **Part II**.

$Y_{fnd} := \emptyset$;

// Loop through each marked state in G_{IL}

for y_m **in** Y_{IL_m}

// Check that we haven't already processed this state

// while processing another marked state

if $(y_m \notin Y_{fnd})$ **then**

$Y_{fnd} := Y_{fnd} \cup \{y_m\}$;

$Y_{pend} := \{y_m\}$;

while $(Y_{pend} \neq \emptyset)$ **do**

select $y \in Y_{pend}$;

$Y_{pend} := Y_{pend} - \{y\}$;

for σ **in** Σ_L *// Do reverse reachability search only using* $\sigma \in \Sigma_L$

if $(\delta_{IL}^{-1}(y,\sigma) \neq \emptyset)$ **then**

// Determine set of next states

$Y := \delta_{IL}^{-1}(y,\sigma)$;

for y' **in** Y

if $(y' \notin Y_{fnd})$ **then**

// Add state to list to be examined

$Y_{fnd} := Y_{fnd} \cup \{y'\}$;

$Y_{pend} := Y_{pend} \cup \{y'\}$;

if $(y' \in Y_{ck\_mk})$ **then**

// State is one we are trying to reach. Remove from list

$Y_{ck\_mk} := Y_{ck\_mk} - \{y'\}$;

if $(Y_{ck\_mk} = \emptyset)$ **then**

// We have reached all required states so **Point 6** *has been*

// satisfied. We know **Point 5** *is satisfied from* **Part I**

return *"points 5 and 6 pass"* ;

end if

end if

end if

end for

end if

end for

end while

end if

end for

// If we reach here, then we were unable to reach one of the states marked by G_I (the interface),

// but not marked by G_L (ie. a state in Y_{ck_mk})

return *"point 6 fails"* ;

Simple Illustrative Example

As the algorithm in this section is the only truly new algorithm,[1] we present here a small example to illustrate it. In the DES diagrams that follow, *uncontrollable events* are shown in italics; all other events are *controllable.* Initial states can be recognized by a thick outline, and marked states are filled.

We present here a simple *low level* to be examined. Our *interface* is shown in Figure 6.1. The *request event start2jb* signals that two items have been put into the input buffer. The *answer event finish2jb* signals that the two items have been processed by **Machine1** (Figure 6.2) and are in the output buffer, ready to be picked up. DES **Signal** (Figure 6.3) allows the event sig2done to occur to mark internally that two items have been processed. This is done so that the system will not be *serial interface strict marking,* and we will thus have to evaluate **Part II** of the algorithm. A better example would have been to have two machines using the same input/output buffers. We could then have had the situation that one machine breaks down and the other alone processes the two parts. We would then have the *interface* in a marked state, and one machine in the *down* state and thus not in a marked state. We would then have to evaluate **Part II** of the algorithm. This example was not used as it would have been too large to do easily by hand.

Figure 6.1: Interface for Algorithm Example

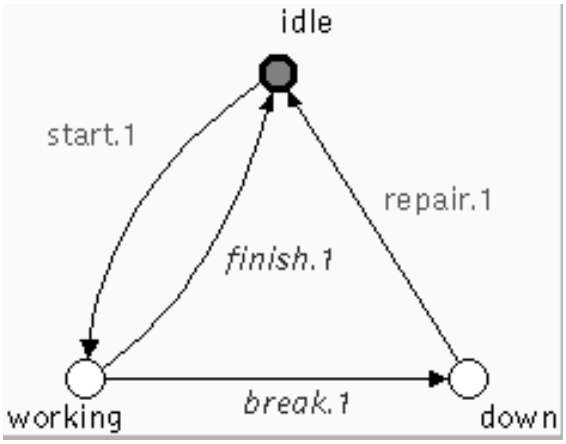


Figure 6.2: DES Machine1

Next, the input buffer supervisor (Figure 6.5) ensures that **Machine1** is idle until *start2jb* occurs, and then is idle again after the two items are successfully processed. If event *break.1* occurs, it's assumed that the part is undamaged and is replaced immediately

[1]The other conditions can be checked using existing algorithms in new ways.

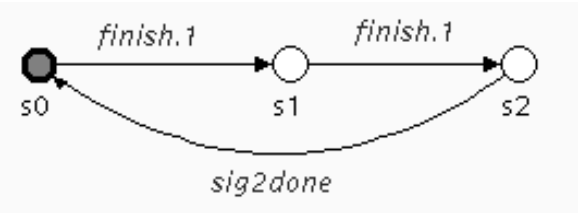


Figure 6.3: Signaling DES

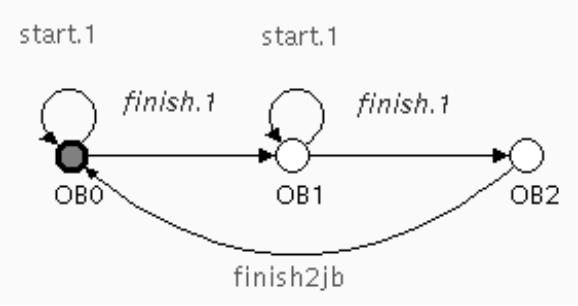


Figure 6.4: Output Buffer

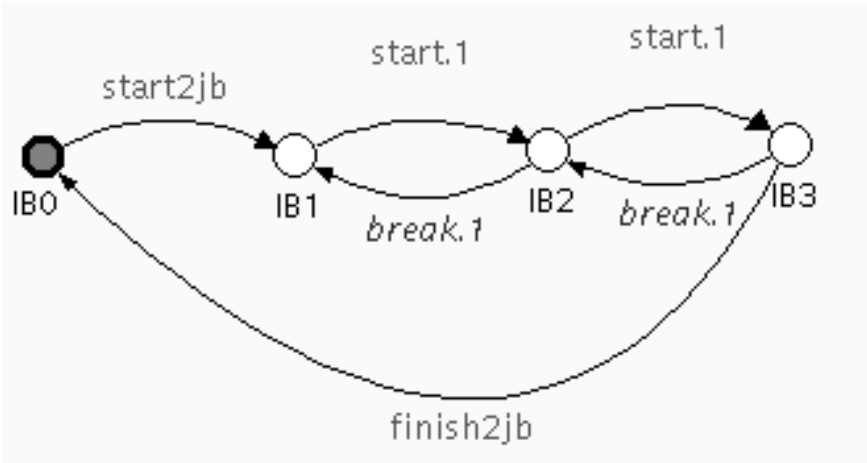


Figure 6.5: Input Buffer

into the input buffer. The machine will then try to process the item again as soon as it's repaired.

Finally, the output buffer supervisor (Figure 6.4) ensures that **Machine1** doesn't try to process more than 2 items per cycle, and allows event *finish2jb* to occur only after two items have been successfully processed.

Figure 6.6 shows the definition of DES **sync.des** which is the synchronous product of the DES in Figures 6.1 to 6.5. We will take **sync.des** to be our DES G_{IL}. Examining the listing, we see that it is composed of a list of the component DES that make up the synchronous product (and a list of their state labels), followed by the marked states of **sync.des**, the initial state, the set of controllable events, uncontrollable events, and finally a list of transitions in the form *event* : *start state* $\rightarrow$ *end state*. We next note that states of **sync.des** are given as tuples of the states of the component DES. Finally, our set of states for G_{IL}, Y_{IL}, can be constructed by listing all of the states that appear in the list of transitions.

Analyzing Part I

We now start by evaluating **Part I** of the algorithm. We set $Y_{ck\_mk} := \emptyset$ and select y = *(idle,s0,OB0, Itf0,IB0)* (the initial state) from Y_{IL}. As state *Itf0* is marked by the *interface*, we check that $y \notin Y_{IL_m}$. From the listing, *(idle,s0,OB0,Itf0,IB0)* is marked by **sync.des**, so we don't add y to Y_{ck_mk}.

We now process all *request events* defined at y. This is only event *start2jb* which takes us to state y' = *(idle,s0,OB0,Itf1,IB1)*. We now need to find a string that takes us to event *finish2jb*, the only *answer event* that can follow *start2jb*. Analyzing the set of transitions, we see that string
start.1 finish.1 start.1 finish.1 takes us to state *(idle,s2,OB2,Itf1,IB3)* at which event *finish2jb* is defined. We have thus found a path to all *answer events* that can follow *start2jb*, so we return to the top of the algorithm, and select another y **in** Y_{IL}.

Examining the states in the list of transitions, we see that the only state left that contains a state component marked by the *interface* is state *(idle,s2,OB0,Itf0,IB0)*. As these other states would not satisfy $y \in Y_L \times X_m$, we can safely ignore them.

We thus take y = *(idle,s2,OB0,Itf0,IB0)*, and evaluate it. From the listing, we see that $y \notin Y_{IL_m}$ so we set $Y_{ck\_mk} := \{$*(idle,s2,OB0,Itf0,IB0)*$\}$. This means the system is not *serial*

sync.des

```
@des sync {
  @stateComponents {
    interface {
      Itf1
      Itf0
    }
    l_Inbuff {
      IB3
      IB2
      IB1
      IB0
    }
    l_Outbuff {
      OB2
      OB1
      OB0
    }
    l_mach.1 {
      down
      working
      idle
    }
    l_signal {
      s2
      s1
      s0
    }
  }
  @markedStates {
    (idle,s0,OB0,Itf0,IB0)
  }
  @initial {
    (idle,s0,OB0,Itf0,IB0)
  }
  @controllable {
    start.1
    repair.1
    finish2jb
    start2jb
  }
  @uncontrollable {
    sig2done
    finish.1
    break.1
  }
  @transitions {
    [start.1] (idle,s0,OB0,Itf1,IB1) -> (working,s0,OB0,Itf1,IB2)
    [start.1] (idle,s1,OB1,Itf1,IB2) -> (working,s1,OB1,Itf1,IB3)
    [start.1] (idle,s2,OB0,Itf1,IB1) -> (working,s2,OB0,Itf1,IB2)
    [repair.1] (down,s0,OB0,Itf1,IB1) -> (idle,s0,OB0,Itf1,IB1)
    [repair.1] (down,s1,OB1,Itf1,IB2) -> (idle,s1,OB1,Itf1,IB2)
    [repair.1] (down,s2,OB0,Itf1,IB1) -> (idle,s2,OB0,Itf1,IB1)
    [finish2jb] (idle,s2,OB2,Itf1,IB3) -> (idle,s2,OB0,Itf0,IB0)
    [finish2jb] (idle,s0,OB2,Itf1,IB3) -> (idle,s0,OB0,Itf0,IB0)
    [start2jb] (idle,s0,OB0,Itf0,IB0) -> (idle,s0,OB0,Itf1,IB1)
    [start2jb] (idle,s2,OB0,Itf0,IB0) -> (idle,s2,OB0,Itf1,IB1)
    [sig2done] (idle,s2,OB2,Itf1,IB3) -> (idle,s0,OB2,Itf1,IB3)
    [sig2done] (idle,s2,OB0,Itf0,IB0) -> (idle,s0,OB0,Itf0,IB0)
    [sig2done] (idle,s2,OB0,Itf1,IB1) -> (idle,s0,OB0,Itf1,IB1)
    [sig2done] (working,s2,OB0,Itf1,IB2) -> (working,s0,OB0,Itf1,IB2)
    [sig2done] (down,s2,OB0,Itf1,IB1) -> (down,s0,OB0,Itf1,IB1)
    [finish.1] (working,s0,OB0,Itf1,IB2) -> (idle,s1,OB1,Itf1,IB2)
    [finish.1] (working,s1,OB1,Itf1,IB3) -> (idle,s2,OB2,Itf1,IB3)
    [break.1] (working,s0,OB0,Itf1,IB2) -> (down,s0,OB0,Itf1,IB1)
    [break.1] (working,s1,OB1,Itf1,IB3) -> (down,s1,OB1,Itf1,IB2)
    [break.1] (working,s2,OB0,Itf1,IB2) -> (down,s2,OB0,Itf1,IB1)
  }
}
```

Figure 6.6: DES Listing for sync.Des

interface strict marking.

We now process all *request events* defined at y. This is only event *start2jb* which takes us to state $y' =$ *(idle,s2,OB0,Itf1,IB1).* We now need to find a string that takes us to event *finish2jb*, the only *answer event* that can follow *start2jb*. Analyzing the set of transitions, we see that string
sig2done start.1 finish.1 start.1 finish.1 takes us to state *(idle,s2,OB2,Itf1,IB3)* at which event *finish2jb* is defined. We have thus found a path to all *answer events* that can follow *start2jb*, so we return to the top of the algorithm, to select another y **in** Y_{IL}.

We have now evaluated all y **in** Y_{IL}, so we exit the top-most loop. We thus have determined that **Point 5** of the *serial interface consistent* definition is satisfied. We then note that $Y_{ck\_mk} \neq \emptyset$ so we must evaluate **Part II** of the algorithm to determine if our system satisfies **Point 6** of the *serial interface consistent* definition.

Analyzing Part II

From the listing, we see that there is only one marked state in Y_{IL_m} so we set
$y_m =$ *(idle,s0,OB0,Itf0,IB0).* We now must find a low level string (no interface events) that takes the only state in $Y_{ck\_mk} := \{$*(idle,s2,OB0,Itf0,IB0)*$\}$ to y_m. Examining the list of transitions, we see immediately that string *sig2done* takes *(idle,s2,OB0,Itf0,IB0)* to state $y_m =$ *(idle,s0,OB0,Itf0,IB0).* This tells us that *(idle,s2,OB0,Itf0,IB0)* $\in \delta_{IL}^{-1}(y_m,$ *sig2done*$)$.

We can now remove state *(idle,s2,OB0,Itf0,IB0)* from Y_{ck_mk}, and we thus have $Y_{ck\_mk} = \emptyset$. This means that our system satisfies **Point 6** of the *serial interface consistent* definition so we exit the algorithm. We have now applied **Part I** and **II** of the algorithm, and successfully verified that the system satisfies **Point 5** and **6** of the *serial interface consistent* definition.

6.4 Evaluating *Serial Level-wise Nonblocking*

We now discuss evaluating the *serial level-wise nonblocking* definition. To do this we need to evaluate **Point I** and **Point II** of the definition. For **Point I**, we simply need to evaluate that DES $G_L||_sG_I$ is nonblocking. This can be evaluated by using algorithms like the **nonconflict** function of CTCT (see [60]). Similarly for **Point II**, we can verify that DES $G_H||_sG_I$ is nonblocking by using the **nonconflict** function of CTCT.

6.5 Evaluating *Serial Level-wise Controllability*

We next discuss evaluating the *serial level-wise controllable* definition. To do this we need to evaluate **Points I-III** of the definition as discussed in the following sections. The *serial level-wise controllable* definition also has the implied condition $\Sigma = \Sigma_H \dot{\cup} \Sigma_L \dot{\cup} \Sigma_R \dot{\cup} \Sigma_A$. This can be evaluated as in Section 6.3.1.

6.5.1 Point I

To check the first point, we simply check that the $\Sigma_{\mathcal{G}_H} = \Sigma_{IH}$, $\Sigma_{\mathcal{S}_H} = \Sigma_{IH}$, $\Sigma_{\mathcal{G}_L} = \Sigma_{IL}$, $\Sigma_{\mathcal{S}_L} = \Sigma_{IL}$, and that $\Sigma_{G_I} = \Sigma_I$. As this is straightforward, we will not present a specific algorithm.

6.5.2 Point II

This property can be evaluated using normal controllability algorithms, after defining a few parameters. We define **ThePlant** $= \mathcal{G}_L$, $TheSpec = \mathcal{S}_L||_s G_I$, $\Sigma_u = \Sigma_u \cap \Sigma_{IL}$, and $\Sigma_c = \Sigma_c \cap \Sigma_{IL}$. We then check that **TheSpec** is controllable for **ThePlant** using algorithms like CTCT's **condat** function (see [60]). DES $\mathcal{S}_L||_s G_I$ can be constructed using CTCT's **sync** function.

6.5.3 Point III

This property can be evaluated using normal controllability algorithms, after defining a few parameters. We define **ThePlant** $= \mathcal{G}_H||_s G_I$, $TheSpec = \mathcal{S}_H$, $\Sigma_u = \Sigma_u \cap \Sigma_{IH}$, and $\Sigma_c = \Sigma_c \cap \Sigma_{IH}$. We then check that **TheSpec** is controllable for **ThePlant** using algorithms like CTCT's **condat** function (see [60]). DES $\mathcal{G}_H||_s G_I$ can be constructed using CTCT's **sync** function.

6.6 Complexity Analysis

From the above sections, we see that to verify that a *serial interface system* satisfies the *serial level-wise nonblocking, serial level-wise controllable,* and *serial interface consistent* definitions, we must perform the following tasks:

System Properties:

1) Construct event sets $\Sigma_{IH} = \Sigma_R \cup \Sigma_A \cup \Sigma_H$ and $\Sigma_{IL} = \Sigma_R \cup \Sigma_A \cup \Sigma_L$.

2) Verify that G_I is a *star interface.*

3) Verify that sets Σ_H, Σ_L, Σ_R, and Σ_A are pairwise disjoint. This means evaluating:

$$\begin{aligned} \Sigma_H \cap \Sigma_a &= \emptyset, \quad a \in \{L, R, A\} \\ \Sigma_L \cap \Sigma_b &= \emptyset, \quad b \in \{R, A\} \\ \Sigma_R \cap \Sigma_A &= \emptyset \end{aligned}$$

High Level Properties:

4) Verify that $\Sigma_{\mathcal{G}_H} = \Sigma_{IH}$ and $\Sigma_{\mathcal{S}_H} = \Sigma_{IH}$.

5) Verify **Point 3** of the *serial interface consistent* definition. This requires one controllability evaluation (after suitably defining the supervisor, plant and uncontrollable events).

6) Verify **Point I** of the *serial level-wise nonblocking* definition. This requires one nonblocking evaluation.

7) Verify **Point III** of the *serial level-wise controllable* definition. This requires one controllability evaluation.

Low Level Properties:

8) Verify that $\Sigma_{\mathcal{G}_L} = \Sigma_{IL}$ and $\Sigma_{\mathcal{S}_L} = \Sigma_{IL}$.

9) Verify **Point 4** of the *serial interface consistent* definition. This requires one controllability evaluation (after suitably defining the supervisor, plant and uncontrollable events).

10) Verify **Points 5,** and **6** of the *serial interface consistent* definition. This can be accomplished using the algorithms in Section 6.3.6.

11) Verify **Point II** of the *serial level-wise nonblocking* definition. This requires one nonblocking evaluation.

12) Verify **Point II** of the *serial level-wise controllable* definition. This requires one controllability evaluation.

We now note that the algorithms required for verifying the *high level* properties (tasks 4-7) are contained in the algorithms used to verify the *low level* properties (specifically tasks 8-9, and 11-12). This means that the time complexity of evaluating the *low level* provides an upper bound for evaluating the *high level*. We can thus evaluate the time complexity for

the *low level* and use it as a general per component complexity estimate.

6.6.1 Analyzing Per Component Algorithm

The next logical step is to perform an analytic analysis of the worst case time complexity of the algorithms required to verify the above properties. Unfortunately, we do not have the proper resources. To do an analytic analysis of the above algorithms would require program sourcecode, at a minimum, to provide details of the data structures used and how they are accessed. The only sourcecode available is copyrighted by Siemens and cannot be released.

Instead, we follow the advice of Goodrich et al. [24] and use experimental algorithm analysis to estimate the worse case time complexity for per component analysis. As we will see, this is sufficient as the per component complexity only contributes a constant term to the overall complexity of evaluating a serial system.

To perform this analysis, we will use the *power test* discussed in [24]. With this method, we will take x to be the state size of our component and $y = t(x)$ to be our running time. We assume that $t(x)$ is of the form bx^c for some constants $b > 0$ and $c > 0$. If we take $y' = \log_2(y)$ and $x' = \log_2(x)$, we would then have $y' = cx' + \log_2(b)$. We see immediately that if we plot the (x', y') determined experimentally, we can do a simple line fit and determine the values of b and c from the line's y-intercept and slope, respectively. Should the data pairs grow significantly, we can confidently conclude that $t(x)$ is super-polynomial. Goodrich et al. claim that this test is at best accurate to the range $[c - 0.5, c + 0.5]$.

To keep things simple, we want to be able to express our runtime on a per component basis. To do this, we will represent our per component time as the time to perform the *low level* properties (tasks 8-12) as well as the system properties (tasks 1-3). This will be a conservative estimate as the system properties only need to be performed once per system, and there are fewer tasks to perform if the component happens to be the *high level*. However, we can now represent our running time for a serial system as simply performing the per component evaluation twice.

We now use the examples from Chapters 9, and 11 to construct seven *low levels* of varying sizes to evaluate. We then evaluated these *low levels* (ie. performed tasks 1-3, and 8-12) using our software tool that we discussed in Section 4.3.4. We ran the software 4 times per *low level*, and recorded the runtime in seconds. We then took the average value. The result is shown in Table 6.1.

State Size (x)	$x' = \log_2(x)$	Average Runtime (y)	$y' = \log_2(y)$
68	6.087	0.0407	-4.618
98	6.615	0.0497	-4.331
203	7.665	0.0900	-3.473
204	7.672	0.0792	-3.659
41,651	15.346	10.8212	3.436
291,614	18.154	431.7915	8.754
1,170,600	20.159	1284.3450	10.327

Table 6.1: Experimental Data

We next plotted the (x', y') pairs and performed a line fit. The results are shown in Figure 6.7. Of interest are the two slopes represented by lines Y_1 and Y_2. Line Y_2 appears to represent the average runtime of the per component algorithms, while line Y_1 has the steepest slope and is thus our best estimate of the worst case running time of our algorithm. We thus have $t(x) = (8.56 \times 10^{-9})x^{1.96}$ as our worst case estimate. From above, we know that this estimate for c is at best in the range $[1.96 - 0.5, 1.96 + 0.5] = [1.46\ ,\ 2.46]$. As we wish to be very conservative, we will take $c = 3$. We thus have $t(x) = (8.56 \times 10^{-9})x^3$. We thus conclude, based on our experimental data, that our per component algorithm (including system properties and irrespective of whether the component represents a *high level* or a *low level*) is $\mathbf{O}(x^3)$.

6.6.2 Analyzing Per System Algorithm

We now need to determine the complexity on a per system basis. For our purposes, we let m be the number of components to be verified, and we assume that the statespace and of each component and the cardinality of the system's event set (Σ) are bounded, with upper bounds N and N_Σ respectively (ie. $x \leq N$ and $|\Sigma| \leq N_\Sigma$). We now note that to verify that a *serial interface system* is *serial level-wise nonblocking, serial level-wise controllable,* and *serial interface consistent,* we simply need to apply the per component algorithm m times. This means that the time complexity for verifying a system is thus $\mathbf{O}(mx^3)$. Since our statespace is bounded by N, we can replace x by N and we thus have that $x^3 = N^3$, which is now a constant term. This implies that our system is actually $\mathbf{O}(m)$. In other words, as we add more components, we are simply repeating the work of analyzing one component. Of course, this only remains practical as long as N isn't so large that it contributes a

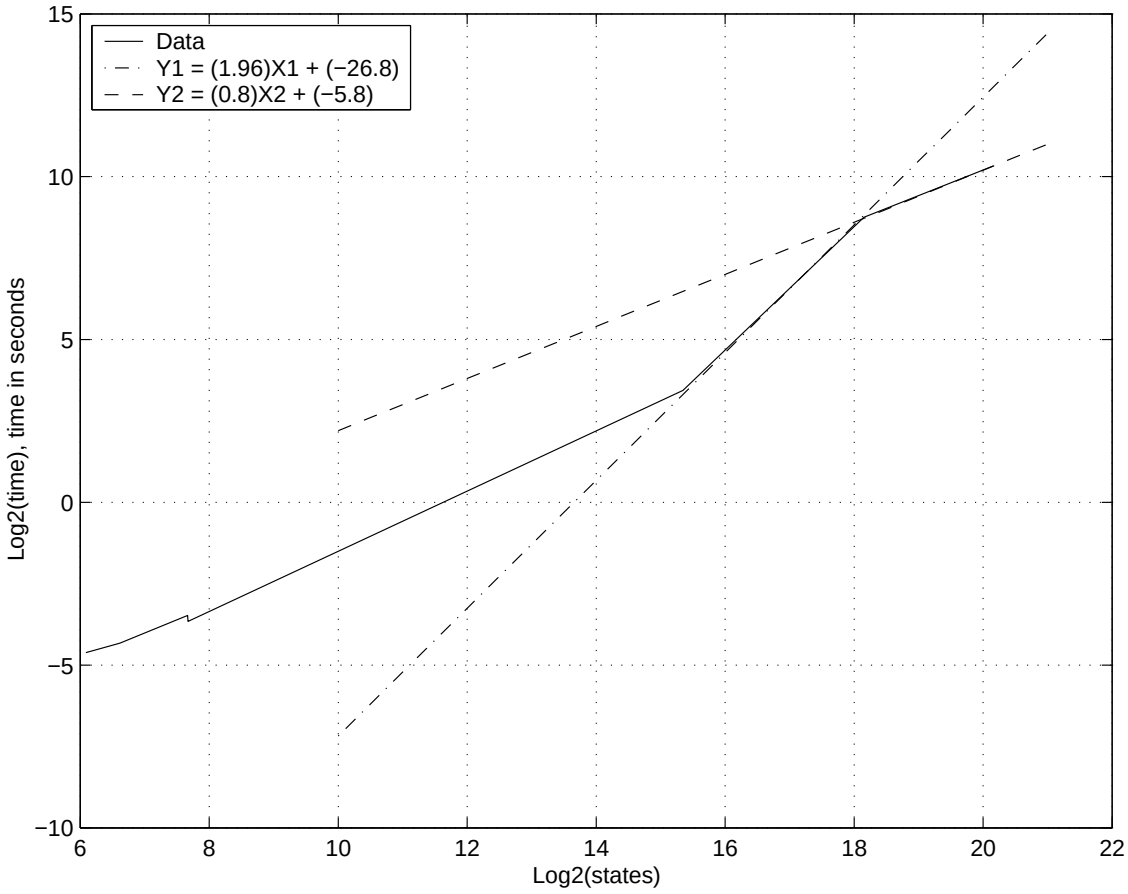


Figure 6.7: Timing Data for Experimental Analysis

prohibitively large constant term.

6.6.3 Comparison to Monolithic Algorithm

We now compare the complexity of our approach to a standard verification on a monolithic system (ie. the *flat system*). As our method can tell us if a system is nonblocking and the supervisors controllable, we should compare it to a nonblocking and controllability verification on the *flat plant.* To keep things simple, we will only consider the complexity of a nonblocking verification. This will be a conservative estimate, as the complexity of a nonblocking and controllability analysis would be additive.

We now turn to the work of Rudie to determine the time complexity for a monolithic

nonblocking analysis. In [51], Rudie presented an algorithm for constructing a trim[2] DES that was $\mathbf{O}(x+e)$, where x is the state size of the original DES, and e was the number of transitions it contained. In this algorithm, Rudie constructed sets *Accessinit* and *CoaccessMarked2* which represent the reachable and coreachable states, respectively, of the original DES. Clearly, if and only if *Accessinit* = *CoaccessMarked2* would the original DES be nonblocking. It's easy to show that if the sets are stored as arrays, then set equality can be checked in $\mathbf{O}(|Accessinit||CoaccessMarked2|)$. As both sets are bounded by x, we can conclude that checking the equality is $\mathbf{O}(x^2)$. This implies that verifying that the DES is nonblocking is $\mathbf{O}(x+e+x^2)$. As we are only considering deterministic DES, we know that $e \leq N_\Sigma x$. We thus have that the algorithm is $\mathbf{O}(x+N_\Sigma x+x^2) = \mathbf{O}((1+N_\Sigma)x+x^2)$, which reduces to $\mathbf{O}(x^2)$. However, this algorithm will be operating on the synchronous product of our m components (ie. the *flat system*). This means that $x \leq N^m$. Substituting this in, we thus see that our monolithic algorithm is $\mathbf{O}((N^m)^2) = \mathbf{O}(N^{2m})$.

Comparing the complexity of our algorithm to the monolithic algorithm above, we see that our interface-based method scales significantly better at $\mathbf{O}(m)$ than the other at $\mathbf{O}(N^{2m})$. To illustrate this, let's examine the two for a few values of N and m, shown in Table 6.2. For $m = 1$, the degenerative case, we see the monolithic algorithm outstrips our approach as expected.[3] For $m = 2$, the serial case, we see that our method scales significantly better than the monolithic approach. For $N = 10^6$, our method is six orders of magnitude better. To put this into perspective, if our algorithm ran for one hour, the monolithic algorithm would require 114 years! Finally for $m = 9$, we see that if we allow more than one *low level*, the potential gain is incredible. For $N = 10^6$, the running time of our approach is only multiplied by nine (relative to the time for $m = 1$), while the monolithic algorithm increases from 10^{12} to 10^{108}. Our approach would be 90 orders of magnitude better!

Of course, their is a cost for this increase in computational efficiency. As the saying goes, "there is no such thing as a free lunch." The trade-off is a more restrictive architecture. The interface approach restricts knowledge about internal details of components, and only

[2] A DES is *trim* if it is reachable and coreachable.

[3] The degenerative case is equivalent to having a normal *high level* with $\Sigma_L = \Sigma_R = \Sigma_A = \emptyset$, and DES $\mathcal{G}_L$ and $\mathcal{S}_L$ only containing an initial state, which is marked. In actuality, the interface specific conditions are trivially satisfied in this case, and the required checks are only that the *high level* is nonblocking and controllable which is equivalent to the monolithic algorithm.

	$m = 1$		$m = 2$		$m = 9$	
N	N^{2m}	mN^3	N^{2m}	mN^3	N^{2m}	mN^3
10	10^2	10^3	10^4	2×10^3	10^{18}	9×10^3
10^3	10^6	10^9	10^{12}	2×10^9	10^{54}	9×10^9
10^6	10^{12}	10^{18}	10^{24}	2×10^{18}	10^{108}	9×10^{18}
10^{20}	10^{40}	10^{60}	10^{80}	2×10^{60}	10^{360}	9×10^{60}

Table 6.2: Serial Algorithm Comparison

allows supervisors to disable local events and interface events. However, if the interface is well designed, the restrictions should only have the effect of removing unnecessary clutter. In other words, most of the options/choices removed by the restrictions placed on the system's design by the interface would have been unused anyway. As similar interface-based approaches are common in both hardware and software, we are confident that are method will be widely applicable.

6.6.4 Quality of Complexity Analysis

In Section 6.6.1 above, we determined that our per component algorithm is $\mathbf{O}(x^3)$ and our per system algorithm is $\mathbf{O}(m)$. This raises the question "How accurate are these results?" Well, our experimental complexity analysis for our per component algorithm is only as good as the example systems are representative of the worse case running time for the algorithm. Unfortunately, it is no easy task determining how "representative" a system is. To compensate for this, we chose the steepest slope to calculate our exponent c, as well as increased it by one. This should be a reasonable estimate. Also, examining the algorithm in Section 6.3.6 (the only new algorithm), we see that it is very similar to a nonblocking analysis; thus, it makes sense that their worse case complexity be close.

In any case, even if our per component algorithm complexity estimate is a bit off, we showed that it only contributes a constant term to the overall algorithm for evaluating a *serial interface system.* This means that the difference can be compensated for by using a faster computer.

Chapter 7

Parallel Case: Nonblocking

In Chapter 3, we described our method for verifying nonblocking for the serial case where the number of *low levels* (n) is restricted to one. Such a system is also referred to as a *serial interface system.* We now extend our work to the more general setting where we have $n \geq 1$ *low levels* and we will refer to such a system as a *parallel interface system.* Figure 7.1 shows conceptually the structure and flow of information of a *parallel interface system.* In this new setting, we still have a single *high level,* but this time it is interacting with $n \geq 1$ autonomous low levels,[1] communicating with each *low level* in parallel through a separate *interface.* We will refer to the number of *low levels*, n, as the *degree* of the parallel system.

7.1 Definitions and Notation

We now introduce some terminology and notation that will be useful in simplifying proofs. For an n^{th} degree parallel system, we assume the *high level subsystem* is modelled by DES G_H (defined over event set $\dot{\cup}_{k\in\{1,\ldots,n\}}[\Sigma_{R_k} \dot{\cup} \Sigma_{A_k}] \dot{\cup} \Sigma_H$), the j^{th} *low level subsystem* is modelled by DES G_{L_j} (defined over event set $\Sigma_{L_j} \dot{\cup} \Sigma_{R_j} \dot{\cup} \Sigma_{A_j}$), the j^{th} *interface* by DES G_{I_j} (defined over event set $\Sigma_{R_j} \dot{\cup} \Sigma_{A_j}$), and that the overall system has the structure shown in Figure 7.2. Furthermore, we will refer to the j^{th} *low level* to mean $G_{L_j} ||_s G_{I_j}$.

As in the serial case, in order to capture the restriction of the flow of information imposed by the *interface,* we partition the alphabet of the system into analogous pairwise disjoint alphabets, as below. For the remainder of this chapter, we define j to be $j \in \{1, \ldots, n\}$.

[1] By autonomous, we mean the event set of each *low level* is pairwise disjoint from the events sets of the other *low levels.*

Σ_H**:** These are events that exist only at the *high level.*

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

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

Σ_{L_j}**:** The set of events that exist only at the j^{th} *low level.*

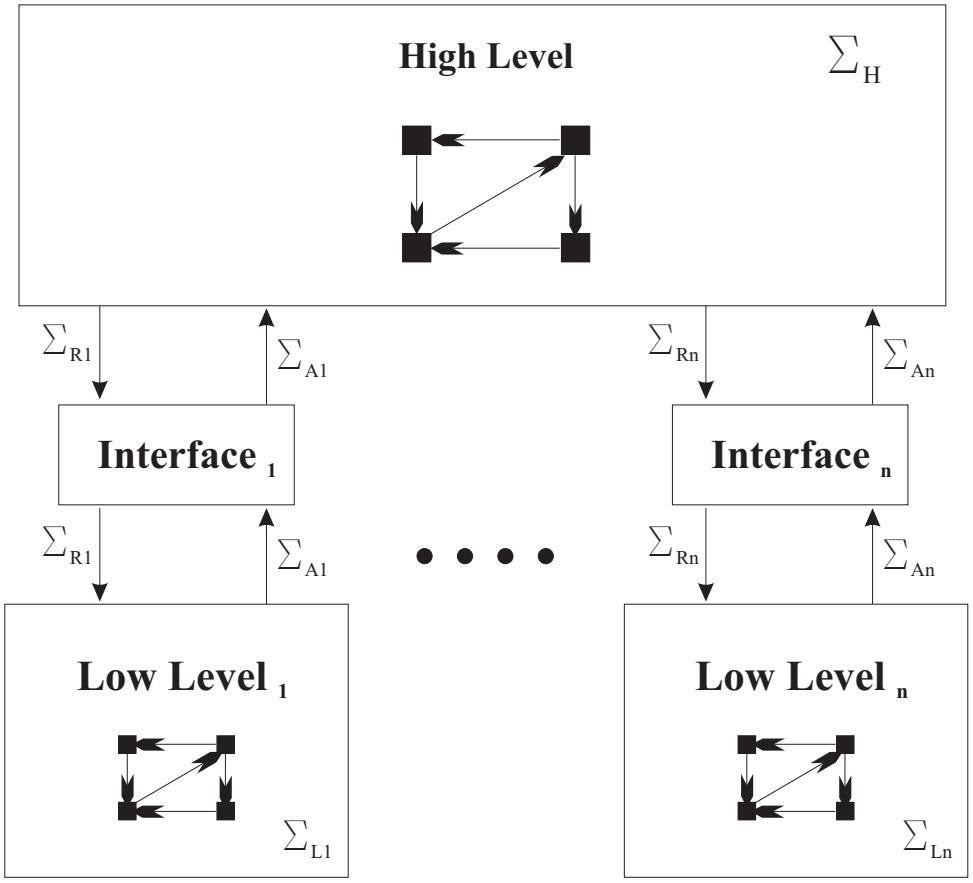


Figure 7.1: Parallel Interface Block Diagram.

We now assume that the alphabet partition is specified by $\Sigma := \dot{\cup}_{j\in\{1,\ldots,n\}}(\Sigma_{L_j} \,\dot{\cup}\, \Sigma_{R_j} \,\dot{\cup}\, \Sigma_{A_j}) \,\dot{\cup}\, \Sigma_H$ and that the *flat system* is taken to be:

$$G = G_H||_s G_{L_1}||_s \ldots ||_s G_{L_n}||_s G_{I_1}||_s \ldots ||_s G_{I_n}$$

We now introduce some useful event sets that we will be referring to often. They are

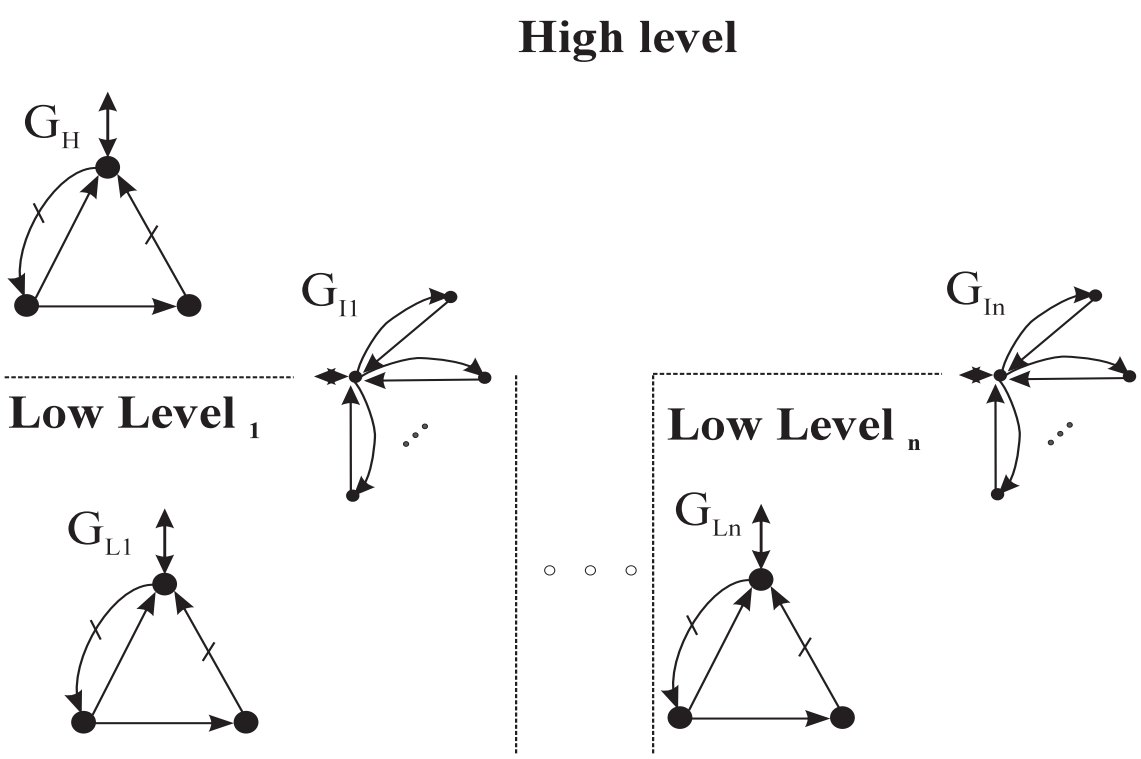


Figure 7.2: Two Tiered Structure of Parallel System

defined as below:

$$\begin{aligned}
\Sigma_{I_j} &:= \Sigma_{R_j} \cup \Sigma_{A_j} && \textit{Interface Events for the } j^{th} \textit{ Interface} \\
\Sigma_{IH} &:= \cup_{k\in\{1,\ldots,n\}}\Sigma_{I_k} \cup \Sigma_H && \textit{Interface and High Level Events} \\
\Sigma_{IL_j} &:= \Sigma_{L_j} \cup \Sigma_{I_j} && j^{th} \textit{ Set of Interface and Low Level Events} \\
\Sigma_{IL} &:= \cup_{k\in\{1,\ldots,n\}}\Sigma_{IL_k} && \textit{Set of all Interface and Low Level Events}
\end{aligned}$$

To be able to work with different languages defined over the above subsets, we define the following natural projections:

$$\begin{aligned}
P_{IH} &: \Sigma^* \rightarrow \Sigma^*_{IH} \\
P_{IL_j} &: \Sigma^* \rightarrow \Sigma^*_{IL_j} \\
P_{I_j} &: \Sigma^* \rightarrow \Sigma^*_{I_j}
\end{aligned}$$

As we want to express the languages of *flat system* in terms of their components, we need

to define the following languages:

$$\begin{aligned}
\mathcal{H} &:= P_{IH}^{-1}(L(G_H)), \quad & \mathcal{H}_m &:= P_{IH}^{-1}(L_m(G_H)) \quad &\subseteq \Sigma^* \\
\mathcal{L}_j &:= P_{IL_j}^{-1}(L(G_{L_j})), \quad & \mathcal{L}_{m_j} &:= P_{IL_j}^{-1}(L_m(G_{L_j})) \quad &\subseteq \Sigma^* \\
\mathcal{I}_j &:= P_{I_j}^{-1}(L(G_{I_j})), \quad & \mathcal{I}_{m_j} &:= P_{I_j}^{-1}(L_m(G_{I_j})) \quad &\subseteq \Sigma^*
\end{aligned}$$

We can now represent the closed behaviour of our *flat system* as follows:

$$\begin{aligned}
L(G) := \quad & L(G_H||_s G_{L_1}||_s \ldots ||_s G_{L_n}||_s G_{I_1}||_s \ldots ||_s G_{I_n}) \\
=& P_{IH}^{-1}(L(G_H)) \cap [\cap_{k\in\{1,\ldots,n\}}(P_{IL_k}^{-1}(L(G_{L_k})) \cap P_{I_k}^{-1}(L(G_{I_k})))] \\
=& \quad \mathcal{H} \cap [\cap_{k\in\{1,\ldots,n\}}(\mathcal{L}_k \cap \mathcal{I}_k)]
\end{aligned}$$

Similarly, the *flat marked language* of *system* is:

$$L_m(G) = \mathcal{H}_m \cap [\cap_{k\in\{1,\ldots,n\}}(\mathcal{L}_{m_k} \cap \mathcal{I}_{m_k})]$$

This allows us to present the proposition below that collects together several similar propositions. As it will be common in the proofs in this report to show that membership in languages such as $\mathcal{H}$ is dependent only on events in specific subsets (for $\mathcal{H}$, events in subset Σ_{IH}), this proposition will be very useful.

Proposition 20

(a) $(\forall s, s' \in \Sigma^*)\ s \in \mathcal{H}$ *and* $P_{IH}(s) = P_{IH}(s') \Rightarrow s' \in \mathcal{H}$

(b) $(\forall s, s' \in \Sigma^*)\ s \in \mathcal{H}_m$ *and* $P_{IH}(s) = P_{IH}(s') \Rightarrow s' \in \mathcal{H}_m$

(c) $(\forall k \in \{1,\ldots,n\})(\forall s, s' \in \Sigma^*)\ s \in \mathcal{L}_k$ *and* $P_{IL_k}(s) = P_{IL_k}(s') \Rightarrow s' \in \mathcal{L}_k$

(d) $(\forall k \in \{1,\ldots,n\})(\forall s, s' \in \Sigma^*)\ s \in \mathcal{L}_{m_k}$ *and* $P_{IL_k}(s) = P_{IL_k}(s') \Rightarrow s' \in \mathcal{L}_{m_k}$

(e) $(\forall k \in \{1,\ldots,n\})(\forall s, s' \in \Sigma^*)\ s \in \mathcal{I}_k$ *and* $P_{I_k}(s) = P_{I_k}(s') \Rightarrow s' \in \mathcal{I}_k$

(f) $(\forall k \in \{1,\ldots,n\})(\forall s, s' \in \Sigma^*)\ s \in \mathcal{I}_{m_k}$ *and* $P_{I_k}(s) = P_{I_k}(s') \Rightarrow s' \in \mathcal{I}_{m_k}$

Proof:

Points a-b:

Identical to the proof of **point a** of **Proposition 8**, after substitution.

Points c-f:

Let $k \in \{1, \ldots, n\}$, then identical to the proof of **point a** of **Proposition 8**, after substitution.

QED

7.2 Serial System Extraction: Subsystem Form

We now present a key definition for parallel interface systems. As the event set of each *low level* is mutually exclusive from the event sets of the other *low levels,* we can consider the *parallel interface system* as n *serial interface systems* by choosing one *low level* and ignoring the others. This will allow us to reuse our existing definitions and results for *serial interface systems.*

We are now ready to introduce the concept of *serial system extractions* for an n^{th} degree ($n \geq 1$) *parallel interface system.* For $j \in \{1, \ldots, n\}$, the j^{th} *serial system extraction* is essentially the original parallel system with the $1^{\text{st}}, \ldots, (j-1)^{\text{th}}, (j+1)^{\text{th}}, \ldots, n^{\text{th}}$ *low levels* removed. Figure 7.3 shows this conceptually.

j^{th} Serial System Extraction: Subsystem Form 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 $\Sigma := \dot{\bigcup}_{k \in \{1,\ldots,n\}} (\Sigma_{L_k} \,\dot{\cup}\, \Sigma_{R_k} \,\dot{\cup}\, \Sigma_{A_k}) \,\dot{\cup}\, \Sigma_H$, the j^{th} *serial system extraction,* denoted by *system*(j), is composed of the following elements:

$$
\begin{aligned}
G_H(j) &:= G_H ||_s G_{I_1} ||_s \ldots ||_s G_{I_{(j-1)}} ||_s G_{I_{(j+1)}} ||_s \ldots ||_s G_{I_n} \\
G_L(j) &:= G_{L_j} \\
G_I(j) &:= G_{I_j}
\end{aligned}
$$

$$
\begin{aligned}
\Sigma_H(j) &:= \dot{\bigcup}_{k \in \{1, \ldots, (j-1), (j+1), \ldots, n\}} \Sigma_{I_k} \,\dot{\cup}\, \Sigma_H \\
\Sigma_L(j) &:= \Sigma_{L_j} \\
\Sigma_R(j) &:= \Sigma_{R_j}
\end{aligned}
$$

$$\Sigma_A(j) \;:=\; \Sigma_{A_j}$$

$$\begin{aligned}\Sigma(j) \;&:=\; \Sigma_H(j) \dot{\cup} \Sigma_L(j) \dot{\cup} \Sigma_R(j) \dot{\cup} \Sigma_A(j) \\ &=\; \Sigma \;-\; \dot{\cup}_{k \in \{1,\ldots,(j-1),(j+1),\ldots,n\}} \Sigma_{L_k}\end{aligned}$$

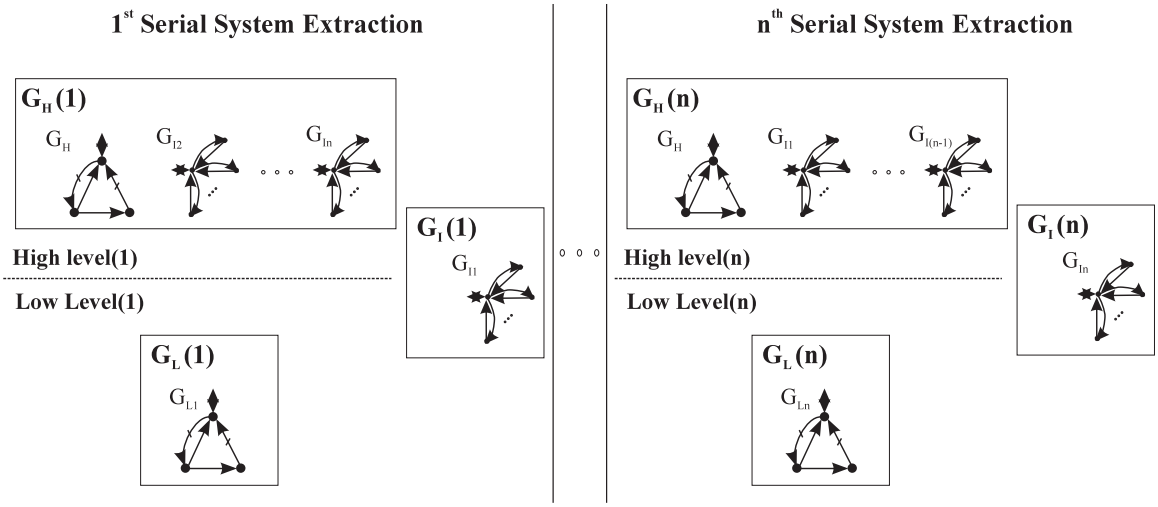


Figure 7.3: The Serial System Extraction

In the above definition, we defined *serial system extractions* in terms of a parallel subsystem based system. As we did for the serial case, we will define in Chapter 8 a general form for parallel systems. We will also provide a corresponding general form definition for *serial system extractions.* We will simply refer to the j^{th} *serial system extraction,* as the type of the parallel system will make clear which definition is intended.

7.3 Interface Properties

We now present some important definitions that are analogous to equivalent definitions for the serial case. We then present some related propositions.

7.3.1 Parallel Interface Definitions

In this section we present a set of properties that are equivalent to their serial interface counterparts. They all involve interpreting the parallel system as n serial systems by using

the *serial system extraction* definition.

Interface Consistent: 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* with respect to alphabet partition $\Sigma := \dot{\bigcup}_{k \in \{1,\ldots,n\}} (\Sigma_{L_k} \,\dot{\cup}\, \Sigma_{R_k} \,\dot{\cup}\, \Sigma_{A_k}) \,\dot{\cup}\, \Sigma_H$, if:

$(\forall j \in \{1, \ldots, n\})$ The j^{th} *serial system extraction* of the system is *serial interface consistent.*

Interface Strict Marking: 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 strict marking* with respect to the alphabet partition
$\Sigma := \dot{\bigcup}_{k \in \{1,\ldots,n\}} (\Sigma_{L_k} \,\dot{\cup}\, \Sigma_{R_k} \,\dot{\cup}\, \Sigma_{A_k}) \,\dot{\cup}\, \Sigma_H$, if:

$(\forall j \in \{1, \ldots, n\})$ The j^{th} *serial system extraction* of the system is *serial interface strict marking.*

Level-wise Nonblocking: 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* with respect to the alphabet partition
$\Sigma := \dot{\bigcup}_{k \in \{1,\ldots,n\}} (\Sigma_{L_k} \,\dot{\cup}\, \Sigma_{R_k} \,\dot{\cup}\, \Sigma_{A_k}) \,\dot{\cup}\, \Sigma_H$, if:

$(\forall j \in \{1, \ldots, n\})$ The j^{th} *serial system extraction* of the system is *serial level-wise nonblocking.*

7.3.2 Related Propositions

Now that we have the above definitions, we can present several related propositions that establish properties about the parallel system that will be useful in later proofs.

Our first proposition uses the *interface consistent* definition to establish the event set that the DES which make up an n^{th} degree ($n \geq 1$) *parallel interface system* are defined

over. This is useful for defining the languages of a DES created by the synchronous product of one or more of these DES.

Proposition 21 *If the n^{th} degree ($n \geq 1$)* parallel interface system *composed of DES G_H, $G_{L_1}, \ldots, G_{L_n}, G_{I_1}, \ldots, G_{I_n}$, is* interface consistent *with respect to the alphabet partition $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}) \dot{\cup} \Sigma_H$ then DES G_H is defined over event set Σ_{IH}, DES G_{I_j} is defined over event set Σ_{I_j}, and DES G_{L_j} is defined over event set Σ_{IL_j}, where $j \in \{1, \ldots, n\}$.*

Proof: See page 104.

We are now ready to state the proposition below which establishes useful properties for often used languages.

Proposition 22 *If the n^{th} degree ($n \geq 1$)* parallel interface system *composed of DES G_H, $G_{L_1}, \ldots, G_{L_n}, G_{I_1}, \ldots, G_{I_n}$, is* interface consistent *with respect to the alphabet partition $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}) \dot{\cup} \Sigma_H$ then, for all $j \in \{1, \ldots, n\}$, the following is true:*

(i) Languages $\mathcal{H}$, $\mathcal{L}_j$, and $\mathcal{I}_j$ are closed.

(ii) $\mathcal{H}_m \subseteq \mathcal{H}$, $\mathcal{L}_{m_j} \subseteq \mathcal{L}_j$, and $\mathcal{I}_{m_j} \subseteq \mathcal{I}_j$

Proof: See page 106.

We now present a proposition that will aid in the use of *serial system extractions* in proofs. The proposition interprets terminology for the j^{th} *serial system extraction* (a *serial interface system*) in terms of the original parallel system.

Before we can present the proposition, we need to first define (for use in the proposition) a new natural projection, P_j, to map strings from Σ^* (the event set of the given parallel system) to strings from $\Sigma(j)^*$ (the event set of the j^{th} extracted system of the given parallel system). It is defined as follows:

$$P_j : \Sigma^* \rightarrow \Sigma(j)^*$$

Proposition 23 *If the n^{th} degree ($n \geq 1$)* parallel interface system *composed of DES G_H, $G_{L_1}, \ldots, G_{L_n}$, $G_{I_1}, \ldots, G_{I_n}$, is* interface consistent *with respect to the alphabet partition $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}) \dot{\cup} \Sigma_H$, then for the j^{th}* serial system extraction, system(j)*, the following is true:*

(i) The flat system *is:* $G(j) = G_H||_s G_{L_j}||_s G_{I_1}||_s \ldots ||_s G_{I_n}$

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

(iii) 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* $P_I(j)^{-1} = P_j \cdot P_{I_j}^{-1}$

(iv) The event set of $G_H(j)$ *is* $\Sigma_{IH}(j)$, *the event set of* $G_L(j)$ *is* $\Sigma_{IL}(j)$, *and the event set of* $G_I(j)$ *is* $\Sigma_I(j)$.

(v) The indicated languages satisfy the following statements:

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

(vi) Languages $\mathcal{H}(j)$, $\mathcal{L}(j)$, *and* $\mathcal{I}(j)$ *are closed.*

(vii) $\mathcal{H}_m(j) \subseteq \mathcal{H}(j)$, $\mathcal{L}_m(j) \subseteq \mathcal{L}(j)$, *and* $\mathcal{I}_m(j) \subseteq \mathcal{I}(j)$

Proof: See page 107.

We close this section by noting that after examining the definition of the j^{th} *serial system extraction* and the above proposition, we see that for $n = 1$, a *parallel interface system* reduces to a single *serial interface system.* We thus see that a *serial interface system* is a special case of *parallel interface systems.* We will now speak of *parallel interface systems* and their definitions as the general case for bi-level interface systems.

7.4 Parallel Nonblocking Theorem and Propositions

We will now present **Propositions 24-26**, followed by our main result for this chapter, **Theorem 3**. The following propositions are analogous to their serial case counterparts.

Parallel Low Level Nonblocking Proposition

Our first proposition is analogous to **Proposition 11** for the serial case. It asserts that a string s accepted by the system, can always be extended to a string accepted by the system, and marked by all *low levels.* In other words, the *low levels* are not dependent on high level events to reach a marked state.

Proposition 24 *If the* n^{th} *degree* ($n \geq 1$) parallel interface system *composed of DES* G_H, $G_{L_1}, \ldots, G_{L_n}$, $G_{I_1}, \ldots, G_{I_n}$, *is* level-wise nonblocking *and* interface consistent *with respect to the alphabet partition* $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}) \dot{\cup} \Sigma_H$, *then*

$(\forall s \in \mathcal{H} \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_j \cap \mathcal{I}_j)])$
$\quad (\exists l \in \Sigma_{IL}^*)$ *s.t.* $(sl \in \mathcal{H} \cap [\cap_{j\in\{1,2,\ldots,n\}}(\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})])$

Proof: See page 112.

7.4.1 Event Agreement Propositions

We group the last two propositions together as **Proposition 26** builds upon **Proposition 25**. The first proposition asserts that any string accepted by the system can always be extended by a string marked by the *high level.* The reader should note that this string is not necessarily accepted by any *low levels.*

Proposition 25 *If the* n^{th} *degree* ($n \geq 1$) parallel interface system *composed of DES* G_H, $G_{L_1}, \ldots, G_{L_n}$, $G_{I_1}, \ldots, G_{I_n}$, *is* level-wise nonblocking *and* interface consistent *with respect to the alphabet partition* $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}) \dot{\cup} \Sigma_H$, *then*

$(\forall s \in \mathcal{H} \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_j \cap \mathcal{I}_j)])$
$\quad (\exists h \in \Sigma_{IH}^*)$ $(sh \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}\mathcal{I}_{m_j}])$

Proof: See page 115.

Our last proposition is analogous to **Proposition 15** for the serial case. It asserts that if string h extends string s such that sh is acceptable to the *high level*, then a string u can be constructed such that u has a high level image equal to h, and that su is marked by the system. In other words, we can use string h as a basis to construct string u by adding low level events so that each *low level subsystem* will accept the request and *answer*

event contained in h. As these events are common to both levels, they must agree on their occurrence.

Proposition 26 *If the n^{th} degree ($n \geq 1$)* parallel interface system *composed of DES G_H, $G_{L_1}, \ldots, G_{L_n}$, $G_{I_1}, \ldots, G_{I_n}$, is* level-wise nonblocking *and* interface consistent *with respect to the alphabet partition $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}) \ \dot{\cup} \Sigma_H$, then*

$(\forall s \in \mathcal{H} \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})])(\forall h \in \Sigma_{IH}^*)$

$(sh \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}\mathcal{I}_{m_j}]) \Rightarrow \ (\exists u \in \Sigma^*) \ s.t. \ (su \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})]) \wedge (P_{IH}(u) = h)$

Proof: See page 117.

7.4.2 Parallel Nonblocking Theorem

We are now ready to present our nonblocking theorem for *parallel interface systems.* It states that, to verify if a parallel system is nonblocking, it is sufficient to check that each of its *serial system extractions* is *serial level-wise nonblocking* and *serial interface consistent.* As the *level-wise nonblocking* and *interface consistent* definitions can be evaluated by examining only one level (the *high level* or one of the *low levels*) of our system at a time, we now have a means of verifying nonblocking of our parallel system using local checks.

Theorem 3 *If the n^{th} degree ($n \geq 1$)* parallel interface system *composed of DES G_H, $G_{L_1}, \ldots, G_{L_n}, G_{I_1}, \ldots, G_{I_n}$, is* level-wise nonblocking *and* interface consistent *with respect to the alphabet partition $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}) \dot{\cup} \Sigma_H$, then*

$L(G) = \overline{L_m(G)}$, *where* $\quad G = G_H||_s G_{L_1}||_s \ldots ||_s G_{L_n}||_s G_{I_1}||_s \ldots ||_s G_{I_n}$

Proof:

Assume system is *level-wise nonblocking* and *interface consistent.* **(1)**

As $\overline{L_m(G)} \subseteq L(G)$ is automatic, it suffices to show $L(G) \subseteq \overline{L_m(G)}$

Let $s \in L(G) = \mathcal{H} \cap [\cap_{w\in\{1,\ldots,n\}}(\mathcal{L}_w \cap \mathcal{I}_w)]$ **(2)**

We will now show this implies $s \in \overline{L_m(G)}$

It is sufficient to show: $(\exists u \in \Sigma^*)\ su \in L_m(G) = \mathcal{H}_m \cap [\cap_{w\in\{1,\ldots,n\}}(\mathcal{L}_{m_w} \cap \mathcal{I}_{m_w})]$

Our first step is to show that we can construct a low level string, accepted by the *high level*, and that will bring all n *low levels* to a marked state. We can achieve this immediately by applying **Proposition 24** and conclude:

$$(\exists l \in \Sigma_{IL}^*) \text{ s.t. } (sl \in \mathcal{H} \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})]) \quad \textbf{(3)}$$

Our next step will be to show that we can construct a string $u' \in \Sigma^*$ such that $slu' \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})]$.

To achieve this, we will apply **Proposition 26**. Before we can apply the proposition, we must first construct a suitable $h \in \Sigma_{IH}^*$.

We first note that **(3)** implies that $sl \in \mathcal{H} \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_j \cap \mathcal{I}_j)])$. We can now apply **Proposition 25**, taking sl to be string s in that proposition, and conclude:

$$(\exists h \in \Sigma_{IH}^*) \ (slh \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}\mathcal{I}_{m_j}]) \quad \textbf{(4)}$$

Combining with **(3)**, we can now apply **Proposition 26**, taking sl to be string s in that proposition, and conclude:

$$(\exists u' \in \Sigma^*) \text{ s.t. } (slu' \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})]$$

We take string $u = lu'$ and we have $su \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})] = L_m(G)$, as required.

QED

7.5 Proofs of Selected Propositions

In order to make this work more readable, the proofs of some propositions in this chapter were not given as the propositions were introduced. They will now be presented in the following sections.

7.5.1 Proof of Proposition 21

Proof for **Proposition 21** on page 100: *If* 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 *with respect to the alphabet partition* $\Sigma := \dot\cup_{k\in\{1,\ldots,n\}}(\Sigma_{L_k}\dot\cup\Sigma_{R_k}\dot\cup\Sigma_{A_k}) \ \dot\cup \ \Sigma_H$ *then DES* G_H *is defined over event set* Σ_{IH}, *DES* G_{I_j} *is defined over event set* Σ_{I_j}, *and DES* G_{L_j} *is defined over event set* Σ_{IL_j}, *where* $j \in \{1, \ldots, n\}$.

Proof:

Assume that the n^{th} degree ($n \geq 1$) *parallel interface system* is *interface consistent* with respect to the alphabet partition.

We will now show this implies that DES G_H is defined over event set Σ_{IH}, DES G_{I_j} is defined over event set Σ_{I_j}, and that DES G_{L_j} is defined over event set Σ_{IL_j}, where $j \in \{1, \ldots, n\}$.

We first note that **(1)** implies that $(\forall j \in \{1, \ldots, n\})$ the j^{th} *serial system extraction*, labelled *system*(j), of the system is *serial interface consistent.*

This allows us to conclude that: $(\forall j \in \{1, \ldots, n\})$ $G_H(j)$ is defined over $\Sigma_{IH}(j)$, $G_L(j)$ is defined over $\Sigma_{IL}(j)$, and that $G_I(j)$ is defined over $\Sigma_I(j)$

From **(1)**, we can now apply **Proposition 23**, **point ii** and conclude:[2]

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

We can now conclude that: $(\forall j \in \{1, \ldots, n\})$ $G_H(j)$ is defined over Σ_{IH}, $G_L(j)$ is defined over Σ_{IL_j}, and that $G_I(j)$ is defined over Σ_{I_j}. **(2)**

This implies: $(\forall j \in \{1, \ldots, n\})$ DES $G_{L_j} = G_L(j)$ is defined over Σ_{IL_j} and that $G_{I_j} = G_I(j)$ is defined
over Σ_{I_j}. **(3)**

All that remains is to show that G_H is defined over alphabet Σ_{IH}. To do this, we first need to prove the following claim.

Claim: $\Sigma_{G_H} \subseteq \Sigma_{IH}$ and $(\forall j \in \{1, \ldots, n\})\ \Sigma_{G_H} \supseteq (\Sigma_H \cup \Sigma_{I_j})$

Let $j \in \{1, \ldots, n\}$. We will now show this implies $\Sigma_{G_H} \subseteq \Sigma_{IH}$ and $\Sigma_{G_H} \supseteq (\Sigma_H \cup \Sigma_{I_j})$.

We start by noting $G_H(j) := G_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}}||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n}$. By the definition of the $||_s$ operator, we know $\Sigma_{G_H(j)} = \Sigma_{G_H} \cup [\cup_{k \in \{1, \ldots, (j-1), (j+1), \ldots, n\}} \Sigma_{G_{I_k}}]$. This implies that $\Sigma_{G_H} \subseteq \Sigma_{G_H(j)}$. As $\Sigma_{G_H(j)} = \Sigma_{IH}$ (from **(2)**), we immediately have $\Sigma_{G_H} \subseteq \Sigma_{IH}$.

From **(3)**, we have $\Sigma_{G_H(j)} = \Sigma_{G_H} \cup [\cup_{k \in \{1, \ldots, (j-1), (j+1), \ldots, n\}} \Sigma_{I_k}]$.

We now note that $\Sigma_{I_j} \subseteq \Sigma_{IH}$ but $\Sigma_{I_j} \cap [\cup_{k \in \{1, \ldots, (j-1), (j+1), \ldots, n\}} \Sigma_{I_k}] = \emptyset$ because of our event partition.

This implies: $\Sigma_{I_j} \subseteq \Sigma_{G_H}$

[2] **Proposition 23**, **point iv** requires the proposition we are currently proving, but **point ii** of **Proposition 23** is independent of **point iv** and does not.

We next note that $\Sigma_H \subseteq \Sigma_{IH}$ but $\Sigma_H \cap [\cup_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}}\Sigma_{I_k}] = \emptyset$ because of our event partition.

This implies: $\Sigma_H \subseteq \Sigma_{G_H}$

We thus have $\Sigma_{G_H} \supseteq (\Sigma_H \cup \Sigma_{I_j})$ as required.

Claim proven.

From the claim, we have $\Sigma_{G_H} \subseteq \Sigma_{IH}$. To show that $\Sigma_{G_H} = \Sigma_{IH}$ we now only have to show $\Sigma_{G_H} \supseteq \Sigma_{IH}$.

From the claim, we also have $(\forall j \in \{1,\ldots,n\})\ \Sigma_{G_H} \supseteq (\Sigma_H \cup \Sigma_{I_j})$.

This implies $\Sigma_{G_H} \supseteq \Sigma_H \cup [\cup_{k\in\{1,\ldots,n\}}\Sigma_{I_k}] = \Sigma_{IH}$. We thus have $\Sigma_{G_H} = \Sigma_{IH}$.

We can now conclude that DES G_H is defined over Σ_{IH}, as required.

QED

7.5.2 Proof of Proposition 22

Proof for **Proposition 22** on page 100: If the n^{th} degree ($n \geq 1$) *parallel interface system* composed of DES G_H, $G_{L_1},\ldots,G_{L_n},G_{I_1},\ldots,G_{I_n}$, is *interface consistent* with respect to the alphabet partition
$\Sigma := \dot\cup_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot\cup \Sigma_{R_k} \dot\cup \Sigma_{A_k}) \dot\cup \Sigma_H$ then, for all $j \in \{1,\ldots,n\}$, the following is true:

(i) Languages $\mathcal{H}$, $\mathcal{L}_j$, and $\mathcal{I}_j$ are closed.

(ii) $\mathcal{H}_m \subseteq \mathcal{H}$, $\mathcal{L}_{m_j} \subseteq \mathcal{L}_j$, and $\mathcal{I}_{m_j} \subseteq \mathcal{I}_j$

Proof:

Assume system is *interface consistent* and let $j \in \{1,\ldots,n\}$. **(1)**

Will now show this implies that **Points i** and **ii** are satisfied.

We first note that by **(1)**, the system is *interface consistent.* This allows us to apply **Proposition 21** and conclude:

$$L(G_H), L_m(G_H) \subseteq \Sigma_{IH}^*,\ L(G_{L_j}), L_m(G_{L_j}) \subseteq \Sigma_{IL_j}^*, \text{ and } L(G_{I_j}), L_m(G_{I_j}) \subseteq \Sigma_{I_j}^*.$$

This tells us that languages $\mathcal{H} = P_{IH}^{-1}(L(G_H))$, $\mathcal{L}_j = P_{IL}^{-1}(L(G_{L_j}))$, $\mathcal{I}_j = P_I^{-1}(L(G_{I_j}))$, $\mathcal{H}_m = P_{IH}^{-1}(L_m(G_H))$, $\mathcal{L}_{m_j} = P_{IL}^{-1}(L_m(G_{L_j}))$, and $\mathcal{I}_{m_j} = P_I^{-1}(L_m(G_{I_j}))$ are defined.

Point i: Show that Languages $\mathcal{H}$, $\mathcal{L}_j$, and $\mathcal{I}_j$ are closed.

We now note that languages $L(G_H)$, $L(G_{L_j})$, and $L(G_{I_j})$ are closed by the definition of the closed behaviour of a DES.

We can now apply **Proposition 1** repeatedly and conclude that $\mathcal{H}$, $\mathcal{L}_j$, and $\mathcal{I}_j$ are closed, as required.

Point ii: Show that $\mathcal{H}_m \subseteq \mathcal{H}$, $\mathcal{L}_{m_j} \subseteq \mathcal{L}_j$, and $\mathcal{I}_{m_j} \subseteq \mathcal{I}_j$.

From the definition of the closed behaviour and the marked language of a DES, we can conclude that:

$$L_m(G_H) \subseteq L(G_H),\ L_m(G_{L_j}) \subseteq L(G_{L_j}),\ \text{and}\ L_m(G_{I_j}) \subseteq L(G_{I_j}).$$

Applying **Proposition 3** repeatedly, we can conclude:

$$\begin{array}{rcl} P_{IH}^{-1}(L_m(G_H)) = \mathcal{H}_m & \subseteq & \mathcal{H} = P_{IH}^{-1}(L(G_H)) \\ P_{IL}^{-1}(L_m(G_{L_j})) = \mathcal{L}_{m_j} & \subseteq & \mathcal{L}_j = P_{IL}^{-1}(L(G_{L_j})) \\ P_I^{-1}(L_m(G_{I_j})) = \mathcal{I}_{m_j} & \subseteq & \mathcal{I}_j = P_I^{-1}(L(G_{I_j})) \end{array}$$

QED

7.5.3 Proof of Proposition 23

Proof for **Proposition 23** on page 100: *If the* n^{th} *degree* ($n \geq 1$) parallel interface system *composed of DES* G_H, $G_{L_1}, \ldots, G_{L_n}$, $G_{I_1}, \ldots, G_{I_n}$, *is* interface consistent *with respect to the alphabet partition* $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \,\dot{\cup}\, \Sigma_{R_k} \,\dot{\cup}\, \Sigma_{A_k}) \,\dot{\cup}\, \Sigma_H$, *then for the* j^{th} serial system extraction, system(j), *the following is true:*

(i) The flat system *is:* $G(j) = G_H||_s G_{L_j}||_s G_{I_1}||_s \ldots ||_s G_{I_n}$

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

(iii) 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* $P_I(j)^{-1} = P_j \cdot P_{I_j}^{-1}$

(iv) The event set of $G_H(j)$ *is* $\Sigma_{IH}(j)$, *the event set of* $G_L(j)$ *is* $\Sigma_{IL}(j)$, *and the event set of* $G_I(j)$ *is* $\Sigma_I(j)$.

(v) The indicated languages satisfy the following statements:

$$\mathcal{H}(j) \;=\; P_j(\mathcal{H}) \cap [\cap_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}} P_j(\mathcal{I}_k)]$$

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

(vi) Languages $\mathcal{H}(j)$, $\mathcal{L}(j)$, and $\mathcal{I}(j)$ are closed.

(vii) $\mathcal{H}_m(j) \subseteq \mathcal{H}(j)$, $\mathcal{L}_m(j) \subseteq \mathcal{L}(j)$, and $\mathcal{I}_m(j) \subseteq \mathcal{I}(j)$

Proof:

Assume that the n^{th} degree ($n \geq 1$) *parallel interface system* is *interface consistent* with respect to the alphabet partition. **(1)**

Let *system*(j) be the j^{th} *serial system extraction* of our parallel system. **(2)**

We will now show this implies *system*(j) satisfies **points i-vii.**

Point i: Show that $G(j) = G_H||_s G_{L_j}||_s G_{I_1}||_s \ldots ||_s G_{I_n}$

By definition, the *flat system* for a *serial interface system* is defined as follows:

$$G(j) = G_H(j)||_s G_L(j)||_s G_I(j)$$

Substituting in for DES $G_H(j)$, $G_L(j)$, and $G_I(j)$) (by **(2)**) gives as required:

$$G(j) = G_H||_s G_{L_j}||_s G_{I_1}||_s \ldots ||_s G_{I_n}$$

Point ii: Show that $\Sigma_I(j) = \Sigma_{I_j}$, $\Sigma_{IH}(j) = \Sigma_{IH}$, and $\Sigma_{IL}(j) = \Sigma_{IL_j}$

$$
\begin{aligned}
\Sigma_I(j) &:= \Sigma_R(j) \dot{\cup} \Sigma_A(j), \text{ by definition.} \\
&= \Sigma_{R_j} \dot{\cup} \Sigma_{A_j}, \text{ by } \mathbf{(2)}. \\
&= \Sigma_{I_j} \\
\Sigma_{IH}(j) &:= \Sigma_H(j) \dot{\cup} \Sigma_R(j) \dot{\cup} \Sigma_A(j), \text{ by definition.} \\
&= \dot{\cup}_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}} \Sigma_{I_k} \dot{\cup} \Sigma_H \dot{\cup} \Sigma_{R_j} \dot{\cup} \Sigma_{A_j}, \text{ by } \mathbf{(2)}. \\
&= \Sigma_{IH} \\
\Sigma_{IL}(j) &:= \Sigma_L(j) \dot{\cup} \Sigma_R(j) \dot{\cup} \Sigma_A(j), \text{ by definition.} \\
&= \Sigma_{L_j} \dot{\cup} \Sigma_{R_j} \dot{\cup} \Sigma_{A_j}, \text{ by } \mathbf{(2)}. \\
&= \Sigma_{IL_j}
\end{aligned}
$$

Point iii: Show that $P_{IH}(j)^{-1} = P_j \cdot P_{IH}^{-1}$, $P_{IL}(j)^{-1} = P_j \cdot P_{IL_j}^{-1}$, and $\Sigma_I(j) = P_j \cdot P_{I_j}^{-1}$

By definition, the following natural projections for a *serial interface system* are defined as follows:

$$\begin{aligned} P_{IH}(j) : \Sigma(j)^* &\rightarrow \Sigma_{IH}(j)^* \\ P_{IL}(j) : \Sigma(j)^* &\rightarrow \Sigma_{IL}(j)^* \\ P_I(j) : \Sigma(j)^* &\rightarrow \Sigma_I(j)^* \end{aligned}$$

Substituting from **point ii**, gives:

$$\begin{aligned} P_{IH}(j) : \Sigma(j)^* &\rightarrow \Sigma_{IH}^* \\ P_{IL}(j) : \Sigma(j)^* &\rightarrow \Sigma_{IL_j}^* \\ P_I(j) : \Sigma(j)^* &\rightarrow \Sigma_{I_j}^* \end{aligned} \qquad \textbf{(3)}$$

We first examine $P_{IH}(j)$. We first note that the following natural projections are defined as $P_{IH} : \Sigma^* \rightarrow \Sigma_{IH}^*$ and $P_j : \Sigma^* \rightarrow \Sigma(j)^*$. As $\Sigma_{IH} = \Sigma_{IH}(j) \subseteq \Sigma(j)$ and $\Sigma(j) \subseteq \Sigma$ (by **(2)** and **point ii**), we can see by **(3)** and the definition of the natural projection that the diagram in Figure 7.4 commutes. Similarly, we can see that the diagram in Figure 7.5 commutes.[3]

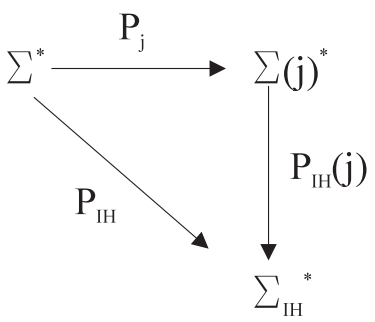


Figure 7.4: Commutative Diagram

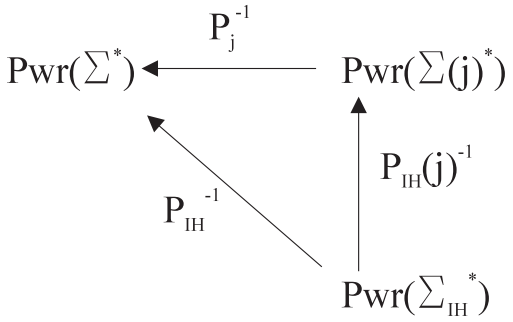


Figure 7.5: Commutative Diagram for Inverse Function

We thus have $P_{IH}^{-1} = P_j^{-1} \cdot P_{IH}(j)^{-1}$

$\Rightarrow P_j \cdot P_{IH}^{-1} = P_j \cdot P_j^{-1} \cdot P_{IH}(j)^{-1}$

$\Rightarrow P_j \cdot P_{IH}^{-1} = P_{IH}(j)^{-1}$

$\Rightarrow P_{IH}(j)^{-1} = P_j \cdot P_{IH}^{-1}$

Similarly, as $\Sigma_{IL_j} = \Sigma_{IL}(j) \subseteq \Sigma(j)$ (by **(2)** and **point ii**) and the following natural projection is defined as $P_{IL_j} : \Sigma^* \rightarrow \Sigma_{IL_j}^*$, we can conclude:

[3]To make the commutative diagram work, we extended the natural projections in the natural way to operate on subsets instead of strings.

$$P_{IL}(j)^{-1} = P_j \cdot P_{IL_j}^{-1}$$

Similarly, as $\Sigma_{I_j} = \Sigma_I(j) \subseteq \Sigma(j)$ (by **(2)** and **point ii**) and the following natural projection is defined as $P_{I_j} : \Sigma^* \to \Sigma_{I_j}^*$, we can conclude:

$$P_I(j)^{-1} = P_j \cdot P_{I_j}^{-1}$$

Point iv: Show that the event set of $G_H(j)$ is $\Sigma_{IH}(j)$, the event set of $G_L(j)$ is $\Sigma_{IL}(j)$, and the event set of $G_I(j)$ is $\Sigma_I(j)$.

From **(1)**, we have that the system is *interface consistent*. This implies that *system*(j) is *serial interface consistent*. The result follows immediately.

Point v:
First, we must show that $\mathcal{H}(j) = P_j(\mathcal{H}) \cap [\cap_{k \in \{1,\ldots,(j-1),(j+1),\ldots,n\}} P_j(\mathcal{I}_k)]$

By definition, the language $\mathcal{H}(j)$ for a *serial interface system* is defined as follows:

$$\mathcal{H}(j) = P_{IH}(j)^{-1}(L(G_H(j)))$$

Substituting using **(2)** and **point iii**, we get:

$$\mathcal{H}(j) = P_j \cdot P_{IH}^{-1}(L(G_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}}||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n})) \qquad \textbf{(4)}$$

We next note that, by **(1)**, we can apply **Proposition 21** and conclude that DES G_H is defined over event set Σ_{IH}, DES G_{I_i} is defined over event set Σ_{I_i}, and DES G_{L_i} is defined over event set Σ_{IL_i}, where $i \in \{1, \ldots, n\}$. **(5)**

This tells us that DES $G_H(j)$ is defined over Σ_{IH}.

To evaluate $L(G_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}}||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n})$, we need a natural projection $P_{I|IH_k} : \Sigma_{IH}^* \to \Sigma_{I_k}^*$, for $k \in \{1, \ldots, (j-1), (j+1), \ldots, n\}$.

After noting that $\Sigma_{I_k} \subseteq \Sigma_{IH}$ and $\Sigma_{IH} \subseteq \Sigma$, we can apply the same logic as in **point iii** and conclude:

$$P_{I|IH_k}^{-1} = P_{IH} \cdot P_{I_k}^{-1}$$

We can now evaluate $L(G_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}}||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n})$ and conclude:

$L(G_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}}||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n}) = L(G_H) \cap [\cap_{k \in \{1,\ldots,(j-1),(j+1),\ldots,n\}} P_{IH} \cdot P_{I_k}^{-1}(L(G_{I_k}))]$

Substituting into **(4)** gives:

$$\mathcal{H}(j) = P_j \cdot P_{IH}^{-1}(L(G_H) \cap [\cap_{k \in \{1,\ldots,(j-1),(j+1),\ldots,n\}} P_{IH} \cdot P_{I_k}^{-1}(L(G_{I_k}))])$$

$$\Rightarrow \mathcal{H}(j) = P_j{\cdot}P_{IH}^{-1}(L(G_H)) \cap [\cap_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}} P_j{\cdot}P_{IH}^{-1}{\cdot}P_{IH}{\cdot}P_{I_k}^{-1}(L(G_{I_k}))] \quad \textbf{(6)}$$

As $\Sigma_{I_k} \subseteq \Sigma_{IH}$, we can now apply **Proposition 6** by taking $\Sigma_a = \Sigma_{IH}$, and $\Sigma_b = \Sigma_{I_k}$ ($k \in \{1, \ldots, (j-1), (j+1), \ldots, n\}$) and thus conclude:

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

Substituting into **(6)** gives:

$$\mathcal{H}(j) = P_j \cdot P_{IH}^{-1}(L(G_H)) \cap [\cap_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}} P_j \cdot P_{I_k}^{-1}(L(G_{I_k}))]$$

$$\Rightarrow \mathcal{H}(j) = P_j\left(\mathcal{H}\right) \cap [\cap_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}} P_j\left(\mathcal{I}_k\right)]$$

Next, we must show that $\mathcal{H}_m(j) = P_j\left(\mathcal{H}_m\right) \cap [\cap_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}} P_j\left(\mathcal{I}_{m_k}\right)]$

Proof is identical to proof for $\mathcal{H}(j)$, after relabelling.

The proofs for the remaining languages for **point iv** are straightforward, and are presented together below.

$$\begin{aligned}
\mathcal{L}(j) &:= P_{IL}(j)^{-1}(L(G_L(j))), \text{ by definition.} \\
&= P_j \cdot P_{IL_j}^{-1}(L(G_{L_j})), \text{ by } \textbf{(2)} \text{ and } \textbf{point iii}. \\
&= P_j\left(\mathcal{L}_j\right) \\
\mathcal{L}_m(j) &:= P_{IL}(j)^{-1}(L_m(G_L(j))), \text{ by definition.} \\
&= P_j \cdot P_{IL_j}^{-1}(L_m(G_{L_j})), \text{ by } \textbf{(2)} \text{ and } \textbf{point iii}. \\
&= P_j\left(\mathcal{L}_{m_j}\right) \\
\mathcal{I}(j) &:= P_I(j)^{-1}(L(G_I(j))), \text{ by definition.} \\
&= P_j \cdot P_{I_j}^{-1}(L(G_{I_j})), \text{ by } \textbf{(2)} \text{ and } \textbf{point iii}. \\
&= P_j\left(\mathcal{I}_j\right) \\
\mathcal{I}_m(j) &:= P_I(j)^{-1}(L_m(G_I(j))), \text{ by definition.} \\
&= P_j \cdot P_{I_j}^{-1}(L_m(G_{I_j})), \text{ by } \textbf{(2)} \text{ and } \textbf{point iii}. \\
&= P_j\left(\mathcal{I}_{m_j}\right)
\end{aligned}$$

Points vi-vii: Show that languages $\mathcal{H}(j)$, $\mathcal{L}(j)$, and $\mathcal{I}(j)$ are closed and that $\mathcal{H}_m(j) \subseteq \mathcal{H}(j)$, $\mathcal{L}_m(j) \subseteq \mathcal{L}(j)$, and $\mathcal{I}_m(j) \subseteq \mathcal{I}(j)$

From **(2)**, we know that $G_H(j) = G_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}}||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n}$

We can now apply **Proposition 5** and conclude that language $L(G_H(j))$ is closed, and $L_m(G_H(j)) \subseteq L(G_H(j))$

We next note that languages $L(G_L(j))$, and $L(G_I(j))$ are closed as $G_L(j) = G_{L_j}$ and $G_I(j) = G_{I_j}$ (by **(2)**), and by the definition of the closed behaviour of a DES.

From the definition of the closed behaviour and the marked language of a DES, we can conclude that:

$L_m(G_L(j)) \subseteq L(G_L(j))$, and $L_m(G_I(j)) \subseteq L(G_I(j))$.

We now apply **Proposition 1** repeatedly and conclude that $\mathcal{H}(j) = P_{IH}(j)^{-1}(L(G_H(j)))$, $\mathcal{L}(j) = P_{IL}(j)^{-1}(L(G_L(j)))$, and $\mathcal{I}(j) = P_I(j)^{-1}(L(G_I(j)))$ are closed.

We next apply **Proposition 3** repeatedly, and conclude:

$$\begin{aligned} P_{IH}(j)^{-1}(L_m(G_H(j))) = \mathcal{H}_m(j) &\subseteq \mathcal{H}(j) = P_{IH}(j)^{-1}(L(G_H(j))) \\ P_{IL}(j)^{-1}(L_m(G_L(j))) = \mathcal{L}_m(j) &\subseteq \mathcal{L}(j) = P_{IL}(j)^{-1}(L(G_L(j))) \\ P_I(j)^{-1}(L_m(G_I(j))) = \mathcal{I}_m(j) &\subseteq \mathcal{I}(j) = P_I(j)^{-1}(L(G_I(j))) \end{aligned}$$

QED

7.5.4 Proof of Proposition 24

Proof for **Proposition 24** on page 102: *If the n^{th} degree ($n \geq 1$)* parallel interface system *composed of DES G_H, $G_{L_1}, \ldots, G_{L_n}, G_{I_1}, \ldots, G_{I_n}$, is* level-wise nonblocking *and* interface consistent *with respect to the alphabet partition $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}) \dot{\cup} \Sigma_H$, then*

$(\forall s \in \mathcal{H} \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_j \cap \mathcal{I}_j)])$

$(\exists l \in \Sigma_{IL}^*)$ *s.t.* $(sl \in \mathcal{H} \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})])$

Proof:

Assume system is *level-wise nonblocking* and *interface consistent.* **(1)**

Let $s \in \mathcal{H} \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_j \cap \mathcal{I}_j)]$ **(2)**

We will now show this implies: $(\exists l \in \Sigma_{IL}^*)$ s.t. $(sl \in \mathcal{H} \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})])$

To do this, we will use an inductive proof. We define $\Sigma_{IL_0}^* = \Sigma^*$, $\mathcal{L}_{m_0} = \mathcal{I}_{m_0} = \Sigma^*$ and $\mathcal{L}_{n+1} = \mathcal{I}_{n+1} = \Sigma^*$. Here we are using the fact that intersection with Σ^* is the identity operator as all other languages are subsets of Σ^*. This is to handle the boundary cases of $k = 0$ and $k = n$ in order to avoid intersection with $\emptyset$.

Claim to be proven:

For $k \in \{0, 1, \ldots, n\}$, there exist strings $l_i \in \Sigma_{IL_i}^*$, $i \in \{0, 1, \ldots, k\}$, such that:

$sl_0l_1 \ldots l_k \in \mathcal{H} \cap [\cap_{v\in\{0,1,\ldots,k\}}(\mathcal{L}_{m_v} \cap \mathcal{I}_{m_v})] \cap [\cap_{w\in\{k+1,\ldots,n+1\}}(\mathcal{L}_w \cap \mathcal{I}_w)]$ **(3)**

We will first prove the initial case $k = 0$, and then the general case of $k \in \{1, \ldots, n\}$. We can then conclude by induction that the claim has been proven.

Initial Case: $k = 0$

We take $l_0 = \epsilon \in \Sigma^*_{IL_0} = \Sigma^*$. We immediately have $sl_0 = s \in \mathcal{H} \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_j \cap \mathcal{I}_j)]$ (by **(2)**)

We have automatically $sl_0 \in (\mathcal{L}_{m_0} \cap \mathcal{I}_{m_0} \cap \mathcal{L}_{n+1} \cap \mathcal{I}_{n+1}) = \Sigma^*$

Initial case complete.

Inductive Step:

Let $k \in \{1, \ldots, n\}$. Assume there exist strings $l_i \in \Sigma^*_{IL_i}$, $i \in \{0, 1, \ldots, k-1\}$, and that they satisfy **(3)** when $k-1$ is substituted for k.

$$\Rightarrow sl_0l_1 \ldots l_{k-1} \in \mathcal{H} \cap [\cap_{v\in\{0,1,\ldots,(k-1)\}}(\mathcal{L}_{m_v} \cap \mathcal{I}_{m_v})] \cap [\cap_{w\in\{k,\ldots,n+1\}}(\mathcal{L}_w \cap \mathcal{I}_w) \qquad \textbf{(4)}$$

We will show this implies that we can construct string $l_k \in \Sigma^*_{IL_k}$, such that $sl_0l_1 \ldots l_k \in \mathcal{H} \cap [\cap_{v\in\{0,1,\ldots,k\}}(\mathcal{L}_{m_v} \cap \mathcal{I}_{m_v})] \cap [\cap_{w\in\{k+1,\ldots,n+1\}}(\mathcal{L}_w \cap \mathcal{I}_w)]$

Our approach will be to apply **Proposition 11** to *system*(k), the k^{th} *serial extraction system* of our parallel system.

We first note that **(4)** implies $sl_0l_1 \ldots l_{k-1} \in \mathcal{H} \cap [\cap_{w\in\{1,\ldots,n\}}(\mathcal{L}_w \cap \mathcal{I}_w)]$

$\Rightarrow P_k(sl_0l_1 \ldots l_{k-1}) \in P_k(\mathcal{H}) \cap P_k(\mathcal{L}_k) \cap [\cap_{w\in\{1,\ldots,n\}} P_k(\mathcal{I}_w)] = \mathcal{H}(k) \cap \mathcal{L}(k) \cap \mathcal{I}(k)$ by **Proposition 23**

We can now apply **Proposition 11** to *system* (k) by taking $P_k(sl_0l_1 \ldots l_{k-1})$ to be string s in that proposition. We thus conclude:

$$(\exists l_k \in \Sigma^*_{IL_k}) \text{ s.t. } P_k(sl_0l_1 \ldots l_{k-1})l_k \in \mathcal{H}(k) \cap \mathcal{L}_m(k) \cap \mathcal{I}_m(k)$$

We now note that $P_k(l_k) = l_k$ as $\Sigma_{IL_k} \subseteq \Sigma(k)$. We can thus conclude:

$$P_k(sl_0l_1 \ldots l_{k-1}l_k) \in \mathcal{H}(k) \cap \mathcal{L}_m(k) \cap \mathcal{I}_m(k) \qquad \textbf{(5)}$$

We will now show that this implies:

$$sl_0l_1 \ldots l_{k-1}l_k \in \mathcal{H} \cap [\cap_{w\in\{1,\ldots,k-1,k+1,\ldots,n\}}\mathcal{I}_w] \cap \mathcal{L}_{m_k} \cap \mathcal{I}_{m_k}$$

Substituting into **(5)** for $\mathcal{H}(k)$, $\mathcal{L}_m(k)$, and $\mathcal{I}_m(k)$ (by **Proposition 23**), we have:

$$P_k(sl_0l_1 \ldots l_{k-1}l_k) \in P_k \cdot P^{-1}_{IH}(L(G_H)) \cap [\cap_{w\in\{1,\ldots,k-1,k+1,\ldots,n\}} P_k \cdot P^{-1}_{I_w}(L(G_{I_w}))] \cap P_k \cdot P^{-1}_{IL_k}(L_m(G_{L_k})) \cap P_k \cdot P^{-1}_{I_k}(L_m(G_{I_k})) \qquad \textbf{(6)}$$

From **Proposition 23**, we have $\Sigma_{IH} \subseteq \Sigma(k)$. We can now apply **Corollary 2** by taking $\Sigma_a = \Sigma(k)$, $\Sigma_b = \Sigma_{IH}$, and $L_b = L(G_H)$ and thus conclude:

$$sl_0l_1 \dots l_{k-1}l_k \in P_{IH}^{-1}(L(G_H)) = \mathcal{H} \quad \textbf{(7)}$$

Similarly, we have $\Sigma_{I_w} \subseteq \Sigma(k)$ for $w \in \{1, \dots, k-1, k+1, \dots, n\}$. We can now apply **Corollary 2** by taking $\Sigma_a = \Sigma(k)$, $\Sigma_b = \Sigma_{I_w}$, and $L_b = L(G_{I_w})$ and thus conclude:

$$sl_0l_1 \dots l_{k-1}l_k \in P_{I_w}^{-1}(L(G_{I_w})) = \mathcal{I}_w \quad \textbf{(8)}$$

Similarly, we have $\Sigma_{IL_k} \subseteq \Sigma(k)$. We can now apply **Corollary 2** by taking $\Sigma_a = \Sigma(k)$, $\Sigma_b = \Sigma_{IL_k}$, and $L_b = L_m(G_{L_k})$ and thus conclude:

$$sl_0l_1 \dots l_{k-1}l_k \in P_{IL_k}^{-1}(L_m(G_{L_k})) = \mathcal{L}_{m_k} \quad \textbf{(9)}$$

Similarly, we have $\Sigma_{I_k} \subseteq \Sigma(k)$. We can now apply **Corollary 2** by taking $\Sigma_a = \Sigma(k)$, $\Sigma_b = \Sigma_{I_k}$, and $L_b = L_m(G_{I_k})$ and thus conclude:

$$sl_0l_1 \dots l_{k-1}l_k \in P_{IL_k}^{-1}(L_m(G_{I_k})) = \mathcal{I}_{m_k}$$

Combining with **(7)** - **(9)**, we thus have:

$$sl_0l_1 \dots l_{k-1}l_k \in \mathcal{H} \cap [\cap_{w \in \{1,\dots,k-1,k+1,\dots,n\}} \mathcal{I}_w] \cap \mathcal{L}_{m_k} \cap \mathcal{I}_{m_k} \quad \textbf{(10)}$$

We also have automatically $sl_0l_1 \dots l_{k-1}l_k \in (\mathcal{L}_{n+1} \cap \mathcal{I}_{n+1}) = \Sigma^*$ **(11)**

We will now show that

$$sl_0l_1 \dots l_{k-1}l_k \in \mathcal{H} \cap [\cap_{v \in \{0,1,\dots,k\}} (\mathcal{L}_{m_v} \cap \mathcal{I}_{m_v})] \cap [\cap_{w \in \{k+1,\dots,n+1\}} (\mathcal{L}_w \cap \mathcal{I}_w)]$$

This means showing: $sl_0l_1 \dots l_{k-1}l_k \in \cap_{v \in \{0,1,\dots,k-1\}} (\mathcal{L}_{m_v} \cap \mathcal{I}_{m_v}) \cap [\cap_{w \in \{k+1,\dots,n\}} \mathcal{L}_w]$

As $\mathcal{L}_{m_0} = \mathcal{I}_{m_0} = \Sigma^*$, $sl_0l_1 \dots l_{k-1}l_k \in \mathcal{L}_{m_0} \cap \mathcal{I}_{m_0}$ is automatic. **(12)**

We next note that by **(4)**, we have $sl_0l_1 \dots l_{k-1} \in \mathcal{H} \cap [\cap_{v \in \{0,1,\dots,(k-1)\}} (\mathcal{L}_{m_v} \cap \mathcal{I}_{m_v})] \cap [\cap_{w \in \{k+1,\dots,n\}} \mathcal{L}_w]$

From **(1)** we have $\Sigma_{IL_k} \cap \Sigma_{IL_v} = \emptyset$, for $v \in \{1, \dots, (k-1)\}$. As $l_k \in \Sigma_{IL_k}$, we have $P_{IL_v}(l_k) = \epsilon$. This implies $P_{IL_v}(sl_0l_1 \dots l_{k-1}l_k) = P_{IL_v}(sl_0l_1 \dots l_{k-1})$. We can now apply **Proposition 20**, **point d**, and conclude:

$$sl_0l_1 \dots l_{k-1}l_k \in \mathcal{L}_{m_v} \quad \textbf{(13)}$$

Similarly we have $P_{I_v}(sl_0l_1 \dots l_{k-1}l_k) = P_{I_v}(sl_0l_1 \dots l_{k-1})$, for $v \in \{1, \dots, (k-1)\}$. We can now apply **Proposition 20**, **point f**, and conclude:

$$sl_0l_1 \dots l_{k-1}l_k \in \mathcal{I}_{m_v} \quad \textbf{(14)}$$

Similarly we have $P_{IL_w}(sl_0l_1 \ldots l_{k-1}l_k) = P_{IL_w}(sl_0l_1 \ldots l_{k-1})$, for $w \in \{k+1, \ldots, n\}$. We can now apply **Proposition 20**, **point c**, and conclude:

$sl_0l_1 \ldots l_{k-1}l_k \in \mathcal{L}_w$

Combining with **(12)** - **(14)**, we have:

$sl_0l_1 \ldots l_{k-1}l_k \in \cap_{v\in\{0,1,\ldots,k-1\}}(\mathcal{L}_{m_v} \cap \mathcal{I}_{m_v}) \cap [\cap_{w\in\{k+1,\ldots,n\}}\mathcal{L}_w]$

Combining with **(10)** and **(11)**, we have:

$sl_0l_1 \ldots l_{k-1}l_k \in \mathcal{H} \cap [\cap_{v\in\{0,1,\ldots,k\}}(\mathcal{L}_{m_v} \cap \mathcal{I}_{m_v})] \cap [\cap_{w\in\{k+1,\ldots,n+1\}}(\mathcal{L}_w \cap \mathcal{I}_w)]$, as required.

Inductive step complete.

We have now proven the **Initial case** and the **Inductive step**. We now conclude that the **Claim** is true, by induction.

Taking $k = n$ and using fact that $l_0 = \epsilon$, we thus can conclude there exists strings $l_i \in \Sigma^*_{IL_i}$, $i \in \{1, \ldots, n\}$, such that: $sl_1 \ldots l_n \in \mathcal{H} \ \cap \ [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})]$

We thus take $l = l_1 \ldots l_n$ and we have $l \in \Sigma^*_{IL}$ and $sl \in \mathcal{H} \ \cap \ [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})]$, as required.

QED

7.5.5 Proof of Proposition 25

Proof for **Proposition 25** on page 102: *If the* n^{th} *degree* ($n \geq 1$) parallel interface system *composed of DES* G_H, $G_{L_1}, \ldots, G_{L_n}$, $G_{I_1}, \ldots, G_{I_n}$, *is* level-wise nonblocking *and* interface consistent *with respect to the alphabet partition* $\Sigma := \dot\cup_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \ \dot\cup \ \Sigma_{R_k} \ \dot\cup \ \Sigma_{A_k}) \ \dot\cup \ \Sigma_H$, *then*

$(\forall s \in \mathcal{H} \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_j \cap \mathcal{I}_j)])$
$\quad (\exists h \in \Sigma^*_{IH}) \ (sh \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}\mathcal{I}_{m_j}])$

Proof:

Assume system is *level-wise nonblocking* and *interface consistent.* **(1)**

Let $s \in \mathcal{H} \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_j \cap \mathcal{I}_j)]$ **(2)**

We will now show this implies:

$$(\exists h \in \Sigma_{IH}^*)\ \ (sh \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}\mathcal{I}_{m_j}])$$

We will start by examining the 1^{st} *serial extraction system* of the parallel system. We will show that we can construct a string $h \in \Sigma_{IH}^*$ with the property that $P_1(s)h \in \mathcal{H}_m(1) \cap \mathcal{I}_m(1)$.

We first note that from **(2)**, we have $s \in \mathcal{H} \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_j \cap \mathcal{I}_j)]$

From **Proposition 23**, we thus have:

$$P_1(s) \in P_1(\mathcal{H}) \cap [\cap_{j\in\{2,\ldots,n\}} P_1(\mathcal{I}_j)] \cap P_1(\mathcal{L}_1) \cap P_1(\mathcal{I}_1) = \mathcal{H}(1) \cap \mathcal{L}(1) \cap \mathcal{I}(1) \qquad \textbf{(3)}$$

We start be noting that *system*(1) is *serial level-wise nonblocking*, as the parallel system is *level-wise nonblocking* (by **(1)**). Combining with **(3)**, we can thus apply **Point I** of the *serial level-wise nonblocking* definition and conclude:

$$(\exists h' \in \Sigma(1)^*) \text{ s.t. } P_1(s)h' \in \mathcal{H}_m(1) \cap \mathcal{I}_m(1)$$

We next note that:

$$\begin{aligned} P_{IH}(1)\,(P_1(s)h') &= P_{IH}(1)(P_1(s))P_{IH}(1)(h') \\ &= P_{IH}(1)(P_1(s))P_{IH}(1)(P_{IH}(1)(h')) \text{ as the natural projection} \\ &\quad \text{is idempotent.} \\ &= P_{IH}(1)(P_1(s)P_{IH}(1)(h')) \qquad (4) \end{aligned}$$

We can now apply **Proposition 8**, **point b**, and conclude:

$$P_1(s)P_{IH}(1)(h') \in \mathcal{H}_m(1) \qquad \textbf{(5)}$$

As $\Sigma_I(1) \subseteq \Sigma_{IH}(1)$, we can conclude by **(4)** that $P_I(1)(P_1(s)h') = P_I(1)(P_1(s)P_{IH}(1)(h'))$

We can now apply **Proposition 8**, **point f**, and conclude:

$$P_1(s)P_{IH}(1)(h') \in \mathcal{I}_m(1) \qquad \textbf{(6)}$$

We next note that as $\Sigma(1) \subseteq \Sigma$ and $\Sigma_{IH} = \Sigma_{IH}(1)$ (by **Proposition 23**), we can conclude $P_{IH}(h') = P_{IH}(1)(h')$.

Combining with **(5)** and **(6)**, we can conclude:

$$P_1(s)P_{IH}(h') \in \mathcal{H}_m(1) \cap \mathcal{I}_m(1)$$

We then take $h = P_{IH}(h')$ and we have:

$$h \in \Sigma_{IH}^* \text{ and } P_1(s)h \in \mathcal{H}_m(1) \cap \mathcal{I}_m(1) \qquad \textbf{(7)}$$

We will now show that this implies: $sh \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}\mathcal{I}_{m_j}]$

From **(7)**, substituting for $\mathcal{H}_m(1)$ and $\mathcal{I}_m(1)$ (by **Proposition 23**) and noting $P_1(h) = h$ as $h \in \Sigma_{IH} \subseteq \Sigma(1)$, we can conclude that:

$$P_1(sh) \in P_1 \cdot P_{IH}^{-1}(L_m(G_H)) \cap [\cap_{j\in\{2,\ldots,n\}} P_1 \cdot P_{I_j}^{-1}(L_m(G_{I_j}))] \cap P_1 \cdot P_{I_1}^{-1}(L_m(G_{I_1}))$$

From **Proposition 23**, we have $\Sigma_{IH} \subseteq \Sigma(1)$. We can now apply **Corollary 2** by taking $\Sigma_a = \Sigma(1)$, $\Sigma_b = \Sigma_{IH}$, and $L_b = L_m(G_H)$ and thus conclude:

$$sh \in P_{IH}^{-1}(L_m(G_H)) = \mathcal{H}_m \qquad \textbf{(8)}$$

Similarly, we have $\Sigma_{I_j} \subseteq \Sigma(1)$ for $j \in \{1, \ldots, n\}$. We can now apply **Corollary 2** by taking $\Sigma_a = \Sigma(1)$, $\Sigma_b = \Sigma_{I_j}$, and $L_b = L_m(G_{I_j})$ and thus conclude:

$$sh \in P_{I_j}^{-1}(L_m(G_{I_j})) = \mathcal{I}_{m_j}$$

Combining with **(8)**, we have:

$$sh \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}\mathcal{I}_{m_j}]$$

Combining with **(7)**, we thus have $h \in \Sigma_{IH}^*$ with the required property that $(sh \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}\mathcal{I}_{m_j}])$

QED

7.5.6 Proof of Proposition 26

Proof for **Proposition 26** on page 103: *If the n^{th} degree ($n \geq 1$)* parallel interface system *composed of DES G_H, $G_{L_1}, \ldots, G_{L_n}$, $G_{I_1}, \ldots, G_{I_n}$, is* level-wise nonblocking *and* interface consistent *with respect to the alphabet partition $\Sigma := \dot\cup_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot\cup \Sigma_{R_k} \dot\cup \Sigma_{A_k}) \ \dot\cup \Sigma_H$, then*

$(\forall s \in \mathcal{H} \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})])(\forall h \in \Sigma_{IH}^*)$

$(sh \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}\mathcal{I}_{m_j}]) \Rightarrow (\exists u \in \Sigma^*)\ s.t.\ (su \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})]) \wedge (P_{IH}(u) = h)$

Proof:

Assume system is *level-wise nonblocking* and *interface consistent.* **(1)**

Let $s \in \mathcal{H} \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})]$, $h \in \Sigma_{IH}^*$, and $sh \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}\mathcal{I}_{m_j}]$ **(2)**

We will now show this implies $(\exists u \in \Sigma^*)$ s.t. $(su \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})]) \wedge (P_{IH}(u) = h)$

To do this, we will use string h and apply **Proposition 15** iteratively, to construct n strings labelled $u_i \in \Sigma(i)^*$, $i \in \{1,\ldots,n\}$, with the properties $su_i \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}\mathcal{I}_{m_j}] \cap \mathcal{L}_{m_i}$ (i.e. string u_i takes the *high level* and the i^{th} *low level system* to a marked state) and $P_{IH}(u_i) = h$ (string h is the high level image of each string u_i). We will then use these n strings to construct the desired string u.

Iterative step:

For each $i \in \{1,\ldots,n\}$, construct u_i as follows:

To apply **Proposition 15** to *system*(i), the i^{th} *serial extraction system* of our parallel system, we must first show that $h \in \Sigma_{IH}^*$ (auto from **(2)**), $P_i(s) \in \mathcal{H}_m(i) \cap \mathcal{I}_m(i)$, and that $P_i(s) \in \mathcal{H}(i) \cap \mathcal{L}(i) \cap \mathcal{I}_m(i)$.

From **(2)**, we have $sh \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}}\mathcal{I}_{m_j}]$

As $P_i(h) = h$ since $h \in \Sigma_{IH}^* \subseteq \Sigma(i)$ (by **Proposition 23**), we can conclude:

$$P_i(sh) = P_i(s)h \in P_i(\mathcal{H}_m) \cap [\cap_{j\in\{1,\ldots,i-1,i+1,\ldots,n\}}P_i(\mathcal{I}_{m_j})] \cap P_i(\mathcal{I}_{m_i})$$

$\Rightarrow P_i(s)h \in \mathcal{H}_m(i) \cap \mathcal{I}_m(i)$ (by **Proposition 23**) **(3)**

Our last step before we can apply **Proposition 15** is to show that $P_i(s) \in \mathcal{H}(i) \cap \mathcal{L}(i) \cap \mathcal{I}_m(i)$.

From **(2)**, we have $s \in \mathcal{H} \cap [\cap_{j\in\{1,\ldots,n\}}(\mathcal{L}_j \cap \mathcal{I}_j)]$ **(4)**

From **Proposition 23**, we thus have:

$P_i(s) \in P_i(\mathcal{H}) \cap [\cap_{j\in\{1,\ldots,i-1,i+1,\ldots,n\}}P_i(\mathcal{I}_j)] \cap P_i(\mathcal{L}_i) \cap P_i(\mathcal{I}_i) = \mathcal{H}(i) \cap \mathcal{L}(i) \cap \mathcal{I}(i)$ **(5)**

We have $P_i(s) \in \mathcal{H}(i) \cap \mathcal{L}(i)$ from **(5)**, so all that remains is to show that $P_i(s) \in \mathcal{I}_m(i)$.

From **(2)**, we have $s \in \mathcal{I}_{m_i}$. This implies $P_i(s) \in P_i(\mathcal{I}_{m_i}) = \mathcal{I}_m(i)$

Combining with **(2)** **(3)**, and **(5)**, we now apply **Proposition 15** by taking $P_i(s)$ to be string s in that proposition and conclude:

$$(\exists u_i \in \Sigma(i)^*) \text{ s.t. } P_i(s)u_i \in (\mathcal{H}_m(i) \cap \mathcal{L}_m(i) \cap \mathcal{I}_m(i)) \wedge (P_{IH}(i)(u_i) = h) \quad \textbf{(6)}$$

We next note that as $\Sigma(i) \subseteq \Sigma$ and $\Sigma_{IH} = \Sigma_{IH}(i)$ (by **Proposition 23**), we can conclude $P_{IH}(u_i) = P_{IH}(i)(u_i) = h$. **(7)**

We now note that $u_i \in \Sigma(i)^*$ implies that $P_i(u_i) = u_i$. Combining with **(6)** and

substituting for $\mathcal{H}_m(i)$, $\mathcal{L}_m(i)$ (using **Proposition 23**), and $\mathcal{I}_m(i)$, we have:

$$P_i(su_i) \in P_i \cdot P_{IH}^{-1}(L_m(G_H)) \cap [\cap_{j\in\{1,\ldots,n\}} P_i \cdot P_{I_j}^{-1}(L_m(G_{I_j}))] \cap P_i \cdot P_{IL_i}^{-1}(L_m(G_{L_i})) \quad \textbf{(8)}$$

From **Proposition 23**, we have $\Sigma_{IH} \subseteq \Sigma(i)$. We can now apply **Corollary 2** by taking $\Sigma_a = \Sigma(i)$, $\Sigma_b = \Sigma_{IH}$, and $L_b = L_m(G_H)$ and thus conclude:

$$su_i \in P_{IH}^{-1}(L_m(G_H)) = \mathcal{H}_m \quad \textbf{(9)}$$

Similarly, we have $\Sigma_{I_j} \subseteq \Sigma(i)$ for $j \in \{1,\ldots,n\}$. We can now apply **Corollary 2** by taking $\Sigma_a = \Sigma(i)$, $\Sigma_b = \Sigma_{I_j}$, and $L_b = L_m(G_{I_j})$ and thus conclude:

$$su_i \in P_{I_j}^{-1}(L_m(G_{I_j})) = \mathcal{I}_{m_j} \quad \textbf{(10)}$$

Similarly, we have $\Sigma_{IL_i} \subseteq \Sigma(i)$. We can now apply **Corollary 2** by taking $\Sigma_a = \Sigma(i)$, $\Sigma_b = \Sigma_{IL_i}$, and $L_b = L_m(G_{L_i})$ and thus conclude:

$$su_i \in P_{IL_i}^{-1}(L_m(G_{L_i})) = \mathcal{L}_{m_i}$$

Combining with **(9)** and **(10)**, we have:

$su_i \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}} \mathcal{I}_{m_j}] \cap \mathcal{L}_{m_i}$, as required.

Iterative step complete.

Now that we have completed the **iterative step**, we have shown the following:

$$(\forall i \in \{1,\ldots,n\})(\exists u_i \in \Sigma(i)^*)\ \ (su_i \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}} \mathcal{I}_{m_j}] \cap \mathcal{L}_{m_i}) \wedge (P_{IH}(u_i) = h) \quad \textbf{(11)}$$

We will now use this information to construct a string $u \in \Sigma^*$ with the property:
$(su \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}} (\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})]) \wedge (P_{IH}(u) = h)$

We take u to be any string in set $\cap_{i\in\{1,\ldots,n\}} P_i^{-1}(u_i)$ **(12)**

We know that the set is non-empty for the following reasons:

- For each $i \in \{1,\ldots,n\}$, we have $u_i \in \Sigma(i)^*$ where:
  $\Sigma(i) := \Sigma_{IH} \dot\cup \Sigma_{L_i} = \Sigma - (\dot\cup_{j\in\{1,\ldots,i-1,i+1,\ldots,n\}} \Sigma_{L_j})$.
- The only events strings u_i have in common are $\sigma \in \Sigma_{IH}$.
- All strings u_i agree on common events as $P_{IH}(u_i) = h$

From **(12)**, we have $(\forall i \in \{1,\ldots,n\}) P_i(u) = u_i$. As $\Sigma_{IH} \subseteq \Sigma(i)$ (by **Proposition 23**) and $h \in \Sigma_{IH}^*$ (by **(2)**), we can conclude:

$$P_{IH}(u) = P_{IH}(u_i) = h = P_{IH}(h). \quad \textbf{(13)}$$

From **(2)**, we have: $sh \in \mathcal{H}_m \cap [\cap_{j\in\{1,\ldots,n\}} \mathcal{I}_{m_j}]$

We can now apply **Proposition 20**, **point b**, and conclude:

$$su \in \mathcal{H}_m \qquad \textbf{(14)}$$

As $\Sigma_{I_j} \subseteq \Sigma_{IH}$ for $j \in \{1, \ldots, n\}$, we can conclude by **(13)** that $P_{I_j}(u) = P_{I_j}(h)$. We can now apply **Proposition 20**, **point f**, and conclude:

$$su \in \mathcal{I}_{m_j}$$

Combining with **(14)**, we can conclude:

$$su \in \mathcal{H}_m \cap [\cap_{j \in \{1,\ldots,n\}} \mathcal{I}_{m_j}] \qquad \textbf{(15)}$$

All that remains is to show $su \in \cap_{j \in \{1,\ldots,n\}} \mathcal{L}_{m_j}$

From **(11)**, we have $su_j \in \mathcal{L}_{m_j}$ for $j \in \{1, \ldots, n\}$. From **(12)**, we have $P_j(u) = u_j = P_j(u_j)$ as $u_j \in \Sigma(j)^*$. As $\Sigma_{IL_j} \subseteq \Sigma(j)$ (by **Proposition 23**), we can conclude $P_{IL_j}(u) = P_{IL_j}(u_j)$. We can now apply **Proposition 20**, **point d**, and conclude:

$$su \in \mathcal{L}_{m_j}$$

Combining with **(15)**, we have: $su \in \mathcal{H}_m \cap [\cap_{j \in \{1,\ldots,n\}} (\mathcal{L}_{m_j} \cap \mathcal{I}_{m_j})]$

From **(13)**, we have $P_{IH}(u) = h$, as required.

QED

Chapter 8

Parallel Case: Controllability

Now that we have discussed nonblocking in the parallel interface setting, we now consider controllability. In the remainder of this chapter, we will define our setting and notation and then present some supporting propositions, followed by the serial controllability theorem.

8.1 Definitions and Notation

We now present some definitions and notation that will be useful in simplifying proofs. As in the serial case, we need to decompose the n^{th} degree ($n \geq 1$) *parallel interface system* into its plant and supervisor components. For the remainder of this section, the index j is defined to be $j \in \{1, \ldots, n\}$.

We now define the *high level plant* to to be $\mathcal{G}_H$, and the *high level supervisor* to be $\mathcal{S}_H$ (both defined over Σ_{IH}). Similarly, the j^{th} *low level plant* and *supervisor* are $\mathcal{G}_{L_j}$ and $\mathcal{S}_{L_j}$(defined over Σ_{IL_j}). To be consistent with our definitions in Chapter 7, we define the following identities for the *high level subsystem* and j^{th} *low level subsystem* as follows:

$$G_H := \mathcal{G}_H ||_s \mathcal{S}_H \qquad\qquad G_{L_j} := \mathcal{G}_{L_j} ||_s \mathcal{S}_{L_j}$$

We now have two ways to describe our system for the parallel case, depending on the level of detail required. We will call the original method described in Chapter 7 in terms of an *interface* and *high* and *low subsystems*, the *parallel subsystem based form.* This form is useful as it simplifies nonblocking definitions and proofs. We call the above method, given in terms of an interface and plants and supervisors, the *parallel general form* as the *parallel*

subsystem based form can be recovered by applying the above identities. When we refer to terms applicable to both forms (e.g. the *high level*), we will simply state the term, allowing the type of the system to make our meaning clear.

Our next step is to define the *flat supervisor* and *plant* for our system.

$$\textbf{Plant} := \mathcal{G}_H||_s\mathcal{G}_{L_1}||_s \ldots ||_s\mathcal{G}_{L_n} \qquad \textbf{Sup} := \mathcal{S}_H||_s\mathcal{S}_{L_1}||_s \ldots ||_s\mathcal{S}_{L_n}||_sG_{I_1}||_s \ldots ||_sG_{I_n}$$

We next want to express the languages of **Plant** and **Sup** in terms of their components. To do this, we need to first define the following useful languages::

$$\begin{aligned} \mathbf{H} &:= P_{IH}^{-1}L(\mathcal{G}_H), \qquad \mathbf{H}_{\mathcal{S}} := P_{IH}^{-1}L(\mathcal{S}_H), \qquad \subseteq \Sigma^* \\ \mathbf{L}_j &:= P_{IL_j}^{-1}L(\mathcal{G}_{L_j}), \qquad \mathbf{L}_{\mathcal{S}_j} := P_{IL_j}^{-1}L(\mathcal{S}_{L_j}), \qquad \subseteq \Sigma^* \end{aligned}$$

We can now express the languages of **Plant** and **Sup** as follows:

$$L(\textbf{Plant}) = \mathbf{H} \cap [\cap_{k\in\{1,\ldots,n\}}\mathbf{L}_k] \qquad L(\textbf{Sup}) = \mathbf{H}_{\mathcal{S}} \cap [\cap_{k\in\{1,\ldots,n\}}(\mathbf{L}_{\mathcal{S}_k} \cap \mathcal{I}_k)]$$

This allows us to present the proposition below that collects together several similar propositions. As it will be common in the proofs in this report to show that membership in languages such as **H** are dependent only on events in specific subsets (for **H**, events in subset Σ_{IH}), this proposition will be very useful.

Proposition 27

(a) $(\forall s, s' \in \Sigma^*)\ s \in \mathbf{H}$ *and* $P_{IH}(s) = P_{IH}(s') \Rightarrow s' \in \mathbf{H}$

(b) $(\forall s, s' \in \Sigma^*)\ s \in \mathbf{H}_{\mathcal{S}}$ *and* $P_{IH}(s) = P_{IH}(s') \Rightarrow s' \in \mathbf{H}_{\mathcal{S}}$

(c) $(\forall k \in \{1, \ldots, n\})(\forall s, s' \in \Sigma^*)\ s \in \mathbf{L}_k$ *and* $P_{IL_k}(s) = P_{IL_k}(s') \Rightarrow s' \in \mathbf{L}_k$

(d) $(\forall k \in \{1, \ldots, n\})(\forall s, s' \in \Sigma^*)\ s \in \mathbf{L}_{\mathcal{S}_k}$ *and* $P_{IL_k}(s) = P_{IL_k}(s') \Rightarrow s' \in \mathbf{L}_{\mathcal{S}_k}$

Proof:

Points a-b:

Identical to the proof of **point a** of **Proposition 8**, after substitution.

Points c-d:

Let $k \in \{1, \ldots, n\}$, then identical to the proof of **point a** of **Proposition 8**, after substitution.

QED

8.2 Serial System Extraction: General Form

We now extend the definition of the j^{th} *serial system extraction*, defined in Section 7.2, to operate on the general form of an n^{th} degree ($n \geq 1$) *parallel interface system*. The subsystem form of the definition can be recovered by using the identities $G_H(j) = \mathcal{G}_H(j)||_s\mathcal{S}_H(j)$, $G_L(j) = \mathcal{G}_L(j)||_s\mathcal{S}_L(j)$. Normally, we will simply refer to the j^{th} *serial system extraction,* as the type of the parallel system will make clear which definition is intended.

j^{th} Serial System Extraction: General Form For the n^{th} degree ($n \geq 1$) *parallel interface system* composed of DES $\mathcal{G}_H$, $\mathcal{G}_{L_1}, \ldots, \mathcal{G}_{L_n}$, $\mathcal{S}_H$, $\mathcal{S}_{L_1}, \ldots, \mathcal{S}_{L_n}$, $G_{I_1}, \ldots, G_{I_n}$, with alphabet partition $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}) \dot{\cup} \Sigma_H$, the j^{th} *serial system extraction,* denoted by *system*(j), is composed of the following elements:

$$
\begin{aligned}
\mathcal{G}_H(j) &:= \mathcal{G}_H||_sG_{I_1}||_s \ldots ||_sG_{I_{(j-1)}}||_sG_{I_{(j+1)}}||_s \ldots ||_sG_{I_n} \\
\mathcal{S}_H(j) &:= \mathcal{S}_H \\
\mathcal{G}_L(j) &:= \mathcal{G}_{L_j} \\
\mathcal{S}_L(j) &:= \mathcal{S}_{L_j} \\
G_I(j) &:= G_{I_j}
\end{aligned}
$$

$$
\begin{aligned}
\Sigma_H(j) &:= \dot{\cup}_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}}\Sigma_{I_k} \dot{\cup} \Sigma_H \\
\Sigma_L(j) &:= \Sigma_{L_j} \\
\Sigma_R(j) &:= \Sigma_{R_j} \\
\Sigma_A(j) &:= \Sigma_{A_j}
\end{aligned}
$$

$$
\begin{aligned}
\Sigma(j) &:= \Sigma_H(j) \dot{\cup} \Sigma_L(j) \dot{\cup} \Sigma_R(j) \dot{\cup} \Sigma_A(j) \\
&= \Sigma - \dot{\cup}_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}}\Sigma_{L_k}
\end{aligned}
$$

8.3 Controllability Properties

The goal in this chapter is to develop a means of verifying that our system's *flat supervisor* is controllable for the *flat plant* that uses only local checks. To achieve this, we will extend the *serial level-wise controllability* definition to the parallel system case by using the *serial system extraction* definition.

Level-wise Controllable: The n^{th} degree ($n \geq 1$) *parallel interface system* composed of DES $\mathcal{G}_H,\ \mathcal{G}_{L_1}, \ldots, \mathcal{G}_{L_n}, \mathcal{S}_H, \mathcal{S}_{L_1}, \ldots, \mathcal{S}_{L_n}, G_{I_1}, \ldots, G_{I_n}$, is *level-wise controllable* with respect to alphabet partition $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup}\ \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}) \dot{\cup} \Sigma_H$, if:

$(\forall j \in \{1, \ldots, n\})$ The j^{th} *serial system extraction* of the system is *serial level-wise controllability.*

Now that we have the above definitions, we can present several related propositions that establish properties about the parallel system that will be useful in later proofs.

Our first proposition uses the *level-wise controllable* definition to establish the event set that the DES that make up an n^{th} degree ($n \geq 1$) *parallel interface system* are defined over. This is useful for defining the languages of a DES created by the synchronous product of one or more of these DES.

Proposition 28 *If n^{th} degree ($n \geq 1$)* parallel interface system *composed of DES $\mathcal{G}_H$, $\mathcal{G}_{L_1}, \ldots, \mathcal{G}_{L_n}, \mathcal{S}_H, \mathcal{S}_{L_1}, \ldots, \mathcal{S}_{L_n}, G_{I_1}, \ldots, G_{I_n}$, is* level-wise controllable *with respect to the alphabet partition $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}) \dot{\cup} \Sigma_H$ then DES $\mathcal{G}_H$ and $\mathcal{S}_H$ are defined over event set Σ_{IH}, DES G_{I_j} is defined over event set Σ_{I_j}, and DES $\mathcal{G}_{L_j}$ and $\mathcal{S}_{L_j}$ are defined over event set Σ_{IL_j}, where $j \in \{1, \ldots, n\}$.*

Proof: See page 129

We are now ready to state the proposition below which establishes useful properties for often used languages.

Proposition 29 *If n^{th} degree ($n \geq 1$)* parallel interface system *composed of DES $\mathcal{G}_H$, $\mathcal{G}_{L_1}, \ldots, \mathcal{G}_{L_n}, \mathcal{S}_H, \mathcal{S}_{L_1}, \ldots, \mathcal{S}_{L_n}, G_{I_1}, \ldots, G_{I_n}$, is* level-wise controllable *with respect to the alphabet partition $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}) \dot{\cup} \Sigma_H$ then, for all $j \in \{1, \ldots, n\}$, languages $\mathbf{H}$, $\mathbf{H}_{\mathcal{S}}$, $\mathbf{L}_j$, $\mathbf{L}_{\mathcal{S}_j}$, $\mathcal{I}_j$, $L(\mathbf{Plant})$, and $L(\mathbf{Sup})$ are closed.*

Proof: See page 130.

We now present a proposition that will aid in the use of the general form of *serial system extractions* in proofs. The proposition will interpret terminology for the j^{th} *serial system extraction* (a *serial interface system*) in terms of the original parallel system.

Before we can present the proposition, we need to first define (for use in the proposition) a new natural projection, P_j, to map strings from Σ^* (the event set of a given parallel system) to strings from $\Sigma(j)^*$ (the event set of the j^{th} extracted system of the given parallel system). It is as defined as follows:

$$P_j : \Sigma^* \rightarrow \Sigma(j)^*$$

Proposition 30 *If the* n^{th} *degree* ($n \geq 1$) parallel interface system *composed of DES* $\mathcal{G}_H$, $\mathcal{G}_{L_1}, \ldots, \mathcal{G}_{L_n}$, $\mathcal{S}_H$, $\mathcal{S}_{L_1}, \ldots, \mathcal{S}_{L_n}$, $G_{I_1}, \ldots, G_{I_n}$, *is* level-wise controllable *with respect to the alphabet partition* $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}) \dot{\cup} \Sigma_H$, *then for the* j^{th} serial system extraction, system(j), *the following is true:*

(i) The flat plant *is* $\mathbf{Plant}(j) = \mathcal{G}_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}}||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n}||_s \mathcal{G}_{L_j}$ *and the* flat supervisor *is* $\mathbf{Sup}(j) = \mathcal{S}_H||_s \mathcal{S}_{L_j}||_s G_{I_j}$

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

(iii) 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* $P_I(j)^{-1} = P_j \cdot P_{I_j}^{-1}$

(iv) The alphabet of $\mathcal{G}_H(j)$ *and* $\mathcal{S}_H(j)$ *is* $\Sigma_{IH}(j)$, *the alphabet of* $\mathcal{G}_L(j)$ *and* $\mathcal{S}_L(j)$ *is* $\Sigma_{IL}(j)$, *and*
the alphabet of $G_I(j)$ *is* $\Sigma_I(j)$

(v) The indicated languages satisfy the following statements:

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

$$\begin{aligned} \mathcal{I}(j) &= P_j\left(\mathcal{I}_j\right) \\ L(\mathbf{Plant}(j)) &= P_j\left(\mathbf{H}\right) \cap \left[\cap_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}} P_j\left(\mathcal{I}_k\right)\right] \cap P_j\left(\mathbf{L}_j\right) \\ L(\mathbf{Sup}(j)) &= P_j(\mathbf{H}_{\mathcal{S}}) \cap P_j\left(\mathbf{L}_{\mathcal{S}_j}\right) \cap P_j\left(\mathcal{I}_j\right) \end{aligned}$$

(vi) Languages $\mathbf{H}(j)$, $\mathbf{H}_{\mathcal{S}}(j)$, $\mathbf{L}(j)$, $\mathbf{L}_{\mathcal{S}}(j)$, $\mathcal{I}(j)$, $L(\mathbf{Plant})(j)$, *and* $L(\mathbf{Sup})(j)$ *are closed.*

Proof: See page 131.

We close this section by noting that after examining the definition of the j^{th} *serial system extraction: general form* and the above proposition, we see that for $n = 1$, a general form *parallel interface system* reduces to a single general form *serial interface system.* We thus see that a *serial interface system* is a special case of a *parallel interface system.* We will now talk of *parallel interface systems* and their definitions as the general case for bi-level interface systems.

8.4 Theorem and Propositions

We are now ready to present our main results for this chapter. We will first present two supporting propositions, followed by our parallel case controllability theorem. The following propositions are analogous to the serial controllability propositions.

8.4.1 Parallel Low level Controllability Proposition

We start with the parallel low level controllability proposition. It asserts that if the system is *level-wise controllable*, then each pair of low level supervisor and *interface* is controllable for the *flat plant.*

Proposition 31 *If the* n^{th} *degree* ($n \geq 1$) parallel interface system *composed of plant components* $\mathcal{G}_H, \mathcal{G}_{L_1}, \ldots, \mathcal{G}_{L_n}$, *supervisors* $\mathcal{S}_H, \mathcal{S}_{L_1}, \ldots, \mathcal{S}_{L_n}$, *and interfaces* $G_{I_1}, \ldots, G_{I_n}$, *is* level-wise controllable *with respect to the alphabet partition* $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}) \dot{\cup} \Sigma_H$, *then*

$$(\forall j \in \{1,\ldots,n\})\,(\forall s \in L(\mathbf{Plant}) \cap \mathbf{L}_{\mathcal{S}_j} \cap \mathcal{I}_j) \quad Elig_{L(\mathbf{Plant})}(s) \cap \Sigma_u \subseteq Elig_{\mathbf{L}_{\mathcal{S}_j} \cap \mathcal{I}_j}(s)$$

where **Plant** $:= \mathcal{G}_H||_s\mathcal{G}_{L_1}||_s \ldots ||_s\mathcal{G}_{L_n}$ *is the system's* flat plant.

Proof: See page 134.

8.4.2 Parallel High level Controllability Proposition

We now present the parallel high level controllability proposition. It asserts that if the system is *level-wise controllable*, then $\mathcal{S}_H$ is controllable for *flat plant* when the *flat Plant* is already under the control of the *interfaces.*

Proposition 32 *If the* n^{th} *degree* ($n \geq 1$) parallel interface system *composed of plant components* $\mathcal{G}_H, \mathcal{G}_{L_1}, \ldots, \mathcal{G}_{L_n}$, *supervisors* $\mathcal{S}_H, \mathcal{S}_{L_1}, \ldots, \mathcal{S}_{L_n}$, *and interfaces* $G_{I_1}, \ldots, G_{I_n}$, *is* level-wise controllable *with respect to the alphabet partition* $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}) \dot{\cup} \Sigma_H$, *then*

$$(\forall s \in L(\mathbf{Plant})\cap\mathbf{H}_{\mathcal{S}}\cap[\cap_{k\in\{1,\ldots,n\}}\mathcal{I}_k]) \quad Elig_{L(\mathbf{Plant})\cap[\cap_{k\in\{1,\ldots,n\}}\mathcal{I}_k]}(s)\cap\Sigma_u \subseteq Elig_{\mathbf{H}_{\mathcal{S}}}(s)$$

where **Plant** $:= \mathcal{G}_H||_s\mathcal{G}_{L_1}||_s \ldots ||_s\mathcal{G}_{L_n}$ *is the system's* flat plant.

Proof: See page 136.

8.4.3 Parallel Controllability Theorem

We now present our main result for this chapter, the parallel controllability theorem. It states that, to verify if a parallel system is controllable, it is sufficient to check that each of its *serial system extractions* is *serial level-wise controllable.* As the *level-wise controllable* definition can be evaluated by examining only one level of our system at a time (the *high level* or one of the *low levels*), we now have a means to verify controllability of our system using local checks.

Theorem 4 *If the* n^{th} *degree* ($n \geq 1$) parallel interface system *composed of plant components* $\mathcal{G}_H, \mathcal{G}_{L_1}, \ldots, \mathcal{G}_{L_n}$, *supervisors* $\mathcal{S}_H, \mathcal{S}_{L_1}, \ldots, \mathcal{S}_{L_n}$, *and interfaces* $G_{I_1}, \ldots, G_{I_n}$, *is* level-wise controllable *with respect to the alphabet partition* $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k})$

$\dot{\cup}\, \Sigma_H$, *then*

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

where $\mathbf{Plant} := \mathcal{G}_H||_s\mathcal{G}_{L_1}||_s \ldots ||_s\mathcal{G}_{L_n}$ *is the system's* flat plant, *and* $\mathbf{Sup} := \mathcal{S}_H||_s\mathcal{S}_{L_1} ||_s \ldots ||_s\mathcal{S}_{L_n}||_s\ G_{I_1}||_s \ldots ||_sG_{I_n}$ *is the system's* flat supervisor.

Proof:

Assume that the n^{th} degree ($n \geq 1$) *parallel interface system* is *level-wise controllable.* **(1)**

Let $s \in L(\mathbf{Plant}) \cap L(\mathbf{Sup})$, and $\sigma \in \text{Elig}_{L(\mathbf{Plant})}(s) \cap \Sigma_u$ **(2)**

We will now show this implies $\sigma \in \text{Elig}_{L(\mathbf{Sup})}(s)$. It's sufficient to show $s\sigma \in L(\mathbf{Sup}) = \mathbf{H}_{\mathcal{S}} \cap [\cap_{k\in\{1,\ldots,n\}}(\mathbf{L}_{\mathcal{S}_k} \cap \mathcal{I}_k)]$

We first note that $s, s\sigma \in \mathbf{H} \cap [\cap_{k\in\{1,\ldots,n\}}\mathbf{L}_k] = L(\mathbf{Plant})$ and $s \in \mathbf{H}_{\mathcal{S}} \cap [\cap_{k\in\{1,\ldots,n\}}(\mathbf{L}_{\mathcal{S}_k} \cap \mathcal{I}_k)]$ by **(2)**. **(3)**

We next note that, by **(1)**, we can apply **Proposition 31** and conclude:

$$s\sigma \in \cap_{k\in\{1,\ldots,n\}}(\mathbf{L}_{\mathcal{S}_k} \cap \mathcal{I}_k) \quad \textbf{(4)}$$

All that remains is to show that $s\sigma \in \mathbf{H}_{\mathcal{S}}$

From **(3)**, we have $s \in L(\mathbf{Plant}) \cap \mathbf{H}_{\mathcal{S}} \cap [\cap_{k\in\{1,\ldots,n\}}\mathcal{I}_k]$

From **(3)** and**(4)**, we have $s\sigma \in L(\mathbf{Plant}) \cap [\cap_{k\in\{1,\ldots,n\}}\mathcal{I}_k]$

$\Rightarrow \sigma \in \text{Elig}_{L(\mathbf{Plant})\cap[\cap_{k\in\{1,\ldots,n\}}\mathcal{I}_k]}(s) \cap \Sigma_u$

We can now apply **Proposition 32**, and conclude $\sigma \in \text{Elig}_{\mathbf{H}_{\mathcal{S}}}(s)$

$\Rightarrow s\sigma \in \mathbf{H}_{\mathcal{S}}$

From **(4)**, we can now conclude $s\sigma \in \mathbf{H}_{\mathcal{S}} \cap [\cap_{k\in\{1,\ldots,n\}}(\mathbf{L}_{\mathcal{S}_k} \cap \mathcal{I}_k)]$, as required

QED

8.5 Proofs of Selected Propositions

In order to make this work more readable, the proofs of some propositions in this chapter were not given as the propositions were introduced. They will now be presented in the

following sections.

8.5.1 Proof of Proposition 28

Proof for **Proposition 28** on page 124: *If* n^{th} *degree* ($n \geq 1$) parallel interface system *composed of DES* $\mathcal{G}_H$, $\mathcal{G}_{L_1}, \ldots, \mathcal{G}_{L_n}$, $\mathcal{S}_H$, $\mathcal{S}_{L_1}, \ldots, \mathcal{S}_{L_n}$, $G_{I_1}, \ldots, G_{I_n}$, *is* level-wise controllable *with respect to the alphabet partition* $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k}\dot{\cup}\Sigma_{R_k}\dot{\cup}\Sigma_{A_k}) \,\dot{\cup}\, \Sigma_H$ *then DES* $\mathcal{G}_H$ *and* $\mathcal{S}_H$ *are defined over event set* Σ_{IH}, *DES* G_{I_j} *is defined over event set* Σ_{I_j}, *and DES* $\mathcal{G}_{L_j}$ *and* $\mathcal{S}_{L_j}$ *are defined over event set* Σ_{IL_j}, *where* $j \in \{1, \ldots, n\}$.

Proof:

Assume that the n^{th} degree ($n \geq 1$) *parallel interface system* is *level-wise controllable* with respect to the alphabet partition. **(1)**

We will now show this implies that DES $\mathcal{G}_H$ and $\mathcal{S}_H$ are defined over event set Σ_{IH}, DES G_{I_j} is defined over event set Σ_{I_j}, and DES $\mathcal{G}_{L_j}$ and $\mathcal{S}_{L_j}$ are defined over event set Σ_{IL_j}, where $j \in \{1, \ldots, n\}$.

We first note that **(1)** implies that $(\forall j \in \{1, \ldots, n\})$ the j^{th} *serial system extraction*, labelled *system*(j), of the system is *serial level-wise controllable.*

This allows us to conclude that: $(\forall j \in \{1, \ldots, n\})$ DES $\mathcal{G}_H(j)$ and $\mathcal{S}_H(j)$ are defined over $\Sigma_{IH}(j)$, $\mathcal{G}_L(j)$ and $\mathcal{S}_L(j)$ are defined over $\Sigma_{IL}(j)$, and that $G_I(j)$ is defined over $\Sigma_I(j)$.

From **(1)**, we can now apply **Proposition 30**, **point ii** and conclude:[1]

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

We can now conclude that: $(\forall j \in \{1, \ldots, n\})$ DES $\mathcal{G}_H(j)$ and $\mathcal{S}_H(j)$ are defined over Σ_{IH}, $\mathcal{G}_L(j)$ and $\mathcal{S}_L(j)$ are defined over Σ_{IL_j}, and that $G_I(j)$ is defined over Σ_{I_j}. **(2)**

This implies: $(\forall j \in \{1, \ldots, n\})$ DES $\mathcal{S}_H = \mathcal{S}_H(j)$ is defined over Σ_{IH}, $\mathcal{G}_{L_j} = \mathcal{G}_L(j)$ and $\mathcal{S}_{L_j} = \mathcal{S}_L(j)$ are defined over Σ_{IL_j} and that $G_{I_j} = G_I(j)$ is defined over Σ_{I_j}. **(3)**

All that remains is to show that DES $\mathcal{G}_H$ is defined over alphabet Σ_{IH}. To do this, we first need to prove the following claim.

Claim: $\Sigma_{\mathcal{G}_H} \subseteq \Sigma_{IH}$ and $(\forall j \in \{1, \ldots, n\})\ \Sigma_{\mathcal{G}_H} \supseteq (\Sigma_H \cup \Sigma_{I_j})$

Let $j \in \{1, \ldots, n\}$. We will now show this implies $\Sigma_{\mathcal{G}_H} \subseteq \Sigma_{IH}$ and $\Sigma_{\mathcal{G}_H} \supseteq (\Sigma_H \cup \Sigma_{I_j})$.

[1] **Proposition 30**, **point iv** requires the proposition we are currently proving, but **point ii** of **Proposition 30** is independent of **point iv** and does not.

We start by noting $\mathcal{G}_H(j) := \mathcal{G}_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}}||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n}$. By the definition of the $||_s$ operator, we know $\Sigma_{\mathcal{G}_H(j)} = \Sigma_{\mathcal{G}_H} \cup [\cup_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}} \Sigma_{G_{I_k}}]$. This implies that $\Sigma_{\mathcal{G}_H} \subseteq \Sigma_{\mathcal{G}_H(j)}$. As $\Sigma_{\mathcal{G}_H(j)} = \Sigma_{IH}$ (from **(2)**), we immediately have $\Sigma_{\mathcal{G}_H} \subseteq \Sigma_{IH}$.

From **(3)**, we have $\Sigma_{\mathcal{G}_H(j)} = \Sigma_{\mathcal{G}_H} \cup [\cup_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}} \Sigma_{I_k}]$.

We now note that $\Sigma_{I_j} \subseteq \Sigma_{IH}$ but $\Sigma_{I_j} \cap [\cup_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}} \Sigma_{I_k}] = \emptyset$ because of our event partition.

This implies: $\Sigma_{I_j} \subseteq \Sigma_{\mathcal{G}_H}$

We next note that $\Sigma_H \subseteq \Sigma_{IH}$ but $\Sigma_H \cap [\cup_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}} \Sigma_{I_k}] = \emptyset$ because of our event partition.

This implies: $\Sigma_H \subseteq \Sigma_{\mathcal{G}_H}$

We thus have $\Sigma_{\mathcal{G}_H} \supseteq (\Sigma_H \cup \Sigma_{I_j})$ as required.

Claim proven.

From the claim, we have $\Sigma_{\mathcal{G}_H} \subseteq \Sigma_{IH}$. To show that $\Sigma_{\mathcal{G}_H} = \Sigma_{IH}$ we now only have to show $\Sigma_{\mathcal{G}_H} \supseteq \Sigma_{IH}$.

From the claim, we also have $(\forall j \in \{1,\ldots,n\})\ \Sigma_{\mathcal{G}_H} \supseteq (\Sigma_H \cup \Sigma_{I_j})$.

This implies $\Sigma_{\mathcal{G}_H} \supseteq \Sigma_H \cup [\cup_{k\in\{1,\ldots,n\}} \Sigma_{I_k}] = \Sigma_{IH}$. We thus have $\Sigma_{\mathcal{G}_H} = \Sigma_{IH}$.

We can now conclude that DES $\mathcal{G}_H$ is defined over Σ_{IH}, as required.

QED

8.5.2 Proof of Proposition 29

Proof for **Proposition 29** on page 124: If n^{th} degree ($n \geq 1$) *parallel interface system* composed of DES $\mathcal{G}_H$, $\mathcal{G}_{L_1}, \ldots, \mathcal{G}_{L_n}$, $\mathcal{S}_H$, $\mathcal{S}_{L_1}, \ldots, \mathcal{S}_{L_n}$, $G_{I_1}, \ldots, G_{I_n}$, is *level-wise controllable* with respect to the alphabet partition $\Sigma := \dot\cup_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot\cup \Sigma_{R_k} \dot\cup \Sigma_{A_k}) \dot\cup \Sigma_H$ then, for all $j \in \{1,\ldots,n\}$, languages $\mathbf{H}$, $\mathbf{H}_\mathcal{S}$, $\mathbf{L}_j$, $\mathbf{L}_{\mathcal{S}_j}$, $\mathcal{I}_j$, $L(\mathbf{Plant})$, and $L(\mathbf{Sup})$ are closed.

Proof:

Assume system is *level-wise controllable.* Let $j \in \{1,\ldots,n\}$. **(1)**

Will now show this implies that the indicated languages are closed.

We first note that by **(1)**, the system is *level-wise controllable.* This allows us to apply **Proposition 28** and conclude:

$$L(\mathcal{G}_H), L(\mathcal{S}_H) \subseteq \Sigma^*_{IH},\ L(\mathcal{G}_{L_j}), L(\mathcal{S}_{L_j}) \subseteq \Sigma^*_{IL_j}, \text{ and } L(G_{I_j}) \subseteq \Sigma^*_{I_j}.$$

This tells us that languages $\mathbf{H} = P^{-1}_{IH}(L(\mathcal{G}_H))$, $\mathbf{H}_\mathcal{S} = P^{-1}_{IH}L(\mathcal{S}_H)$, $\mathbf{L}_j = P^{-1}_{IL}(L(\mathcal{G}_{L_j}))$, $\mathbf{L}_{\mathcal{S}_j} = P^{-1}_{IL}L(\mathcal{S}_{L_j})$, $\mathcal{I}_j = P^{-1}_{I}(L(G_{I_j}))$ are properly defined.

We will start by showing that languages $\mathbf{H}$, $\mathbf{H}_\mathcal{S}$, $\mathbf{L}_j$, $\mathbf{L}_{\mathcal{S}_j}$, and $\mathcal{I}_j$ are closed.

We now note that languages $L(\mathcal{G}_H)$, $L(\mathcal{S}_H)$, $L(\mathcal{G}_{L_j})$, $L(\mathcal{S}_{L_j})$, and $L(G_{I_j})$ are closed by the definition of the closed behaviour of a DES.

We can now apply **Proposition 1** repeatedly and conclude that $\mathbf{H}$, $\mathbf{H}_\mathcal{S}$, $\mathbf{L}_j$, $\mathbf{L}_{\mathcal{S}_j}$, and $\mathcal{I}_j$ are closed, as required. **(2)**

We will now show that languages $L(\mathbf{Plant})$, and $L(\mathbf{Sup})$ are closed.

We next note that $L(\mathbf{Plant}) = \mathbf{H} \cap [\cap_{k\in\{1,\ldots,n\}}\mathbf{L}_k]$ and $L(\mathbf{Sup}) = \mathbf{H}_\mathcal{S} \cap [\cap_{k\in\{1,\ldots,n\}}(\mathbf{L}_{\mathcal{S}_k} \cap \mathcal{I}_k)]$.

Combining with **(2)**, we can now apply **Proposition 2** repeatedly and conclude that $L(\mathbf{Plant})$, and $L(\mathbf{Sup})$ are closed, as required.

QED

8.5.3 Proof of Proposition 30

Proof for **Proposition 30** on page 125: *If the* n^{th} *degree* ($n \geq 1$) parallel interface system *composed of DES* $\mathcal{G}_H$, $\mathcal{G}_{L_1}, \ldots,$ $\mathcal{G}_{L_n}$, $\mathcal{S}_H$, $\mathcal{S}_{L_1}, \ldots, \mathcal{S}_{L_n}$, $G_{I_1}, \ldots, G_{I_n}$, *is* level-wise controllable *with respect to the alphabet partition* $\Sigma := \dot\cup_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot\cup \Sigma_{R_k} \dot\cup \Sigma_{A_k}) \dot\cup \Sigma_H$, *then for the* j^{th} serial system extraction, system(j), *the following is true:*

(i) The flat plant *is* $\mathbf{Plant}(j) = \mathcal{G}_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}}||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n}||_s \mathcal{G}_{L_j}$ *and the* flat supervisor *is* $\mathbf{Sup}(j) = \mathcal{S}_H||_s \mathcal{S}_{L_j}||_s G_{I_j}$

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

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

(iv) The alphabet of $\mathcal{G}_H(j)$ *and* $\mathcal{S}_H(j)$ *is* $\Sigma_{IH}(j)$, *the alphabet of* $\mathcal{G}_L(j)$ *and* $\mathcal{S}_L(j)$ *is* $\Sigma_{IL}(j)$, *and*

the alphabet of $G_I(j)$ *is* $\Sigma_I(j)$

(v) The indicated languages satisfy the following statements:

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

(vi) Languages $\mathbf{H}(j)$, $\mathbf{H}_{\mathcal{S}}(j)$, $\mathbf{L}(j)$, $\mathbf{L}_{\mathcal{S}}(j)$, $\mathcal{I}(j)$, $L(\mathbf{Plant})(j)$, *and* $L(\mathbf{Sup})(j)$ *are closed.*

Proof:

Assume that the n^{th} degree ($n \geq 1$) *parallel interface system* is *level-wise controllable* with respect to the alphabet partition. **(1)**

Let *system*(j) be the j^{th} *serial system extraction* of our parallel system. **(2)**

We will now show this implies *system*(j) satisfies **points i-vi.**

Point i: Show that the *flat plant* is $\mathbf{Plant}(j) = \mathcal{G}_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}}||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n} ||_s \mathcal{G}_{L_j}$ and the *flat supervisor* is $\mathbf{Sup}(j) = \mathcal{S}_H||_s \mathcal{S}_{L_j}||_s G_{I_j}$

$$\begin{aligned}
\mathbf{Plant}(j) &:= \mathcal{G}_H(j)||_s \mathcal{G}_L(j), \text{ by definition.} \\
&= \mathcal{G}_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}}||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n}||_s \mathcal{G}_{L_j}, \text{ by } \mathbf{(2)}. \\
\mathbf{Sup}(j) &:= \mathcal{S}_H(j)||_s \mathcal{S}_L(j)||_s G_I(j), \text{ by definition.} \\
&= \mathcal{S}_H||_s \mathcal{S}_{L_j}||_s G_{I_j}, \text{ by } \mathbf{(2)}.
\end{aligned}$$

Point ii: Show that the following event sets are: $\Sigma_I(j) = \Sigma_{I_j}$, $\Sigma_{IH}(j) = \Sigma_{IH}$, and $\Sigma_{IL}(j) = \Sigma_{IL_j}$

Proof is identical to the proof of **point ii** of **Proposition 23**.

Point iii: Show that $P_{IH}(j)^{-1} = P_j \cdot P_{IH}^{-1}$, $P_{IL}(j)^{-1} = P_j \cdot P_{IL_j}^{-1}$, and $\Sigma_I(j) = P_j \cdot P_{I_j}^{-1}$

Proof is identical to the proof of **point iii** of **Proposition 23**.

Point iv: Show that the alphabet of $\mathcal{G}_H(j)$ and $\mathcal{S}_H(j)$ is $\Sigma_{IH}(j)$, the alphabet of $\mathcal{G}_L(j)$

and $\mathcal{S}_L(j)$ is $\Sigma_{IL}(j)$, and
the alphabet of $G_I(j)$ is $\Sigma_I(j)$

From **(1)**, we have that the system is *level-wise controllable*. This implies that *system*(j) is *serial level-wise controllable*. The result follows immediately.

Point v:

First, we must show that $\mathbf{H}(j) = P_j(\mathbf{H}) \cap [\cap_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}} P_j(\mathcal{I}_k)]$

Proof is identical to the proof for $\mathcal{H}(j)$ of **point v** of **Proposition 23** after relabelling and substituting **Proposition 28** for **Proposition 21**. **(3)**

The proofs for the remaining languages for **point v** are straightforward, and are presented together below.

$$
\begin{aligned}
\mathbf{H}_{\mathcal{S}}(j) &:= P_{IH}(j)^{-1}L(\mathcal{S}_H(j)), \text{ by definition.} \\
&= P_j \cdot P_{IH}^{-1}(L(\mathcal{S}_H)), \text{ by } \textbf{(2)} \text{ and } \textbf{point iii}. \\
&= P_j(\mathbf{H}_{\mathcal{S}}) \qquad (\mathbf{4}) \\
\mathbf{L}(j) &:= P_{IL}(j)^{-1}(L(\mathcal{G}_L(j))), \text{ by definition.} \\
&= P_j \cdot P_{IL_j}^{-1}(L(\mathcal{G}_{L_j})), \text{ by } \textbf{(2)} \text{ and } \textbf{point iii}. \\
&= P_j(\mathbf{L}_j) \qquad (\mathbf{5}) \\
\mathbf{L}_{\mathcal{S}}(j) &:= P_{IL}(j)^{-1}(L(\mathcal{S}_L(j))), \text{ by definition.} \\
&= P_j \cdot P_{IL_j}^{-1}(L(\mathcal{S}_{L_j})), \text{ by } \textbf{(2)} \text{ and } \textbf{point iii}. \\
&= P_j(\mathbf{L}_{\mathcal{S}_j}) \qquad (\mathbf{6}) \\
\mathcal{I}(j) &:= P_I(j)^{-1}(L(G_I(j))), \text{ by definition.} \\
&= P_j \cdot P_{I_j}^{-1}(L(G_{I_j})), \text{ by } \textbf{(2)} \text{ and } \textbf{point iii}. \\
&= P_j(\mathcal{I}_j) \qquad (\mathbf{7}) \\
L(\mathbf{Plant}(j)) &= \mathbf{H}(j) \cap \mathbf{L}(j), \text{ by definition.} \\
&= P_j(\mathbf{H}) \cap [\cap_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}} P_j(\mathcal{I}_k)] \cap P_j(\mathbf{L}_j), \text{ by } \textbf{(3)} \text{ and } \textbf{(5)}. \\
L(\mathbf{Sup}(j)) &= \mathbf{H}_{\mathcal{S}}(j) \cap \mathbf{L}_{\mathcal{S}}(j) \cap \mathcal{I}(j), \text{ by definition.} \\
&= P_j(\mathbf{H}_{\mathcal{S}}) \cap P_j(\mathbf{L}_{\mathcal{S}_j}) \cap P_j(\mathcal{I}_j), \text{ by } \textbf{(4)}, \textbf{(6)}, \text{ and } \textbf{(7)}.
\end{aligned}
$$

Point vi: Show that the languages $\mathbf{H}(j)$, $\mathbf{H}_{\mathcal{S}}(j)$, $\mathbf{L}(j)$, $\mathbf{L}_{\mathcal{S}}(j)$, $\mathcal{I}(j)$, $L(\mathbf{Plant})(j)$, and $L(\mathbf{Sup})(j)$ are closed.

From **(2)**, we know that $\mathcal{G}_H(j) = \mathcal{G}_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}}||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n}$, $\mathbf{Plant} = \mathcal{G}_H||_s G_{I_1}\ ||_s \ldots ||_s G_{I_{(j-1)}}||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n}||_s \mathcal{G}_{L_j}$, and $\mathbf{Sup} = \mathcal{S}_H||_s \mathcal{S}_{L_j}||_s G_{I_j}$.

We can now apply **Proposition 5** and conclude that languages $L(\mathcal{G}_H(j))$, $L(\mathbf{Plant})$, and $L(\mathbf{Sup})$ is closed.

We next note that languages $\mathcal{S}_H(j)$, $\mathcal{G}_L(j)$, $\mathcal{S}_L(j)$, and $G_I(j)$ are closed as $\mathcal{S}_H(j) = \mathcal{S}_H$,

$\mathcal{G}_L(j) = \mathcal{G}_{L_j}$, $\mathcal{S}_L(j) = \mathcal{S}_{L_j}$, and $G_I(j) = G_{I_j}$ (by **(2)**), and by the definition of the closed behaviour of a DES.

We now apply **Proposition 1** repeatedly and conclude that $\mathbf{H}(j) = P_{IH}(j)^{-1}L(\mathcal{G}_H(j))$, $\mathbf{H}_{\mathcal{S}}(j) = P_{IH}(j)^{-1}L(\mathcal{S}_H(j))$, $\mathbf{L}(j) = P_{IL}(j)^{-1}L(\mathcal{G}_L(j))$, $\mathbf{L}_{\mathcal{S}}(j) = P_{IL}(j)^{-1}L(\mathcal{S}_L(j))$, and $\mathcal{I}(j) = P_I(j)^{-1}L(G_I(j))$ are closed.

QED

8.5.4 Proof of Proposition 31

Proof for **Proposition 31** on page 126: *If the* n^{th} *degree* $(n \geq 1)$ parallel interface system *composed of plant components* $\mathcal{G}_H, \mathcal{G}_{L_1}, \ldots, \mathcal{G}_{L_n}$, *supervisors* $\mathcal{S}_H, \mathcal{S}_{L_1}, \ldots, \mathcal{S}_{L_n}$, *and interfaces* $G_{I_1}, \ldots, G_{I_n}$, *is* level-wise controllable *with respect to the alphabet partition* $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \,\dot{\cup}\, \Sigma_{R_k} \,\dot{\cup}\, \Sigma_{A_k}) \,\dot{\cup}\, \Sigma_H$, *then*

$$(\forall j \in \{1,\ldots,n\})\,(\forall s \in L(\mathbf{Plant}) \cap \mathbf{L}_{\mathcal{S}_j} \cap \mathcal{I}_j) \quad \mathit{Elig}_{L(\mathbf{Plant})}(s) \cap \Sigma_u \subseteq \mathit{Elig}_{\mathbf{L}_{\mathcal{S}_j} \cap \mathcal{I}_j}(s)$$

where $\mathbf{Plant} := \mathcal{G}_H ||_s \mathcal{G}_{L_1} ||_s \ldots ||_s \mathcal{G}_{L_n}$ *is the system's* flat plant.

Proof:

Assume that the n^{th} degree $(n \geq 1)$ *parallel interface system* is *level-wise controllable.* **(1)**

Let $j \in \{1,\ldots,n\}$, $s \in L(\mathbf{Plant}) \cap \mathbf{L}_{\mathcal{S}_j} \cap \mathcal{I}_j$, and $\sigma \in \text{Elig}_{L(\mathbf{Plant})}(s) \cap \Sigma_u$. **(2)**

We will now show that this implies $\sigma \in \text{Elig}_{\mathbf{L}_{\mathcal{S}_j} \cap \mathcal{I}_j}(s)$.

It's sufficient to show that $s\sigma \in \mathbf{L}_{\mathcal{S}_j} \cap \mathcal{I}_j$.

We first note that $s, s\sigma \in \mathbf{H} \cap [\cap_{k\in\{1,\ldots,n\}}\mathbf{L}_k] = L(\mathbf{Plant})$ by **(2)**. **(3)**

We have two cases: **I)** $\sigma \notin \Sigma_{IL_j}$ and **II)** $\sigma \in \Sigma_{IL_j}$.

case I)

Assume $\sigma \notin \Sigma_{IL_j}$. This implies: $P_{IL_j}(\sigma) = \epsilon$, where ϵ is the empty string.

$\Rightarrow\ P_{IL_j}(s\sigma) = P_{IL_j}(s)P_{IL_j}(\sigma) = P_{IL_j}(s)$, as the natural projection is catenative. Similarly, we have $P_{I_j}(s\sigma) = P_{I_j}(s)$ as $\sigma \notin \Sigma_{I_j}$ since $\Sigma_{I_j} \subseteq \Sigma_{IL_j}$. **(4)**

From **(2),** we have $s \in \mathbf{L}_{\mathcal{S}_j} \cap \mathcal{I}_j$.

As $s \in \mathbf{L}_{\mathcal{S}_j}$ and $P_{IL_j}(s\sigma) = P_{IL_j}(s)$, we can apply **Proposition 27**, **point c**, and conclude $s\sigma \in \mathbf{L}_{\mathcal{S}_j}$.

Similarly, we can apply **Proposition 20**, **point e**,and conclude $s\sigma \in \mathcal{I}_j$.

We thus have $s\sigma \in \mathbf{L}_{\mathcal{S}_j} \cap \mathcal{I}_j$.

Case I complete.

case II)

Assume $\sigma \in \Sigma_{IL_j}$.

We now examine *system*(j), the j^{th} *serial system extraction* of our parallel system.

We first note that we have $\sigma \in \Sigma_{IL}(j)$ as $\Sigma_{IL}(j) = \Sigma_{IL_j}$ by **Proposition 30**.

$\Rightarrow \ \sigma \in \Sigma(j) \supseteq \Sigma_{IL}(j)$

$\Rightarrow P_j(\sigma) = \sigma$. See Section 8.3 for the definition of the natural projection P_j.

$\Rightarrow \ P_j(s\sigma) = P_j(s)\sigma$

From **(1),** we can conclude that *system*(j) is *serial level-wise controllable.*

We will use **point II** of this definition to show that $P_j(s)\sigma \in \mathbf{L}_{\mathcal{S}}(j) \cap \mathcal{I}(j)$.

To do this, we first need to show that $P_j(s), P_j(s)\sigma \in \mathbf{L}(j)$.

As $s, s\sigma \in \mathbf{H} \cap [\cap_{k\in\{1,\ldots,n\}}\mathbf{L}_k]$ by **(3),** we have $s, s\sigma \in \mathbf{L}_j$.

$\Rightarrow \ P_j(s) \in P_j\mathbf{L}_j$ and $P_j(s\sigma) = P_j(s)\sigma \in P_j\mathbf{L}_j$

$\Rightarrow \ P_j(s), P_j(s)\sigma \in \mathbf{L}(j)$, by **Proposition 30**.

As we have $\sigma \in \Sigma_u$ from **(2),** we can conclude $\sigma \in \text{Elig}_{\mathbf{L}(j)}(P_j(s)) \cap \Sigma_u$.

We now only need to show $P_j(s) \in \mathbf{L}_{\mathcal{S}}(j) \cap \mathcal{I}(j)$.

From **(2),** we have $s \in L(\mathbf{Sup})$ and thus $s \in \mathbf{L}_{\mathcal{S}_j} \cap \mathcal{I}_j$.

$\Rightarrow \ P_j(s) \in P_j\mathbf{L}_{\mathcal{S}_j} \cap P_j\mathcal{I}_j$

$\Rightarrow \ P_j(s) \in \mathbf{L}_{\mathcal{S}}(j) \cap \mathcal{I}(j)$, by **Proposition 30**.

We now have $P_j(s) \in \mathbf{L}(j) \cap \mathbf{L}_{\mathcal{S}}(j) \cap \mathcal{I}(j)$ and $\sigma \in \text{Elig}_{\mathbf{L}(j)}(P_j(s)) \cap \Sigma_u$ and can conclude by **point II** of the *serial level-wise controllable* definition that:

$$\sigma \in \text{Elig}_{\mathbf{L}_\mathcal{S}(j)\cap\mathcal{I}(j)}(P_j(s)) \text{ and thus } P_j(s)\sigma = P_j(s\sigma) \in \mathbf{L}_\mathcal{S}(j) \cap \mathcal{I}(j)$$

Substituting in for $\mathbf{L}_\mathcal{S}(j)$ and $\mathcal{I}(j)$ (by **Proposition 30**) gives: $P_j(s\sigma) \in P_j P_{IL_j}^{-1} L(\mathcal{S}_{L_j}) \cap P_j P_{I_j}^{-1} L(G_{I_j})$

We note that since $\Sigma_{IL}(j) = \Sigma_{IL_j}$ and $\Sigma_I(j) = \Sigma_{I_j}$ we have $\Sigma_{IL_j} \subseteq \Sigma(j)$ and $\Sigma_{I_j} \subseteq \Sigma(j)$. We can thus apply **Corollary 2** twice, taking first $\Sigma_a = \Sigma(j)$, $\Sigma_b = \Sigma_{IL_j}$, and $L_b = L(\mathcal{S}_{L_j})$ and then $\Sigma_a = \Sigma(j)$, $\Sigma_b = \Sigma_{I_j}$, and $L_b = L(G_{I_j})$. We can thus conclude:

$$s\sigma \in P_{IL_j}^{-1} L(\mathcal{S}_{L_j}) \cap P_{I_j}^{-1} L(G_{I_j}) = \mathbf{L}_{\mathcal{S}_j} \cap \mathcal{I}_j$$

Case II complete.

By **Cases I** and **II**, we have $s\sigma \in \mathbf{L}_{\mathcal{S}_j} \cap \mathcal{I}_j$, as required.

QED

8.5.5 Proof of Proposition 32

Proof for **Proposition 32** on page 127: *If the* n^{th} *degree* ($n \geq 1$) parallel interface system *composed of plant components* $\mathcal{G}_H, \mathcal{G}_{L_1}, \ldots, \mathcal{G}_{L_n}$, *supervisors* $\mathcal{S}_H, \mathcal{S}_{L_1}, \ldots, \mathcal{S}_{L_n}$, *and interfaces* $G_{I_1}, \ldots, G_{I_n}$, *is* level-wise controllable *with respect to the alphabet partition* $\Sigma := \dot{\cup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \,\dot{\cup}\, \Sigma_{R_k} \,\dot{\cup}\, \Sigma_{A_k}) \,\dot{\cup}\, \Sigma_H$, *then*

$$(\forall s \in L(\mathbf{Plant}) \cap \mathbf{H}_\mathcal{S} \cap [\cap_{k\in\{1,\ldots,n\}} \mathcal{I}_k]) \quad \mathit{Elig}_{L(\mathbf{Plant})\cap[\cap_{k\in\{1,\ldots,n\}}\mathcal{I}_k]}(s) \cap \Sigma_u \subseteq \mathit{Elig}_{\mathbf{H}_\mathcal{S}}(s)$$

where $\mathbf{Plant} := \mathcal{G}_H ||_s \mathcal{G}_{L_1} ||_s \ldots ||_s \mathcal{G}_{L_n}$ *is the system's* flat plant.

Proof:

Assume that the n^{th} degree ($n \geq 1$) *parallel interface system* is *level-wise controllable.* **(1)**

Let $s \in L(\mathbf{Plant}) \cap \mathbf{H}_\mathcal{S} \cap [\cap_{k\in\{1,\ldots,n\}} \mathcal{I}_k]$, and $\sigma \in \text{Elig}_{L(\mathbf{Plant})\cap[\cap_{k\in\{1,\ldots,n\}}\mathcal{I}_k]}(s) \cap \Sigma_u$ **(2)**

We will now show that this implies $\sigma \in \text{Elig}_{\mathbf{H}_\mathcal{S}}(s)$

It's sufficient to show that $s\sigma \in \mathbf{H}_\mathcal{S}$

We first note that:

$$s, s\sigma \in \mathbf{H} \cap [\cap_{k\in\{1,\ldots,n\}}(\mathbf{L}_k \cap \mathcal{I}_k)] = L(\mathbf{Plant}) \cap [\cap_{k\in\{1,\ldots,n\}} \mathcal{I}_k] \text{ by } \mathbf{(2)}. \qquad (3)$$

By examining the definition of $\Sigma(j)$ for some $j \in \{1, \ldots, n\}$ (see definition of j^{th} serial

system extraction: general form on page 123), we see that $\Sigma = \cup_{k\in\{1,\ldots,n\}}\Sigma(k)$

$\Rightarrow\ (\exists j \in \{1,\ldots,n\})\ \sigma \in \Sigma(j)$ **(4)**

We use this j and note that by **(1)**, we can conclude that *system*(j), the j^{th} *serial system extraction* of our parallel system, is *serial level-wise controllable.* **(5)**

We will use **point III** of the *serial level-wise controllable* definition to show that $P_j(s)\sigma \in \mathbf{H}_{\mathcal{S}}(j)$. See Section 8.3 for the definition of the natural projection P_j.

We first need to show that $P_j(s) \in \mathbf{H}(j) \cap \mathcal{I}(j) \cap \mathbf{H}_{\mathcal{S}}(j)$ and $\sigma \in \text{Elig}_{\mathbf{H}(j)\cap\mathcal{I}(j)}(s) \cap \Sigma_u$

From **(2)** and **(3)**, we have $s \in \mathbf{H} \cap \mathbf{H}_{\mathcal{S}} \cap [\cap_{k\in\{1,\ldots,n\}}\mathcal{I}_k]$

$\Rightarrow P_j(s) \in P_j(\mathbf{H}) \cap P_j(\mathbf{H}_{\mathcal{S}}) \cap [\cap_{k\in\{1,\ldots,n\}}P_j(\mathcal{I}_k)]$

$\Rightarrow P_j(s) \in \mathbf{H}(j) \cap \mathcal{I}(j) \cap \mathbf{H}_{\mathcal{S}}(j)$, by **Proposition 30**.

Similarly, from **(3)** we can conclude $P_j(s\sigma) \in \mathbf{H}(j) \cap \mathcal{I}(j)$

We next note that $\sigma \in \Sigma(j)$ (from **(4)**) implies that $P_j(s\sigma) = P_j(s)\sigma$.

$\Rightarrow \sigma \in \text{Elig}_{\mathbf{H}(j)\cap\mathcal{I}(j)}(P_j(s)) \cap \Sigma_u$

We can now conclude by **point III** of the *serial level-wise controllable* definition that:

$$\sigma \in \text{Elig}_{\mathbf{H}_{\mathcal{S}}(j)}(P_j(s)) \text{ and thus } P_j(s)\sigma = P_j(s\sigma) \in \mathbf{H}_{\mathcal{S}}(j)$$

$\Rightarrow P_j(s\sigma) \in P_j(\mathbf{H}_{\mathcal{S}})$, by **Proposition 30**.

$\Rightarrow P_j(s\sigma) \in P_j P_{IH}^{-1} L(\mathcal{S}_H)$

As $\Sigma_{IH} = \Sigma_{IH}(j)$ by **Proposition 30**, we have $\Sigma_{IH} \subseteq \Sigma(j)$. We can thus apply **Corollary 2** by taking $\Sigma_a = \Sigma(j)$, $\Sigma_b = \Sigma_{IH}$, and $L_b = L(\mathcal{S}_H)$ and thus conclude:

$$s\sigma \in P_{IH}^{-1} L(\mathcal{S}_H) = \mathbf{H}_{\mathcal{S}}, \text{ as required.}$$

QED

Chapter 9

Parallel Manufacturing Example

To illustrate the parallel case, we will look at the simple manufacturing system shown in Figure 9.1. The system is composed of three manufacturing units running in parallel, a testing unit, material feedback, a packaging unit, plus three buffers to insure a proper flow of material.

For the manufacturing units (indexed by $j = \mathrm{I}, \mathrm{II}, \mathrm{III}$), we will reuse the systems developed in Chapter 5 with the packaging unit removed and treating the system as a flat model (i.e. ignoring for now the system's own interface structure). Figure 9.2 shows the plant models for each manufacturing unit.

For the source, sink, test unit, and packaging unit, we introduced new plant models. They are shown in Figure 9.3. For the three buffers, they will be implemented as supervisors.

9.1 Design Details

For this example, we want to design a parallel case interface system with the structure shown in Figure 9.4. We will treat the three independent manufacturing units as our *low levels.* Our first step is to define which plant model exists at which level. This is shown in Figure 9.5.

We now need to define *interfaces* between the *high level* and each of the three *low levels.* Normally, each *interface* would be quite different, but since each *low level* is an instance

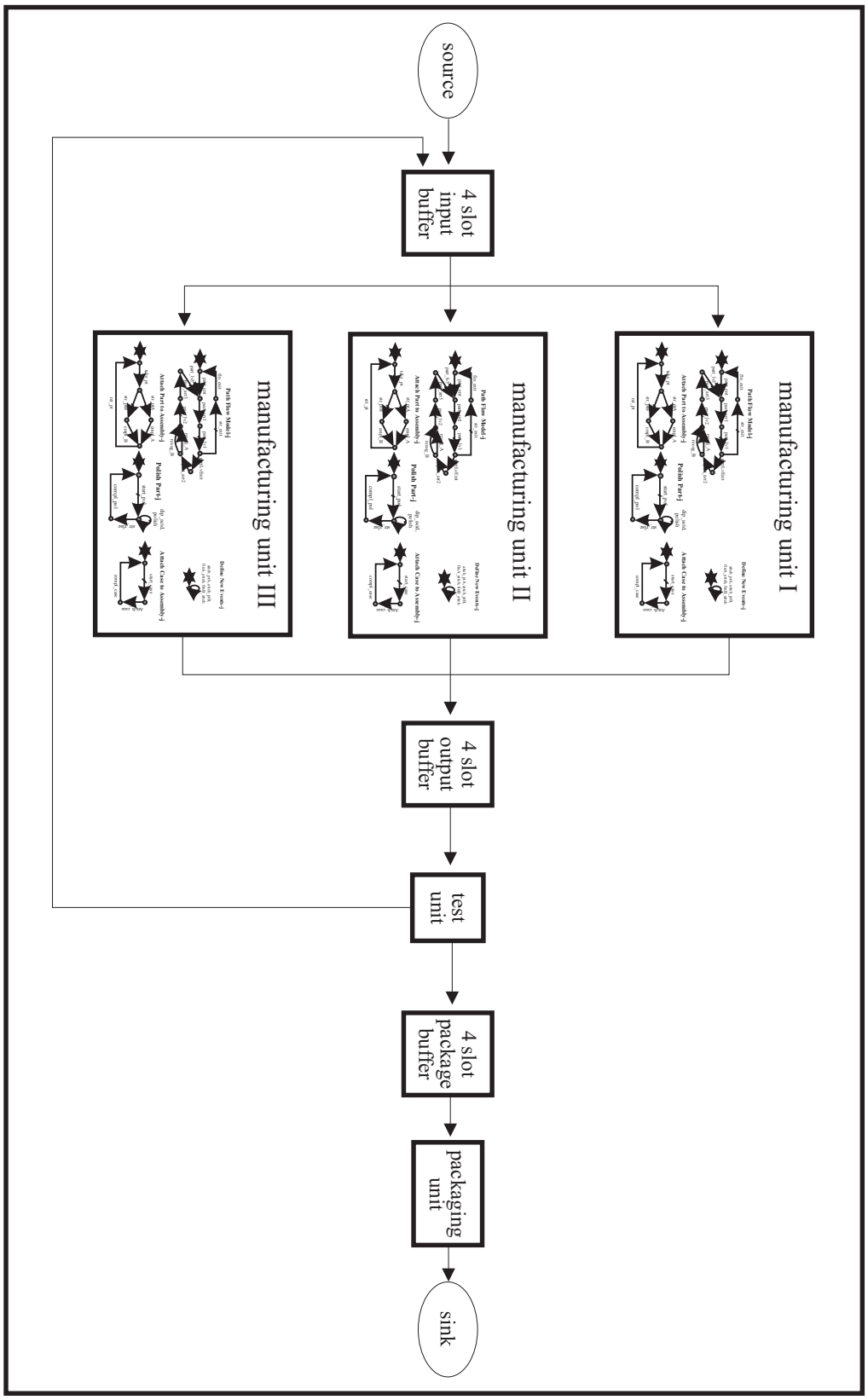


Figure 9.1: Block Diagram of Parallel Plant

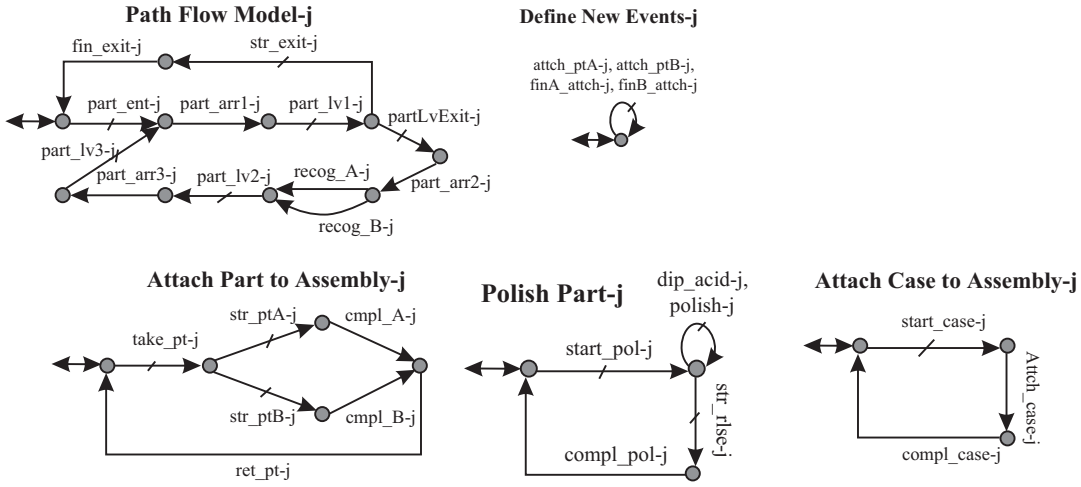


Figure 9.2: Plant Models for Manufacturing Unit j

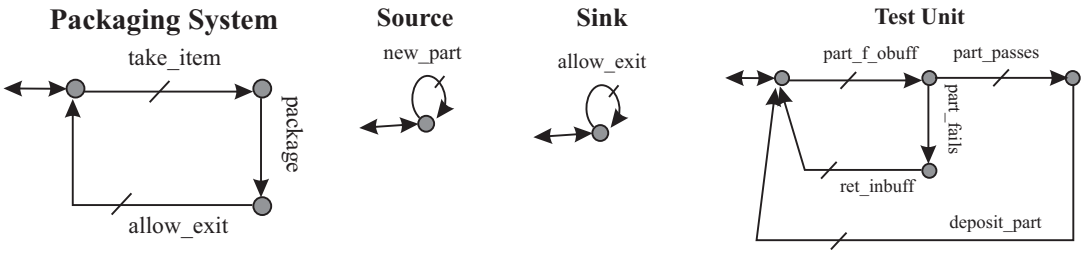


Figure 9.3: New Plant Models

of the same manufacturing unit, it makes sense that the *interfaces* are also all of the same form. Figure 9.6 shows the *interface* to *low level* j. For the remainder of this chapter, we will take $j = \mathrm{I}, \mathrm{II}, \mathrm{III}$.

We can now define the alphabet partition $\Sigma := [\dot{\cup}_{k\in\{\mathrm{I},\mathrm{II},\mathrm{III}\}}(\Sigma_{L_k}\dot{\cup}\Sigma_{R_k}\dot{\cup}\Sigma_{A_k})] \dot{\cup} \Sigma_H$ as below:

Σ_H = {*take_item*, *package*, *allow_exit*, *new_part*, *part_f_obuff*, *part_passes*, *part_fails*, *ret_inbuff*, *deposit_part*}

Σ_{R_j} = {*part_ent-j*}

Σ_{A_j} = {*fin_exit-j*}

Σ_{L_j} = {*start_pol-j*, *attch_ptA-j*, *attch_ptB-j*, *start_case-j*, *comp_pol-j*, *finA_attch-j*, *finB_attch-j*, *compl_case-j*, *part_arr1-j*, *part_lv1-j*, *partLvExit-j*, *str_exit-j*, *part_arr2-j*, *recog_A-j*, *recog_B-j*, *part_lv2-j*, *part_arr3-j*, *part_lv3-j*, *take_pt-j*, *str_ptA-j*, *str_ptB-j*, *compl_A-j*, *compl_B-j*,

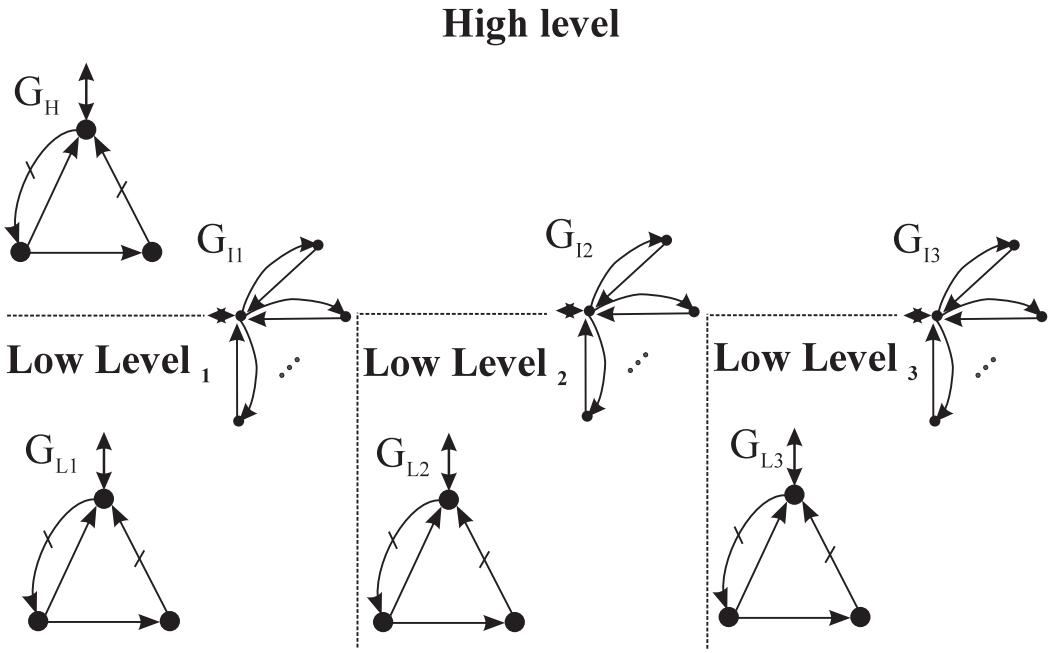


Figure 9.4: Desired Interface Structure

ret_pt-j, dip_acid-j, polish-j, str_rlse-j, attch_case-j}

Our next step is to design new supervisors for our *low levels.* As we are reusing the manufacturing unit designed in Chapter 5, we already have a system that is designed to accept a new part, process it appropriately, and then allow the part to leave the unit; thus we can simply reuse supervisors from that chapter. They are shown in Figure 9.7 for *low level* j.

For the *high level,* we need to design supervisors to implement the input buffer, the output buffer, and the package buffer. Each buffer should have four slots and should never underflow or overflow. The corresponding supervisors are shown in Figure 9.8. Finally, we note that the above supervisors were designed by hand, but we could have also employed synthesis methods.

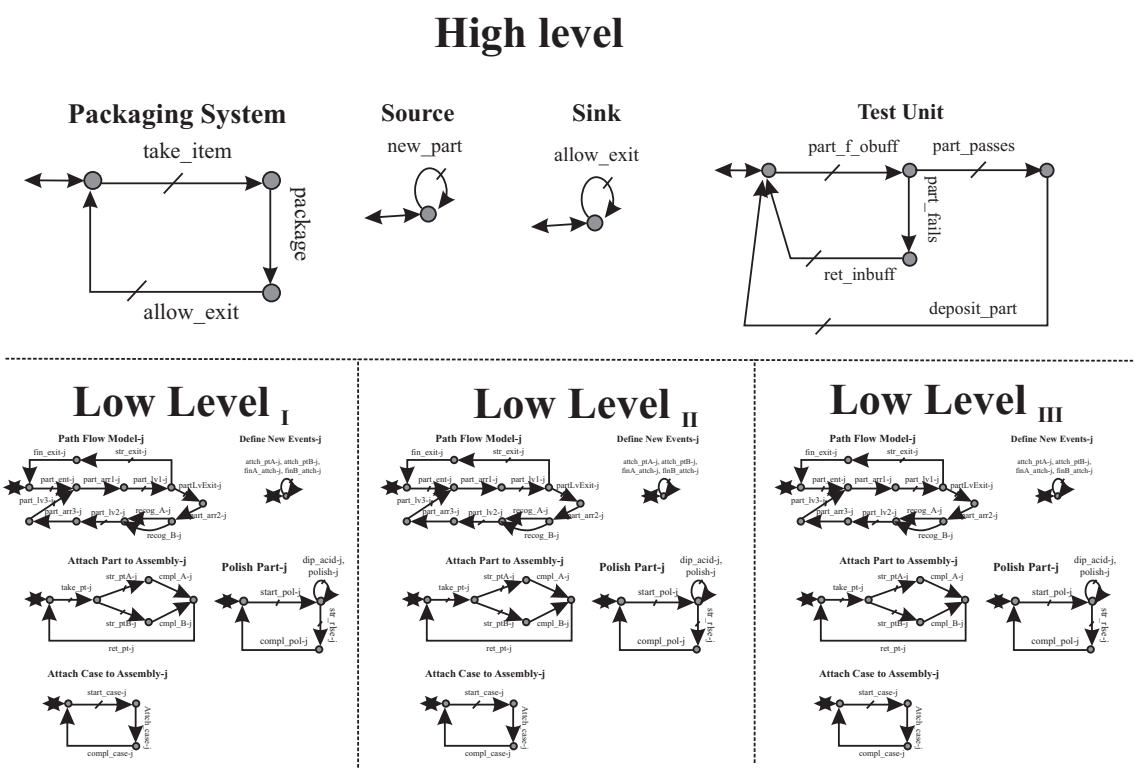


Figure 9.5: Plant Models for Parallel System

9.2 The Final System

Now that we have defined the individual components of the *system,* it is time to put everything together. We start by examining the j^{th} *low level subsystem.* This is shown in Figure 9.9, where we have labelled which DES belong to the j^{th} *low level subsystem,* the j^{th} low level plant, and the j^{th} low level supervisor (each formed by the synchronous product of the indicated DES). We can now assemble the complete parallel system shown in Figure 9.10, minus DES **Ensure_matFb** which we will introduce in Section 9.3. In addition to

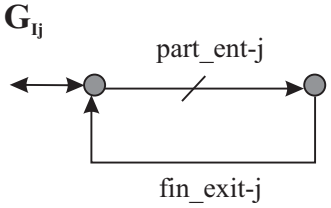


Figure 9.6: Interface Model for *Low Level j*

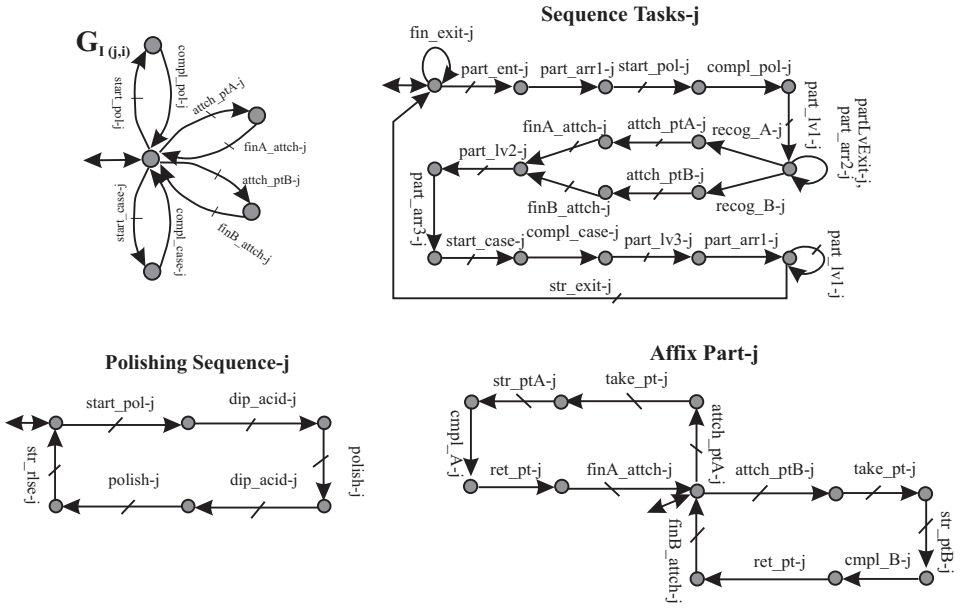


Figure 9.7: Supervisors for *Low Level j*

the *low level subsystems,* Figure 9.10 shows which DES belong to the *high level subsystem,* high level plant, and the high level supervisor (each formed by the synchronous product of the indicated DES).

We now define the *flat system,* the *flat plant,* and the *flat supervisor* as follows:

$$
\begin{aligned}
G &= G_H||_s G_{L_{\mathrm{I}}}||_s G_{L_{\mathrm{II}}}||_s G_{L_{\mathrm{III}}}||_s G_{I_{\mathrm{I}}}||_s G_{I_{\mathrm{II}}}||_s G_{I_{\mathrm{III}}} \\
\textbf{Plant} &:= \mathcal{G}_H||_s \mathcal{G}_{L_{\mathrm{I}}}||_s \mathcal{G}_{L_{\mathrm{II}}}||_s \mathcal{G}_{L_{\mathrm{III}}} \\
\textbf{Sup} &:= \mathcal{S}_H||_s \mathcal{S}_{L_{\mathrm{I}}}||_s \mathcal{S}_{L_{\mathrm{II}}}||_s \mathcal{S}_{L_{\mathrm{III}}}||_s G_{I_{\mathrm{I}}}||_s G_{I_{\mathrm{II}}}||_s G_{I_{\mathrm{III}}}
\end{aligned}
$$

9.3 Evaluating Properties

Our next step is to verify that the *flat system* is nonblocking and that the *flat supervisor* is controllable for the *flat plant.* To achieve this, we will show that the system is *level-wise nonblocking and controllable,* and *interface consistent.* Our first step is to show that sets Σ_H, Σ_{R_j}, Σ_{A_j}, and Σ_{L_j} are pairwise disjoint. This can be seen by inspection of their definitions.

As we have a parallel system of degree $n = 3$, we must verify that the j^{th} *serial system extraction: subsystem form* are *serial level-wise nonblocking,* and *serial interface consistent,*

and that the j^{th} *serial system extraction: general form* is *serial level-wise controllable*. We start by defining the *serial extraction systems: system*(I), *system*(II), and *system*(III). *System*(I) is defined below, with the DES definitions shown in Figure 9.11, minus DES **Ensure_matFb** which we will introduce later in the section. The remaining systems are left as an exercise.

Parallel Subsystem Based Form:

$$\begin{aligned}
G_H(I) &:= G_H||_s G_{I_{\mathrm{II}}}||_s G_{I_{\mathrm{III}}} \\
G_L(I) &:= G_{L_{\mathrm{I}}} \\
G_I(I) &:= G_{I_{\mathrm{I}}} \\
\Sigma_H(I) &:= [\dot{\cup}_{k\in\{\mathrm{II},\mathrm{III}\}}\Sigma_{I_k})]\,\dot{\cup}\,\Sigma_H \\
\Sigma_L(I) &:= \Sigma_{L_{\mathrm{I}}} \\
\Sigma_R(I) &:= \Sigma_{R_{\mathrm{I}}} \\
\Sigma_A(I) &:= \Sigma_{A_{\mathrm{I}}} \\
\Sigma(I) &:= \Sigma_H(I)\,\dot{\cup}\,\Sigma_L(I)\,\dot{\cup}\,\Sigma_R(I)\,\dot{\cup}\,\Sigma_A(I)
\end{aligned}$$

Parallel General Form:

$$\begin{aligned}
\mathcal{G}_H(I) &:= \mathcal{G}_H||_s G_{I_{\mathrm{II}}}||_s G_{I_{\mathrm{III}}} \\
\mathcal{S}_H(I)) &:= \mathcal{S}_H \\
\mathcal{G}_L(I) &:= \mathcal{G}_{L_{\mathrm{I}}} \\
\mathcal{S}_L(I) &:= \mathcal{S}_{L_{\mathrm{I}}}
\end{aligned}$$

The reader should note that the *interfaces* and the three *low level subsystems,* plants, and supervisors are identical up to event relabelling, and thus isomorphic. This means that the *serial level-wise nonblocking and controllability,* and *serial interface consistency* verifications for them will be identical. We thus need to evaluate only one *low level,* and the results will apply to all three.

We now apply our software tool to the *serial extraction systems* and we find that the *high level* is blocking. The reason for this is that the supervisor for the input buffer does not take into account material feedback. This can be seen by analysing the sequence of events shown in Figure 9.12, and seeing how it leads DES **Test Unit, in_buff,** and **out_buff** to a blocking

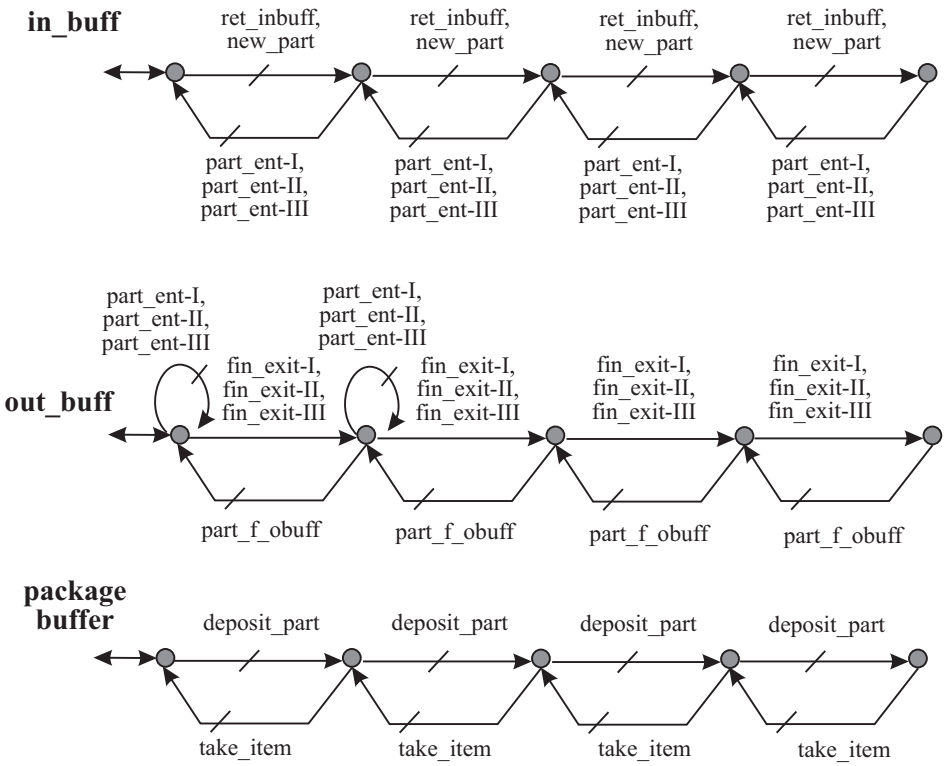


Figure 9.8: Supervisors for *High Level*

state.[1] We see seven *new_part* events occur, but only three *part_ent-*{*I, II, III* } events. This means **in_buff** is now full, preventing any *ret_inbuff* events from occurring. We next note that one *part_f_obuff* event has occurred, but three *fin_exit-*{*I, II, III* } events. This means **out_buff** is in the third state from the left. This state disables events *part_ent-*{*I, II, III* }. These events can't be re-enabled until an event *part_f_obuff*. However, a *part_f_obuff* can't occur until event *ret_inbuff* occurs, but this is prevented by **in_buff** as mentioned above, since the buffer is full. The buffer can only be emptied by events *part_ent-*{*I, II, III* } which are disabled by **out_buff**. The problem is that the input buffer didn't ensure there was space to receive a part rejected by the test unit. It allowed the buffer to be filled with new parts.

To properly handle material feedback, we add the supervisor shown in Figure 9.13. This supervisor would be added to the *high level supervisor*, as shown in Figure 9.10. Figure 9.11 shows the new *system*(I). The new supervisor prevents deadlock by ensuring that there can

[1]The material feedback oversight was left in purposely to show that the interface structure alone does not guarantee nonblocking.

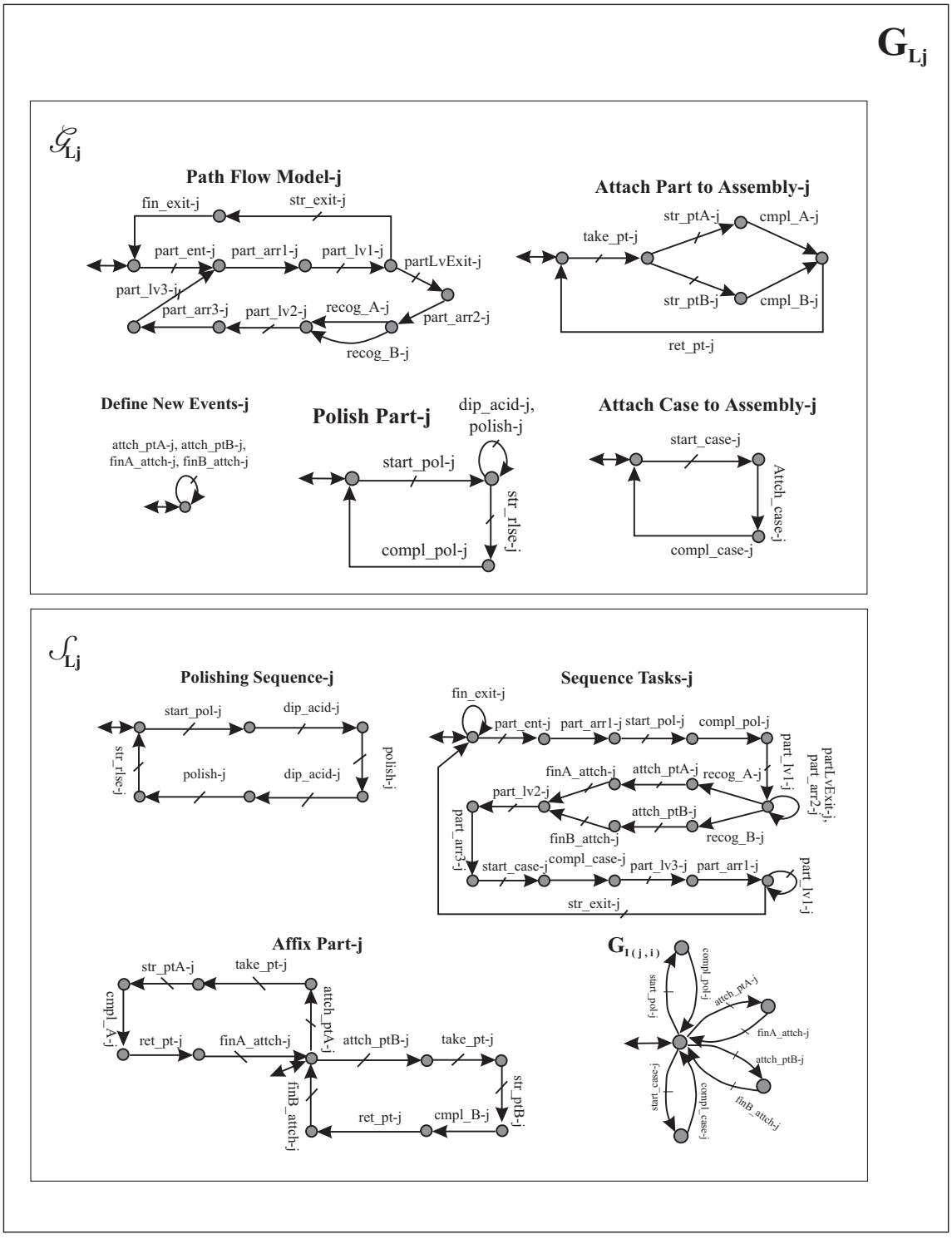


Figure 9.9: *Low Level Subsystem j*

High level Subsystem

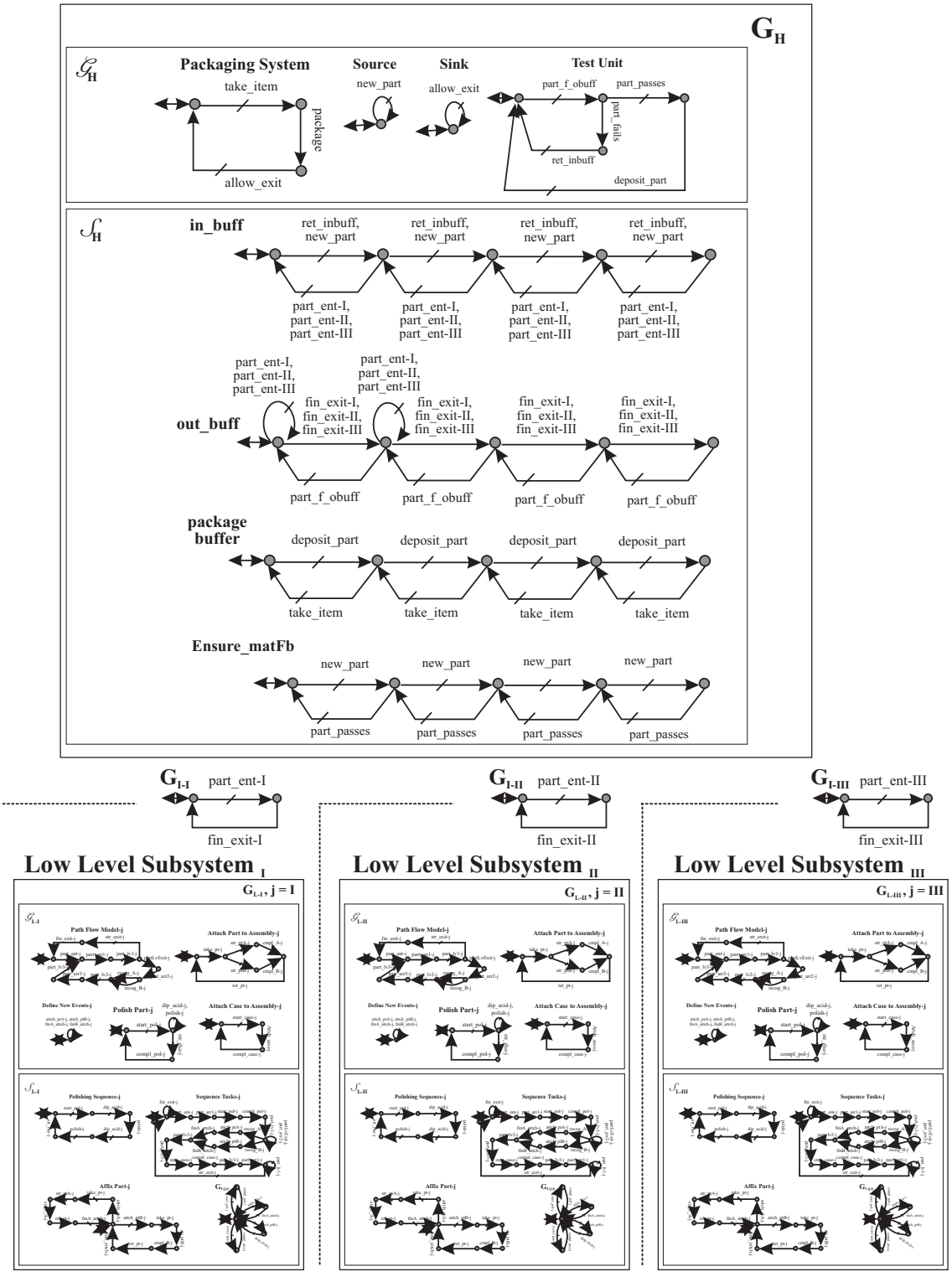
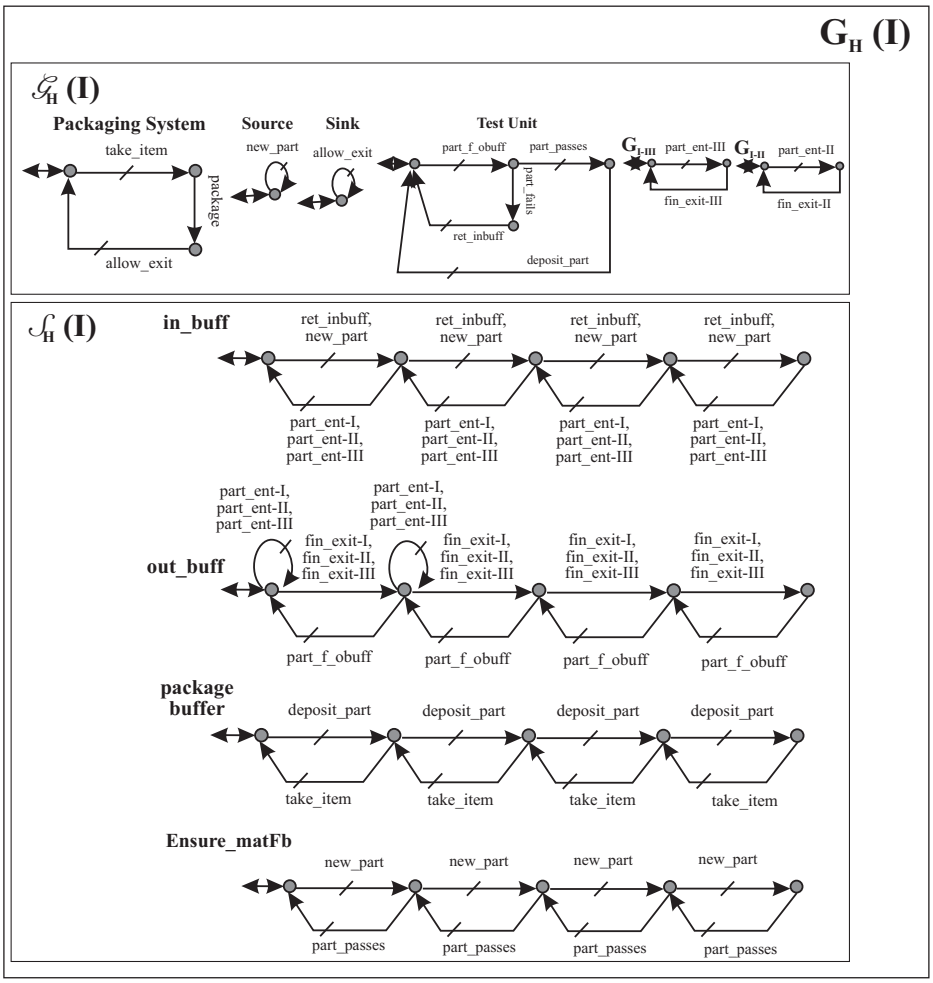
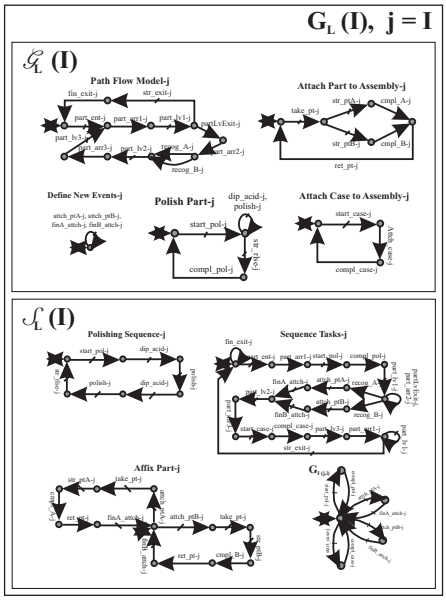


Figure 9.10: Complete Parallel System



High level Subsystem (I)

Low Level Subsystem (I)



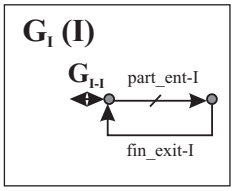


Figure 9.11: *Serial Extraction System I*

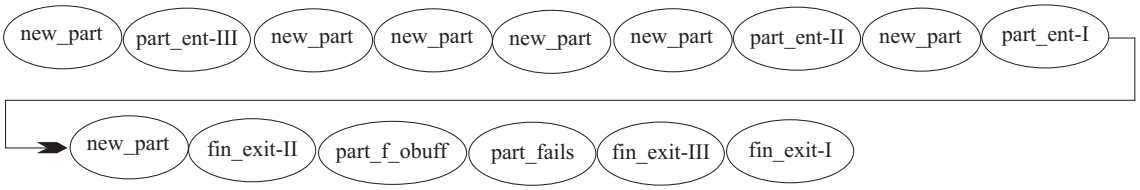


Figure 9.12: Deadlock Sequence

be at most four parts in the system between the input buffer and the test unit. After four *new_part* events occur, the event is disabled until a part passes the tester (event *part_passes*) and thus can't be returned to the input buffer.

We now apply our research tool to the *serial extraction systems* and we find that each system is *serial level-wise nonblocking and controllable,* and *serial interface consistent.* We can thus conclude by Theorems 3 and 4, that the *flat system* is nonblocking and that the *flat supervisor* is controllable for the *flat plant.*

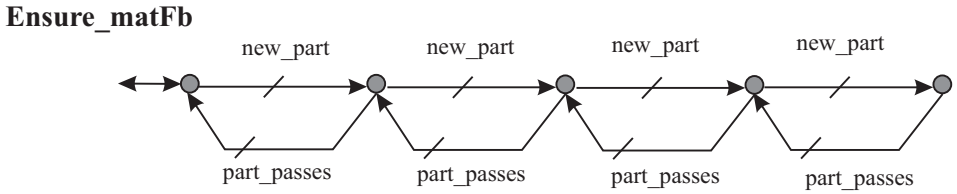


Figure 9.13: Material Feedback Supervisor

9.4 Comparison to Standard Method

The above computation was run on a 750MHz Athlon system, with 512MB of RAM, 2GB of swap, and running Redhat Linux 6.2. The *high level* consisted of 3120 states and each *low level* of 35 states. The computation took 0.15s to run and required 5MB of memory.

A standard nonblocking verification was done on the *flat system* which consists of 5,702,550 states. The computation ran for 40 minutes, required 850MB of memory, and found the system to be nonblocking.[2] In short, the standard method took 16438 times longer, and required 170 times more memory.

[2]A controllability check was also run using standard methods and the *flat supervisor* was found to be controllable for the *flat plant.*

From this example, we can see that the *interface method* not only can offer a significant reduction in verification time and required resources, but that it also greatly improves the re-usability of the system. For example, a system designer would only need to design supervisors for the manufacturing unit once, and could then use them in each instantiation. Treating the manufacturing system as a *low level subsystem,* we only have to verify the subsystem once even though we use it in multiple places. In addition, we can change the *low level subsystem* without affecting the *high level* as long as the *interface* remains the same, and the *low level* remains *serial nonblocking, controllable,* and *interface consistent.*[3] This offers a potentially great savings in design and verification time.

9.5 Applying the HISC Method

The important thing to remember about applying the HISC method to a system, is that it is intended as a verification **and** design method. In other words, the idea is to design your system and plant models with the HISC method in mind, as opposed to taking an existing design and modifying it to fit the HISC architecture. By designing from the beginning for the HISC method, one will be able to achieve the many benefits listed in Section 1.2. One can still modify an existing design, of course.

In this chapter, and in Chapter 5 we have presented detailed tutorials on how to design for the HISC method so we will not present a set of rules here. Instead, we will speak in more general terms.

The first step in designing for this method, is to decide what system behavior belongs on the *high level,* and which part of the system behavior belongs in the $n \geq 1$ *low levels.* In general, a system's behavior is of three types: local, interface, and interacting. Local behavior is behavior that is relevant and self-contained to/in a specific part (component) of the system. This usually deals with the internal state and operations of the component. For instance, the internal behavior of a given machine in a manufacturing unit would be an example of local behavior. Say the machine can perform *tasks A* and *B.* How the machine performs these tasks, and the internal state of the machine would be the local behavior of the machine. Once the machine starts a task, it's oblivious to the rest of the system until the task is complete, and thus this behavior is clearly self-contained. In other words, local

[3] More correctly, the *low level* continues to satisfy its portion of these properties.

behavior often answers the question as to "how" the component operates and performs its tasks. This behavior will belong to a specific *low level.*

This brings us to interface behavior. Interface behavior answers the question "what" can the component do and contains status information about the performance of the interface tasks. In other words, the interface behavior (represented by the interface DES) is the link between the local and interacting behavior. For the machine above, the interface behavior would tell us that the machine can perform two tasks, and what the possible results are of performing these tasks (ie. for *task A,* was it completed successfully or not).

The third type of behavior, interacting, contains information that is global, and thus not specific to any one component. It usually answers the question "when" does a component perform a specific task and thus "interact" with the rest of the system. Interacting behavior only cares that the task is started, and results of performing the task so that it can decide what to do next. In other words, interacting behavior is concerned about global operating decisions that are then translated into local behavior. This behavior belongs to the high level.

With these types of behavior in mind, the designer would then partition the system appropriately, and then design supervisors for the *high level* and each of the *low levels.* The key here is to keep in mind the limit of scope of operation permitted by the HISC method. Low level supervisors are concerned about local behavior and implementing the relevant (to that *low level*) tasks specified by the interface behavior. High level supervisors are concerned with interacting behavior. However, the control actions can only be expressed in terms of *high level events* and *interface events.* In other words, control actions are expressed in terms of *high level events* and in manipulating the interface to achieve the desired goal.

Chapter 10

Parallel Case Algorithms

In this chapter, we provide evaluation methods for definitions *Interface Consistent, level-wise nonblocking,* and *level-wise controllable.* Our purpose here is to provide enough details that the definitions can be evaluated, but skipping over aspects that are straightforward or have been investigated elsewhere (e.g. controllability algorithms). We will be discussing a naive "proof of concept" method (horribly inefficient but easy to construct based on existing algorithms), and we will leave the investigation of detailed efficient algorithms for later work. The reason for this is partly time constraints but primarily because the strength of our method doesn't depend on the individual algorithms but on the ability to decompose our system into subsystems and perform local checks. Finally, we present a complexity analysis for verifying a *parallel interface system.*

10.1 Preliminary Definitions

In the following sections, we take index j to have range $\{1, \ldots, n\}$ ($n \geq 1$ the degree of our *parallel interface system*) and assume that we will be given G_H, G_{L_j}, $\mathcal{G}_H$, $\mathcal{S}_H$, $\mathcal{G}_{L_j}$, $\mathcal{S}_{L_j}$, Σ_H, Σ_{L_j}, Σ_{R_j}, Σ_{A_j}, Σ_u, Σ_c, and the maps $\mathbf{Answer}_j : \Sigma_{R_j} \to \mathrm{Pwr}(\Sigma_{A_j})$ (see definition on page 25), but will have to construct DES G_{I_j}. Also, we assume that all DES are deterministic and have finite state and event sets.

In particular, we define notation for the following DES:

$$\begin{aligned} G_H &:= (Y_H, \Sigma_{G_H}, \delta_H, y_{H_o}, Y_{H_m}) \\ G_{L_j} &:= (Y_{L_j}, \Sigma_{G_{L_j}}, \delta_{L_j}, y_{L_{o_j}}, Y_{L_{m_j}}) \end{aligned}$$

$$G_{I_j} \quad := \quad (X,\, \Sigma_{G_I},\, \xi,\, x_o,\, X_m)$$

We next define our system's event set as

$$\Sigma := \cup_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \cup \Sigma_{R_k} \cup \Sigma_{A_k}) \cup \Sigma_H$$

Finally, we will restrict all our *interfaces* to *star interfaces*, as we did in Chapter 6. We will construct them "correct by design" as discussed in Section 6.2.

10.2 Evaluating *Interface Consistent* Definition

We now evaluate that the system satisfies the *interface consistent* definition. Our first step will be to evaluate the implied condition $\Sigma := \dot\cup_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \,\dot\cup\, \Sigma_{R_k} \,\dot\cup\, \Sigma_{A_k}) \,\dot\cup\, \Sigma_H$. This includes two implicit assumptions. The first is that $\Sigma = \cup_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \cup \Sigma_{R_k} \cup \Sigma_{A_k}) \cup \Sigma_H$. This is automatic from our definitions in Section 10.1.

The second implicit assumption is that the event sets are pairwise disjoint. This means checking the four statements below. Again, this is straightforward so we won't present a specific algorithm.

1. $\Sigma_{R_j} \cap \Sigma_{A_j} = \emptyset$
2. $(\Sigma_{R_j} \cup \Sigma_{A_j}) \cap [\cup_{k\in\{1,\ldots,j-1,j+1,\ldots,n\}}(\Sigma_{R_k} \cup \Sigma_{A_k})] = \emptyset$
3. $\Sigma_{L_j} \cap [\cup_{k\in\{1,\ldots,n\}}(\Sigma_{R_k} \cup \Sigma_{A_k})] = \emptyset$
4. $\Sigma_H \cap [\cup_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \cup \Sigma_{R_k} \cup \Sigma_{A_k})] = \emptyset$

All that remains now is to verify that the j^{th} *serial system extraction: subsystem form* is *serial interface consistent*. To achieve this, we simply construct the j^{th} *serial system extraction* as specified in Section 7.2 and then apply the *serial interface consistent* algorithm from Chapter 6. DES such as $G_H(j) := G_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}}\ ||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n}$ can be constructed using the CTCT **sync** operator (see [60]) repeatedly, but modifying it to require that the event set of a given DES be specified explicitly, instead of taken to be the event labels of transitions in that DES.

10.3 Evaluating *Level-wise Nonblocking* Definition

We now evaluate the *level-wise nonblocking* definition. Our first step will be to evaluate the implied condition $\Sigma := \dot{\bigcup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}) \dot{\cup} \Sigma_H$. This is the same as in Section 10.2.

All that remains now is to verify that the j^{th} *serial system extraction: subsystem form* is *serial level-wise nonblocking*. To achieve this, we simply construct the j^{th} *serial system extraction* as specified in Section 7.2 and then apply the *serial level-wise nonblocking* algorithm from Chapter 6. DES such as $G_H(j) := G_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}} ||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n}$ can be constructed using the CTCT **sync** operator (see [60]) repeatedly, but modifying it to require that the event set of a given DES be specified explicitly, instead of taken to be the event labels of transitions in said DES.

10.4 Evaluating *Level-wise Controllable* Definition

We now evaluate the *level-wise controllable* definition. Our first step will be to evaluate the implied condition $\Sigma := \dot{\bigcup}_{k\in\{1,\ldots,n\}}(\Sigma_{L_k} \dot{\cup} \Sigma_{R_k} \dot{\cup} \Sigma_{A_k}) \dot{\cup} \Sigma_H$. This is the same as in Section 10.2.

All that remains now is to verify that the j^{th} *serial system extraction: general form* is *serial level-wise controllable*. To achieve this, we simply construct the j^{th} *serial system extraction* as specified in Section 8.2 and then apply the *serial level-wise controllable* algorithm from Chapter 6. DES such as $\mathcal{G}_H(j) := \mathcal{G}_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}}||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n}$ can be constructed using the CTCT **sync** operator (see [60]) repeatedly, but modifying it to require that the event set of a given DES be specified explicitly, instead of taken to be the event labels of transitions in said DES.

10.5 Software Tool

The software tool we discussed in Section 4.3.4 does not contain any algorithms specifically for *parallel interface systems*, only for serial systems. This is due to time constraints and the fact that almost all parallel case conditions can be evaluated by constructing the *serial system extractions* and then using the software's serial case algorithms.

10.6 Complexity Analysis

From the above sections, we see that to verify that a *parallel interface system* satisfies the *level-wise nonblocking, level-wise controllable,* and *interface consistent* definitions, we must perform the following tasks:

Event Set Disjoint Properties:

1) Verify that sets Σ_H, $\Sigma_{L_1}, \ldots, \Sigma_{L_n}$, $\Sigma_{R_1}, \ldots, \Sigma_{R_n}$ and $\Sigma_{A_1}, \ldots, \Sigma_{A_n}$ are pairwise disjoint. This means evaluating:

$$\begin{aligned}
\Sigma_H \cap \Sigma_a &= \emptyset, \quad a \in \{\Sigma_{L_1}, \ldots, \Sigma_{L_n}, \Sigma_{R_1}, \ldots, \Sigma_{R_n}, \Sigma_{A_1}, \ldots, \Sigma_{A_n}\} \\
\Sigma_{L_1} \cap \Sigma_a &= \emptyset, \quad a \in \{\Sigma_{L_2}, \ldots, \Sigma_{L_n}, \Sigma_{R_1}, \ldots, \Sigma_{R_n}, \Sigma_{A_1}, \ldots, \Sigma_{A_n}\} \\
&\vdots \\
\Sigma_{A_{(n-2)}} \cap \Sigma_a &= \emptyset, \quad a \in \{\Sigma_{A_{(n-1)}}, \Sigma_{A_n}\} \\
\Sigma_{A_{(n-1)}} \cap \Sigma_{A_n} &= \emptyset
\end{aligned}$$

Serial Extraction System Properties:

2) For the n *serial extraction systems*, we must verify that they are *Interface Consistent, level-wise nonblocking,* and *level-wise controllable* using the algorithms from Chapter 6.

As in Section 6.6.2, we let m be the number of components to be verified. The components are the *high level* and the n *low levels*, thus $m = n + 1$. We also assume that the statespace of each component and the cardinality of the system's event set (Σ) are bounded, with upper bounds $N \geq 0$ and $N_\Sigma \geq 0$ respectively (ie. $x \leq N$ and $|\Sigma| \leq N_\Sigma$). We further assume that the cardinality of the other event sets is each bounded by $N_{\Sigma'} \geq 0$ (ie. $|\Sigma_H| \leq N_{\Sigma'}$).

10.6.1 Analysing Event Set Disjoint Properties

To analyse task 1 above, we need to do multiple empty intersection tests. Examining the definition of task 1, we see that we first have to perform $3n$ tests (against Σ_H), then $3n-1$,

down to one test ($\Sigma_{A_{(n-1)}} \cap \Sigma_{A_n}$). So, the total number of tests, labelled *EI*, is

$$\begin{aligned} EI &= 3n + (3n-1) + \ldots + 2 + 1 = \Sigma_{i=1}^{3n} i \\ &= \frac{3n(3n+1)}{2} \quad \text{using well know identity } \Sigma_{i=1}^{k} i = \tfrac{k(k+1)}{2} \\ &= \frac{9}{2}m^2 - \frac{15}{2}m + 3 \quad \text{after substituting } n = m-1 \text{ and simplifying.} \end{aligned}$$

Now, according to Rudie [51], the test that the intersection of sets S and T is empty is $\mathbf{O}(|S||T|)$. As each of our sets is bounded by $N_{\Sigma'}$, the test is $\mathbf{O}(N_{\Sigma'}^2)$. We perform this test *EI* times; thus the whole process is $\mathbf{O}(EI \cdot N_{\Sigma'}^2) = \mathbf{O}((\frac{9}{2}m^2 - \frac{15}{2}m + 3)N_{\Sigma'}^2) = \mathbf{O}(\frac{9}{2}m^2N_{\Sigma'}^2 - \frac{15}{2}mN_{\Sigma'}^2 + 3N_{\Sigma'}^2)$. This reduces to $\mathbf{O}(m^2)$ as $N_{\Sigma'}$ is a constant.

10.6.2 Analysing Serial Extraction System Properties

To perform task 2, we must verify that *n serial extraction systems* are *serial interface consistent, serial level-wise nonblocking,* and *serial level-wise controllable* using the algorithms from Chapter 6. We know from Section 6.6.2 that verifying a *serial interface system* means applying our $\mathbf{O}(x^3) = \mathbf{O}(N^3)$ (x is the number of states the component has and is bounded by N) per component algorithm twice; once for the system's *high level* and once for its *low level.* Two analyse *n serial extraction systems*, we thus apply the per component algorithm $2n$ times. This means the time complexity of performing task 2 is $\mathbf{O}(2n \cdot N^3) = \mathbf{O}((2m-2)N^3) = \mathbf{O}(2mN^3 - 2N^3)$ after substituting for $n = m-1$. This reduces to $\mathbf{O}(m)$ as N is a constant.

10.6.3 Analysing Per System Algorithm

To analyse a complete *parallel interface system*, we must perform tasks 1 and 2. This means that our per system algorithm is $\mathbf{O}(\frac{9}{2}m^2N_{\Sigma'}^2 - \frac{15}{2}mN_{\Sigma'}^2 + 3N_{\Sigma'}^2 + 2mN^3 - 2N^3) = \mathbf{O}(\frac{9}{2}m^2N_{\Sigma'}^2 + (2N^3 - \frac{15}{2}N_{\Sigma'}^2)m + (3N_{\Sigma'}^2 - 2N^3))$, which reduces to $\mathbf{O}(m^2)$.

It's important to note that $\mathbf{O}(m^2)$ isn't the whole story as it hides some potentially large constants. As the number of components grows slowly, and $N_{\Sigma'}$ tends to be small in comparison to N, m would have to become quite large before the $m^2N_{\Sigma'}^2$ term would dominate the mN^3 term). Also, many development environments can often guarantee that most of the event sets are disjoint, making these checks unnecessary. For example, if *low level event sets* are equated with program variables local to a function, the compiler will

guarantee that these are unique. In practice, it is the per system algorithm that dominates the running time of this method but it's important to note that breaking a system into multiple components does come with a penalty that must be considered.

UPDATE: The analysis presented here relies on the assumption that the statespace of each component is bounded by the constant N. As long as this assumption is reasonable, the analysis is correct. For the DES $G_H, G_{L_1}, \ldots, G_{L_n}, G_{I_1}, \ldots, G_{I_n}$, this assumption is reasonable.

However, when analyzing the conditions *interface consistent, level-wise nonblocking,* and *level-wise controllable,* we must construct *serial extraction systems* (see sections 7.2 and 8.2) to analyze the corresponding serial conditions. For example, to verify that the *parallel interface system* is *interface consistent,* we must verify that all n *serial system extractions (subsystem form)* are *serial interface consistent.* To verify the latter condition, we must use the component $G_H(j) := G_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}}||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n}$ with the serial algorithms we developed in Chapter 6. Unlike the DES G_H, component $G_H(j)$ grows proportionally to n, thus the assumption that $G_H(j)$ is bounded by N is questionable. In this view, the above analysis is a bit too optimistic and is thus more in line with an average or best case analysis. This does not mean that the approach does not have great potential to scale. For a good scalability discussion, see [32].

10.6.4 Comparing to Monolithic Algorithm

From our discussion in Section 6.6.3, we saw that the monolithic algorithm was $\mathbf{O}(N^{2m})$. Comparing the complexity of our algorithm to the monolithic algorithm above, we see that our interface-based method scales significantly better at $\mathbf{O}(m^2)$ than the other at $\mathbf{O}(N^{2m})$. To illustrate this, let's examine the two algorithms for a few values of N, $N_{\Sigma'}$, and m. Table 10.1 shows the results for terms $T_1 = N^{2m}$, $T_2 = 2mN^3 - 2N^3$, and $T_3 = \frac{9}{2}m^2N_{\Sigma'}^2 - \frac{15}{2}mN_{\Sigma'}^2 + 3N_{\Sigma'}^2$. As we see, the *serial extraction systems* component (T_2) does dominate the time complexity of our algorithm. We also see that even for $m = 2$, our approach is six orders of magnitude better. To put this into perspective, if our algorithm ran for one hour, the monolithic algorithm would require 114 years! As m increases, we see very modest increases in complexity for our algorithm. However, even at $m = 9$, the monolithic approach is completely unusable.

It's important to note here that for $m = 1000$, this represents evaluating 1000 compo-

		$m = 2$			$m = 9$			$m = 1000$		
N	$N_{\Sigma'}$	T_1	T_2	T_3	T_1	T_2	T_3	T_1	T_2	T_3
10^3	10^2	10^{12}	2×10^9	6×10^4	10^{54}	1.60×10^{10}	3×10^6	10^{6000}	2×10^{12}	4.5×10^{10}
10^6	10^2	10^{24}	2×10^{18}	6×10^4	10^{108}	1.60×10^{19}	3×10^6	10^{12000}	2×10^{21}	4.5×10^{10}
10^6	10^3	10^{24}	2×10^{18}	6×10^6	10^{108}	1.60×10^{19}	3×10^8	10^{12000}	2×10^{21}	4.49×10^{12}

Table 10.1: Parallel Algorithm Comparison

nents. As each one can be evaluated separately, we could utilise a cluster of 100 workstations and process 100 components in parallel. For $N = 10^6$, this would bring down our time estimate to 2×10^{19}, only slightly more than the time for processing 9 components!

Chapter 11

AIP Example

To demonstrate the utility of our method, we now present its application to a large manufacturing system, the Atelier Inter-établissement de Productique (AIP). In this chapter, we introduce the AIP and describe its components. We present our control specifications and describe the systems structure. In the following chapters, we will describe individual subsystems and then discuss the results of applying our method to this example. All supervisors are designed as discussed in Section 5.5.

11.1 Overview of the AIP

In this section, we introduce the automated manufacturing system of the Atelier Inter-établissement de Productique (AIP), as described in [7] and [12]. The AIP, shown in Figure 11.1, is a highly automated manufacturing system consisting of a central loop (CL) and four external loops (EL), three assembly stations(AS), an input/output (I/O) station, and four inter-loop transfer units (TU). The I/O station is where the pallets enter and leave the system. Pallets can be of type 1 or of type 2, and it is assumed that the type of the pallet entering is random.

11.1.1 Assembly Stations

The structure of the assembly stations is shown in Figure 11.2. Each station consists of a robot to perform assembly tasks, an extractor to transfer the pallet from the conveyor loop to the robot, sensors to determine the location of the extractor, and a raising platform to present the pallet to the robot. The station also contains pallet sensors to detect a pallet

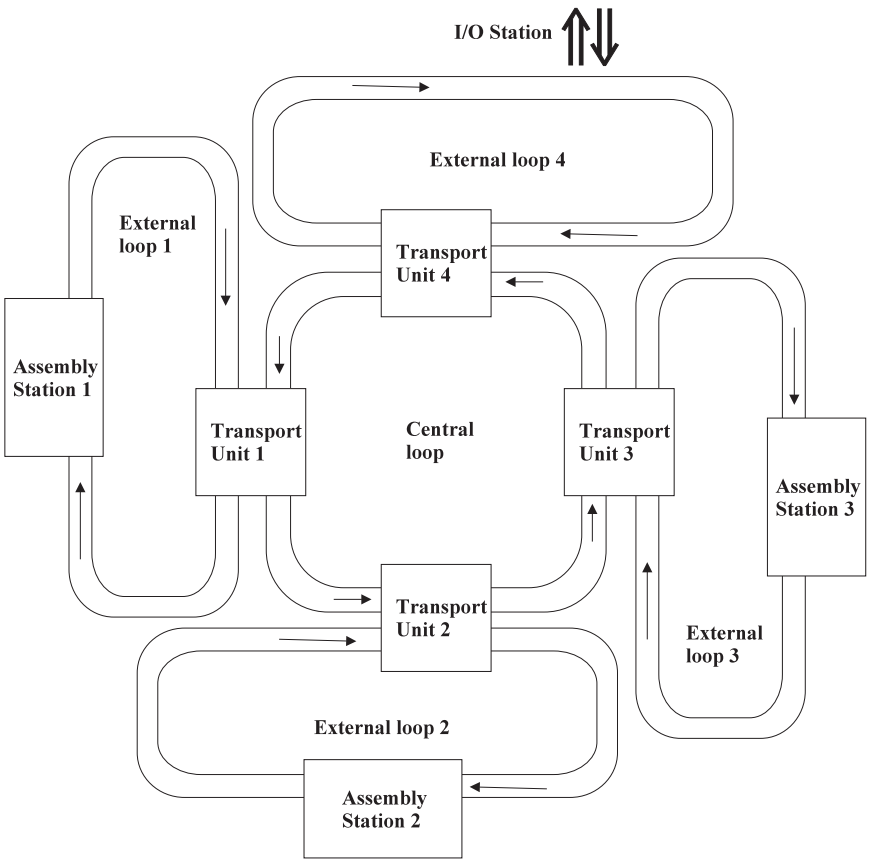


Figure 11.1: The Atelier Inter-établissement de Productique

at the pallet gate, the pallet stop, and to detect when a pallet has left the station. Finally, the assembly station contains a read/write (R/W) device to read and write to the pallet's electronic label. The pallet label contains information about the pallet type, error status, and assembly status (which tasks have been performed).

Whereas the assembly stations contain the same basic components, they differ with respect to functionality. Station 1 is capable of performing task1A and task1B, while station 2 can perform task2A and task2B. Station 3 can perform all four tasks as well as function as a repair station allowing an operator to repair a damaged pallet. The assembly stations also differ with respect to reliability. Stations 1 and 2 can break down and must be repaired, while station 3 is of higher quality and is assumed never to break down. Station 3 is used to substitute for the other stations when they are down.

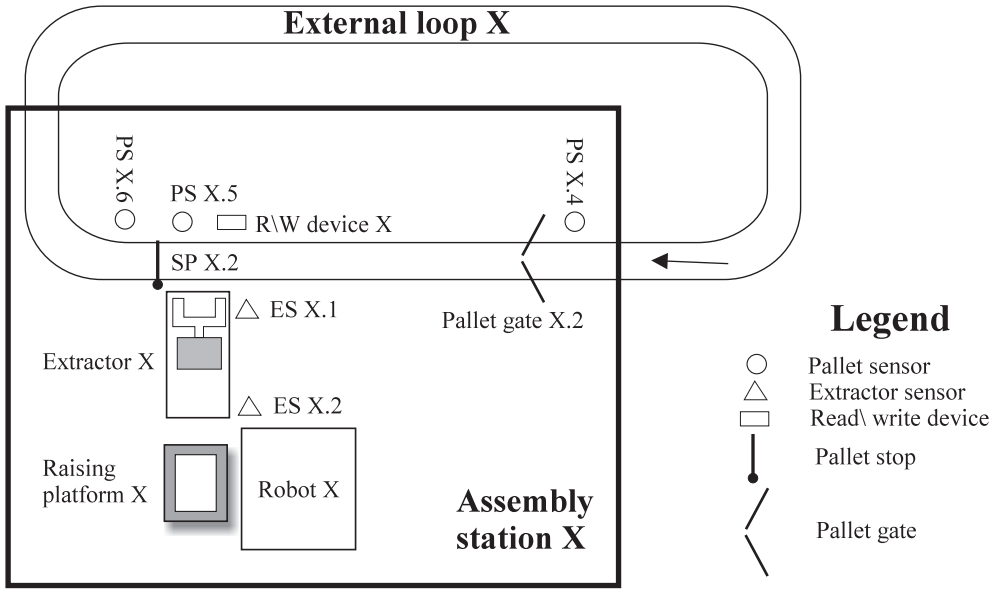


Figure 11.2: Assembly Station of External Loop $X = 1, 2, 3$.

11.1.2 Transport Units

The structure of the four identical transport units are shown in Figure 11.3. The transport units are used to transfer pallets between the central loop, and the external loops. Each one consists of a transport drawer which physically conveys the pallet between the two loops, plus sensors to determine the drawer's location. At each loop, the unit contains a pallet gate and a pallet stop, to control access to the unit from the given loop. The unit also contains multiple pallet sensors to detect when a pallet is at a gate, drawer, or has left the unit. Also, each unit contains an R/W device located before the central loop gate.

11.2 Control Specifications

For this example, we adopt the control specifications and assumptions used in [7] and [12] and restated below. To this we add specification 7 to make the assembly stations more interesting.

Assumptions:

1. The system is initially empty.

2. Two types of pallets are randomly introduced to the system, subjected to assembly

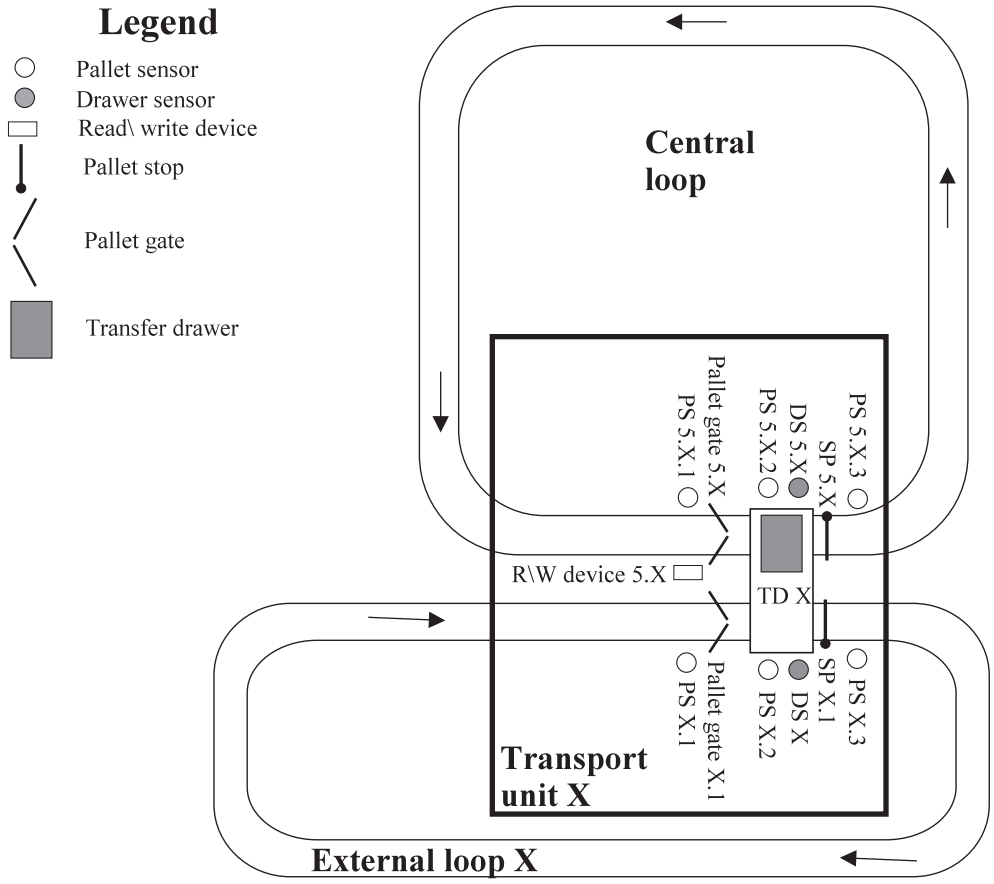


Figure 11.3: Transport Unit for External Loop $X = 1, 2, 3, 4$

operations, and then leave.

Specifications:

1. **Routing:** Pallets follow a certain route based on their type. A type 1 pallet must go first to AS1, then AS2 before leaving the system. Type 2 pallets go first to AS2, then AS1 before leaving the system. A pallet is not allowed to leave the system until all four assembly tasks have been successfully performed on it.

2. **Maximum capacity of external loops 1 and 2:** The maximum allowed number of pallets in either loop at a given time is one.

3. **Order of pallets exiting from system:** The pallets must exit the system in the following order: type 1, type 2, type 1, ...

4. **Assembly errors:** When a robot makes an assembly error, the pallet is marked damaged and routed to AS3 for maintenance. After maintenance, the pallet is returned to the original assembly station to undergo the assembly operation again.

5. **Assembly station breakdown:** The robots of external loops 1 and 2 are susceptible to breakdowns. When a station is down, pallets are routed to assembly station 3 which is capable of performing all tasks of the other two stations. When the failed station is repaired, all pallets not already in external loop 3 are rerouted to the original station.

6. **Maximum capacity of assembly stations:** To avoid collisions, only one pallet is allowed in a given station at a time.

7. **Assembly task ordering:** Assembly tasks are performed in a different order for pallets of different types. For pallets of type 1, task1A is performed before task1B, and task2A is performed before task2B. For pallets of type 2, task1B is performed before task1A, and task2B is performed before task2A.

11.3 System Structure

To cast the AIP into a *parallel interface system,* we break the system down into a *high level,* and seven *low levels* corresponding to the three assembly stations and four transport Units, as shown in Figure 11.4. The *high level,* and each *low level* are described in the following chapters.

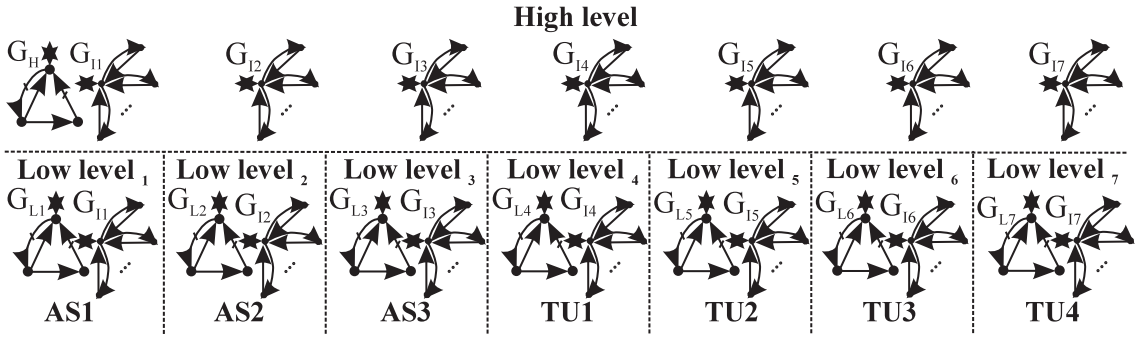


Figure 11.4: Structure of Parallel System

The alphabet partition that we will use is $\Sigma := \dot{\cup}_{j\in\{1,\ldots,7\}}[\Sigma_{L_j}\dot{\cup}\Sigma_{R_j}\dot{\cup}\Sigma_{A_j}] \dot{\cup} \Sigma_H$, with the individual event sets defined below where $r = \text{TU1}, \text{TU2}$ when $v = 4, 5$, and $k = \text{AS1}, \text{AS2}$

when $w = 1, 2$, respectively:

Σ_H = {*DetStnsUp*, *IsPalletCL.TU1*, *IsPalletCL.TU2*, *IsPalletCL.TU3*, *IsPalletCL.TU4*,
IsPalletEL.TU1, *IsPalletEL.TU2*, *IsPalletEL.TU3*, *IsPalletEL.TU4*,
NoPalletCL.TU1, *NoPalletCL.TU2*, *NoPalletCL.TU3*, *NoPalletCL.TU4*,
NoPalletEL.TU1, *NoPalletEL.TU2*, *NoPalletEL.TU3*, *NoPalletEL.TU4*,
PalletArvGEL_2.AS3, *QPalletAtCL.TU1*, *QPalletAtCL.TU2*, *QPalletAtCL.TU3*,
QPalletAtCL.TU4, *QPalletAtEL.TU1*, *QPalletAtEL.TU2*, *QPalletAtEL.TU3*,
QPalletAtEL.TU4, *QStnUp.AS1*, *QStnUp.AS2*, *StnDwn.AS1*, *StnDwn.AS2*, *StnUp.AS1*,
StnUp.AS2}

Σ_{L_1} = {*DoneRead.AS1*, *DoneWrite.AS1*, *ExtrArvConv.AS1*, *ExtrArvPlatf.AS1*, *GClosesEL_2.AS1*,
GOpensEL_2.AS1, *IsType1.AS1*, *IsType2.AS1*, *MvExtrToConv.AS1*, *MvExtrToPlatf.AS1*,
PalletArvGEL_2.AS1, *PalletLvAS.AS1*, *PalletLvGEL_2.AS1*, *PArvAtExtractor.AS1*,
PStopClosesEL_2.AS1, *PStopOpensEL_2.AS1*, *QType.AS1*, *ReadLabel.AS1*,
RelPallet.AS1, *WrtCplT1A_1 B.AS1*, *WrtErr1A.AS1*, *WrtErr1B.AS1*,
ProcTyp1.AS1, *ProcTyp2.AS1*, *RTasksCpl.AS1*,
RobDwn.AS1, *AssmbErrA.AS1*, *AssmbErrB.AS1*,
AError1A.AS1, *AError1B.AS1*, *RobRprCpl.AS1*, *RtaskCpl1A.AS1*,
RtaskCpl1B.AS1, *Rtimeout.AS1*, *StrRobRepair.AS1*, *StrRtask1A.AS1*,
StrRtask1B.AS1, *StrRtimer.AS1*}

Σ_{L_2} = {*DoneRead.AS2*, *DoneWrite.AS2*, *ExtrArvConv.AS2*, *ExtrArvPlatf.AS2*,
GClosesEL_2.AS2, *GOpensEL_2.AS2*, *IsType1.AS2*, *IsType2.AS2*,
MvExtrToConv.AS2, *MvExtrToPlatf.AS2*, *PalletArvGEL_2.AS2*, *PalletLvAS.AS2*,
PalletLvGEL_2.AS2, *PArvAtExtractor.AS2*, *PStopClosesEL_2.AS2*, *PStopOpensEL_2.AS2*,
QType.AS2, *ReadLabel.AS2*, *RelPallet.AS2*, *WrtCplT2A_2B.AS2*,
WrtErr2A.AS2, *WrtErr2B.AS2*, *ProcTyp1.AS2*, *ProcTyp2.AS2*,
RTasksCpl.AS2, *RobDwn.AS2*, *AssmbErrA.AS2*, *AssmbErrB.AS2*,
AError2A.AS2, *AError2B.AS2*, *RobRprCpl.AS2*, *RtaskCpl2A.AS2*,
RtaskCpl2B.AS2, *Rtimeout.AS2*, *StrRobRepair.AS2*, *StrRtask2A.AS2*,
StrRtask2B.AS2, *StrRtimer.AS2*}

Σ_{L_3} = {*DetNProcPallet.AS3*, *DoneRead.AS3*, *DoneReset.AS3*, *DoneWrite.AS3*,
ExtrArvConv.AS3, *ExtrArvPlatf.AS3*, *GClosesEL_2.AS3*, *GOpensEL_2.AS3*,
MvExtrToConv.AS3, *MvExtrToPlatf.AS3*, *PalletLvAS.AS3*, *PalletLvGEL_2.AS3*,
PArvAtExtractor.AS3, *PStopClosesEL_2.AS3*, *PStopOpensEL_2.AS3*, *ReadLabel.AS3*,

RelPallet.AS3, *ResetAssmbInfo.AS3*, *WrtCplT1A_1B.AS3*, *WrtCplT2A_2B.AS3*,
WrtErr1A.AS3, *WrtErr1B.AS3*, *WrtErr2A.AS3*, *WrtErr2B.AS3*, *DoRepPallet.AS3*,
PalletRepaired.AS3, *RepNotNeeded.AS3*, *MaintCpl.AS3*, *SigMaintT1A.AS3*,
SigMaintT1B.AS3, *SigMaintT2A.AS3*, *SigMaintT2B.AS3*, *QError.AS3*, *NoError.AS3*,
IsErr1A.AS3, *IsErr1B.AS3*, *IsErr2A.AS3*, *IsErr2B.AS3*, *NoErr.AS3*, *QErr1A.AS3*,
QErr1B.AS3, *QErr2A.AS3*, *QErr2B.AS3*, *DetNProcPallet.AS3*, *CplT2A_2B.AS3*,
CplT1A_1B.AS3, *AssmbErr2A.AS3*, *AssmbErr2B.AS3*, *AssmbErr1A.AS3*, *AssmbErr1B.AS3*,
IsCpl.AS3, *IsType1.AS3*, *IsType2.AS3*, *NotCpl.AS3*, *QCplT1A_1B.AS3*, *QCplT2A_2B.AS3*,
QType.AS3, *ProcTyp1AsR1.AS3*, *ProcTyp2AsR2.AS3*, *ProcTyp1AsR2.AS3*,
ProcTyp2AsR1.AS3, *RTasksCpl.AS3*, *RtaskCpl1A.AS3*, *RtaskCpl1B.AS3*, *RtaskCpl2A.AS3*,
RtaskCpl2B.AS3, *StrRtask1A.AS3*, *StrRtask1B.AS3*, *StrRtask2A.AS3*, *StrRtask2B.AS3*}

Σ_{L_v} = {*DoneRead.r*, *DrwArvCL.r*, *DrwArvEL.r*, *DrwAtCL.r*,
DrwAtEL.r, *GClosesCL.r*, *GClosesEL_1.r*, *GOpensCL.r*,
GOpensEL_1.r, *MvDrwToCL.r*, *MvDrwToEL.r*, *PalletArvDrwCL.r*,
PalletArvDrwEL.r, *PalletLvGCL.r*, *PalletLvGEL_1.r*, *PalletLvTUAtCL.r*,
PalletLvTUAtEL_1.r, *PStopClosesCL.r*, *PStopClosesEL_1.r*, *PStopOpensCL.r*,
PStopOpensEL_1.r, *QDrwLoc.r*, *ReadLabel.r*, *QError.r*,
NoError.r, *IsErr1A.r*, *IsErr1B.r*, *IsErr2A.r*, *IsErr2B.r*,
NoErr.r, *QErr1A.r*, *QErr1B.r*, *QErr2A.r*, *QErr2B.r*,
QOpNeeded.r, *OpNeeded.r*, *NotOpNeeded.r*, *IsCpl.r*, *IsType1.r*, *IsType2.r*, *NotCpl.r*,
QCplT1A_1B.r, *QCplT2A_2B.r*, *QType.r*}

Σ_{L_6} = {*DoneRead.TU3*, *DrwArvCL.TU3*, *DrwArvEL.TU3*, *DrwAtCL.TU3*,
DrwAtEL.TU3, *GClosesCL.TU3*, *GClosesEL_1.TU3*, *GOpensCL.TU3*,
GOpensEL_1.TU3, *MvDrwToCL.TU3*, *MvDrwToEL.TU3*, *PalletArvDrwCL.TU3*,
PalletArvDrwEL.TU3, *PalletLvGCL.TU3*, *PalletLvGEL_1.TU3*, *PalletLvTUAtCL.TU3*,
PalletLvTUAtEL_1.TU3, *PStopClosesCL.TU3*, *PStopClosesEL_1.TU3*, *PStopOpensCL.TU3*,
PStopOpensEL_1.TU3, *QDrwLoc.TU3*, *ReadLabel.TU3*, *SkipDwnOpChk.TU3*,
QError.TU3, *NoError.TU3*, *IsErr1A.TU3*, *IsErr1B.TU3*, *IsErr2A.TU3*, *IsErr2B.TU3*,
NoErr.TU3, *QErr1A.TU3*, *QErr1B.TU3*,
QErr2A.TU3, *QErr2B.TU3*, *QDwnOpNeeded_1D.TU3*, *QDwnOpNeeded_2D.TU3*,
QDwnOpNeeded_BD.TU3, *DwnOpNeeded.TU3*, *NotDwnOpNeeded.TU3*,
IsCpl.TU3, *IsType1.TU3*, *IsType2.TU3*, *NotCpl.TU3*,
QCplT1A_1B.TU3, *QCplT2A_2B.TU3*, *QType.TU3*}

Σ_{L_7} = {*CorrType.TU4*, *DoneRead.TU4*, *DrwArvCL.TU4*, *DrwArvEL.TU4*,
DrwAtCL.TU4, *DrwAtEL.TU4*, *GClosesCL.TU4*, *GClosesEL_1.TU4*,
GOpensCL.TU4, *GOpensEL_1.TU4*, *IsCpl.TU4*, *IsType1.TU4*,
IsType2.TU4, *MvDrwToCL.TU4*, *MvDrwToEL.TU4*, *NotCpl.TU4*,
PalletArvDrwCL.TU4, *PalletArvDrwEL.TU4*, *PalletLvGCL.TU4*, *PalletLvGEL_1.TU4*,
PalletLvTUAtCL.TU4, *PalletLvTUAtEL_1.TU4*, *PStopClosesCL.TU4*, *PStopClosesEL_1.TU4*,
PStopOpensCL.TU4, *PStopOpensEL_1.TU4*, *QCorrType.TU4*, *QCplT1A_1B.TU4*,
QCpTl2A_2B.TU4, *QDrwLoc.TU4*, *ReadLabel.TU4*, *WrongType.TU4*}

Σ_{R_w} = {*DoRpr.k*, *ProcPallet.k*}

Σ_{A_w} = { *ASDwn.k*, *ProcCpl.k*, *ProcErr.k*, *RobUp.k*}

Σ_{R_3} = { *ProcPallet.AS3*}

Σ_{A_3} = { *ProcCpl.AS3*, *ProcErr.AS3*, *PalletRepd.AS3*}

Σ_{R_v} = { *TrnsfToEL.r*, *TrnsfELToCL.r*, *LibPallet.r*}

Σ_{A_v} = { *NoTrnsfEL.r*, *TrnsfCplToEL.r*, *TrnsfCplToCL.r*, *PalletRlsd.r*}

Σ_{R_6} = { *TrnsfToEL3_Up*, *TrnsfToEL3_1D*, *TrnsfToEL3_2D*, *TrnsfToEL3_BD*,
TrnsfELToCL.TU3, *LibPallet.TU3*, }

Σ_{A_6} = { *NoTrnsfEL.TU3*, *TrnsfCplToEL.TU3*, *TrnsfCplToCL.TU3*, *PalletRlsd.TU3*}

Σ_{R_7} = { *TrnsfToEL.TU4*, *TrnsfELToCL.TU4*, *LibPallet.TU4*}

Σ_{A_7} = { *NoTrnsfEL.TU4*, *TrnsfCplToEL.TU4*, *TrnsfCplToCL.TU4*, *PalletRlsd.TU4*}

In the DES diagrams that follow, *uncontrollable events* are shown in italics; all other events are *controllable.* Initial states can be recognized by a thick outline, and marked states are filled. Finally, all supervisors were designed by hand, using modular techniques.

The models and supervisors developed for this example are based on the automata presented in [7] and [12]. We have altered them to fit our setting, and we have extended them to fill in the missing details of several events that were defined as "macro events" in the cited references.

Adapting the AIP design from [7] and [12] was quite straightforward. As the three assembly stations were very similar, it was natural that they were modelled as a group by Charbonnier et al. Similarly, the four transport units were modelled as a group. That we had seven well defined units in the original design made it easy for us to make these our seven *low levels.*

The next step was to create interfaces for each *low level.* To do this, we examined the behavior of the assembly stations and transport units and extracted this as a set of commands and responses. For example, we noted that assembly station 1 performed two tasks; it processed a pallet, or it initiated a repair of itself. When processing a pallet, it either completed successfully, encountered a processing error, or its robot broke down. When it initiated a repair, it eventually was fixed and reported that it was operational. We captured this behavior in Figure 13.2.

We now need to separate the internal behavior of each level (behavior specifying how the *low level* operated), from its interaction behavior (behavior specifying how the rest of the system interacts with it, as well as specifying when it should perform its services). For assembly station 1, this meant moving details about keeping track of the breakdown status of the station (plant **ASStoreUpState.AS1** in Figure 12.2) as well as when to start processing a pallet (supervisor **OFProtEL1** in Figure 12.9) to the *high level.* For transport unit 1, this meant moving the supervisors that enforced the maximum capacity of external loops 1 and 2 (Figures 12.9 and 12.10) and the supervisors that decided when a pallet should be transported to/from external loop 1 (Figure 12.5) to the *high level.*

The last step is to ensure a clear line of causality between events in the *high level, interfaces,* and *low levels.* As an example, let's examine assembly station 3. If the reader examines Figure 11.2, they will see a pallet must reach pallet sensor PS 3.4 before it can leave pallet gate 3.2. In the modelling used in [7] and [12], this was captured in a single DES. However, the pallet arriving at the gate is needed at the *high level* to decide when to tell the assembly station to start processing a pallet. The pallet leaving the gate is needed internally to express the relationship that a pallet can't arrive at pallet stop SP 3.2 until it has left the gate (plant model **DepGateNExtraSen.AS3** in Figure 13.5). This can be easily handled by making *request event Proc.Pallet.AS3* dependent on the pallet arriving at the gate (Figure 12.3) and the pallet leaving the gate dependent on event *Proc.Pallet.AS3* (Figure 14.4). This preserves the dependency between the pallet arriving at the gate before it can leave the gate, while at the same time clearly delineates the relationship to the *interface* of assembly station 3.

Chapter 12

The AIP High Level

The *high level* contains the 15 DES shown in Figure 12.1, which shows the definition of the *high level's* subsystem G_H, plant component $\mathcal{G}_H$, and supervisor component $\mathcal{S}_H$. They are defined to be the synchronous product of the indicated automata.

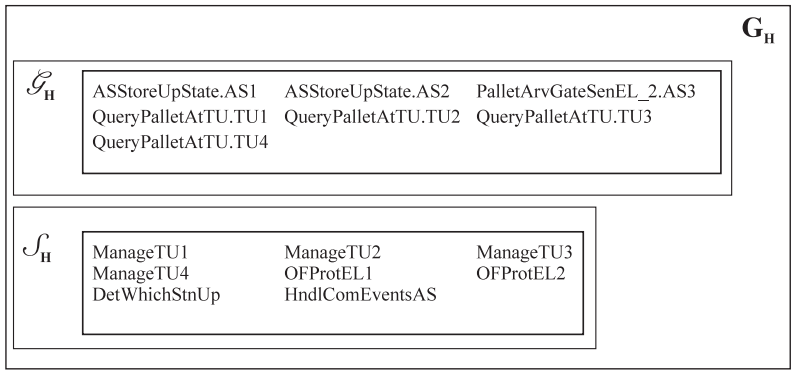


Figure 12.1: High Level

The *high level* keeps track of the breakdown status of assembly stations 1 and 2, as well as enforces the maximum capacity of external loops 1 and 2. It controls the operation of all transport units and all assembly stations, as well as tracking the pallets' progress around the manufacturing system.

12.1 Plant Component

We now discuss the plant models for the *high level*. The first set of models are DES **ASStoreUpState.k**, where $k = \text{AS1}, \text{AS2}$. They are shown in Figure 12.2. These DES track

the breakdown status of assembly stations 1 and 2. They also disable processing pallets and initiate repairs when their respective station is down. Our next DES is **PalletArvGateSenEL_2.AS3**, shown in Figure 12.3. This models the pallet sensor at pallet gate 3.2, the gate which controls access to assembly station 3. It also ensures that AS3 cannot start processing a pallet until a pallet arrives at the gate. The corresponding sensors for assembly stations 1 and 2 are not shown here because they're included as part of the nodes for the assembly stations. The reason for this difference is that EL 3 is not restricted to the number of pallets that it may contain, and the other loops are.

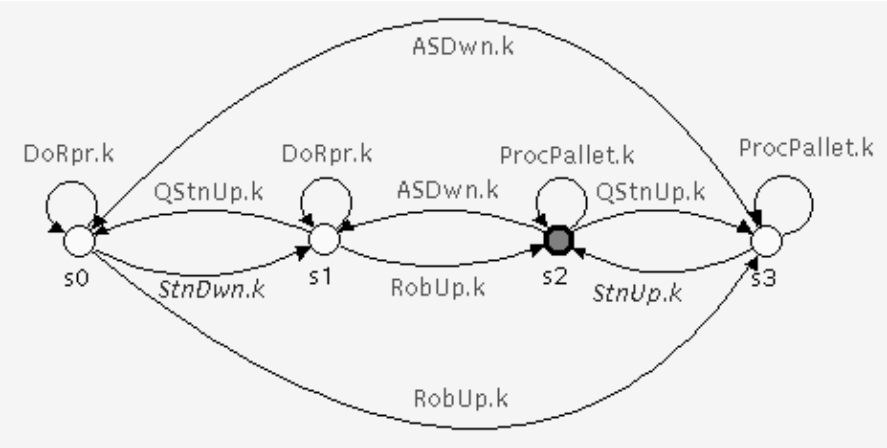


Figure 12.2: ASStoreUpState.k

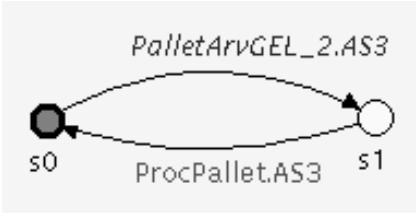


Figure 12.3: PalletArvGate-SenEL_2.AS3

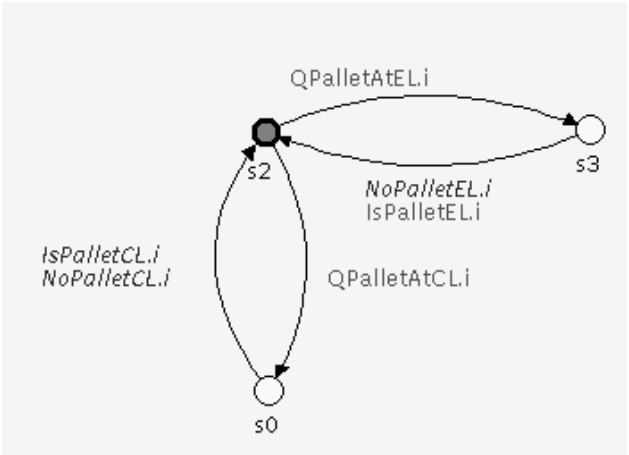


Figure 12.4: QueryPalletAtTU.i

The next set of DES are **QueryPalletAtTU.i,** where $i = \text{TU1}, \text{TU2}, \text{TU3}, \text{TU4}$. They

are shown in Figure 12.4. They provide a means to determine if a pallet is waiting to enter the indicated transport unit at either the associated external loop or the central loop.

12.2 Supervisor Component

We now discuss the supervisors for the *high level*. The first are **ManageTU1** and **ManageTU2** shown in Figures 12.5 and 12.6, respectively. They are identical up to event relabelling. They control the transfer of pallets between the central loop and the indicated external loops, and they permit pallets on the central loop to pass through a transport unit (to be liberated) without being transferred to its attached external loop. Pallets are liberated if the attached external loop is at maximum capacity (determined by supervisors **OFProtEL1** and **OFProtEL2**), the associated assembly station is down, or the transport unit determines that the pallet is not to be transferred (see the definition of the transport unit subsystems, *low levels 4-7*).

The corresponding supervisors for transport units 3 and 4 are DES **ManageTU3** and **ManageTU4**, shown in Figures 12.7 and 12.8. Supervisor **ManageTU3** is similar to **ManageTU1** except that it always tries to transfer a waiting pallet to external loop 3 (EL 3 has no capacity restriction). Also, DES **ManageTU3** first determines the breakdown status of assembly stations 1 and 2 before attempting to transfer a pallet to EL 3. It also passes this information to TU3. Finally, DES **ManageTU4** is the same as **ManageTU3** except that it does not determine the breakdown status of assembly stations 1 and 2.

The next set of supervisors are **OFProtEL1** and **OFProtEL2,** shown in Figures 12.9 and 12.10. These supervisors, identical up to relabelling, ensure that external loops 1 and 2 have at most one pallet in them at a given time. These supervisors also coordinate pallet transfers to the external loops with the operation of the appropriate assembly station.

The next supervisor the we discuss is **DetWhichStnUp,** shown in Figure 12.11. This supervisor is used by supervisor **ManageTU3** to determine the breakdown status of assembly stations 1 and 2. It also encodes this information to be passed on to TU3.

The last supervisor in this section is **HndlComEventsAS,** shown in Figure 12.12. The supervisor facilitates the lookup of the breakdown status of assembly stations 1 and 2 by multiple supervisors. Supervisor **HndlComEventsAS** allows **ManageTU1, ManageTU2,** and **DetWhichStnUp** to be designed independent of each other but not cause

each other to deadlock.

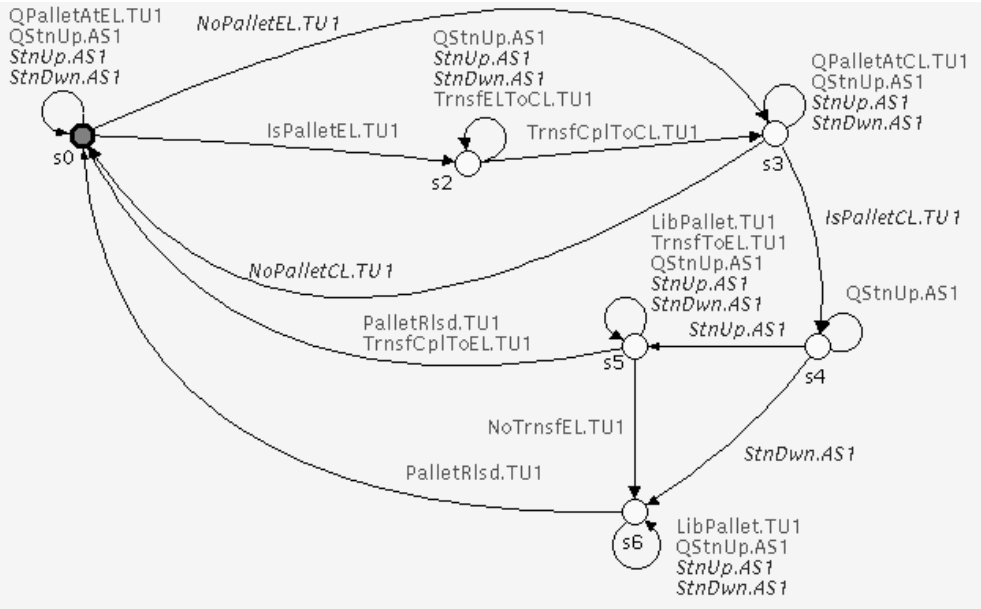


Figure 12.5: ManageTU1.

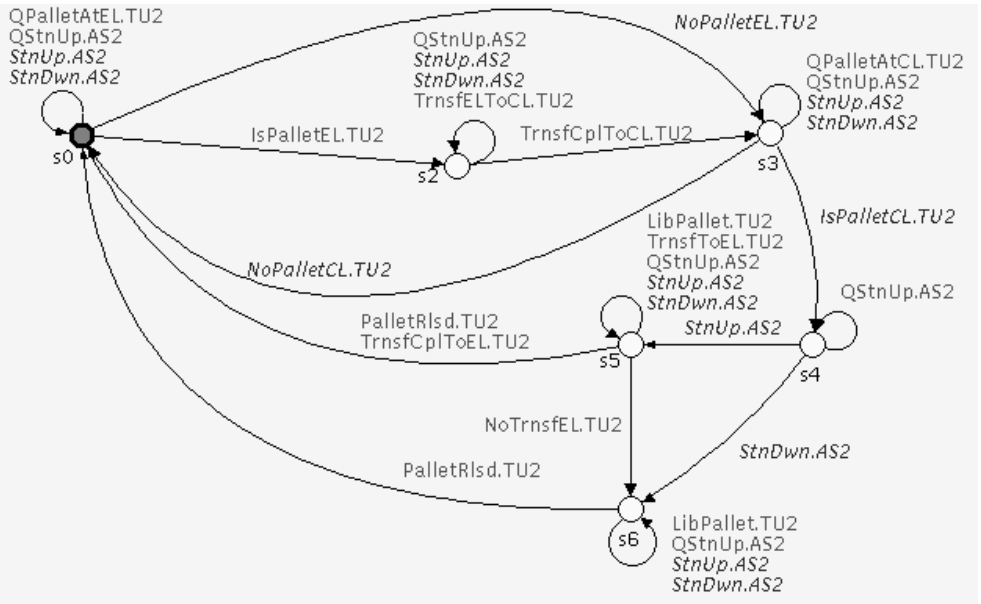


Figure 12.6: ManageTU2.

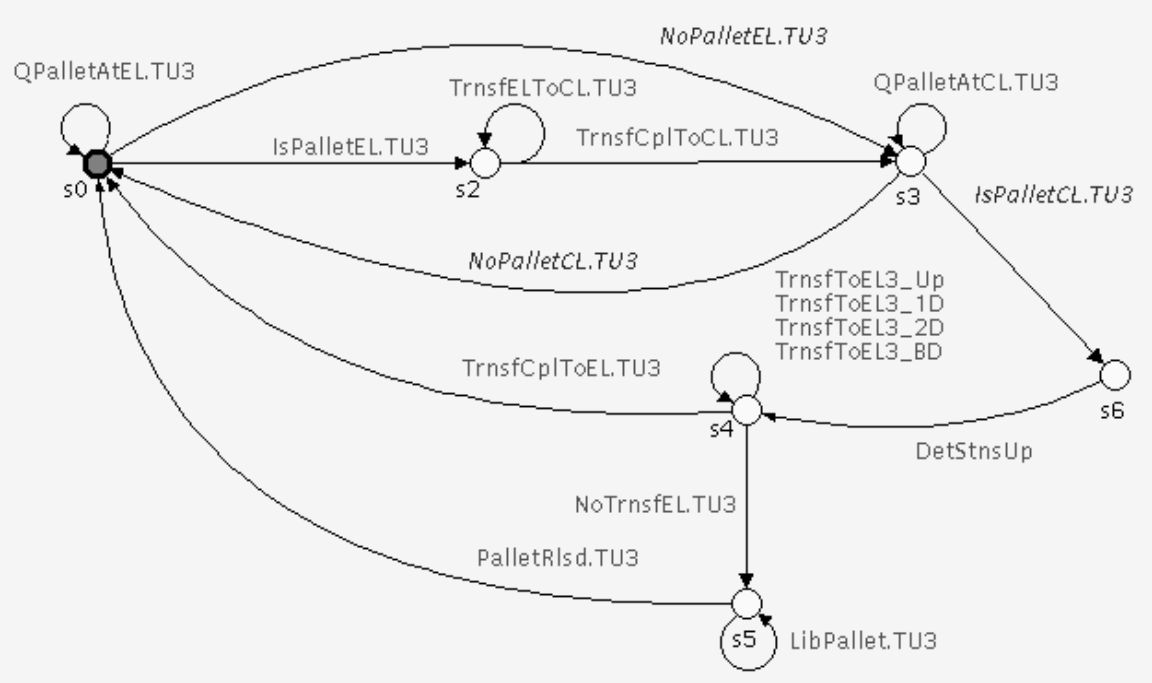


Figure 12.7: ManageTU3.

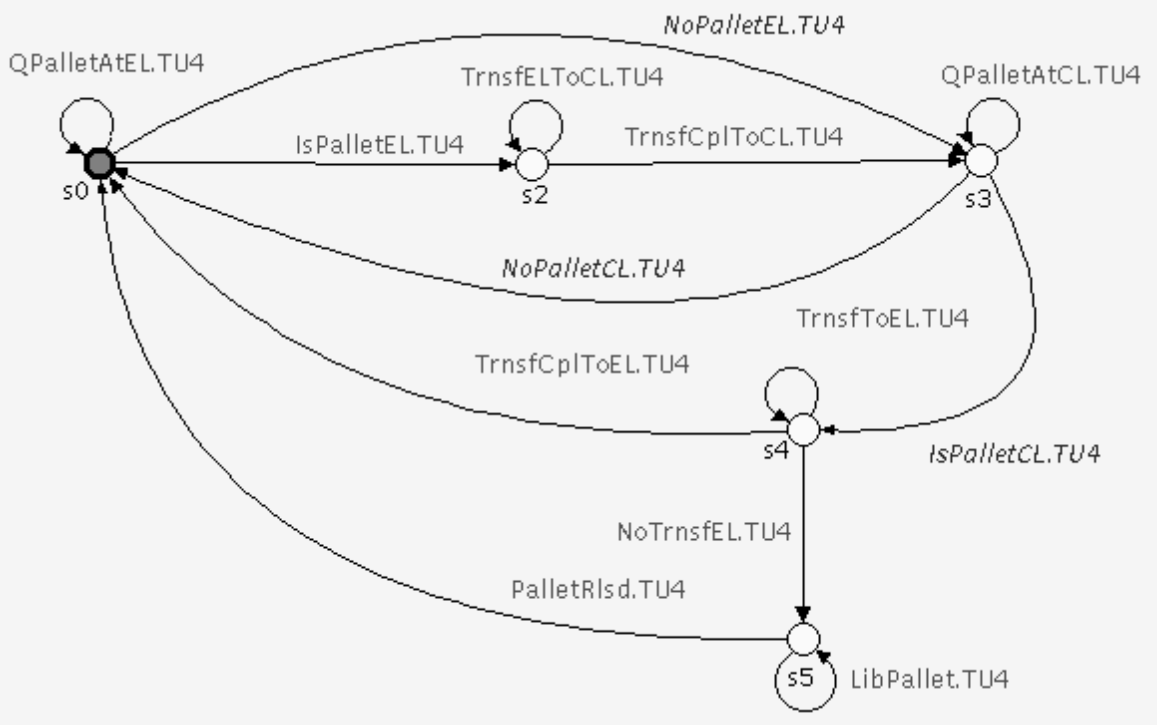


Figure 12.8: ManageTU4.

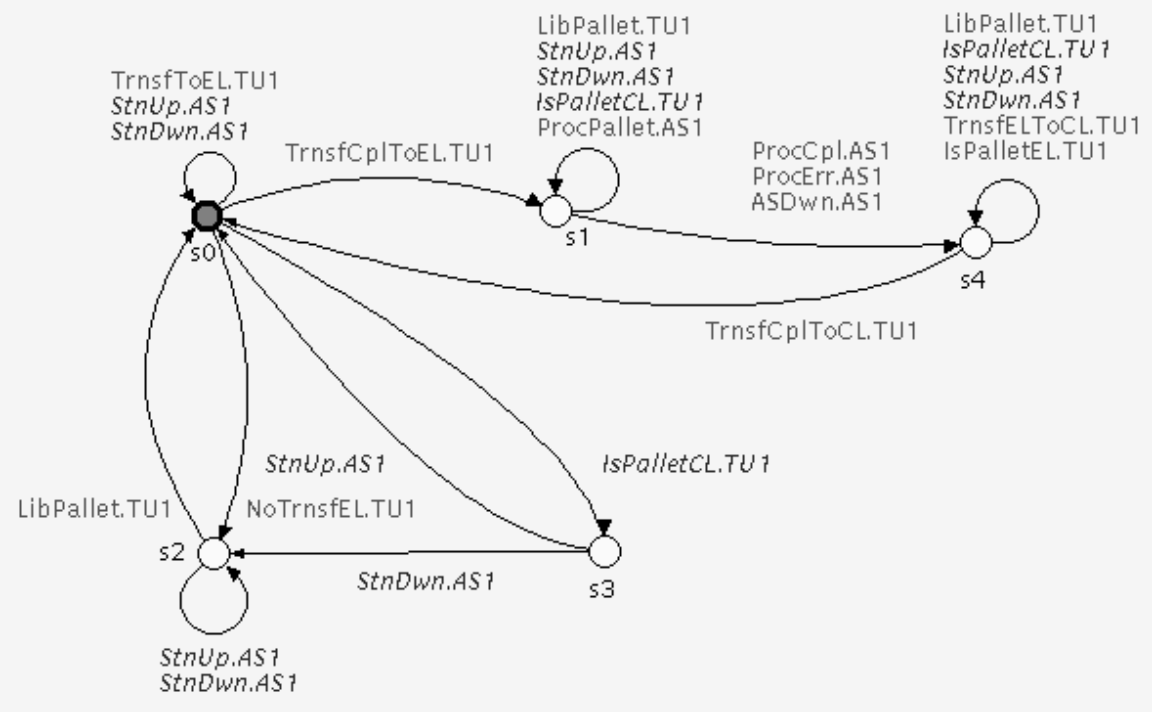


Figure 12.9: OFProtEL1.

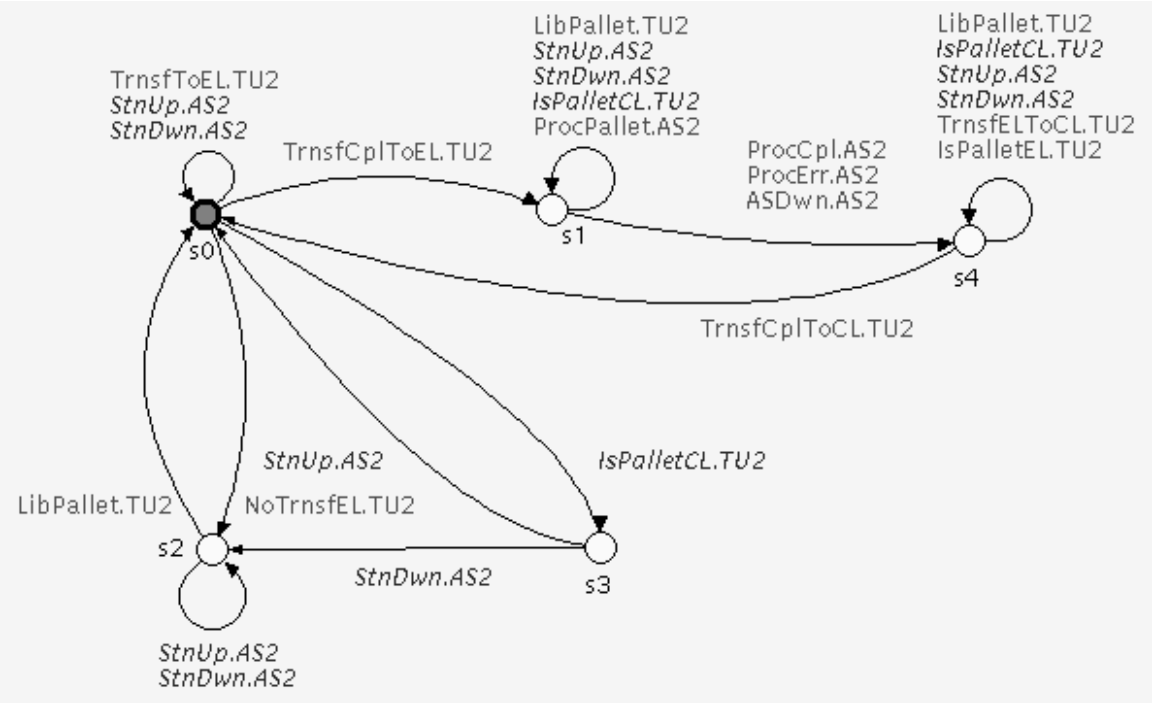


Figure 12.10: OFProtEL2.

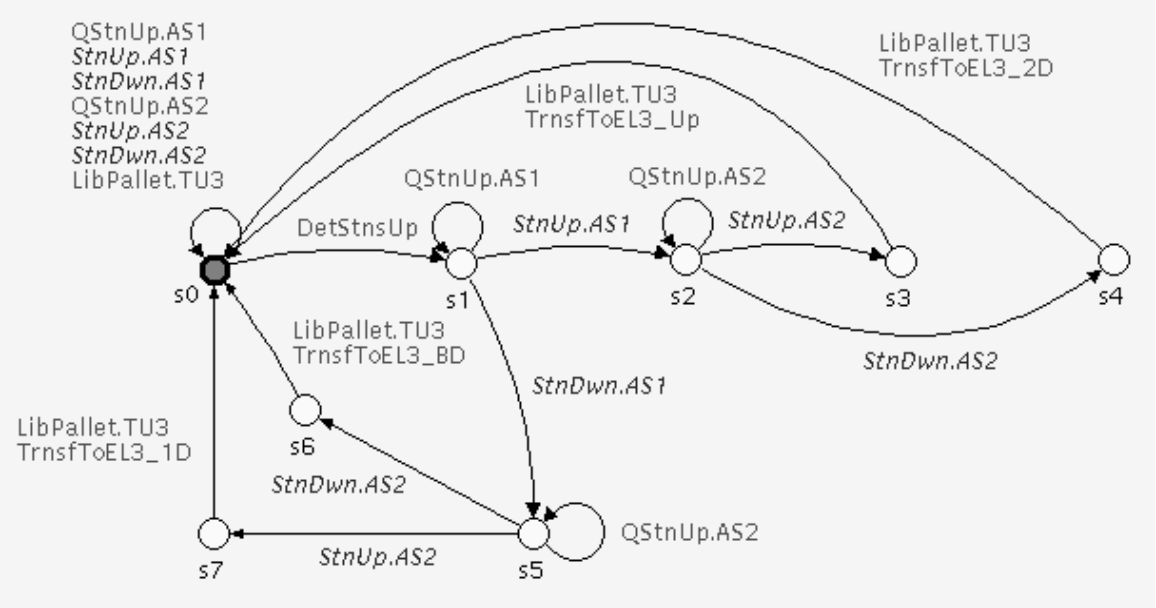


Figure 12.11: DetWhichStnUp.

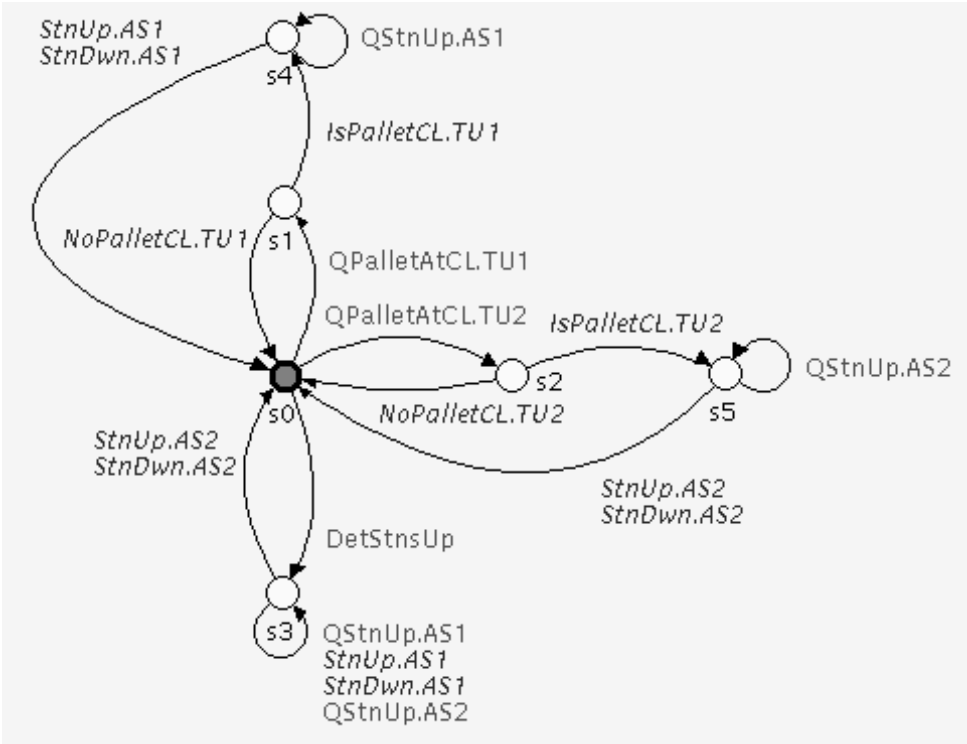


Figure 12.12: HndlComEventsAS.

Chapter 13

AIP Low Levels 1 and 2 (AS1 and AS2)

We now describe the *low levels* that represent assembly stations 1 and 2. As they are identical, we will describe them collectively as *low level subsystem* w, where $w = 1, 2$. We also define the companion index $k = \text{AS1}, \text{AS2}$, which takes its values relative to w (eg. $k = \text{AS1}$ when $w = 1$). As some DES presented in this chapter are identical to those of *Low level 3* (AS3), the index j is used in some diagrams. In this chapter, we will always set $j = k$.

Low level w contains the 17 DES shown in Figure 13.1, which shows the definition of *low level* w*'s* subsystem G_{L_w}, plant component $\mathcal{G}_{L_w}$, and supervisor component $\mathcal{S}_{L_w}$. They are defined to be the synchronous product of the indicated automata.

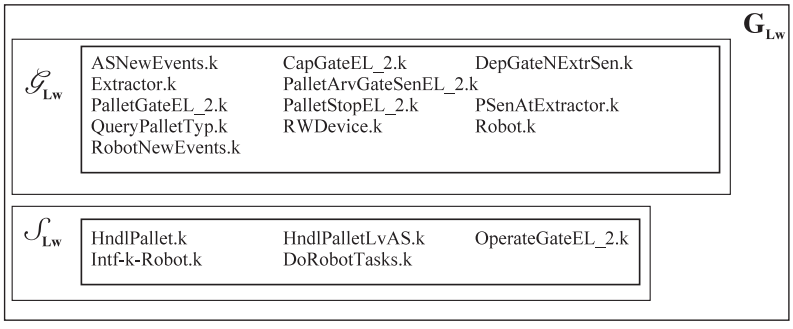


Figure 13.1: Low Level w

Low level w provides the functionality specified in its interface, shown in Figure 13.2. An assembly station accepts the pallet at its gate, and then presents it to its robot for assembly. It then releases the pallet, and reports on the success of the assembly operation. If the robot breaks down, this is reported through the interface, and the pallet is released. *Low level w* then waits for a repair command to return the robot to operation.

It's important to note that *request event ProcPallet.k* should be avoided while the robot is down. While the robot is down, which *answer event* that follows event *ProcPallet.k* is random and thus meaningless. This is a "fix" to satisfy the interface conditions when *star interfaces* are used. Ideally, the interface should have state information that makes event *ProcPallet.k* unavailable until the robot is repaired. This is possible with *command-pair interfaces*. Unfortunately, the *command-pair interface* definition was only just developed and there wasn't time to update our software and examples.

13.1 Plant Component

We now discuss the plant models for *low level w.* We start with plant models **ASNewEvents.k** and **CapGateEL_2.k,** shown in Figures 13.3 and 13.4. The first DES simply introduces new events that are used by the assembly station's interface.[1] Plant **CapGateEL_2.k** models how many pallets can fit into gate $w.2$ at once. More importantly, it creates a dependency between pallets arriving and pallets leaving the gate.

The next two plant models are **DepGateNExtrSen.k** and **Extractor.k,** shown in Figures 13.5 and 13.6. The first DES shows that a pallet must first leave gate $w.2$ before reaching the extractor, while the second one specifies the behaviour of the extractor which moves a pallet between the conveyor belt and the robot.

We now discuss plant models **PalletArvGateSenEL_2.k**, **PalletGateEL_2.k**, **PalletStopEL_2.k** and **PSenAtExtractor.k,** shown in Figures 13.7, 13.8, 13.9, and 13.11. The first DES shows that the pallet sensor at gate $w.2$ is gated by the event *ProcPallet.k*. The next two DES show the operation of pallet gate $w.2$ and pallet stop $w.2$. Finally, DES **PSenAtExtractor.k** shows the dependency between a pallet arriving at the extractor, and a pallet leaving the assembly station.

[1] Actually, event *RelPallet.k* coordinates between supervisors **HndlPalletLvAS.k** and **HndPallet.k**, the release of the pallet from the assembly station. It's the exception.

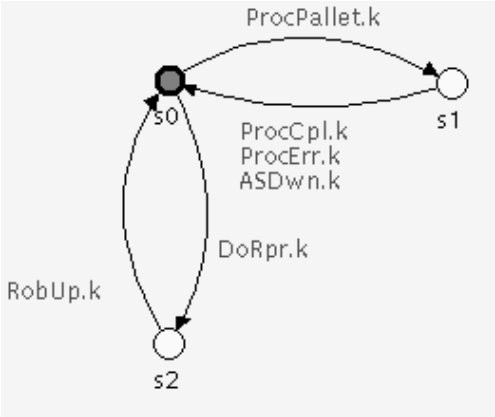


Figure 13.2: Interface to *Low Level w*.

Figure 13.3: ASNewEvents.k

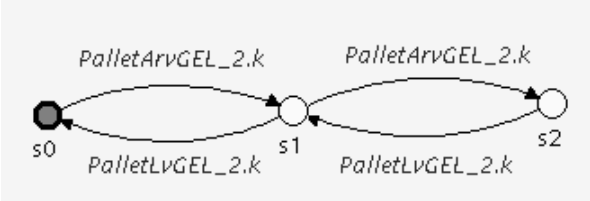


Figure 13.4: CapGateEL_2.k

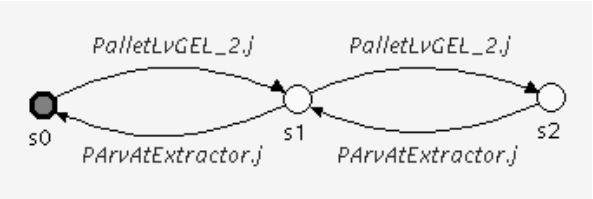


Figure 13.5: DepGateNExtrSen.j

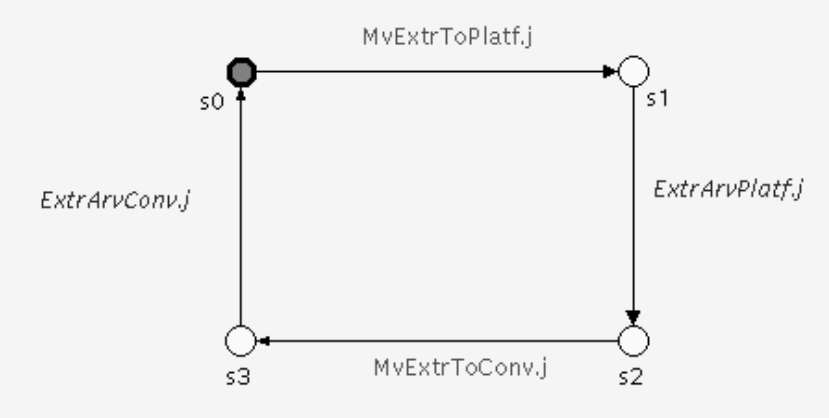


Figure 13.6: Extractor.j

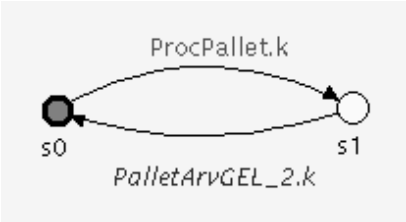


Figure 13.7: PalletArvGate-SenEL_2.k

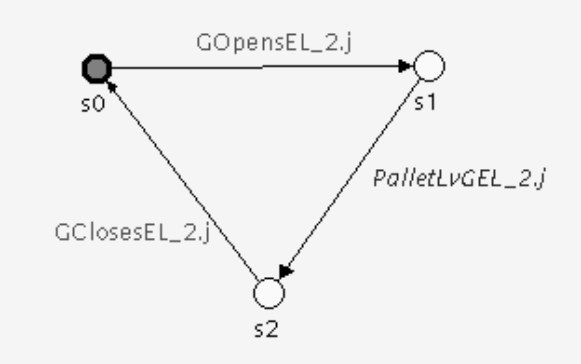


Figure 13.8: PalletGateEL_2.j

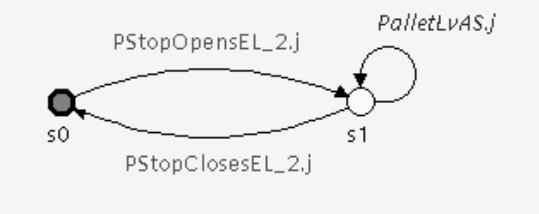


Figure 13.9: PalletStopEL_2.j

Figure 13.10: RobotNewEvents.k

Figure 13.11: PSenAtExtractor.j

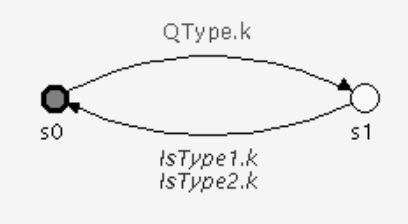


Figure 13.12: QueryPalletTyp.k

The next plant models are **QueryPalletTyp.k** and **RWDevice.k**, shown in Figures 13.12, 13.13, and 13.14. DES **RWDevice.k** models reading and writing to the pallet's electronic label. DES **QueryPalletTyp.k** provides a means of determining the pallet's type. **QueryPalletTyp.k** uses the data from the last read operation performed by the Assembly station's R/W device. If a read has not yet been performed, the results are undefined.

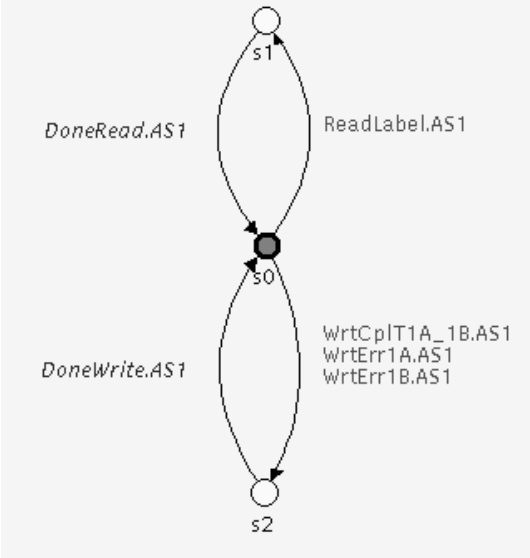


Figure 13.13: RWDevice.AS1

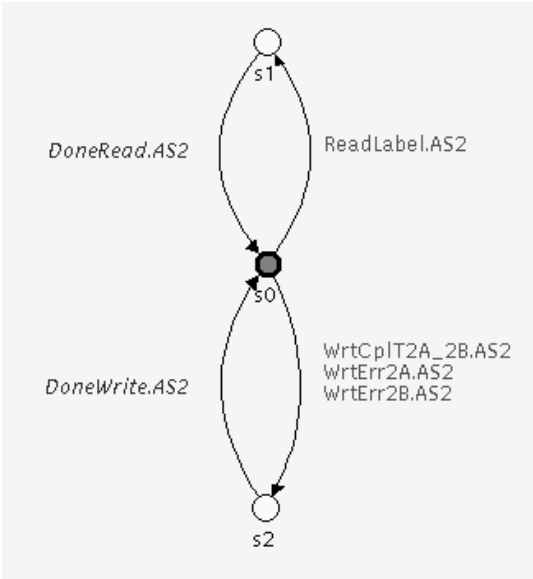


Figure 13.14: RWDevice.AS2

We now describe the plant models for the assembly stations' robots. The robots can perform two assembly tasks each, can successfully complete the assembly, generate an assembly error, or break down. This behaviour is modelled in **Robot.AS1** and **Robot.AS2,** shown in Figures 13.15, and 13.16. DES **RobotNewEvents.k,** shown in Figure 13.10, introduces new events used by the robots' supervisors.

13.2 Supervisor Component

We now discuss the supervisors for*low level w*. We start with supervisors **HndlPallet.AS1** and **HndlPallet.AS2,** shown in Figures 13.17 and 13.18. These supervisors handle the task of processing a pallet once it reaches the extractor for their respective assembly station. They read the pallet's label, present the pallet to the robot, and have the robot perform the appropriate tasks on the pallet. The supervisor then allows the pallet to leave the assembly

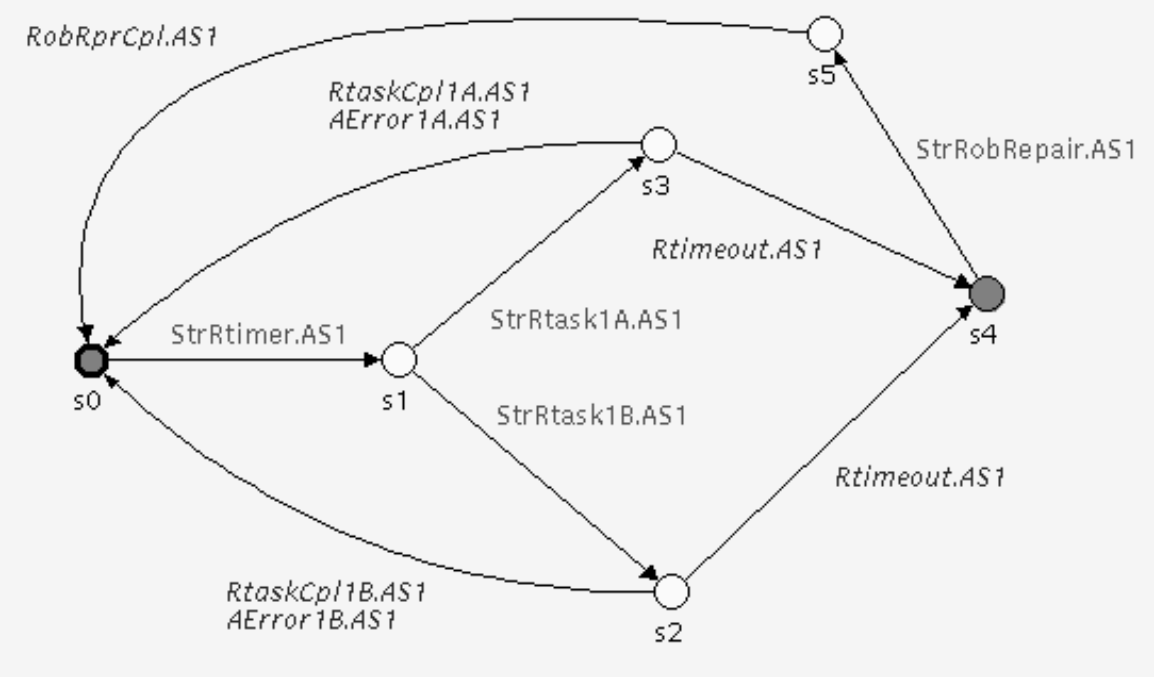


Figure 13.15: Robot.AS1

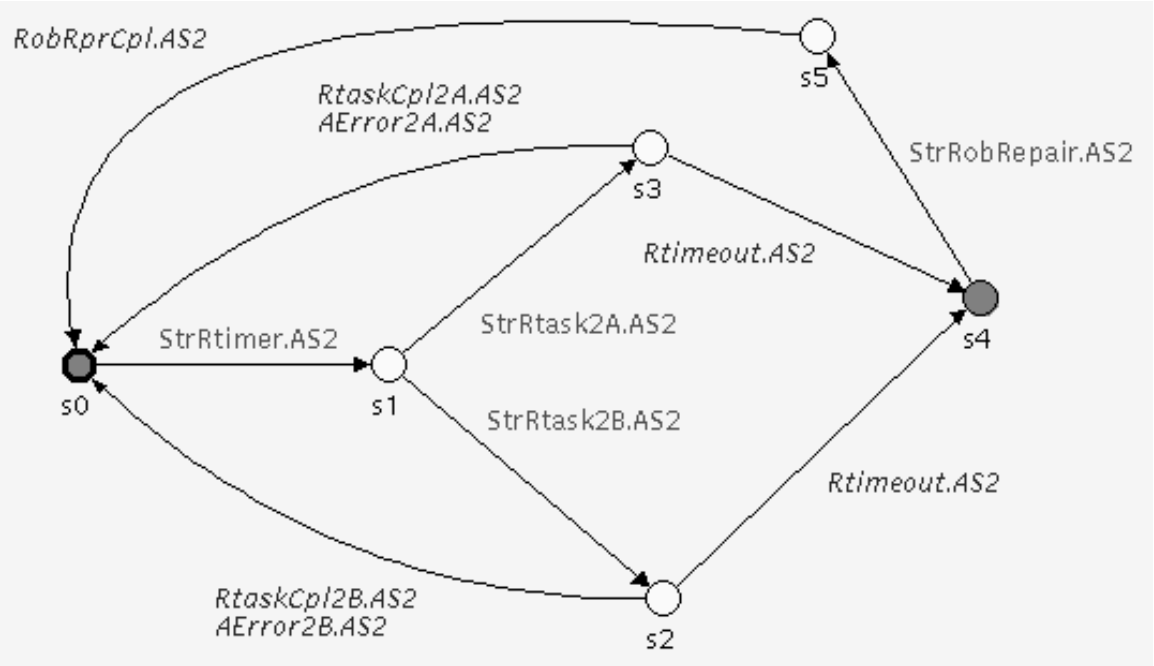


Figure 13.16: Robot.AS2

station and reports on the success of the processing operation by updating the pallet's label, and through the station's interface.

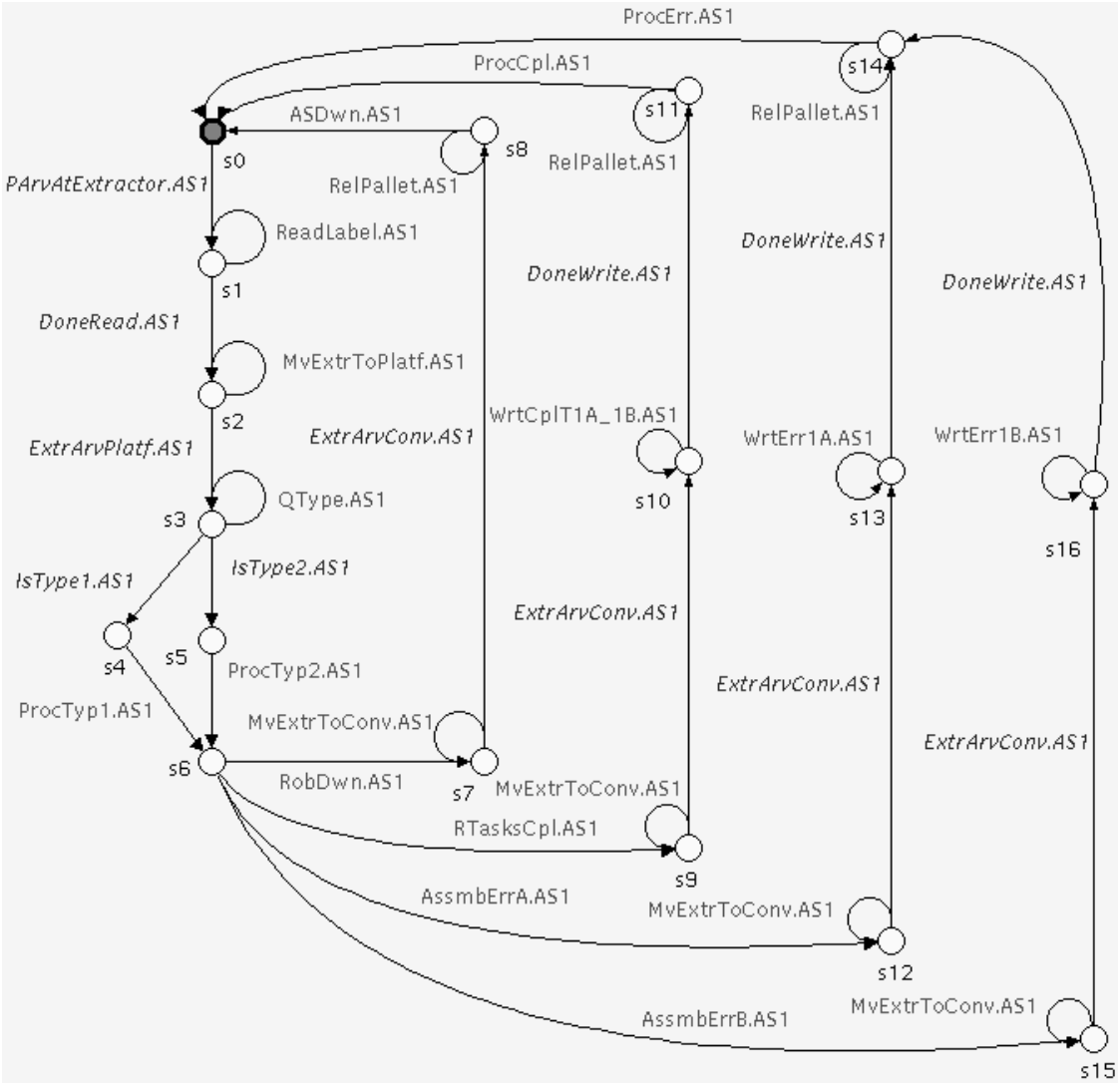


Figure 13.17: HndlPallet.AS1

We now discuss supervisor **HndlPalletLvAS.k,** shown in Figure 13.19. Upon receiving the signal (event *RelPallet.k*) from supervisor **HndlPallet.k**, **HndlPalletLvAS.k** opens the pallet stop and allows the pallet to leave the assembly station.

The next supervisor **OperateGateEL_2.k,** is shown in Figure 13.20. Once a pallet has arrived at gate $w.2$, the supervisor opens the gate and allows the pallet to enter. This supervisor, in conjunction with **HndlPalletLvAS.k,** guarantees that there is at most one

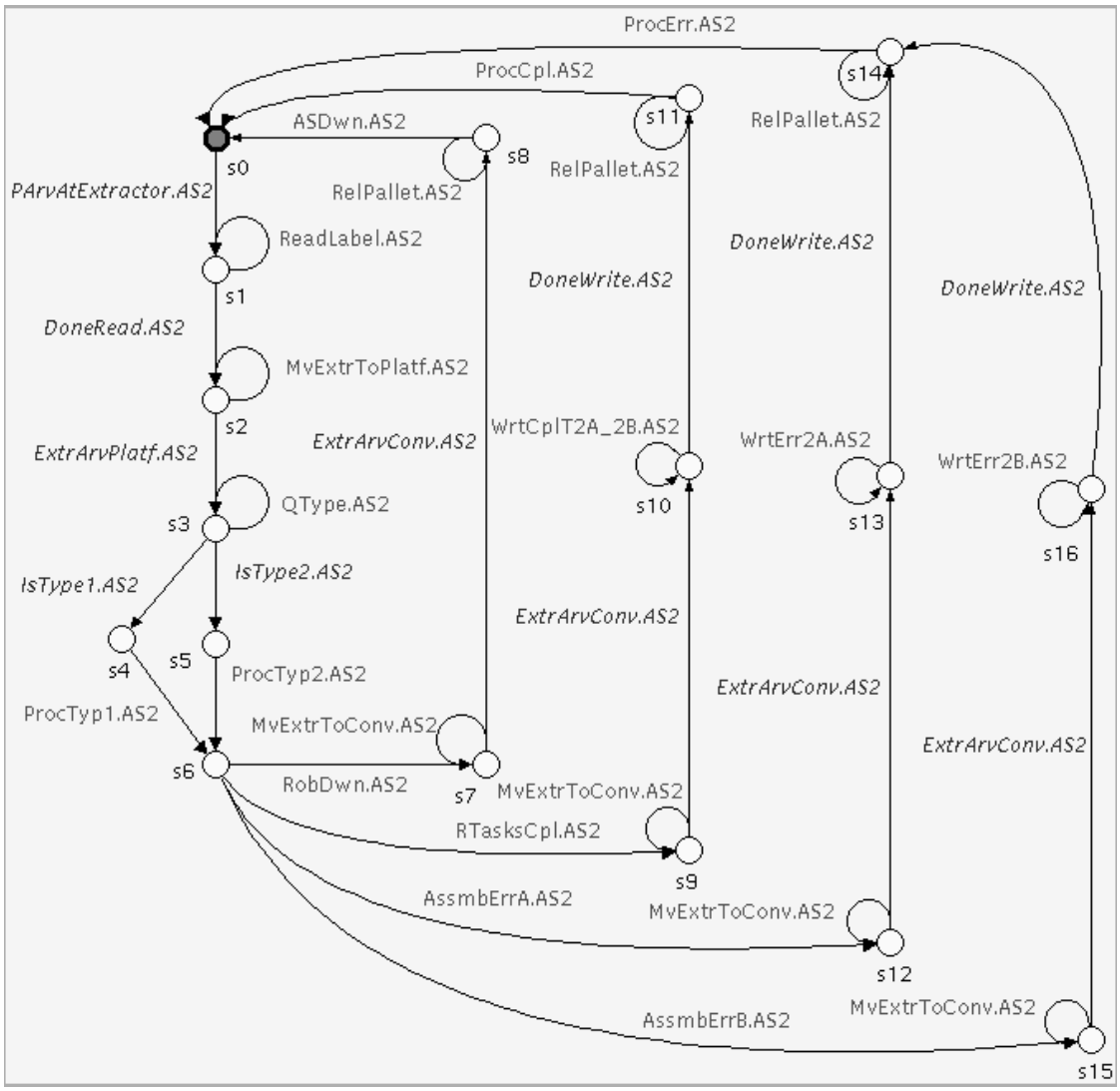


Figure 13.18: HndlPallet.AS2

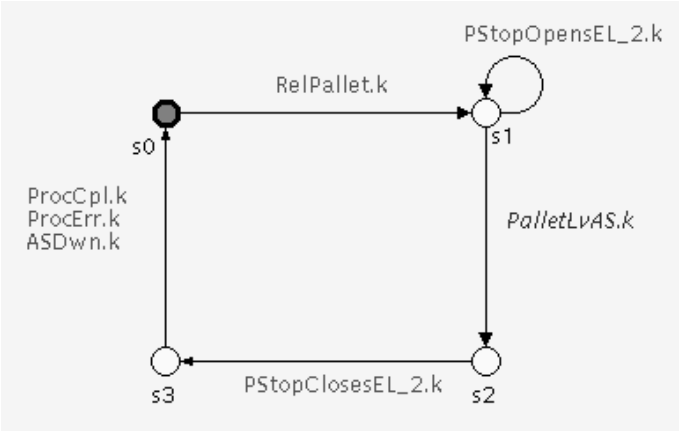


Figure 13.19: HndlPalletLvAS.k

pallet in the assembly station at a given time.

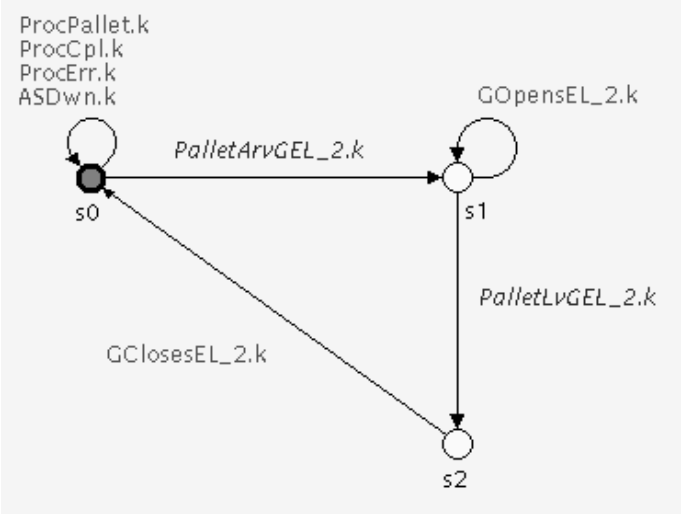


Figure 13.20: OperateGateEL_2.k

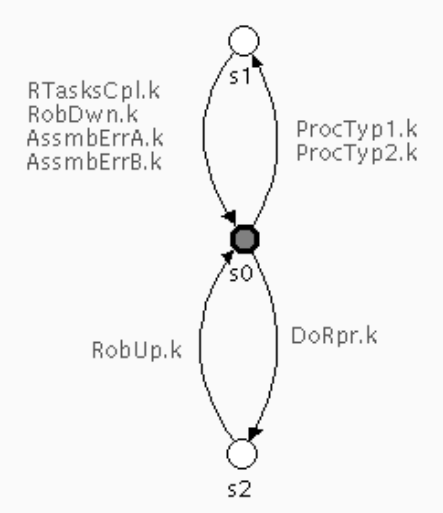


Figure 13.21: Intf-k-Robot.k

We now examine the supervisors for the assembly stations' robots. We start with supervisor **Intf-k-Robot.k** shown in Figure 13.21. It defines the tasks that the robot can perform.

Supervisors **DoRobotTasks.AS1** and **DoRobotTasks.AS2,** shown in Figures 13.22 and 13.23, control the operation of the robots. They make sure that the assembly tasks are performed in the correct order for a given type of pallet, they report on the success of the assembly operation, and they handle repairs when the robot breaks down.

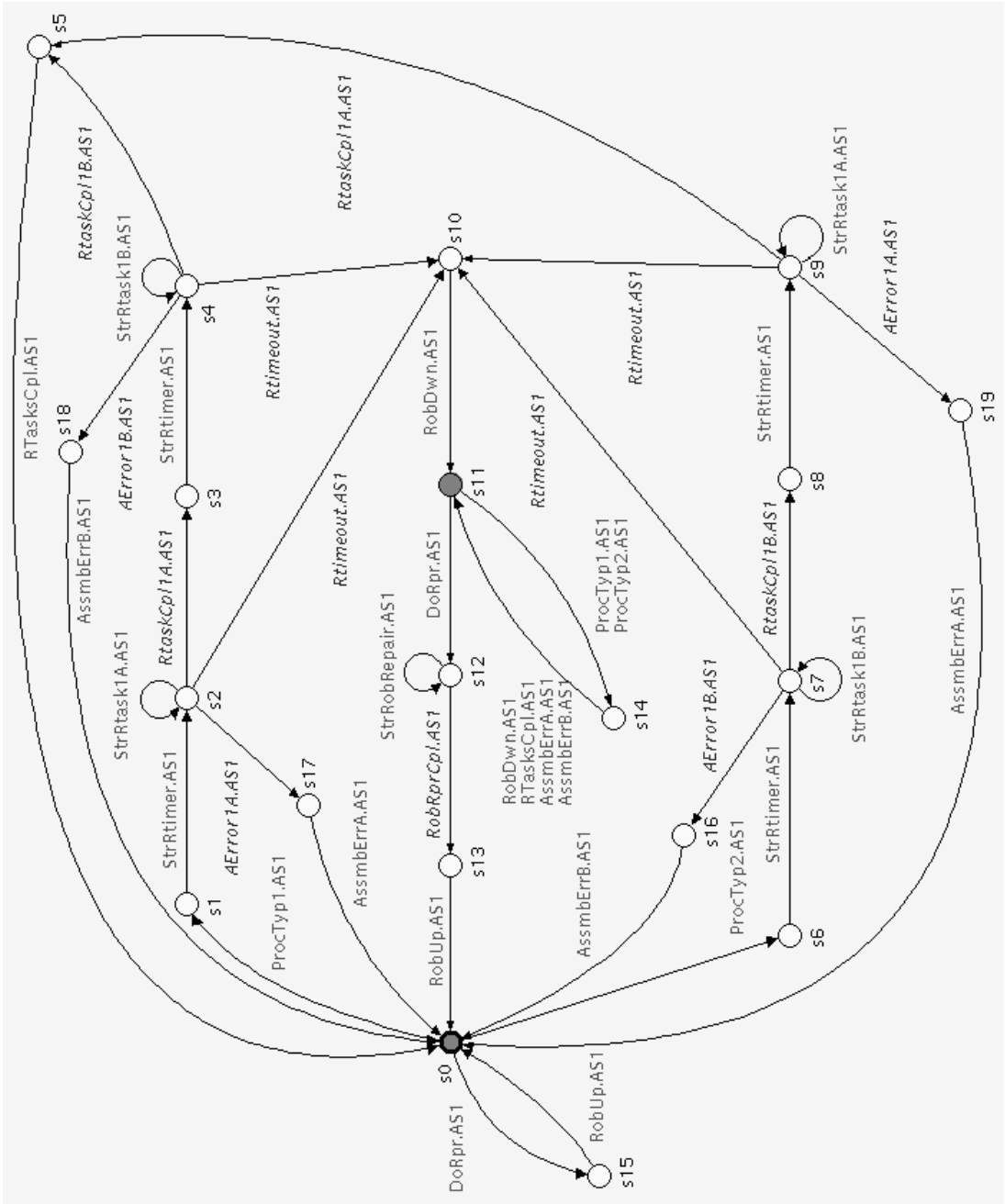


Figure 13.22: DoRobotTasks.AS1

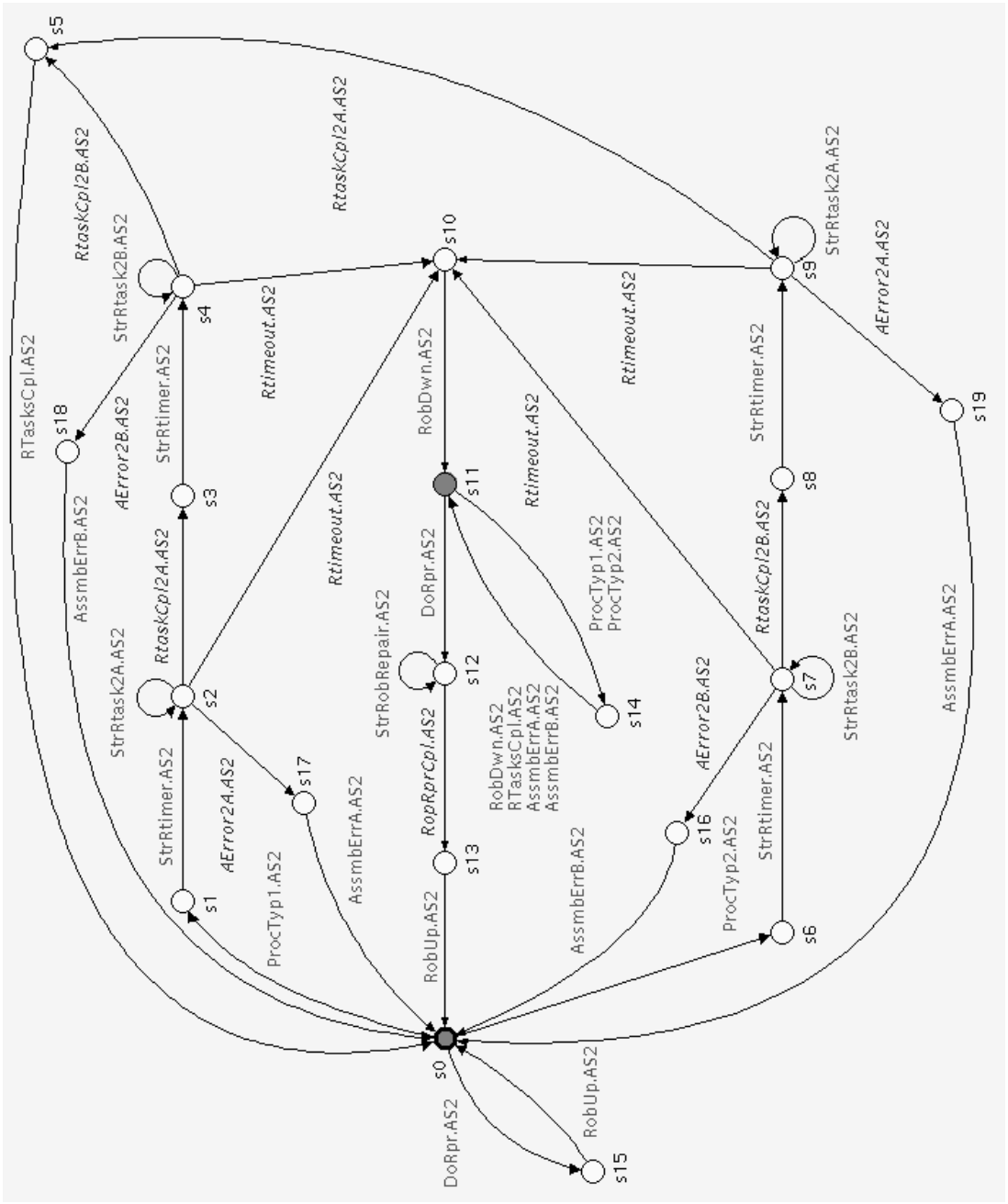


Figure 13.23: DoRobotTasks.AS2

Chapter 14

AIP Low Level 3 (AS3)

We now describe the *low level* that represents assembly station 3. *Low level* 3 contains the 27 DES shown in Figure 14.1, which shows the definition of *low level* 3's subsystem G_{L_3}, plant component $\mathcal{G}_{L_3}$, and supervisor component $\mathcal{S}_{L_3}$. They are defined to be the synchronous product of the indicated automata.

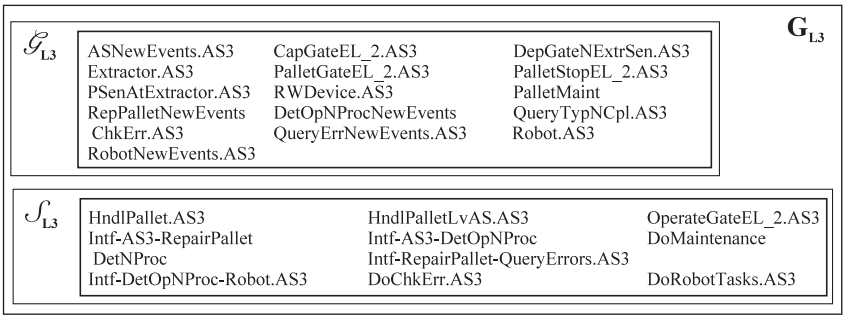


Figure 14.1: Low Level 3

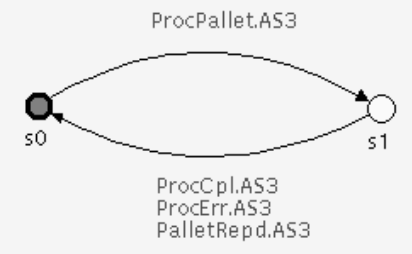


Figure 14.2: Interface to *Low Level* 3.

Low level 3 provides the functionality specified in its interface, shown in Figure 14.2. It describes the behaviour of assembly station 3, which is very similar to stations 1 and 2. The main differences are that station 3 can repair damaged pallets, is assumed not to

breakdown, and it can substitute for either AS1 or AS2 when they are down.

14.1 Plant Component

We now discuss the plant models for *low level* 3. As *low level* 3 is so similar to *low levels* 1 and 2, several of its plant models are identical up to relabelling, to those of *low levels* 1 and 2. In particular, DES **DepGateNExtrSen.AS3**, **Extractor.AS3**, **PalletGateEL_2.AS3**, **PalletStopEL_2.AS3** and **PSenAtExtractor.AS3** can be obtained by setting variable j = AS3 in Figures 13.5, 13.6 and 13.8 to 13.11.

The next plant models are **ASNewEvents.AS3, CapGateEL_2.AS3,** and **RWDevice.AS3,** shown in Figures 14.3, 14.4, and 14.5. The first DES simply introduces new events that are used by the assembly station's interface.[1] Plant **CapGateEL_2.AS3** models how many pallets can fit into gate 3.2 at once. More importantly, it creates a dependency between the event *ProcPallet.AS3* and pallets leaving the gate. As event *ProcPallet.AS3* is dependent on a pallet arriving at gate 3.2 (see Figure 12.3), this creates a dependency between pallets arriving and pallets leaving the gate. Finally, DES **RWDevice.AS3** models reading and writing to the pallet's electronic label.

We now describe the plant models that handle pallet repairs. These tasks consist of checking if a pallet is damaged, and if so, repairing it. It is assumed that the pallet has only sustained one assembly error, and thus only the first error found is repaired. We start with plant models **RepPalletNewEvents** and **PalletMaint,** shown in Figures 14.6 and 14.7. The first DES simply introduces new events that are used by related supervisors. The second DES models informing the operator of the type of error encountered, and waiting for the repair to be complete.

We now discuss the plant models for tasks related to querying if a pallet has been damaged. These tasks consist of determining if a pallet has been damaged, and if so, what type of error has been sustained. Only the first error encountered is reported. We start with plant models **QueryErrNewEvents.AS3** and **ChkErr.AS3,** shown in Figures 14.8 and 14.9. The first DES simply introduces new events that are used by related supervisors. The second DES provides a means to query about specific assembly errors.

[1] Actually, event *RelPallet.AS3* coordinates between supervisors **HndlPalletLvAS.AS3** and **HndPallet.AS3**, the release of the pallet from the assembly station. It's the exception.

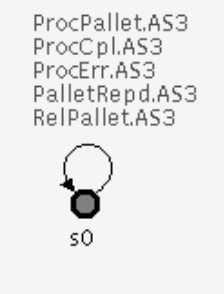


Figure 14.3: ASNewEvents.AS3

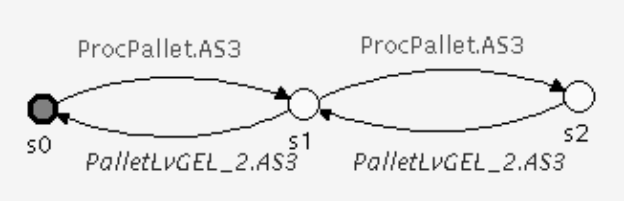


Figure 14.4: CapGateEL_2.AS3

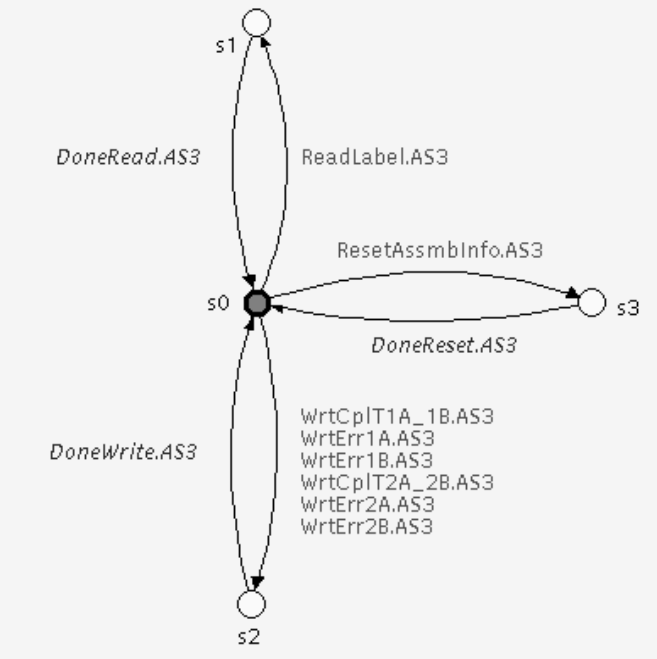


Figure 14.5: RWDevice.AS3

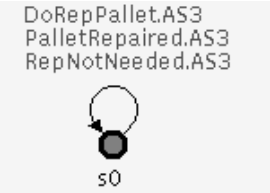


Figure 14.6: RepPalletNew-Events

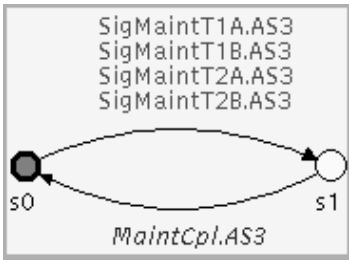


Figure 14.7: PalletMaint

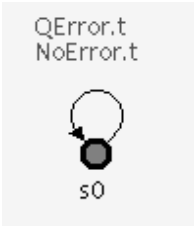


Figure 14.8: QueryErrNewEvents.t, with t = AS3

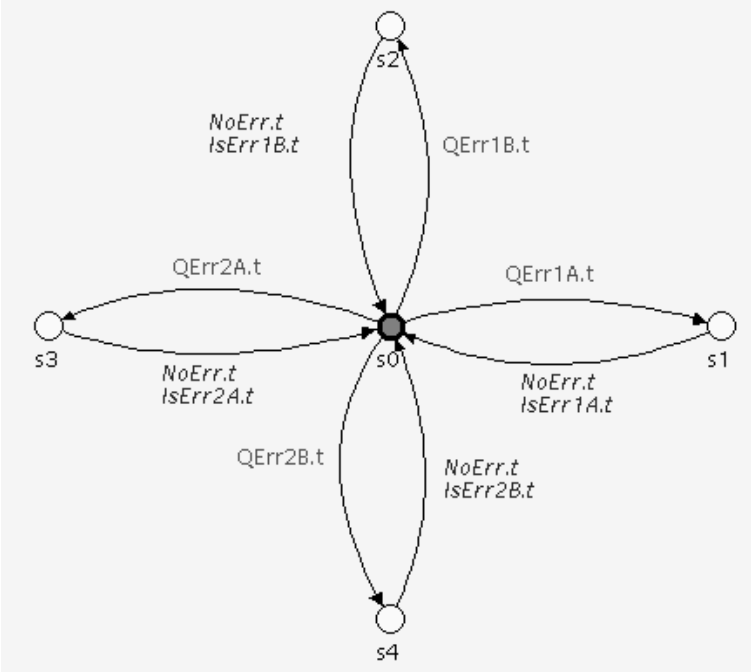


Figure 14.9: ChkErr.t, with t = AS3

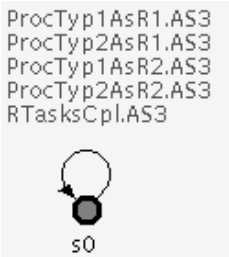


Figure 14.10: RobotNew-Events.AS3

We now describe the plant models for tasks related to determining which operations are needed to be performed on the pallet. Tasks consist of determining the assembly operations needed by the pallet, based upon the task completion data read from the pallet's label. The required assembly task is then performed. For simplicity and to prevent assembly station 3's resources from being monopolised, tasks for only one assembly station are performed, and then the pallet is released. For example, if the pallet is of type 1, and all assembly tasks are still pending, then task1A and task1B will be performed, and then the pallet would be released.

We start with plant models **DetOpNProcNewEvents** and **QueryTypNCpl.AS3,** shown in Figures 14.11 and 14.12. The first DES simply introduces new events that are used by related supervisors. The second DES provides a means of querying the type and assembly status of a pallet.

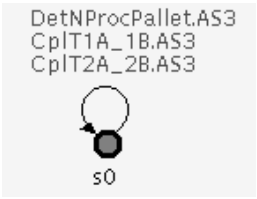


Figure 14.11: DetOpNProcNewEvents

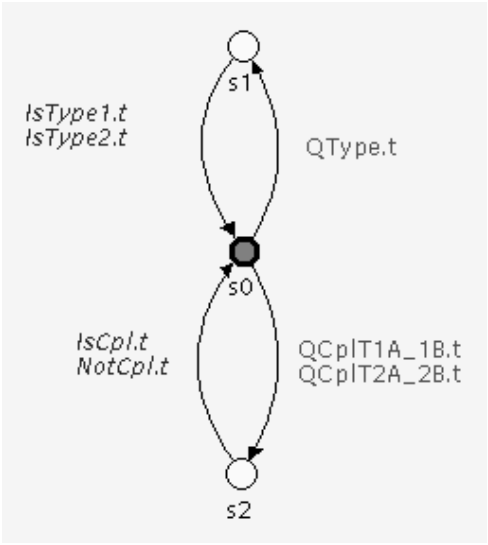


Figure 14.12: QueryTypNCpl.AS3, with t = AS3

We now discuss the plant components for the assembly station's robot. Assembly station 3's robot is similar to the robots in stations 1 and 2, with the difference that this robot is assumed not to breakdown, and can substitute for the other two robots when they are down. The plant models are shown in Figures 14.13 and 14.10.

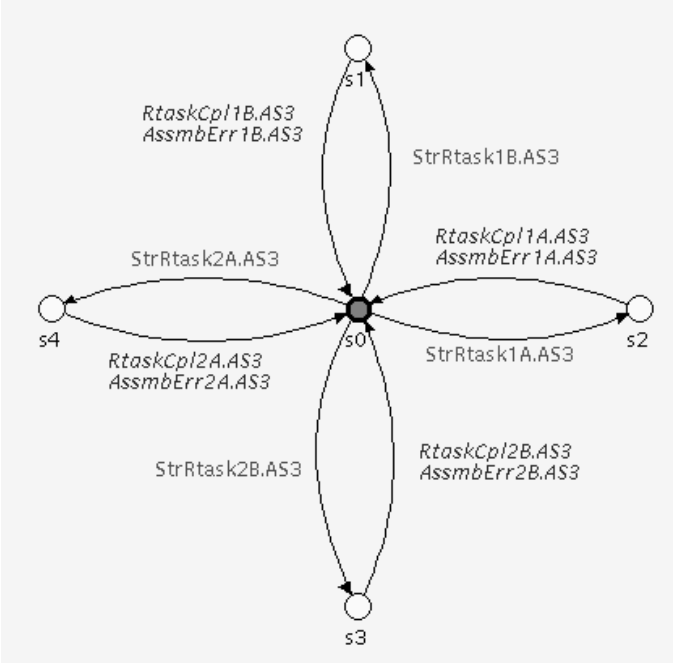


Figure 14.13: Robot.AS3

14.2 Supervisor Component

We now discuss the supervisors for *low level 3*. We start with supervisor **HndlPallet.AS3,** shown in Figure 14.16. DES **HndlPallet.AS3** handles the task of processing a pallet once it reaches the extractor. It first reads the pallet's label, and performs any needed repairs. If repairs were required, the pallet's label is updated, the pallet is released, and the results reported through the station's interface. If no repairs were required, the pallet is presented to the robot, and the appropriate tasks are performed on the pallet. The supervisor then allows the pallet to leave the assembly station and reports on the success of the processing operation by updating the pallet's label, and signalling through the station's interface.

We now discuss supervisor **HndlPalletLvAS.AS3,** shown in Figure 14.14. Upon receiving the signal (event *RelPallet.AS3*) from supervisor **HndlPallet.AS3**, **HndlPalletLvAS.AS3** opens the pallet stop and allows the pallet to leave the assembly station.

The next supervisor **OperateGateEL_2.AS3,** is shown in Figure 14.15. Once a pallet has arrived at gate 3.2, the supervisor opens the gate and allows the pallet to enter. This supervisor, in conjunction with **HndlPalletLvAS.AS3,** guarantees that there is at most one pallet in the assembly station at a given time.

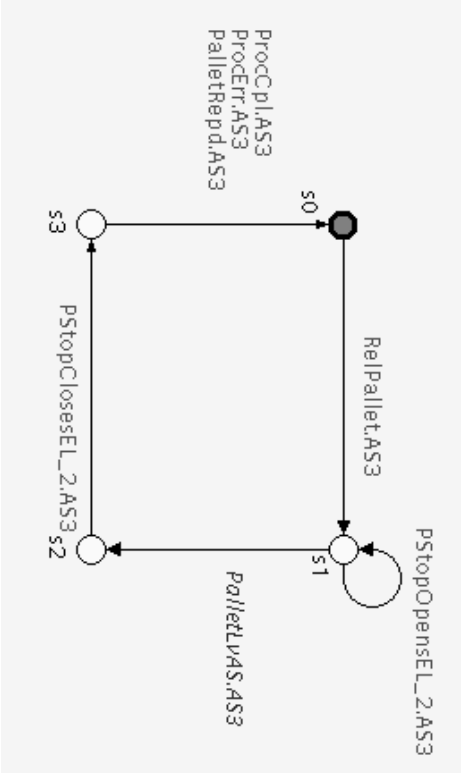


Figure 14.14: HndlPLvAS.AS3

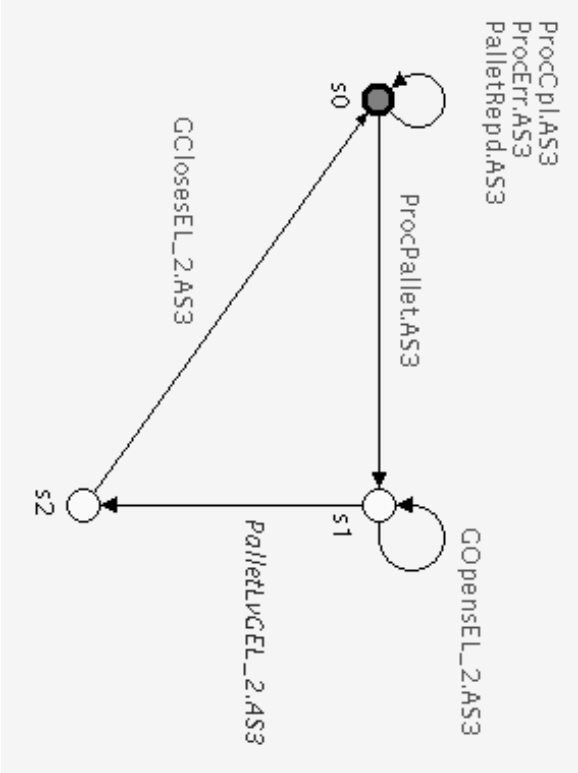


Figure 14.15: OperateGateEL_2.AS3

We now describe the supervisors that handle pallet repairs. We start with supervisor **Intf-AS3-RepairPallet**, shown in Figure 14.17. It defines the tasks required for repairing pallets.

We now discuss supervisor **DoMaintence,** shown in Figure 14.19. This supervisor checks if the pallet is damaged and needs repair. If so, it informs the operator of the type of error, waits for the pallet to be repaired, and then reports the results. If no repairs are required, then that is reported.

We now discuss the supervisors for tasks related to querying if a pallet has been damaged. We start with supervisor **Intf-RepairPallet-QuerryErrors.AS3**, shown in Figure 14.18. It defines the tasks required for querying if a pallet has errors.

We now discuss supervisor **DoChkErr.AS3,** shown in Figure 14.20. The supervisor sequentially checks for all four possible assembly errors, and reports on the first one it finds. If no errors are present, then that is reported.

We now describe the supervisors for tasks related to determining which operations are needed to be performed on the pallet. We start with supervisor **Intf-AS3-DetOpNProc**, shown in Figure 14.21. It defines the tasks required for determining which operations are

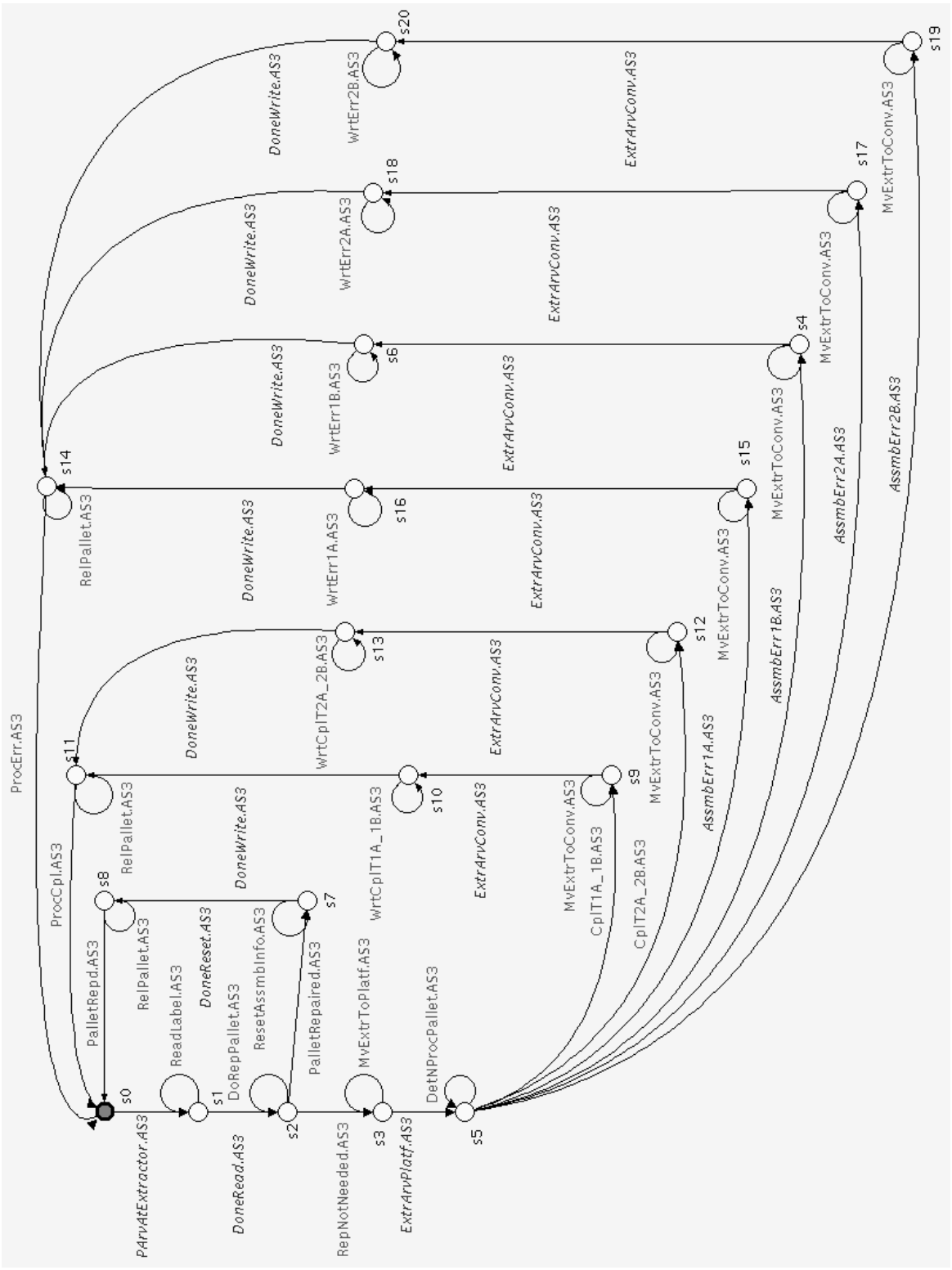


Figure 14.16: HndlPallet.AS3

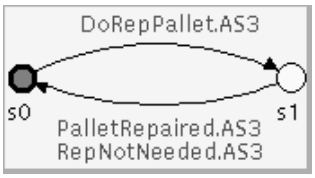


Figure 14.17: Intf-AS3-RepairPallet

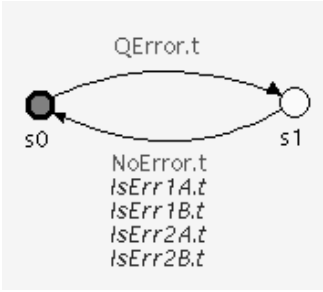


Figure 14.18: Intf-RepairPallet-QuerryErrors.AS3, with t = AS3

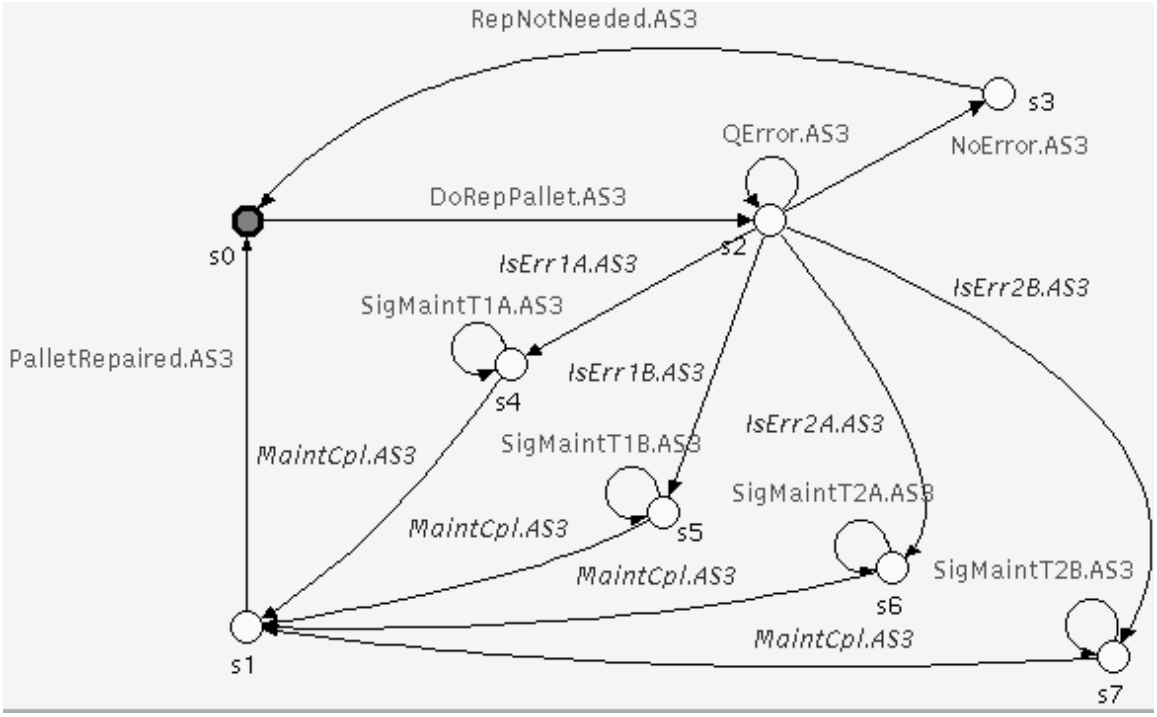


Figure 14.19: DoMaintence

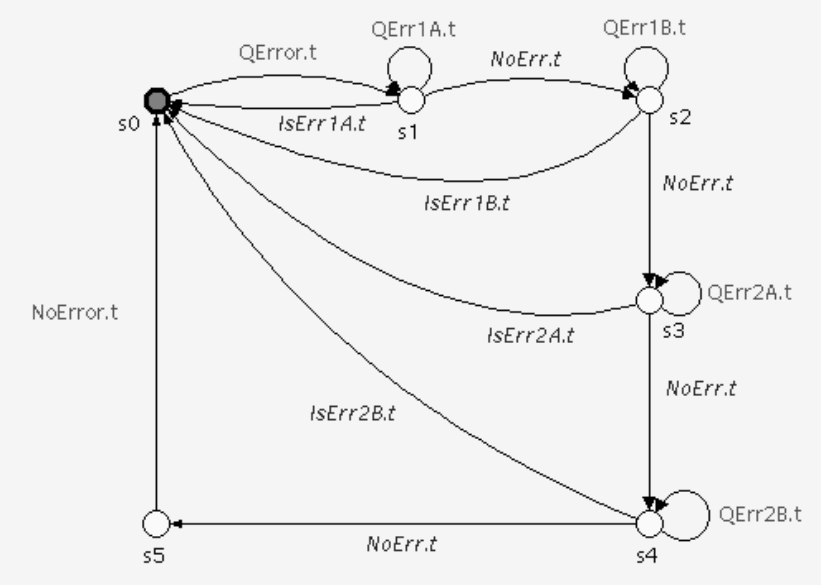


Figure 14.20: DoChkErr.t, with t = AS3

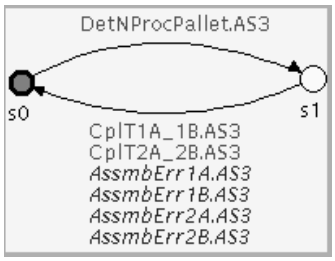


Figure 14.21: Intf-AS3-DetOpNProc

needed to be performed on the pallet.

We next discuss supervisor **DetNProc,** shown in Figure 14.22. This supervisor determines the next set of tasks pending for the pallet, performs them and then reports on the results.

We now discuss the supervisors for the assembly station's robot. We start with supervisor **Intf-DetOpNProc-Robot.AS3**, shown in Figure 14.23. It defines the tasks that the robot can perform.

The supervisor for the robot is given in Figure 14.24, and labelled **DoRobotTasks.AS3**. It's essentially a combined version of supervisors **Robot.AS1** and **Robot.AS2**, minus breakdowns.

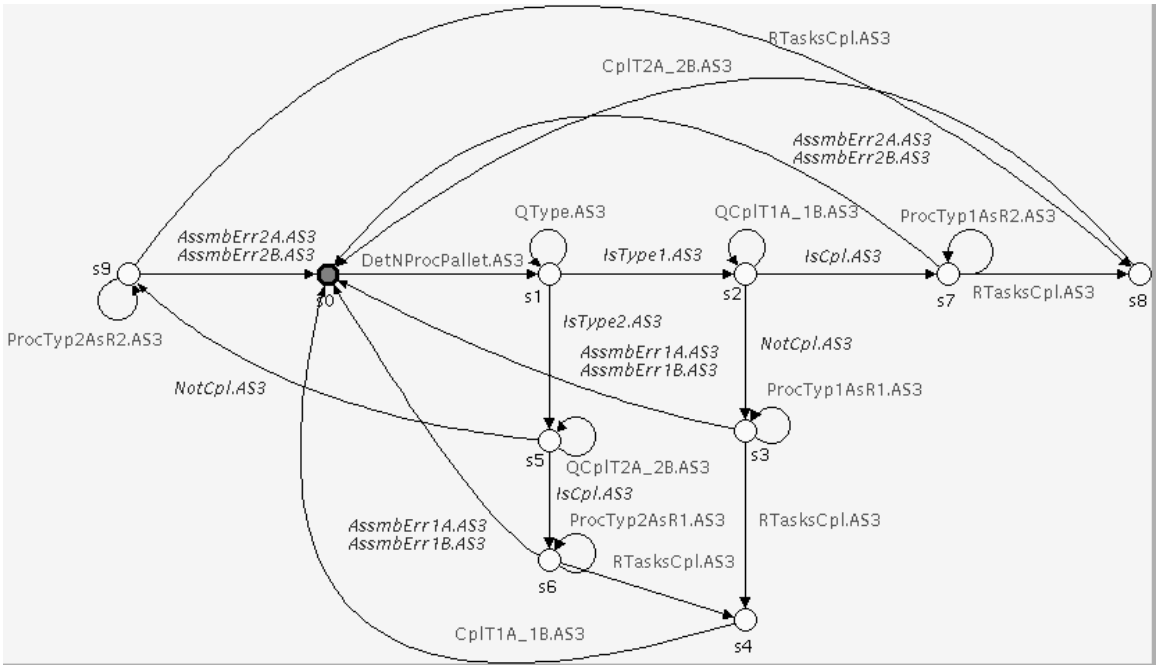


Figure 14.22: DetNProc

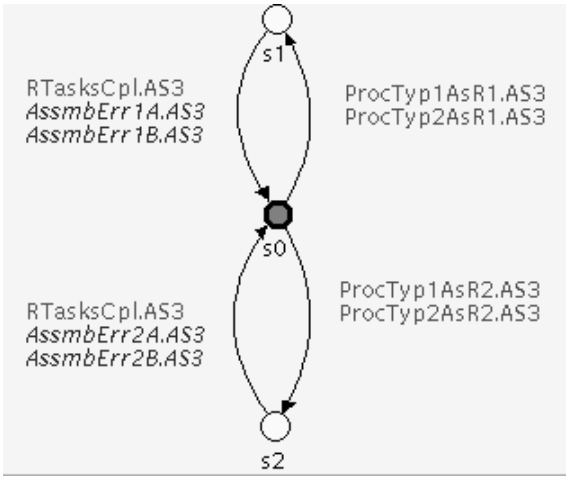


Figure 14.23: Intf-DetOpNProc-Robot.AS3

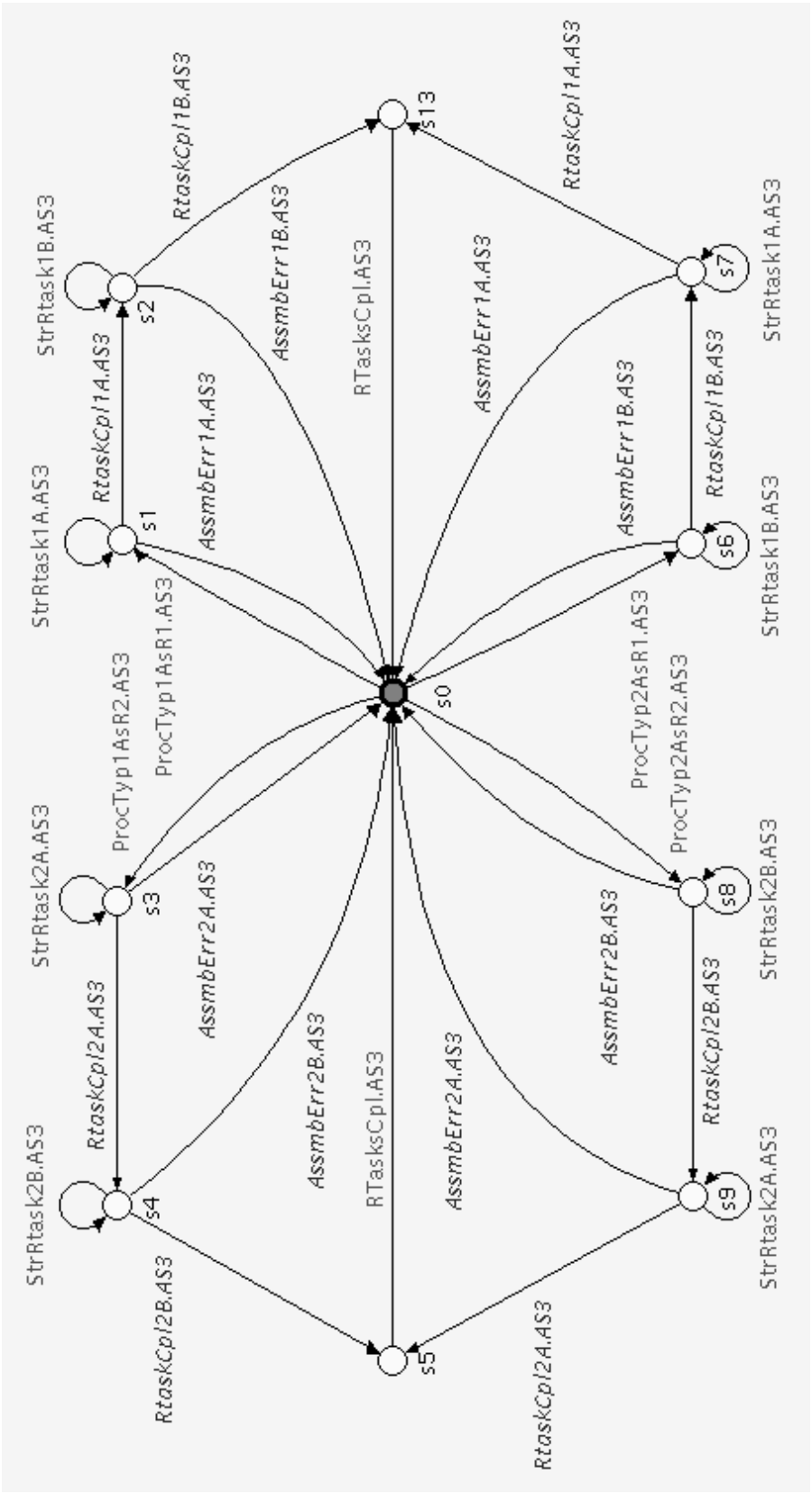


Figure 14.24: DoRobotTasks.AS3

Chapter 15

AIP Low Levels 4 and 5 (TU1 and TU2)

We now describe the *low levels* that represent transport units 1 and 2. As they are identical, we will describe them collectively as *low level* v, where $v = 4, 5$. We also define the companion indexes $X = 1, 2$, and $r = \text{TU1}, \text{TU2}$, who take their values relative to v (eg. $X = 1$ and $r = \text{TU1}$ when $v = 4$). Finally, for this chapter the variables q and i in the diagrams should be both set to r.

Low level v contains the 25 DES shown in Figure 15.1, which shows the definition of *low level* v*'s* subsystem G_{L_v}, plant component $\mathcal{G}_{L_v}$, and supervisor component $\mathcal{S}_{L_v}$. They are defined to be the synchronous product of the indicated automata.

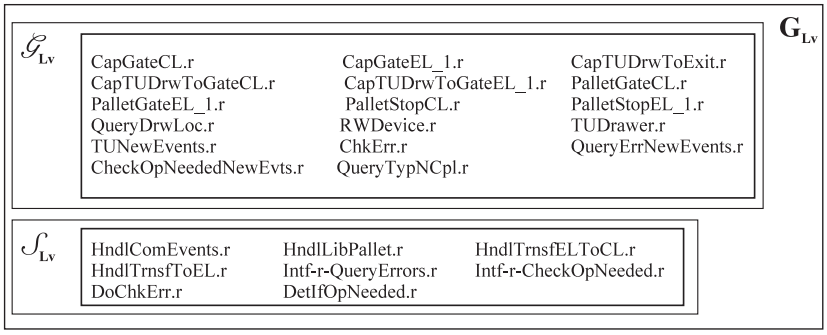


Figure 15.1: Low Level v

Low level v provides the functionality specified in its interface, shown in Figure 15.2. The transport units are used to transfer pallets between the central loop, and the external

loops (ie. TU1 transfers pallets between CL and EL 1). Transport unit X has two entry points for pallets, gate 5.X on the central loop, and gate X.1 on the external loop. If a pallet is at gate X.1, *low level v* transfers the pallet to the central loop. If a pallet is at gate 5.X, *low level v* can be requested to liberate the pallet (allow it to pass through and continue on the central loop), or to transfer the pallet to the external loop. When requested to transfer a pallet to the EL, *low level v* will only transfer the pallet if the pallet is undamaged, and if the next assembly task required by the pallet is performed by the external loops assembly station, and the station is not down.

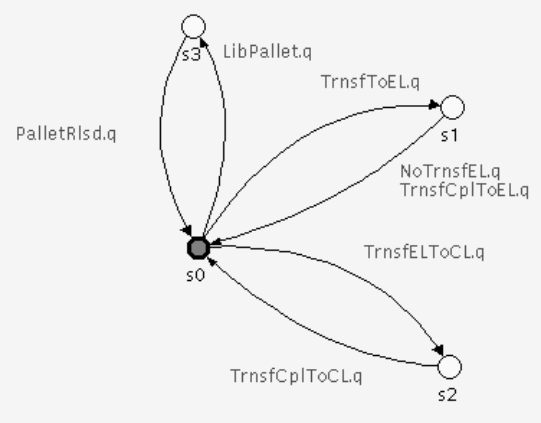


Figure 15.2: Interface to *Low Level v*.

Figure 15.3: TUNewEvents.q

15.1 Plant Component

We now discuss the plant models for *low level v*. We start with plant models **TUNewEvents.r, CapGateCL.r,** and **CapGateEL_1.r,** shown in Figures 15.3, 15.4, and 15.5. The first DES simply introduces new events that are used by the transport unit's interface. Plant **CapGateCL.r** models how many pallets can fit into gate 5.X at once. More importantly, it creates a dependency between events *TrnsfToEL.r* and *LibPallet.r* and pallets leaving the gate. As events *TrnsfToEL.r* and *LibPallet.r* are dependent on a pallet arriving at gate 5.X, this creates a dependency between pallets arriving and pallets leaving the gate. Similarly, DES **CapGateEL_1.r** creates a dependency between event *TrnsfELToCL.r* and pallets leaving gate X.1, and thus between pallets arriving and pallets leaving the gate.

We now discuss plants **CapTUDrwToExit.r, CapTUDrwToGateCL.r,** and **CapTUDrwToGateEL_1.r,** shown in Figures 15.6, 15.7, 15.8. DES **CapTUDrwToExit.r**

creates a dependency between pallets arriving at the transport drawer and pallets leaving the transport unit. DES **CapTUDrwToGateCL.r** creates a dependency between pallets leaving gate 5.X and arriving at the transport drawer. DES **CapTUDrwToGateEL_1.r** creates a dependency between pallets leaving gate X.1 and arriving at the transport drawer.

Next we discuss plants **PalletGateCL.r, PalletGateEL_1.r, PalletStopCL.r,** and **PalletStopEL_1.r,** shown in Figures 15.10, 15.11, 15.12, 15.13. These DES show the operation of pallet gates 5.X, X.1 and pallet stops 5.X, X.1, respectively.

The next plant models are **QueryDrwLoc.r, TUDrawer.r,** and **RWDevice.r,** shown in Figures 15.14, 15.15, and 15.16. DES **TUDrawer.r** models the operation of the transport drawer, while DES **QueryDrwLoc.r** stores the location (at the central loop or at EL X) of the transport drawer and provides a means of querying the location. Finally, DES **RWDevice.r** models reading from the pallet's electronic label.

We now discuss the plant models for tasks related to querying if a pallet has been damaged. These tasks consist of determining if a pallet has been damaged, and if so, what type of error has been sustained. These plant models, **QueryErrNewEvents.r** and **ChkErr.r,** are identical to those discussed on page 187 after substituting $t = r$.

We now discuss the plant models for tasks related to determining if a pallet needs to be transfered to transport unit X's attached external loop for assembly operation. We start with plant models **CheckOpNeededNewEvents** and **QueryTypNCpl.r ,** shown in Figures 15.9 and 14.12 (see page 190, with $t = r$). The first DES simply introduces new events that are used by related supervisors. The second DES provides a means of querying the type and assembly status of a pallet.

15.2 Supervisor Component

We now discuss the supervisors for *low level v.* We start with supervisor **HndlComEvents.r,** shown in Figure 15.17. This supervisor allows **HndlLibPallet.r, HndlTrnsfELToCL.r,** and **HndlTrnsfToEL.r** to be designed independent of each other but not cause each other to deadlock even though they use common events.

We now discuss supervisor **HndlLibPallet.r,** shown in Figure 15.19. The supervisor handles liberating pallets. It allows a pallet at gate 5.X on CL, to pass through the transport unit without being transfered to EL X.

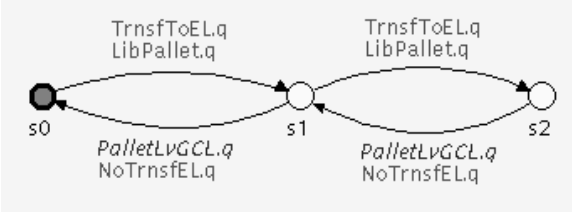


Figure 15.4: CapGateCL.q

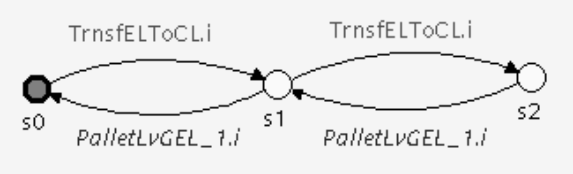


Figure 15.5: CapGateEL_1.i

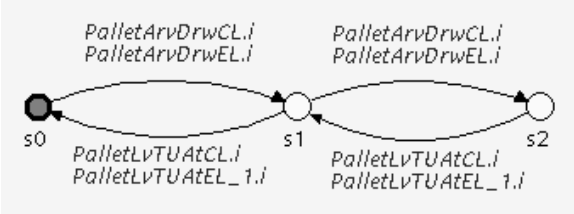


Figure 15.6: CapTUDrwToExit.i

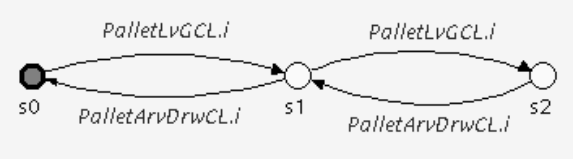


Figure 15.7: CapTUDrwToGateCL.i

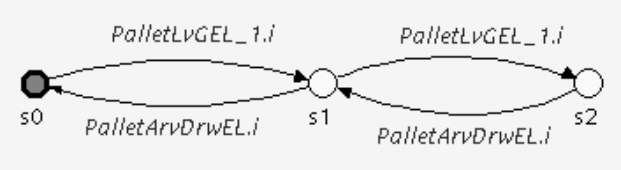


Figure 15.8: CapTUDrwToGateEL_1.i

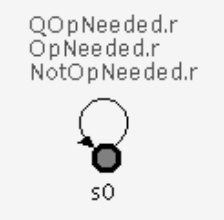


Figure 15.9: CheckOpNeededNewEvents.r

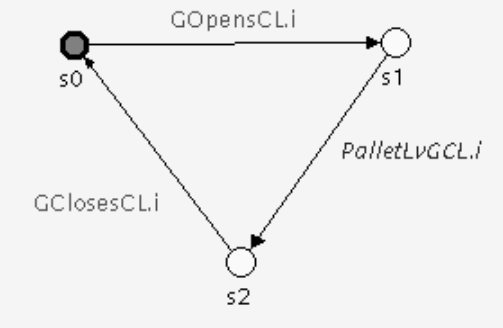


Figure 15.10: PalletGateCL.i

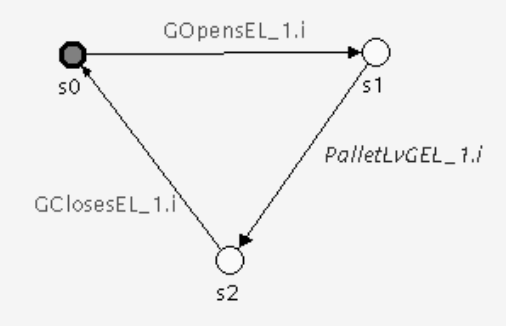


Figure 15.11: PalletGateEL_1.i

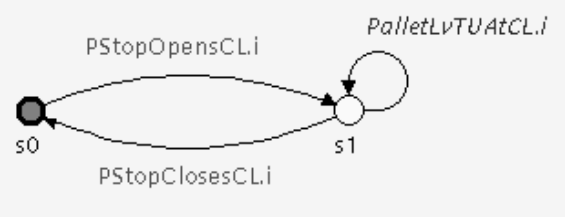


Figure 15.12: PalletStopCL.i

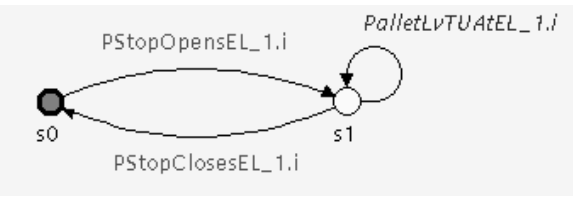


Figure 15.13: PalletStopEL_1.i

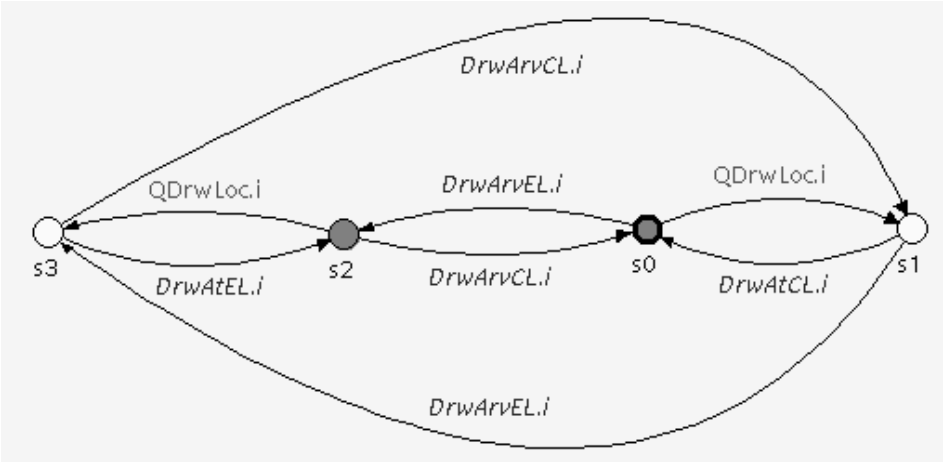


Figure 15.14: QueryDrwLoc.i

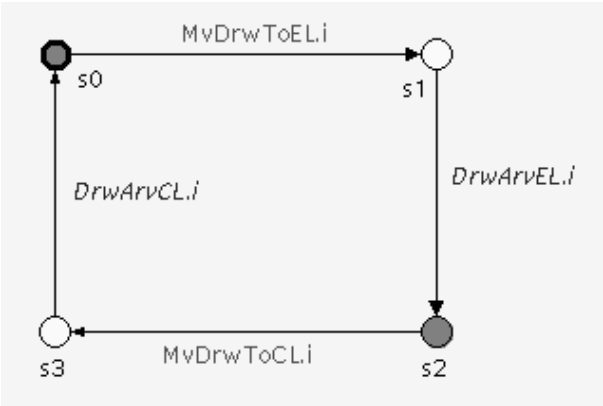


Figure 15.15: TUDrawer.i

Figure 15.16: RWDevice.i

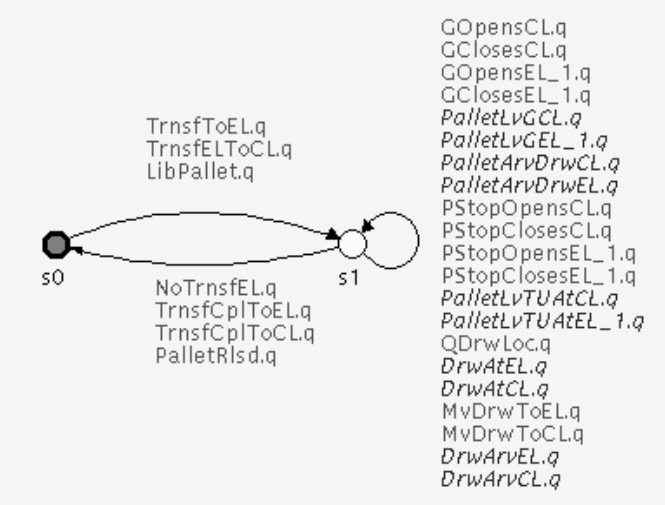


Figure 15.17: HndlComEvents.q

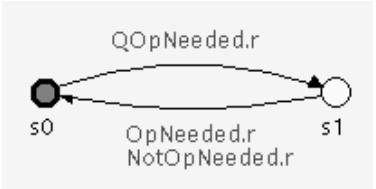


Figure 15.18: Intf-r-CheckOpNeeded.r

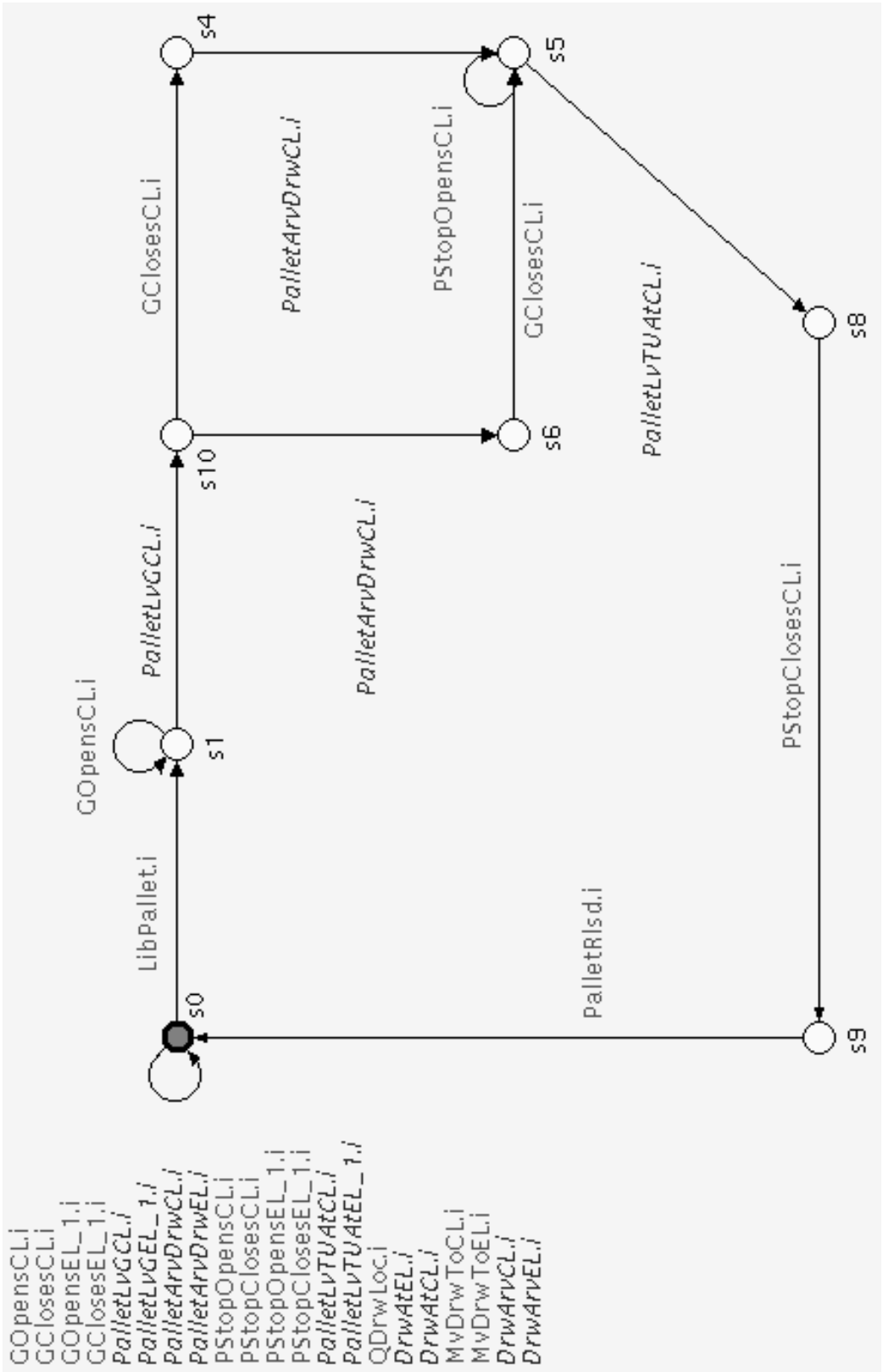


Figure 15.19: HndlLibPallet.i

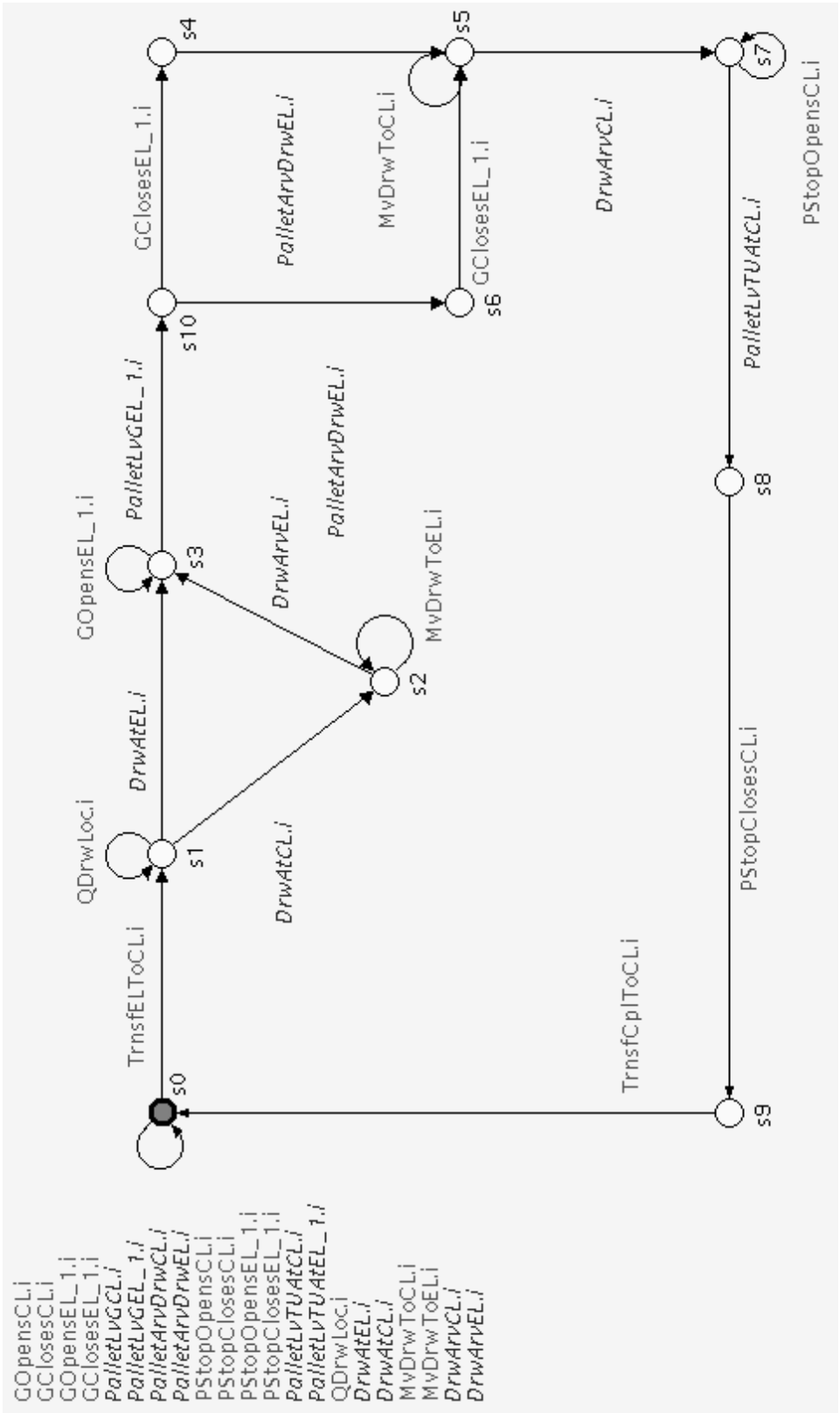


Figure 15.20: HndlTrnsfELToCL.i

We now discuss supervisor **HndlTrnsfELToCL.r,** shown in Figure 15.20. The supervisor handles transporting pallets from EL X to the central loop.

We now discuss supervisor **HndlTrnsfToEL.r,** shown in Figure 15.21. The supervisor handles transporting pallets from the central loop to EL X. It only transfers pallets if they are undamaged, and if the next assembly task required by the pallet is performed by assembly station X.

We now discuss the supervisors for tasks related to querying if a pallet has been damaged. These supervisors, **Intf-r-QuerryErrors.r**, and **DoChkErr.r,** are identical to those discussed on page 192 after substituting $t = r$.

We now discuss the supervisors for tasks related to determining if a pallet needs to be transfered to the transport units external loop. We start with supervisor **Intf-r-CheckOpNeeded.r**, shown in Figure 15.18. It defines the tasks required to determine if a pallet needs to be transfered.

We now discuss supervisors **DetIfOpNeeded.TU1** and **DetIfOpNeeded.TU2,** shown in Figures 15.22 and 15.23. These supervisors determine if the pallet needs to be transfered to external loop X for assembly operation.

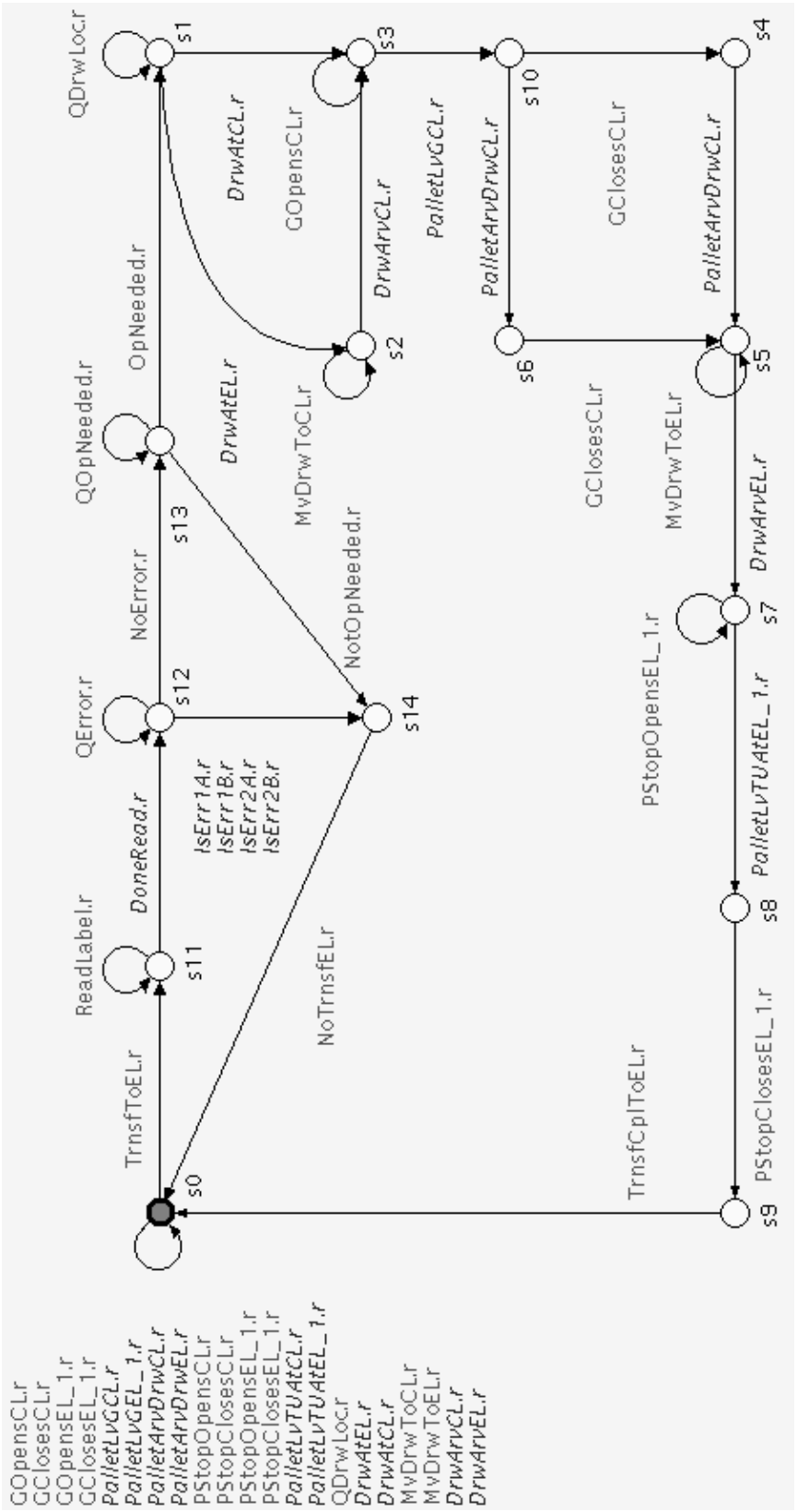


Figure 15.21: HndlTrnsfToEL.r

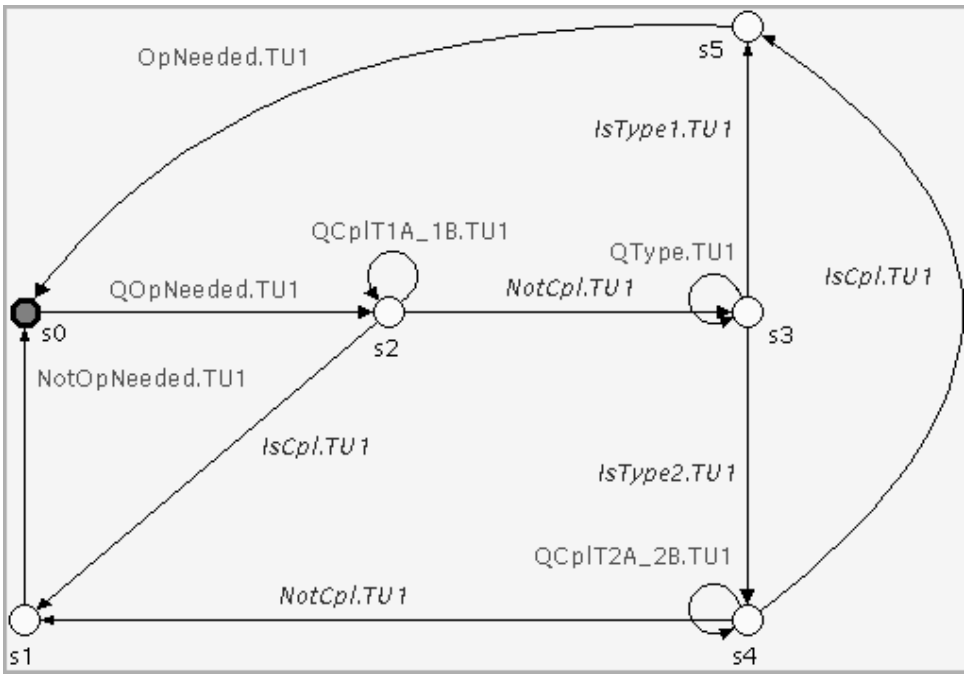


Figure 15.22: DetIfOpNeeded.TU1

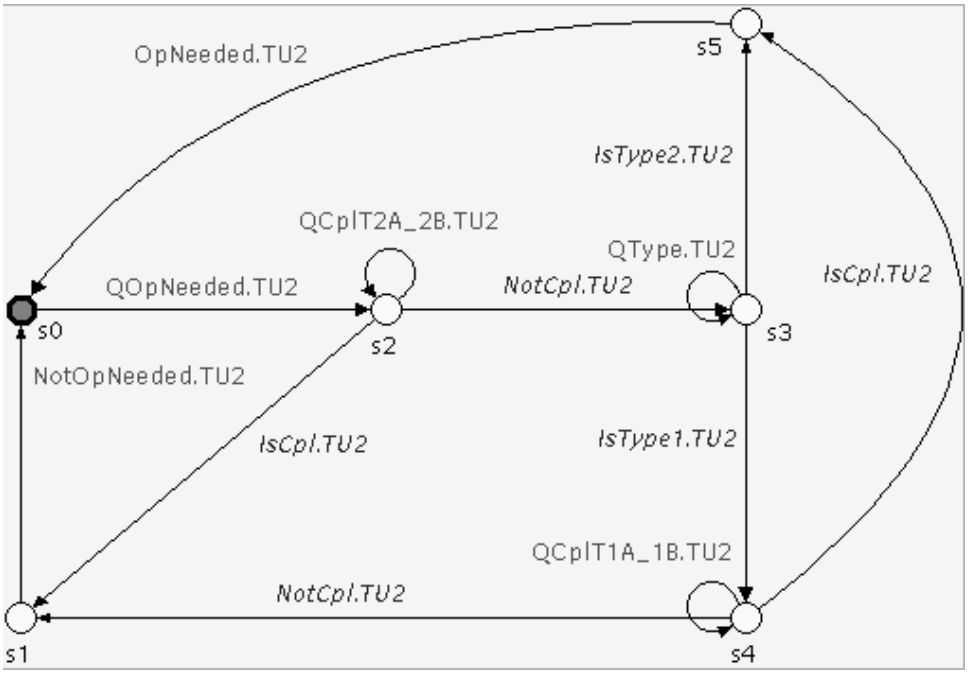


Figure 15.23: DetIfOpNeeded.TU2

Chapter 16

AIP Low Level 6 (TU3)

We now describe the *low level* that represents transport unit 3. *Low level 6* contains the 29 DES shown in Figure 16.1, which shows the definition of *low level 6's* subsystem G_{L_6}, plant component $\mathcal{G}_{L_6}$, and supervisor component $\mathcal{S}_{L_6}$. They are defined to be the synchronous product of the indicated automata.

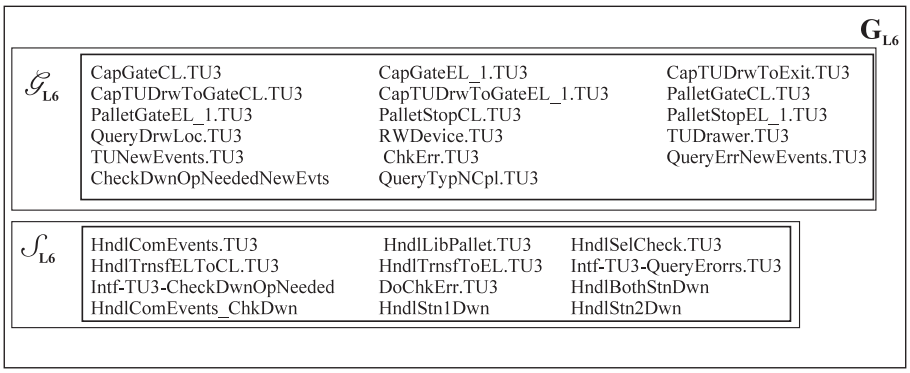


Figure 16.1: Low Level 6

Low level 6 provides the functionality specified in its interface, shown in Figure 16.2. *Low level 6* describes the behaviour of transport unit 3, which is very similar to TU1 and TU2. It differs in how it decides if a pallet should be transferred from the central loop to external loop 3. First, all damaged pallets are to be transferred to EL 3 for maintenance. Second, if an assembly station is down and it performs the next pending task for the pallet, then the pallet is to be transferred. As *low level 6* must know the breakdown status of assembly stations 1 and 2, this information is passed in explicitly as separate *request events*

(see Figure 16.2, transitions from state s_0 to s_1).

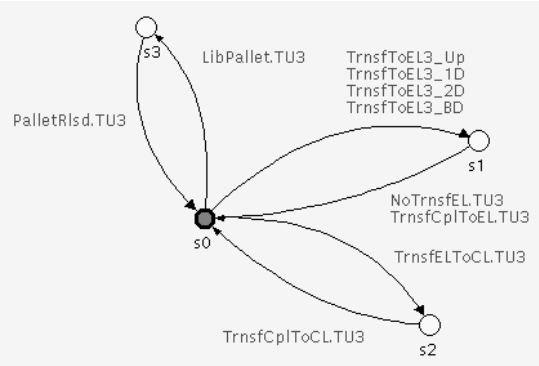


Figure 16.2: Interface to *Low Level 6*.

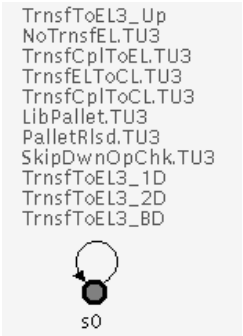


Figure 16.3: TUNewEvents.TU3

16.1 Plant Component

We now discuss the plant models for *low level 6*. We start with plant models **TUNewEvents.TU3,** and **CapGateCL.TU3,** shown in Figures 16.3 and 16.4. The first DES simply introduces new events that are used by the transport unit's interface.[1] Plant **CapGateCL.TU3** models how many pallets can fit into gate 5.3 at once. More importantly, it creates a dependency between events *TrnsfToEL3_Up, TrnsfToEL3_1D, TrnsfToEL3_2D, TrnsfToEL3_BD,* and *LibPallet.TU3* and pallets leaving the gate. As events *TrnsfToEL3_Up, TrnsfToEL3_1D, TrnsfToEL3_2D, TrnsfToEL3_BD,* and *LibPallet.TU3* are dependent on a pallet arriving at gate 5.3, this creates a dependency between pallets arriving and pallets leaving the gate.

The next 11 plant models are identical to those of *low levels 4* and *5* and are shown in Figures 15.5 to 15.8, and 15.10 to 15.16, with $i = \text{TU3}$.

We now discuss the plant models for tasks related to querying if a pallet has been damaged. These tasks consist of determining if a pallet has been damaged, and if so, what type of error has been sustained. These plant models, **QueryErrNewEvents.TU3** and **ChkErr.TU3,** are identical to those discussed on page 187 after substituting $t = TU3$.

We now discuss the plant models for tasks related to determining if a pallet needs to be transfered to transport unit 3's attached external loop for assembly operation. This is a

[1]Actually, event *SkipDwnOpChk.TU3* is used by supervisor **HndlSelCheck.TU3** to indicate that AS3 does not need to substitute for AS1 or AS2 (ie. both are up).

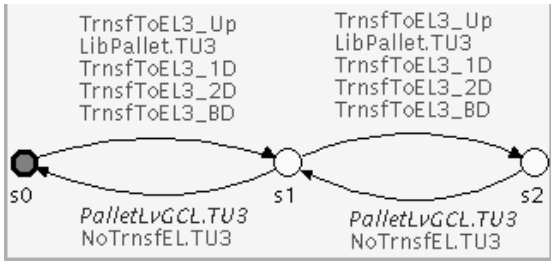


Figure 16.4: CapGateCL.TU3

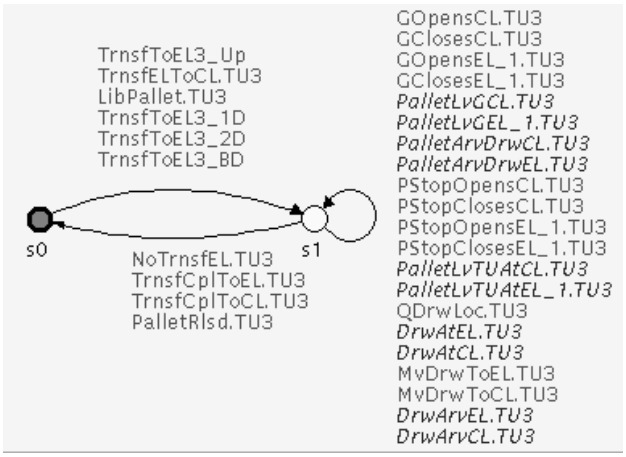


Figure 16.5: HndlComEvents.TU3

based on the next pending tasks for the pallet, and the breakdown status of the assembly station that would normally perform these tasks.

We start with plant models **CheckDwnOpNeededNewEvts** and **QueryTypNCpl.TU3,** shown in Figures 15.9 and 14.12 (see page 190, with t = TU3). The first DES simply introduces new events that are used by related supervisors. The second DES provides a means of querying the type and assembly status of a pallet.

16.2 Supervisor Component

We now discuss the supervisors for *low level 6.* We start with supervisor **HndlComEvents.TU3,** shown in Figure 16.5. This supervisor allows **HndlLibPallet.TU3, HndlTrnsfELToCL.TU3,** and **HndlTrnsfToEL.TU3** to be designed independent of each other but not cause each other to deadlock even though they use common events.

The next supervisor we discuss is **HndlSelCheck.TU3,** shown in Figure 16.7. This supervisor assists supervisor **HndlTrnsfToEL.TU3** by mapping the request events signalling a transfer to EL 3 and containing the breakdown status of assembly stations 1 and 2, to the appropriate events used by supervisor **Intf-TU3-CheckDwnOpNeeded**.

We now discuss supervisor **HndlTrnsfToEL.TU3,** shown in Figure 16.10. The supervisor handles transporting pallets from the central loop to EL 3. It only transfers pallets if they are damaged, or if an assembly operation is required by the pallet as a substitute for a down assembly station.

The next two supervisors for node **TU3**, **HndlLibPallet.TU3** and **HndlTrnsfEL-**

ToCL.TU3, are identical to those of nodes **TU1** and **TU2** and are shown in Figures 15.19 and 15.20, with i = TU3.

We now discuss the supervisors for tasks related to querying if a pallet has been damaged. These supervisors, **Intf-TU3-QuerryErrors.TU3**, and **DoChkErrTU3,** are identical to those discussed on page 192 after substituting $t = TU3$.

We now discuss the supervisors for tasks related to determining if a pallet needs to be transfered to the unit's attached external loop for assembly operation. We start with supervisor **Intf-TU3-CheckDwnOpNeeded**, shown in Figure 16.8. It defines the tasks required to determine if a pallet needs to be transfered.

We now discuss supervisor **HndlComEvents_ChkDwn,** shown in Figure 16.9. The supervisor allows **HndlStn1Dwn, HndlStn2Dwn,** and **HndlBothStnDwn** to be designed independent of each other but not cause each other to deadlock even though they use common events.

The remaining supervisors are **HndlStn1Dwn, HndlStn2Dwn, HndlBothStnDwn.** They are shown in Figures 16.11, 16.12, and 16.13. These supervisors determine if the pallet needs to be transfered to TU3's attached external loop for assembly operation. DES **HndlStn1Dwn** handles the case that only AS1 has broken down. It checks if the next tasks pending for the pallet are handled by AS1, and if so, the pallet is to be transferred. DES **HndlStn2Dwn** performs the identical task for the case AS2 has broken down. DES **HndlBothStnDwn** handles the case both assembly stations are down. In this case, it is sufficient to check that the pallet still has assembly tasks pending.

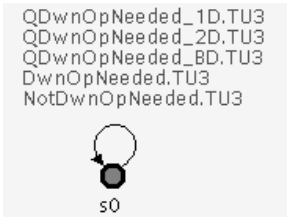


Figure 16.6: CheckDwnOpNeededNew-Evts

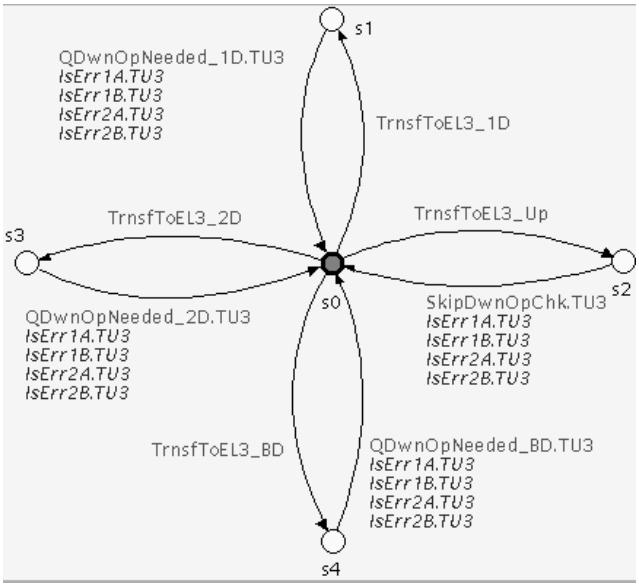


Figure 16.7: HndlSelCheck.TU3

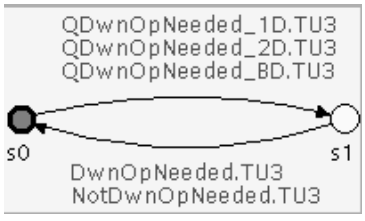


Figure 16.8: Intf-TU3-CheckDwnOpNeeded

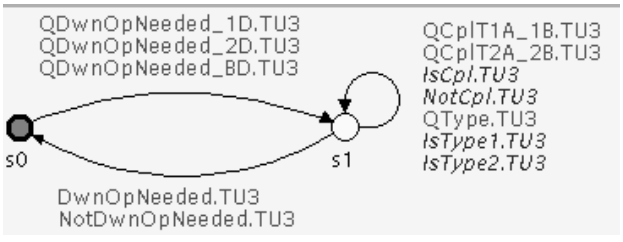


Figure 16.9: HndlComEvents_ChkDwn

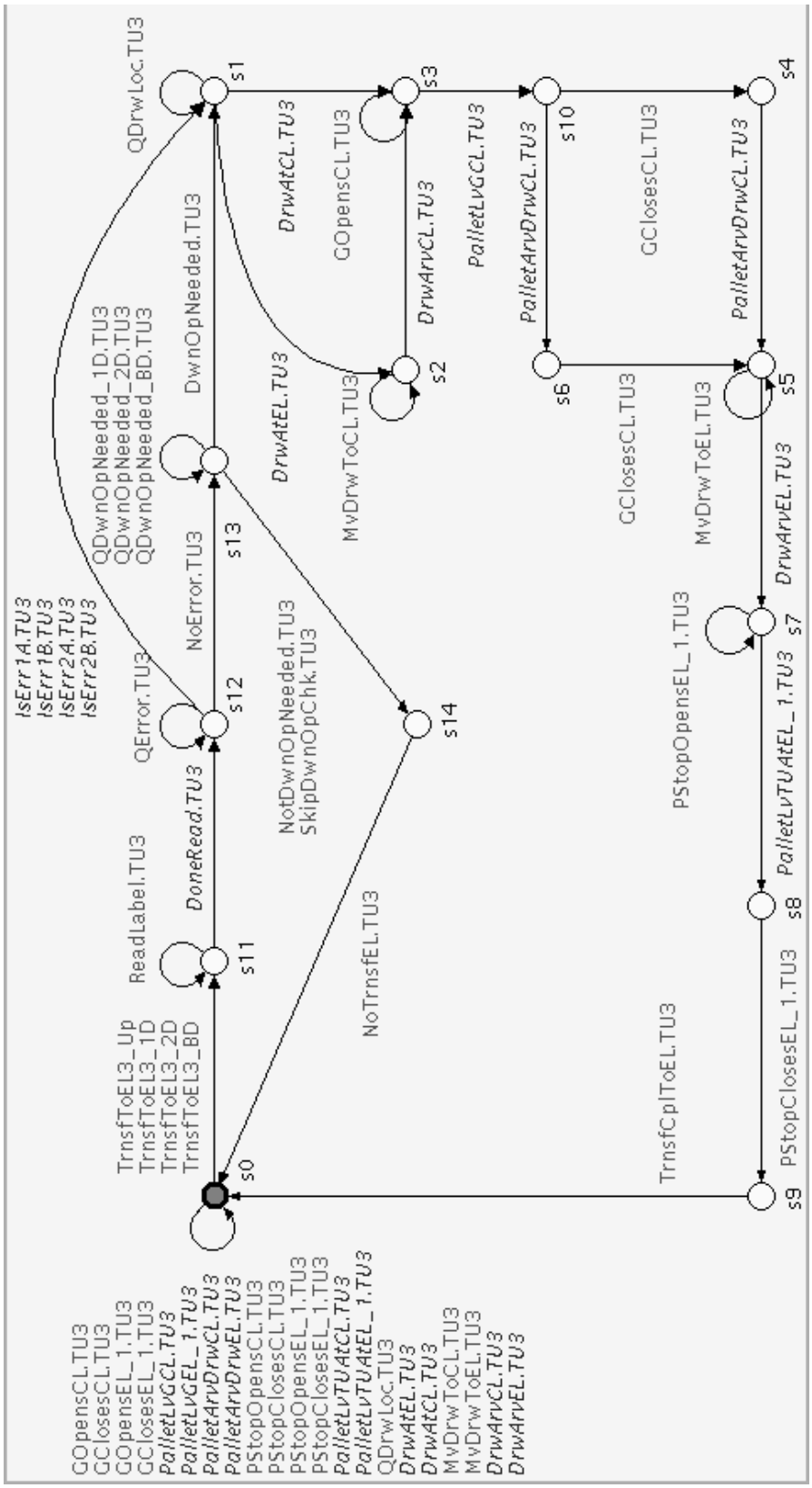


Figure 16.10: HndlTrnsfToEL.TU3

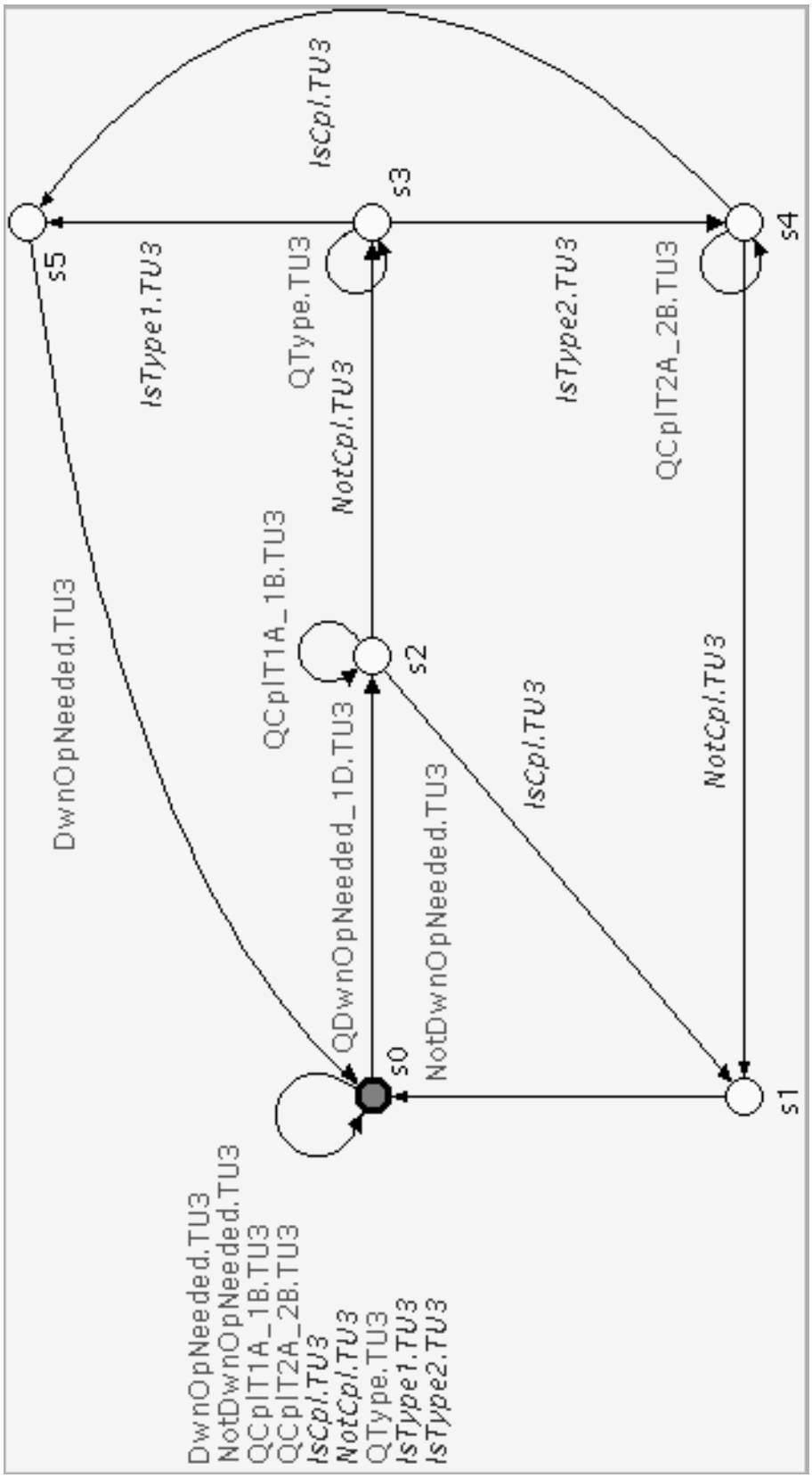


Figure 16.11: HndlStn1Dwn

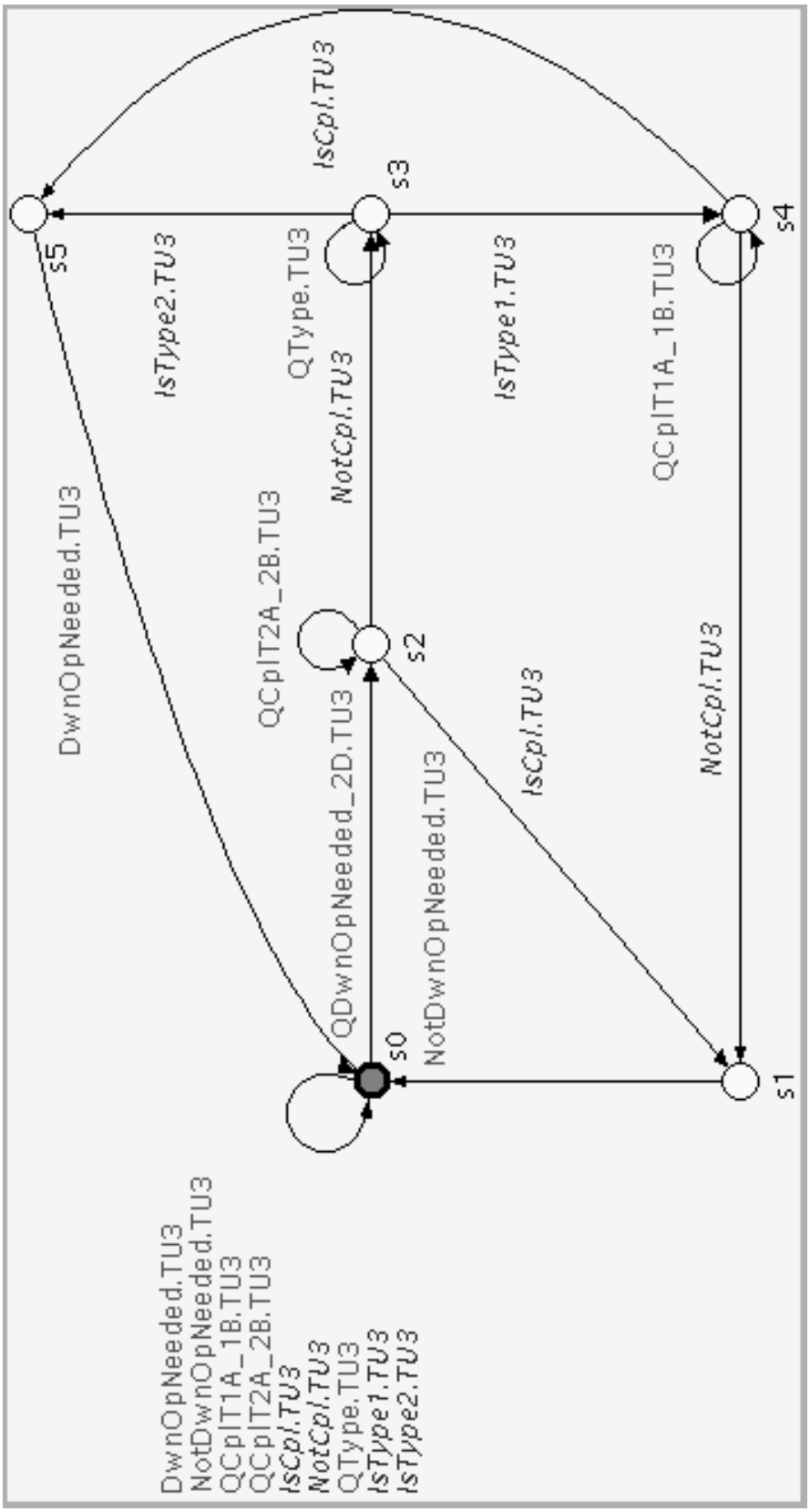


Figure 16.12: HndlStn2Dwn

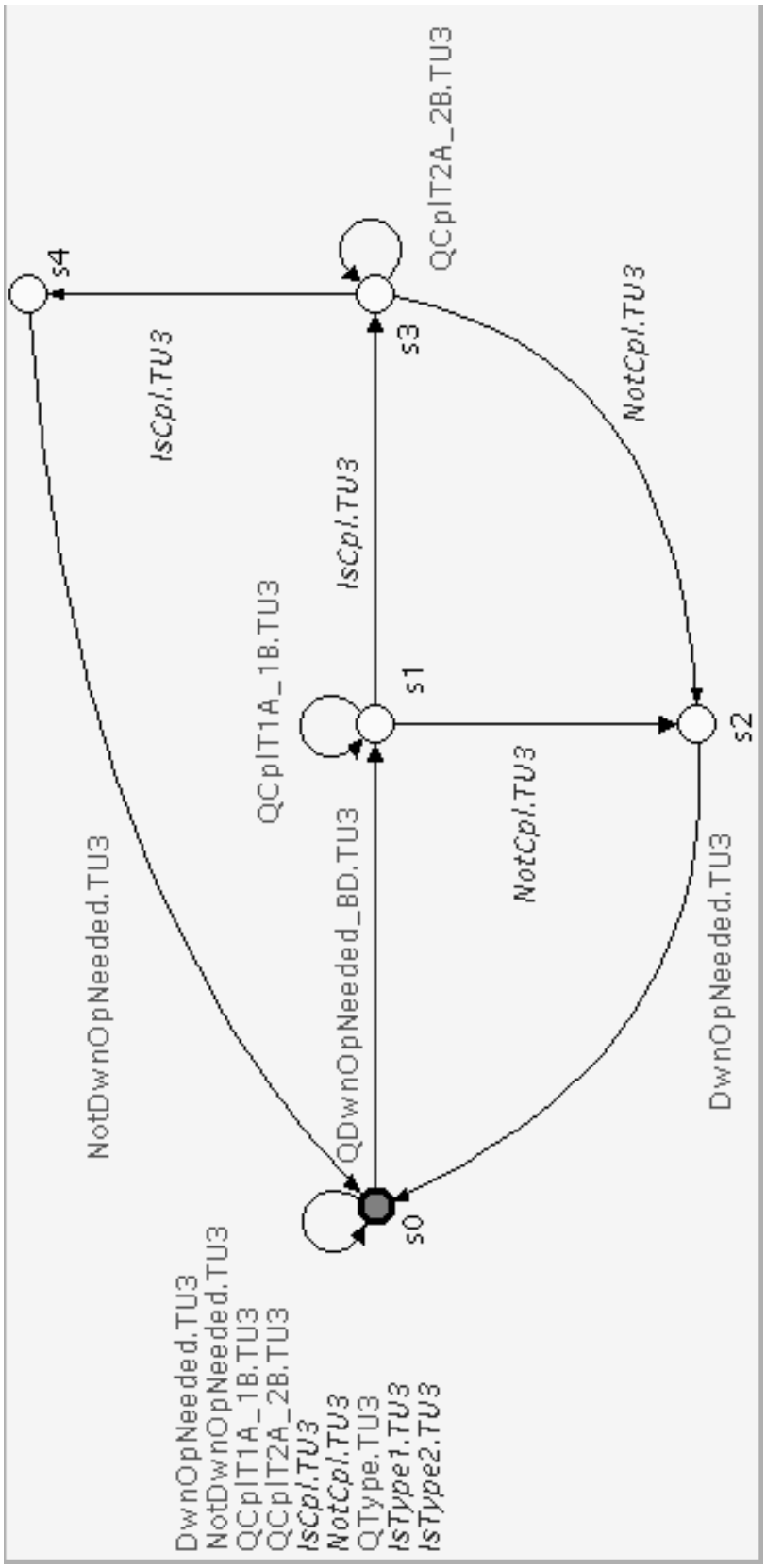


Figure 16.13: HndlBothStnDwn

Chapter 17

AIP Low Level 7 (TU4)

We now describe the *low level* that represents transport unit 4. *Low level 7* contains the 19 DES shown in Figure 17.1, which shows the definition of *low level 7's* subsystem G_{L_7}, plant component $\mathcal{G}_{L_7}$, and supervisor component $\mathcal{S}_{L_7}$. They are defined to be the synchronous product of the indicated automata.

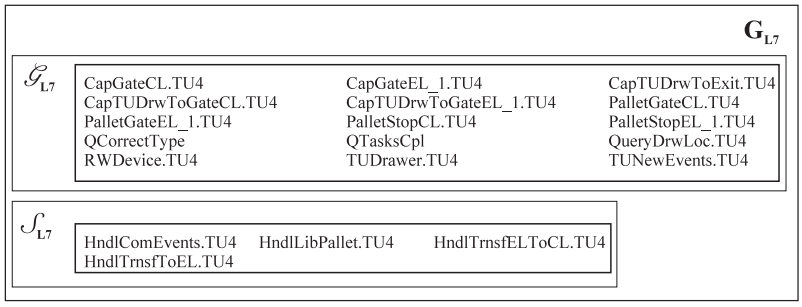


Figure 17.1: Low Level 7

Figure 17.2: QTasksCpl

Low level 7 provides the functionality specified in its interface, shown in Figure 15.2 (See page 199, with $q = TU4$). It describes the behaviour of transport unit 4, which is very similar to TU1 and TU2. *Low level 7* differs in how it decides if a pallet should be transferred from the central loop to external loop 4, which contains the I/O station (the I/O station is where pallets enter and leave the system). As pallets are required to leave the system in a particular order (ie. type 1, type 2, type 1, ...), *Low level 7* keeps track of the type of the last pallet to be transferred to EL 4 and will only transfer the current pallet if it is of the type required by the sequence, and if all required assembly tasks have been successfully performed on the pallet.

17.1 Plant Component

We now discuss the plant models for *low level 7*. We start with plant models **QTasksCpl,** and **QCorrectType,** shown in Figures 17.2 and 17.3. The first DES provides a means to query the assembly status of the pallet. **QTasksCpl** uses the data from the last read operation performed by TU4's R/W device. If a read has not yet been performed, the results are undefined. The second DES stores the type of last pallet to be transferred to EL 4, and provides a means to determine if the current pallet is of the next required type.

The remaining plant models for *low level 7* are identical to those for TU1 and TU2 and are shown in Figures 15.5 to 15.8, and 15.10 to 15.16, with i = TU4, and q = TU4.

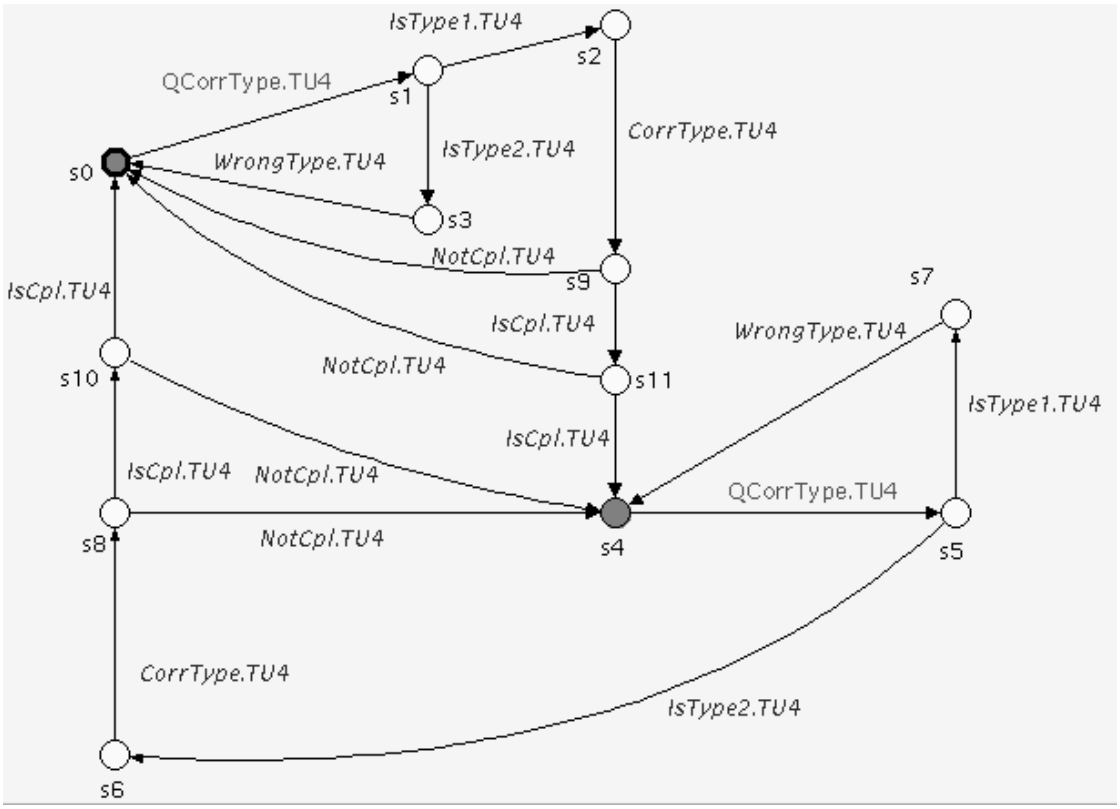


Figure 17.3: QCorrectType

17.2 Supervisor Component

We now discuss the supervisors for *low level 7*. We start with supervisor **HndlTrnsfToEL.TU4,** shown in Figure 17.4. The supervisor handles transporting pallets from the

central loop to EL 4. It only transfers the current pallet if it is of the type required by the exit sequence, and if all required assembly tasks have been successfully performed on the pallet.

The remaining three supervisors for *low level 7* are identical to those for TU1 and TU2 and are shown in Figures 15.17, 15.19 and 15.20, with $i = \text{TU4}$ and $q = \text{TU4}$.

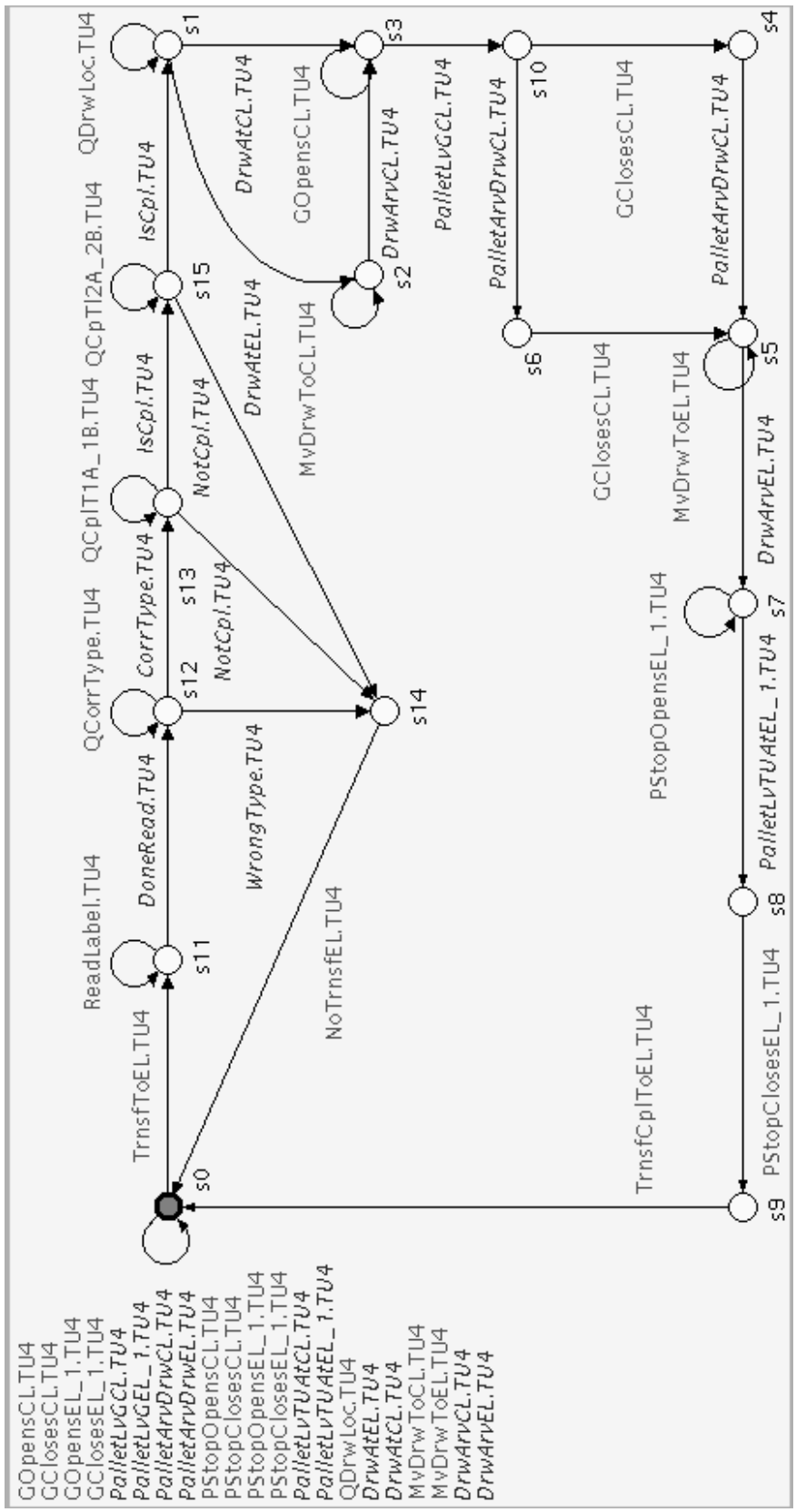


Figure 17.4: HndlTrnsfToEL.TU4

Chapter 18

AIP Results and Discussion

In this Chapter, we evaluate our system to determine if it satisfies the parallel case conditions, and is thus controllable and nonblocking. We then discuss the results, and present our conclusions.

18.1 Evaluating Properties

We will now show that the system shown in Figure 11.4 and defined in Chapters 12-17 is controllable and nonblocking; that is to say, its corresponding *flat system* is nonblocking, and its *flat supervisor* is controllable for its *flat plant.*

To verify controllability and nonblocking, we will use **Theorems 3** and **4**. To apply the theorems, we must first verify that the system is *level-wise non-blocking, level-wise controllable,* and *interface consistent.* As we have a *parallel system* of degree $n = 7$, this means we must verify that the seven *serial extraction systems* are *serial level-wise non-blocking, serial level-wise controllable,* and *serial interface consistent.*

Our first step is to show that sets Σ_H, Σ_{R_j}, Σ_{A_j}, and Σ_{L_j} $(j = 1, \ldots, 7)$ are pairwise disjoint. This can be seen by inspection from their definitions on page 163.

Next, we define the j^{th}*serial extraction system, system*(j),[1] where $j = 1, \ldots, 7$.

$$\begin{aligned} G_H(j) &:= G_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}}||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n} \\ \mathcal{G}_H(j) &:= \mathcal{G}_H||_s G_{I_1}||_s \ldots ||_s G_{I_{(j-1)}}||_s G_{I_{(j+1)}}||_s \ldots ||_s G_{I_n} \end{aligned}$$

[1] Actually, we are giving both the subsystem form and the general form components as one group.

$$
\begin{aligned}
\mathcal{S}_H(j) &:= \mathcal{S}_H \\
G_L(j) &:= G_{L_j} \\
\mathcal{G}_L(j) &:= \mathcal{G}_{L_j} \\
\mathcal{S}_L(j) &:= \mathcal{S}_{L_j} \\
G_I(j) &:= G_{I_j} \\
\Sigma_H(j) &:= [\dot{\bigcup}_{k\in\{1,\ldots,(j-1),(j+1),\ldots,n\}}\Sigma_{I_k}]\,\dot{\cup}\,\Sigma_H \\
\Sigma_L(j) &:= \Sigma_{L_j} \\
\Sigma_R(j) &:= \Sigma_{R_j} \\
\Sigma_A(j) &:= \Sigma_{A_j} \\
\Sigma(j) &:= \Sigma_H(j)\,\dot{\cup}\,\Sigma_L(j)\,\dot{\cup}\,\Sigma_R(j)\,\dot{\cup}\,\Sigma_A(j)
\end{aligned}
$$

We now apply our research tool to the seven *serial extraction systems* and we find that they are all *serial level-wise non-blocking, serial level-wise controllable,* and *serial interface consistent.* We can thus conclude that the system is *level-wise non-blocking, level-wise controllable,* and *interface consistent.* This allows us to conclude by **Theorems 3** and **4**, that the *flat system* is nonblocking and that the system's *flat supervisor* is controllable for the *flat plant.*

18.2 Discussion of Results

The AIP was an excellent example for our method due to its size, complexity, and natural partition of tasks. The system was quite large, containing 181 DES in total, with an estimated closed-loop state space of size 7.01×10^{21}. This estimate was calculated by determining the closed-loop state space of the *high level,* and each *low level* and then these values were multiplied together to create a worst case state estimate. The computation ran for 25 minutes and used 760MB of memory. The machine used was a 750MHz Athlon system, with 512MB of RAM, 2GB of swap, and running Redhat Linux 6.2. A standard nonblocking verification was also attempted on the monolithic system, but it quickly failed due to lack of memory.

It is worth noting at this point that Zhang et al. have recently developed algorithms

that use Integer Decision Diagrams, an extension of Binary Decision Diagrams, to verify centralized DES systems on the order of 10^{23} states [65, 66]. These results are complementary to the hierarchical method illustrated here, as their approach can be used to verify many of the required conditions, allowing HISC to scale to even larger systems.

18.3 *Star* and *Command-pair Interface* Comparison

The AIP example illustrates very well the primary advantage of a *command-pair interface* over a *star interface*: the ability to store state information about the *low level* and to alter the selection of *request events* (and *answer events*) based on this information. In the AIP example, *star interfaces* were very effective for all *low levels* except for *low levels* 1 and 2 (assembly stations 1 and 2). For AS1, the *star interface* is as given in Figure 18.1. Because all *request events* must be eligible at the initial state, we are forced to make both events *ProcPallet.AS1* and *DoRpr.AS1* eligible at state 0. The problem is that it doesn't make sense for event *DoRpr.AS1* to occur until after event *ASDwn.AS1* has occurred. Similarly, event *ProcPallet.AS1* shouldn't occur after event *ASDwn.AS1* until event *RobUp.AS1* has occurred. This can easily be avoided using *command-pair interfaces,* as shown in Figure 18.1.

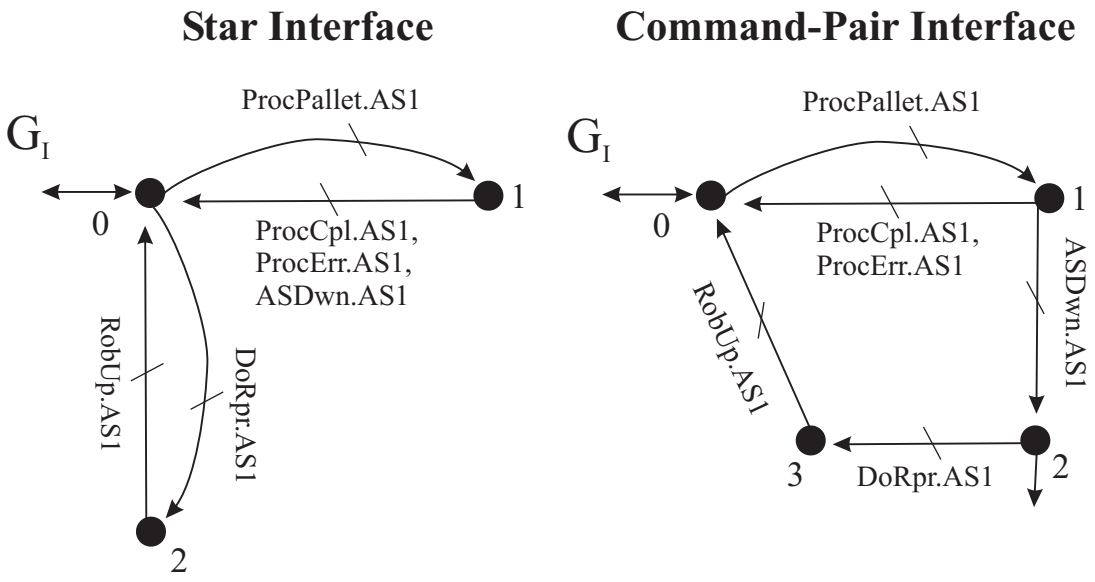


Figure 18.1: Interfaces for Assembly Station 1

As we can see from this discussion, the more restrictive *star interface* is excellent for representing *low levels* that are stateless with respect to *interface events*. As many systems fall into this category, *star interfaces* are useful due to their regular structure, making them easy to construct. The ability of *command-pair interfaces* to store state information allows them to be more flexible, elegant, and handle a broader range of systems.

18.4 Systems with Dynamic Architecture

The HISC approach can handle some systems with architectures that are not static. Obviously, a *high level* can't become a *low level* and vice versa, but the method can certainly handle some variation in the type and number of *low levels.*

When a system's structure is static, we simply model each *low level* and how the *high level* uses them. For a system able to handle a variety of client (*low level*) types and a variable number of clients, we can model the system in a more general way as long as we know the details of each client type, and the maximum number of each. We would create a *low level* and *interface* for each instance of each client type. We would then have an event to activate and deactivate each *low level.* Deactivating a *low level* would be accomplished by the *high level* disabling all *request events* for the *low level.* It could then be activated by the *high level* starting to enable its *request events.* The interface conditions guarantee that this can be done without making the system nonblocking, or the supervisors uncontrollable. We would also want the *serial system extractions* to be *serial interface strict marking* as this would guarantee that a given *low level* is in a marked state when its corresponding *interface* is. We are assuming that the *low level* is in a quiescent state when it is in a marked state and that it's safe to disconnect it.

When a new client is physically added to the system, the corresponding activation event would occur and the *high level* could then begin to use its *interface.* When we want to remove a client, we would generate the deactivation event (perhaps a switch, or a computer console command) and then wait for the client's interface to return to a marked state. Once this has occurred, we can physically disconnect the client.

In the AIP example, we have four external loops with distinct capabilities. We could have designed our *high level* to be able to handle two of each type of external loop, but to start up with only one of each type activated. We could then physically add additional loops (and later remove) as production required.

Chapter 19

Conclusions and Future Work

In this chapter, we discuss extensions to our work, and then present our conclusions.

19.1 Future Work

The HISC method provides an excellent foundation for future work. Some outstanding issues to address are:

- *Generalizing the Interface:* It would be useful to allow the behavior representable by an *interface* to be more general. In particular, to allow multiple commands before a reply is returned (this would be analogous to providing additional information while a command is being processed, or perhaps an interrupt condition), and to allow for bi-directional flow of commands (for example, the *low level* might request a status report on another part of the system).

- *Interface-based Synthesis:* It would be useful to develop a per component supervisor synthesis method that respects the interface conditions.

- *Multi-level Case:* The method presented here scales well horizontally (ie. adding more *low levels*), but not vertically. If a given component becomes too complicated, the method would fail. What we need is to be able to decompose the system into a multi-level structure, with components communicating with the component above them and the components below them in the hierarchy via interfaces.

- *Model Correctness:* The approach here focuses on verification of the system model, but this does not imply that the results will hold true for the physical system. It

would be useful to extend the results to ensuring that the model accurately describes the physical system, and that the implementation of the supervisors produces results consistent with the modelled behavior.

- *Parameterization and Guard Conditions:* It is often the case that actions are parameterized and have guard conditions (ie. [19, 28, 42, 49]). Parameterization can be handled explicitly in automata by associating a separate event for each parameter value. Guard conditions can be handled by associating an explicit state with the guard condition such that the event is not possible until the system reaches this state. It would be useful to be able to handle these in the HISC method without having to "automatonize" them as this would allow us to express this information in a more compact form.

- Infinite State Systems: This work deals only with finite state systems. Many infinite state systems (ie. a system with temperature as a state variable, or clocks as in the timed automata model of [49]) can be handled if we can represent them as equivalent (for our purposes) finite state systems. For instance, we may only need to know if the temperature (T) is less then some minimum value (t_{min}), greater than some maximum value (t_{max}), or that $t_{min} \leq T \leq t_{max}$. We can then replace temperature by a variable with three values. It would be interesting to extend the HISC method to be able to deal with infinite state systems explicitly.

19.2 Conclusions

In this thesis, we have presented a method for DES design and verification that implements many of the concepts of "information hiding" from software engineering and thus provides us with the benefits discussed in Section 1.2 such as independent development, high degree of changeability and comprehensibility, and an excellent means to manage complexity by hiding unnecessary detail behind interfaces.

Our method, *hierarchical interface-based supervisory control,* offers an effective means to model systems with a natural master-slave structure. The method offers an intuitive way to model and design the system. Using multiple ($n \geq 1$) *low level subsystems* allows the subsystems to be independently modelled and verified, while still allowing a high degree of concurrent operation. As each requirement can be verified using only one subsystem,

the entire plant model never needs to be constructed or traversed (in computer memory), offering potentially significant savings in computation. This is immediately apparent by noting that the time complexity for analyzing a system by our method is $\mathbf{O}(m^2)$ ($m = n+1$ is the total number of subsystems), as compared to a monolithic analysis which is $\mathbf{O}(N^{2m})$ ($N \geq 0$ is an upper bound for the statespace size of the subsystems).

Finally, we discussed a large example based on the automated manufacturing system of the Atelier Inter-établissement de Productique (AIP). As the example contains 181 DES with an estimated closed-loop statespace of size 7×10^{21}, it demonstrates that the HISC method can be applied to interesting systems of realistic complexity that were previously beyond the means of monolithic, modular, or hierarchical supervisor design techniques. Once this approach is coupled with the Integer Decision Diagrams work of Zhang et al. [66, 65], the size of systems that can be handled will greatly increase again.

Bibliography

[1] N. Alsop. *Formal Techniques for the Procedural Control of Industrial Processes*. PhD thesis, Department of Chemical Engineering and Chemical Technology, Imperial College of Science, Technology and Medicine, London, 1996.

[2] Rajeev Alur and Thomas A. Henzinger. Local liveness for compositional modelling of fair reactive systems. In *Proc. of seventh Int. Conf. on Computer-aided Verification, Lecture Notes in Computer Science*, pages 166–179, 1995.

[3] A. Arnold. *Finite Transition Systems*. Prentice Hall, 1994.

[4] Adnan Aziz, Vigyan Singhal, and Gitanjali M. Swamy. Minimizing interacting finite state machines: A compositional approach to language containment. In *Proc. of IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors*, pages 255–261, Cambridge, Massachusetts, Oct 1994.

[5] George Barrett and Stephane Lafortune. Decentralized supervisory control with communicating controllers. *IEEE Trans. Automatic Control*, 45(9):1620–1638, 2000.

[6] John Bourne. *Object-Oriented Engineering: Building Engineering Systems Using Smalltalk-80*. Aksen Associates, 1992.

[7] Bertil Brandin and François Charbonnier. The supervisory control of the automated manufacturing system of the AIP. In *Proc. Rensselaer's 1994 Fourth International Conference on Computer Integrated Manufacturing and Automation Technology*, pages 319–324, Troy, Oct 1994.

[8] Y. Brave and M. Heymann. Control of discrete event systems modeled as hierarchical state machines. *IEEE Trans. on Automatic Control*, 38(12):1803–1819, Dec 1993.

[9] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, C-35(8), 1986.

[10] J.R. Burch, Edmund M. Clarke, and K.L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.

[11] P.E. Caines and Y.J. Wei. The hierarchical lattices of a finite machine. *Systems Control Letters*, 25:257–263, July 1995.

[12] F. Charbonnier. Commande par supervision des systèmes à événements discrets: application à un site expérimental l'Atelier Inter-établissement de Productique. Technical report, Laboratoire d'Automatique de Grenoble, Grenoble, France, 1994.

[13] Haoxun Chen and Hans-Michael Hanisch. Model aggregation for hierarchical control synthesis of discrete event systems. In *Proc. 39th Conf. Decision Contr.*, pages 418–423, Sydney, Australia, December 2000.

[14] S.-L. Chen. Existence and design of supervisors for vector discrete event systems. Master's thesis, Department of Electrical Engineering, University of Toronto, Toronto, Ont, 1992.

[15] S.-L. Chen. *Control of Discrete-Event Systems of Vector and Mixed Structural Type*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ont, 1996.

[16] Yi-Liang Chen and Feng Lin. Hierarchical modeling and abstraction of discrete event systems using finite state machines with parameters. In *Proc. 40th Conf. Decision Contr.*, pages 4110–4115, Orlando, USA, December 2001.

[17] Edmund M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Programming Languages and Systems*, 8(2):244–263, April 1986.

[18] Edmund M. Clarke, O. Grümberg, and K. Hamaguchi. Another look at LTL model checking. In *Proc. of 6th Conf. on Computer Aided Verification*, number 818 in LNCS, pages 415–427. Springer-Verlag, 1994.

[19] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2001.

[20] M. Courvoisier, M.Combacau, and A. de Bonneval. Control and monitoring of large discrete event systems: a generic approach. In *Proc. of ISIE 93*, pages 571–576, Budapest, 1993.

[21] E. W. Endsley, M. R. Lucas, and D. M. Tilbury. Modular design and verification of logic control for reconfigurable machining systems. Submitted to Discrete Event Dynamic Systems: Theory and Applications.

[22] Jose M. Eyzell and Jose E.R. Cury. Exploiting symmetry in the synthesis of supervisors for discrete event systems. In *Proc. of American Control Conference*, pages 244–248, Philadelphia, USA, June 1998.

[23] 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.

[24] Michael T. Goodrich and Roberto Tamassia. *Algorithm Design*. Wiley, 2001.

[25] O. Grümberg and D.E. Long. Model checking and modular verification. In *Proc. of CONCOUR'91*, number 527 in LNCS, pages 361–375. Springer-Verlag, 1991.

[26] Daniel M. Hoffman and David M. Weiss, editors. *Software Fundamentals. Collected Papers by David L. Parnas*. Addison Wesley, 2001.

[27] Paul Hubbard and Peter E. Caines. Trace-DC hierarchical supervisory control with applications to transfer-lines. In *Proc. 37th Conf. Decision Contr.*, pages 3293–3298, Tampa, Florida USA, December 1998.

[28] Mark Lawford. *Model Reduction of Discrete Real-Time Systems*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ont, 1997.

[29] R. Leduc, M. Lawford, and W. Murray Wonham. Hierarchical interface-based supervisory control: AIP example. In *Proc. of 39th Annual Allerton Conference on Comm., Contr., and Comp.*, pages 396–405, Oct 2001.

[30] R.J. Leduc, B.A. Brandin, and W. Murray Wonham. Hierarchical interface-based non-blocking verification. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, pages 1–6, May 2000.

[31] R.J. Leduc, B.A. Brandin, W. Murray Wonham, and M. Lawford. Hierarchical interface-based supervisory control: Serial case. In *Proc. of 40th Conf. Decision Contr.*, pages 4116–4121, Orlando, USA, December 2001.

[32] R.J. Leduc, M. Lawford, and W. Murray Wonham. Hierarchical interface-based supervisory control, part II: Parallel case. Submitted to IEEE Trans. Automatic Control, Aug, 2003. An earlier version is available as Software Quality Research Laboratory Report No. 13, Dept. of Computing and Software, McMaster University, Hamilton, ON. [ONLINE] Available: http://www.cas.mcmaster.ca/sqrl/sqrl_reports.html.

[33] R.J. Leduc, W. Murray Wonham, and M. Lawford. Hierarchical interface-based supervisory control: Parallel case. In *Proc. of 39th Annual Allerton Conference on Comm., Contr., and Comp.*, pages 386–395, Oct 2001.

[34] 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.

[35] Y. Li. *Control of Vector Discrete-Event Systems*. PhD thesis, Department of Electrical Engineering, University of Toronto, Toronto, Ont, 1991.

[36] F. Lin and W. Murray Wonham. Decentralized control and coordination of discrete-event systems with partial observations. In *Proc. 27th IEEE Conf. Decision Contr.*, pages 1125–1130, Dec 1988.

[37] Hong Liu, Jun-Cheol Park, and Raymond E. Miller. On hybrid synthesis for hierarchical structured petri nets. Technical report, Department of Computer Science, University of Maryland, College Park, MD, 1996.

[38] Chuan Ma. A computational approach to top-down hierarchical supervisory control of DES. Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ont, 1999.

[39] E.M. Clarke M.C. Brown and O. Grümberg. Characterizing kripke structures in temporal logic. In G. Levi H. Erhig, R. Kowalski and U. Montanari, editors, *TAPSOFT'87, vol. I*, number 249 in LNCS, pages 256–270. Springer-Verlag, 1987.

[40] K.L. McMillan. *Symbolic Model Checking*. Kluwer, 1992.

[41] John O. Moody and Panos J. Antsaklis. *Supervisory Control of Discrete Event Systems using Petri Nets*. Kluwer Academic Publishers, 1998.

[42] J.S. Ostroff. *Temporal Logic for Real-Time Systems*. Research Studies Press/ Wiley, Taunton, UK, 1989.

[43] D. L. Parnas. Use of abstract interfaces in the development of software for embedded computer systems. NRL Report 8047, Naval Research Laboratory, 1977.

[44] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, December:1053–1058, December 1972.

[45] David Lorge Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–66, March 1985.

[46] 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.

[47] 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(6):573–586, 1999.

[48] M.H. de Queiroz and J.E.R. Cury. Modular supervisory control of large scale discrete event systems. In *Proceedings of WODES 2000*, pages 103–110, Ghent, Belgium, Aug 2000.

[49] C. Costas R. Alur and D. Dill. Model-checking for real-time systems. In *Proc. of 5th IEEE Symp. Logic in Computer Science*, pages 414–425, 1990.

[50] P. Ramadge and W. Murray Wonham. Supervisory control of a class of discrete-event processes. *SIAM J. Control Optim*, 25(1):206–230, 1987.

[51] K. Rudie. Software for the control of discrete-event systems: A complexity study. Master's thesis, Dept. of Electrical and Computer Engineering, University of Toronto, Toronto, Ont, 1988.

[52] Karen Rudie and Jan C. Willems. The computational complexity of decentralized discrete-event control problems. *IEEE Trans. Automatic Control*, 440(7):1313–1319, 1995.

[53] Karen Rudie and W. Murray Wonham. Think globally, act locally: decentralized supervisory control. *IEEE Trans. on Automatic Control*, 37(11):1692–1708, Nov 1992. Reprinted in F.A. Sadjadi (Ed.), Selected Papers on Sensor and Data Fusion, 1996; ISBN 0-8194-2265-7.

[54] Gang Shen and Peter E. Caines. Hierarchically accelerated dynamic programming for finite-state machines. *IEEE Trans. Automatic Control*, 47(2):271–283, 2002.

[55] 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(9):1490–1496, 2001.

[56] J. Wakerley. *Digital Design Principles*. Prentice-Hall, Inc., 1990.

[57] 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.

[58] K.C. Wong. *Discrete-Event Control Architecture: An Algebraic Approach*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, Toronto, Ont, 1994.

[59] K.C. Wong and J.H. van Schuppen. Decentralized supervisory control of discrete event systems with communication. In *Proc. of WODES 1996*, pages 284–289, Edinburgh, UK, Aug 1996.

[60] W. Murray Wonham. *Notes on Control of Discrete-Event Systems*. Department of Electrical and Computer Engineering, University of Toronto, 2002. Notes and CTCT software can be downloaded at http://odin.control.toronto.edu/DES/.

[61] W. Murray Wonham and P. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM J. Control Optim*, 25(3):637–659, 1987.

[62] Weimin Wu, Hongye Su, Jian Chu, and Haifeng Zhai. Hierarchical control of DES based on colored petri nets. In *Proc. of IEEE Systems, Man, and Cybernetics*, volume 3, pages 1571–1576, 2001.

[63] T. Yoo and S. Lafortune. A general architecture for decentralized supervisory control of discrete-event systems. In *Proc. of WODES 2000*, pages 111–118, Ghent, Belgium, Aug 2000.

[64] Bernard Zeigler. *Object-Oriented Simulation with Hierarchical, Modular Models: Intelligent Agents and Endomorphic Systems.* Academic Press Inc, 1990.

[65] 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.

[66] Z.H. Zhang and W. Murray Wonham. STCT: an efficient algorithm for supervisory control design. In *Proc. of SCODES 2001*, INRIA, Paris, July 2001.

[67] H. Zhong and W. Murray Wonham. On the consistency of hierarchical supervision in discrete-event systems. *IEEE Trans. on Automatic Control*, 35(10):1125–1134, Oct 1990.

[68] 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(2):287–300, 1998.

[69] MengChu Zhou and Frank DiCesare. *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems.* Kluwer Academic Publishers, 1993.