Centre de recherche
informatique de Montréal

CRIM

550, rue Sherbrooke Ouest, bureau 100
Montréal (Québec) H3A 1B9
Téléphone : (514) 840-1234
Télécopieur : (514) 840-1244
http://www.crim.ca

CRIM - Documentation/Communications

Supervisory Control with Divergence Freedom in Two Safety Models

Première version

CRIM-00/01-02

Hessham Hallal
Radu Negulescu
Alexandre Petrenko

Version of September, 2000

Collection scientifique et technique

ISBN 2-89522-001-8

Pour tout renseignement, communiquer avec:
CRIM Centre de documentation
Centre de recherche informatique de Montréal (CRIM)
550, rue Sherbrooke Ouest, bureau 100
Montréal (Québec) H3A 1B9

Téléphone : (514) 840-1234
Télécopieur : (514) 840-1244

Bibliothèque nationale du Québec
Bibliothèque nationale du Canada
ISBN 2-98522-001-8

Abstract

A method is presented to solve supervisory control problems of discrete event systems and produce divergence-free controllers, which guarantee to avoid unbounded internal communications. The method is described in terms of process spaces, a formalism to model concurrent systems, and generalized to various safety models. Semantics mappings are described, which relate process spaces to I/O automata and other labeled transition systems. Divergence-free solutions are obtained based on a strong notion of divergence freedom. Our method, which produces the solution for a supervisory control problem, can also detect the absence of such solution and helps to obtain synthesizable specifications for the solutions as well. Finally, a case study is considered where the supervisory control formulation is used to derive the specification of a divergence-free protocol converter to interface two mismatched communication protocols.

Table of Contents

List of Figures

Chapter 1

Introduction

Modeling concurrent systems is one basic step towards automated design and verification of such systems. Several formalisms have been introduced in the literature to serve these purposes; I/O automata [9, 10, 11] and process spaces [14, 15, 16] are among these models. The question arises on how these models relate one to another and whether it is possible to transfer methods and results from one model to another.

In this report, we describe the two models mentioned above, relate them by means of semantic mappings, and use the mappings to transfer results from one model to the other.

As an application of practical importance following from the relationship we derive, we present a method to solve supervisory control problems, also known as asynchronous equations. This means to derive a specification for a missing part of a system from the overall specification of that system and the known part of its implementation, all entities being modeled in process spaces. Then we apply the mappings to automate the process of solving, in I/O automata, asynchronous equation problems formulated and solved in terms of I/O finite state machines in [19].

Finally, we apply our method to solve asynchronous equations in deriving the specifications of protocol converters to interface heterogeneous network systems.

1.1 Motivation

The motivation for our work is to transfer methodologies between the I/O automata and the process spaces formalisms, specifically, solving asynchronous equations in I/O automata by means of methods and tools from process spaces using a BDD based tool from process spaces. Through an example, we illustrate how our mapping can be used to serve this purpose.

Asynchronous equations are believed to have many applications; e.g., design-reuse of hardware and software blocks.

As an application, we rework the problem of designing protocol converters in the framework of asynchronous equations [1, 8, 17, 23]. We use our method to derive the specification of a divergence-free protocol converter, which interfaces two mismatched protocols: an Alternating bit protocol and a Non-sequenced protocol. In addition, we show that the solution obtained can be synthesized into an asynchronous circuit using available tools. Our interest in asynchronous implementations stems from the fact that they overpass a major design constraint, the global clocking. Hence they can offer some performance advantages and alternative solutions to design problems, e.g. small area overhead and low power. A divergence-free implementation avoids switching activities due to clock transitions in a circuit.

1.2 Previous work

The problem of asynchronous equations has been discussed from several viewpoints: model matching [4], sub-module construction [5], design equation [13], supervisory control [18], I/O FSMs [19] to name just a few. Drissi and Bochmann have devised an algorithm to solve asynchronous equations in I/O automata in [5]. However, our study yields, in addition to general solutions for this type of equations, divergence-free solutions for asynchronous equations; i.e., solutions that avoid unbounded internal interactions. In [18] a notion of divergence based on unbounded interactions, involving

internal actions, was used. However, in our case, we seek divergence freedom in terms of sets of processes rather than individual ones. In [13], divergence-free solutions that avoid unbounded internal communications are often possible to obtain. However, [13] uses divergence-freedom as a precondition for the validity of the operations; whereas we are interested in divergence-freedom as a criterion for the correctness of the obtained solution. Divergence-free solutions to the asynchronous equations have been proposed in [19], but only in terms of I/O FSMs, where input and output events alternate. We are interested in extending the method of [19] to more general models that capture safety properties.

1.3 Organization of the Report

The presentation of this work proceeds as follows: Chapter 2 presents an overview of the two models used: I/O automata and process spaces and describes the mappings from I/O automata to process spaces and analyses the properties of these mappings. In Chapter 3, we present our method to obtain solutions and divergence-free solutions for asynchronous equations. An example is used to illustrate the method and the use of the mappings. In addition, we describe the mapping of process spaces into I/O automata. Chapter 4 presents a case study on the application of asynchronous equations in the area of protocol converters. Finally, the conclusions of our work and the possible future steps are presented in Chapter 5.

Chapter 2

Overview of the Models

2.1 Input/Output Automata

Basic Definitions

Input/output automata were introduced by Lynch and Tuttle in 1987. This model is based on non-deterministic automata. The I/O automaton model best suits representing asynchronous systems; however, various applications to modeling synchronous systems as well were proposed [9]. A basic property of the model is that an automaton differentiates between actions that are controlled by the environment and those under the control of the automaton itself. Each automaton has an action set *acts* whose elements are classified either as inputs: *In*, outputs: *Out*, or internals: *Int*. The inputs are generated by the environment, they are always enabled, and the automaton cannot block them. The outputs and internal actions, on the other hand, are controlled by the automaton itself [9, 10, 11]. However, input enabling is restricted by the correctness assumption that when the environment behaves correctly, the I/O automaton behaves correctly. In other words, if the environment behaves improperly, e.g. issues inputs continuously without letting the automaton respond, the automaton can behave arbitrarily or issue an error message [9,10].

Definition 2.1 [9, 10, 11] An *input output automaton* (I/O automaton) is a tuple $\alpha = (S, I, \Sigma, \lambda, \delta)$ where

1. $S(\alpha)$ is a set of states.
2. $I(\alpha)$ is a nonempty set of initial states
3. $\Sigma(\alpha)$ is an *action signature*, which partitions the actions *acts* of an automaton into three disjoint sets: $In(\alpha)$, $Out(\alpha)$, and $Int(\alpha)$. The $Out(\alpha)$, and $Int(\alpha)$ are called the locally controlled actions. The $In(\alpha)$ and $Out(\alpha)$ are called the *external* actions.
4. $\lambda(\alpha) \subseteq S(\alpha) \times \Sigma(\alpha) \times S(\alpha)$ is a *transition relation*. The elements of $\lambda(\alpha)$ are triples of the form (s, π, s'), where s and s' belong to $S(\alpha)$ and $\pi \in \Sigma(\alpha)$.
5. $\delta(\alpha)$ is an *equivalence relation*, which partitions the set of locally controlled actions of α into a countable set of equivalence classes. Each class is meant to represent the set of locally controlled actions of some system component.

The equivalence relation $\delta(\alpha)$ is used to represent fairness properties of the I/O automaton. For more details on how to relate the notion of fairness in I/O automata to the equivalence relation of each automaton, we refer the readers to [9] and [10].

Definition 2.2 An *execution* of an I/O automaton is a sequence, $s_0 \pi_1 s_1 \pi_2 s_2 \pi_3 \ldots \pi_n s_n$ of alternating states and actions such that each triple $(s_i, \pi_{i+1}, s_{i+1})$ is an element of $\lambda(\alpha)$, and $s_0 \in I(\alpha)$ [10]. The set of all executions of an automaton is denoted $executions(\alpha)$.
A state s of an I/O automaton α is *reachable* when s is the final state of a finite execution e of α [9].

Definition 2.3 A *trace* of an I/O automaton is a sequence $\pi_1 \pi_2 \pi_3 \ldots \pi_n$ of actions. A trace is obtained from an execution by projecting out (eliminating) the states. We refer to the set of traces of an automaton as $traces(\alpha)$. An *external trace* of an automaton is a sequence of external actions. It is obtained by dropping all the internal actions from a trace of an automaton. The set of external traces of an automaton α is referred to as $etraces(\alpha)$.

We use the example of a candy machine and its customer to illustrate the use of I/O automata to model systems. This is a popular example in the literature, which can give an idea of the main properties of the model [10].

Our candy machine *CM* has the following action signature:

In (*CM*) = {*P*1, *P*2} (where the action labels *P*1 and *P*2 are shorthands for *Push*1 and *Push*2, respectively) .

Out (*CM*) = {*S, H, A*} (shorthands for *Skybar*, *Heathbar*, and *Almondjoy*).

Int (*CM*) = ∅ (i.e., no internal actions).

The state of the machine expresses one variable "*button-pushed*", which is affected by the input actions *P*1 and *P*2. Initially, this variable is set to 0. When *P*1 arrives, the variable is set to 1, and to 2 when *P*2 arrives. The machine *CM* dispenses three types of candy bars: *S, H,* and *A*. When *P*1 arrives, the machine gives *S* and returns to state 0. Otherwise, when *P*2 is pushed it issues *H* or *A* and returns to state 0. The choice between *H* and *A* is non-deterministic. Since the I/O automaton *CM* is input enabled, it can not prevent the input events from occurring at any time.

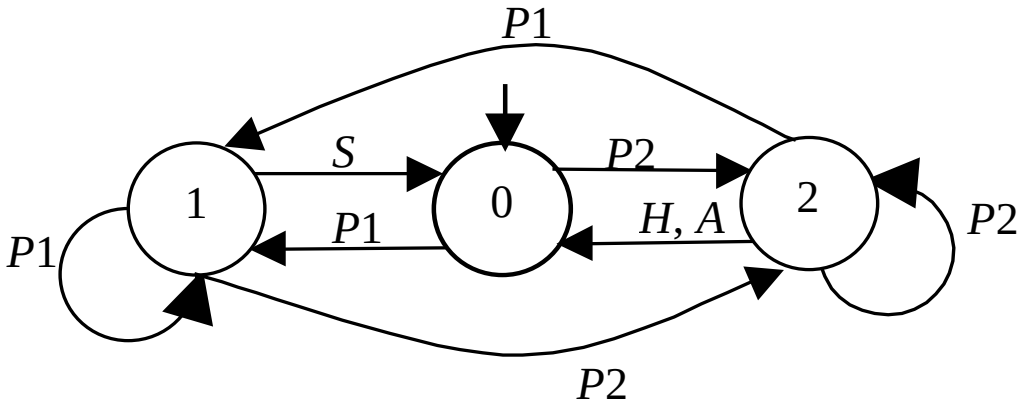


Figure 2.2: The I/O automaton for Candy Machine *CM*.

Figure 2.1 shows the I/O automaton of *CM*. The initial state is indicated by the incoming arrow that points to it. We consider that the three output actions *S, H,* and *A* belong to the same equivalence class.

The sequence (0, *P*1, 1, *P*2, 2, *A*, 0) represents an execution of *CM* from which we can extract the trace (*P*1, *P*2, *A*). This trace is also an external trace since it contains only inputs and outputs.

The customer of the candy machine can also be modeled by an I/O automaton. This I/O automaton, *CUST*, has the following action signature:

In(*CUST*) = {*S*, *H*, *A*} .

Out(*CUST*) = {*P*1, *P*2}.

Int(*CUST*) = {*BS*} (where BS stands for *Become-Satiated*).

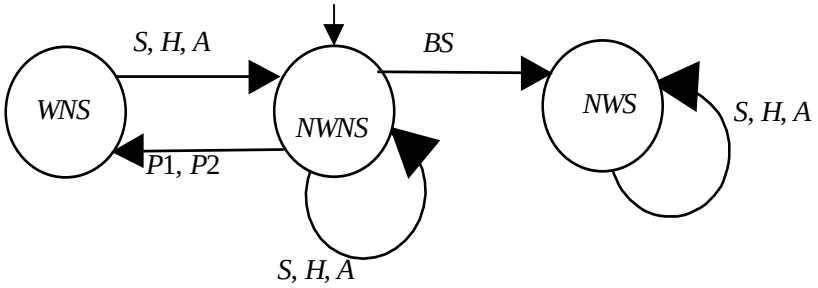


Figure 2.2: I/O automaton for the Customer *CUST*.

The state of the customer expresses the value of two variables, "*waiting*" and "*satiated*". Initially, *CUST* is in the *Not Waiting Not Satiated* (*NWNS*) state. The customer presses either *P*1 or *P*2 and waits for a candy bar at the state (*WNS*). When the customer receives the bar, the customer returns to the (*NWNS*) state. However, the customer chooses non-deterministically to stop ordering bars after a random number of transitions by executing *BS* and entering the *Not Waiting*-S*atiated* (*NWS*) state. *CUST* is represented in Figure 2.2. In addition, we consider all the locally controlled actions of *CUST* (*P*1, *P*2, and *BS*) to belong to the same equivalence class.

In the following, we will summarize some of the relations and manipulations applicable to I/O automata.

Compatibility

Two I/O automata α_1 and α_2 are said to be *compatible* (i.e., can be composed) if the action signatures $\Sigma(\alpha_1)$ and $\Sigma(\alpha_2)$ are compatible themselves [9, 10]. The action signatures of two I/O automata are considered *compatible* when the following conditions are met:

- $Out(\alpha_1) \cap Out(\alpha_2) = \varnothing$ and
- $Int(\alpha_1) \cap acts(\alpha_2) = Int(\alpha_2) \cap acts(\alpha_1) = \varnothing$.

The compatibility relation between automata can be generalized to any countable collection of automata $\{\alpha_i\}_{i \in I}$. However, one more condition needs to be satisfied for compatibility in the case of infinite collections of automata [9]. This condition requires that no action is contained in infinitely many sets $acts(\alpha_i)$.

The two automata *CM* and *CUST* of our example are compatible since
$Out(CM) \cap Out(CUST) = \{S, H, A\} \cap \{P1, P2\} = \varnothing$,
$Int(CM) \cap acts(CUST) = \varnothing \cap \{S, H, A, P1, P2, BS\} = \varnothing$, and
$Int(CUST) \cap acts(CM) = \{BS\} \cap \{S, H, A, P1, P2\} = \varnothing$.

Equivalence

The notion of *equivalence* for I/O automata is based on trace equivalence. Two I/O automata are *equivalent* if they generate equal sets of external traces, *etraces*.

The two automata of our example, *CM* and *CUST*, do not have equal sets of inputs and outputs. In addition, Figure 2.2.1 and Figure 2.2 show that *CM* and *CUST* do not generate equal sets of external traces. For example, (*P*1, *P*1) is an external trace of *CM* but not *CUST*. Similarly, (*S*, *S*) is an external trace of *CUST* and not *CM*. Hence, they are not equivalent.

Composition

The *composition* of a countable collection $\{\alpha_i\}_{i\in I}$ of compatible automata is an automaton α, where $\alpha = \Pi_{i\in I}\alpha_i$ is defined by the following [9, 10]:

1. A set of states $S(\alpha) = \Pi_{i\in I} S(\alpha_i)$.
2. A nonempty set of initial states $I(\alpha) = \Pi_{i\in I} I(\alpha_i)$.
3. An *action signature* $\Sigma(\alpha)$ formed by the composition of the individual action signatures of the automata $\Sigma(\alpha_i)$ in the following manner:

- $In(\alpha) = \cup_{i\in I} In(\alpha_i) - \cup_{i\in I} Out(\alpha_i)$.
- $Out(\alpha) = \cup_{i\in I} Out(\alpha_i)$.
- $Int(\alpha) = \cup_{i\in I} Int(\alpha_i)$.

4. A *transition relation* $\lambda(\alpha)$ of triples $(s_1, \pi, s_2) \in S(\alpha) \times acts(\alpha_i) \times S(\alpha)$ such that for all $i\in I$, if $\pi\in acts(\alpha_i)$ then $(s_1[i], \pi, s_2[i]) \in \lambda(\alpha_i)$, and if $\pi\notin acts(\alpha_i)$ then $s_1[i]= s_2[i]$.
5. A *relation* $\delta(\alpha) = \cup_{i\in I} \delta(\alpha_i)$.

The partition in the relation $\delta(\alpha)$ is the union of the partitions in the individual automata relations. Therefore, each equivalence class of each individual automaton becomes an equivalence class of the composition. In this definition, partition union coincides with set union of partitions because composition in I/O automata requires compatibility of the action signatures of the automata to compose, i.e. disjoint sets of locally controlled actions of all automata in the composition. As a result, the composition includes a representation of each individual automaton, based on its locally controlled actions, so that the fairness of the composition becomes easily expressed in terms of its components.

When I is the finite set $\{1,\ldots, n\}$, the composition is denoted [10]: $\alpha = \alpha_1\cdot\alpha_2\cdot\ldots\cdot\alpha_n$.

For example, the composition of the two automata *CM* and *CUST*, written *CM·CUST*, could be represented by an automaton *CMCUST*, which has the action signature:

$In(CMCUST) = \{P1, P2\}\cup\{S, H, A\} - \{S, H, A\}\cup\{P1, P2\} = \varnothing$.

$Out(CMCUST) = \{S, H, A\}\cup\{P1, P2\} = \{S, H, A, P1, P2\}$.

$Int(CMCUST) = \varnothing \cup\{BS\} = \{BS\}$.

Figure 2.3 shows the automaton *CMCUST*. The states of the *CMCUST* correspond to pairs consisting of the states of each component. The initial state 0*NWNS* corresponds to the pair (0, *NWNS*). Executions of the composition induce executions of the individual automata. For example, the following execution of *CMCUST* (0*NWNS*, *P*1, 1*WNS*, *P*2, 2*WNS*, *A*, 0*NWNS*, *BS*, 0*NWS*) induces (0, *P*1, 1, *P*2, 2, *A*, 0) of the automaton *CM*. In the new automaton, the equivalence relation puts the locally controlled actions of each component automaton in a different class. We have, therefore, two classes: {*S*, *H*, *A*} and {*P*1, *P*2, *BS*}.

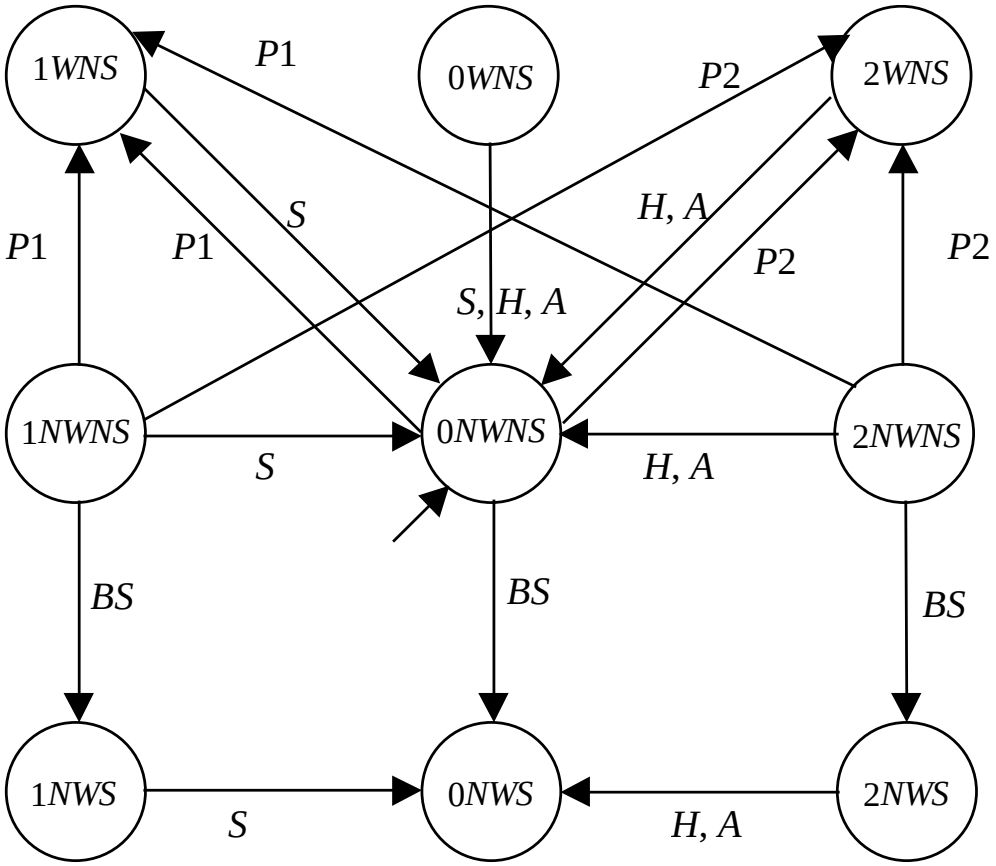


Figure 2.3: The I/O Automaton *CMCUST*.

In *CMCUST*, states 0*NWNS*, 0*NWS*, 1*WNS*, and 2*WNS* are reachable states since they all may appear at the end of at least one execution of *CMCUST*. On the other hand, states 0*WNS*, 1*NWNS*, 1*NWS*, 2*NWNS*, and 2*NWS* are all unreachable. It is clear that these states do not appear at the end of any finite execution of *CMCUST*.

In general, the interest is in states that the automaton can reach through its executions. Therefore, we will not consider the unreachable states of an automaton. The diagram of the automaton *CMCUST*, after dropping the unreachable states, is shown in Figure 2.4.

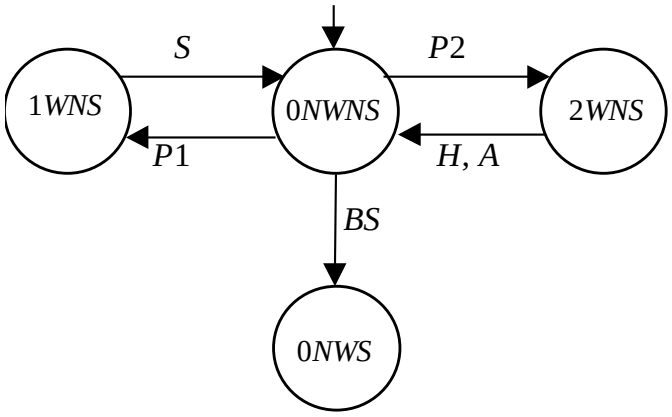


Figure 2.4: The reachable states of the I/O automaton *CMCUST*.

Fairness and Quiescence

The *fairness* of an I/O automaton α is expressed in terms of the equivalence classes generated by the relation $\delta(\alpha)$. It is based on the fairness of the executions of the automaton. An execution *e* of an automaton α is *fair* if, for each class *C* of $\delta(\alpha)$ [9, 10]:

1. if *e* is finite, no action of *C* is enabled (can be executed) at the final state of *e*.
2. if *e* is infinite, either actions of *C* appear infinitely often in *e*, or states at which no action of *C* is enabled appear infinitely often in *e*.

Based on this definition, we can define *fair traces, ftraces*(α), of an automaton. They are the sequences of actions extracted from fair executions of the automaton. We shall refer to the sequences of external actions in fair executions of an I/O automaton as *fair external traces* or *fetraces*(α).

Throughout the report, we confine our study of fairness to finite executions of automata; i.e.:
No locally controlled action is enabled at the final state of any fair execution of an I/O automaton.

Definition 2.4 A state *s* of an automaton is called *quiescent* if the only actions that are enabled in *s* are input actions [11]. A trace *t* of an automaton is called *quiescent* if there exists a finite execution *e* of the automaton, out of which *t* can be extracted, where the last state is quiescent.

Based on this definition and the condition we set for fairness of finite traces and finite external traces, we can conclude that:
A finite execution e of an I/O automaton is fair if the final state in e is quiescent.

Consider our example automaton *CM* in Figure 2.2.1. The execution (0,*P*1,1, *P*2, 2) is not fair for the *CM* automaton. The reason is that at state 2, either *H* or *A* is enabled but not executed. However, the execution (*NWNS*, *P*1, *WNS*, *A*, *NWNS*, *BS*, *NWS*) is fair for the *CUST* automaton, shown in Figure 2.2, since no output action (*P*1, *P*2) or internal action (*BS*) is enabled at the *NWS* state. An example of a fair trace of *CUST*, we consider (*P*1, *A*, *BS*). Notice how a trace is obtained by eliminating the states in an execution. On the other hand, (*P*1, *A*) is an example of *fetraces*(*CUST*).

Refinement

Refinement is a relation between automata that allows for one automaton α_1 to fully replace a second automaton α_2. This replacement can be viewed from two perspectives: function and fairness.

In the first case, refinement imposes that all functions executed, with respect to the environment, by α_1 be functions of α_2. This, of course, means that the external traces generated by α_1 form a subset of the traces generated by α_2. We refer to this aspect of

refinement as *implementation* [10] or *external trace preorder* [21], which could be formalized in the following way:

An automaton α_1 *implements* an automaton α_2 if [9, 21]:

- They have the same sets of inputs and outputs.
- The finite external traces of α_1 are a subset of the finite external traces of α_2: $etraces(\alpha_1) \subseteq etraces(\alpha_2)$. Notice, however, that this inclusion does not mean that α_1 does less than α_2 in terms of the outputs only. This is because both automata are input enabled and they have the same set of inputs (first condition).
- The set $etraces(\alpha_1)$ is not empty. This eliminates the case of trivial implementation between automata.

Based on this definition of implementation between I/O automata, we conclude the following*:*

If α_1 *implements* α_2 *and* α_2 *implements* α_1*, then* α_1 *and* α_2 *are equivalent* (Section 2.1.3).

In the second case, if we take fairness as the criterion for refinement, we require that α_1 also satisfies every fairness condition satisfied by α_2. This aspect of refinement is referred to as *satisfaction* [10] or *fair preorder* [21], and it can be described as follows:

An automaton α_1 *satisfies* an automaton α_2, if [9, 10]:

- They have the same sets of inputs and outputs.
- The fair external traces of α_1 are a subset of the fair external traces of α_2: $fetraces(\alpha_1) \subseteq fetraces(\alpha_2)$.
- The set $fetraces(\alpha_1)$ is not empty. This eliminates the case of trivial satisfaction between automata.

In the following example, we show a case where implementation between automata does not imply satisfaction. We use, here, the example of a combination of candy machine and customer, *CCUST*. We will check for implementation and satisfaction between *CMCUST* (Figure 2.4) and *CCUST*. The new automaton, *CCUST*, is shown in Figure 2.5. It has the following action signature:

In (*CCUST*) =∅.

Out (*CCUST*) ={*S*, *H*, *A*, *P*1, *P*2}.

Int (*CCUST*) =∅.

Here *S*, *H*, *A*, *P*1, and *P*2 represent the same *Skybar*, *Heathbar*, *Almondjoy*, *Push*1, and *Push*2 actions from the previous examples. The new automaton is similar to *CMCUST* except that no satiation state is reached. The system keeps running infinitely. The equivalence relation of the automaton *CCUST* puts its actions in two different classes: {*S*, *H*, *A*} and {*P*1, *P*2}.

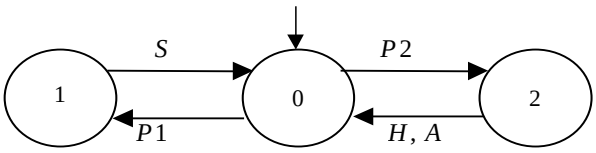


Figure 2.5: The I/O automaton *CCUST*.

The automaton *CMCUST* implements *CCUST*: The two automata have equal sets of inputs and outputs. In addition, Figures 2.4 and 2.5 show that
etraces(*CMCUST*) ⊆ *etraces*(*CCUST*).

In fact, the two automata generate equal sets of finite external traces. This means that the implementation is verified in both directions since the inclusion relation is verified in both directions.

At the same time, *CMCUST* does not satisfy *CCUST*. *CMCUST* generates traces, which are fair. In fact, only the traces of *CMCUST* that end with *BS* are fair traces. Definition 4 states that we can obtain *fetraces* of *CMCUST* out of these fair traces by eliminating *BS*. However, *CCUST* has an empty set of fair external traces since all actions are outputs and they are enabled in all states. In addition, we can prove that *CCUST* does not satisfy *CMCUST*. The reason is that *CCUST* has an empty set of fair external traces, and trivial satisfaction is not allowed in automata.

Hiding

Hiding is performed on output actions of I/O automata [11]. This operation reclassifies output actions as internal actions, and hides them from external observation. As a result, the hidden actions are not included in external traces of an automaton. Hiding is defined on the action signature of an I/O automaton as well as on the automaton itself [11].

For an I/O automaton α, a signature Σ (partitioned into *In*, *Out*, *and Int*), and a subset $O \subseteq Out$, a new signature $hide_O(\Sigma)$ is defined, and a new I/O automaton α' is obtained. The new signature consists of the disjoint sets: *In*, *Out* - *O*, *and Int* $\cup$ *O*. As for α', it is obtained by replacing Σ with $hide_O(\Sigma)$.

Process Spaces

Basic definitions

Process spaces were introduced by R. Negulescu [14, 15, 16]. A *process p* represents a contract between a device and its environment over a set of executions [14]. Executions can be sequences of actions, functions of time, etc. [14]. In a process, both the device and the environment guarantee that only executions from specified sets of allowable executions could occur. Note that executions here do not refer to Definition 2.2.

In the following, we will list some basic definitions of process spaces, which are independent of the level of detail of executions. In the next section, we use FIREMAPS, a computer tool to handle processes, to carry out examples and illustrations. Therefore, some of the examples are delayed until section 2.3 to avoid redundancy.

Definition 2.5 A *process p* over a set of executions *E* is a pair (*X*, *Y*) of subsets of *E* such that

$$X \cup Y = E$$

where X is called the set of *accessible* executions of p, and Y is called the set of *acceptable* executions of p. The set E is arbitrary; its elements can be sequences of actions, functions of time, etc [14]. The accessible and acceptable sets are denoted by **as**(p) and **at**(p), respectively. The *acceptable* executions represent what the environment provides, and the *accessible* executions represent what the device offers in reply.
The process *space* of E is the set of all possible processes over E; this set is denoted by S_E.

Since the intersection of their complements is empty, the accessible and acceptable sets of a process induce a tri-partition of the executions set E.

1-The *reject* executions of a process p are the executions considered violations (illegal behavior) by the environment. The set of such executions is denoted by **r**(p), and it is the complement of the set of acceptable executions. Therefore, $\mathbf{r}(p) = \overline{\mathbf{at}(p)}$.

2-The *goal* executions of a process p are the desired executions that both the environment and the device should produce. The set of such executions is denoted by **g**(p), and it is the intersection of the acceptable and accessible sets of p.

3-The *escape* executions of a process p are the executions representing violations committed by the device. The set of such executions is denoted by **e**(p), and it is the complement of the set of accessible executions, or $\mathbf{e}(p) = \overline{\mathbf{as}(p)}$.

Note that $\mathbf{as}(p) = \mathbf{r}(p) \cup \mathbf{g}(p)$, and $\mathbf{at}(p) = \mathbf{e}(p) \cup \mathbf{g}(p)$.

Therefore, a process can be represented by **r**(p), **g**(p), and, **e**(p). And, we can say that two processes p and q are *equal*, denoted $p = q$, if and only if they have equal reject, goal, and escape sets of executions. Also note that the three sets **r**(p), **g**(p), and, **e**(p) are disjoint and cover E; i.e., an execution can be either reject, escape, or goal, but not two of these at the same time.

Figure 2.6 shows the execution set of a process and how it can be partitioned into accessible and acceptable subsets or, alternatively, into goal, reject, and escape subsets [14].

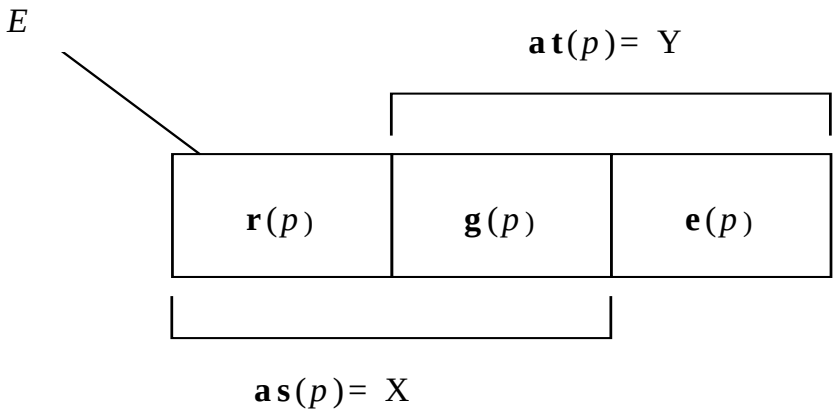


Figure 2.6: Execution set of a process *p* and its partitions.

Safety and Finalization Processes

Safety represents the guarantee that nothing bad happens or the avoidance of illegal incidents [14] in general; e.g., illegal inputs. *Finalization*, on the other hand, ensures that something good happens and shows avoidance of illegal stopping. The safety and finalization properties can be expressed by means of sets of finite executions. In process spaces, safety and finalization are expressed as processes (pairs of acceptable and accessible sets of executions) over an execution set E equal to the set U^* of finite words where U is a universal set of actions [14].

Although they are expressed as processes, safety and finalization are both attached to a concurrent system to model that system. The *safety process*, denoted σ, deals with partial executions and records the occurrence of illegal inputs and outputs. On the other hand, the *finalization process*, denoted φ, is constructed for the total finite executions. In fact, it considers every sequence of actions a total execution and records the violations it

contains [14]. These violations include, in addition to illegal inputs and outputs, illegal stopping.

As an example, we show how the behavior of a logic AND gate can be described in terms of safety and finalization. Figure 2.7 shows a two-input AND gate and its truth table with *a* and *b* being the inputs and *c* the output of the gate. The safety information is conveyed by the process σ(AND), and the finalization information is expressed by φ(AND).

We consider the execution *abc* which means that after the two inputs of the gate become high, the output is also set to high. This execution is a goal for both safety and finalization processes of the AND gate. On the other hand, consider the execution *ab*, which means that the two inputs of the gate are high. From the safety point of view, this is a legal behavior since it does not include any illegal input or output. However, this execution is not legal from the finalization point of view. The gate does not have right to forbid *c* from rising after the two inputs are high, i.e. the gate commits a violation if it stops after receiving the input pulses.

a	*b*	*c*
0	0	0
0	1	0
1	0	0
1	1	1

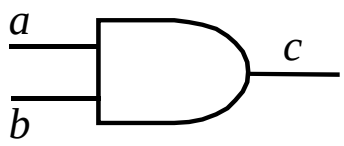


Figure 2.7: Logic AND gate: truth table (left) and symbol (right).

Liveness and Progress

Two other properties of interest in discrete event systems are liveness and progress [14]. Roughly speaking, *liveness* requires that allowed events not be postponed forever. *Progress*, on the other hand, requires that expected events not be postponed for an unbounded time.

Although liveness and progress provide more detailed modeling than finalization [14], they require the manipulation of sets of infinite executions, which is computationally difficult. As in the case of I/O automata, our study of process spaces will be confined to finite executions. Therefore, we will use only the safety and finalization processes throughout the report.

One of the main characteristics of processes is the absence of the distinction between input, output, and internal actions [14]. In the general process space formalism, there are no notions of states or variables involved in a process or its executions. These notions are used, however, in particular instances of process spaces to capture correctness concerns at various levels of detail. *Safety*, *finalization*, *liveness*, and *progress* are examples of such instances.

Process Automata

When the execution sets of a (safety or finalization) process p are regular languages, p can be represented as a finite automaton [14]. Finite automata used to represent safety or finalization processes are called *process automata*. These finite automata represent processes defined over a set of actions U; however, they only use finite subsets of U as alphabets. The effect of actions from outside the alphabet set is simply ignored, i.e., they lead to self loops in the process automaton.

Definition 2.6 A *process automaton* is a tuple $Pr = (A, Q, I, Q_{as}, Q_{at}, Ed)$ [14], where

1. A is a finite set of actions, called the alphabet of Pr.
2. Q is a finite set of states.
3. $I \subseteq Q$ is a set of initial states.
4. Q_{as} and Q_{at} whose elements are called accessible and acceptable states, respectively, are subsets of Q such that $Q = Q_{as} \cup Q_{at}$.
5. $Ed \subseteq Q \times A \times Q$ is a set of transitions.

In this report, we work with deterministic process automata. These process automata have exactly one initial state, and, for each state $q \in Q$ and action $a \in A$, there exists exactly one state $q' \in Q$ so that the transition (q, a, q') is in Ed [14].

Another representation of a process automaton considers using the notations Q_r, Q_g, and Q_e to represent states. Based on this partition, a state q is qualified as a reject state, a goal state, or an escape state depending on the executions that lead to it. In addition, a correspondence between the two representations of states in a process automaton could be formulated as follows:
In a process automaton, the set of states $Q = Q_{as} \cup Q_{at}$ can be partitioned into Q_r, Q_g, and Q_e, where
$Q_{as} = Q_r \cup Q_g$ and
$Q_{at} = Q_e \cup Q_g$.

Notice that, as in the case of executions, the sets Q_r, Q_g, and Q_e are disjoint.

Furthermore, we can state the following condition for two states in a process automaton to be equivalent:

Two states, $q1$ and $q2$, are equivalent if they enable equal future sets of accessible and acceptable traces. These sets of traces can be seen as processes (pairs of acceptable and accessible executions), and are called processes generated by process automata. Formally, we have:

Definition 2.7 With each process automaton $Pr = (A, Q, I, Q_{as}, Q_{at}, Ed)$ a pair $p(Pr) = (X, Y)$ of subsets of U^* is associated as follows. For every word $w \in U^*$,

1. $w \in X$ if and only if there exists a path that starts at the initial state in Pr and that spells w and ends in Q_{as}.
2. $w \in Y$ if and only if there are no paths that start at the initial state in Pr and that spell w and end in $Q \setminus Q_{at}$.

Based on Definition 2.7, $q1$ and $q2$ are equivalent if $p(Pr1 = (A, Q, \{q1\}, Q_{as}, Q_{at}, Ed))$ and $p(\text{Pr}2 = (A, Q, \{q2\}, Q_{as}, Q_{at}, Ed))$ are equal, where $p(Pr)$ is the process generated by the process automaton Pr. Note that $q1$ is the initial state of $Pr1$, and $q2$ is the initial state of $Pr2$.

We use process automata to illustrate how safety and finalization processes are used to model systems and how they differ from each other. We model an inertial logic AND gate, in which an output transition is canceled upon retraction of the input excitation. Figure 2.8 and Figure 2.9 show the *safety* and *finalization process automata* of the AND gate, respectively. An initial state is indicated by an incoming arrow that points to it. In addition, states marked with g or e denote goal states or escape states, respectively. On the other hand, states reachable by violations due to illegal inputs and outputs are considered as trap states marked E (permanent escape) and R (permanent reject), respectively. The automaton cannot resume normal executions when it reaches any of the trap states. For example, *abbc* is a reject execution because it leads to the state marked R.

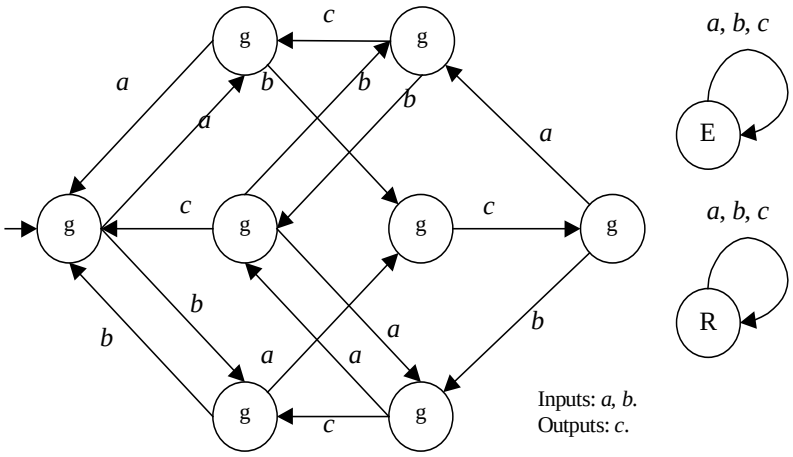


Figure 2.8: The safety process automaton for the inertial AND gate. (Missing inputs lead to the state indicated R. Missing outputs lead to the state indicated E)

Notice the difference between the two process automata. In Figure 2.8, the execution *ab* is a goal execution since it includes no illegal events. This same execution is an escape execution for the finalization process in Figure 2.9 since it does not represent a complete

behavior of the AND gate (after both inputs *a* and *b* rise the gate should issue the output *c*).

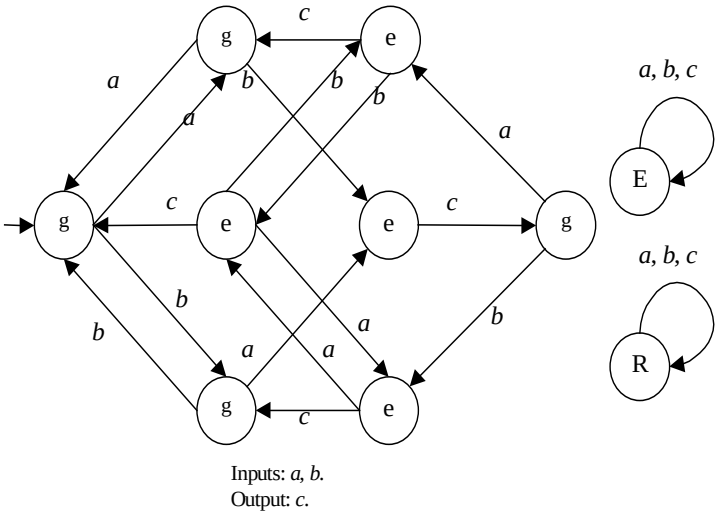


Figure 2.9: The finalization process automaton for the inertial AND gate.(Missing inputs lead to the state indicated R. Missing outputs lead to the state indicated E)

Composition

Joint behavior (parallel composition) is expressed in process spaces by the *product* operation [14]. The *product* of two processes p and q is a process $p \times q$ such that

$\mathbf{as}(p \times q) = \mathbf{as}(p) \cap \mathbf{as}(q)$

$\mathbf{at}(p \times q) = (\mathbf{at}(p) \cap \mathbf{at}(q)) \cup \overline{(\mathbf{as}(p) \cap \mathbf{as}(q))}$,

or equivalently,

$\mathbf{r}(p \times q) = (\mathbf{r}(p) \cup \mathbf{r}(q)) \cap \overline{\mathbf{e}(p)} \cap \overline{\mathbf{e}(q)}$

$\mathbf{e}(p \times q) = \mathbf{e}(p) \cup \mathbf{e}(q)$

$\mathbf{g}(p \times q) = \mathbf{g}(p) \cap \mathbf{g}(q)$.

This means that the product of two processes guarantees to avoid (reject) all the traces they avoid (reject) themselves, and to execute what they both consider as goal. However, it is not clear from the definition of process spaces [14] how to handle traces that reject to one process and escapes to the other when performing the composition.

In addition, process spaces define the joint behavior of environments as the *exclusive sum* of the processes [14]. The *exclusive sum* of two processes p and q is a process $p \oplus q$ such that

$\mathbf{at}(p \oplus q) = (\mathbf{as}(p) \cap \mathbf{as}(q)) \cup \overline{(\mathbf{at}(p) \cap \mathbf{at}(q))}$

$\mathbf{at}(p \oplus q) = \mathbf{at}(p) \cap \mathbf{at}(q)$,

or

$\mathbf{r}(p \oplus q) = \mathbf{r}(p) \cup \mathbf{r}(q)$

$\mathbf{e}(p \oplus q) = (\mathbf{e}(p) \cup \mathbf{e}(q)) \cap \overline{\mathbf{r}(p)} \cap \overline{\mathbf{r}(q)}$

$\mathbf{g}(p \oplus q) = \mathbf{g}(p) \cap \mathbf{g}(q)$.

Notice that there are no compatibility conditions on the processes involved in product or exclusive sum, i.e., any two processes in a process space can be involved in these operations.

Product and exclusive sum are associative and commutative, and therefore can be applied to any number of processes. In addition, these operations can be extended to arbitrary sets of processes [14].

As an example of how to carry out the composition of two processes, we show how two inertial buffers can be composed. Figure 2.10 shows two buffers modeled as safety processes, and their composition. The two processes are defined over the same execution set, $\{a, b, c\}^*$. Notice how the actions which are considered external for each process lead to self loops in all states, e.g. a for the process $p2$. The composition process automaton is formed by the Cartesian product of the two individual processes in terms of the states and edges. Each state in the composition is qualified as goal, escape, or reject according to the qualifiers of the corresponding states in the individual processes. For example, when two goal states are composed, the resulting is a goal state. However, when a goal state is composed with an escape state, the resulting state is an escape.

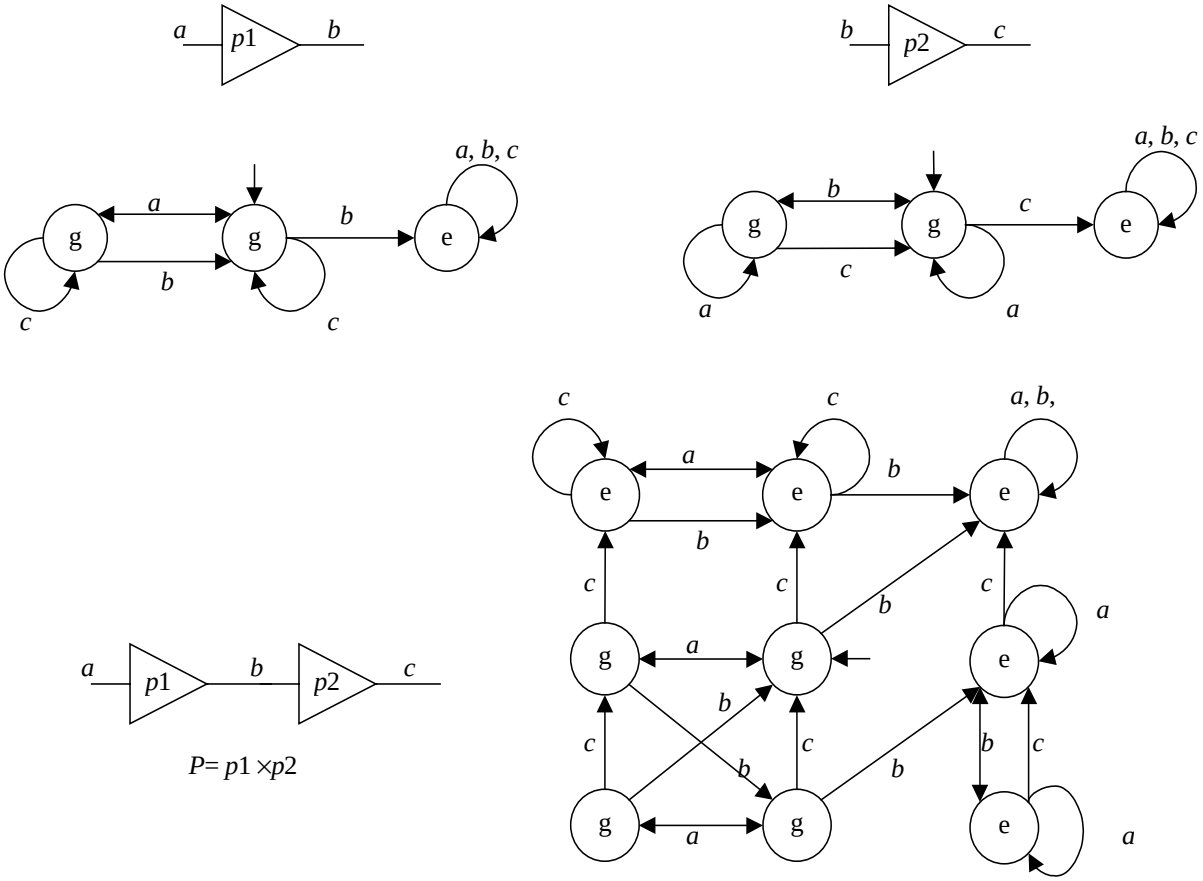


Figure 2.10: Two inertial buffers and their composition.

Robustness

Robustness expresses the property of a process that the device it represents imposes no constraints on the environment it deals with [14, 16]. For safety processes, this means that the device accepts all the inputs it receives from the environment. Therefore, a process *p* is robust when it has an empty reject set **r**(*p*).

Consider, for example, the safety process of the AND gate represented by the automaton in Figure 2.8. This is not a robust process since the reject set is not empty. Execution *aba*, for instance, is a reject since it includes an illegal input (the second *a* pulse cannot occur

before the output changes). On the contrary, the process automaton in Figure 2.11 represents a robust safety process that models the same candy machine of Section 2.1.1. Notice that inputs (*P*1: *Push*1 and *P*2: *Push*2) are accepted in every state. The permanent escape state E is reached when illegal outputs are issued, and out of it all actions are enabled randomly.

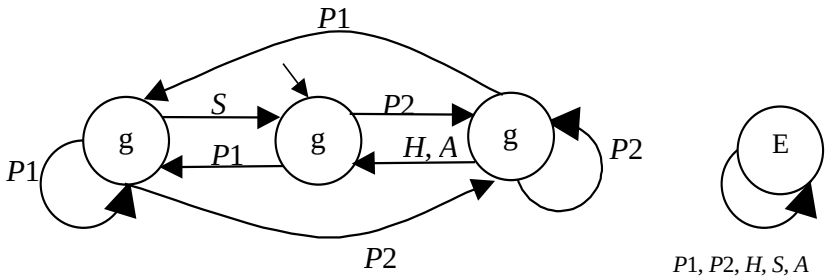


Figure 2.11: A robust safety process of a candy machine.

Refinement

Refinement is a relation between two processes that allows for one process *p* to fully replace a second process *q*. This relation is expressed in terms of the accessible and acceptable sets of executions of processes [14]:

A process *p* is said to *refine* a process *q*, written $p \sqsupseteq q$, if

$$(\mathbf{at}(p) \supseteq \mathbf{at}(q)) \wedge (\mathbf{as}(p) \subseteq \mathbf{as}(q)),$$

or, equivalently, if

$$\mathbf{r}(p) \subseteq \mathbf{r}(q) \wedge \mathbf{e}(p) \supseteq \mathbf{e}(q).$$

In the case when refinement exists in both directions between two processes, i.e., $p \sqsupseteq q$ and $q \sqsupseteq p$, we say that the two processes are equal since they have equal accessible and acceptable sets.

To illustrate how to check the refinement relation between two processes, we reuse the same example from Section 2.1.6 (Refinement of I/O automata). We model *CMCUST* and *CCUST* as safety processes. Figure 2.12 shows the process automata that represent the safety processes *CMCUST* and *CCUST*. Notice that all actions belong to the same alphabet, $U = \{P1, P2, S, H, A, BS\}$.

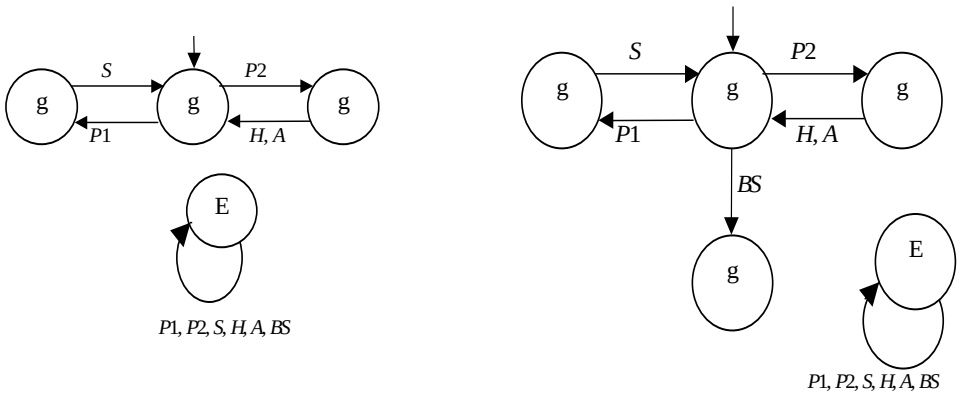


Figure 2.12: Process automata for a) Safety process *CCUST* and b) Safety process *CMCUST*. (Missing transitions in both process automata lead the permanent escape states denoted E.)

We check whether *CCUST* refines *CMCUST*.

$\mathbf{r}(CCUST) = \mathbf{r}(CMCUST) = \varnothing$, (both processes have no reject executions)

$\mathbf{e}(CCUST) = \mathbf{e}(CMCUST) \cup \{P1S, P2H, P2A\}^*BS \Rightarrow \mathbf{e}(CCUST) \supseteq \mathbf{e}(CMCUST)$.

Therefore, we conclude that $CCUST \sqsupseteq CMCUST$.

On the other hand, *CMCUST* does not refine *CCUST* because $P1SBS \notin \mathbf{e}(CMCUST)$ although $P1SBS \in \mathbf{e}(CCUST)$.

Hiding

Following [14] (chapter 8) and [16], we define the optimistic hiding operation performed on processes, and we show that this operation preserves refinement between processes.

We start by defining an operation that eliminates certain actions from the executions of a process.

Definition 2.8 For alphabet U and a subset $B \subseteq U$, let $<\downarrow B> \subseteq U^* \times U^*$ be a binary relation on finite words over U such that:

a) $(\varepsilon,\varepsilon) \in <\downarrow B>$

b) $(u, v) \in <\downarrow B> \wedge a \notin B \Rightarrow (ua, v) \in <\downarrow B>$

c) $(u, v) \in <\downarrow B> \wedge b \in B \Rightarrow (ub, vb) \in <\downarrow B>$

d) each pair from $<\downarrow B>$ satisfies a), b), or c).

Let $<\downarrow B>'$ be the inverse of the relation $<\downarrow B>$.

In words, $<\downarrow B>$ eliminates from an execution all actions from outside B, while $<\downarrow B>'$ inserts in an execution actions from outside B in arbitrary numbers and at arbitrary places. For example, we have the following: *abcba* $<\downarrow\{a,c\}>$ *aca*, *aca* $<\downarrow\{a,c\}>'$ *abcba*, and *aca* $<\downarrow\{a,c\}>'$ *bbacbbbab*.

For a subset X of U^*, let $<\downarrow B>X = \{<\downarrow B> u \mid u \in X\}$

For a process p, let opt-hide$_B(p)$ (read optimistic hide of p over B) be defined such that:

at(opt-hide$_B(p)$) = $<\downarrow B>' <\downarrow B>$ **at**(p), and

e(opt-hide$_B(p)$) = $<\downarrow B>' <\downarrow B>$ **e**(p).

For example, let P be the composed process from Figure 2.10. The process $<\downarrow\{a,c\}>$ is illustrated in Figure 2.13 below.

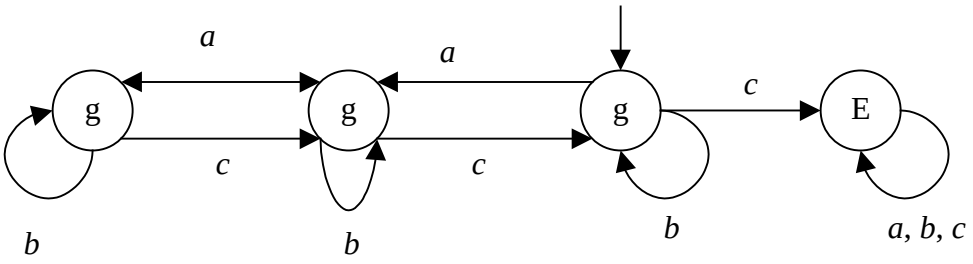


Figure 2.13: The process automaton from Figure 2.10 subject to the projection relation.

Proposition 2.1 For a process p defined over U^* and an alphabet $B \subseteq U$,

$p \sqsubseteq$ opt-hide$_B(p)$.

Proof:

First, we show that $\forall\, X \subseteq U^*$, $<\downarrow B>' <\downarrow B> X \supseteq X$

Let $u \in X$ and $\{v\} = <\downarrow B>\{u\}$

We have $v<\downarrow B>\, u$, and also $u<\downarrow B>'\, v$, so $u<\downarrow B>'<\downarrow B> u$.

Therefore, $\mathbf{e}(\text{opt-hide}_B(p)) = <\downarrow B>'<\downarrow B>\, \mathbf{e}(p) \supseteq \mathbf{e}(p) \Rightarrow \mathbf{as}(\text{opt-hide}_B(p)) \subseteq \mathbf{as}(p)$

and $\mathbf{at}(\ (\text{opt-hide}_B(p)) = <\downarrow B>'<\downarrow B>\, \mathbf{at}(p) \supseteq \mathbf{at}(p)$

$\Rightarrow p \sqsubseteq$ opt-hide$_B(p)$. □

Reflection

The *reflection* [14] of a process p is a process $q = -p$ represented by

$\mathbf{as}(q) = \mathbf{at}(p)$ and

$\mathbf{at}(q) = \mathbf{as}(p)$.

Again, we can define reflection in terms of **r**, **g**, and **e**:

$\mathbf{r}(q) = \mathbf{e}(p)$

$\mathbf{e}(q) = \mathbf{r}(p)$, and

$\mathbf{g}(q) = \mathbf{g}(p)$.

2.3 FIREMAPS

FIREMAPS was developed by R. Negulescu at the University of Waterloo. The name stands for finitary and regular manipulation of processes and systems. This tool automates the operations and manipulation of processes. In this section, we present a brief description of the tool and its uses for our purposes. For a more extensive reference about the FIREMAPS tool, we refer the reader to [14].

FIREMAPS uses the Polish notation for expressions, where an expression consists of the operator, followed by the operands. Operands in this notation may be constants, variables, or even expressions. The Polish notation offers several advantages. First of all, it does not need parentheses, and expressions are represented with less complexity. Second, parsing expressions is made easy with Polish notation. Third, there is no need for operator precedence, which means that users have less to memorize.

FIREMAPS has several data types: process (*p*), action list (*a*), integer list (*l*), string (*t*), integer (*n*), bit (*b*), etc. Each data type is designated by a single letter, which is used to form some of the operator names as follows. For each data type, there are operators for reading, writing, and variable assignment, plus some file operations. If *x* is the letter of a data type, the operators for read, write, and variable assignment are (*xr*), (*wx*), and (=*x*), respectively. The reading operator (*xr*) must be followed by an ASCII representation of the object being read, and it returns that object. The writing operator takes as operand an object and returns nothing. The variable assignment operator takes as operands the variable name and the object being assigned, and it returns nothing. In general, the operators that return nothing constitute commands and can be invoked from the FIREMAPS prompt. For example, the following two commands read a string, assign it to a variable *my_string*, and print the value of *my_string*.

```
(=t) my_string (tr) "sample string"
(wt) my_string
# the reply was:  "sample string"
```

The # sign starts a comment which ends at the end of the line in which # appears. In response to the first command, the tool does not produce any output. In response to the second command, the tool prints "sample string" as claimed in the comment. Action and integer lists have a simple format, containing the number of elements in the list, a colon, and then the list itself. For instance,

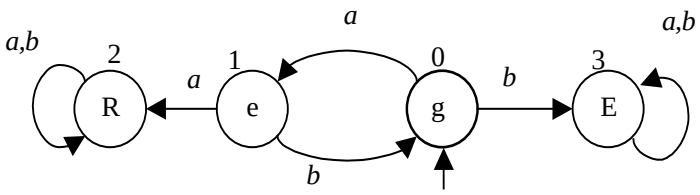
```
5: a b a c a
```

is a valid action list, and

```
2: 15 22
```

is a valid integer list.

Processes are represented in FIREMAPS by process automata. For example, the process automaton in Figure 2.14(a) is represented by the text in Figure 2.14(b). For reference, we have annotated the states in Figure 2.14(a) by state codes. The FIREMAPS format for processes is self-explaining except for the short strings next to the state codes in Figure 2.14(b). These short strings describe the type of a state: if 's' is present, the state is accessible; if 't' is present, the state is acceptable; if 'i' is present, the state is an initial state. There can be more than one initial state. On the other hand, a goal state is both accessible and acceptable, i.e., 's' and 't' should be both present. To comply with process space theory, any state that is reachable from an initial state must be either accessible, acceptable, or both. In addition, each state is assigned an integer qualifier that reflects its type (accessible, acceptable, or both). Therefore, the state qualifier indicates whether the state is goal (g), reject (r), or escape (e). An initial state is indicated by an incoming arrow that points to it. Notice that states 2 and 3 are called R and E respectively. The capital letters are used to denote permanent reject and escape states. Permanent reject and escape states are states from which the process cannot exit.



(a)

```
2 actions; 4 states; 8 edges;
  actions: a, b;
  states: 0 sti, 1 t, 2 s, 3 t;
  edges:
  from 0: a 1, b 3;
  from 1: a 2, b 0;
  from 2: a 2, b 2;
  from 3: a 3, b 3.
```

(b)

Figure 2.14: Example (a) process automaton and (b) FIREMAPS representation.

Table 1 lists the FIREMAPS commands that are used in the examples in this report. In the following, we describe some of the most used commands, and we illustrate the descriptions by examples.

Table 1: Some FIREMAPS notations.

Operators and notations	Function and meaning
=	Process Assignment
(pr)	Read Process
(nr)	Read integer list
(ar)	Read action list
(tr)	Read string
P	System of processes
(add)	Adds processes to a system
(fill)	Completes a process w.r.t. an action list by edges to a state
(empty)	Empty system operator
(wa)	Prints an action list
(wb)	Prints a bit value
(R)	Verifies robustness of a process
*	Product operator
^	Exclusive sum operator
-	Reflection operator
\	Difference and hiding operator
(mdm)	Minimization and determinization operator
(apply)	Simulation operator that returns states visited
(sti_apply)	Simulation operator to returns qualifiers of the states visited
[=	Refinement operator
(in)	Reads in a file
(quit)	Exits from the tool

There are several simulation commands in FIREMAPS, taking a process and an action list as arguments, and yielding an integer list as the result. (apply) and (sti_apply) are examples of these commands, and they both produce an integer list as an output. The integers in the obtained integer lists represent the states that have been reached following the given action list. The integers can represent the actual state codes if the command used is (apply) or the state qualifier (accessible, acceptable, and initial) if the command used is (sti_apply). For example, in the case of (sti_apply), the qualifier obtained is 7 (in binary: 111) if the state reached is an initial goal state and 6 (in binary: 110) when the state is simply a goal state. To illustrate, we use (sti-apply) to simulate the trace *abb* on the process in Figure 2.12. We write

```
(sti_apply) p (ar) 3: a b b
```

to signify that we are simulating an action list of three actions on the process *p*. The output is an integer list that represents the qualifiers of the states visited. In this case, we obtain:

```
7 6 6 2
```

This means that starting from the initial state, we reach a permanent escape as result of applying the trace *abb*.

To check refinement, robustness, or to find a counter-example to robustness, FIREMAPS uses a BDD-based breadth-first search of the state-spaces. The search consists of several passes, where at each pass a new layer of states is visited.

The refinement operator takes as arguments two processes and returns a bit whose value reflects the validity of the refinement between the two processes. The value of the bit can be printed using the (wb) command. For example, the following command line checks process *p*1 refines *p*2 and prints the resulting bit value.

```
(wb) [=  p2  p1
```

On the other hand, the robustness operator takes as argument a process and returns a bit. The value of the bit reflects whether the process is robust and can be printed. As an example the following command line checks if process *p* is robust and prints the corresponding bit value.

(wb) (R) *p*

When robustness fails, a counter example can be obtained to indicate the reason of the failure, i.e. the reject execution that prevents the process from being robust. In this case, we use (aR) instead of (R).

In addition, all operations on processes are possible in the tool. Composition, reflection, and hiding are examples of these operations. Symbol '∗' represents the product operator, the '-' represents reflection, and '\' is used to denote hiding.

Finally, we briefly discuss minimization and determinization of process automata. In

minimization, equivalent states are merged based on their equivalent behavior. Determinization, needed after hiding actions in a process automaton, is based on the usual state subset construction. At a given time, the non-deterministic automaton (obtained after hiding) may be in a set of states. This set is replaced by one state, and that state is assigned a qualifier (reject, goal, or escape) that is the 'weakest' qualifier of all qualifiers of the states in the set. Referring to a refinement order, the reject qualifier is considered to be weaker than the goal qualifier, which is considered to be weaker than the escape qualifier. For example, if a set includes a reject and an escape state, the set is replaced by a reject state. On the other hand, if a goal state and an escape state are present in the set, the replacing state becomes goal.

As an illustration, we use the example of the two buffers and their composition from Section 2.2.5. The product process *P* needs not to observe the intermediate signal *b*. Therefore, we apply hiding to eliminate *b* from the alphabet of *P*. The resulting process is shown in Figure 2.15 where the occurrences of *b* are replaced by the empty word. Notice the nondeterminism that is introduced into the behavior of the product buffer as result of eliminating *b*. Notice that the resulting automaton has two initial states indicated by the incoming arrows that point to them. However, one initial state is a permanent escape state, which means that the automaton might start a normal execution and be trapped in the permanent escape state.

Since a process automaton needs to be deterministic, we apply determinization as described above. Figure 2.16 shows the determinization of the process automaton *P*. In Figure 2.16 a, we show the automaton *P* before determinization, and we label each state with a number (0, 1, 2, …). This helps us distinguish between the states with the same qualifier when we form the sets of equivalence states. For example, the process automaton can be in states 0 (goal) and 4 (escape) at the same time since they are both initial states. Therefore, we group them into one set. Similarly, we group states 1, 2, and 4 into one set. Finally, when no new sets can be created, we merge the states in each set into one state, and we give the resulting state the least of the qualifiers of the component states. For example, the set of states {0, 4} is replaced by one state, which is qualified as

goal. The resulting deterministic process automaton is shown in Figure 2.16 b.

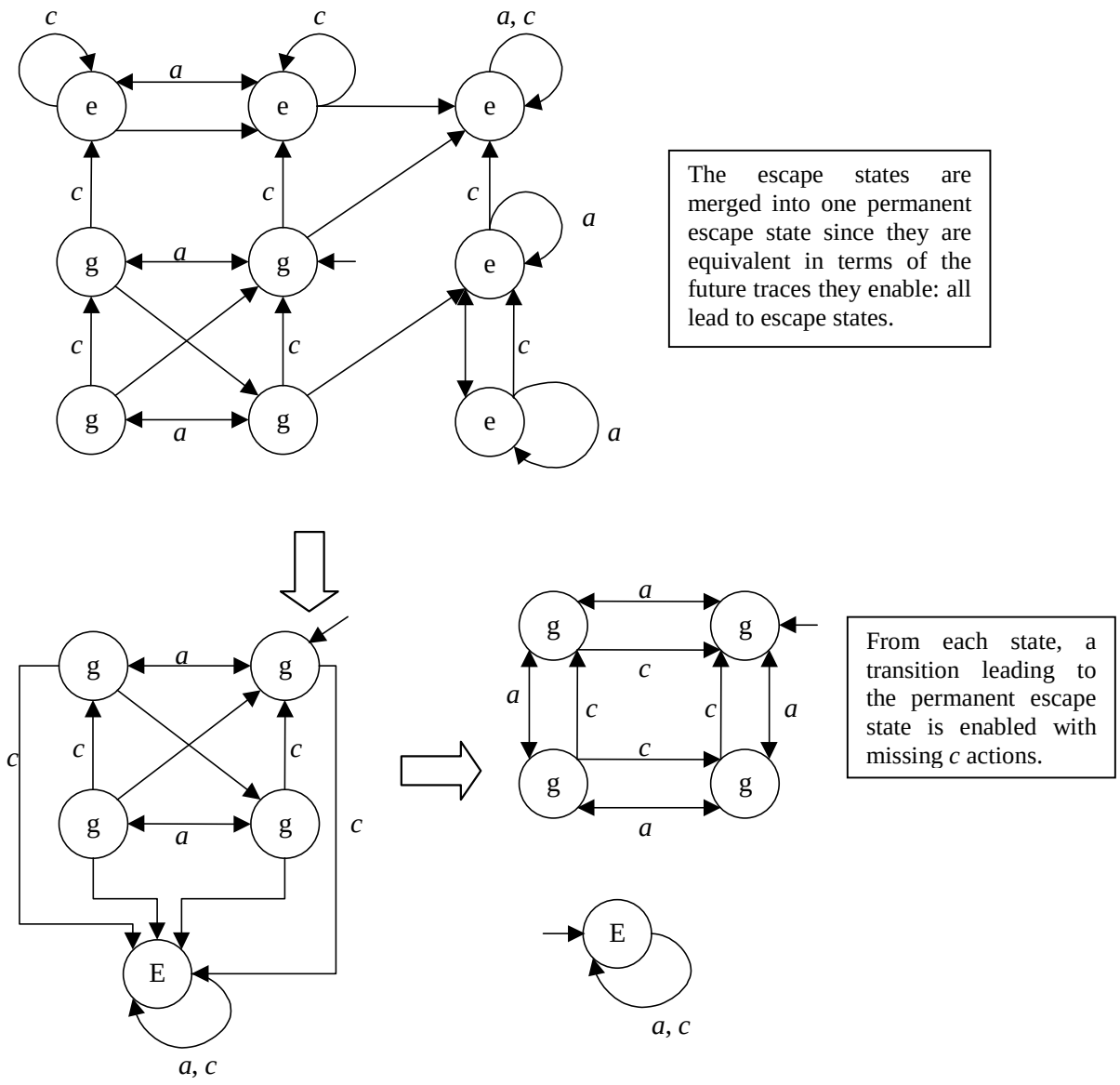


Figure 2.15: The process automaton *P* after hiding the action *b*.

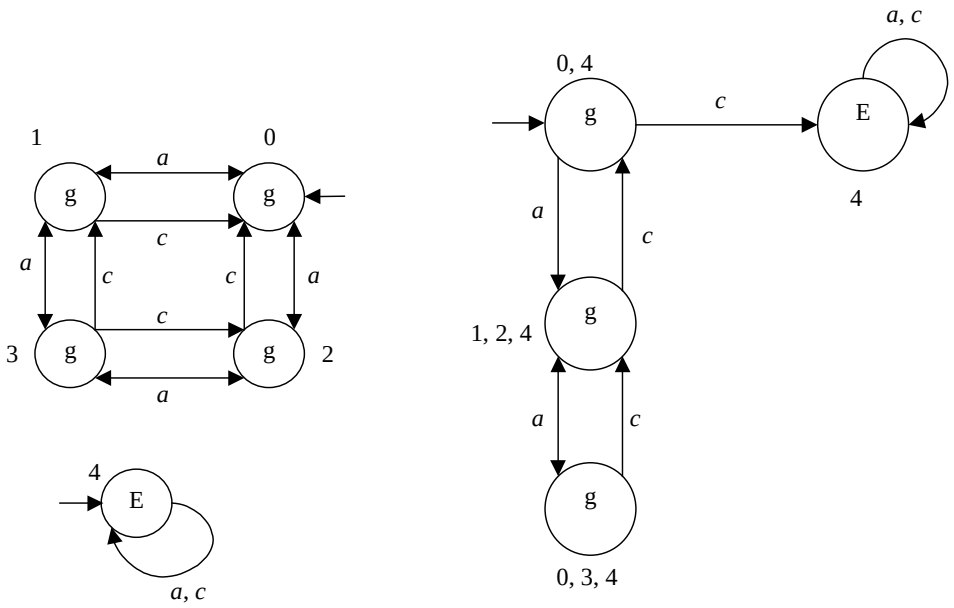


Figure 2.16: The process automaton *P* before and after determinization.

2.4 Mapping I/O Automata to Process Spaces

In this section, we build a relation between I/O automata and process spaces. In particular, we form semantics mappings to transform I/O automata into safety and finalization processes. Then, we study the two mappings to verify some homomorphism properties.

2.4.1 Motivation and Limitations

In this chapter, we described the I/O automata and process spaces models and summarized the notions and means each model uses to represent the behavior of a system. Finally, we concluded the existence of a common background between the two models, especially in terms of the operations and relations applicable to each model. In addition, we described the FIREMAPS, which automates the application of operations

and relations in process spaces. These reasons motivate the quest for semantics mappings that link the two models and allow the interchangeability of methods and results between them. Therefore, in this section, we form semantics mappings between I/O automata and process spaces. These mappings are finitary and relate I/O automata to safety and finalization processes.

On the other hand, for the mappings between I/O automata and process spaces to be transparent with respect to the relations and operations in both models, we need to state some limitations on the domain over which the mappings are defined. First of all, we limit our interest to the deterministic class of I/O automata since the process automata used to model safety and finalization processes are deterministic. In addition, all I/O automata that can be mapped into processes have an empty set of internal actions. This requirement ensures the preservation of the refinement relation through the mappings.

2.4.2 The Mappings

First of all, we form the mappings of an I/O automaton α to the corresponding safety and finalization processes. These are finitary semantics mappings, which yield processes defined over a finitary execution set U^*, where U is an alphabet that includes $In(\alpha) \cup Out(\alpha) \cup Int(\alpha)$. Therefore, in the sequel, we consider the following:

traces(α) and *etraces*(α) represent the sets of finite traces and external traces of an automaton, respectively.

ftraces(α) and *fetraces*(α) represent the sets of finite fair traces and external traces of an automaton, respectively.

In the following, we describe the two mappings from I/O automata to process spaces. The *safety mapping*, which transforms an I/O automaton into a safety process, and the *finalization mapping*, which yields the corresponding finalization process of the I/O automaton. Then, we analyze some homomorphic properties of the two mappings and verify that the mappings do not affect the relations and operations applicable to I/O automata when transformed into safety or finalization processes.

Safety mapping: The *safety process* of an automaton α is a process denoted $\sigma(\alpha)$, which has

- The set of reject executions:

r($\sigma(\alpha)$)={all finite traces, which include "illegal" inputs}=$\emptyset$. The reason for the emptiness of **r**($\sigma(\alpha)$) is that an I/O automaton is input enabled, i.e. accepts all inputs.

- The set of goal executions:

g($\sigma(\alpha)$)={all finite traces of α}= *traces*(α).

- The set of escape executions:

e($\sigma(\alpha)$)={all words from *acts*(α)$*$ that include illegal locally controlled actions; i.e., disabled outputs or disabled internal actions}= U^* - *traces*(α) .

Finalization mapping: The *finalization process* of an automaton α is a process $\varphi(\alpha)$, which has

- The set of reject executions:

r($\varphi(\alpha)$)={all finite traces of α that lead to an error; i.e., traces which include illegal inputs}= $\emptyset$. The reason for the emptiness of **r**($\varphi(\alpha)$) is that an I/O automaton is input enabled, i.e. accepts all inputs.

- The set of goal executions:

g($\varphi(\alpha)$)={all fair finite traces of α }= *ftraces*(α).

- The set of escape executions:

e($\varphi(\alpha)$)={all words from *acts*(α)* that contain illegal local actions} $\cup$ {all traces that enable output or internal actions at their end}=U^* - *ftraces*(α).

Next, we study the properties of the mappings from I/O automata to safety and finalization processes based on the definitions used above.

2.4.3 Properties of the Mappings

In this section, we check the following characteristics based on the definitions of the safety and finalization mappings:

- Injectivity.
- Surjectivity.
- The preservation of composition.
- The preservation of satisfaction.

2.4.3.1 Injectivity

In this section, we show that the safety and finalization mappings described previously are not injective using a counter example. Let us consider the two I/O automata in Figure 2.17. Both have the same action signature.

$In = \emptyset$.

$Out = \{b\}$.

$Int = \emptyset$.

Therefore, $U = \{b\}$, and $U^* = b^*$.

Figure 2.17: Two equivalent I/O automata.

The two automata α_1 and α_2 are not equal since they have different sets of states. However, they have the same images through the safety and finalization mappings.

Safety mapping: The mapping of the I/O automaton α_1 to a safety process is $\sigma(\alpha_1)$ such that

$\mathbf{r}(\sigma(\alpha_1))$ = {all traces of α_1 that contain "illegal" inputs}= $\emptyset$,

$\mathbf{g}(\sigma(\alpha_1))$ = {all traces of α_1 }= $traces(\alpha) = b^*$, and

$\mathbf{e}(\sigma(\alpha_1))$ = {all words of α_1 that contain "illegal" (disabled) outputs and internal actions}= $U^* - traces(\alpha) = b^* - b^* = \emptyset$.

As for the second automaton α_2, $\sigma(\alpha_2)$ is the corresponding safety process with

$\mathbf{r}(\sigma(\alpha_2))$ = {all traces of α_2 that contain "illegal" input}=$\emptyset$,

$\mathbf{g}(\sigma(\alpha_2))$ = {all traces of α_2 }= $traces(\alpha) = b^*$, and

$\mathbf{e}(\sigma(\alpha_2))$ ={all words of α_2 that contain "illegal" (disabled) outputs and internal actions}= $U^* - traces(\alpha) = b^* - b^* = \emptyset$.

Here, we notice that

$\mathbf{r}(\sigma(\alpha_1)) = \mathbf{r}(\sigma(\alpha_2)) = \emptyset$,

$\mathbf{g}(\sigma(\alpha_1)) = \mathbf{g}(\sigma(\alpha_2)) = b^*$, and

$\mathbf{e}(\sigma(\alpha_1)) = \mathbf{e}(\sigma(\alpha_2)) = \emptyset$.

This leads to $\sigma(\alpha_1) = \sigma(\alpha_2)$.

Finalization mapping: The mapping of automaton α_1 to a finalization process is given by $\varphi(\alpha_1)$ such that

$\mathbf{r}(\varphi(\alpha_1))$ = {all traces of α_1 that have illegal inputs}=$\emptyset$,

$\mathbf{g}(\varphi(\alpha_1))$ = {all traces of α_1 that are fair}= $ftraces(\alpha) = \emptyset$, and

$\mathbf{e}(\varphi(\alpha_1))$ = {all traces of α_1 that enable outputs at the end, all words of α_1 that contain "illegal" outputs}= $U^* - ftraces(\alpha) = b^* - \emptyset = b^*$.

For α_2, we define the finalization process $\varphi(\alpha_2)$ such that

$\mathbf{r}(\varphi(\alpha_2))$ ={all traces of α_2 that have illegal inputs}=$\emptyset$,

$\mathbf{g}(\varphi(\alpha_2))$ ={all traces of α_2 that are fair }= $ftraces(\alpha) = \emptyset$, and

$\mathbf{e}(\varphi(\alpha_2))$ ={all traces of α_2 that enable outputs at the end, all words of α_2 that contain "illegal" outputs }= $U^* - ftraces(\alpha) = b^* - \emptyset = b^*$.

Again, we notice that

$\mathbf{r}(\varphi(\alpha_1)) = \mathbf{r}(\varphi(\alpha_2)) = \emptyset$,

$\mathbf{g}(\varphi(\alpha_1)) = \mathbf{g}(\varphi(\alpha_2)) = b^*$, and

$\mathbf{e}(\varphi(\alpha_1)) = \mathbf{e}(\varphi(\alpha_2)) = \emptyset$.

Therefore, $\varphi(\alpha_1) = \varphi(\alpha_2)$.

As a conclusion, the safety and finalization mappings are not injective since the two different automata of Figure 3.1 lead to the same safety and finalization processes through the two mappings, respectively.

2.4.3.2 Surjectivity

Here, we show that the safety and finalization mappings are not surjective. This means that not every safety or finalization process is an image of an I/O automaton through the mappings, respectively.

Safety mapping: For any safety process σ represented by
r(σ) ={all finite traces, which include "illegal" inputs}=$\varnothing$
g(σ)={all finite traces of α}= *traces*(α), and
e(σ)={all words from *acts*(α)$*$ that include illegal locally controlled actions; i.e., disabled outputs or disabled internal actions}= U^* - *traces*(α).
In the case of nonempty **r** set, we cannot find any I/O automaton to be the inverse image of σ because of the input enabling property. An automaton accepts all inputs. This indicates that the mapping of I/O automata to safety processes is not a surjective one in general.

Finalization mapping: We follow the same way to verify that the finalization mapping is not surjective in general, and we use the following counter example.

Consider the finalization process *P* represented by the process automaton in Figure 2.18. This process is defined over the set of executions $U^* = a^*$. The two states, $q1$ and $q2$, of the process automaton are labeled e and g for escape and goal, respectively. The initial state is indicated by the incoming arrow that points to it. For **g**(*P*) to be the set of fair traces of an I/O automaton, *a* must be an input to the automaton. The reason is that $q2$ should be quiescent. In this case, $q1$ should also be quiescent because it enables only input actions, namely *a*. Therefore, **g**(*P*) has no inverse image in the sets of fair traces of I/O automata, and the mapping is not surjective.

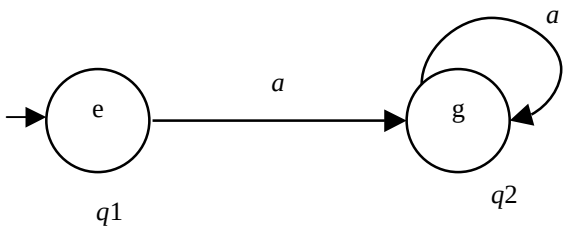


Figure 2.18: Process automaton of a finalization process.

2.4.3.3 Preservation of composition

In this section, we show that the composition operation on I/O automata mapped into a composition operation on process spaces. This means that the composition of two I/O automata maps into a composition of the corresponding safety processes or finalization processes.

Safety mapping: Here we show that the composition of two I/O automata maps into a composition of the corresponding safety processes, or formally speaking:
$\sigma(\alpha_1 \cdot \alpha_2) = \sigma(\alpha_1) \times \sigma(\alpha_2)$.

Let α be the composition of two I/O automata α_1 and α_2. The safety process $\sigma(\alpha_1 \cdot \alpha_2)$ could be represented by $\sigma(\alpha)$ with

$\mathbf{r}(\sigma(\alpha)) = \emptyset$.

$\mathbf{e}(\sigma(\alpha))$ = {words over the set of actions of α that have illegal locally controlled actions (outputs or internal actions)}=

= {words over the set of actions of α with "illegal" actions from the sets $Out(\alpha_1) \cup Out(\alpha_2)$ and $Int(\alpha_1) \cup Int(\alpha_2)$}=

= {words over the set of actions of α_1 with "illegal" locally controlled actions }$\cup${words over the set of actions of α_2 with "illegal" locally controlled actions}= U^* - $traces(\alpha_1) \cup U^*$ - $traces(\alpha_2)$=

= $\mathbf{e}(\sigma(\alpha_1)) \cup \mathbf{e}(\sigma(\alpha_2))$.

Then, $\mathbf{e}(\sigma(\alpha)) = \mathbf{e}(\sigma(\alpha_1)) \cup \mathbf{e}(\sigma(\alpha_2))$.

$\mathbf{g}(\sigma(\alpha)) = \overline{\mathbf{r}(\sigma(\alpha)) \cup \mathbf{e}(\sigma(\alpha))} = \overline{\mathbf{e}(\sigma(\alpha))} = \overline{\mathbf{e}(\sigma(\alpha_1)) \cup \mathbf{e}(\sigma(\alpha_2))}$.

On the other hand, let $p = \sigma(\alpha_1) \times \sigma(\alpha_2)$. We have:

$\mathbf{r}(P) = \varnothing$.

$\mathbf{e}(P) = \overline{\mathbf{as}(P)} = \overline{(\mathbf{as}\ \sigma(\alpha_1) \cap \mathbf{as}\ \sigma(\alpha_2))} = \overline{\mathbf{as}\ \sigma(\alpha_1)} \cup \overline{\mathbf{as}\ \sigma(\alpha_2)} = \mathbf{e}(\sigma(\alpha_1)) \cup \mathbf{e}(\sigma(\alpha_2))$.

$\mathbf{g}(P) = \overline{\mathbf{e}(P) \cup \mathbf{r}(P)} = \overline{\mathbf{e}(P)} = \overline{\mathbf{e}(\sigma(\alpha_1)) \cup \mathbf{e}(\sigma(\alpha_2))}$.

Notice that

$\mathbf{r}(\sigma(\alpha)) = \mathbf{r}(p)$,

$\mathbf{g}(\sigma(\alpha)) = \mathbf{g}(p)$, and

$\mathbf{e}(\sigma(\alpha)) = \mathbf{e}(p)$.

Since $\sigma(\alpha) = \sigma(\alpha_1 \cdot \alpha_2)$, we have $\sigma(\alpha_1 \cdot \alpha_2) = p = \sigma(\alpha_1) \times \sigma(\alpha_2)$.

As we can see, the mapping of I/O automata to safety processes preserves composition.

Finalization mapping: Here we show that the composition of two I/O automata maps into a composition of the corresponding finalization processes, or formally speaking:

$\varphi(\alpha_1 \cdot \alpha_2) = \varphi(\alpha_1) \times \varphi(\alpha_2)$.

To show the correspondence of the composition of I/O automata to a composition of finalization processes through the mapping, we follow the same line of proof used for safety mapping.

Let α be the composition of the two automata α_1 and α_2. Then, $\varphi(\alpha_1 \cdot \alpha_2)$ could be represented by the finalization process $\varphi(\alpha)$ with

$\mathbf{r}(\varphi(\alpha)) = \varnothing$.

$\mathbf{g}(\varphi(\alpha))$ = {all traces that are fair for α_1 and α_2} =

= {all traces that are fair for α_1}$\cap$ {all traces that are fair for α_2} =

$= \mathbf{g}(\varphi(\alpha_1)) \cap \mathbf{g}(\varphi(\alpha_2)) = \overline{\mathbf{e}(\varphi(\alpha_1))} \cap \overline{\mathbf{e}(\varphi(\alpha_2))} = \overline{\mathbf{e}(\varphi(\alpha_1)) \cup \mathbf{e}(\varphi(\alpha_2))}$.

$\mathbf{g}(\varphi(\alpha)) = \overline{\mathbf{e}(\varphi(\alpha_1)) \cup \mathbf{e}(\varphi(\alpha_2))}$.

$\mathbf{e}(\varphi(\alpha))$ = {all words over the set of actions of α with "illegal" locally controlled actions, and all traces of α that enable outputs at the end}=

= ({all words over the set of actions of α_1 with "illegal" locally controlled actions} $\cup$ {all words over the set of actions of α_2 with "illegal" locally controlled actions}) $\cup$ ({all traces of α_1 that enable outputs at the end } $\cup$ {all traces of α_2 that enable outputs at the end}) =

= ({all words over the set of actions of α_1 with "illegal" locally controlled actions, all traces of α_1 that enable outputs at the end} $\cup$ {all words over the set of actions of α_2 with "illegal" locally controlled actions, all traces of α_2 that enable outputs at the end}) =

= $\mathbf{e}(\varphi(\alpha_1)) \cup \mathbf{e}(\varphi(\alpha_2))$.

$\mathbf{e}(\varphi(\alpha))$ = $\mathbf{e}(\varphi(\alpha_1)) \cup \mathbf{e}(\varphi(\alpha_2))$.

On the other hand, let $\varphi(\alpha_1) \times \varphi(\alpha_2)$ be represented by the finalization process p such that

$\mathbf{r}(p)$ = $\varnothing$.

$\mathbf{e}(p)$ = $\overline{\mathbf{as}(P)} = (\overline{\mathbf{as}\ \varphi(\alpha_1) \cap \mathbf{as}\ \varphi(\alpha_2)}) = \overline{\mathbf{as}\ \varphi(\alpha_1)}\ \cup \overline{\mathbf{as}\ \varphi(\alpha_2)}$ = $\mathbf{e}(\varphi(\alpha_1)) \cup \mathbf{e}(\varphi(\alpha_2))$.

$\mathbf{g}(p)$ = $\overline{\mathbf{e}(\varphi(\alpha_1)) \cup \mathbf{r}(\varphi(\alpha_2))} = \overline{\mathbf{e}(\varphi)} = \overline{\mathbf{e}(\varphi(\alpha_1)) \cup \mathbf{e}(\varphi(\alpha_2))}$.

It follows that

$\mathbf{r}(\varphi(\alpha)) = \mathbf{r}(p)$,

$\mathbf{g}(\varphi(\alpha)) = \mathbf{g}(p)$, and

$\mathbf{e}(\varphi(\alpha)) = \mathbf{e}(p)$

Since $\sigma(\alpha) = \sigma(\alpha_1 \cdot \alpha_2)$, we have, $\varphi(\alpha_1 \cdot \alpha_2) = p = \varphi(\alpha_1) \times \varphi(\alpha_2)$.

In conclusion, the composition of I/O automata is preserved through the mapping to finalization processes.

2.4.3.4 Preservation of refinement

In this section, we prove that the refinement relation between I/O automata is preserved through the safety and finalization mappings to processes. We will associate the

implementation relation with refinement between safety processes, and the satisfaction relation with refinement between finalization processes. The motivation for this association is the following. The satisfaction relation between I/O automata reveals whether a system satisfies the requirement that something desired eventually happens. This requirement applies to finalization processes as well. At the same time, the implementation relation requires that an automaton correctly replaces another automaton in terms of the functionality, i.e., doing what it does and avoiding what it avoids. This applies to the definition of safety in process spaces.

Safety mapping: Here we show that the implementation relation between I/O automata maps into a refinement relation between the corresponding safety processes. Formally, this translates to the following:

α_1 implements $\alpha_2 \Leftrightarrow \sigma(\alpha_1) \sqsupseteq \sigma(\alpha_2)$.

Let $\sigma(\alpha_1)$ and $\sigma(\alpha_2)$ be the safety process images of two I/O automata α_1 and α_2, respectively.

α_1 implements α_2 $\Leftrightarrow$ $\mathit{etraces}(\alpha_1) \subseteq \mathit{etraces}(\alpha_2)$

$\Leftrightarrow$ $\mathit{traces}(\alpha_1) \subseteq \mathit{traces}(\alpha_2)$ $\quad (\mathit{Int}(\alpha_1) = \mathit{Int}(\alpha_2) = \emptyset)$

$\Leftrightarrow$ $\mathbf{g}(\sigma(\alpha_1)) \subseteq \mathbf{g}(\sigma(\alpha_2))$

$\Leftrightarrow$ $\overline{\mathbf{e}(\sigma(\alpha_1)) \cup \mathbf{r}(\sigma(\alpha_2))} \subseteq \overline{\mathbf{e}(\sigma(\alpha_2)) \cup \mathbf{r}(\sigma(\alpha_2))}$

$\Leftrightarrow$ $\overline{\mathbf{e}(\sigma(\alpha_1))} \subseteq \overline{\mathbf{e}(\sigma(\alpha_2))}$ $\quad$ (since $\mathbf{r}(\sigma(\alpha_1)) = \mathbf{r}(\sigma(\alpha_2)) = \emptyset$)

$\Leftrightarrow$ $\mathbf{e}(\sigma(\alpha_1)) \supseteq \mathbf{e}(\sigma(\alpha_2))) \wedge (\mathbf{r}(\sigma(\alpha_1)) \subseteq \mathbf{r}(\sigma(\alpha_2)))$

$\Leftrightarrow$ $(\mathbf{as}(\sigma(\alpha_1)) \subseteq \mathbf{as}(\sigma(\alpha_2))) \wedge (\mathbf{at}(\sigma(\alpha_1)) \supseteq \mathbf{at}(\sigma(\alpha_2)))$

$\Leftrightarrow$ $\sigma(\alpha_1) \sqsupseteq \sigma(\alpha_2)$.

Therefore, the implementation relation between automata is transformed, through the mapping, to a refinement relation between safety processes.

Finalization mapping: Here we show that the satisfaction relation between I/O automata maps into a refinement relation between the corresponding finalization processes.

Formally, we can state this requirement as follows:

α_1 satisfies $\alpha_2 \Leftrightarrow \varphi(\alpha_1) \sqsupseteq \varphi(\alpha_2)$.

We let $\varphi(\alpha_1)$ and $\varphi(\alpha_2)$ be the respective finalization process images of the I/O automata α_1 and α_2.

α_1 satisfies $\alpha_2 \Leftrightarrow$ α_1 and α_2 have the same sets of inputs and outputs, and

$fetraces(\alpha_1) \subseteq fetraces(\alpha_2)$

$\Leftrightarrow$ $ftraces(\alpha_1) \subseteq ftraces(\alpha_2)$ $(Int(\alpha_1) = Int(\alpha_2) = \emptyset)$

$\Leftrightarrow$ $\overline{fetraces(\alpha_1)} \supseteq \overline{fetraces(\alpha_2)}$

since $\mathbf{r}(\varphi(\alpha_1)) = \mathbf{r}(\varphi(\alpha_2)) = \emptyset$,

$\mathbf{e}(\varphi(\alpha_1)) = \overline{fetraces(\alpha_1)}$, and

$\mathbf{e}(\varphi(\alpha_2)) = \overline{fetraces(\alpha_2)}$, we have:

α_1 satisfies $\alpha_2 \Leftrightarrow$ $(\mathbf{r}(\varphi(\alpha_1)) \subseteq \mathbf{r}(\varphi(\alpha_2))) \wedge (\mathbf{e}(\varphi(\alpha_1)) \supseteq \mathbf{e}(\varphi(\alpha_2)))$

$\Leftrightarrow$ $(\mathbf{at}(\varphi(\alpha_1)) \supseteq \mathbf{at}(\varphi(\alpha_2))) \wedge (\mathbf{as}(\varphi(\alpha_1)) \subseteq \mathbf{as}(\varphi(\alpha_2)))$

$\Leftrightarrow$ $\varphi(\alpha_1) \sqsupseteq \varphi(\alpha_2)$.

Therefore, the satisfaction relation between automata is transformed, through the mapping, to a refinement relation between finalization processes.

2.5 Conclusions

In this chapter, we summarized the main notions and definitions of two formalisms used in modeling concurrent systems, I/O automata and process spaces, and presented a relation between I/O automata and process spaces. In the following, we compare the two modes based on the definitions and descriptions of both models from Section 2.1 and Section 2.2.

We base our comparison between I/O automata and process spaces on the notions and means each model uses to represent the behavior of a system. We list the following observations:

- I/O automata differentiate between actions and classify them as inputs, outputs, or internal. Process spaces do not formally differentiate between inputs, outputs, and internal actions. All actions are handled as elements of one alphabet; inputs, outputs, and internals are differentiated solely by the effects of their occurrences.

- I/O automata enable all inputs at every state. Process automata enable all actions at every state; illegal outputs lead to a "permanent escape" state.

- I/O automata do not record any violations due to “illegal” inputs or outputs. Instead, “illegal” inputs, when they occur, lead to regular states. “Illegal” outputs, however, are simply disabled. On the other hand, process spaces, in general, record every illegal incident and treat it as a violation. These violations are considered escapes or rejects based on whether the violation is committed by the device or the environment, respectively.

- In I/O automata, composition is possible under the condition of compatibility of action signatures. In process space, there are no compatibility conditions; any two processes from a process space can be composed by product or exclusive sum, and can be compared by refinement.

- The refinement relations in both models are based on trace inclusion. However, I/O automata differentiate between two aspects of refinement: implementation, which compares I/O automata based on their external traces; and satisfaction, which compares the fair external traces of two I/O automata. On the other hand, process spaces consider only one aspect of refinement, which compares two processes, based on their permissible and prohibited behaviors.

These observations confirm the existence of a common background for I/O automata and process spaces, which allows the interchangeability of properties and means between the two models. In section 2.4, we formulated this relation in the form of semantics mappings from I/O automata to process spaces.

The mappings we developed above are believed to have applications in formal modeling and verification of distributed systems. These applications are based on the properties that we verified. We are interested mainly in the correspondence of composition of I/O automata to composition of processes and the refinement between I/O automata to refinement between processes. The mappings offer an alternative to handling the verification problems in the I/O automata model. Using these mappings, a verification problem in I/O automata, for example, could be mapped to an equivalent verification problem in process spaces. Then, the problem is solved automatically using the FIREMAPS tool. On the other hand, we believe the mappings give an alternative to solving asynchronous equations, deriving a specification for a missing part of a system from the overall specification of that system and the known part of its implementation, in I/O automata and FSM's. An asynchronous equation in I/O automata could be mapped to an asynchronous equation in process spaces through the mappings. The problem is then solved using FIREMAPS, and the results could be mapped back to the initial I/O automata model.

In the next chapter (Chapter 3), we describe how to solve the asynchronous equation problem in process spaces and generalize the method to I/O automata and other labeled transition systems using the mappings we formed in this chapter. Then, we describe how to transfer the results back to I/O automata once an asynchronous equation has been solved in process spaces.

Chapter 3

Asynchronous Equations

3.1 Introduction

The problem of asynchronous equations, supervisory control, or model-matching, consists of finding a controller for a given open loop system so that the resulting closed loop system matches a desired input/output behavior. In other words, it could be described as follows:

Given a reference model M, which represents a specification, and a plant M1 that presents part of the implementation, we seek a controller M2 such that the composition of M1 and M2 matches M.

The model-matching problem has many applications, which could be summarized in the following:

- Classical control problems: In such problems, a system is to be controlled to exhibit a certain desired behavior [4].
- Engineering changes: In the design of large digital systems, if errors are detected, the cost of redoing the entire system design is tremendous. Therefore, a solution is to change the behavior of the present system to meet the correct desired one. This could be achieved by model matching [4].
- System factorization and decomposition: Verifying large systems could be achieved by hierarchical proof and decomposition to avoid large resource usage. Model matching helps finding complementary portions of the already known ones. In general, this reduces the verification efforts [4].

In this chapter, we devise a method to solve asynchronous equations. This method is described in terms of process spaces. However, we use the mappings we developed in Section 2.4 to show that the method presents an alternative to solving asynchronous equations in other models as well, namely I/O automata [5] and I/O FSM's [19]. Our main concern, in transferring the problem from one model to the other, is to profit from the automated tool FIREMAPS in solving the equations.

In the following sections, we present our method to solve asynchronous equations illustrated by means of an example. In addition, the example illustrates the properties of the mappings we formed in Chapter 2 and the use of process spaces to model systems. This serves to show how the mappings of Chapter 2 transfer the method of solving the asynchronous equations from process spaces to other models. For this purpose, we use the example of an asynchronous equation from [19]. Furthermore, we use the example to illustrate how to use FIREMAPS tool to manipulate processes, especially to automate the process of solving an asynchronous equation.

3.2 Problem Statement

We mainly consider asynchronous equations in I/O automata. However, we can address the problem in any input/output labeled transition system I/O LTS [2]. A labeled transition system (LTS) consists of states and transitions between states that are labeled with actions. The only difference between I/O LTS and I/O automata is that in I/O LTS, not all inputs are necessarily enabled in all states like in I/O automata. Although an incomplete specification of inputs in a model shows an extra degree of freedom in modeling, it loosens the constraints on the relation between a system and its environment. This might lead to misconception of the responsibilities of each within a model. To account for this, we consider three possible options:

1- Make all missing inputs lead to a trap state, out of which there is no way to resume normal execution of actions, and in which the automaton refrains from executing locally controlled actions.

2- Make all missing inputs produce self-loops in the automaton. This amounts to ignoring the occurrence of these missing events.
3- Make all missing inputs lead to a trap state out of which there is no way to resume normal execution of actions, but the automaton can issue locally controlled events arbitrarily.

We choose option 3 so that we can capture as much as possible the realistic behavior of the system in consideration. Informally speaking, this choice agrees with the viewpoint expressed in [6] that invalid events result in divergence. For example, consider the I/O LTS that has one input (*a*) and one output (*b*), Figure 3.1. The initial state is indicated by the incoming arrow that points to it. This system is not input complete (*a* is not enabled in both states). In order to transform it into an I/O automaton, we need to specify the behavior of the system in response to *a* in state 1. Following option 3, we add a trap state to the system, Figure 3.1 b, which is entered after receiving *a* in state 1. After that, the system cannot return to normal execution and the output *b* might be issued arbitrarily.

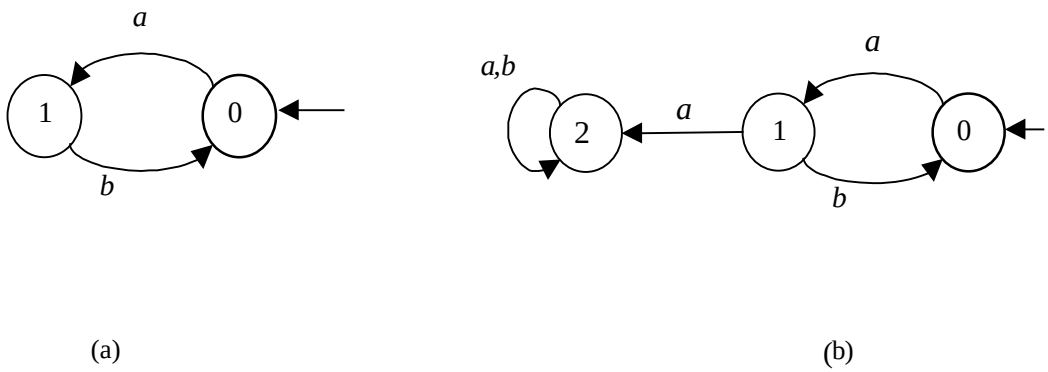


Figure 3.1: An I/O LTS and its corresponding I/O automaton.

In specific, we consider the case when the original problem is described in terms of input/output finite state machines (I/O FSM's), which use atomic I/O transitions that alternate inputs and outputs. This means that a transition on input *i* generates output *o* and is labeled *i*/*o* [19]. We start by converting the I/O FSM's into I/O automata. An I/O FSM is converted into an I/O automaton by
a) Splitting each atomic I/O transition into two consecutive transitions. In the newly introduced intermediate states, inputs are not enabled.

b) Making all missing inputs lead to a trap state from which there is no way to resume normal execution of actions, but the automaton can issue output events arbitrarily.

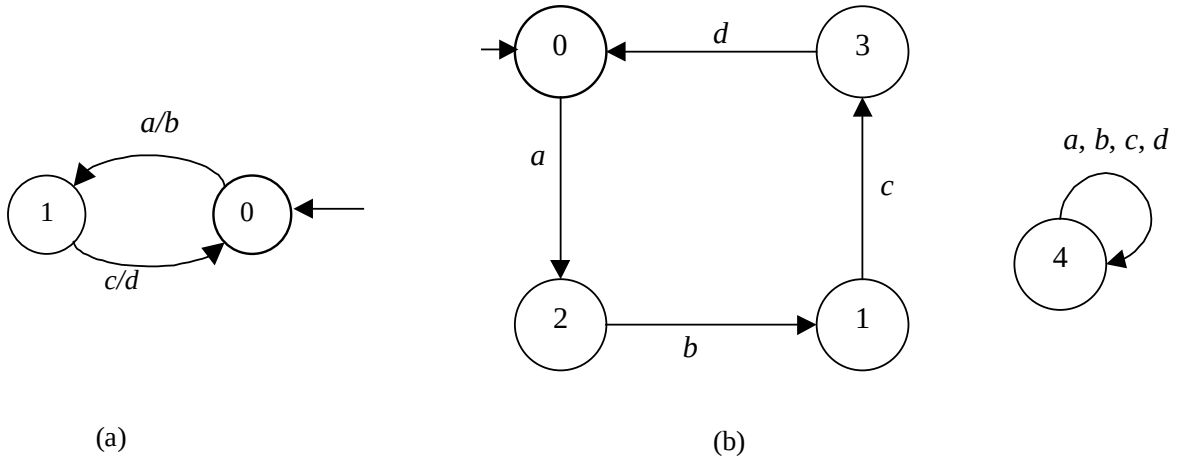


Figure 3.2:a) I/O FSM and b) the corresponding I/O automaton.

As an example, we transform a simple I/O FSM (Figure 3.2 a) into an I/O automaton (Figure 3.2 b). The events *a* and c represent the inputs of the system, and *b* and *d* represent the outputs. Notice that each atomic transition, e.g. *a*/*b*, is split into two consecutive transitions and intermediate states (2 and 3) are added. Also a trap state is added to the system. This state is reached by the missing inputs.

Then, we use our mapping to transform the problem to a problem in process spaces. The next step is to solve the asynchronous equation in terms of processes. We use FIREMAPS to compute the solution.

To illustrate, we use an example from [19], in which we address the following problem: given the behaviors of the coffee shop (reference) and the waiter (plant), we need to determine which behaviors of the coffee machine (controller) would fulfil the environment's requirements and additional constraints imposed by us. Following [19], we are interested both in a general solution and in a solution constrained to have only a limited number of internal communication events (freedom from divergence or livelock).

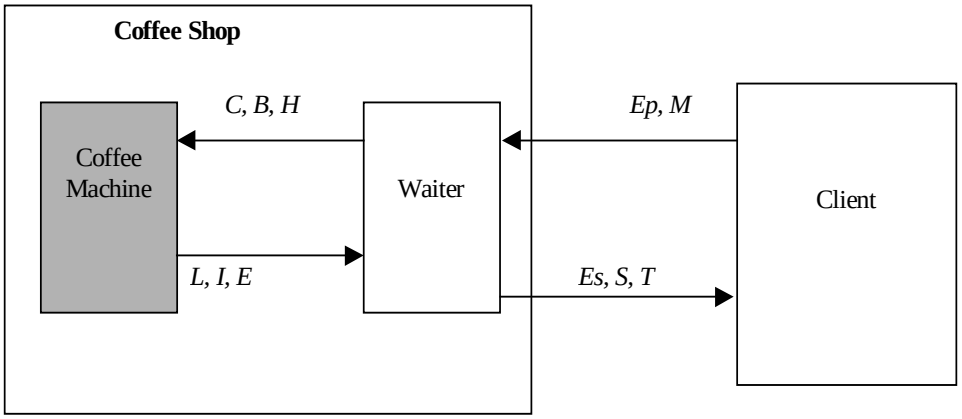


Figure 3.3: Block diagram for the coffee shop and environment.

Figure 3.3 shows that the coffee machine and the waiter are grouped together while the environment forms another part. The implication of this grouping is to view the system as an environment-device combination. In this combination, the client represents the environment, and the waiter-machine group represents the device, called the coffee shop. The figure also shows the input and output actions of each part of the system.

The problem was originally described in [19], in terms of I/O FSM's. Therefore, the first step is to convert the I/O FSM's in the original example into I/O automata using the method described above. First, the coffee shop is modeled by an I/O automaton, *Coffee-shop*. This I/O automaton specifies the desired interactions between our coffee shop and the client, and it has the following action signature:

In(*Coffee-shop*) = {*Ep*, *M*} (these labels stand for *Espresso please* and *Money*).
Out(*Coffee-shop*) = { *Es, S, T*} (these labels stand for *Espresso served*, *Sorry, Thanks*).
Int(*Coffee-shop*) = ∅.

The I/O automaton *Coffee-shop* is shown in Figure 3.4. The initial state is indicated by the incoming arrow that points to it. In this automaton, state 5 represents a trap state, where the automaton is stuck after the environment supplies "illegal" inputs. For instance, when the environment executes consecutive *Ep* actions, the coffee shop is not bounded to any specific behavior.

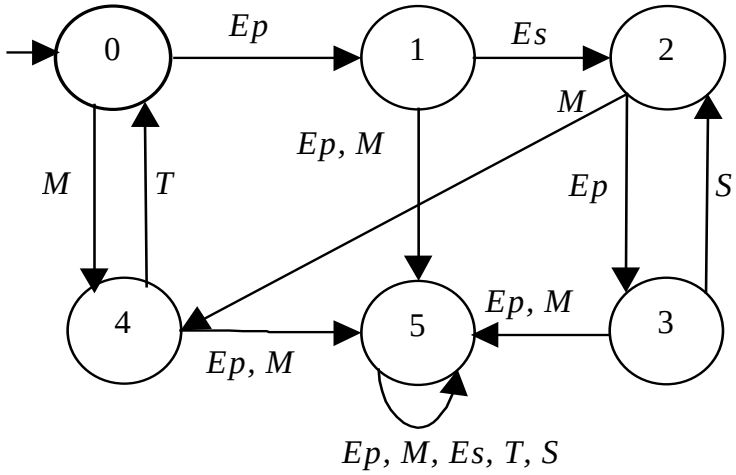


Figure 3.4: The I/O automaton *Coffee-shop*.

The behavior of the waiter is also described by an I/O automaton. The automaton *Waiter* has the following action signature:

In(*Waiter*) = {*Ep, M, L, E, I*} (these labels stand for *Espresso please, Money, Lamp, Espresso, Idle*).

Out(*Waiter*) = {*Es, S, T, C, B, H*} (these labels stand for *Espresso served, Sorry, Thanks, Coin Button Hit*).

Int(*Waiter*) = ∅.

Here, *Ep, M, Es, S,* and *T* represent the same actions from the automaton *Coffee-shop*, and *Lamp, Espresso, Idle, Coin, Button,* and *Hit* are actions of the interaction between the waiter and the coffee machine. Figure 3.5 shows the I/O automaton *Waiter*. Again, in this figure, the illegal inputs to the waiter, either from the environment or the coffee machine, lead to a trap state, state 12. However, for legibility, we do not show the edges leading to state 12. We consider that every missing input action from the diagram leads to state 12. State 12 has self-loops on every action.

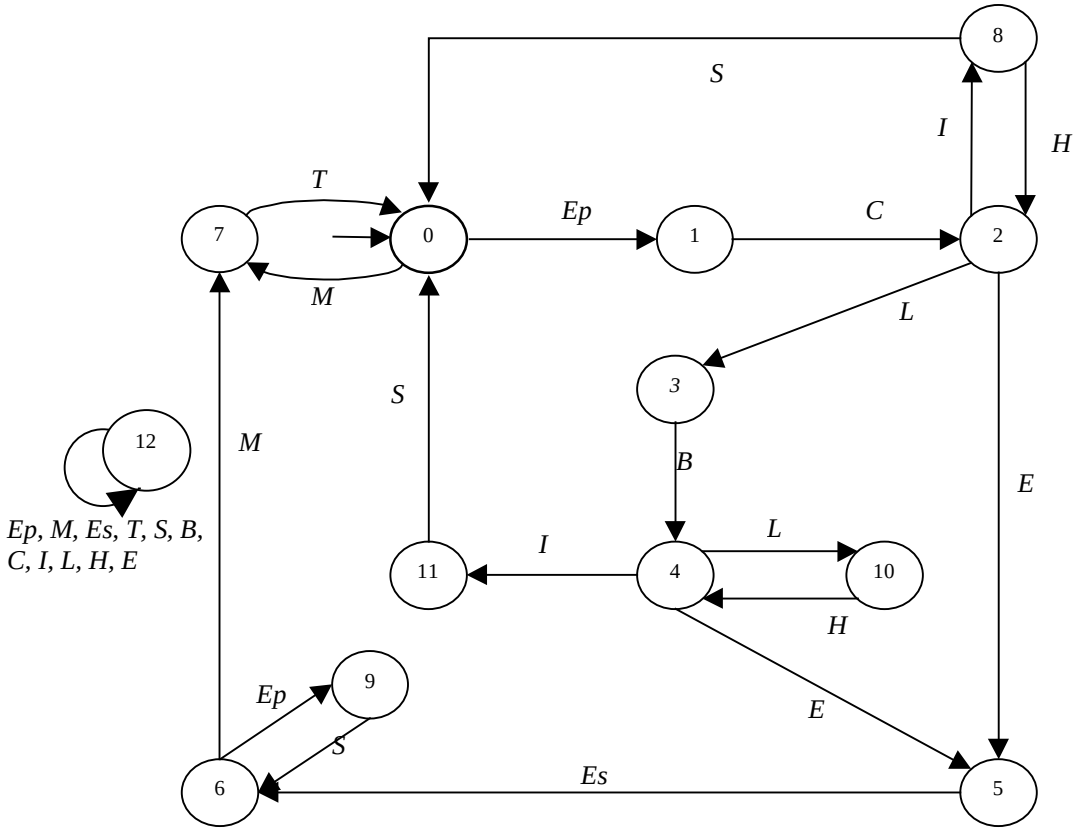


Figure 3.5: The I/O automaton *Waiter*. (Missing input transitions lead to state 12)

3.3 Finding the General Solution

Finding a general solution for the problem consists of finding the automaton *Coffee-machine*, whose composition with the automaton *Waiter*, after hiding communications between the two automata, implements the automaton *Coffee-shop*. This solution is denoted general to indicate that it is implemented by any other solution.

In order to solve this problem, we map the two automata *Waiter* and *Coffee-shop* to processes, using the mappings from Section 2.4. Then, we use the tool FIREMAPS to obtain the solution. Later, we describe how to obtain the divergence-free solution for the problem, and we show how to map the results, from process spaces, back into I/O automata.

We solve the problem in terms of safety and finalization processes. In the following, we show the procedure of obtaining the general solution for safety processes. As for the case of finalization processes, we only present the results and compare them to the safety results.

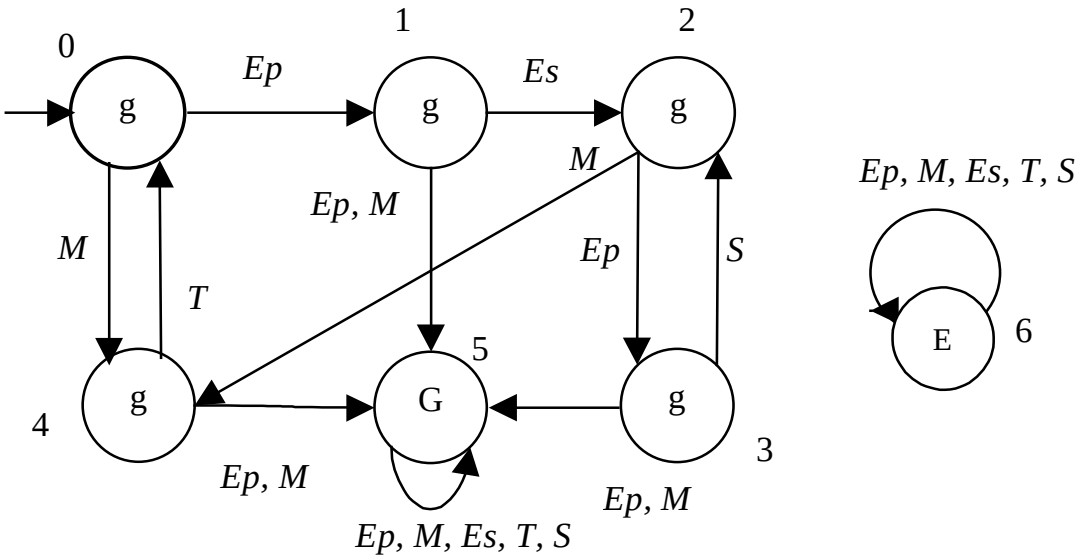


Figure 3.6: The safety process for the automaton *Coffee-shop*. (Missing transitions lead to state 6.)

The safety processes corresponding to the automata *Coffee-shop* and *Waiter* are represented using process automata. Figure 3.6 and Figure 3.7 show the process automata *Coffee-shop* and *Waiter*, respectively. States are marked as g, e, or r to indicate that executions leading to those states are goals, escapes, or rejects, respectively. The initial state is indicated by an incoming arrow that points to it. To increase clarity, we use capital letters, G and E, to indicate the permanent goal state and the permanent escape state, respectively. Such states are trap states reachable by traces including illegal inputs and outputs, respectively.

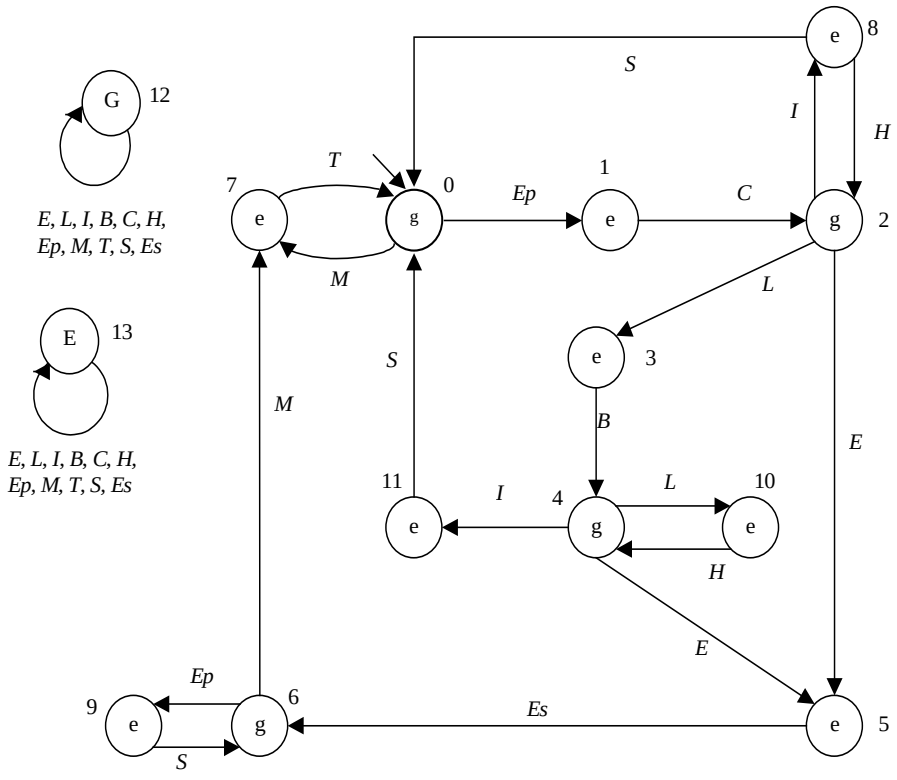


Figure 3.7: The safety process for the I/O automaton *Waiter*. (Missing input transitions lead to state 12. Missing output transitions lead to state 13.)

The same actions for the I/O automata *Coffee-shop* and *Waiter* are used in their process automata; however, process automata make no formal distinction between inputs and outputs. All actions must be enabled in all states of the process; however, illegal inputs and outputs are only distinguished by their effects. What was meant to be a disabled output in the I/O automata model is considered as permanent escape here. The permanent escapes lead to the escape state labeled E. However, for legibility, we do not show the actual transitions to this. Thus for both process automata, *Coffee-shop* and *Waiter*, any disabled output leads to the permanent escape state in the corresponding safety process. On the other hand, we have modeled the trap states of the automata as goal states in the safety processes. The next step is to describe the safety processes corresponding for *Waiter* and *Coffee-shop* in a script file to be read and executed by FIREMAPS. The listing is shown in Figure 3.8.

```
= coffee-shop (pr)
5 actions;  7 states; 35 edges;
actions: ep, m, es, s, t;
states: 0 sti, 1 st, 2 st, 3 st, 4 st, 5 t, 6 st ;
edges:
from 0: ep 1, m 3, es 5, s 5, t 5;
from 1: ep 6, m 6, es 2, s 5, t 5;
from 2: ep 4, m 3, es 5, s 5, t 5;
from 3: ep 6, m 6, es 5, s 5, t 0;
from 4: ep 6, m 6, es 5, s 2, t 5;
from 5: ep 5, m 5, es 5, s 5, t 5;
from 6: ep 6, m 6, es 6, s 6, t 6.


= waiter (fill) (fill) (pr)
11 actions; 14 states; 41 edges;
actions: ep, es, m, t, s, c, b, h, l, e, i;
states: 0 sti, 1 st, 2 st, 3 st, 4 st, 5 st, 6 st, 7 st, 8 st, 9 st, 10 st,
11 t, 12 st, 13 st;
edges:
from 0: ep 1, m 7;
from 1: c 2;
from 2: i 8, l 3, e 5;
from 3: b 4;
from 4: e 5, i 13, l 10;
from 5: es 6;
from 6: m 7, ep 9;
from 7: t 0;
from 8: s 0, h 2;
from 9: s 6;
from 10: h 4;
from 11: es 11, t 11, s 11, c 11, b 11, h 11, ep 11, m 11, l 11, e 11, i 11;
from 12: es 12, t 12, s 12, c 12, b 12, h 12, ep 12, m 12, l 12, e 12, i 12;
from 13: s 0.

(ar) 6: es t s c b h  (nr) 11 # missing outputs in process lead to state 11
(ar) 5: ep m l e i    (nr) 12 # missing inputs in process lead to state 12


= coffee-machine - (mdm) - \ - * waiter - coffee-shop (ar) 5: ep m es t s
      # the formulation of the solution
(print) (tr) "solution:"
(wp) coffee-machine
```

Figure 3.8: The listing of the script file.

The FIREMAPS descriptions in Figure 3.8 correspond to the process automata in Figure 3.6 and 3.7. In addition to process descriptions, we include in the script file, Figure 3.8, the following commands, which implements the equation solving procedure and write it to a file.

```
= sol - (mdm) \ * waiter - coffee-shop (ar) 5: ep m es t s
(wp) coffee-machine
```

The equation solving procedure consists of:

1. Reflecting the reference process, *Coffee-shop.*

2. Obtaining the product of *Waiter* and the reflection of *Coffee-shop*. This will produce the environment of the missing part of the system, *Coffee-machine*.
3. Hiding the actions (see Definition 2.7), which are irrelevant to the produced process. The produced solution needs not to observe *Ep*, *M*, *Es*, *T*, and *S*. These actions are communications between the *Coffee-shop* and *Waiter*.
4. Determinizing and minimizing the obtained process.
5. Reflecting the result to obtain the solution, *Coffee-machine*.

The output of these manipulations is a process in FIREMAPS format listed in a separate file, Figure 3.9. Figure 3.10 shows the corresponding process automaton *Coffee-machine*. States 0 and 6 represent trap states, and actual transitions to them are not shown to increase clarity.

```
Solution:
6 actions; 7 states; 42 edges;
actions: i, l, e, b, c, h;
states: 6 t, 4 st, 5 sti, 0 s, 2 st, 1 st, 3 st;
edges:
   from 6: i 6, l 6, e 6, b 6, c 6, h 6;
   from 4: i 6, l 3, e 5, b 0, c 0, h 0;
   from 5: i 6, l 6, e 6, b 0, c 2, h 0;
   from 0: i 0, l 0, e 0, b 0, c 0, h 0;
   from 2: i 6, l 1, e 5, b 0, c 0, h 0;
   from 1: i 6, l 6, e 6, b 4, c 0, h 0;
   from 3: i 6, l 6, e 6, b 0, c 0, h 4.
```

Figure 3.9: The listing for the solution *Coffee-machine.*

The same procedure is followed to solve the problem in terms of finalization processes. We map the I/O automata to finalization processes and use the same manipulations in FIREMAPS to obtain the solution for the problem. The solution obtained is a finalization process, and it is shown in Figure 3.11.

The two processes in Figure 3.10 and Figure 3.11 illustrate the difference between the safety and finalization properties. The difference between the two processes can be identified in terms of the traces they execute. For example, notice that the trace (*C L B*) is a goal for the safety process (Figure 3.10) while the finalization process (Figure 3.11) considers (*C L B*) an escape. A safety process marks as violations only those traces, which include illegal inputs or outputs. (*C L B*) shows none of these. On the other hand, a

finalization process marks as violations, in addition to illegal inputs and outputs, any illegal stopping. In this case (Figure 3.11), *coffee-machine* is not allowed to stop after receiving an input *B* (button) without responding with an output, *E* (*Espresso*) or *L* (*Lamp*).

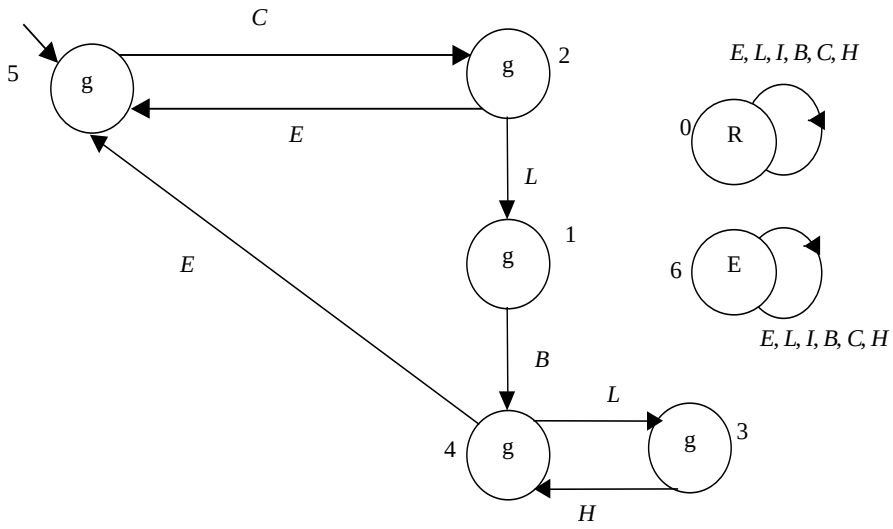


Figure 3.10: The solution process automaton *Coffee-machine* in terms of safety processes. (Missing input transitions lead to state 0. Missing output transitions lead to state 6)

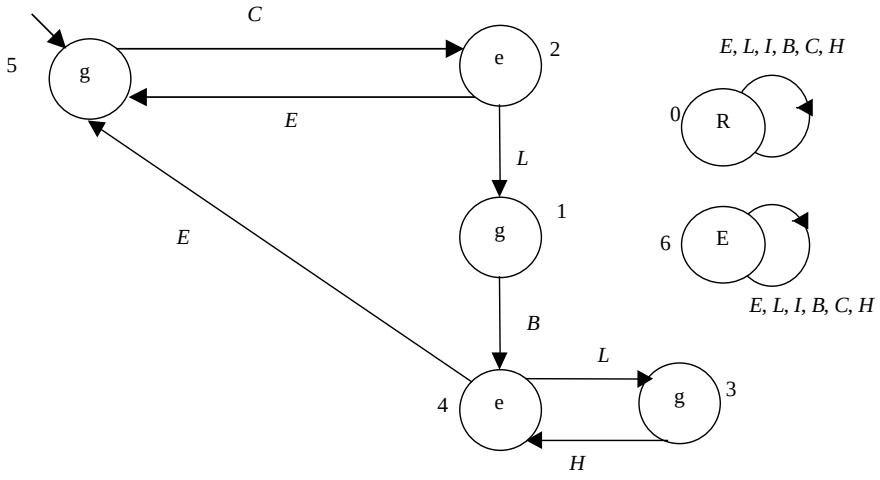


Figure 3.10: The solution process automaton *coffee-machine* in terms of finalization processes. (Missing input transitions lead to state 0. Missing output transitions lead to state 6)

3.4 Divergence-free Solution

In this section, we describe how to obtain a solution for the asynchronous equation that guarantees divergence-freedom in the behavior of the system consisting of the *Client*, *Waiter*, and *Coffee-machine*. *Client* represents the environment of the coffee shop. In fact, we show that we obtain a divergence-free solution in the case of safety processes only. The results show that using finalization processes to solve asynchronous equations does not necessarily yield a divergence-free solution.

3.4.1 Basic Definitions

In general, divergences are infinite internal executions that occur between various components of a system. Sometimes avoiding such divergences is desired (eg., to reduce power consumption in CMOS implementations), and the system that avoids them is called divergence-free. First, we define divergence-freedom in terms of I/O automata and process spaces. Then, we show that the method we follow to obtain the solution for the asynchronous equation results in a divergence-free system.

I/O automata are input enabled, i.e., they cannot refuse input actions. Therefore, divergence freedom can only be stated in terms of locally controlled actions. In I/O automata, a composition of automata avoids an infinite sequence of local actions, *e*, when, at least, one component automaton avoids them.

Similarly, the composition of processes, in process spaces, guarantees that if a component in the composition avoids some execution, the whole system formed by the composition avoids that execution. Therefore, the definition of divergence-freedom in process spaces could be expressed in terms of systems of processes directly.

Definition 3.1. An I/O automaton α defined over a set of actions $acts(\alpha)$ is *divergence-free* with respect to an alphabet $A \subseteq Out(\alpha) \cup Int(\alpha)$ when the automaton avoids all infinite words from A^{ω}. This could be expressed as follows:

$acts(\alpha)^*A^\omega \cap traces(\alpha) = \emptyset$. ($A^\omega$ denotes the set of all infinite sequences of actions from A.)

Definition 3.1 can be directly extended to a composition of I/O automata. In a composition of I/O automata, each locally controlled action is under the control of one individual I/O automaton (compatibility of I/O automata). Therefore, when one automaton in the composition avoids a sequence, this sequence is avoided by the composition as well.

We introduce the following notations:
For words u and v from U^*, we write $u \leq v$ if u is a prefix of v.
The set of all the prefixes of goal executions of a process is defined as:
prefg(p)= $\{u \mid \exists\, v: uv \in \mathbf{g}(p)\}$, where u and v are words from U^*.

Definition 3.2 A set S of processes defined over U^* is *weakly-divergence-free* (*w.d.f.*) with respect to an alphabet $A \subseteq U$ if for any infinite sequence l, there exists at least one process, in S, which avoids that sequence. This can be stated more formally as follows:
$\forall$ infinite sequence $l \in U^*A^\omega$, $\exists$ a finite prefix t of l, $t \in$ **prefg**(p), such that $\forall$ finite prefix u of l such that $t \leq u$, $\exists$ a process $p \in S : u \in \mathbf{e}(p)$.

Definition 3.3 A set S of processes defined over U^* is *strongly-divergence-free* (*s.d.f.*) with respect to an alphabet $A \subseteq U$ if there exists, at least, one process in S which avoids every infinite sequence from U^*A^ω. In other words, we want that
$\forall$ infinite sequence $l \in U^*A^\omega$, $\exists$ a finite prefix t of l, $t \in$ **prefg**(p), and $\exists$ a process $p \in S$ such that $\forall$ finite prefix u of l such that $t \leq u$, we have $u \in \mathbf{e}(p)$.

Definition 3.4 A process p defined over U^* is *safety-healthy* if:
$\forall$ sequence $t \in U^*$, $\forall$ sequence $u \in U^*$, we have $t \in \mathbf{e}(p) \Rightarrow tu \in \mathbf{e}(p)$.

This means that a safety process p is safety-healthy process if it has a suffix-closed set of escape executions. Therefore, when p avoids one execution, it avoids all the successors of this execution.

In the following propositions, Proposition 3.1 and 3.2, we show that replacing a process by its refinement in a set does not affect weakly-divergence-freedom and that safety-healthiness guarantees strongly-divergence-freedom to a w.d.f. set of processes.

Proposition 3.1 For a set S of processes defined over U^*, for an alphabet $A \subseteq U$, and for processes q_1 and q_2 defined over U^*, if $q_1 \in S$ and S is w.d.f. with respect to A and $q_1 \sqsubseteq q_2$; then, $S^1 = (S \setminus \{ q_1\}) \cup \{q_2\}$ is w.d.f. with respect to A.

Proof: Since $q_1 \sqsubseteq q_2$, we have $\mathbf{e}(q_1) \subseteq \mathbf{e}(q_2)$.

S is w.d.f. with respect to $A \Rightarrow$ (by the definition of weakly-divergence-freedom) $\exists$ a process $h \in S$ such that: $\forall$ infinite sequence $l \in U^*A^\omega$, $\exists$ a finite prefix $t \leq l$, $t \in \mathbf{prefg}(p)$, and $\forall$ finite prefix u of l such that $t \leq u$, we have $u \in \mathbf{e}(h)$.

Therefore, we have two cases:

(Case 1) $h = q_1$, notice that $u \in \mathbf{e}(q_1) \Rightarrow u \in \mathbf{e}(q_2)$ since $q_1 \sqsubseteq q_2$.

(Case 2) $h \neq q_1$, $\exists\ h \in S^1$ by the construction of S^1.

In both cases, $\exists\ h \in S^1$ such that $u \in \mathbf{e}(h)$. □

Proposition 3.2 For a set S of processes defined over U^* and for an alphabet $A \subseteq U$, if S is w.d.f. with respect to A and each process h in S is safety-healthy; then S is s.d.f. with respect to A.

Proof: Let $l \in U^*A^\omega$.

S is w.d.f. with respect to $A \Rightarrow \exists\ t \leq l$, $t \in \mathbf{e}(p)$: $\forall\ u$ such that $t \leq u \leq l$, $\exists\ h_u \in S$: $u \in \mathbf{e}(h_u)$.

In particular, $\exists\ h_t \in S : t \in \mathbf{e}(h_t)$. Since $h_t \in S$ and all elements of S are safety-healthy, we have: $\forall\ u$ such that $t \leq u \leq l$, $u \in \mathbf{e}(h_t)$. □

It can be noticed from Proposition 3.2 that for safety processes, *weakly-divergence-freedom* and *strongly-divergence-freedom* are equivalent. However, in the context of this study, the check for both properties is considered for some technical reasons that will be discussed later.

3.4.2 Divergence-free Solution in Safety Processes

In our example *B*, *C*, *H*, *L*, *E*, and *I* are actions that involve the *Waiter* and the *Coffee-machine*, but are ignored by the *Client*. Any infinite sequence involving these actions, not interleaved with actions noticed by the *Client*, are considered as divergences. In the previous section, we notice that the sequence (*L*, *H*), that does not include any actions involving *Client*, is accessible to both processes *Waiter* and *Coffee-machine*. Therefore, the composition of the two processes enables this cycle as well, and the system obtained is not divergence-free.

To obtain a divergence-free system, we redo our example with a new constraint: we limit the number of consecutive internal interactions between *Waiter* and *Coffee-machine* that are not interleaved with external actions. Following [19], we will seek solutions for the cases when the number of consecutive internal interactions between *Waiter* and *Coffee-machine* is limited to 2 and 4, respectively.

The same procedure of mapping I/O automata into processes and solving for the *Coffee-machin*e using FIREMAPS is used to obtain the divergence-free solution. However, we augment the specification by a new process, $llock_n$. This process counts the internal interactions between the *Waiter* and *Coffee-machine* and signals a violation when the count goes higher than the desired limit *n*. In addition, $llock_n$ is robust, which means that it accepts all inputs from its environment.

In Figure 3.12, we show $llock_2$ and $llock_4$ for both cases when the limit is 2 and 4, respectively. We compose this process with the process *Coffee-shop* eliminate divergence.

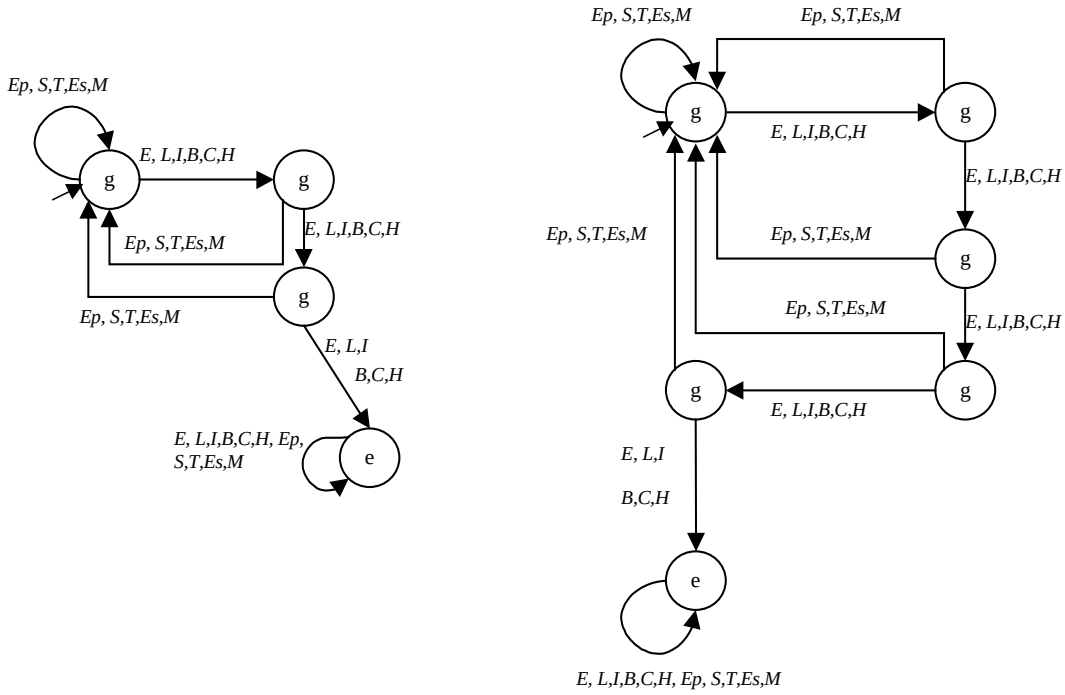


Figure 3.12: The process automata *llock*$_2$ (left) and *llock*$_4$ (right).

The new solution is obtained as follows:

```
= sol2 - (mdm) \ * waiter - ^ llock2 serve (ar) 5: ep m es t s
```

for the limit=2, and

```
= sol4 - (mdm) \ * waiter - ^ llock4 serve (ar) 5: ep m es t s
```

for the limit=4.

Where sol2 and sol4 represent the solutions obtained in the cases limit=2 and limit=4, respectively.

First, we obtain the exclusive sum of *Coffee-shop* and $llock_2$ or $llock_4$ depending on the limit required. The reason behind using the exclusive sum to compose the limiting process with *Coffee-shop* is that we need a composition that does not affect the refinement between the new solution and the original one, Figure 3.10. The limiting

process is a robust process, and the result of the exclusive sum of it with *Coffee-shop* refines *Coffee-shop*. The outcome of the product is reflected and composed with the *Waiter* by product again. After that, the same hiding of irrelevant actions is applied. Finally, the result is reflected to obtain the *Coffee-machine* as *sol2* for $llock_2$ and *sol4* for $llock_4$. The solutions for limit=2 and limit=4 are described by the process automata in Figures 3.13 and 3.14, respectively.

The solutions obtained are both safety processes. However, Figure 3.13 and 3.14 show that the solutions do not necessarily correspond to physical devices. In addition, they do not represent safety-healthy processes. First, the solutions obtained include some illegal inputs leading to escapes and illegal outputs leading to rejects. That is why they do not necessarily correspond to physical device. However, such solutions do not contradict the general theory of process spaces. For example, in Figure 3.13, the transition (1, *B*, 8) leads to state 8 which is an escape state, although *B* is an input action. This means that the *Coffee-machine* has the obligation to avoid *B* in state 1. Second, notice that, in both cases, the escape set is not suffix-closed, the condition for safety-healthiness, i.e., the process does not avoid all the suffixes of an escape trace. For example, the trace (*C L B*) is an escape for the process in Figure 3.13 while the trace (*C L B B*) is a reject.

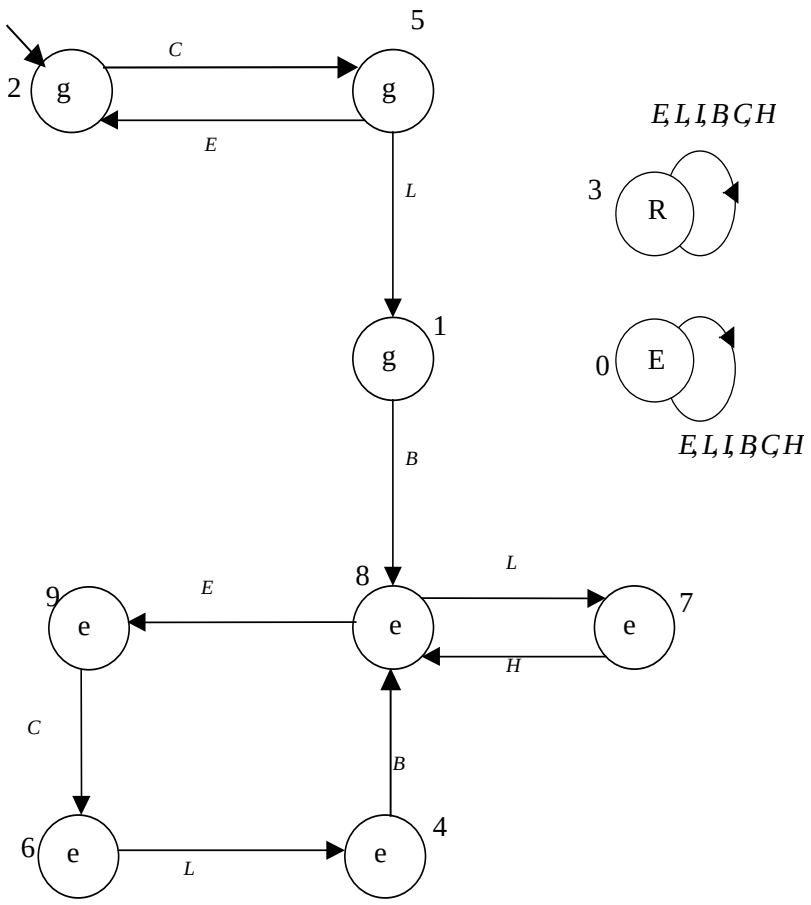


Figure 3.13: Safety process automaton for the solution with limit = 2.

In order to overcome these problems, we devise a procedure that transforms arbitrary processes to physically realizable processes that refine the original processes. This procedure is based on general transformations that can be applied outside the context of asynchronous equations as well. At the same time, the procedure guarantees that the produced processes are safety-healthy, i.e., they have suffix-closed escape sets.

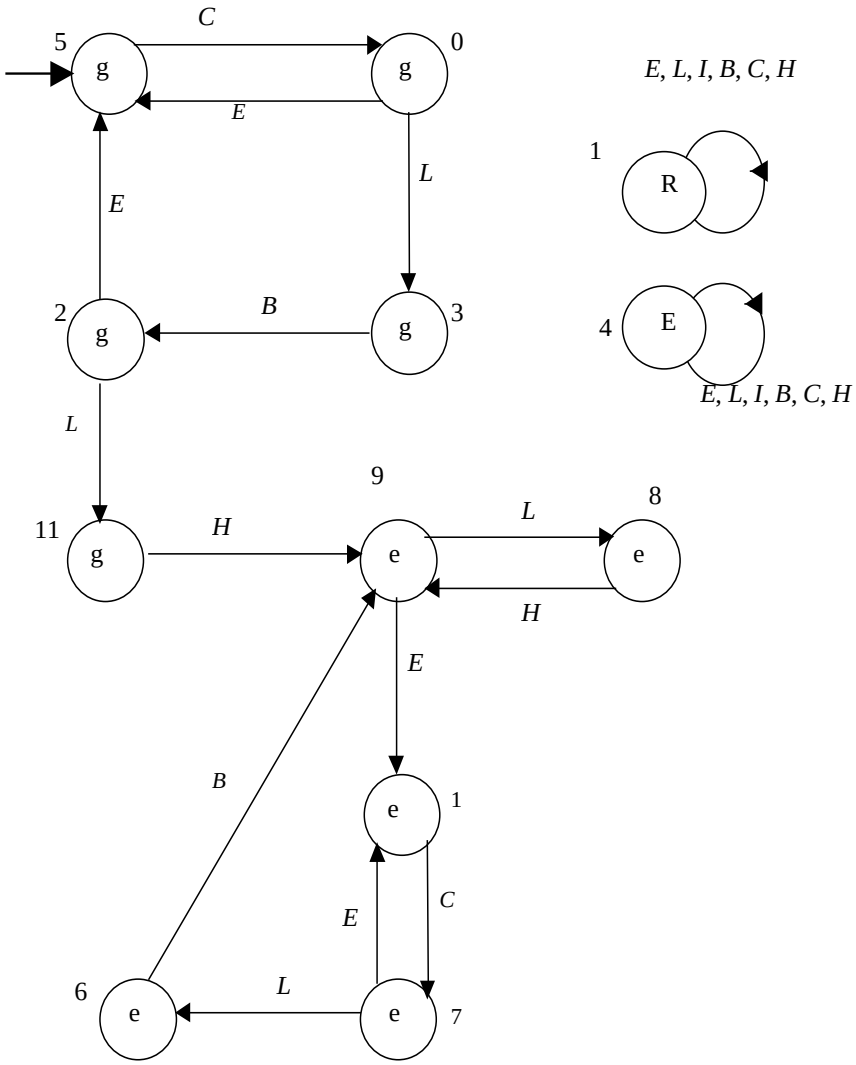


Figure 3.14: Safety process automaton for the solution with limit = 4.

3.4.3 Upgrade Procedure

The divergence-free solutions obtained in Section 3.4.2 are not satisfactory (safety-healthy) in the sense that they are not images of I/O automata through the safety mapping described in section 2.4. In addition to relating to I/O automata, safety-healthy processes can be easily synthesized into circuit implementations, using available tools. Therefore, they correspond to physical systems.

In this section, we describe the steps of the upgrade procedure, which refines processes into processes that are images of I/O automata. However, sometimes a process can not be upgraded to a physically realizable one. In this case, the procedure yields a top process, which, by definition [14], consists of only one escape state. This implies that the problem in hand has no solution. Usually, the upgrade procedure needs to handle three obstacles: inputs that lead to escape states, outputs that lead to reject states, and escape set that is not suffix closed. Therefore, the procedure is done in three steps. In the first step, we handle the inputs that lead to escape states, the second step deals with the problem of outputs leading to reject states, and the third step makes the escape set suffix-closed. In each step, we show that the resulting process refines the input to that step, and we illustrate the step by means of an example.

The procedure accepts process automata defined over a set U^*, where U is an alphabet formed by two disjoint subsets *In* and *Out*. The transformations in the procedure have no effect on the alphabet itself, i.e., inputs remain inputs and outputs remain outputs. What the procedure does is to change how the process, represented by an automaton, views the executions that contain inputs and outputs. Therefore, all the process automata in the procedure have the same input and output sets, *In* and *Out*. The resulting processes are defined in terms of execution languages (enabled traces) and can be represented using process automata.

Step 1. Dealing with escape-inputs.

Input $P1$: **as**($P1$), **at**($P1$).

Output $P2$: **as**($P2$), **at**($P2$).

Escape-inputs are traces that:

- are not prefixes of goal traces and
- include illegal actions controlled by the environment, inputs, that lead from a goal state to an escape state.

As an example of an escape-input traces, consider the trace *CLBLH* in Figure 3.14.

To deal with the problem of escape-inputs, the procedure traces back all the executions that lead to this escape-input until an output is reached. Then, all states that are reachable as a result of executing this output are declared escapes. This guarantees obtaining a safety-healthy process while eliminating escape-inputs.

escape-inputs($P1$) = { $ua \mid u \in$ **prefg**($P1$), $ua \notin$ **prefg**($P1$), $a \in In$, and $ua \in$ **e**($P1$)}
where **e**($P1$) is the escape set of the process $P1$.

The set of traces from U^* that are transformed into escapes is denoted **eliminate**($P1$).
eliminate($P1$) = {$uavw \mid ua \in$ **prefg**($P1$), $a \in Out$, $v \in In^*$, $w \in U^*$, $uav \in$ **escape-input**($P1$) }.

We have $P2 \sqsupseteq P1$ since

as($P2$) = **as**($P1$)\ **eliminate**($P1$)$\Rightarrow$ **as**($P2$) $\subseteq$ **as**($P1$).

at($P2$) = **at**($P1$) $\cup$ **eliminate**($P1$)$\Rightarrow$ **at**($P2$) $\supseteq$ **at**($P1$). Therefore, $P2 \sqsupseteq P1$.

We propose to perform the above transformation on a process automaton, which represents a safety process, state by state as follows:

3- We inspect the states of the automaton until we reach a state, which enables an input that leads to an escape state.
4- Whenever such a state is found, we parse the automaton back until we find the first state that enables an output. This output is considered responsible for the escape-input transition.
5- We declare the state reached by the output and all its successors as escapes.

We use the solution obtained in the previous section, for the limit 2, to illustrate how to fix the problem of the inputs that lead to escape states rather than reject states. Figure 3.13 shows that the transition (1, *B*, 8) leads to an escape state although *B* is an input to the *Coffee-machine*. To handle this problem, we look for the output action that proceeds the occurrence of the escape-input. We find *L* in the transition (5, *L*, 1), so we mark all the states reachable by executing *L* as escapes. The only change in this case is to

transform state 1 into escape. This means that the *Coffee-machine* guarantees not to issue *L* in order to avoid receiving the *B* action. In addition, this transformation marks all the states reachable from state 5 by executing *L* as escapes. This means that state 1 becomes equivalent to a permanent escape state. Figure 3.15 shows the safety process *Coffee-machine* after Step 1 of the upgrade procedure.

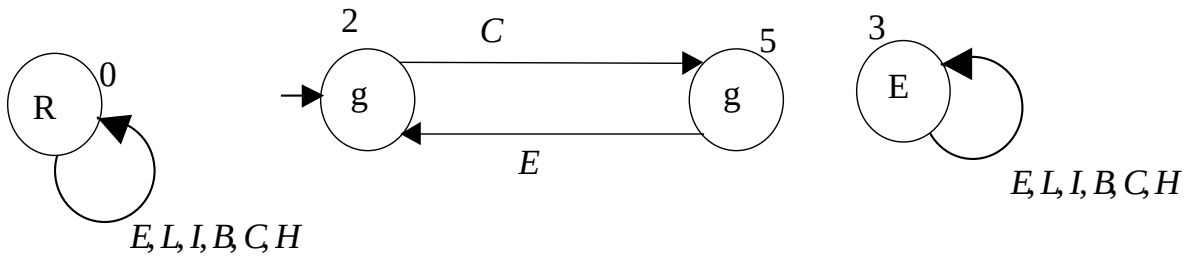


Figure 3.15: The outcome of step 1 of the upgrade procedure. (Missing input transitions lead to state 0. Missing output transitions lead to state 3)

Step 2. Dealing with reject-outputs.

Input *P*2: **as**(*P*2), **at**(*P*2).

Output *P*3: **as**(*P*3), **at**(*P*3).

Reject-outputs are traces that:

- are not prefixes of goal traces and
- include illegal actions executed by the device, outputs, that lead to reject states.

The problem of reject-outputs is solved as follows: the procedure labels all the states as reachable by executing the reject-output as escapes. Therefore, a reject-output will end up leading to a permanent violation committed by the device, a permanent escape, which ensures that the process obtained is a safety-healthy process.

reject-outputs(*P*2)={*ua* | *u*∈ **prefg**(*P*2), *ua* ∉ **prefg**(*P*2), *a*∈ *Out*, and *ua* ∈ **r**(*P*2)}.

r(*P*2) represents the reject set of *P*2.

The set of the words from *U** that are transformed into escapes is denoted **eliminate**(*P*2).

eliminate(*P*2)= {*uav* | *v*∈ *U**, *a*∈ *Out*, *u*∈ **prefg**(*P*2): *ua*∈ **reject-outputs**(*P*2)}.

We have $P3 \sqsupseteq P2$ since

$\mathbf{as}(P3) = \mathbf{as}(P2) \setminus \mathbf{eliminate}(P2) \Rightarrow \mathbf{as}(P3) \subseteq \mathbf{as}(P2)$.

$\mathbf{at}(P3) = \mathbf{at}(P2) \cup \mathbf{eliminate}(P2) \Rightarrow \mathbf{at}(P3) \supseteq \mathbf{at}(P2)$. Therefore, $P3 \sqsupseteq P2$.

We propose to perform the above transformation on a process automaton, which represents a safety process, state by state as follows:

1- We inspect the states of the automaton until we reach a state that enables an output that leads to a reject state.
2- Whenever such a state is found, we declare the state reached by the output and all its successors as escapes.

We apply this step of the upgrade procedure to the outcome of Step 1, Figure 3.15. However, in this process, there are no output actions that lead to rejects. This means that the process remains unchanged after Step 2, Figure 3.15.

Step 3. Making the escape set suffix closed.

Input $P3$: $\mathbf{as}(P3)$, $\mathbf{at}(P3)$.

Output $P4$: $\mathbf{as}(P4)$, $\mathbf{at}(P4)$.

The escape set of a safety process is not suffix-closed when the successors of an escape state, in the corresponding process automaton, are not all escape states. This problem is solved by declaring all the successors of any escape state, in the process automaton that represents the safety process, as escape states themselves.

We denote the traces of $P3$, which lead to a non suffix-closed escape set, by

$\mathbf{Traces\text{-}non\text{-}suffix\text{-}closure}(P2) = \{ uv \,/\, u \in \mathbf{e}(P3) \wedge uv \notin \mathbf{e}(P3) \text{ where } v \in U^* \}$

The set of the words from U^* that are transformed into escapes is denoted $\mathbf{eliminate}(P3)$.

$\mathbf{eliminate}(P2) = \{uav \,/\, u \in \mathbf{e}(P3), v \in U^*, \text{and } a \in U : ua \notin \mathbf{e}(P3) \}$.

We have $P4 \sqsupseteq P3$ since

$\mathbf{as}(P4) = \mathbf{as}(P3) \setminus \mathbf{eliminate}(P3) \Rightarrow \mathbf{as}(P4) \subseteq \mathbf{as}(P3)$.

$\mathbf{at}(P4) = \mathbf{at}(P3) \cup \mathbf{eliminate}(P3) \Rightarrow \mathbf{at}(P4) \supseteq \mathbf{at}(P3)$. Therefore, $P4 \sqsupseteq P3$.

We propose to perform the above transformation on a process automaton, which represents a safety process, state by state as follows:

1- We inspect the states of the automaton until we reach an escape state whose successors are not all escape states.
2- Whenever such a state is found, we declare all its successors as escape states.

We apply this step of the upgrade procedure to the outcome of Step 2, Figure 3.15. However, in this process, there are no output actions that lead to rejects. This means that the process remains unchanged after Step 2.

As another application of the procedure, we illustrate how to upgrade the solution obtained in the last section for the limit = 4, Figure 3.14. First, the procedure checks for the input actions that lead to escape states. There is only one transition that shows such abnormality, (11, *H*, 9). In this transition, *H* is an input to the *Coffee-machine*; however, it leads to an escape state. In order to fix this problem, the procedure tracks the last output that led to this escape-input. There is *L* in the transition (2, *L*, 11). So, all the states reachable by executing *L* are declared to be escapes. This means that the *Coffee-machine* guarantees not to issue *L* in order to avoid receiving the *H* action. In addition, this transformation means that state 11 becomes equivalent to a permanent escape state. Figure 3.16 shows the safety process *Coffee-machine* after Step 1 of the upgrade procedure.

Next, we apply the second step of the procedure to fix the problem of any output actions that lead to reject states. However, there are none in the process automaton of Figure 3.16. Therefore, the outcome of the Step 2 for the case when limit = 4 is the process

automaton in Figure 3.16. Similarly, Step 3 won't introduce any changes to the process automaton in Figure 3.16.

Although not all the problems need to exist simultaneously in a safety process automaton, each of the three upgrades guarantees a suffix-closed safety process as a result. However, in the presence of escape-inputs, one needs to apply Step 1 of the procedure to make sure that conflicts between input and output violations do not exist in the final safety-healthy process. In the case of the *Coffee-machine*, we only needed to execute Step 1 of the procedure in order to achieve the safety-healthiness of the automaton.

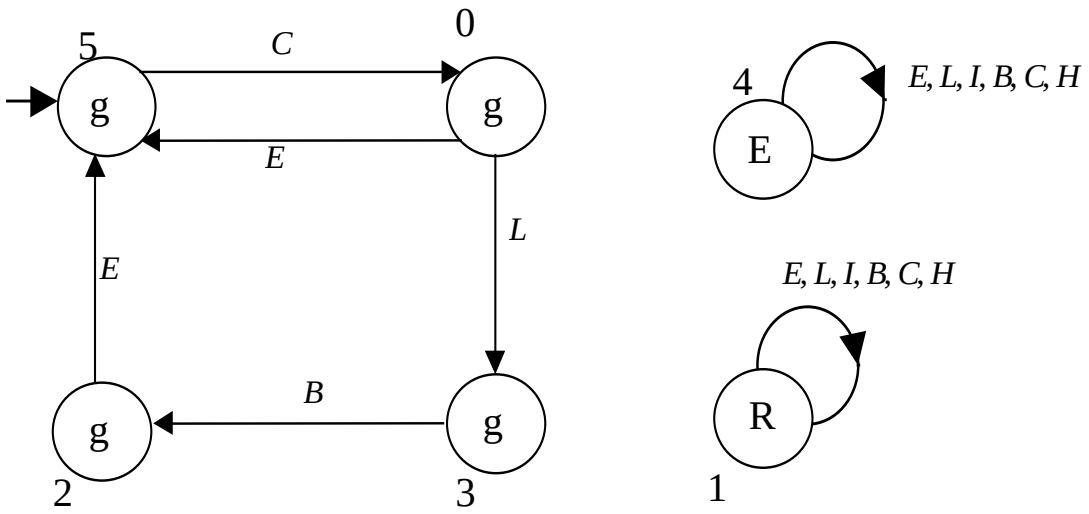


Figure 3.16: The solution for limit=4 after the first step of the procedure. (Missing input transitions lead to state 1. Missing output transitions lead to state 4)

On the other hand, we use an example to illustrate how the upgrade procedure fails. Figure 3.17 shows a safety process *p* represented by a process automaton with *i* and *o* being the input and output, respectively. This process shows an escape-input execution (0, *i*, 2). We apply Step 1 of the upgrade procedure. We trace back the execution wich ends with an escape-input until the first output is reached. In this case, it is the empty word in the initial state. Then we declare all the states reached by executing the empty word as escapes. This leaves us with a process automaton whose all states are escape

states. As a result, we can merge all the states into one escape state, which enables all the actions to cause self-loops. Such process does not correspond to any physical system.

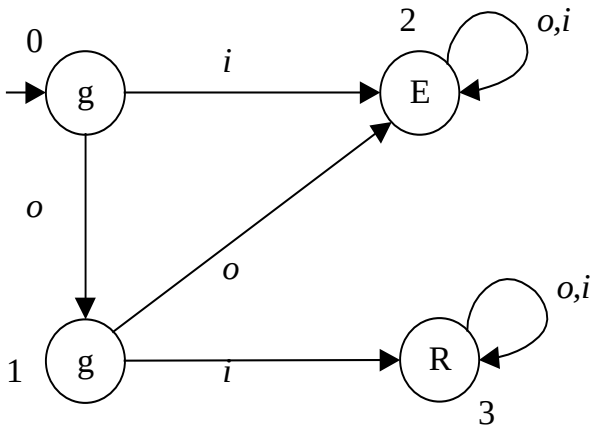


Figure 3.17: Process automaton of a process that does not correspond to a physical system.

3.4.4 Divergence-freedom

In the previous section, we saw that our method to obtain a divergence-free solution for asynchronous equations might yield processes with some technical problems. We described the upgrade procedure, which fixes these problems. To prove the correctness of this method; first, we show that the obtained process is a solution to the problem, and that it yields a weakly-divergence-free system. Next, we show that the solution can be "upgraded" to obtain a strongly-divergence-free system.

The following, Proposition 3.3, shows that our method leads to a solution that guarantees weakly-divergence-freedom in the system. We show that using the limiting process $llock_n$ still leads to a solution for the problem. Next, we show that the obtained solution, with the help of the limiting process, insures weakly-divergence-freedom in the system formed by the *Waiter*, the obtained *Coffee-machine*, and their environment, i.e. *Client*.

Proposition 3.3. Consider a set of processes $\{p, q, r\}$ defined over U^*, an alphabet $A \subseteq U$, and an integer n. Let $llock_n$ be a process over U^* such that

$\mathbf{at}(llock_n) = U^*$ or $\mathbf{r}(llock_n) = \varnothing$

$\mathbf{as}(llock_n) = ((U \setminus A)\ (\varepsilon \cup A^n))^*$ or $\mathbf{e}(llock_n) = \overline{((U \setminus A)\ (\varepsilon \cup A^n))^*}$.

If $r \sqsupseteq -(-p \times -llock_n \times q)$, then

$p \sqsubseteq q \times r$ and

$\{-p, q, r\}$ is weakly-divergence-free with respect to A.

Proof:

$p \sqsubseteq q \times r$

Note that

$-p \times -llock_n = (\mathbf{as}(-p) \cap \mathbf{as}(-llock_n), \mathbf{at}(-p) \cap \mathbf{at}(-llock_n) \cup \overline{\mathbf{as}(-p) \cap \mathbf{as}(-llock_n)})$

$= (\mathbf{as}(-p), \mathbf{at}(-p) \cap \mathbf{at}(-llock_n) \cup \overline{\mathbf{as}(-p)}) \sqsubseteq -p$

since $\mathbf{at}(-p) \cap \mathbf{at}(-llock_n) \subseteq \mathbf{at}(-p)$

and $\overline{\mathbf{as}(-p)} = \mathbf{e}(-p) \subseteq \mathbf{at}(-p)$.

Then $-p \times -llock_n \times q \sqsubseteq -p \times q$ [14]

So $r \sqsupseteq -(-p \times -llock_n \times q) \sqsupseteq -(-p \times q)$

$\Rightarrow r \sqsupseteq -(-p \times q) \Rightarrow p \sqsubseteq q \times r$ [14].

$\{-p, q, r\}$ is weakly-divergence-free with respect to A

$-(-p \times -llock_n) \sqsubseteq q \times r$ since $llock_n$ is a robust process [14]

$\Rightarrow -p \times -llock_n \times q \times r$ is a robust process [14].

$\Rightarrow llock_n \sqsubseteq -p \times q \times r$

so, $\forall\ l = v\xi$ such that $v \in U^*$ and $\xi \in A^\omega$

$\exists\ t = vA^{n+1}$ and

$\forall\ u$ such that $t \le u \le l$: $u \in \mathbf{e}(llock_n) \Rightarrow u \in \mathbf{e}(-p) \cup \mathbf{e}(q) \cup \mathbf{e}(r)$

$\Rightarrow \exists\ h \in \{-p, q, r\} : u \in \mathbf{e}(h)$. □

Propositions 3.1, 3.2, and 3.3 prove the correctness of our procedure for obtaining strongly-divergence-free solutions to asynchronous equations.

By Proposition 3.3, process q_1 = -(- *Coffee-shop* × - $llock_n$ × *Waiter*) is a solution to the asynchronous equation *Coffee-shop* ⊑ *Waiter* × q_1, and has weak-divergence-freedom. Proposition 3.1 proves that q_2, which refines q_1 also has weak-divergence-freedom. At the same time q_2 is a solution to the asynchronous equation by Theorem 2.14 in [14]. Therefore, Proposition 3.1 guarantees that the upgrade procedure preserves weak-divergence-freedom. Finally, we use Proposition 3.2 to prove strong-divergence-freedom, because the upgraded process q_2 is safety-healthy.

Thus, a strongly-divergence-free solution always exists. However, the solution obtained could be a process for which all executions are escapes; this is because the empty word may be chipped-off as a result of the first step of the upgrade algorithm. This occurs, for example, if an input action is enabled from the initial state and leads to an escape. Hence, no physical device is guaranteed to correspond to such process. In other words, the upgrade procedure allows us to detect the absence of a solution to the supervisory control problem by yielding an empty solution, which consists of a single escape state.

3.4.5 Solution in Finalization Processes

Again, we follow the same procedure to obtain the divergence-free solution for the asynchronous equation in terms of finalization processes. We use the same limiting processes to achieve both limits. The limiting process satisfies both safety and finalization properties. The results are shown in Figure 3.18 and 3.19 for the limit = 2 and limit = 4, respectively.

The solutions obtained in terms of finalization processes do not guarantee divergence-freedom in the system. The reason for this is that they do not force divergences to be labeled as escapes in the system formed by the *Waiter*, *Coffee-machine*, and their environment. For example, the solution in Figure 3.18 has the sequence (*H*, *L)* in its accessible set. On the other hand, the *Waiter* finalization process automaton (Figure 3.20) does not avoid the same divergence. Therefore, the system resulting from the composition of the *Waiter, Coffee-machine,* and the environment is not guaranteed to

avoid the divergences as well. This means that the system is not weakly-divergence-free, (Definition 3.2). Therefore, strongly-divergence-freedom is not guaranteed either.

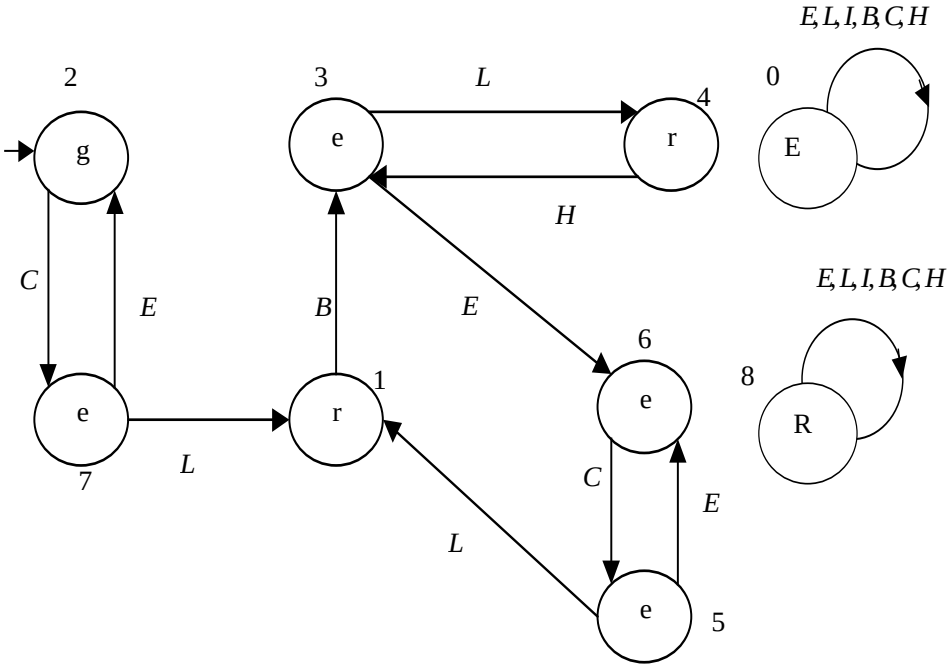


Figure 3.18: Finalization process automaton for the solution with limit = 2. (Missing input transitions lead to state 0. Missing output transitions lead to state 8)

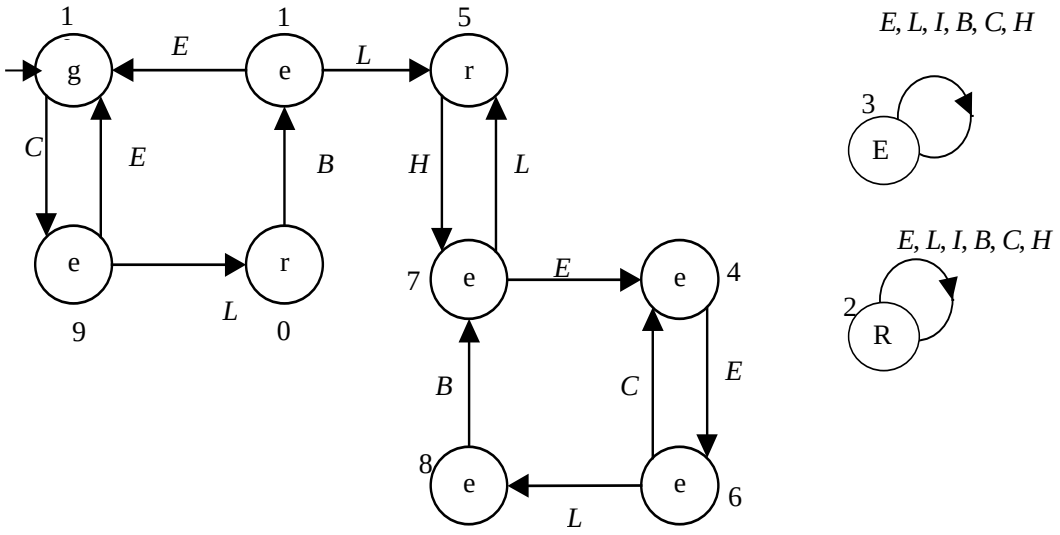


Figure 3.19: Finalization process automaton for the solution with limit = 4. (Missing input transitions lead to state 2. Missing output transitions lead to state 3)

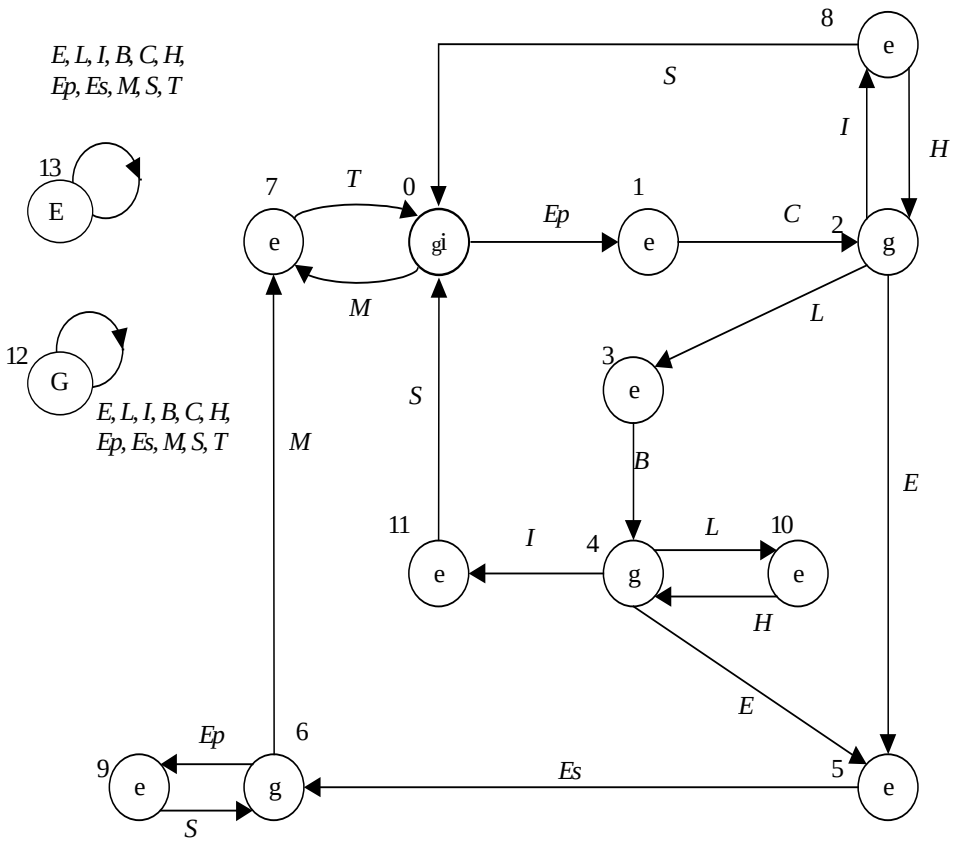


Figure 3.20: Finalization process automaton *Waiter*. (Missing input transitions lead to state 12. Missing output transitions lead to state 13)

In the following section, we describe how to transform processes into I/O automata in order to obtain the solution for the asynchronous equation in terms of I/O automata. In particular, we describe the transformation in terms of safety processes only. The main reason is that we need a transformation that serves reaching a divergence-free solution in I/O automata. Since the solution obtained in terms of finalization processes was shown not to be divergence-free, we only consider mapping of safety processes into I/O automata.

3.5 Backmapping Processes to I/O Automata

In our effort to solve asynchronous equations of I/O automata, we have developed semantics-mappings from I/O automata to process spaces. This helps us make use of the

method we described in terms of process spaces to find solutions and divergence free solutions for asynchronous equations. At the same time, the mapping allows the use of FIREMAPS to automate the process of solving the asynchronous equation problem. However, we still have to reconstruct the corresponding I/O automaton for the solution obtained as a process. To achieve this, we devise a transformation from process spaces to I/O automata. We will call this transformation the backmapping from process spaces to I/O automata.

The formal definition of I/O automata necessitates the distinction between automaton actions as inputs, outputs, and internal actions. Therefore, we impose on any process to be transformed via the backmapping to an I/O automaton that an action signature, which shows the distinction between the process actions, be clearly defined.

3.5.1 Backmapping Safety Processes to I/O Automata

We consider a safety process *pr* that could be identified by the three sets **r**(*pr*), **g**(*pr*), and **e**(*pr*). The image of *pr* in I/O automata through the backmapping is an automaton $\alpha = (S, I, \Sigma, \lambda, \delta)$ where

S is a set of states.
I is a nonempty set of initial states
Σ is an action signature.
λ is a transition relation.
δ is an equivalence relation.

In other words, our goal is to map a safety process into an I/O automaton α whose image through the safety mapping of Section 2.4 is the given process.

We describe the backmapping in terms of traces of an I/O automaton. Thus, we consider an I/O automaton α represented by:
The set of traces of the automaton: *traces*(α).

The set of disabled words $D(\alpha)$ = ($acts(\alpha)$)* - $traces(\alpha)$, where $acts(\alpha)$ is the set of actions of α.

Now the backmapping is defined in terms of the two sets:

$Traces(\alpha)$ = $\mathbf{g} \cup \mathbf{r}$ = {all traces of *pr* that are goals and all traces that are rejects},

$D(\alpha)$ = {all escapes traces of *pr*}.

In the following, we propose how to apply the above transformations to process automata, which represent safety processes, in order to obtain the corresponding I/O automata:

All goal states are mapped into reachable states of the I/O automaton.

The transitions that lead to the permanent escape state are disabled.

The initial state is transformed into an initial state in the I/O auotmaton.

The permanent reject state is transformed into a trap state which enables both inputs and outputs.

Finally, the action signature of the I/O automaton is obtained by partitioning the alphabet of the safety process into two disjoint sets:

In, which contains all the actions considered as inputs to the process, and

Out, which includes all the actions considered as outputs to the process.

It is clear from the above partition that the produced I/O automata, by the backmapping, do not have any internal actions.

On the other hand, as a limitation of the backmapping, we restrict the safety processes that can be transformed into I/O automata to the class of safety-healthy processes.

3.5.2 Example

As an example, we consider mapping the process automata in Figure 3.15 and 3.16 to I/O automata.

We transform the safety processes in Figure 3.15 and Figure 3.16 into I/O automata. To do so, we follow the backmapping algorithm, and the resulting automata are shown in Figure 3.21. As noted, the missing input actions (*B*, *C*, and *H*) from the figure lead to the trap states.

The most important property of the backmapping that we investigate in this study is the conservation of divergence freedom. In process spaces, divergence freedom was described in terms of infinite words that are avoided by at least one process in a system. For this to happen, the actions in the avoided infinite words must be outputs of the process that avoids them. This means that the process treats these words as escapes. Through the backmapping, the escape traces of a process are disabled when obtaining the corresponding I/O automaton. Therefore, the obtained I/O automaton guarantees to avoid the infinite words that cause the divergence. Hence, the divergence freedom in process spaces corresponds to divergence freedom in I/O automata.

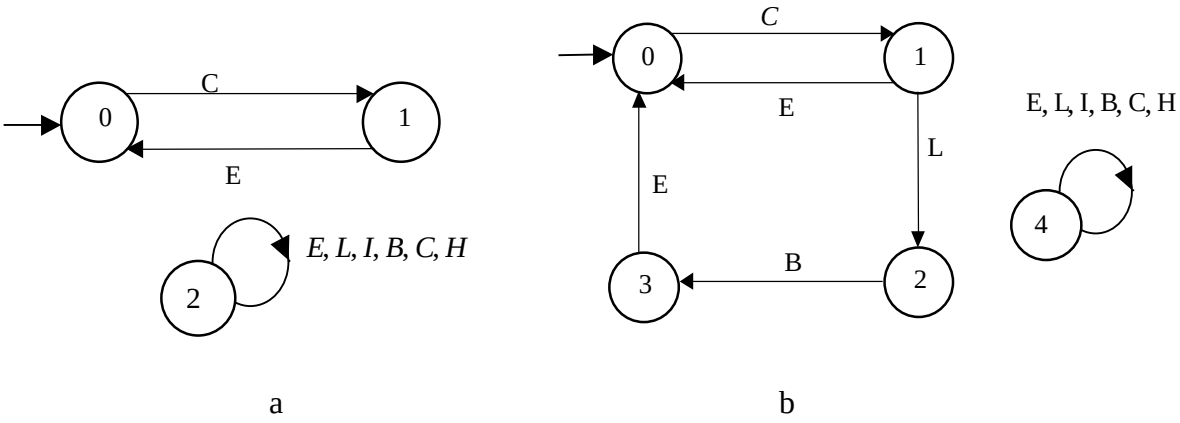


Figure 3.21: I/O automata *Coffee-shop*: a) solution for limit =2 and b) solution for limit=4. (Missing input transitions lead to states 2 and 4 respectively.)

It is clear from Figure 3.21 that the I/O automata obtained are divergence-free with respect to interactions with the *Waiter* automaton. For example, the two automata avoid infinite sequences such as (*L*, *H*). This guarantees that the I/O automaton *Coffee-shop*1, formed by the composition of each of these solutions with the I/O automaton *Waiter*

(Figure 3.5), avoids the divergences. The automaton *Coffee-shop*1 has the following action signature:

In(*Coffee-shop*1) = {*Ep, M*} (*Espresso please*, *Money*).
Out(*Coffee-shop*1) = {*Es, S, T, M, L, E, I, C, B, H*} (*Espresso served*, *Sorry*, *Thanks*, *Lamp*, *Espresso*, *Idle*, *Coin*, *Button*, *Hit*).
Int(*Coffee-shop*1) = ∅.

Notice that the action signature of *Coffee-shop*1, mainly the *In* and *Out* sets, differ from the action signature of the automaton *Coffee-shop* (Figure 3.4). This means that the set of external traces of *Coffee-shop*1 is not necessarily included in the set of external traces of *Coffee-shop*. As a result, implementation between the two automata cannot be established (Section 2.1.6). However, hiding (Section 2.1.7) the output actions of *Coffee-shop*1, which are not observed by *Coffee-shop* (*C, L, E, B, H, I*), transforms the action signature of *Coffee-shop*2 into:

In(*Coffee-shop*2) = {*Ep, M*}.
Out(*Coffee-shop1*) = {*Es, S, T*}.
Int(*Coffee-shop*2) = {*L, E, I, C, B, H*}.

The new action signature shows that the *In* and *Out* sets of *Coffee-shop*2, after hiding, are equal to the *In* and *Out* sets of *Coffee-shop*. This allows comparing the sets of external traces of both automata to check for implementation. For example, consider the automaton in Figure 3.22. It represents the composition of *Waiter* and *Coffee-machine* for limit = 2. The figure shows that when the communications between *Waiter* and *Coffee-machine* (*C* and *E*) are hidden, *Coffee-shop*2 becomes equivalent to *Coffee-shop* since implementation holds in both directions between the two automata. The same argument is used to show that the coffee shop version produced by composing the *Waiter* with the *Coffee-machine* for limit = 4 implements the original *Coffee-shop*.

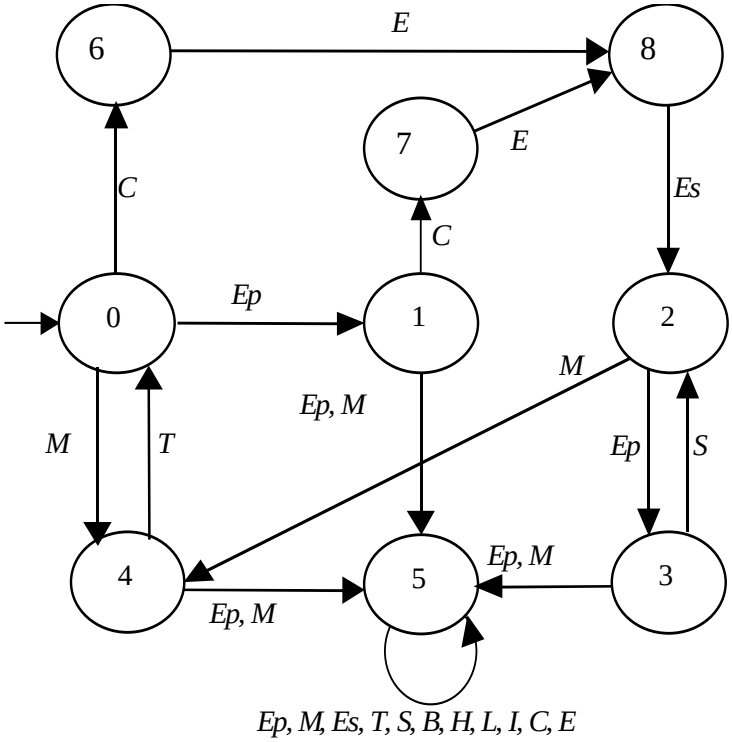


Figure 3.22: The I/O automaton *Coffee-shop*1 for the limit = 2.

Chapter 4

Divergence-Free Protocol Converters

In this chapter, we apply our method to solve asynchronous equations (from chapter 3) to design protocol converters. Protocol converters, also called gateways, are used to interface heterogeneous network systems that use different protocols. We demonstrate how to use our method to obtain general and divergence-free solutions for such problems. In addition, we derive an asynchronous implementation of the obtained converter using an automated synthesis tool.

4.1 Protocol Conversion

The fast emergence of computer and network systems in industrial applications has led to versatility in the designs of such systems. Combined with the principle of modularity, the tendency for designing application specific systems has caused a growing demand for internetworking (connecting several network systems together). However, the absence of a universal standard that unifies the underlying protocols, interconnecting heterogeneous computer networks is still a major problem to overcome. This problem remains a motivation to design gateway devices (protocol converters) that convert signaling of one communication protocol into another. One of the important problems in the area of the protocol conversion is the avoidance of divergence, i.e., cycles of internal transitions in the communication system. In gateway circuitry, an example of divergence is the switching activities due to clock transitions in a circuit, while the circuit is idle. Switching activities usually increase power consumption, which is often undesirable, e.g., for CMOS implementations in wireless communication applications.

Several techniques have been devised to solve the protocol conversion problem using the supervisory control theory [20], which derives a specification for a missing part of a system from the overall service specification of that system and the known part of its implementation; see for instance [5, 8, 18, 19]. The available techniques can be classified as [22]:

1- Top-down techniques, which use a service specification of the interconnected systems combined and solve for a converter by means of an asynchronous equation [19]. Examples of such approaches are presented in [1] and [8].

2- Bottom-up techniques, which depend mainly on heuristically guessing a partial specification of the desired converter and solving for the rest using the specification of the existing protocols. [23] and [17] contain examples of such techniques.

The top down techniques are more popular and more precise [22]. As opposed to bottom up approaches, top down methods show no need to verify the results obtained against the overall service specification. In addition, coming out with a heuristic partial specification of the converter, in bottom up techniques, might be a very complex and tedious task.

Our method, using an asynchronous equation, to design gateways devices follows the main guidelines of the top down approach. In other words, we synthesize a converter based on the specifications of the two protocols involved, and of the overall service that they should provide when combined. This allows us to determine the general solution to the equation that is the loosest specification of the protocol converter that meets the given overall service specification. A particular converter can always be obtained by restricting the behavior of the general solution. At the same time, the method yields divergence-free protocol converters; i.e., converters that guarantee avoidance of divergence in the interconnected system.

4.2 Problem Formulation

A communication system usually consists of a sending part and a receiving part that follow a specific protocol to exchange data. A mismatch occurs when two systems with heterogeneous protocols try to establish communications among them. The solution to the mismatch problem consists of designing a converter that acts as a translator between the sender and the receiver while respecting the overall service specification that describes the behavior of the composite communication system in response to external excitations. In other words, the specification expresses how the communication system should behave in response to requests of its users considered as the environment of the system. We formulate the problem in terms of an asynchronous equation as follows: given the overall service specification of a communication system, a sender, and a receiver, we need to find a converter whose composition with the sender and receiver meets the specification of the overall service after hiding internal communications.

4.3 Case Study

As an example, we consider the problem of designing a protocol converter to interface two heterogeneous entities: an *Alternating-Bit* (AB) sender and a *Non-Sequenced* (NS) receiver. This problem is adapted from [8].

An alternating bit (AB) protocol based communication system is composed of two processes, a sender and a receiver, that communicate over a half duplex channel that can transfer data in either directions, but not simultaneously. Each process uses a control bit called the alternating bit, whose value is updated by each message sent over the channel in either way. Acknowledgement is also based on the alternating bit concept. To each message received by either process in the system corresponds an acknowledgment message that depends on the bit value. If the acknowledgement received by a process does not correspond to the message it has sent originally, the message is sent again until the correct acknowledgement is received. On the other hand, a communication system is said to be non-sequenced (NS) when no distinction is made among the consecutive messages received or their corresponding acknowledgements. This means that neither

messages nor their acknowledgements are distinguished by any flags such as the alternating bit.

Figure 4.1 shows the block diagram of the composite system. Each entity is represented by a rectangle with incoming and outgoing labeled arrows to indicate inputs and outputs, respectively. The sender consists of an AB "protocol sender" (*PS*) and an AB "protocol channel" (*PC*). Meanwhile, the receiving part includes an NS "protocol receiver" *PR*. The converter *X* must interface the two mismatched protocols and guarantee that its composition with *PS*, *PC*, and *PR* refines, the "service specification" (*SS*) of the composite system. The events *Acc* (*Accept*) and *Del* (*Deliver*) represent the interface of the communication system with its environment (the user).

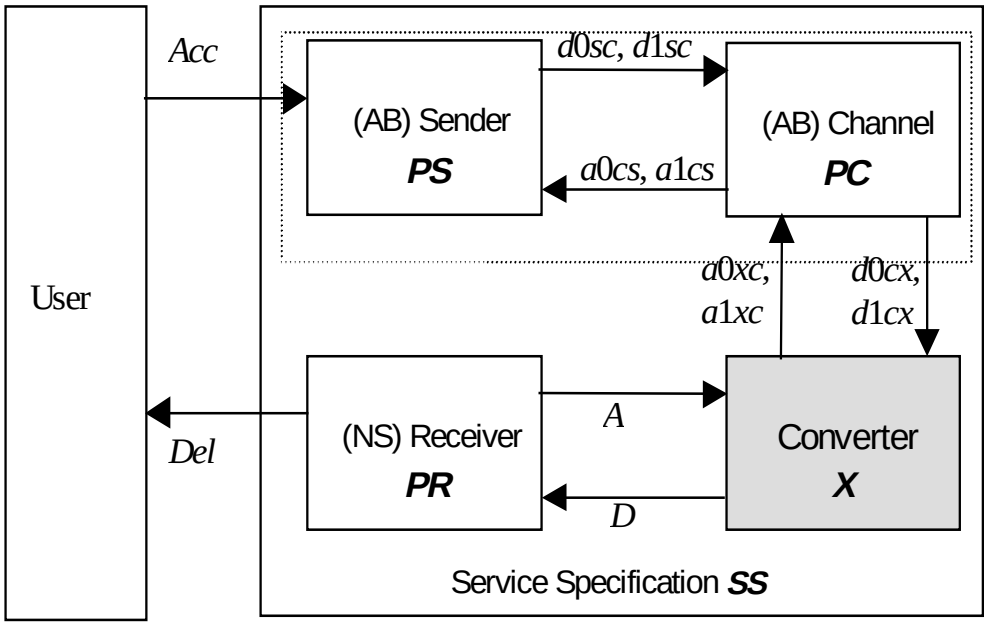


Figure 4.3: Block diagram of the interconnected system.

In Figure 4.1, *PS* and *PC* are grouped together since they constitute one entity, the sending part. The converter *X* translates the messages delivered by the sender *PS* (following the alternating bit protocol) into a format that the receiver *PR* understands (following the non-sequenced protocol). For example, acknowledgement messages *A* delivered to the converter by the receiver on the channel, are transformed into

acknowledgments of the alternating bit protocol (*a*0*xc* to acknowledge a 0 bit and *a*1*xc* to acknowledge a 1 bit) and passed to the sender through the channel.

4.3.1 Original Model

In order to model the behavior of the entities in our example, we follow [8] and use the same specifications. However, in [8], the models described are not input complete, i.e., input events are not enabled in every state. For example, in the protocol sender, it is not specified how to deal with two consecutive messages (*Acc*) from the environment. Although a suggestion is made in [8] to complete the specification of the input events using a trap state, there was no clear definition of this trap state or of the behavior of the system after reaching it. Therefore, we follow our procedure from Section 3.1 in order to build the I/O automata for the entities in our example. We complete the specification of each model by letting the missing input events lead to a trap state from which there is no way to resume normal execution of actions, but the automaton can issue output events arbitrarily. For example, in Figure 4.2 a, state 6 is a trap state for the I/O automaton *PS*.
For each I/O automaton, we define the corresponding action signature as follows.

PS:
In (*PS*) = {*Acc*, *a*0*cs*, *a*1*cs*}.
Out (*PS*) = {*d*0*sc*, *d*1*sc* }.
Int (*PS*) = ∅.

PR:
In (*PR*) = {*D*}.
Out (*PR*) = {*Del*, *A* }.
Int (*PR*) = ∅.

PC:
In (*PC*) = {*a*0*cx*, *a*1*cx*, *a*0*cs*, *d*0*sc*}.
Out (*PC*) = {*a*0*cs*, *a*1*cs*, *d*0*cx*, *d*1*cx* }.
Int (*PC*) = ∅.

SS:
In (*SS*) = {*Acc*}.
Out (*SS*) = {*Del*}.
Int (*SS*) = ∅.

The I/O automata that represent *PS*, *PC*, *PR*, and *SS* are shown in Figure 4.2 a, b, c, and d, respectively.

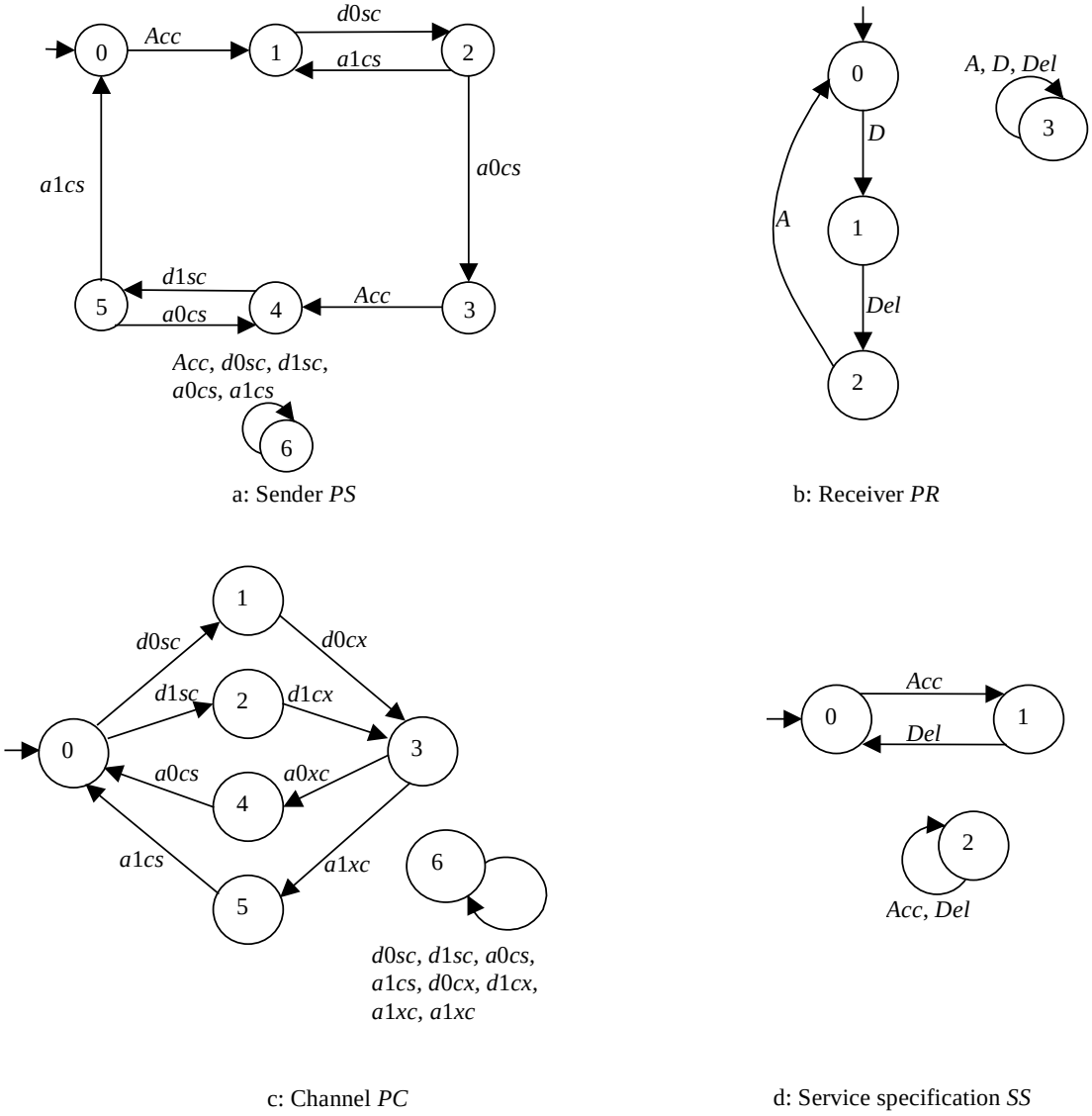


Figure 4.4: I/O automata *PS*, *PR*, *PC*, and *SS*. (Missing input transitions in *PS*, *PR*, *PC*, and *SS*, lead to states 6, 3, 6, and 2, respectively.)

Next, we map each I/O automaton into a corresponding safety process. To do this, we use our mapping from I/O automata to safety processes. We represent the safety processes corresponding to the I/O automata *PS*, *PC*, *PR*, and *SS* by process automata.

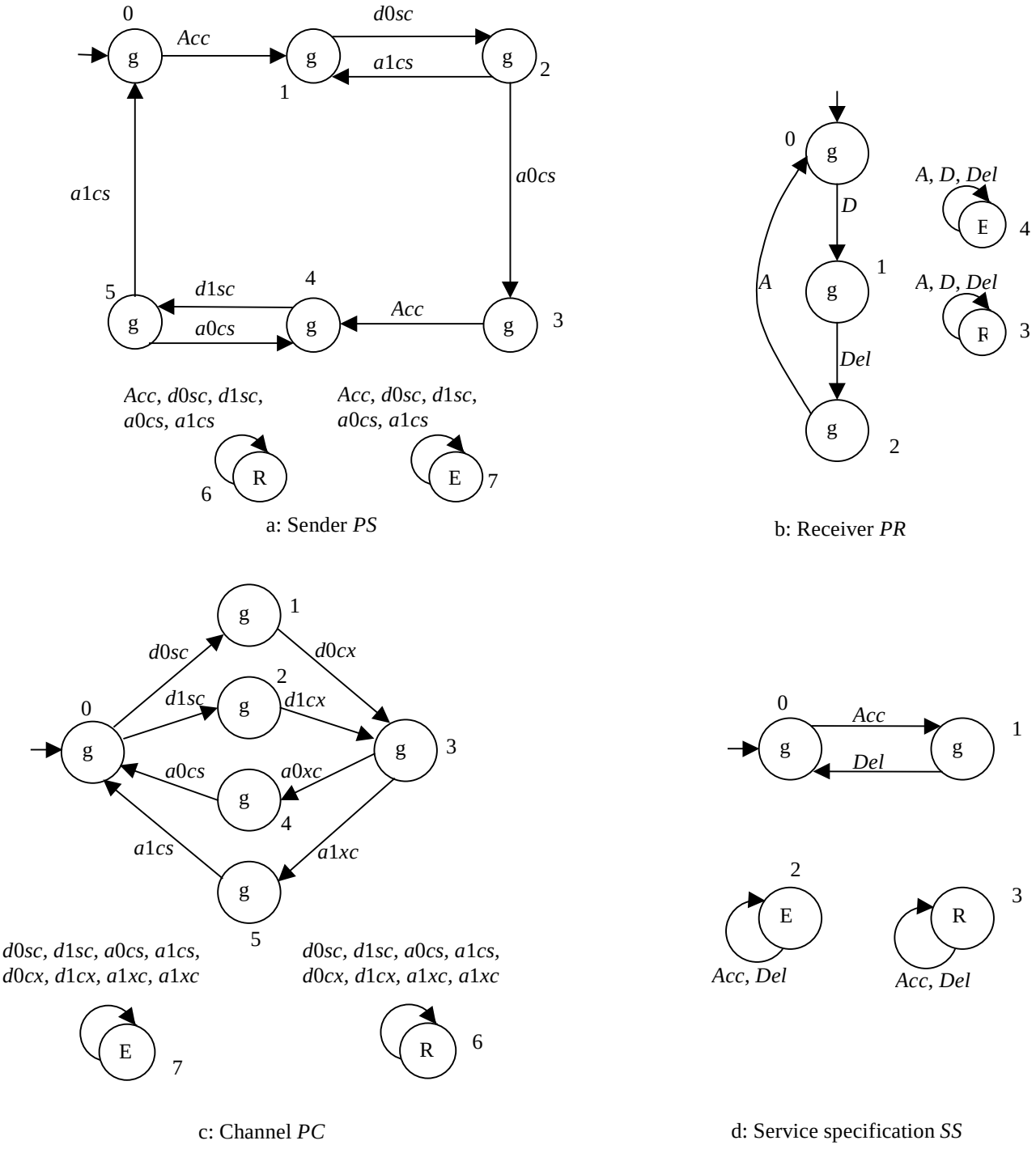


Figure 4.5: Process automata *PS*, *PR*, *PC*, and *SS*. (Missing input transitions in *PS*, *PR*, *PC*, and *SS*, lead to states 6, 3, 6, and 2, respectively. Missing output transitions in *PS*, *PR*, *PC*, and *SS*, lead to states 7, 4, 7, and 3, respectively. External actions to each process automaton produce self-loops at every state of that automaton.)

The resulting safety process automata that represent *PS*, *PR*, *PC*, and *SS* are shown in Figure 4.3 a, b, c, and d, respectively. Notice that the disabled outputs in the I/O automata

are modeled as escapes, output violations, in the safety processes. Also notice that the trap states in the I/O automata are modeled as reject states in the safety processes. This is done to indicate that the transitions leading to these trap states are violations committed by the environment. For legibility, the transitions to the trap states (escape and reject) are not shown in the figures. However, the missing transitions due to input actions lead to the reject states, and the missing transitions due to output actions lead to the escape states, respectively. For example, state 6 represents an escape state in Figure 4.3 a. This state is reached by traces that include illegal outputs such as *ACC*, *d*0*sc*, *d*0*sc*. On the other hand, state 7 is a reject state. This state is the image of the trap state in the I/O automaton *PS*. It is reached by the traces that include "illegal inputs".

4.3.2 Absence of a Solution in the Original Model

Now we can solve the equation for the converter *X* using generic concurrency operations implemented in FIREMAPS. The following is a description of the procedure applied to our working example:

1. Obtain the process automaton *Known* from the product of all the known modules in the system:

a) Obtain the product of *PS*, *PC*, and *PR*,

b) Hide *d*0*sc*, *d*1*sc*, *a*0*cs*, and *a*1*cs*. These actions are invisible to the missing converter.

c) Determinize and minimize the obtained process since hiding introduces non-determinism in the resulting process *Known*. Figure 4.4 (left) shows a part of the non-deterministic automaton, *Known*, as a result of hiding. Notice that this automaton has two initial states, indicated by the incoming arrows, and that the same event might lead to more than one state. Figure 4.4 (right) shows how the determinization of the process *Known* is carried. Notice that states 3 and 4, which have qualifiers g and e, respectively, are replaced by one goal state.

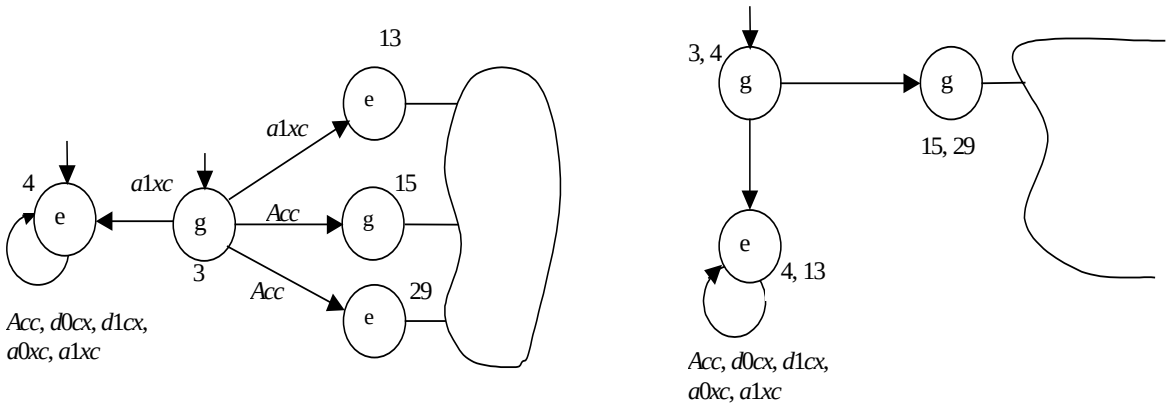


Figure 4.6: Determinization of the process automaton *Known*.

2. Determine the product of *Known* with the reflection of *SS*. This yields the environment of the missing converter *X*.
3. Hide the actions, which are irrelevant to the converter. *X* doesn't observe *Acc* and *Del*.
4. Determinize and then minimize the obtained process to avoid the non-determinism that results from hiding.
5. Reflect the result to obtain the solution, *X*.

Generally speaking, applying the hide operation after each composition (Step 1 and 4) has the same effect as hiding the irrelevant actions once in Step 4. However, we chose to execute hiding twice, and determinize twice, in Step 1 and Step 4. This allows us to reduce the computation time of FIREMAPS.

Figure 4.5 shows the script file used to feed the FIREMAPS tool with the description of the problem. The script file includes, in addition to process descriptions, the following two lines of commands, which summarize the equation solving procedure.

```
= known (mdm) \ * * PS PC PR (ar) 4: d0sc d1sc a0cs a1cs # step 1.
= X - (mdm) \ * known - spec (ar) 2: acc del # steps 2, 3, 4, 5, and 6.
```

```
                                          = PS (pr)
5 actions; 8 states; 40 edges;
actions: acc, d0sc, d1sc, a0cs, a1cs;
states: 0 sti, 1 st, 2 st, 3 st, 4 st, 5 st, 6 t, 7 s;
edges:
from 0: acc 1, d0sc 6, d1sc 6, a0cs 7, a1cs 7;
from 1: acc 7, d0sc 2, d1sc 6, a0cs 7, a1cs 7;
from 2: acc 7, d0sc 6, d1sc 6, a0cs 3, a1cs 1;
from 3: acc 4, d0sc 6, d1sc 6, a0cs 7, a1cs 7;
from 4: acc 7, d0sc 6, d1sc 5, a0cs 7, a1cs 7;
from 5: acc 7, d0sc 6, d1sc 6, a0cs 4, a1cs 0;
from 6: acc 6, d0sc 6, d1sc 6, a0cs 6, a1cs 6;
from 7: acc 7, d0sc 7, d1sc 7, a0cs 7, a1cs 7.

= channel (pr)
8 actions; 8 states; 64 edges;
actions: a0cs, a1cs, d0sc, d1sc, d0cx, d1cx, a0xc, a1xc;
states: 0 sti, 1 st, 2 st, 3 st, 4 st, 5 st, 6 t, 7 s;
edges:
from 0: a0cs 6, a1cs 6, d0sc 1, d1sc 2, d0cx 6, d1cx 6, a0xc 7, a1xc 7;
from 1: a0cs 6, a1cs 6, d0sc 7, d1sc 7, d0cx 3, d1cx 6, a0xc 7, a1xc 7;
from 2: a0cs 6, a1cs 6, d0sc 7, d1sc 7, d0cx 6, d1cx 3, a0xc 7, a1xc 7;
from 3: a0cs 6, a1cs 6, d0sc 7, d1sc 7, d0cx 6, d1cx 6, a0xc 4, a1xc 5;
from 4: a0cs 0, a1cs 6, d0sc 7, d1sc 7, d0cx 6, d1cx 6, a0xc 7, a1xc 7;
from 5: a0cs 6, a1cs 0, d0sc 7, d1sc 7, d0cx 6, d1cx 6, a0xc 7, a1xc 7;
from 6: a0cs 6, a1cs 6, d0sc 6, d1sc 6, d0cx 6, d1cx 6, a0xc 6, a1xc 6;
from 7: a0cs 7, a1cs 7, d0sc 7, d1sc 7, d0cx 7, d1cx 7, a0xc 7, a1xc 7.

= PR (pr)
3 actions; 5 states; 15 edges;
actions: d, del, a;
states: 0 sti, 1 st, 2 st, 3 t, 4 s;
edges:
from 0: d 1, del 3, a 3;
from 1: d 4, del 2, a 3;
from 2: d 4, del 3, a 0;
from 3: d 3, del 3, a 3;
from 4: d 4, del 4, a 4.

= SS (pr)
2 actions; 4 states; 8 edges;
actions: acc, del;
states: 0 sti, 1 st, 2 t, 3 s;
edges:
from 0: acc 1, del 2;
from 1: acc 3, del 0;
from 2: acc 2, del 2;
from 3: acc 3, del 3.

= known (mdm) \ * * PS PC PR (ar) 4: d0sc d1sc a0cs a1cs
= C  - (mdm) \ * known - spec (ar) 2: acc del
(wp) C
```

Figure 4.7: The listing of the script file.

The solution obtained is a process whose description in FIREMAPS format is written to the file in Figure 4.6. The corresponding process automaton *X* is shown in Figure 4.7. For *X*, the actions ***a***1***xc, D,*** and ***a***0***xc*** constitute the outputs and *A*, *d*1*cx*, and *d*0*cx* represent the inputs.

```
6 actions; 7 states; 42 edges;
actions: a, a1xc, d, d1cx, d0cx, a0xc;
states: 6 t, 4 st, 5 t, 0 sti, 2 st, 1 s, 3 t;
edges:
   from 6: a 6, a1xc 6, d 6, d1cx 6, d0cx 6, a0xc 6;
   from 4: a 1, a1xc 3, d 5, d1cx 1, d0cx 1, a0xc 3;
   from 5: a 3, a1xc 5, d 6, d1cx 5, d0cx 5, a0xc 5;
   from 0: a 1, a1xc 3, d 5, d1cx 1, d0cx 2, a0xc 3;
   from 2: a 1, a1xc 0, d 5, d1cx 6, d0cx 1, a0xc 4;
   from 1: a 1, a1xc 1, d 1, d1cx 6, d0cx 1, a0xc 1;
   from 3: a 1, a1xc 3, d 5, d1cx 3, d0cx 3, a0xc 3.
```

Figure 4.8: The description of the process automaton *X*.

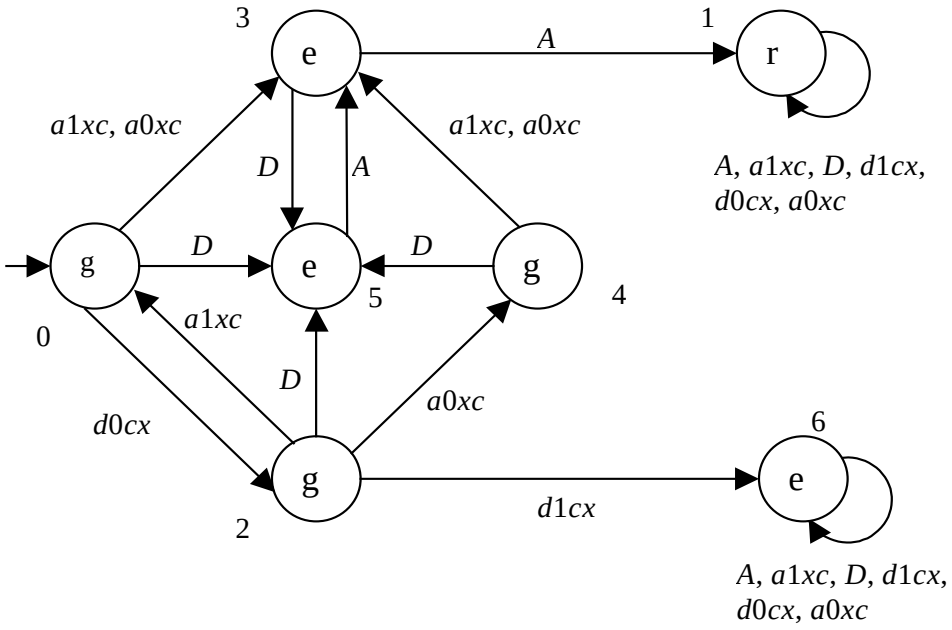


Figure 4.9: The process automaton *X*. (Missing input transitions in *X* lead to state 1. Missing output transitions in *X* lead to state 6.)

Notice that *X* has several states, which indicate violations. One reject state, 1, is reached by the traces including illegal inputs, e.g., (*d*0*cx*, *d*0*cx*). On the other hand, illegal outputs lead to the escape states: 3, 5, and 6. These states are reached by traces like (*d*0*cx*, *a*0*xc*, *a*0*xc*). However, the successors of these states are not all escape states. For instance, after reaching state 3, the input *A* leads to the reject state 1, the output *D* leads to state 5 (also an escape), and the other actions generate self loops. In addition, notice that *X* has an escape-input trace, i.e. (*d*0*cx*, *d*1*cx*). This means that the automaton is not safety-healthy because the escape set (of traces) produced is not suffix-closed. Therefore, we apply the

upgrade algorithm to refine *X* to a safety-healthy process. In this case, we need to apply steps 2 and 3 of the procedure to handle escape-inputs and make the escape set suffix-closed, respectively. However, when applying Step 2, we have to trace back the execution which ends with the escape input until an output is reached. The only output that precedes the escape-input is the empty word enabled in the initial state. The next step is to mark all the successors of the initial state, which enables the last output before the escape-input, as escape states. This leads to the automaton shown in Figure 4.8. However, such process automaton is equivalent to a single escape state in which all the actions generate self-loops, and which does not correspond to any physical system. In conclusion, we end up with no solution for the protocol conversion problem, as stated in [8].

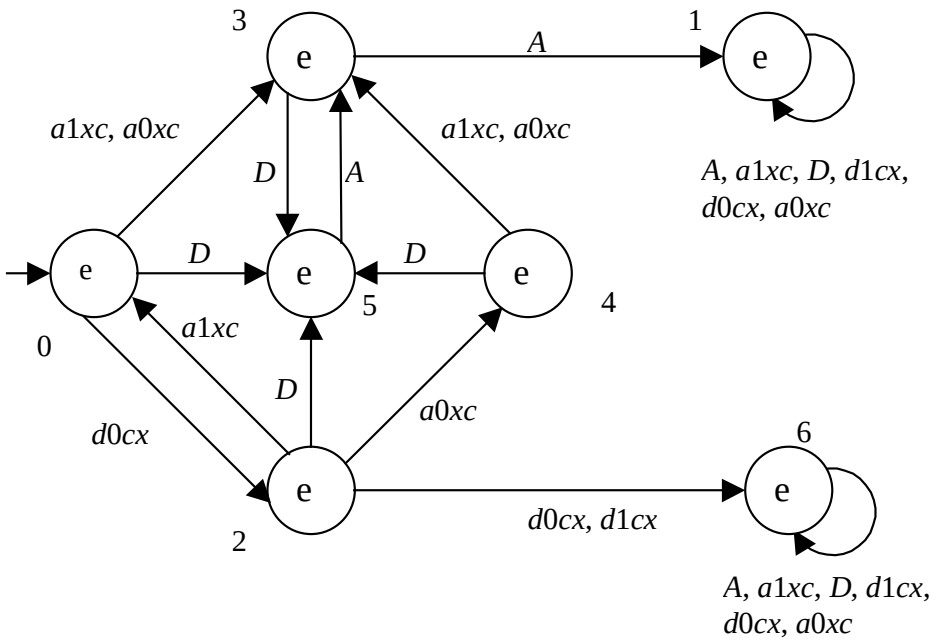


Figure 4.10: The process automaton *X* after step 1 of the upgrade algorithm. (Missing input transitions in *X* lead to state 1. Missing output transitions in *X* lead to state 6.)

Let us take a closer look at the specification of the modules *PS*, *PR*, *PC* and *SS* to find any errors in the original problem statement. We notice the following. After the first message *Acc* arrives, *PS* issues a *d0sc* and rejects any consecutive *Acc* until the correct acknowledgement *a*0*cs* is received. This means that traces like (*Acc*, *d*0*sc*, *d*0*cx*, *a*0*xc*, *D*, *Del*, *A*, *Acc*) are not allowed for the sender since the correct acknowledgement *a*0*cs* was

not received. This means that some module in the model has to take responsibility to avoid generating the sequence (*Acc*, *d*0*sc*, *d*0*cx*, *a*0*xc*, *D*, *Del*, *A*, *Acc*). In the specification of other modules and the composite system, this trace is not avoided. For example, *SS* does not avoid the trace since it respects interleaving the consecutive *Acc* messages with *Del*. The only way to avoid the trace, in the composite system, and respect the specification of the sender is to force the converter *X* to avoid such traces. However, the only way *X* can detect the arrival of consecutive *Acc* messages is through the arrival of *d*0*cx* or *d*1*cx*. Thus it is impossible to realize a converter *X* that can avoid the arrival of consecutive *Acc* messages without interleaving correct acknowledgements. This "contradiction" in specifying the behavior of the entities of the communication system was detected by our method when the upgrade procedure is applied (obtaining the empty solution).

4.3.3 Solving the Modified Problem

In the previous section, we concluded that using the original specification for the sender [8] yields no solution that corresponds to a physical system. This is depicted by obtaining a solution, which is not safety-healthy and results in an empty solution when refined by the upgrade algorithm.

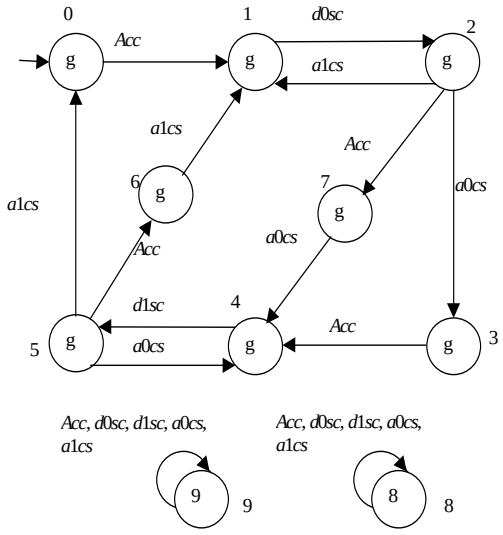


Figure 4.11: The modified specification of the sender *PS*. (Missing input transitions lead to state 8. Missing output transitions lead to state 9.)

To eventually obtain a non-empty solution, we slightly modify the specification of the sender so that consecutive *Acc* messages are not directly rejected but stored until the previous acknowledgement has arrived. Next, we use the new sender in the asynchronous equation and determine the converter *X*. The modified sender process automaton is shown in Figure 4.9, and the corresponding general solution *X* is represented by the process automaton in Figure 4.10.

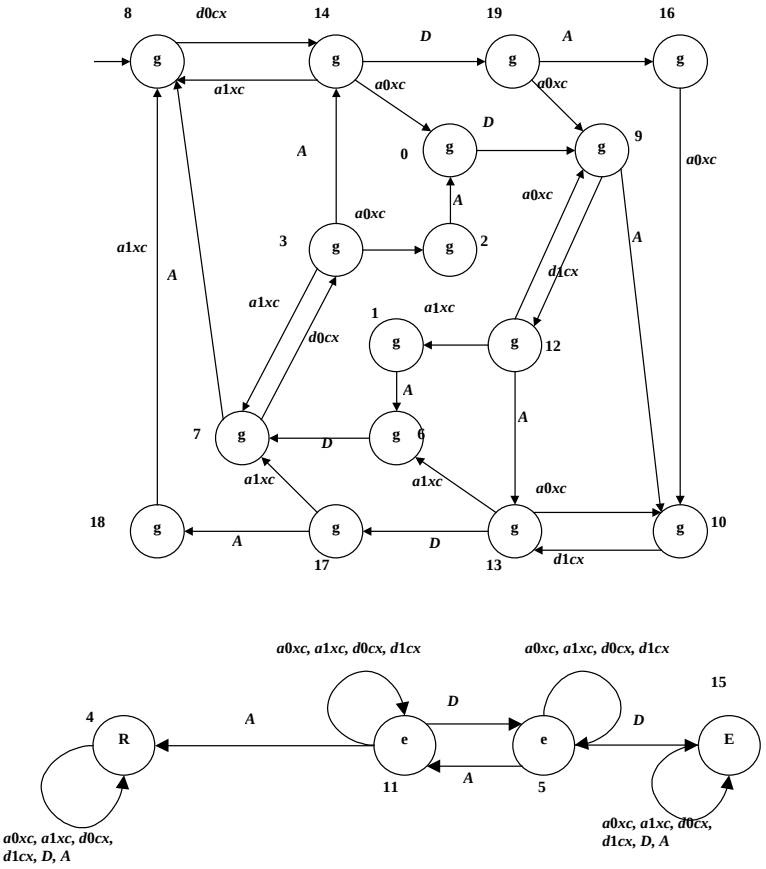


Figure 4.12: The process automaton *X* of the solution for the modified problem. (Missing input transitions in *X* lead to state 1. Missing output transitions in *X* lead to state 11, 5, and 15.)

Notice that *X* is non-deterministic in issuing outputs. For example, when a message (*d*0*cx*) arrives, the converter randomly chooses between acknowledging the sender through the channel (*a*0*xc*) and delivering the message to the receiver (*D*). This non-determinism is usually unnecessary and can be resolved to obtain a deterministic converter using various heuristics.

In addition, *X* has several states, which indicate violations. One trap state, 4, is reached by the traces including illegal inputs, e.g., (*d*0*cx*, *d*0*cx*). On the other hand, illegal outputs lead to the escape states: 11, 5, and 15. These states are reached by traces like (*d*0*cx*, *a*0*xc*, *a*0*xc*). However, the successors of these states are not all escape states. After reaching an escape state, inputs might lead to state 4, and outputs lead to other escape states.

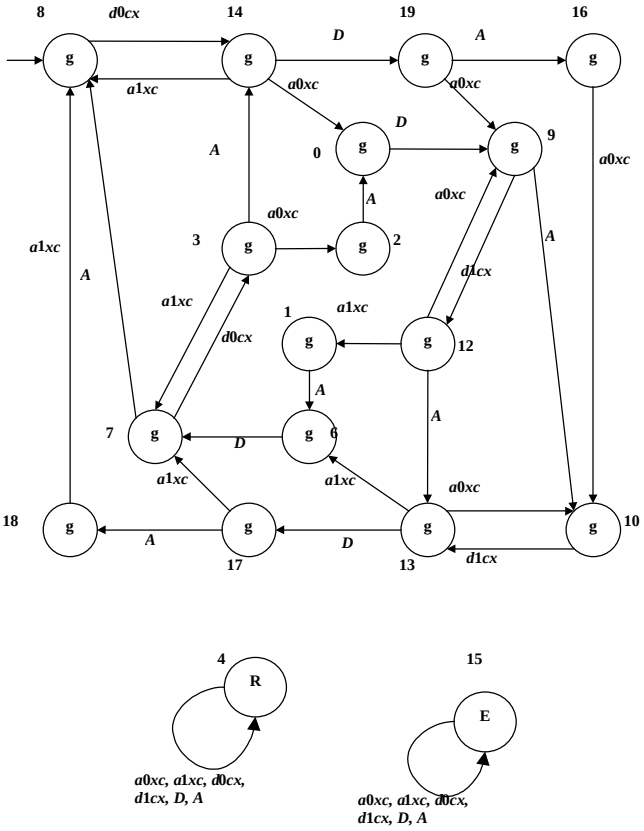


Figure 4.13: The upgraded process automaton *X*. (Missing input transitions in *X* lead to state 4. Missing output transitions in *X* lead to state 15.)

This means that the automaton is not safety-healthy because the escape set (of traces) produced is not suffix-closed. We apply the upgrade algorithm to the process automaton in Figure 4.10 in order to refine it into a safety-healthy process automaton. The only problem to overcome is the escape set that is not suffix-closed. Therefore, only Step 3 of the upgrade algorithm (Section 4.3.3) needs to be executed to close the escape set. This merges the three escape states (11, 5, and 15) into one escape state. The resulting process

automaton is shown in Figure 4.11. This is a safety-healthy process automaton whose escape set consists of one state out of which only escape traces are executed.

4.4 Divergence Freedom

In the previous section, we described how to design a protocol converter constrained only by the overall service specification. This solution, however, shows internal communications between the converter and the other entities, in the composite communication system, that might lead to divergence. For example, the sequence *d0cx a1xc*, Figure 4.12, might be executed infinitely many times before the communication system can respond to its environment.

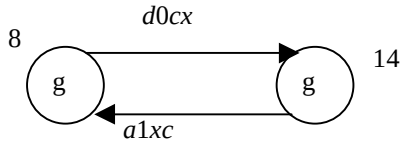


Figure 4.14: The sequence *d0cx a1xc*, which leads to divergence.

In this section, we describe how to design a divergence-free protocol converter. Such a converter avoids unbounded communications with other parts in the system while satisfying the overall service specification. We aim to avoid both "catastrophic" divergence which leads to unfair behavior and "non catastrophic" divergence which leads to undue power consumption. In order to obtain such converter, we augment the overall service specification by combining *SS* with the auxiliary process automaton, $limit_n$, where *n* is a positive integer.

The automaton $limit_n$ monitors the actions on the boundaries of the converter, and prevents them from causing divergence. For example, when $limit_n$ observes the sequence *d0cx a1xc*, it signals a violation and enters a trap state.

Figure 4.13 shows the process automaton $limit_1$, which forbids the converter from issuing negative acknowledgments to the messages it receives because they might lead to cycles.

Notice that state 3 is an escape state from which the process cannot resume normal behavior.

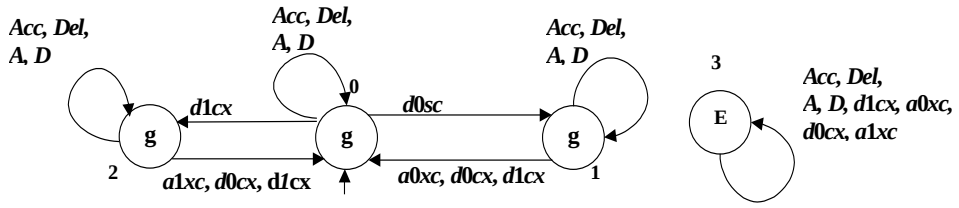


Figure 4.15: The process automaton *limit*$_1$. (Missing transitions lead to state 3.)

With the introduction of the limiting process, the modified algorithm to obtain the divergence-free converter becomes as follows:

1. Obtain the process automaton *Known* from the product of all the known modules in the system the same way as in the previous algorithm.
2. Obtain the product of *Known* with the reflection of the modified specification SS_n obtained as the product of *SS* with $limit_n$. Hide the actions, which are irrelevant to the converter. *X* doesn't observe *Acc* and *Del*.
3. Determinize and then minimize the obtained process to avoid the non-determinism that results from hiding.
4. Reflect the result to obtain the solution, *X*.

The following command line shows how the original procedure of obtaining a solution is modified to result in a divergence-free solution with a limit equal to 1:

```
= C – (mdm) \ * Known  -  ^ limit1 SS1 (ar) 2: Acc Del
```

The solution obtained, after upgrade, is shown in Figure 4.14. Notice how the unbounded internal communications are avoided. The execution (*d0cx, a1xc*), for example, does not cause a cycle anymore. It leads to an escape state meaning that the converter guarantees to avoid such executions. On the other hand, *X* preserves the its right to choose how to behave when messages, e.g. *d0cx.*, arrive through the channel. In fact, *X* chooses non-deterministically between forwarding the messages to the receiver and acknowledging the sender through the channel.

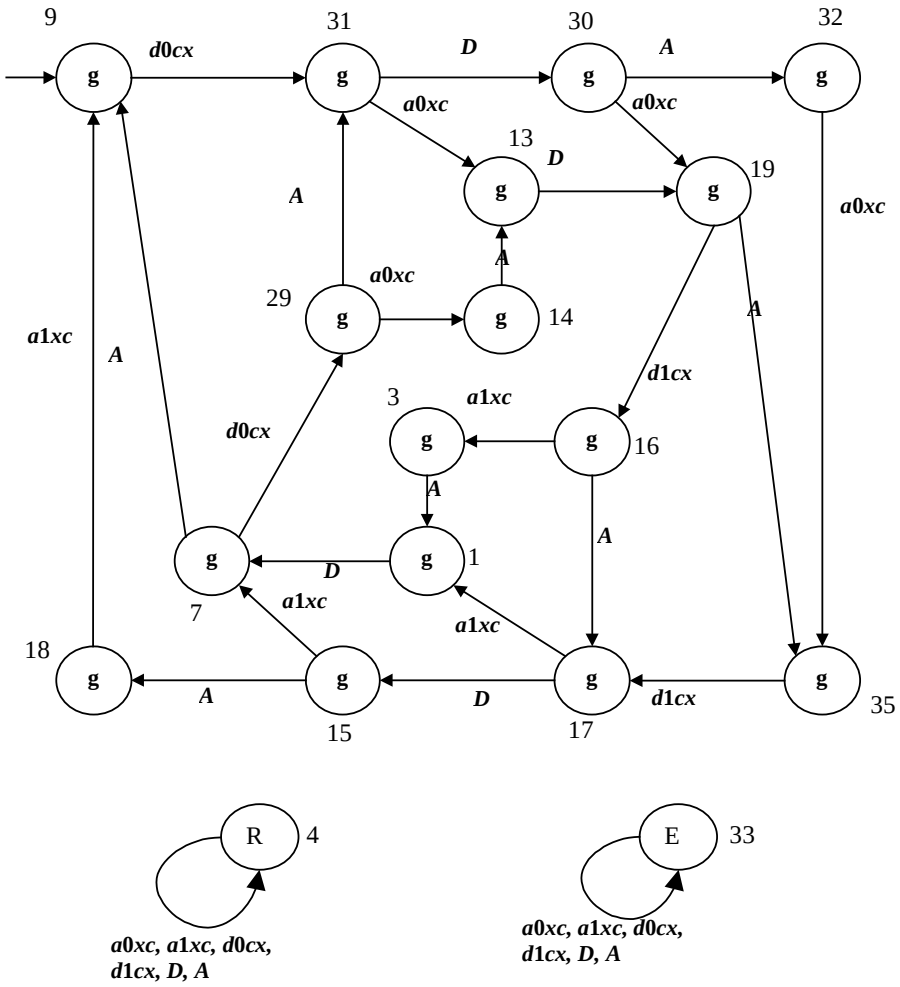


Figure 4.16: The process automaton of the divergence-free converter *X*. (Missing input transitions in *X* lead to state 4. Missing output transitions in *X* lead to state 33.)

4.5 The Solution in I/O Automata

In this section, we describe how to map the solution process automaton *X* into an I/O automaton using the backmapping presented in Chapter 3. First of all, we know that the I/O automaton, which corresponds to *X*, has the following action signature (Section 4.3.1):

In(*X*)={ *d*1*cx*, *d*0*cx*, *A* }.

Out(*X*)={ *a*1*cx*, *a*0*cx*, *D* }.

Int(*X*)=∅.

In addition, we represent the transition relation by the finite state machine in Figure 4.15. Notice that the reject state (state 4) of the process automaton has been mapped into a trap state (state 16), and the illegal outputs that caused divergence are simply disabled.

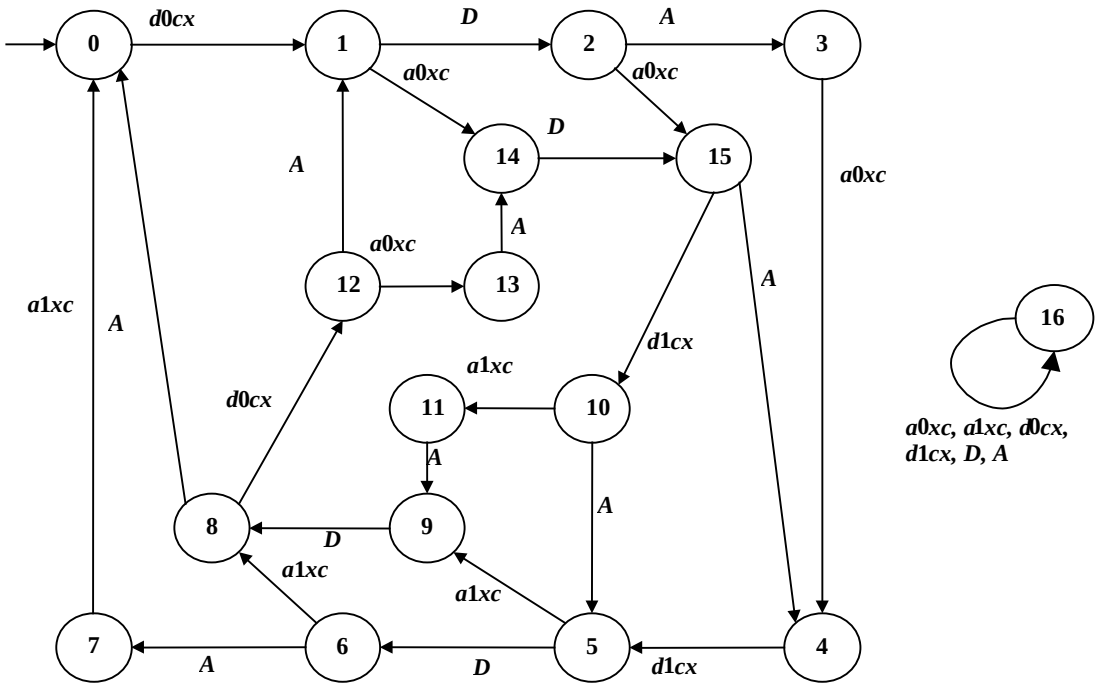


Figure 4.17: The corresponding I/O automaton *X*. (Missing input transitions lead to state 16)

4.6 Synthesis of the Solution

Finally, we derive an asynchronous implementation of our converter using an automated synthesis tool, Petrify [7], which is used for the synthesis of Petri nets and asynchronous controllers. Petrify accepts behavioral models in either two formats: Petri nets or state graphs, and both can be obtained from process automata that represent safety-healthy processes. However, for a state graph (SG) to be implementable into a circuit using Petrify, the following main property must hold in the circuit [7]:

Complete state encoding: This means that any two states, in the graph with the same code have the same set of enabled non-input signals (outputs).

In the case of the converter of Figure 4.15, a complete state encoding could not be easily achieved due to the non-deterministic output generation. Therefore, for simplicity, we synthesize a protocol converter based on a deterministic refinement X_d of the divergence-free converter *X*. This simplified converter respects the overall service specification of the interconnected system while having a restricted behavior in terms of choosing between acknowledgement and delivery of messages. In fact, it executes only the sequence *d*0*cx*, *a*0*xc*, *D*, *A*, *d*1*cx*, *a*1*xc*, *D*, *A*, and it can be transformed directly into the state graph shown in Figure 4.16.

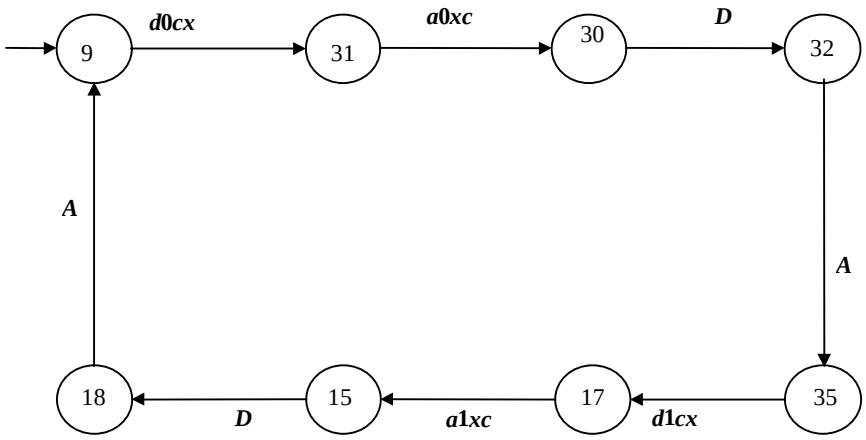


Figure 4.18: The process automaton of the simplified protocol converter X_d.

Next, we describe the state graph of X_d in the Petrify format in order to implement the corresponding circuit. This is done in a file that can be read by Petrify. Figure 4.17 shows the listing of the file with the Petrify description of X_d.

The resulting implementation could be represented by the following logic functions, output by the tool, that represent a speed independent circuit:

[*D*] = *d*0*cx d*1*cx*' + *d*0*cx*' *d*1*cx*.

[*a*0*xc*] = *A*' *d*1*cx* + *A d*0*cx*.

[*a*1*xc*] = *A*' *a*0*xc* + *A a*1*xc*.

```
.model conv
.inputs a d0cx d1cx
.outputs d a0xc a1xc
.state graph
s9  d0cx s31
s31 d s30
s30  a s32
s32 a0xc s35
s35  d1cx s17
s17 d s15
s15 a s18
s18 a1xc s9
.marking {s9}
.end
```

Figure 4.19: The state graph description of the simplified protocol converter X_d in Petrify format.

From the above logic functions, a simple circuit can be implemented to interface the two mismatching protocols. The corresponding circuit is shown in Figure 4.18. Notice that we assume the presence of each signal and its conversion as primary inputs, e.g. *A* and *A'*.

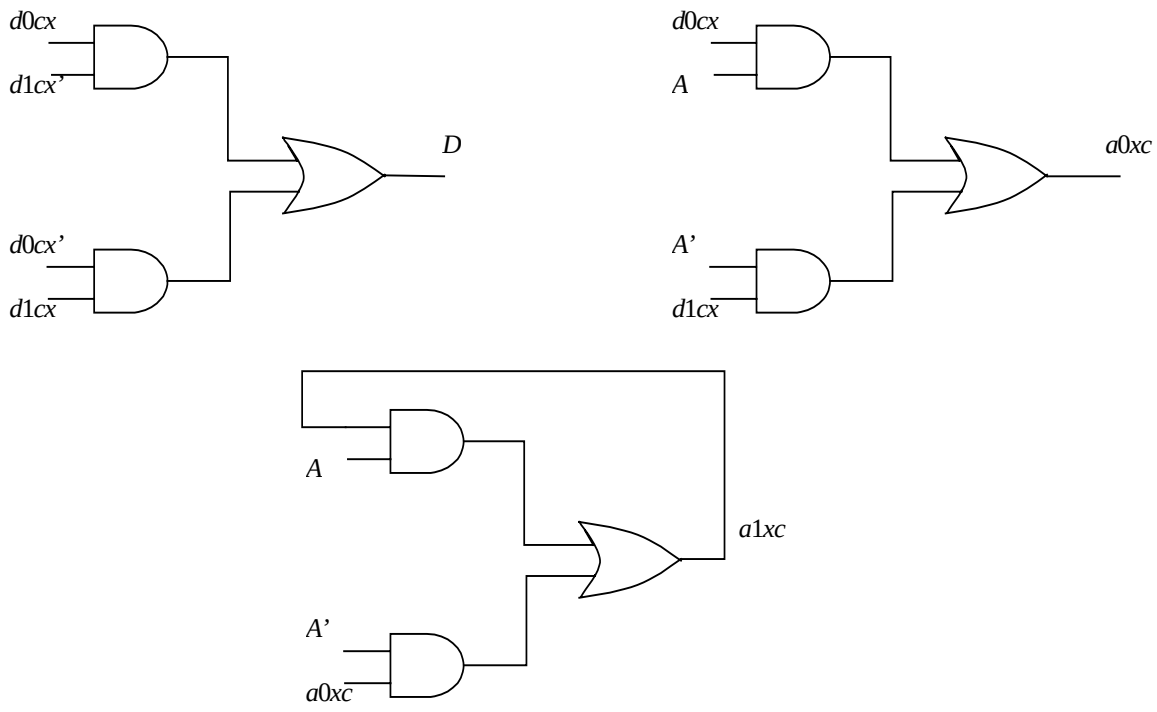


Figure 4.18: Circuit implementation of the converter *X*.

Chapter 5

Conclusions

In this report, we have studied the applications and relationships of I/O automata and process spaces models for concurrent systems. They have been briefly presented, compared, and related through semantics mappings. The mappings allow interchanging relations and operations between the two models:

- The refinement relations between I/O automata correspond to refinement relations between processes.
- Composition of I/O automata is mapped to a composition of processes.

As an application of the mapping, we presented a method to solve asynchronous equations formulated in I/O automata using methods and tools available in process spaces. Our method allowed us to obtain a divergence-free solution based on a strong definition of divergence-freedom for the asynchronous equation in terms of safety processes. At the same time, the application of the method showed that using finalization processes does not guarantee a divergence-free solution. Furthermore, the mapping helped simplify the process of obtaining solutions and divergence-free solutions by using FIREMAPS. In contrast to previous attempts to compute divergence-free solutions to the asynchronous equation problem [18, 19], we insert a limiting process in the specification itself. In other words, we apply a general technique for deriving supervisory control on an altered specification that includes a divergence-freedom requirement in the form of a distinct process. A benefit entailed is uniformity, which means that both the theoretical underpinnings and the tools can be reused for other problems. The algorithms from Chapter 3 have already been implemented, with the exception of the upgrade procedure.

Future work should include implementing this procedure and integrating it into FIREMAPS.

We also devised a backmapping from process spaces to I/O automata, which allows the reconstruction of I/O automata out of safety processes. On the other hand and in order to make the mapping easier to apply, we still need to build a front end to FIREMAPS that implements both the mapping and the backmapping. Also, we expect future work to reveal additional usage for our mapping.

Finally, we presented a case study on how to apply asynchronous equations in obtaining specifications for protocol converters, which interface heterogeneous network systems with mismatched protocols. A typical application is the synthesis of low power asynchronous converters from state graphs, using techniques from [3, 7]. Our case study is based on an example from [8]. In contrast to [8] however, our approach deals with divergence freedom and can indicate the absence of a solution for some specifications. A state graph was obtained by producing a safety-healthy process automaton, and the specification obtained was synthesized using an automated tool.

References

1. G. V. Bochmann. *Deriving Protocol Converters for Communication Gateways.* IEEE transactions on Communications. Vol. 9, Sept. 1990.
2. E. Brinksma, L. Heerink, and J. Tretmans. *Developments in Testing Transition Systems.* International Workshop on Testing of Communicating Systems X, pages 143-166. Chapman & Hall, 1997.
3. R-D. Chen, J-M. Jou and Y-H. Shiau. *An Efficient Method for the Decomposition and Resynthesis of Speed-Independence Circuits*. In Proc. of the International Conference on Electronics, Circuits, and Systems, 1999, pp.339-342.
4. M. D. Dibenedetto, A. Saldanha, and A. Sangiovanni-Vincentelli. *Model Matching for Finite State Machines.* In Proc. of the 33rd conference on Decision and Control. Lake Buena Vista, Florida, December, 1994.
5. J. Drissi and G. Bochmann. *Submodule Construction for Systems of I/O Automata*. Technical Report no. 1133, Department d'Informatique et de Recherche Operationelle, University of Montreal, 1999.
6. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, UK, 1985.
7. Kondratyev, J. Cortadella, M. Kishinevsky, L. Lavagno, and A. Yakovlev. *Technology Mapping for speed-independent circuits: Decomposition and Resynthesis*. In Proc. of the Symposium on Advanced Research in Asynchronous Circuits and Systems, 1997, pp. 240-253.
8. R. Kumar, S. Nelvagal, and S. Marcus. *Protocol Conversion Using Supervisory Control Techniques*. In Proc. of the 1996 IEEE International Symposium on Computer-Aided Control System Design, 1996 , pp. 32-37
9. N. Lynch and M. Tuttle. *Hierarchical Correctness Proofs for Distributed Algorithms.* Technical Report MIT/LCS/TR-387, Laboratory for Computer Science. Massachusetts Institute of technology, April 1987.
10. N. Lynch and M. Tuttle. *An Introduction to Input/Output Automata.* Laboratory for Computer Science. Massachusetts Institute of Technology, November 1988.

11. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc San Francisco, 1996.

12. W. Mallon and J. Udding. *Building Finite Automata from DI Specifications.* In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 184-193, 1998.

13. W. Mallon and T. Verhoeff. *Analysis and Applications of the XDI model*. In Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, pages 231-242, 1999.

14. R. Negulescu. *Process Spaces and Formal Verification of Asynchronous Circuits*. Ph.D. Thesis, Department of Computer Science, University of Waterloo, August 1998. http://www.macs.ece.mcgill.ca/~radu/Papers/Rnthesis.ps.

15. R. Negulescu. *Process Spaces.* Technical Report CS-95-48, Computer Science, University of Waterloo,1995.

16. R. Negulescu. *Process Spaces.* In Proc. of the 11th International Conference on Concurrency Theory (CONCUR 2000). Pennsylvania, USA. August, 2000.

17. K. Okumura. *Generation of Proper Adapters and Converters from a Formal Service Specification*. In Proc. of ACM SIGCOMM'86.

18. Overkamp. *Supervisory Control Using Failure Semantics and Partial Specifications*. IEEE transactions on Automatic Control, Vol. 42, No. 4, April 1997.

19. A. Petrenko and N. Yevtushenko. *Solving Asynchronous Equations*. In Proc. of FORTE/PSTV'98. Paris, France.

20. P. J. Ramadge and W. M. Wonham. *The control of discrete event systems.* In Proc.of IEEE, vol. 77, pp. 81-98, Jan. 1989.

21. R. Segala. *Quiescence, Fairness, Testing, and the Notion of Implementation.* 4th International Conference on Concurrency Theory. Hildestein, Germany, August 1993.

22. Z. P. Tao. *Formal Method for the Design of Real-Time Communicating Subsystems and Controllers*. Department d'Informatique et de Recherche Operationelle, University of Montreal, 1996.

23. Y. W. Yao, W. S. Chen, and M. T. Liu. *A Modular Approach to Constructing Protocol Converters.* In Proc. of IEEE INFOCOM'90, San Francisco, CA, 1990.