Sequences that Lead to Failure in Decentralized Supervisory Control

by

Arezou Mohammadi

A thesis submitted to the

Department of Electrical and Computer Engineering

in conformity with the requirements for the degree of

Master of Science (Engineering)

Queen's University

Kingston, Ontario, Canada

September 2003

Abstract

In decentralized supervisory control problems a necessary condition for supervisors to exist, called *co-observability*, can be checked using an algorithm reviewed in this work. It is proved that the algorithm also can be employed to check whether the legal language is *controllable* with respect to the plant, which together with the co-observability, is necessary and sufficient for decentralized supervisors to exit. Although controllability and co-observability can be checked using the algorithm, it is necessary to develop a method to determine all the problematic sequences of events that cause failure of controllability or co-observability. This general method, called the *all-paths method*, is developed based on the algorithm. Running the all-paths method results in a set of tuples of sequences that violate controllability or co-observability. Another method is proposed, namely the *some-paths method*, which is simpler to implement than the all-paths method. The some-paths method gives a non-empty subset of the tuples of the sequences resulting from the all-paths method, if the legal language is not controllable or co-observable with respect to the plant. The some-paths method is implemented and applied to some examples. Telecommunication protocols are examples whose problematic sequences could be extracted using the suggested methods. Protocols of the data link layer of OSI architecture are a class of telecommunication protocols. Three examples of these protocols are provided and modelled to verify whether they fail. The protocols are verified manually.

Acknowledgements

I would like to extend my special thanks to my supervisor Professor Karen Rudie for her guidance, encouragement, and support throughout my Master's program. She is a great source of knowledge and I was lucky to be her student. I would like to thank Professor Kai Salomaa for the guidance and helpful comments that he kindly gave me throughout my work. I would also like to thank Professor Fadi Alajaji and Professor Tamás Linder. Without their support I could not study in my favorite area. I am thankful of my colleagues in the DES lab, especially Lenko for the burden of changing the format of the figures.

I can never thank my dear husband Firouz enough for his boundless support, kindness, encouragement, and the lucky life that he has provided for me. Great appreciation goes to my parents and my sister and brother for their unconditional love and their encouragement throughout my study and my life that gives me motivation to continue this way.

I am grateful to Professor K. Salomaa, Professor I. M. Kim, Professor C. H. Yeh, and Professor C. MacDougall for their useful comments in my defence.

I would like to thank Professor Karen Rudie for the financial support she provided. Also, thanks are due to the School of Graduate Studies at Queen's University for its financial support through the Queen's Graduate Awards program.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

A discrete-event system (DES) is a set of consecutive discrete events. All sequences of the events begin from an initial state and a state transition is caused when an event occurs. However, all possible sequences of events that could happen in the system may not be desired sequences. The system behavior should be restricted to achieve a demanded behavior. Therefore, the system should be controlled, and so a controller (or supervisor) needs to be determined. The controller restricts the behavior of the system by preventing some controllable events from occurring and allowing others to occur. Discrete-Event Systems considers this approach.

Given a fixed system, a single controller that is able to obtain the desired behavior is a solution to the centralized control problem. However, sometimes to solve the problem it is necessary to find two or more controllers. If they can be found, a solution to the decentralized control problem is determined. In this work we focus on the case of two supervisors that co-operate to control a system. Sometimes there do not exist supervisors that can effect decentralized

control. This situation arises if and only if there exist certain tuples of illegal and legal sequences of events in the system. If the tuples are found, we may be able to recognize how to change the system physically. The new system could then be controlled by decentralized supervisors to achieve the desired behavior. However, this thesis does not address how to change the system to obtain the new system.

In this thesis, we find a method to represent the tuples of illegal and legal sequences that cause failure in a decentralized control problem with two controllers.

1.2 Literature Review

Supervisory control theory studies discrete-event systems in a framework was initiated by Ramadage and Wonham [14, 16, 15]. It models a system and applies control-theoretic techniques to find some controllers to constrain the system's behavior (*plant*) to achieve a desired subset of it (*legal language*). In fact the goal of supervisory control theory is to prevent undesired sequences of events from happening. Therefore, some agents (*supervisors*) should be found to restrict the behavior of the plant via a control decision [14, 15, 33, 11, 2, 32].

Centralized and decentralized supervisory control are two different approaches to look at DES problems. Centralized supervisory control studies control techniques to solve a DES problem using a single supervisor [14, 33, 11, 2].

Decentralized supervisory control gives more flexibility to solve complex problems. Also, some problems such as telecommunication protocols [26, 28], distributed computer problems [30, 29, 27, 1], parallel computing systems [1, 3, 6], multiprocessor systems with shared memory [27, 30], and distributed database systems [12] are naturally distributed and should be dealt with using a decentralized approach. Decentralized supervisory control looks for decentral-

ized controllers whose combined control actions achieve the desired behavior.

One may ask under what conditions there exist some distributed controllers as a solution to a supervisory control problem. Rudie and Wonham have presented necessary and sufficient conditions for the existence of a solution to the problem of finding decentralized supervisors [23]. These conditions are referred to as *co-observability* and *controllability*. Later, Rudie and Willems introduced a polynomial-time algorithm to check co-observability for prefix-closed problems [20, 19]. It is shown in this thesis that this algorithm can be used to check controllability too.

On the other hand, it is shown by Rudie and Wonham that in some cases the sequences of events that cause the failure of a distributed protocol, such as communication protocols, are sequences that violate co-observability or controllability [24, 23]. In this thesis it is proved that the algorithm offered in [20, 19, 21] can also be used to check controllability. Based on some ideas provided in [21, 19], we present a method to determine the sequences that violate co-observability or controllability and a version of that method that determines *some* sequences that cause failure of co-observability or controllability. We also model some communication protocol examples to show how one can use the techniques of [24, 20] to find failures of the protocols.

1.3 Contribution

The following items summarize the contribution of this research:

- It is proved that the algorithm in [21, 19] can be applied to check the controllability of a legal language with respect to a plant.
- A method (called the *all-paths method*) is provided to find *all* sequences of events that cause failure of controllability or co-observability of the legal

language with respect to the plant. Some concepts used in the method are taken from [19] and [25].

- A simplified version of the all-paths method (called the *some-paths method*) is provided. The simplified version finds *some* sequences of events that violate controllability or co-observability of the legal language with respect to the plant.
- A software implementation in C of the some-paths method is developed.
- Three protocols for the data link layer of OSI architecture are modelled and verified manually using the concepts of discrete-event systems.

1.4 Thesis Overview

This thesis is organized as follows. The second chapter contains the fundamental concepts of discrete-event systems such as centralized and decentralized supervisory control, controllability, and co-observability.

In the third chapter, the algorithm given in [21, 19], which checks co-observability, is presented. Then, we prove that the automaton construction given in the paper can be used to check controllability at the same time. The next section of this chapter introduces a general method that we developed to find all illegal sequences of events and their corresponding legal sequences of events that violate controllability or co-observability. The method uses some ideas from the algorithm in [21, 19]. Another method is then presented to find some counterexamples for co-observability and/or controllability which are a subset of the results of the general method. The computational complexity of the second method is given.

The second method is the one which is implemented in software for the thesis. In Chapter 4 some examples are presented. Using the software, some counterex-

amples for co-observability and controllability of each example are found. This chapter continues with a brief overview of telecommunication protocols and explains why they are examples of decentralized control problems and why the aforementioned method can be used to find sequences that cause failures in a telecommunication protocol. This chapter also describes some protocols in the data link layer of OSI architecture. These protocols are modelled and manually verified. It is explained why the implementation can be used theoretically, but not practically, to find the sequences of events that lead to failure in each of the protocols.

Finally, the last chapter contains conclusions and some suggestions for future work.

In appendix A, the concept of regular expression is explained.

Chapter 2

Background

2.1 Basic Definitions

Supervisory control theory models discrete-event processes and uses some control techniques to restrict the behavior of a process to prevent the occurrence of undesired behavior.

A DES can be modelled by automaton $G = (\Sigma, Q, \delta, q_0, Q_m)$, where Σ is the set of all possible events that can occur within the system, Q is the set of states, $q_0 \in Q$ is the state from which the system starts, and $Q_m \subseteq Q$ is the set of marked states. The function $\delta : \Sigma \times Q \longrightarrow Q$ is the transition function. Let σ be an event. The notation $q_j = \delta(\sigma, q_i)$ indicates that when the system is in state q_i, occurrence of σ transfers the system to state q_j. So, G can be represented by a directed graph whose nodes are the states in Q and whose edges are the transitions between any two states. A sequence of events can be represented by a concatenation of symbols from Σ. The set Σ^* is the set of all finite sequences of events over Σ including the null string (ϵ). Any subset of Σ^* is called a language. Given a language L, the *prefix-closure* of L, denoted by $\bar{L}$, is the set containing all prefixes of all strings in L. Typically, given a sequence s, $\bar{s}$ represents the

set containing all the prefixes of s. Since all strings are prefixes of themselves, $L \subseteq \bar{L}$. The language L is said to be prefix-closed if $L = \bar{L}$.

Now we can extend the definition of δ from a domain of $\Sigma \times Q$ to $\Sigma^* \times Q$ as follows:

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

$$(\forall \sigma \in \Sigma, s \in \Sigma^*)\delta(s\sigma, q) = \delta(\sigma, \delta(s, q))$$

Hence, δ indicates to which state a *sequence* of events will lead. A state $q \in Q$ is *reachable* if $\exists s \in \Sigma^*$ such that $q = \delta(s, q_0)$. A state $q \in Q$ is *co-reachable* if $\exists s \in \Sigma^*$ such that $\delta(s, q) \in Q_m$, *i.e.*, if there is a path from the state q to a marked state. An automaton whose states are all reachable and co-reachable is called *trim*.

A DES can be expressed by two languages $L_m(G)$ and $L(G)$. The language $L(G)$ is the set of all possible event sequences which may be generated in the plant, i.e.,$L(G) = \{s \mid s \in \Sigma^* \text{ and } \delta(s, q_0) \text{ is defined}\}$. The language $L(G)$ is called the *generated language*. The language $L_m(G)$ is the set of all event sequences that represent the completion of some tasks, i.e., $L_m(G) = \{s \mid s \in \Sigma^* \text{ and } \delta(s, q_0) \in Q_m\}$. The language $L_m(G)$, which is a subset of $L(G)$, is called the *marked language*.

Consider two automata G_1 and G_2 as follows:

$$G_1 = (X_1, \Sigma_1, \zeta_1, x_{0,1}, X_{m,1})$$

$$G_2 = (X_2, \Sigma_2, \zeta_2, x_{0,2}, X_{m,2})$$

The *parallel composition* ($\|$) of the automata G_1 and G_2 is the following automaton:

$$G_1 \parallel G_2 = (X_1 \times X_2, \Sigma_1 \cup \Sigma_2, \delta, (x_{0,1}, x_{0,2}), X_{m,1} \times X_{m,2})$$

where

$$\delta((x_1,x_2),e) = \begin{cases} (\delta_1(x_1,e),\delta_2(x_2,e)) & \text{if } \delta_1(x_1,e) \text{ and } \delta_2(x_2,e) \text{ are defined.} \\ (\delta_1(x_1,e),x_2) & \text{if } \delta_1(x_1,e) \text{ is defined and } \delta_2(x_2,e) \text{ is not.} \\ (x_1,\delta_2(x_2,e)) & \text{if } \delta_2(x_2,e) \text{ is defined and } \delta_1(x_1,e) \text{ is not.} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The definition of an automaton can be extended to what is called a *non-deterministic automaton*. A non-deterministic automaton has a transition function $\delta : \Sigma \times Q \rightarrow 2^Q$, i.e., an event from state Q can lead to more than one other state.

2.2 Centralized Control

The aim of supervisory control is to find a controller to impose supervision on the plant to achieve a prescribed legal behavior. From the controller perspective, some events are controllable and others are uncontrollable. So, Σ_c, the set of controllable events, consists of those events that the controller may enable (permit to occur) or disable (prevent from occurring) and Σ_{uc}, the set of uncontrollable events, contains those which cannot be prevented from occurring. Also, the event set Σ can be divided into disjoint sets Σ_o and Σ_{uo}, which are observable and unobservable events, respectively.

The mapping $P : \Sigma^* \longrightarrow \Sigma_o^*$ is called the *canonical projection*. It is defined recursively as follows:

$$\begin{aligned} &P(\epsilon) = \epsilon \\ &P(\sigma) = \epsilon && \text{if } \sigma \in \Sigma \backslash \Sigma_o \\ &P(\sigma) = \sigma && \text{if } \sigma \in \Sigma_o \\ &P(s\sigma) = P(s)P(\sigma) && \forall \sigma \in \Sigma, s \in \Sigma^*. \end{aligned}$$

The projection operator P *erases* the events from the given sequences that are

unobservable to the supervisor. Given a language K, $P(K)$ stands for the language defined by $\{P(s) \mid s \in K\}$. The *inverse projection* of P is specified as $P^{-1} : 2^{\Sigma_o^*} \longrightarrow 2^{\Sigma^*}$. It is defined on the sets of strings (or languages) as $P^{-1}(K) = \{t \mid P(t) \in K\}$. A supervisor observes a filtered version of sequences of events that are physically possible in the plant. It disables or enables any of the controllable events throughout its observation to yield a desired subset of the plant behavior. This subset is called the *legal language*, E. So the behavior of the plant should be constrained by a supervisor to a legal pre-determined language.

A supervisor $\mathcal{S}$ is identified by a pair (T, ψ), where T is an automaton which recognizes a language over the same event set as the plant G, and ψ, called a *feedback map*, is a map from the event set and states of T to the set {enable, disable}. Let X be the set of states of T. The map ψ must satisfy the following constraint:

$$\psi(\sigma, x) = \text{enable if } \sigma \in \Sigma_{uc} \text{ and } x \in X$$

i.e., uncontrollable events cannot be disabled. The automaton T tracks and controls the behavior of G. The automaton T changes state based on the events generated by G. At each state x of T, the control rule $\psi(\sigma, x)$ dictates whether σ is to be enabled or disabled at the corresponding state of G.

The behavior of the closed-loop system is the set of generated sequences of events when the plant is under the control of $S = (T, \psi)$. It can be represented by an automaton S/G. The language $L(S/G)$ is the set of strings generated by both G and S, and where each event in the string is enabled by ψ. The language $L_m(S/G)$ denotes the marked behavior of the system. It contains those strings in $L(S/G)$ that are marked by both G and S.

A closed-loop system is *nonblocking* if every string generated by the closed-loop system can be completed to a marked string in the system, i.e., if $\overline{L_m(S/G)} =$

$L(S/G)$.

In centralized supervisory control, the aim is to look for *one* supervisor that observes the sequences $P(L(G))$ and controls the plant by manipulation of the controllable events, to guarantee a desired behavior.

2.3 Decentralized Control

In contrast to centralized control, decentralized supervisory control addresses systems, which because of their physical specifications, require us to apply multiple supervisors to yield the desired subset of plant behavior.

In this thesis we are going to study a class of decentralized control problems, called Global Problems (GP) in [23]. To study the solution to GP, we need to examine a special case of it, which is called Global Problem with Zero Tolerance (GPZT) [23]. For convenience we call GP the "decentralized problem" and GPZT the "narrowed decentralized problem" from now on.

The following definitions are used in the decentralized control problems. Consider a supervisor $\mathcal{S}_i$ which imposes a specific legal behavior on a plant by disabling or enabling the controllable events in $\Sigma_{i,c} \subseteq \Sigma$, when it observes some subset $\Sigma_{i,o} \subseteq \Sigma$. Another supervisor can be defined which makes the same control decision as $\mathcal{S}_i$ on $\Sigma_{i,c}$, enables all events in $\Sigma \backslash \Sigma_{i,c}$, makes the same transitions as $\mathcal{S}_i$ on $\Sigma_{i,o}$ and stays in the same state for events in $\Sigma \backslash \Sigma_{i,o}$. The supervisor is denoted by $\widetilde{\mathcal{S}}_i$. The supervisor $\widetilde{\mathcal{S}}_i$ acts on all of Σ while $\mathcal{S}_i$ acts only on a subset of Σ, so $\mathcal{S}_i$ is called the *local supervisor* and $\widetilde{\mathcal{S}}_i$ is its *global extension*. Consider two local supervisors $\mathcal{S}_1 = (T_1, \phi)$ and $\mathcal{S}_2 = (T_2, \psi)$ acting on G, where $T_1 = (X, \Sigma, \zeta, x_0, X_m)$ and $T_2 = (Y, \Sigma, \eta, y_0, Y_m)$. The *conjunction* of $\mathcal{S}_1$ and $\mathcal{S}_2$ is the supervisor denoted by $\mathcal{S}_1 \wedge \mathcal{S}_2 = (T_1 \times T_2, \phi * \psi)$, where

$$T_1 \times T_2 = (X \times Y, \Sigma, \zeta \times \eta, (x_0, y_0), X_m \times Y_m)$$

and $\forall x \in X, y \in Y, \sigma \in \Sigma$,

$$(\zeta \times \eta)(\sigma,(x,y)) = \begin{cases} (\zeta(\sigma,x),\eta(\sigma,y)) & \text{if } \zeta(\sigma,x) \text{ and } \eta(\sigma,y) \text{ are defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(\phi * \psi)(\sigma,(x,y)) = \begin{cases} \text{disable} & \text{if } \phi(\sigma,x) = \text{disable, and } \psi(\sigma,y) = \text{disable} \\ \text{enable} & \text{otherwise} \end{cases}$$

It is shown in [34] that

$$L(\mathcal{S}_1 \wedge \mathcal{S}_2/G) = L(\mathcal{S}_1/G) \cap L(\mathcal{S}_2/G)$$

$$L_m(\mathcal{S}_1 \wedge \mathcal{S}_2/G) = L_m(\mathcal{S}_1/G) \cap L_m(\mathcal{S}_2/G)$$

In this thesis we focus on the following problem of decentralized supervisory control.

Decentralized Problem (DP) (GP in [23]): Given a plant G over an alphabet Σ, a legal language $L(E) \subseteq L_m(G)$, a minimally adequate language $L(A) \subseteq L(E)$, and sets $\Sigma_{1,c}, \Sigma_{2,c}, \Sigma_{1,o}, \Sigma_{2,o} \subseteq \Sigma$, construct local supervisors[1] *$\mathcal{S}_1$ and $\mathcal{S}_2$ such that*

$$L(A) \subseteq L(\widetilde{\mathcal{S}}_1 \wedge \widetilde{\mathcal{S}}_2/G) \subseteq L(E)$$

and such that $\widetilde{\mathcal{S}}_1 \wedge \widetilde{\mathcal{S}}_2$ is a nonblocking supervisor for G. Here, for $i = 1, 2$, supervisor $\mathcal{S}_i$ can observe only events in $\Sigma_{i,o}$ and control only events in $\Sigma_{i,c}$ and $\widetilde{\mathcal{S}}_i$ is the global extension of $\mathcal{S}_i$. The set of uncontrollable events, Σ_{uc}, consists of $\Sigma \backslash (\Sigma_{1,c} \cup \Sigma_{2,c})$. The language $L(E)$ represents the legal or desired behavior and $L(A)$ is the minimum set of sequences that must be achieved in closed loop.

Any solution to DP should be contained in some given range of languages. Hence, this case introduces a synthesis problem with tolerance. To study DP,

[1] In fact, the supervisors must also satisfy a technical condition called *completeness* [23, 32].

we first consider a special case of DP, namely, $L(A) = L(E)$. The *Narrowed Decentralized Problem* (GPZT in [23]) is defined as follows.

Narrowed Decentralized Problem (NDP): Given a plant G over an alphabet Σ, a legal language $L(E)$, $\emptyset \neq L(E) \subseteq L_m(G)$, and sets $\Sigma_{1,c}, \Sigma_{2,c}, \Sigma_{1,o}, \Sigma_{2,o} \subseteq \Sigma$, construct local supervisors $\mathcal{S}_1$ and $\mathcal{S}_2$ such that

$$L_m(\widetilde{\mathcal{S}}_1 \wedge \widetilde{\mathcal{S}}_2/G) = L(E)$$

and such that $\widetilde{\mathcal{S}}_1 \wedge \widetilde{\mathcal{S}}_2$ is a nonblocking supervisor for G.

For DP, the aim is to find supervisors to constrain the system behavior to a language which lies in a given range of languages. This means that the languages under consideration in this case are prefix-closed. In contrast, since NDP requires supervisors to recognize a language equal to some given language, it is not necessary to require that the language be prefix-closed. If the endpoints of the range of behavior in DP are equal, namely $A = E$, DP is reducible to NDP with a prefix-closed specification. Using this property, a solution to DP can be derived from the solution to NDP. Necessary and sufficient conditions of existence of the solution to NDP are presented in [23, theorem 4.1]. The theorem says that there exist supervisors $\mathcal{S}_1$ and $\mathcal{S}_2$ that solve NDP if and only if $L(E)$ is controllable and co-observable with respect to G, P_1, P_2. We define these properties below.

A language $K \subseteq L(G)$ is *controllable* with respect to G if

$$\overline{K}\Sigma_{uc} \cap L(G) \subseteq \overline{K}.$$

In this definition, for any language L and M, $LM = \{st \mid s \in L, t \in M\}$. Let $L(G)$ be a physically possible behavior and K be a legal behavior. Informally, K is controllable with respect to the plant if for any sequence of events s that starts out as a legal sequence ($s \in \overline{K}$), the occurrence of an uncontrollable event ($\sigma \in \Sigma_{uc}$) which is physically possible ($s\sigma \in L(G)$) does not lead the sequence

out of the legal range ($s\sigma \in \overline{K}$). For example, consider automata E and G in Figure 2.1. In Figure 2.1, dashed-lines represent transitions that are in G and not in E. In this example, the legal language in not controllable with respect to the plant, because the sequence $s = abc$ is a legal sequence, the event $\sigma = b$ is an uncontrollable event, and the sequence $s\sigma = abcb$ is physically possible but is not a legal sequence.

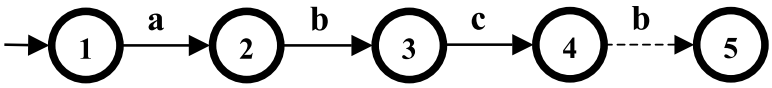


Figure 2.1: Automata E and G, where $\Sigma_{c,1} = \{a\}$, $\Sigma_{c,2} = \{c\}$, and $\Sigma_{uc} = \{b\}$.

Given a plant G over alphabet Σ, sets $\Sigma_{1,c}, \Sigma_{2,c}, \Sigma_{1,o}, \Sigma_{2,o} \subseteq \Sigma$, canonical projections $P_1 : \Sigma^* \longrightarrow \Sigma_{1,o}^*$, $P_2 : \Sigma^* \longrightarrow \Sigma_{2,o}^*$, a language $K \subseteq L_m(G)$ is *co-observable* with respect to G , P_1, and P_2 if
$s, s', s'' \in \Sigma^*, P_1(s) = P_1(s'), P_2(s) = P_2(s'') \Rightarrow$

$$
\begin{aligned}
& \forall \sigma \in \Sigma_{1,c} \cap \Sigma_{2,c} \quad s \in \overline{K} \wedge s\sigma \in L(G) \wedge s'\sigma, s''\sigma \in \overline{K} \Rightarrow s\sigma \in \overline{K} \\
\wedge \quad & \forall \sigma \in \Sigma_{1,c} \backslash \Sigma_{2,c} \quad s \in \overline{K} \wedge s\sigma \in L(G) \wedge s'\sigma \in \overline{K} \Rightarrow s\sigma \in \overline{K} \\
\wedge \quad & \forall \sigma \in \Sigma_{2,c} \backslash \Sigma_{1,c} \quad s \in \overline{K} \wedge s\sigma \in L(G) \wedge s''\sigma \in \overline{K} \Rightarrow s\sigma \in \overline{K} \\
\wedge \quad & s \in \overline{K} \cap L_m(G) \wedge s', s'' \in K \Rightarrow s \in K
\end{aligned}
$$

The control decisions made by an agent are based on the agent's view of the system, i.e., the sequences containing only the events that the agent can observe. Consider any two sequences of events, namely s and s', that look the same to an agent and any event $\sigma \in \Sigma_c$, where $s\sigma$ and $s'\sigma$ can happen physically in the plant. The control decision made by the agent to disable or enable event σ after sequences s or s' would be the same. Language K is co-observable with respect to a plant when for any event $\sigma \in \Sigma_c$, there is at least one agent that

is able to make the correct control decision about σ. The set of controllable events, Σ_c, is $\Sigma_{1,c} \cup \Sigma_{2,c}$. The last conjunct of co-observability expresses that at least one supervisor should determine without ambiguity whether or not a given string should be marked. This conjunct is automatically satisfied when the languages are prefix-closed. For example, consider automata E and G in Figure 2.2. Again, dashed-lines represent transitions that are in G but not in E. In this example, E is not co-observable with respect to G, P_1, and P_2. This can be seen by considering $s = abaa$, $s' = baaa$, $s'' = ba$, and $\sigma = c$. Then $P_1(s) = P_1(s')$, $P_2(s) = P_2(s'')$, $\sigma \in \Sigma_{1,c} \cap \Sigma_{2,c}$, s, $s'\sigma$ and $s''\sigma$ are legal sequences, $s\sigma$ is physically possible, but $s\sigma$ is an illegal sequence. The notion of co-observability is extendible to include any number of fixed local supervisors, not just two.

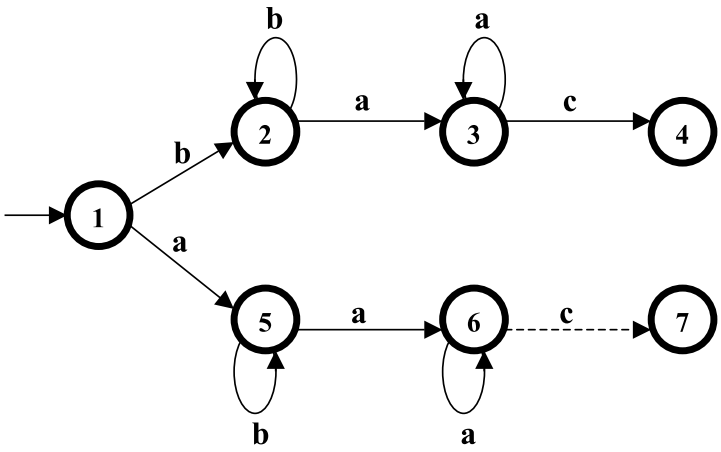


Figure 2.2: Automata E and G, where $\Sigma_{c,1} = \{c\}$, $\Sigma_{c,2} = \{c\}$, $\Sigma_{o,1} = \{a\}$, and $\Sigma_{o,2} = \{b\}$.

An algorithm in [23] is presented to check co-observability. We are going to use this algorithm and its results to present a method for extracting the sequences of events that violate controllability or co-observability of the legal

language with respect to the plant.

Chapter 3

What Violates Co-Observability or Controllability

In Section 3.1.1 of this chapter, we describe the algorithm that is presented in [21, 19]. The algorithm constructs an automaton M from the plant G and the legal language $L(E)$. Automaton M contains some families of paths from the initial state to the marked state.[1] Each path keeps track of strings that violate the co-observability of $L(E)$ with respect to G. In Section 3.1.2 we prove that the automaton M, which is constructed by the algorithm, can also be used to decide the controllability of $L(E)$ with respect to G. The automaton M contains some families of paths each of which corresponds to an ordered n-tuple of legal and illegal sequences of events. Each of these sequences causes failure in co-observability or controllability of $L(E)$ with respect to G. In Section 3.2.1, we present a method to find all such sequences of events using the algorithm and some methods provided in [19, 21]. In Section 3.2.2, another method is given that picks one path of each family of paths in automaton M, to find its

[1] When we say that automaton M "contains a path", we mean that the language generated by M contains a string that appears as labels of a path in M that starts at the initial state (and may contain cycles).

corresponding pair or pairs of (illegal, legal) sequences. The latter method is implemented in this thesis. In Section 3.3, the computational complexity of the implemented method will be studied.

3.1 How to Check Co-observability or Controllability

3.1.1 Co-observability

In [19, 21, 20] an algorithm is given to construct an automaton M; this automaton can be examined in order to decide the co-observability of a prefix-closed language $L(E)$ with respect to G where $L(E) \subseteq L(G)$ and E and G are finite-state automata. Automaton M keeps track of strings that violate the co-observability of $L(E)$ with respect to G, if the strings end with controllable events. Given $G = (Q^G, \Sigma, \delta^G, q_0^G, Q_m^G)$ and $E = (Q^E, \Sigma, \delta^E, q_0^E, Q^E)$, the automaton M can be constructed as follows [19, 21, 20].

Let d indicate an element that is not in $Q^E \cup Q^G$, and call d the dump state.

$$M = (Q^M, \Sigma, \delta^M, q_0^M, Q_m^M)$$

where

$$Q^M = Q^E \times Q^E \times Q^E \times Q^G \cup \{d\}$$

$$q_0^M = (q_0^E, q_0^E, q_0^E, q_0^G)$$

$$Q_m^M = \{d\}$$

In the definition of δ^M, sometimes at state (q_1, q_2, q_3, q_4) of Q^M, the following

set of conditions is referred to:

$$(*)\begin{cases} \delta^E(\sigma, q_3) \text{ is not defined} \\ \delta^G(\sigma, q_4) \text{ is defined} \\ \delta^E(\sigma, q_1) \text{ is defined if } \sigma \in \Sigma_{1,c} \\ \delta^E(\sigma, q_2) \text{ is defined if } \sigma \in \Sigma_{2,c} \end{cases}$$

Each transition in M is specified by a pair which includes an event σ from Σ and a number i from $\{0, 1, ..., 6\}$. The number i in the pair labelling a transition denotes "transition type". The notation (σ, i) is considered as an input symbol. So, each path in M can be specified by sequences of pairs $(\sigma_1, i_1)(\sigma_2, i_2)...(\sigma_n, i_n)$. The transition function δ^M is defined as follows, where the notation $a \overset{(\sigma,i)}{\longmapsto} b$ represents $\delta^M((\sigma, i), a) = b$.

For $\sigma \notin \Sigma_{1,o}$, $\sigma \notin \Sigma_{2,o}$,

$(q_1, q_2, q_3, q_4) \overset{(\sigma,1)}{\longmapsto} (\delta^E(\sigma, q_1), q_2, q_3, q_4)$
$(q_1, q_2, q_3, q_4) \overset{(\sigma,2)}{\longmapsto} (q_1, \delta^E(\sigma, q_2), q_3, q_4)$
$(q_1, q_2, q_3, q_4) \overset{(\sigma,3)}{\longmapsto} (q_1, q_2, \sigma^E(\sigma, q_3), \delta^G(\sigma, q_4))$
$(q_1, q_2, q_3, q_4) \overset{(\sigma,4)}{\longmapsto} (\delta^E(\sigma, q_1), \delta^E(\sigma, q_2), \delta^E(\sigma, q_3), \delta^G(\sigma, q_4))$
$(q_1, q_2, q_3, q_4) \overset{(\sigma,0)}{\longmapsto} d \text{ if } (*)$

For $\sigma \notin \Sigma_{1,o}$, $\sigma \in \Sigma_{2,o}$,

$(q_1, q_2, q_3, q_4) \overset{(\sigma,1)}{\longmapsto} (\delta^E(\sigma, q_1), q_2, q_3, q_4)$
$(q_1, q_2, q_3, q_4) \overset{(\sigma,5)}{\longmapsto} (q_1, \delta^E(\sigma, q_2), \delta^E(\sigma, q_3), \delta^G(\sigma, q_4))$
$(q_1, q_2, q_3, q_4) \overset{(\sigma,4)}{\longmapsto} (\delta^E(\sigma, q_1), \delta^E(\sigma, q_2), \delta^E(\sigma, q_3), \delta^G(\sigma, q_4))$
$(q_1, q_2, q_3, q_4) \overset{(\sigma,0)}{\longmapsto} d \text{ if } (*)$

For $\sigma \in \Sigma_{1,o}$, $\sigma \notin \Sigma_{2,o}$,

$(q_1, q_2, q_3, q_4) \overset{(\sigma,2)}{\longmapsto} (q_1, \delta^E(\sigma, q_2), q_3, q_4)$
$(q_1, q_2, q_3, q_4) \overset{(\sigma,6)}{\longmapsto} (\delta^E(\sigma, q_1), q_2, \delta^E(\sigma, q_3), \delta^G(\sigma, q_4))$

$(q_1, q_2, q_3, q_4) \overset{(\sigma,4)}{\longmapsto} (\delta^E(\sigma, q_1), \delta^E(\sigma, q_2), \delta^E(\sigma, q_3), \delta^G(\sigma, q_4))$
$(q_1, q_2, q_3, q_4) \overset{(\sigma,0)}{\longmapsto} d \text{ if } (*)$

For $\sigma \in \Sigma_{1,o}$, $\sigma \in \Sigma_{2,o}$,

$(q_1, q_2, q_3, q_4) \overset{(\sigma,4)}{\longmapsto} (\delta^E(\sigma, q_1), \delta^E(\sigma, q_2), \delta^E(\sigma, q_3), \delta^G(\sigma, q_4))$
$(q_1, q_2, q_3, q_4) \overset{(\sigma,0)}{\longmapsto} d \text{ if } (*)$

For $\sigma \in \Sigma$, $\delta^M(\sigma, d)$ is undefined

The occurrence of a certain event may lead to several states in automaton M as defined by the transition function δ^M, so the above construction results in a nondeterministic automaton M.

In [19, 21, 20], it is shown that in automaton M, each 4-tuple labelling of a state (q_1, q_2, q_3, q_4) keeps track of strings s, s', and s''. For strings $s, s', s'' \in \Sigma^*$, $\sigma \in \Sigma$, the sequence $s'\sigma$ leads to q_1, the sequence $s''\sigma$ leads to q_2, the sequence s leads to q_3, and the sequence $s\sigma$ leads to q_4. The states q_1, q_2, q_3, and q_4 are used to indicate if $s'\sigma \in L(E)$, $s''\sigma \in L(E)$, $s \in L(E)$, and $s\sigma \in L(G)$, respectively. It is proved in [19] that a path ends in state d in automaton M by a controllable event if and only if prefix-closed language $L(E)$ is not co-observable with respect to G. State d is the only marked state in automaton M.

[19, proposition 3.1]: *Given automaton E and G, the language $L(E)$ is not co-observable with respect to G if and only if M recognizes a nonempty language, in other words, if and only if there is a path in M from the initial state to the dump state where the path ends in a controllable event.*

In Section 3.1.2 of this work, we prove that the language $L(E)$ is not controllable with respect to G if and only if there exists a path in M from the initial state to the dump state that ends in an uncontrollable event. Each path from the initial state to the marked state can be expressed by a sequence of pairs of (event, transition-type). Each pair represents a transition. Any arbitrary

sequence of transitions in M is denoted by s^M. In [21, 19], the authors present three projection maps that take sequences in M as inputs to produce strings s, s', and s''. These strings are counterexamples to co-observability if they end up in an event $\sigma \in \Sigma_{1,c} \cup \Sigma_{2,c}$.

The projection maps are defined as follows:

$$F_1((\alpha, i)) = \begin{cases} \alpha \text{ if } i = 1, 4, 6 \\ \epsilon \text{ otherwise} \end{cases}$$

$$F_2((\alpha, i)) = \begin{cases} \alpha \text{ if } i = 2, 4, 5 \\ \epsilon \text{ otherwise} \end{cases}$$

$$F_3((\alpha, i)) = \begin{cases} \alpha \text{ if } i = 3, 4, 5, 6 \\ \epsilon \text{ otherwise} \end{cases}$$

For $j = 1, 2, 3$, $F_j(\epsilon) = \epsilon$ and for $s \in \Sigma^*$, $\sigma \in \Sigma$, $j \in \{0, 1, 2, ..., 6\}$, $F_j(s(\sigma, i)) = F_j(s)F_j((\sigma, i))$.

Then s, s', s'' are defined as follows:

$$s' = F_1(s^M)$$
$$s'' = F_2(s^M)$$
$$s = F_3(s^M)$$

Transitions between 4-tuples in M are constructed to keep track of sequences s, s', s'' as well as to guarantee that $P_1(s) = P_1(s')$ and $P_2(s) = P_2(s'')$.

In the next subsection, we are going to apply the two following claims,

[19, claim 1]:*If $s^M \in L(M)\backslash L_m(M)$ then $F_3(s^M) \in L(E)$, and if s^M leads to state (q_1, q_2, q_3, q_4) in M, then $F_3(s^M)$ leads to state q_3 in E and $F_3(s^M)$ leads to state q_4 in G.*

[19, claim 6]:*Given $s, s', s'' \in L(E)$ such that $P_1(s) = P_1(s')$ and $P_2(s) = P_2(s'')$, then there exists a sequence $s^M \in L(M)\backslash L_m(M)$ such that $F_3(s^M) = s$,*

$F_1(s^M) = s'$, *and* $F_2(s^M) = s''$.

3.1.2 Controllability

We claim that the algorithm presented in [19, 21] can also be applied to check controllability. In other words, prefix-closed language $L(E)$ is not controllable with respect to G if and only if the automaton M constructed in Section 3.1.1 contains a path ending in marked state d by an uncontrollable event. To prove the above claim, the following two claims should be demonstrated.

(a) If there is a path in automaton M that ends in the marked state d by an uncontrollable event, then $L(E)$ is not controllable with respect to G.

(b) If prefix-closed language $L(E)$ is not controllable with respect to G, then there exists at least one path from the initial state to the marked state in automaton M that terminates with an uncontrollable event.

Proof (a): Let $s^M(\sigma, 0)$ be a sequence of events in M from the initial state to the marked state whose last transition is of the form $(\sigma, 0)$ where $\sigma \in \Sigma_{uc}$. Therefore, s^M is a sequence that leads to a state, which we denote by (q_1, q_2, q_3, q_4), just before the marked state, d.

Let $s := F_3(s^M)$, and apply [19, claim 1] to get

$$s \in L(E) \tag{3.1}$$

The state (q_1, q_2, q_3, q_4) leads to the marked state via the occurrence $(\sigma, 0)$, where $\sigma \in \Sigma_{uc}$ and this happens when $(*)$ holds. Hence,

$$s\sigma \notin L(E) \tag{3.2}$$

$$s\sigma \in L(G) \tag{3.3}$$

Language $L(E)$ is prefix-closed, namely $L(E) = L_m(E)$.

From 3.1, 3.2, 3.3, and the fact that $\sigma \in \Sigma_{uc}$, we conclude that $L(E)$ is not controllable with respect to G.

Proof (b): Suppose that $L(E)$ is not controllable with respect to G. That is,

$$\exists \sigma \in \Sigma_{uc},\ \exists s \in L(E),\ s\sigma \in\ L(G),\ s\sigma \notin L(E) \tag{3.4}$$

Since $L(G)$ is prefix-closed and $s\sigma \in L(G)$,

$$s \in L(G) \tag{3.5}$$

Let $s' := s$ and $s'' := s$. Then, from [19, claim 6], there exists a sequence $s^M \in L(M) \backslash L_m(M)$ such that $F_3(s^M) = s$.

Suppose that s^M leads to (q_1, q_2, q_3, q_4) in automaton M. Consider the following facts

$$\begin{cases} s^M \in L(M) \backslash L_m(M), \\ s = F_3(s^M), \\ s \in L(E), \text{ and} \\ L(E) = L_m(E) \end{cases}$$

and using [19, claim 1], we can conclude,

$$s \text{ leads to } q_3 \text{ in } E \tag{3.6}$$

$$s \text{ leads to } q_4 \text{ in } G \tag{3.7}$$

So, from (3.6) and the fact that $s\sigma \notin L(E)$,

$$\delta^E(\sigma, q_3) \quad \text{is not defined} \tag{3.8}$$

and from (3.7) and the fact that $s\sigma \in L(G)$,

$$\delta^G(\sigma, q_4) \quad \text{is defined.} \tag{3.9}$$

Accordingly, from (3.8), (3.9), and the fact that $\sigma \in \Sigma_{uc}$, we conclude that $(*)$ holds.

Hence, $(\sigma, 0)$ leads (q_1, q_2, q_3, q_4) to the marked state, d, in automaton M, where $\sigma \in \Sigma_{uc}$. Therefore, there is a path that ends with an uncontrollable event from the initial state to the marked state in automaton M.

3.2 Sequences of Events that Violate Co-observability or Controllability

In the general problem of distributed supervisory control, DP, defined in Section 2.3, we seek supervisors S_1 and S_2 such that there exists a language K,

$$L(A) \subseteq K \subseteq L(E),$$

$$K = L(\mathcal{S}_1 \wedge \mathcal{S}_2/G),$$

and K is a prefix-closed language.

If $L(A) = L(E)$, such supervisors can be found if and only if $L(E)$ is controllable and co-observable with respect to G [23]. Using automaton M, co-observability and controllability can be checked. The supervisors do not exist if and only if there are some tuples of legal and illegal sequences of events that cause failure in controllability or co-observability of $L(E)$ with respect to G. In other words, the supervisors do not exist if and only if there is a path in automaton M from the initial state to the marked state. If the path ends with an event in Σ_c, then it corresponds to ordered tuples of legal and illegal sequences that violate co-observability. Otherwise, if it ends with an event in Σ_{uc}, the path corresponds to pairs of (illegal, legal) sequences that cause a failure in controllability of $L(E)$ with respect to G. In general, automaton M may contain some families of paths from the initial state to the marked state, and each family may contain some cycles. Each cycle may repeat any number of times. Accordingly, an infinite number of paths can be identified. Moreover, each family of paths may

correspond to many pairs of (illegal, legal) or triples of (illegal, legal, legal) sequences that violate controllability or co-observability of $L(E)$ with respect to G.

In [19, 21], it is shown that automaton M encodes all sequences that violate co-observability and a method is provided for mapping any given path in automaton M to the sequences that violate co-observability. Let us find the sequences of events that violate co-observability or controllability of the following example.

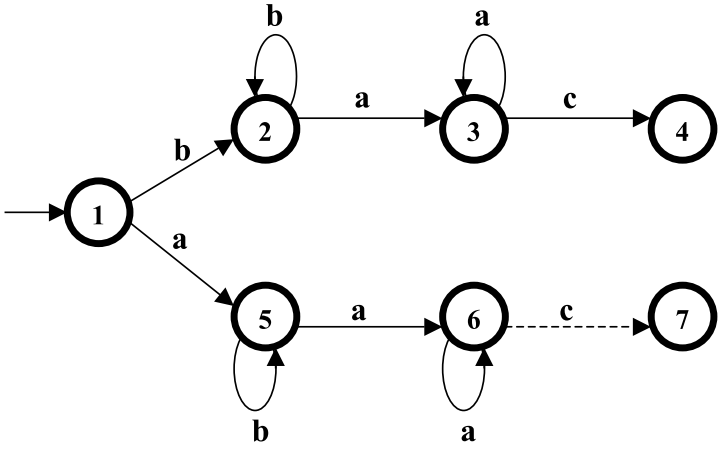


Figure 3.1: Automata E and G, where $\Sigma_{c,1} = \{c\}$, $\Sigma_{c,2} = \{c\}$, $\Sigma_{o,1} = \{a\}$, and $\Sigma_{o,2} = \{b\}$.

Example 3.1 We return again to Figure 2.2, reproduced here as Figure 3.1. As explained in Section 2.3, if $s = abaa$, $s' = baaa$, $s'' = ba$, and $\sigma = c$, then the tuple $(s\sigma, s'\sigma, s''\sigma)$ violates co-observability. The sequences $s = aba$, $s' = baa$, $s'' = ba$, and event $\sigma = c$ are another set that cause failure of co-observability for the above example. For Example 3.1, by going through some of the self-loops an arbitrary number of times, we can find an infinite number of sets of sequences that violate co-observability. In Section 3.2.1 we present a method to extract *all*

tuples that cause failure in co-observability or controllability. In Section 3.2.2 we present a method that is simpler to implement and picks up just one path of each family of paths in M and finds its corresponding tuple of legal and illegal sequences of events. The method provided in 3.2.2 finds *some tuples*, each of which corresponds to a family of paths in automaton M.

3.2.1 All-Paths Method

The following steps should be used to find all tuples of (illegal, legal) or (illegal, legal, legal) sequences of events that violate co-observability or controllability of $L(E)$ with respect to G. We apply the following steps to Example 3.1 to illustrate how the all-paths method works.

(a) Given E, G, and $\Sigma_{1,c}, \Sigma_{2,c}, \Sigma_{1,o}, \Sigma_{2,o} \subseteq \Sigma$, we generate automaton M [21]. Each transition from one state to another state in automaton M is labelled by the pair of ⟨event, transition-type⟩. Automaton M can be constructed so that it consists of only reachable and co-reachable states. Furthermore, the number of its states could be decreased if we minimize automata E and G and then construct automaton M [10].

- Applying this step to Example 3.1 results in the automaton M represented in Figure 3.2.

(b) Using the algorithm presented in [7], we convert automaton M to a regular expression that characterizes $L_m(M)$. In each step of the algorithm, the length of the resulting regular expression can be shortened; at each iteration, the regular expressions that are generated are simplified using the following rules:

$$\epsilon^* \quad \Rightarrow \quad \epsilon$$

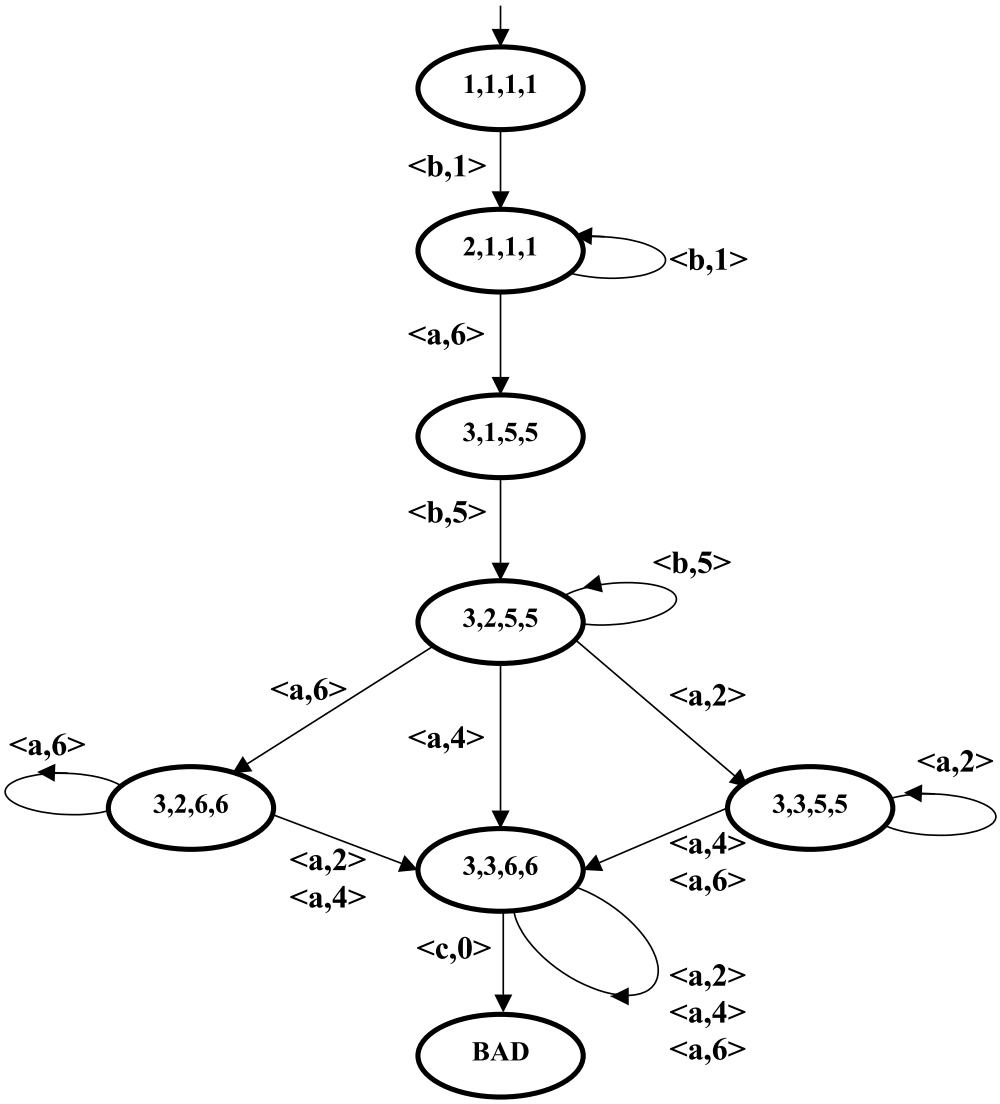


Figure 3.2: Automaton M

$$
\begin{aligned}
(\epsilon) &\Rightarrow \epsilon \\
(R_1R_2)R &\Rightarrow R_1R_2R \\
R(R_1R_2) &\Rightarrow RR_1R_2
\end{aligned}
$$

$$\begin{aligned} \epsilon R &\Rightarrow R \\ R\epsilon &\Rightarrow R \\ \epsilon\epsilon &\Rightarrow \epsilon \\ R + R &\Rightarrow R \\ \epsilon + \epsilon &\Rightarrow \epsilon, \end{aligned}$$

where R, R_1, and R_2 are arbitrary regular expressions and ϵ is a regular expression with length zero.

- For Example 3.1, the resulting regular expression is

  RE=$\langle b,1\rangle\langle b,1\rangle^*\langle a,6\rangle\langle b,5\rangle\langle b,5\rangle^*\langle a,6\rangle\langle a,6\rangle^*(\langle a,4\rangle+\langle a,2\rangle)(\langle a,4\rangle+\langle a,6\rangle$ $+\langle a,2\rangle)^*\langle c,0\rangle+\langle b,1\rangle\langle b,1\rangle^*\langle a,6\rangle\langle b,5\rangle\langle b,5\rangle^*\langle a,2\rangle\langle a,2\rangle^*(\langle a,4\rangle+\langle a,6\rangle)($ $\langle a,4\rangle+\langle a,6\rangle+\langle a,2\rangle)^*\langle c,0\rangle+\langle b,1\rangle\langle b,1\rangle^*\langle a,6\rangle\langle b,5\rangle\langle b,5\rangle^*\langle a,4\rangle(\langle a,4\rangle+$ $\langle a,6\rangle + \langle a,2\rangle)^*\langle c,0\rangle$

  Each operand of the resulting regular expression is a pair of $\langle event, transition - type\rangle$.

(c) We add notation "." between any two consecutive operands or between a "$*$" and an operand in the regular expression. Then, we reduce the resulting regular expression, which is in infix format, to its equivalent postfix format and convert the postfix expression to a binary tree.

In infix notation each operator is placed between its corresponding two operands, for example $a + b$. Postfix format is the case where each operator succeeds its corresponding operands, for example $ab+$. A postfix expression can be reduced to a binary tree easily.

- The infix regular expression obtained is step (b) is transformed to the following postfix expression:

postfix RE=$\langle b,1\rangle\langle b,1\rangle^*\langle a,6\rangle\langle b,5\rangle\langle b,5\rangle^*\langle a,6\rangle\langle a,6\rangle^*\langle a,4\rangle\langle a,2\rangle + \langle a,4\rangle$ $\langle a,6\rangle\langle a,2\rangle{+}{+}^*\langle c,0\rangle$.....$\langle b,1\rangle\langle b,1\rangle^*\langle a,6\rangle\langle b,5\rangle\langle b,5\rangle^*\langle a,2\rangle\langle a,2\rangle^*\langle a,4\rangle$ $\langle a,6\rangle{+}\langle a,4\rangle\langle a,6\rangle\langle a,2\rangle{+}{+}^*\langle c,0\rangle$.....$\langle b,1\rangle\langle b,1\rangle^*\langle a,6\rangle\langle b,5\rangle\langle b,5\rangle^*\langle a,4\rangle$ $\langle a,4\rangle\langle a,6\rangle\langle a,2\rangle + {+}^*\langle c,0\rangle$..... $+ +$

In Figure 3.3, part of the binary tree for this postfix expression is displayed.

(d) A *family of paths* consists of all the paths which are the same, once their cycles are removed. Automaton M might consist of several families of paths. The regular expression and the binary tree constructed in step (c) are other representations of automaton M. Using the binary tree, the regular expression can be decomposed to some smaller regular expressions, each of which corresponds to a set of families of paths in automaton M, that for convenience, we will call a DRE (Decomposed Regular Expression) from now on. The collection of DREs generated in this step captures all sequences accepted by automaton M.

(e) Each DRE is a regular expression in postfix format. We reduce it to a binary tree and then transform the tree to an infix expression. Then, we remove the "." notation from the infix expression.

- For Example 3.1, step (d) results in three DREs in postfix format, and each of them is transformed here to its corresponding infix expression. The three expressions in infix format are

  $DRE_1 = \langle b,1\rangle\langle b,1\rangle^*\langle a,6\rangle\langle b,5\rangle\langle b,5\rangle^*\langle a,6\rangle\langle a,6\rangle^*(\langle a,4\rangle{+}\langle a,2\rangle)(\langle a,4\rangle{+}$ $\langle a,6\rangle + \langle a,2\rangle)^*\langle c,0\rangle$

  $DRE_2 = \langle b,1\rangle\langle b,1\rangle^*\langle a,6\rangle\langle b,5\rangle\langle b,5\rangle^*\langle a,2\rangle\langle a,2\rangle^*(\langle a,4\rangle{+}\langle a,6\rangle)(\langle a,4\rangle{+}$ $\langle a,6\rangle + \langle a,2\rangle)^*\langle c,0\rangle$

  $DRE_3 = \langle b,1\rangle\langle b,1\rangle^*\langle a,6\rangle\langle b,5\rangle\langle b,5\rangle^*\langle a,4\rangle(\langle a,4\rangle{+}\langle a,6\rangle{+}\langle a,2\rangle)^*\langle c,0\rangle$

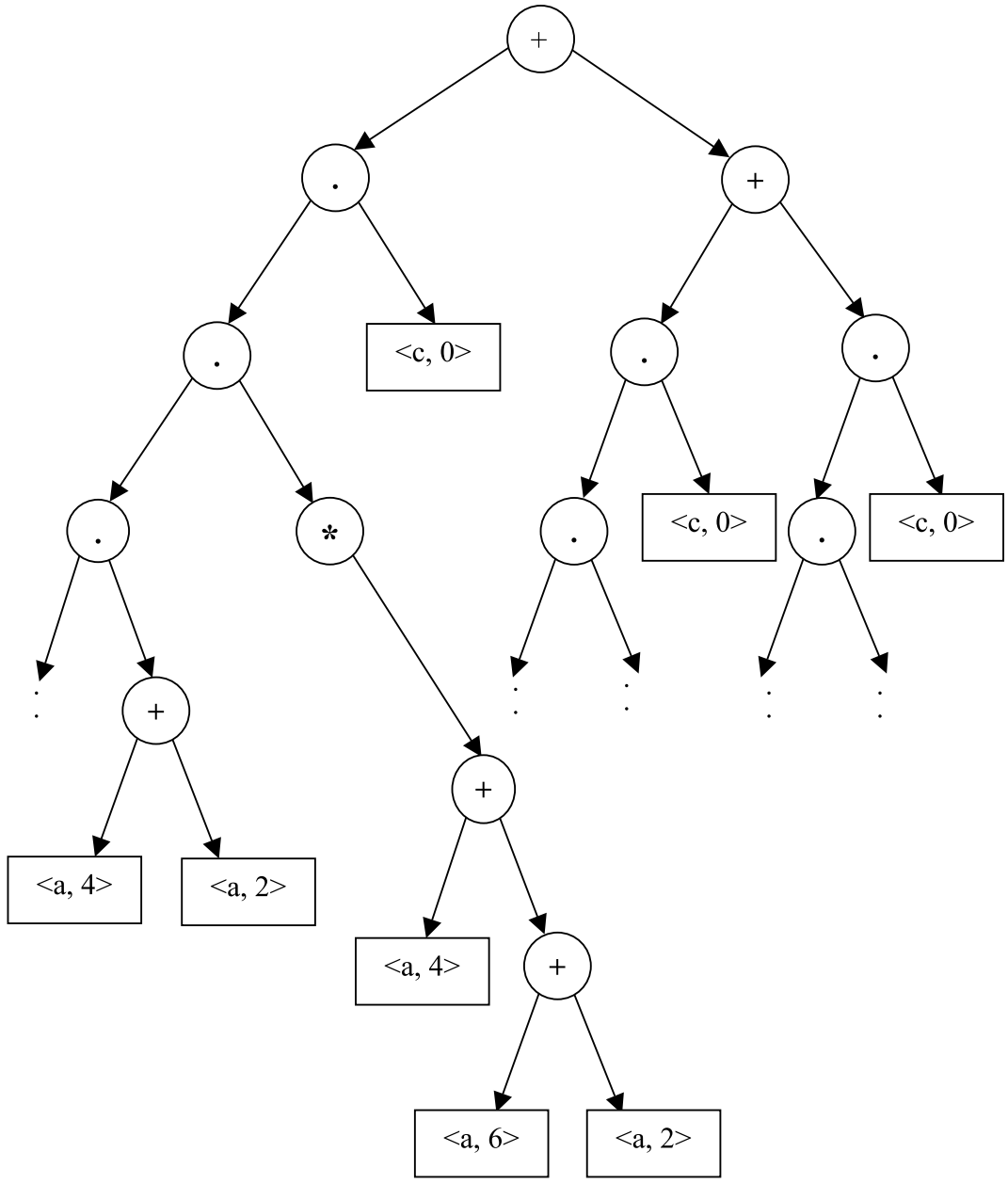


Figure 3.3: The binary tree corresponding to Example 3.1

We show how subsequent steps of the all-paths method work on DRE_1. Similarly, one would also need to apply these steps to DRE_2 and DRE_3.

(f) If the DRE_i contains an expression of the form $(R_1 + R_2 + ...)^*$, then we convert it to its equivalent form, that is $(R_1^* R_2^* ...)^*$ where R_i, $i = 1, 2, ...,$ is a regular expression. The reason will be explained in step (l).

- For Example 3.1, The regular expression DRE_1 obtained in step (e) is converted to the following form in this step:

  $DRE_1 = \langle b,1\rangle\langle b,1\rangle^*\langle a,6\rangle\langle b,5\rangle\langle b,5\rangle^*\langle a,6\rangle\langle a,6\rangle^*(\langle a,4\rangle+\langle a,2\rangle)(\langle a,4\rangle^*$ $\langle a,6\rangle^*\langle a,2\rangle^*)^*\langle c,0\rangle$

(g) If the DRE includes an expression of the form $S(R_1 + R_2 + ...)$ or $(R_1 + R_2+...)S$, then we convert it to its equivalent form, which is $SR_1+SR_2+...$ or $R_1S + R_2S + ...$, respectively, where S and R_i, $i = 1, 2, ...,$ are regular expressions .

- The regular expression resulting from step (f) for Example 3.1 is expressed in a new form, $T_1 + T_2$,

  $T_1 = \langle b,1\rangle\langle b,1\rangle^*\langle a,6\rangle\langle b,5\rangle\langle b,5\rangle^*\langle a,6\rangle\langle a,6\rangle^*\langle a,4\rangle(\langle a,4\rangle^*\langle a,6\rangle^*\langle a,2\rangle^*)^*$ $\langle c,0\rangle$

  and

  $T_2 = \langle b,1\rangle\langle b,1\rangle^*\langle a,6\rangle\langle b,5\rangle\langle b,5\rangle^*\langle a,6\rangle\langle a,6\rangle^*\langle a,2\rangle(\langle a,4\rangle^*\langle a,6\rangle^*\langle a,2\rangle^*)^*$ $\langle c,0\rangle$.

Now, each DRE is a regular expression in the form $T_1 + T_2 + ...$, where T_i, $i = 1, 2, ...,$ is a regular expression that does not contain $+$. We call each T_i a *term*. We do the following steps for each term that is acquired in step (g). For the example, we pick T_1 to illustrate the subsequent steps of the method. We should apply the same steps to T_2 as well.

(h) Before explaining this step, it is illustrative to examine Example 3.1.

– Consider the regular expression T_1 derived in step (g). If we apply F_1, F_2, and F_3 to T_1, then we get the following families of sequences for s, s', and s'', respectively:

$$\begin{aligned} s &\in abb^*aa^*a(a^*a^*)^* \\ s' &\in bb^*aaa^*a(a^*a^*)^* \\ s'' &\in bb^*a(a^*a^*)^* \end{aligned}$$

However, not all triples of elements from the above families will yield counterexamples to co-observability. For example, the sequences

$$\begin{aligned} t &= abaaaa \\ t' &= bbaaa \\ t'' &= baaa \end{aligned}$$

are sequences in s, s', and s'', respectively, but $P_1(t) \neq P_1(t')$ even though $P_2(t) = P_2(t'')$. To see how t, t', and t'' could be produced directly from T_1, we consider two paths of M that are elements of T_1. The first path in T_1 is

$$\langle b,1\rangle\langle a,6\rangle\langle b,5\rangle\langle a,6\rangle\langle a,2\rangle(\langle a,4\rangle\langle a,6\rangle^2\langle a,2\rangle)\langle c,0\rangle$$

and the second path is

$$\langle b,1\rangle\langle b,1\rangle\langle a,6\rangle\langle b,5\rangle\langle b,5\rangle\langle a,6\rangle\langle a,6\rangle\langle a,2\rangle\langle c,0\rangle$$

We will see that their corresponding sequences s, s', and s'' are not the same. The sequences s_1, s'_1, and s''_1 for the first path are $abaaaa$, $baaaaa$, and $baaa$, respectively, and the s_2, s'_2, and s''_2 for the second path are $abbaa$, $bbaaa$, and bba, respectively. We can observe that

$P_1(s_1) \neq P_1(s_2)$ and $P_2(s_1) \neq P_2(s_2)$. Thus, each path should be studied separately.

What the aforementioned example illustrates is that we cannot just apply F_1, F_2, and F_3 to the language (or *family* of sequences) T_1 to yield all the tuples that violate co-observability or controllability. So, what we do is the following: for each regular expression in T_1 of the form R^*, replace R^* by R^{I_i}, i=1, 2, ..., i.e., the Kleene star is replaced by an arbitrary constant.

- If we apply the current step to the regular expression T_1, we get the new form of expression T_1, which is

  $T_1 = \langle b,1\rangle\langle b,1\rangle^{I_1}\langle a,6\rangle\langle b,5\rangle\langle b,5\rangle^{I_2}\langle a,6\rangle\langle a,6\rangle^{I_3}\langle a,4\rangle(\langle a,4\rangle^{I_4}\langle a,6\rangle^{I_5}$
  $\langle a,2\rangle^{I_6})^{I_7}\langle c,0\rangle$

The idea behind this step is that, whereas R^* stands for an arbitrary number of occurrences of the sequences in R, R^{I_i} fixes the number at some specific value and the s and s' will have to share the same index so that if, say, event a appears 6 times in s then it must appear 6 times in s' (presuming that a is observable to agent 1). Similarly, events observable by agent 2 must appear the same number of times in both s and s''.

(k) Using the following projection maps F_1, F_2, and F_3 (Section 3.1.1) and F_4, we find σ, s, s', and s'' to identify the pairs of (illegal, legal) sequences or triples of (illegal, legal, legal) sequences of events that cause failure in controllability or co-observability of $L(E)$ with respect to G:

$$F_1((\alpha,i)) = \begin{cases} \alpha \text{ if } i = 1,4,6 \\ \epsilon \text{ otherwise} \end{cases}$$

$$F_2((\alpha,i)) = \begin{cases} \alpha \text{ if } i = 2,4,5 \\ \epsilon \text{ otherwise} \end{cases}$$

$$F_3((\alpha,i)) = \begin{cases} \alpha \text{ if } i = 3,4,5,6 \\ \epsilon \text{ otherwise} \end{cases}$$

$$F_4((\alpha,i)) = \begin{cases} \alpha \text{ if } i = 0 \\ \epsilon \text{ otherwise} \end{cases}$$

For $j = 1,2,3,4$, $F_j(\epsilon) = \epsilon$ and for $s \in \Sigma^*$, $\beta \in \Sigma$, $F_j(s(\beta,i)) = F_j(s)F_j((\beta,i))$.

- For the example, we get the following s, s', and s'' in this step:

  $s = abb^{I_2}aa^{I_3}a(a^{I_4}a^{I_5})^{I_7}$

  $s' = bb^{I_1}aaa^{I_3}a(a^{I_4}a^{I_5})^{I_7}$

  $s'' = bb^{I_2}a(a^{I_4}a^{I_5})^{I_7}$

(l) Suppose s' contains regular expression R^{I_k}. If R is a regular expression where none of its events is observable by agent 1, then it is not necessary that the sequences represented by R appear the same number of times in s' as in s. In this case, regular expression R^{I_k} should be changed to R^{J_l}, where J_l is unique, and $l = 1,2,....$

Also, when s'' includes regular expression R^{I_k} and none of the events of R is observable by agent 2, by the same reasoning as above, R^{I_k} should be reduced to R^{J_l}.

- Let us consider the sequences s, s', and s'' derived in the previous step for Example 3.1. Since event a is not observable by agent 2, the event a does not need to be restricted to appear as many time as it appears in s'', that is $a(a^{I_4}a^{I_5})^{I_7}$. The number of repetitions of a in s'' , i.e., I_4, I_5 and I_7, should be changed to the new indices J_2, J_3, and J_4, respectively. Thus we can find a more

general regular expression s'', while $P_2(s) = P_2(s'')$ is still guaranteed. Similarly, this point is considered for the sequence s'. Thus, for Example 3.1, one family of sequences of events that violates co-observability of $L(E)$ with respect to G is $(s\sigma, s'\sigma, s''\sigma)$, where $s = abb^{I_2}aa^{I_3}a(a^{I_4}a^{I_5})^{I_7}$, $s' = bb^{J_1}aaa^{I_3}a(a^{I_4}a^{I_5})^{I_7}$, $s'' = bb^{I_2}a(a^{J_2}a^{J_3})^{J_4}$, $\sigma = c$, and $I_2, I_3, ..., J_1, J_2, ... \in \{0, 1, 2, ...\}$.

Had we not done step (f), string s' might have contained $(R_1 + R_2 + ...)^{I_i}$, where some R_k's, $k = 1, 2, ...$, are partly observable by agent 1 and others are not. Variable I_i could not be kept unchanged, since some R_k's are not observable by agent 1 and therefore it would not be necessary that they repeat in s' as many times as in s. So, if we keep I_i, the generality of the solution would be lost. On the other hand, I_i could not be modified to a new J_l, since the projection of those R_k's that are partially observable by agent 1 should appear in s' as many times as they appear in s. Thus, $(R_1 + R_2 + ...)^*$ should be reduced to its equivalent form, which is $(R_1^* R_2^* ...)^*$. Similar reasoning applies to the case of s''.

(m) We list the pairs of (illegal, legal) sequences or triples of (illegal, legal, legal) sequences of events and which property each pair is violating based on the following rules:

$$\begin{aligned}
\sigma \in \Sigma_{1,o}/\Sigma_{2,o} &\Rightarrow (s\sigma, s'\sigma) \quad \text{violates co-observability} \\
\sigma \in \Sigma_{2,o}/\Sigma_{1,o} &\Rightarrow (s\sigma, s''\sigma) \quad \text{violates co-observability} \\
\sigma \in \Sigma_{1,o} \cap \Sigma_{2,o} &\Rightarrow (s\sigma, s'\sigma, s''\sigma) \quad \text{violate co-observability} \\
\sigma \notin \Sigma_{1,o} \cup \Sigma_{2,o} &\Rightarrow (s\sigma, s) \quad \text{violates controllability}
\end{aligned}$$

– For the above example, since $c \in \Sigma_{1,o} \cap \Sigma_{2,o}$, we conclude that $(s\sigma, s'\sigma, s''\sigma)$ violates co-observability, where $s = abb^{I_2}aa^{I_3}a(a^{I_4}a^{I_5})^{I_7}$, $s' = bb^{J_1}aaa^{I_3}a(a^{I_4}a^{I_5})^{I_7}$, and $s'' = bb^{I_2}a(a^{J_2}a^{J_3})^{J_4}$.

Steps (e) to (g) should be done for each DRE and steps (h) to (m) must be employed for each term of it. Finally all tuples of illegal and legal sequences of events that violate controllability or co-observability of $L(E)$ with respect to G will be extracted.

3.2.2 Some-Paths Method

This section outlines how we can tailor the all-paths method of Section 3.2.1 to find a subset of all violating pairs as counterexamples for controllability or co-observability.

Automaton M may contain some families of paths. We are going to pick one of the shortest paths in each family of paths to find its corresponding pair of (illegal, legal) sequences or triple of (illegal, legal, legal) sequences of events. This can be accomplished by dropping cycles in each family of paths and picking one of the remaining paths of the family. The paths obtained by applying the some-paths method are those formed by setting indices i and j to 0 in the results obtained by the all-paths method. Since we drop the cycles of each family of paths in the some-paths method, we do not need to consider the number of repetitions of cycles, which is specified by i and j, in the all-paths method. Therefore the some-paths method contains fewer steps and is thus easier to implement than the all-paths method. Nevertheless the all-paths method finds *all* sequences of events that violate controllability or co-observability, whereas the some-paths method finds only the shortest paths in the families of paths. The following steps yield the desired results.

Steps (a) to (e) are the same as in the all-paths method. So, applying these steps to Example 3.1 results in the following:

$DRE_1 = \langle b,1\rangle\langle b,1\rangle^*\langle a,6\rangle\langle b,5\rangle\langle b,5\rangle^*\langle a,6\rangle\langle a,6\rangle^*(\langle a,4\rangle+\langle a,2\rangle)(\langle a,4\rangle+\langle a,6\rangle+\langle a,2\rangle)^*\langle c,0\rangle$

$DRE_2 = \langle b,1\rangle\langle b,1\rangle^*\langle a,6\rangle\langle b,5\rangle\langle b,5\rangle^*\langle a,2\rangle\langle a,2\rangle^*(\langle a,4\rangle+\langle a,6\rangle)(\langle a,4\rangle+\langle a,6\rangle+\langle a,2\rangle)^*\langle c,0\rangle$

$DRE_3 = \langle b,1\rangle\langle b,1\rangle^*\langle a,6\rangle\langle b,5\rangle\langle b,5\rangle^*\langle a,4\rangle(\langle a,4\rangle + \langle a,6\rangle + \langle a,2\rangle)^*\langle c,0\rangle$

Each DRE_i is the representation of a set of families of paths of the example. We illustrate the remaining steps of the method for DRE_1. The same process should be done for all DREs.

(f) If there exists an R^* in the DRE where R is a regular expression, we exchange it with $\langle\epsilon, 9\rangle$. In fact, we drop the cycles of each family of paths. As mentioned is Section 3.1.1, in automaton M a transition from one state to an adjacent state is labelled by a pair of the form $\langle$*event, transition-type*$\rangle$. In the algorithm of Section 3.1.1 (or of [21, 19]), the transition-type lies in the range 0 to 6. For implementation, we define dummy transition-type 9.

- For Example 3.1,

  $DRE'_1 = \langle b,1\rangle\langle\epsilon,9\rangle\langle a,6\rangle\langle b,5\rangle\langle\epsilon,9\rangle\langle a,6\rangle\langle\epsilon,9\rangle(\langle a,4\rangle + \langle a,2\rangle)\langle\epsilon,9\rangle$
  $\langle c,0\rangle$

(g) If the modified DRE_i, namely DRE'_i, includes $S(R_1 + R_2 + ...)$ or $(R_1 + R_2 + ...)S$, where S and the R_i's are regular expressions, then we convert it to SR_1 or R_1S, respectively, i.e., we extract one sequence in the set of sequences captured by the "+" operation. The resulting expression is the representation of one family of paths.

- For Example 3.1, the new regular expression assigned to DRE'_1 is given by

$$DRE'_1 = \langle b,1\rangle\langle\epsilon,9\rangle\langle a,6\rangle\langle b,5\rangle\langle\epsilon,9\rangle\langle a,6\rangle\langle\epsilon,9\rangle\langle a,4\rangle\langle\epsilon,9\rangle\langle c,0\rangle$$

(h) Using the following projection maps, we can find s, s', s'', and σ of the resulting regular language,

$$F_1((\alpha, i)) = \begin{cases} \alpha \text{ if } i = 1, 4, 6 \\ \epsilon \text{ otherwise} \end{cases}$$

$$F_2((\alpha, i)) = \begin{cases} \alpha \text{ if } i = 2, 4, 5 \\ \epsilon \text{ otherwise} \end{cases}$$

$$F_3((\alpha, i)) = \begin{cases} \alpha \text{ if } i = 3, 4, 5, 6 \\ \epsilon \text{ otherwise} \end{cases}$$

$$F_4((\alpha, i)) = \begin{cases} \alpha \text{ if } i = 0 \\ \epsilon \text{ otherwise} \end{cases}$$

For $j = 1, 2, 3, 4$, $F_j(\epsilon) = \epsilon$ and for $s \in \Sigma^*$, $\beta \in \Sigma$, $F_j(s(\beta, i)) = F_j(s)F_j((\beta, i))$. Then s, s', s'', and σ are determined by

$$\begin{aligned} s' &= F_1(DRE_i') \\ s'' &= F_2(DRE_i') \\ s &= F_3(DRE_i') \\ \sigma &= F_4(DRE_i') \end{aligned}$$

– Applying this step to DRE_1' of step (g) results in the following:

$$\begin{aligned} s' &= baaa \\ s'' &= ba \\ s &= abaa \\ \sigma &= c \end{aligned}$$

(k) If we consider the set that σ belongs to, we can extract and list the pairs of (illegal, legal) sequence or triples of (illegal, legal, legal) sequences of events

and determine whether the pair violates controllability or co-observability, by applying the rules in step (m) of the all-paths method.

- For the example, $\sigma = c \in \Sigma_{1,c} \cap \Sigma_{2,c}$, so the triple $(s\sigma, s'\sigma, s''\sigma)$ violates co-observability.

Steps (e) to (k) should be done for each DRE.

The method gives a list of some (illegal, legal) pairs or (illegal, legal, legal) triples, where each pair or triple corresponds to a family of sequences that violate controllability or co-observability. Moreover, the condition (namely, controllability or co-observability) that is violated can be determined.

3.3 Computational Complexity

We have implemented the some-paths method, because it was easier to implement than the all-paths method. The some-paths method contains 9 steps. The asymptotic computational complexity of each step is determined.

(a) It is shown in [21, 19] that the complexity of this step is $O(n^4)$, where $n = \max\{e, g\}$ and e and g are the number of states in automata E and G, respectively.

(b) Let m be the number of reachable and co-reachable states in automaton M. We can conclude m is at most n^4. The computational complexity of the algorithm in [25] to convert a finite-state machine to its corresponding regular expression is $O(m^3)$. So, the complexity of this step is $O(m^3) = O(n^{12})$.

(c) If L is the number of symbols of the regular expression resulting from the previous step, the complexity of converting it from infix format to postfix

format is $O(L)$, and the complexity of reducing the postfix expression to the binary tree is $O(L \log L)$.

(d) This step requires traversal of all states of the binary tree to acquire the result. If the number of states of the binary tree is L, then the complexity is $O(L \log L)$.

(e) The complexity of converting a postfix expression to its corresponding binary tree is $O(d \log d)$ and the complexity of reducing the binary tree to its infix expression is $O(d \log d)$, where d is the number of symbols of DRE.

The complexity of the algorithms for each of steps (f), (g), and (h) is $O(d)$ and it is $O(1)$ for step (k). Steps (e), (f), (g), (h), and (k) should be done for all DREs. In the worst case, the complexity of the method is $O(n^{12})$, where $n = \max\{e, g\}$ and e and g are the number of states in automata E and G, respectively. The necessary resources, such as time and space to run the algorithms, can be decreased by some manipulations in the algorithm at each step during implementation, as follows:

- By generating trimmed M, we can get rid of unnecessary states of M. Consequently, the program does not occupy any memory for the unnecessary states and does not need extra time to process them in the other steps of the method.

- The automata E and G used to construct M can be *minimized*. The automaton M constructed using the minimized E and G may have fewer states than the automaton M constructed using the original E and G.

- During conversion of the finite-state machine to a regular expression in step (b), some simplifications are considered. Therefore, the length of

the regular expression generated by the algorithm could be made smaller. Accordingly, less space and shorter time is used by the program to process it.

Also the following programming techniques are considered in the implementation to decrease the memory occupied while running the program.

- During the implementation of the data structures *linked lists* are used instead of fixed length arrays. So, the implemented program allocates only as much memory as it needs.

- In the implementation, *variable length* variables are used instead of fixed length variables. Therefore, memory is not wasted by a long fixed length variable of which just a portion is used.

- Memory occupied by variables is released when it is not used anymore.

The some-paths method is a polynomial-time algorithm with computational complexity $O(n^{12})$. Even considering the above points in the implementation, since time and space are not infinite resources, for very large n the program may either halt because of insufficient memory or continue to run for an unreasonably long time. Depending on the strategies used in the program, one of these two cases may happen: insufficient memory or unreasonably long running time. This will be explored in more detail in Chapter 4.

Steps (a) and (b) of the all-paths method are the determining factors in computing the asymptotic computational complexity. In other words, steps (a) and (b) result in the largest exponent in the polynomial bound on computing time of the all-paths method. These steps are the same as those in the some-paths method. Therefore, the asymptotic computational complexity of the all-paths method is the same as that of the some-paths method (namely, $O(n^{12})$).

Chapter 4

Examples

In Section 4.1 of this chapter, controllability and co-observability of some examples are verified using the procedure of Section 3.2.2. When controllability or co-observability does not hold for an example, the program yields some tuples of illegal and legal sequences of events that cause the failure of controllability or co-observability.

Telecommunication protocols can be expressed as decentralized control problems of discrete-event systems. So, the existence of two agents with a reliable communication protocol between them for data transmission over an unreliable channel can be checked. This is explained in [24, 23, 18, 22] and restated in Section 4.2. Then, some protocols in the data-link layer are introduced. The data-link layer is a layer defined in the OSI architecture for telecommunication networks. Therefore, data link layer protocols are a subset of telecommunication protocols. As a result, they can be verified using decentralized DES problem formulations. The protocols in Section 4.2 are studied, modelled by discrete-event system concepts, and verified manually.

Based on the algorithm in [21, 19], two methods have been presented in the previous chapter. The program is the implementation of one of them, namely the

some-paths method. The program can be applied to detect failures caused by lack of co-observability or controllability in a decentralized problem in which its description is given as global specifications and its solution requires decentralized control. The program can be used to verify a communication protocol and determine the cause of the failure. Given the computational complexity of the program, when the number of states in automaton E or automaton G of the input example is a very large number, either very large memory or a very long time is necessary to run the program. Memory and time are not unlimited resources. So, we cannot expect to get the output results for a large input during a limited time and with limited memory. This is argued in Section 4.3.

4.1 Simple Examples

In this section, controllability and co-observability of some examples are checked using the method of Section 3.2.2.

For each example the following 6 files are given to the program as inputs:

• Automaton G which represents the plant.

• Automaton E, where $L(E)$ is a prefix-closed language, $L(E) \subseteq L(G)$, and $L(E)$ is a regular language. Any sequence which is in $L(E)$ is called a legal sequence. Otherwise, if it belongs to $L(G) \backslash L(E)$, it is called an illegal sequence.

• The controllable event set of agent 1, namely $\Sigma_{1,c}$.

• The controllable event set of agent 2, denoted by $\Sigma_{2,c}$.

• The observable event set of agent 1, denoted by $\Sigma_{1,o}$.

• The observable event set of agent 2, denoted by $\Sigma_{2,o}$.

The output of the program is a list of some pairs or triples of illegal and legal sequences of events. Each listed tuple violates either co-observability or controllability of $L(E)$ with respect to G. In addition, the program indicates which condition is violated, namely, co-observability or controllability.

4.1.1 Violation of co-observability due to agent 2

This example shows how co-observability is violated when agent 2 needs to disable an event but does not have sufficient observations to know that the event should be disabled. This is depicted in Figure 4.1.

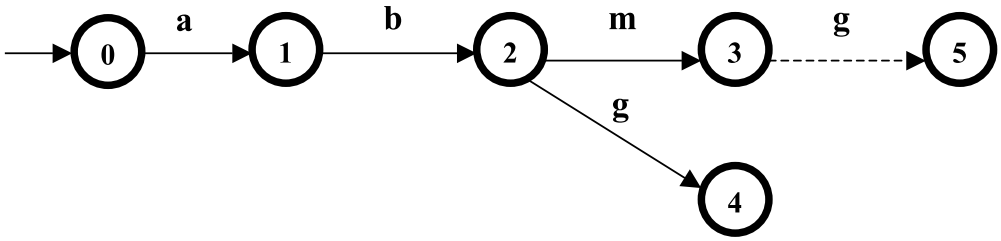


Figure 4.1: An example where co-observability is violated because agent 2 does not have enough observations.

Input:

- Automata E and G depicted in Figure 4.1
- $\Sigma_{1,c} = \{b, m\}$
- $\Sigma_{2,c} = \{g, b\}$
- $\Sigma_{1,o} = \{b, g, m\}$
- $\Sigma_{2,o} = \{a, g\}$

Output:

- $s = abm$, $s'' = ab$, $\sigma = g$

Since $\sigma \in \Sigma_{2,c} \backslash \Sigma_{1,c}$, the pair $(s\sigma, s''\sigma)$ violates co-observability of $L(E)$ with respect to G .

4.1.2 Violation of co-observability due to agents 1 and 2

This examples shows how co-observability is violated when an event should be disabled, but neither agent 1 nor agent 2 have sufficient observations to

determine that the event should be disabled. The example is from [17, example 3.8].

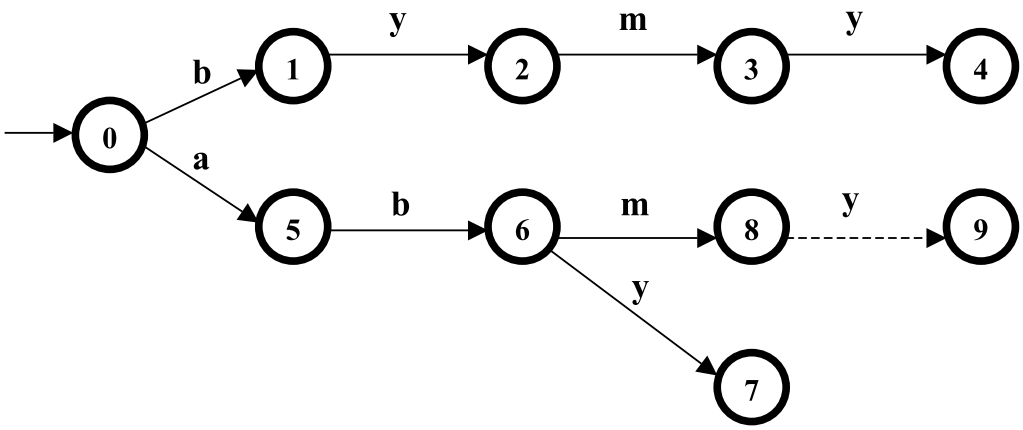


Figure 4.2: An example where co-observability is violated because neither agent 1 nor agent 2 has enough observations.

Input:

- Automata E and G depicted in Figure 4.2
- $\Sigma_{1,c} = \{a, y\}$
- $\Sigma_{2,c} = \{b, m, y\}$
- $\Sigma_{1,o} = \{a\}$
- $\Sigma_{2,o} = \{b, m\}$

Output:

- $s = abm$, $s' = ab$, $s'' = bym$, $\sigma = y$

Since $\sigma \in \Sigma_{1,c} \cap \Sigma_{2,c}$, the triple $(s\sigma, s'\sigma, s''\sigma)$ violates co-observability of $L(E)$ with respect to G .

4.1.3 The triples resulting from the some-paths method and the all-paths method

This examples shows how co-observability is violated when an event should be disabled, but neither agent 1 nor agent 2 have sufficient observations to determine that the event should be disabled. Furthermore, since the finite-state machine has a self-loop, it can be shown that there is also another triple that violates co-observability and can be extracted by applying the all-paths method, but is not produced by the some-paths method.

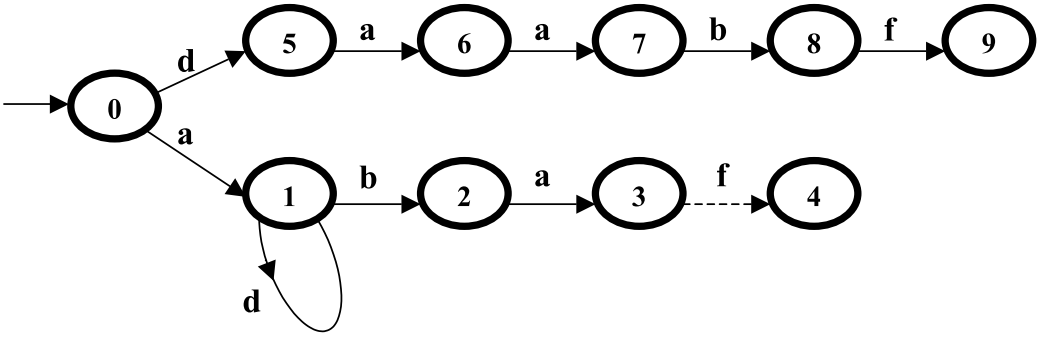


Figure 4.3: An example where co-observability is violated due to agents 1 and 2 both having insufficient observations.

Input:

- Automata E and G depicted in Figure 4.3
- $\Sigma_{1,c} = \{a, f\}$
- $\Sigma_{2,c} = \{a, f, d, b\}$
- $\Sigma_{1,o} = \{a, f\}$
- $\Sigma_{2,o} = \{b\}$

Output:

- $s = aba$, $s' = daab$, $s'' = daab$, $\sigma = f$

Since $\sigma \in \Sigma_{1,c} \cap \Sigma_{2,c}$, the triple $(s\sigma, s'\sigma, s''\sigma)$ violates co-observability of $L(E)$

with respect to G. The following triple of sequences is another counterexample to co-observability of $L(E)$ with respect to G. This triple is a subset of the output of the all-paths method, but not the some-paths method.

- $s = adba$, $s' = daab$, $s'' = daab$, $\sigma = f$

4.1.4 Multiple triples that violate co-observability

This example shows how multiple triples that violate co-observability can be extracted by applying the some-paths method.

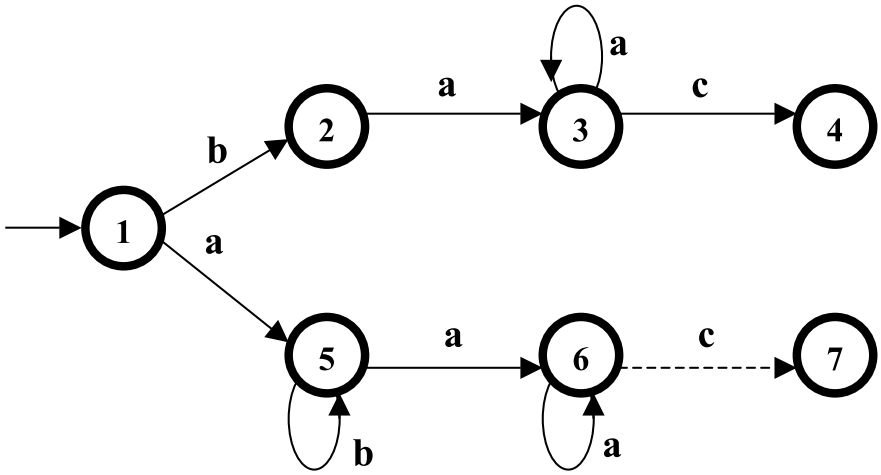


Figure 4.4: An example where multiple triples violate co-observability.

Input:

- Automata E and G depicted in Figure 4.4
- $\Sigma_{1,c} = \Sigma_{2,c} = \{c\}$
- $\Sigma_{1,o} = \{a\}$
- $\Sigma_{2,o} = \{b\}$

Output:

- $s = aba$, $s' = baa$, $s'' = baa$, $\sigma = c$
- $s = abaa$, $s' = baaa$, $s'' = ba$, $\sigma = c$
- $s = aba$, $s' = baa$, $s'' = ba$, $\sigma = c$

In each case, $\sigma \in \Sigma_{1,c} \cap \Sigma_{2,c}$, therefore $(s\sigma, s'\sigma, s''\sigma)$ violates co-observability of $L(E)$ with respect to G.

4.1.5 Triples and pairs that violate co-observability or controllability

This example represents a pair that violates co-observability, because agent 2 does not have sufficient observation to disable an event that should be disabled, and agent 2 is the only agent that can disable the event. Also, there exists another event that should be disabled, but neither agent 1 nor agent 2 has sufficient observations to disable it. It is shown that we can also find a pair of sequences in this example that violates controllability.

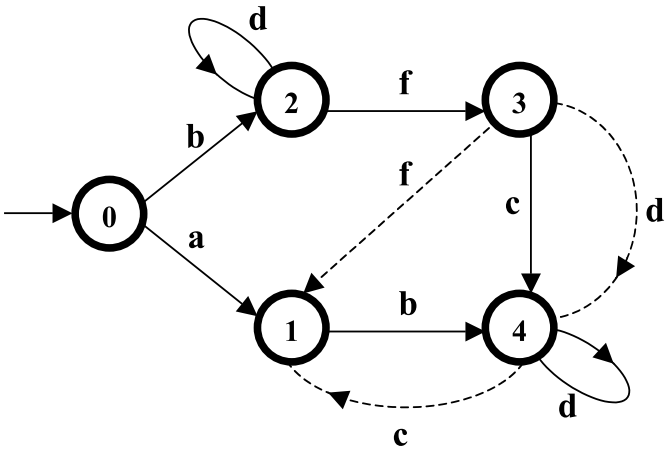


Figure 4.5: An example where both co-observability and controllability are violated.

Input:

- Automata E and G depicted in Figure 4.5
- $\Sigma_{1,c} = \{a, d\}$
- $\Sigma_{2,c} = \{c, d\}$
- $\Sigma_{1,o} = \{a\}$
- $\Sigma_{2,o} = \{b\}$

Output:

- $s = bf$, $\sigma = f$

The pair $(s\sigma, s)$ violates controllability of $L(E)$ with respect to G.

- $s = ab$, $s'' = bf$, $\sigma = c$
- $s = bfc$, $s'' = bf$, $\sigma = c$

In each case, $\sigma \in \Sigma_{2,c} \backslash \Sigma_{1,c}$, therefore $(s\sigma, s''\sigma)$ violates co-observability of $L(E)$ with respect to G.

- $s = bf$, $s' = b$, $s'' = ab$, $\sigma = d$
- $s = bf$, $s' = b$, $s'' = b$, $\sigma = d$

In each case, $\sigma \in \Sigma_{2,c} \cap \Sigma_{1,c}$, therefore $(s\sigma, s'\sigma, s''\sigma)$ violates co-observability of $L(E)$ with respect to G.

4.2 Telecommunication Protocols and Decentralized Supervisory Control

The problem of data transmission between some agents over an unreliable channel in the data link layer of OSI network architecture can be described as a decentralized control problem of discrete-event systems. Various people have looked at this problem from a DES control perspective [24, 18, 13, 5, 22, 8, 9]. In [24], it was shown how the failure of a protocol corresponds to a violation of

co-observability and then a protocol, namely an erroneous version of the alternating bit protocol, was verified. Similarly, a simplified version of Go-Back-n protocol was checked in [4]. In this section, we model three more complicated protocols of the data link layer. Theoretically, the all-paths method given in Section 3.2.1 can be applied to extract all sequences of events that cause failure of each protocol, and the some-paths method finds some of these sequences. However, due to the size of the protocols, the computational complexity of the methods, and the available resources, it is not feasible to apply the methods. Therefore, we verify each protocol by manually finding a single problematic sequence of events that causes failure of the protocol.

In the above problems, the existence of some distributed agents that communicate with each other via a channel is studied. Each agent can make control decisions on a subset of events and can observe a subset of events of the system. The desired behavior in a telecommunication problem is interpreted as reliable transmission of data over an unreliable channel. Despite the fact that the distributed agents make separate observation and control decisions, their joint behavior should produce the desired behavior, which is a subset of all possible sequences of events that may happen in the plant. Therefore, a protocol should be determined for the agents to specify each agent's behavior. Suppose a protocol is suggested for such a system. We study whether the protocol restricts the system's behavior correctly to guarantee the reliable data transmission over an unreliable channel. We concentrate on problems with prefix-closed desired languages. So, their corresponding communication protocol never fails because of blocking. Thus, we can say the answer to the question of existence of agents is positive if and only if the candidate communication protocol does not fail because of one of the following reasons: the desired language either is not controllable with respect to the plant or is not co-observable with respect to the plant.

Given a candidate protocol, we specify the plant and the desired language and verify the protocol by checking whether the desired language is controllable and co-observable with respect to the plant. Moreover, the problematic sequences of events that cause the failure of the protocol can be identified. They are the same sequences that violate controllability or co-observability of the desired language with respect to the plant. In theory, the program, which is the implementation of the some-paths method, can be applied to find some problematic sequences, but this process cannot be done in a reasonable amount of time or with limited memory. Therefore, for each protocol, we indicate a single sequence of events that causes a failure of the protocol.

4.2.1 SW1 Protocol

Protocol Description

A poor channel wastes a lot of bandwidth on retransmitted data frames. A strategy for handling errors is to allow the receiver to accept and buffer the frames following a damaged or lost one. The SW1 protocol does not discard frames merely because an earlier frame was damaged or lost. In the protocol, both sender and receiver maintain a window of acceptable sequence numbers, namely in the set {0, 1, 2, 3}. The window size of the sender and the receiver equals the maximum sequence number, that is, 3. The receiver has a buffer reserved for each sequence number within its window. Associated with each buffer, there exists a bit indicating whether the buffer is full or empty. Whenever a frame arrives, its sequence number is checked to see if it falls within the window. If so, and if it has not already been received, it is accepted and stored. This action is taken whether or not it is the next packet expected by the network layer. In the SW1 protocol, the accepted frame must be kept within the data link layer and not passed to the network layer until all the lower-numbered

frames have been delivered to the network layer in the correct order.

A timer is associated with each buffer of the sender. When a data frame is sent to the physical layer, the timer corresponding to the data frame is started. If the sender's timer elapses before the sender receives an acknowledgment for the sent frame, then the sender will presume that the frame sent has been damaged or lost and will retransmit the frame. We have two agents; namely, agents A and B. Each agent consists of two modules: a sender and a receiver. The acknowledgment from receiver B to sender A is piggybacked to the data frames sent from sender B to receiver A, and vice versa.

The SW1 protocol is a slightly modified version of the sliding window protocol introduced in [26, 28]. Each of the senders and the receivers of the SW1 protocol includes four different states for windows. The protocol causes transitions from one state to another. The states of the windows are **a**, **b**, **c**, and **d**. State **a** contains 3 buffers for the data frames whose sequence numbers are 0, 1, and 2. Similarly, states **b**, **c**, and **d** refer to windows with 3 buffers for data frames with sequence numbers (3, 0, 1), (2, 3, 0), and (1, 2, 3), respectively. Let us consider the window of the sender of an agent. When the window state is **a**, it waits to receive 3 consecutive data frames from the higher level, then sends them to the lower level and sets the timer. The sender advances the window to the new state, that is, state **b**, when it receives an acknowledgment for the transmitted data with sequence number 2. Similarly, the sender advances state **b** to state **c**, state **c** to state **d**, and state **d** to state **a**. On the other hand, the receiver gets data frames whose sequence numbers fall within the range of sequence numbers considered acceptable by its current window. Then, it transfers the ordered data frames to the network layer. For example, if the window is (1, 2, 3) and data frame 3 is received from the physical layer, the receiver waits for data frames 1 and 2 until it sends them in the correct order to the network

layer. But if data frame 1 had been received, it would have sent it right away to the network layer. When a data frame is sent to the network layer, a timer is started. There exists only one timer for each receiver. If the timer is needed when it is already running, the timer is reset to the full timeout interval. The acknowledgment of the recent state of the window is piggybacked to the data frames being sent. This piggybacked package is transferred from the current agent to the other agent. If the acknowledgment is not ready to be piggybacked to the data frames being sent, the previous acknowledgment is retransmitted by piggybacking. We call the repeated acknowledgment a *dummy* acknowledgment. Also, if the receiver recognizes that a data frame that it received is damaged, the receiver piggybacks a NAK to the data frames being sent. The receiver then advances its window. When the current state of the window is **a**, it is moved to state **b**. In the same way, the state can be moved from **b** to **c**, or **c** to **d**, or **d** to **a**.

Solution Overview

The SW1 protocol is designed to restrict the behavior of the communication system to a desired behavior. However, the protocol should be verified to determine whether its resulting restricted behavior is acceptable. In other words, the designer should know if the protocol fails or not. If the restricted behavior is either not controllable or not co-observable with respect to the plant, then the protocol fails. In order to verify the protocol using the methods of this thesis, each of the restricted behaviors and the plant should be modelled by an automaton.

At first glance, the protocol seems too complicated to model by an equivalent finite-state machine directly. Thus, the problem of protocol modelling should be broken into several small problems. Each problem can be studied modularly.

We can design the finite-state machine of each module and then combine them using operations defined on automata such as union, shuffle, Kleene-closure, intersection, and an operation we define called *shuffle-c*.

Finally, we obtain two automata. The two resulting automata, namely the legal language that so far we have called "restricted behavior" and the plant, represent the performance of the protocol.

In this section, automata that are part of the construction of the model of the protocol are presented. We explain how to combine these automata modules to create two larger automata that will capture the plant and the legal behavior. We identify the events that are controllable and observable by each of the two agents in the communication system and then check controllability and co-observability of the legal language with respect to the plant. For the SW1 protocol, some of the problematic sequences of events that violate controllability or co-observability can be found manually. Nevertheless, it is not easy to find all such sequences manually. We find sequences that violate co-observability and show how these sequences relate to failure in the SW1 protocol.

Shuffle-c Operation ($\|_c$)

To facilitate creating the automata that describe the protocol, we define an operation on automata that combines automata to form a larger composite structure. Consider two automata G_1 and G_2 as follows:

$$G_1 = (X_1, \Sigma_1, \zeta_1, x_{0,1}, X_{m,1})$$

$$G_2 = (X_2, \Sigma_2, \zeta_2, x_{0,2}, X_{m,2})$$

The *shuffle-c* ($\|_c$) of the automata G_1 and G_2 is the following nondeterministic automaton:

$$G_1 \|_c G_2 = (X_1 \times X_2, \Sigma_1 \cup \Sigma_2, \delta_c, (x_{0,1}, x_{0,2}), X_{m,1} \times X_{m,2})$$

where

$$\delta_c((x_1, x_2), e) = \begin{cases} (\delta_1(x_1, e), x_2) & \text{if } \delta_1(x_1, e) \text{ is defined} \\ (x_1, \delta_2(x_2, e)) & \text{if } \delta_2(x_2, e) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

For example, if $L(G_1) = \alpha\beta$ and $L(G_2) = \beta\gamma$, shuffle-c yields an automaton $G_1 \parallel_c G_2$, where $L(G_1 \parallel_c G_2) = \{\alpha\beta\beta\gamma, \beta\gamma\alpha\beta, \beta\alpha\gamma\beta, \beta\alpha\beta\gamma, \alpha\beta\gamma\beta\}$. Consider the two automata *CHNL-m* and *CHNL-l* in Figure 4.6. We observe that *abcc* would not be generated by the standard synchronous product operation of [32], but *abcc* is legitimate channel behavior. This is why we use shuffle-c to describe channel behavior; *abcc* $\in$ (*CHNL-m* $\parallel_c$ *CHNL-l*). Since shuffle-c can result in a nondeterministic automaton, in an implementation that uses shuffle-c, the resulting automaton would need to be converted to an equivalent deterministic automaton before the other supervisory control functions could be applied to it.

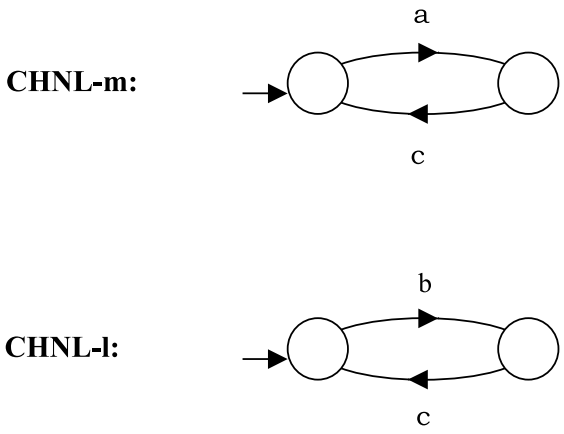


Figure 4.6: Automata *CHNL-m* and *CHNL-l*

Event Table

For the protocol SW1, there are two agents, namely agents A and B. Each agent is able to control a set of events while observing a set of events. To verify the protocol, we refer to the four event sets described as follows:

- The controllable event set of agent A ($\Sigma_{c,A}$)
- The controllable event set of agent B ($\Sigma_{c,B}$)
- The observable event set of agent A ($\Sigma_{o,A}$)
- The observable event set of agent A ($\Sigma_{o,B}$)

Table 4.1 contains the definitions of the symbols used in Table 4.2. Table 4.2 lists all events of the protocol, a short description of each event, and indicates whether the event is in $\Sigma_{c,A}, \Sigma_{c,B}, \Sigma_{o,A}$, and/or $\Sigma_{o,B}$. A similar table could be defined for agent B.

Table 4.1: Symbols

Symbol	**Meaning**
!Bi	The set containing data frames with any legal sequence number j of the sender of agent A, namely Aj, where $j \neq i$.
dummy ack	A dummy value that can be used instead of acknowledgment.
ackBj	The acknowledgment of the data frame received with sequence number j by the receiver of agent A.
NAK	The acknowledgment that the data frame received by the receiver of agent A is damaged.
ackB*	Dummy acknowledgment, NAK, or the acknowledgment of a data frame received with any legal sequence number.
!ackBj	The set containing Dummy acknowledgment, NAK, and ackBi for all legal i, where $i \neq j$.

Table 4.2: Event Table

Event Name	**Description**	**Membership in $\Sigma_{c,A}$, $\Sigma_{c,B}$, $\Sigma_{o,A}$, and $\Sigma_{o,B}$**
gfn-Ai	Agent A gets a frame from the network layer, and assigns the sequence number i to it.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
set-timer-p-Ai	The sender of agent A sets the timer of the frame with sequence number i, after sending the data frame to the physical layer.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
time-out-p-Ai	In the sender of agent A, the timer of the frame with sequence number i elapses.	$\Sigma_{o,A}$
imp-snd-wind-a-A	The sender of agent A advances its window from d to a.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
imp-snd-wind-b-A	The sender of agent A advances its window from a to b.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
imp-snd-wind-c-A	The sender of agent A advances its window from b to c.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
imp-snd-wind-d-A	The sender of agent A advances its window from c to d.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
time-out-n-A	The timer of the receiver of agent A elapses.	$\Sigma_{o,A}$
set-timer-n-A	The timer of the receiver of agent A is started.	$\Sigma_{c,A}$, $\Sigma_{o,A}$

Event Name	**Description**	**Membership in $\Sigma_{c,A}$, $\Sigma_{c,B}$, $\Sigma_{o,A}$, and $\Sigma_{o,B}$**
stn-Bi	Agent A sends a data frame with sequence number i to the network layer. This is the same data frame that was sent from the sender of agent B to the receiver of agent A.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
lose	The channel loses the data while transmitting it from one agent to the other agent.	None
stp(Ai, ackBj)	Agent A piggybacks ackBj onto the data frame with sequence number i of the sender. Then, agent A sends the piggybacked information to the physical layer.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
stp(A*, ackBj)	Agent A piggybacks ackBj onto the data frame with any legal sequence number of the sender or a dummy value. Then, agent A sends the piggybacked information to the physical layer.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
stp(dummy, ackBj)	Agent A piggybacks ackBj onto a dummy value. Then, agent A sends the piggybacked information to the physical layer.	$\Sigma_{c,A}$, $\Sigma_{o,A}$

Event Name	**Description**	**Membership in $\Sigma_{c,A}$, $\Sigma_{c,B}$, $\Sigma_{o,A}$, and $\Sigma_{o,B}$**
stp(Ai, ackB*)	Agent A piggybacks ackB* onto the data frame with sequence number i of the sender. Then, agent A sends the piggybacked information to the physical layer.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
stp(A*, ackB*)	Agent A piggybacks ackB* onto the data frame with any legal sequence number of the sender or a dummy value. Then, agent A sends the piggybacked information to the physical layer.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
stp(dummy, ackB*)	Agent A piggybacks ackB* onto a dummy value. Then, agent A sends the piggybacked information to the physical layer.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
stp(Ai, !ackBj)	Agent A piggybacks !ackBj onto the data frame with sequence number i of the sender. Then, agent A sends the piggybacked information to the physical layer.	$\Sigma_{c,A}$, $\Sigma_{o,A}$

Event Name	**Description**	**Membership in $\Sigma_{c,A}$, $\Sigma_{c,B}$, $\Sigma_{o,A}$, and $\Sigma_{o,B}$**
stp(A*, !ackBj)	Agent A piggybacks !ackBj onto the data frame with any legal sequence number of the sender or a dummy value. Then, agent A sends the piggybacked information to the physical layer.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
stp(dummy, !ackBj)	Agent A piggybacks !ackBj onto a dummy value. Then, agent A sends the piggybacked information to the physical layer.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
stp(Ai, NAK)	Agent A piggybacks NAK onto the data frame with sequence number i of the sender. Then, agent A sends the piggybacked information to the physical layer.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
stp(A*, NAK)	Agent A piggybacks NAK onto the data frame with any legal sequence number of the sender or a dummy value. Then, agent A sends the piggybacked information to the physical layer.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
stp(dummy, NAK)	Agent A piggybacks NAK onto a dummy value. Then, agent A sends the piggybacked information to the physical layer.	$\Sigma_{c,A}$, $\Sigma_{o,A}$

Event Name	**Description**	**Membership in $\Sigma_{c,A}$, $\Sigma_{c,B}$, $\Sigma_{o,A}$, and $\Sigma_{o,B}$**
stp(Ai, dummy ack)	Agent A piggybacks a dummy acknowledgment onto the data frame with sequence number i of the sender. Then, agent A sends the piggybacked information to the physical layer.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
stp(A*, dummy ack)	Agent A piggybacks a dummy acknowledgment onto the data frame with any legal sequence number of the sender or a dummy value. Then, agent A sends the piggybacked information to the physical layer.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
gfp(Bi, ackAj)	Agent A gets the data sent from agent B via the physical layer. The data contains the data frame with sequence number i of the sender of agent B and ackAj.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
gfp(B*, ackAj)	Agent A gets the data sent from agent B via the physical layer. The data contains a data frame with any legal sequence number of the sender of agent B or a dummy value, and ackAj.	$\Sigma_{c,A}$, $\Sigma_{o,A}$

Event Name	**Description**	**Membership in $\Sigma_{c,A}$, $\Sigma_{c,B}$, $\Sigma_{o,A}$, and $\Sigma_{o,B}$**
gfp(dummy, ackAj)	Agent A gets the data sent from agent B via the physical layer. The data contains a dummy value and ackAj.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
gfp(!Bi, ackAj)	Agent A gets the data sent from agent B via the physical layer. The data contains !Bi and ackAj.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
gfp(Bi, ackA*)	Agent A gets the data sent from agent B via the physical layer. The data contains the data frame with sequence number i of the sender of agent B and ackA*.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
gfp(B*, ackA*)	Agent A gets the data sent from agent B via the physical layer. The data contains a data frame with any legal sequence number of the sender of agent B or a dummy value, and ackA*.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
gfp(dummy, ackA*)	Agent A gets the data sent from agent B via the physical layer. The data contains a dummy value and ackA*.	$\Sigma_{c,A}$, $\Sigma_{o,A}$

Event Name	**Description**	**Membership in $\Sigma_{c,A}$, $\Sigma_{c,B}$, $\Sigma_{o,A}$, and $\Sigma_{o,B}$**
gfp(!Bi, ackA*)	Agent A gets the data sent from agent B via the physical layer. The data contains !Bi and ackA*.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
gfp(Bi, *NAK*)	Agent A gets the data sent from agent B via the physical layer. The data contains the data frame with sequence number i of the sender of agent B and *NAK*.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
gfp(dummy, *NAK*)	Agent A gets the data sent from agent B via the physical layer. The data contains a dummy value and *NAK*.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
gfp(B*, *NAK*)	Agent A gets the data sent from agent B via the physical layer. The data contains a data frame with any legal sequence number of the sender of agent B or a dummy value, and *NAK*.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
gfp(!Bi, *NAK*)	Agent A gets the data sent from agent B via the physical layer. The data contains !Bi and *NAK*.	$\Sigma_{c,A}$, $\Sigma_{o,A}$

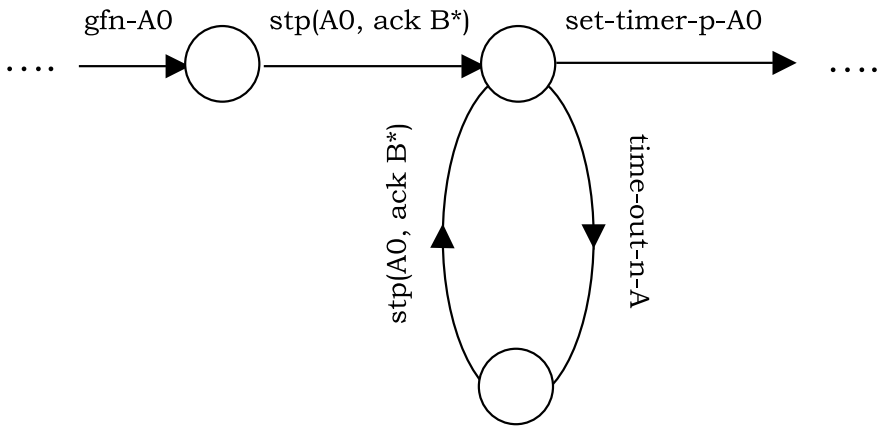
Event Name	**Description**	**Membership in $\Sigma_{c,A}$, $\Sigma_{c,B}$, $\Sigma_{o,A}$, and $\Sigma_{o,B}$**
gfp(Bi, !ackAj)	Agent A gets the data sent from agent B via the physical layer. The data contains the data frame with sequence number i of the sender of agent B and !ackAj.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
gfp(B*, !ackAj)	Agent A gets the data sent from agent B via the physical layer. The data contains a data frame with any legal sequence number of the sender of agent B or a dummy value, and !ackAj.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
gfp(dummy, !ackAj)	Agent A gets the data sent from agent B via the physical layer. The data contains a dummy value and !ackAj.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
gfp(!Bi, !ackAj)	Agent A gets the data sent from agent B via the physical layer. The data contains !Bi and !ackAj.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
gfp(!Bi, dummy ack)	Agent A gets the data sent from agent B via the physical layer. The data contains !Bi and a dummy acknowledgment.	$\Sigma_{c,A}$, $\Sigma_{o,A}$

Event Name	**Description**	**Membership in $\Sigma_{c,A}$, $\Sigma_{c,B}$, $\Sigma_{o,A}$, and $\Sigma_{o,B}$**
gfp(Bi, dummy ack)	Agent A gets the data sent from agent B via the physical layer. The data contains the data frame with sequence number i of the sender of agent B, and a dummy acknowledgment.	$\Sigma_{c,A}$, $\Sigma_{o,A}$
gfp(B*, dummy ack)	Agent A gets the data sent from agent B via the physical layer. The data contains a data frame with any legal sequence number of the sender of agent B or a dummy value, and a dummy acknowledgment.	$\Sigma_{c,A}$, $\Sigma_{o,A}$

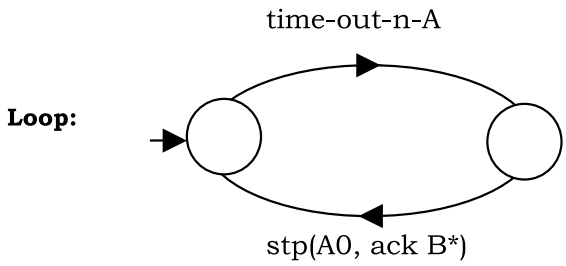
Protocol Model

The following is an explanation of how the SW1 protocol can be modelled. We study the protocol SW1 modularly, and then address each module separately in order to design its corresponding automaton. The following explanations are provided to aid in the interpretation of the subsequent figures.

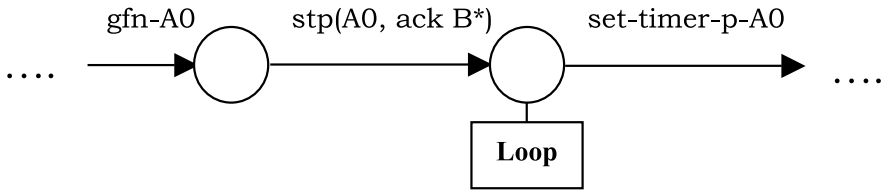
- The automaton in Figure 4.7(a) can be represented by the two structures shown in Figures 4.7(b) and 4.7(c). That is, the cycle of transitions in 4.7(a) is replaced by a rectangular box in 4.7(c). The rectangular box is the representation of the automaton in 4.7(b). In this way, an automaton that would span several pages can be decomposed into smaller figures, each of which fits on one page. We use this strategy to depict automaton



(a)



(b)



(c)

Figure 4.7: Decomposition of a finite-state machine into two smaller structures

A.SNDR-t which spans several pages.

- In the automata represented in Figures 4.8-4.40 some edges are labelled by names which are not listed in Table 4.2. These labels are described in Table 4.3 and stand for sets of events. In an automaton, an edge labelled by a set of events should be interpreted as multiple edges, each of which is labelled by an event from the set.

We make a slight abuse of notation and sometimes write X to stand for both an automaton X and the language recognized by the automaton, $L_m(X)$. So, for example, when we write

$$Y = X \cap V$$

for automata X and V, we mean that Y is an automaton that recognizes the intersection of the language recognized by X and the language recognized by V.

Table 4.3: The meaning of some labels

Label	**Meaning**
Sn-1	$\Sigma_{A.SNDR} \backslash \{$*imp-snd-wind-a-A*$\}$
Sn-2	$\Sigma_{A.SNDR} \backslash \{$*imp-snd-wind-a-A, imp-snd-wind-b-A, gfp(B*, ackA0), gfp(B*, ackA1)*$\}$
Sn-3	$\Sigma_{A.SNDR} \backslash \{$*imp-snd-wind-a-A, imp-snd-wind-b-A, gfp(B*, ackA0), gfp(B*, ackA1), time-out-p-A0*$\}$
Sn-4	$\Sigma_{A.SNDR} \backslash \{$*imp-snd-wind-a-A, imp-snd-wind-b-A, gfp(B*, ackA0), gfp(B*, ackA1), time-out-p-A0, time-out-p-A1*$\}$
Sn-5	$\Sigma_{A.SNDR} \backslash \{$*imp-snd-wind-b-A*$\}$
Sn-6	$\Sigma_{A.SNDR} \backslash \{$*imp-snd-wind-b-A, imp-snd-wind-c-A, gfp(B*, ackA3), gfp(B*, ackA0)*$\}$
Sn-7	$\Sigma_{A.SNDR} \backslash \{$*imp-snd-wind-b-A, imp-snd-wind-c-A, gfp(B*, ackA3), gfp(B*, ackA0), time-out-p-A3*$\}$
Sn-8	$\Sigma_{A.SNDR} \backslash \{$*imp-snd-wind-b-A, imp-snd-wind-c-A, gfp(B*, ackA3), gfp(B*, ackA0), time-out-p-A3, time-out-p-A0*$\}$
Sn-9	$\Sigma_{A.SNDR} \backslash \{$*imp-snd-wind-c-A*$\}$
Sn-10	$\Sigma_{A.SNDR} \backslash \{$*imp-snd-wind-c-A, imp-snd-wind-d-A, gfp(B*, ackA2), gfp(B*, ackA3)*$\}$
Sn-11	$\Sigma_{A.SNDR} \backslash \{$*imp-snd-wind-c-A, imp-snd-wind-d-A, gfp(B*, ackA2), gfp(B*, ackA3), time-out-p-A3*$\}$
Sn-12	$\Sigma_{A.SNDR} \backslash \{$*imp-snd-wind-c-A, imp-snd-wind-d-A, gfp(B*, ackA2), gfp(B*, ackA3), time-out-p-A3, time-out-p-A0*$\}$
Sn-13	$\Sigma_{A.SNDR} \backslash \{$*imp-snd-wind-d-A*$\}$

Label	**Meaning**
Sn-14	$\Sigma_{A.SNDR}\backslash\{$*imp-snd-wind-d-A, imp-snd-wind-a-A, gfp(B*, ackA1), gfp(B*, ackA2)*$\}$
Sn-15	$\Sigma_{A.SNDR}\backslash\{$*imp-snd-wind-d-A, imp-snd-wind-a-A, gfp(B*, ackA1), gfp(B*, ackA2), time-out-p-A2*$\}$
Sn-16	$\Sigma_{A.SNDR}\backslash\{$*imp-snd-wind-d-A, imp-snd-wind-a-A, gfp(B*, ackA1), gfp(B*, ackA2), time-out-p-A2, time-out-p-A3*$\}$
piggy	$\{$*gfp(!B0, ackA*), gfp(!B1, ackA*), gfp(!B2,ackA*), gfp(!B3, ackA*), gfp(dummy, ackA*)*$\}$
Event-1(i), $i = (n+1)(m+1)$, $n = \{0,1,2,3\}$, and $m = \{0,1,2,3\}$	$\{$*stp(A_n, $ackB_m$)*$\}$
Event-2(i), $i = (n+1)(m+1)$, $n = \{0,1,2,3\}$, and $m = \{0,1,2,3\}$	$\{$*gfp(A_n, $ackB_m$), gfp(!A_n, $ackB_m$), gfp(A_n, !$ackB_m$), gfp(!A_n, !$ackB_m$), lose*$\}$
Event-1(i), $i = n + 17$, $n = \{0,1,2,3\}$	$\{$*stp(A_n, NAK)*$\}$
Event-2(i), $i = n + 17$, $n = \{0,1,2,3\}$	$\{$*gfp(A_n, ackB*), gfp(!A_n, ackB*), lose*$\}$
Event-1(i), $i = n + 21$, $n = \{0,1,2,3\}$	$\{$*stp(A_n, dummy ack)*$\}$

Label	**Meaning**
Event-2(i), $i = n + 21$, $n = \{0, 1, 2, 3\}$	{*gfp(*A_n*, dummy ack), gfp(!*A_n*, dummy ack), lose*}
Event-1(i), $i = m + 25$, $m = \{0, 1, 2, 3\}$	{*stp(dummy,* $ackB_m$*)*}
Event-2(i),	{*gfp(dummy,* $ackB_m$*), gfp(dummy, !*$ackB_m$*), lose*}
$i = m + 25$, $m = \{0, 1, 2, 3\}$	
Event-1(i), $i = (n + 1)(m + 1) + 28$, $n = \{0, 1, 2, 3\}$, and $m = \{0, 1, 2, 3\}$	{*stp(*B_n*,* $ackA_m$*)*}
Event-2(i), $i = (n + 1)(m + 1) + 28$, $n = \{0, 1, 2, 3\}$, and $m = \{0, 1, 2, 3\}$	{*gfp(*B_n*,* $ackA_m$*), gfp(!*B_n*,* $ackA_m$*), gfp(*B_n*, !*$ackA_m$*), gfp(!*B_n*, !*$ackA_m$*), lose*}
Event-1(i), $i = n + 45$, $n = \{0, 1, 2, 3\}$	{*stp(*B_n*, NAK)*}
Event-2(i), $i = n + 45$, $n = \{0, 1, 2, 3\}$	{*gfp(*B_n*, NAK), gfp(!*B_n*, NAK), lose*}

Label	**Meaning**
Event-1(i), $i = n + 49$, $n = \{0, 1, 2, 3\}$	$\{stp(B_n,\ dummy\ ack)\}$
Event-2(i), $i = n + 49$, $n = \{0, 1, 2, 3\}$	$\{gfp(B_n,\ dummy\ ack),\ gfp(!B_n,\ dummy\ ack),\ lose\}$
Event-1(i), $i = m + 53$, $m = \{0, 1, 2, 3\}$	$\{stp(dummy,\ ackA_m)\}$
Event-2(i), $i = m + 53$, $m = \{0, 1, 2, 3\}$	$\{gfp(dummy,\ ackA_m),\ gfp(dummy,\ !ackA_m),\ lose\}$
Le(1)	$\Sigma \backslash \{gfn\text{-}A0,\ gfn\text{-}A1,\ gfn\text{-}A2,\ gfn\text{-}A3\}$
Le(2)	$\Sigma \backslash \{stn\text{-}A0,\ stn\text{-}A1,\ stn\text{-}A2,\ stn\text{-}A3\}$
Le(3)	$\Sigma \backslash \{stn\text{-}B0,\ stn\text{-}B1,\ stn\text{-}B2,\ stn\text{-}B3\}$
Le(4)	$\Sigma \backslash \{gfn\text{-}B0,\ gfn\text{-}B1,\ gfn\text{-}B2,\ gfn\text{-}B3\}$
Le(5)	$\Sigma \backslash \{gfn\text{-}A0,\ stn\text{-}A0\}$
Le(6)	$\Sigma \backslash \{gfn\text{-}A1,\ stn\text{-}A1\}$
Le(7)	$\Sigma \backslash \{gfn\text{-}A2,\ stn\text{-}A2\}$
Le(8)	$\Sigma \backslash \{gfn\text{-}A3,\ stn\text{-}A3\}$
Le(9)	$\Sigma \backslash \{gfn\text{-}B0,\ stn\text{-}B0\}$
Le(10)	$\Sigma \backslash \{gfn\text{-}B1,\ stn\text{-}B1\}$
Le(11)	$\Sigma \backslash \{gfn\text{-}B2,\ stn\text{-}B2\}$
Le(12)	$\Sigma \backslash \{gfn\text{-}B3,\ stn\text{-}B3\}$

The automata are explained as follows.

- *Automata L-0 to L-3, Figure 4.8*

  In the following, we explain *L-0*. The other blocks have a similar description. One of these scenarios occurs in *L-0*:

  - The sender of agent A gets an acknowledgement for data frame A0.
  - The sender of agent of A receives a damaged acknowledgement.
  - The timer of data frame A0 elapses. In this case, the sender piggybacks an acknowledgment onto data frame A0, sends A0 and the acknowledgement to the physical layer, and starts the timer of data frame A0.

- *Automata M-0 to M-3, Figure 4.9*

  In the following, we explain *M-0*. The other blocks have a similar description. Either of these scenarios may occur in *M-0*:

  - The sender of agent of A receives a damaged acknowledgement.
  - The timer of data frame A0 elapses. In this case, the sender piggybacks an acknowledgment onto data frame A0, sends A0 plus the acknowledgement to the physical layer, and starts the timer of data frame A0.

- *Automata T-0 to T-3, Figure 4.10*

  Automata *T-0* to *T-3* are explained as follows. The timer of the receiver of agent A elapses. Agent A piggybacks the acknowledgement, which is ready to be sent, onto the last transmitted data frame. Agent A then sends these to the physical layer.

- *Automata A.SNDR-t, Figures 4.11 and 4.12*

  When the sender of agent A advances its window to **a** (state A in Figures 4.11 and 4.12), the following sequence of events happens. The sender receives data frame A0 from the network layer, piggybacks an acknowledgment onto it, and sends the data frame plus the acknowledgement to the physical layer. Then, the timer corresponding to data frame A0 is started. The same scenario happens for data frames A1 and A2. Then, if the sender gets the acknowledgment for data frame 2 while the window is **a** (between states A and B in Figure 4.11), the window is advanced to **b** (state B). The sequence of events that may happen while the window is **b** (between states B and C in Figure 4.11), while the window is **c** (between states C and D in Figure 4.12), and while the window is **d** (between states D and A in Figure 4.12) follow the same logic as that for window **a**. Note that *T-0* to *T-3*, *L-0* to *L-3*, and *M-0* to *M-3* may occur as indicated in the rectangles in Figures 4.11 and 4.12.

- *Automata A.SNDR-a, A.SNDR-b, A.SNDR-c, and A.SNDR-d, Figures 4.13 and 4.14*

  Assume the window of the sender of agent A is advanced to **a**. Before the sender of agent A advances the window to **b**, if agent A receives an acknowledgment of data frame A0, the timer of data frame A0 does not time out and the sender does not wait for another acknowledgment of data frame A0 anymore. If agent A receives an acknowledgment of data frames A1 or A2, the timer of data frames A0 or A1 does not time out and the sender does not wait for the acknowledgment of data frames A0 or A1.

  Automata *A.SNDR-b* in Figure 4.13 and *A.SNDR-c* and *A.SNDR-d* in Figure 4.14 are similar to automaton *A.SNDR-a*.

- *Automaton A.RCVR, Figures 4.15-4.26*

  Suppose the window at the receiver has advanced to **a** (state E in Figures 4.15-4.17, and 4.24-4.26). First, let us follow the sequence of events that occur throughout the path on the right side between states E and F in Figure 4.15. In this path, the receiver gets a packet containing data frame B0, sends it to the network layer, starts the timer of the receiver, and piggybacks the acknowledgment of B0 onto a data frame. Agent A sends this packet to the physical layer. Then, the receiver gets two consecutive data packets containing data frames B2 and B1. The receiver sends data frames B1 and then B2 to the network layer, starts the timer of the receiver, piggybacks the acknowledgment of B2 to a data frame, and agent A sends the packet to the physical layer. Then, the window is advanced to **b** (state F in Figures 4.15-4.17, and 4.18-4.20).

  Now we consider the cycle of transitions in Figure 4.15 going from state S to state T to state U and back to state S. This cycle represents the following sequence of events: The receiver of agent A may get a new packet containing data frame B0 or B3. Neither B0 nor B3 is the correct data frame to be received in this state of the system. The receiver starts the timer and then piggybacks *NAK* onto a data frame, which should be sent from agent A to agent B. Agent A sends the packet containing *NAK* to the physical layer. Using this strategy, the receiver of agent A discards the wrong data frames and informs agent B of this.

  The self-loop at state S in Figure 4.15 can be explained as follows. If the sender of agent A is ready to send a data frame to the physical layer, while the receiver is in state S, the receiver piggybacks a dummy acknowledgement onto the data frame and agent A sends the packet containing the dummy acknowledgement to the physical layer.

The other cycles of three transitions and self-loops in automaton *A.RCVR* (Figures 4.15-4.26) have similar explanations to the aforementioned cycle and self-loop that start at state S.

There are five other possible ways that the events associated with the following actions can be interleaved when the window of the receiver is **a**: the receiver gets B0, B1, and B2, sends them to the network layer, starts the timer of the receiver, and sends the acknowledgments corresponding to data frames B0, B1, or B2 to the physical layer, and finally advances the window to **b**. The preceding actions may place in all possible permutations of order. These six paths are shown in Figures 4.15-4.17. Similarly, Figures 4.18-4.20, 4.21-4.23, and 4.24-4.26 demonstrate the cases where the window is between states F and G, G and H, and H and E, respectively.

- *Automaton A.piggyback, Figures 4.27-4.29*

  This automaton is represented in five parts, each of which is a finite-state machine (FSM). They all start from a common state P. In the first FSM in Figure 4.27, it is shown that when agent A gets a packet of data containing B0, the timer of the receiver is started, agent A piggybacks the acknowledgment of B0 onto the last data frame received from the network layer (which could be A0, A1, A2, or A3), and then sends the data package to the physical layer. But if the timer of the receiver elapses before the timer gets a data frame from the network layer, then agent A piggybacks the acknowledgment of B0 to a dummy data frame and sends this package to the physical layer. The second FSM in Figure 4.27 and the two FSMs in Figure 4.28 capture the above scenario if we replace B0 by B1, B2, or B3. Finally, the FSM presented in Figure 4.29 indicates that when a data package containing a damaged or dummy data frame is received by agent A, the sequences of events that occur are the same as in the aforementioned

cases, except that agent A piggybacks *NAK* to a data frame and sends this data package to the physical layer.

- *Automaton CHNL(i), Figure 4.30*

  In Figure 4.30, it is shown that after *Event-1(i)* its corresponding *Event-2(i)* happens. For example, if *Event-1(i)* = {*stp*(A_n, *dummy ack*)}, then *Event-2(i)* ∈ {*gfp*(A_n, *dummy ack*), *gfp*(!A_n, *dummy ack*), *lose*}. There are 56 possible pais of (*Event-1(i)*, *Event-2(i)*). In other words, Figure 4.30 captures automata *CHNL(1)*, *CHNL(2)*, ..., and *CHNL(56)*.

- *Automata Order-A-SND-a, Order-A-SND-b, Order-A-SND-c, and Order-A-SND-d, Figures 4.31 and 4.32*

  Suppose the window of the sender of agent A is **a**. Automaton *Order-A-SND-a* guarantees that the data frames received from the network layer by the sender of agent A are A0, A1, and A2 in the order specified. Similarly, automaton *Order-A-SND-b* in Figure 4.31 and automata *Order-A-SND-c* and *Order-A-SND-d* in Figure 4.32 present the order of the reception of the data frames from the network layer by the sender of agent A when the window of the sender is **b**, **c**, or **d**.

- *Automata Order-B-RCVR-a, Order-B-RCVR-b, Order-B-RCVR-c, and Order-B-RCVR-d, Figures 4.33 and 4.34*

  Assume the window of the receiver of agent B is **a**. Automaton *Order-B-RCVR-a* guarantees that the data frames being sent to the network layer by the receiver of agent B are A0, A1, and A2, in this order. Similarly, automaton *Order-B-RCVR-b* in Figure 4.33 and automata *Order-B-RCVR-c* and *Order-B-RCVR-d* in Figure 4.34 present the order of the transmission of the data frames to the network layer by the receiver of agent B when the window of the receiver is **b**, **c**, or **d**.

- *Automata Order-A-RCVR-a, Order-A-RCVR-b, Order-A-RCVR-c, and Order-A-RCVR-d, Figures 4.35 and 4.36*

  Automata *Order-A-RCVR-a*, *Order-A-RCVR-b*, *Order-A-RCVR-c*, and *Order-A-RCVR-d* are the dual of automata *Order-B-RCVR-a*, *Order-B-RCVR-b*, *Order-B-RCVR-c*, and *Order-B-RCVR-d*, respectively.

- *Automata Order-B-SND-a, Order-B-SND-b, Order-B-SND-c, and Order-B-SND-d, Figures 4.37 and 4.38*

  Automata *Order-B-SND-a*, *Order-B-SND-b*, *Order-B-SND-c*, and *Order-B-SND-d* are the dual of automata *Order-A-SND-a*, *Order-A-SND-b*, *Order-A-SND-c*, and *Order-A-SND-d*, respectively.

- *Automata SND-A0, SND-A1, SND-A2, and SND-A3, Figure 4.39*

  These automata guarantee that any data frame received from the network layer by agent A should be sent to the network layer by agent B.

- *Automata SND-B0, SND-B1, SND-B2, and SND-B3, Figure 4.40*

  Automata *SND-B0*, *SND-B1*, *SND-B2*, and *SND-B3* are the dual of automata *SND-A0*, *SND-A1*, *SND-A2*, and *SND-A3*, respectively.

We combine the modular automata to form two automata representing the legal language and the plant. Let us start with automaton *A.SNDR-t* which is shown in Figures 4.8-4.12. The automaton *A.SNDR-t* is designed to represent the behavior of a part of the sender of agent A. This automaton shows when the sender may get a frame from the network layer and may send it to the physical layer and also how the sender advances its current window to a new window.

The automata *A.SNDR-a*, *A.SNDR-b*, *A.SNDR-c*, and *A.SNDR-d* are represented in Figure 4.13 and Figure 4.14. These four automata guarantee that in-

valid timeouts do not happen in agent A. Now, we construct automaton *A.SNDR* as follows:

$A.SNDR = A.SNDR\text{-}t \cap A.SNDR\text{-}a \cap A.SNDR\text{-}b \cap A.SNDR\text{-}c \cap A.SNDR\text{-}d$

Given an automaton K, we define Σ_K as the set of events contained in automaton K. By this definition,

$\Sigma_{A.SNDR} = \Sigma_{A.SNDR-t} \cup \Sigma_{A.SNDR-to} \cup \Sigma_{A.SNDR-a} \cup \Sigma_{A.SNDR-b} \cup \Sigma_{A.SNDR-c} \cup \Sigma_{A.SNDR-d}$.

Using the set of events contained in automaton $A.RCVR$, that is, $\Sigma_{A.RCVR}$, we can define the following event set:

$$\Sigma_{Self-A.SNDR} = \Sigma_{A.RCVR} - \Sigma_{A.SNDR}$$

We add self-loops of the events of $\Sigma_{Self-A.SNDR}$ to the states of the automaton *A.SNDR* to construct a new automaton that we call *A-SENDER*. The automaton *A-SENDER* represents the behavior of the sender of agent A.

Next, we model the receiver of agent A. Automaton *A.RCVR* shown in Figures 4.15, 4.16, 4.17, 4.18, 4.19, 4.20, 4.24, 4.25, 4.26, is the representation of the behavior of the receiver of agent A. This automaton illustrates under which conditions the receiver may get a frame from the physical layer, send it to the network layer, and send its corresponding acknowledgement to the other agent. Also the automaton represents how the window of the receiver advances. We define the set $\Sigma_{Self-A.RCVR} = \Sigma_{A.SNDR} - \Sigma_{A.RCVR}$. Then, we add self-loops of the events in $\Sigma_{Self-A.RCVR}$ to the states of the automaton *A.RCVR* to produce a new automaton that we call automaton *A-RECEIVER*. This automaton represents the behavior of the receiver of agent A.

So far, we have designed two automata which capture two modules of agent A: the sender and the receiver. Any acknowledgment of the frame received from agent B to agent A is piggybacked onto the frame being sent from agent A to agent B. Automaton *A.piggyback* is shown in Figures 4.27, 4.28, and 4.29 and

guarantees the correct relation between the sender and the receiver of agent A. We define the automaton A using the following combination:

$$A = \textit{A-SENDER} \cap \textit{A-RECEIVER} \cap \textit{A.piggyback}.$$

Automaton A characterizes the sequences of events that may happen in agent A. In the same way, we construct automaton B which represents the sequences of events that may happen in agent B. Events in set Σ_B are the dual of events in set Σ_A. Agent A and agent B communicate with each other over a channel. Automata *CHNL(i)*, where $i = 1, 2, \cdots, 56$, are provided in Figure 4.30. Using the shuffle-c operation, we construct the automaton

$$\textit{CHNL(1)} \parallel_c \textit{CHNL(2)} \parallel_c \textit{CHNL(3)} \parallel_c \ldots \parallel_c \textit{CHNL(56)}$$

and convert it to an equivalent deterministic automaton, called *CHNL*. The automaton *CHNL* represents the behavior of the channel. As a matter of fact, the sequence $a^i c^i$ is physically possible in a wireless channel, but this specification of the channel cannot be represented by a regular expression. Therefore, we cannot represent this using finite-state machines. Using Petri Nets concepts, we would be able to model the actual channel.

Using automata A, B, and *CHNL*, we construct the plant as follows:

$$\textit{Plant} = A \parallel B \parallel \textit{CHNL}$$

$$\Sigma = \Sigma_A \cup \Sigma_B \cup \Sigma_{\textit{CHNL}}$$

Now, we determine the legal language: the aim of the protocol SW1 is to have reliable data transmission between two agents over an unreliable channel and data frames should be received by the network layer from an agent in the same order as they were sent from the network layer to the other agent. We provide 24 automata, shown in Figures 4.31-4.40, that capture the protocol requirements. We combine the automata as follows:

Legal-A-SND-a = *SND-A0* $\cap$ *SND-A1* $\cap$ *SND-A2* $\cap$ *Order-A-SND-a* $\cap$ *Order-B-RCVR-a*

Legal-A-SND-b = *SND-A3* $\cap$ *SND-A0* $\cap$ *SND-A1* $\cap$ *Order-A-SND-b* $\cap$ *Order-B-RCVR-b*

Legal-A-SND-c = *SND-A2* $\cap$ *SND-A3* $\cap$ *SND-A0* $\cap$ *Order-A-SND-c* $\cap$ *Order-B-RCVR-c*

Legal-A-SND-d = *SND-A1* $\cap$ *SND-A2* $\cap$ *SND-A3* $\cap$ *Order-A-SND-d* $\cap$ *Order-B-RCVR-d*

Legal-A-SND = (*Legal-A-SND-a* $\cup$ *Legal-A-SND-b* $\cup$ *Legal-A-SND-c* $\cup$ *Legal-A-SND-d*)*.

Similarly, we find automaton *Legal-B-SND*. Then, we construct automaton *Legal* using the following combination:

$$Legal = Legal\text{-}A\text{-}SND \parallel Legal\text{-}B\text{-}SND$$

The automaton representing the overall legal language, taking into account possible plant behavior, is

$$Legal\text{-}Language = Plant \cap Legal.$$

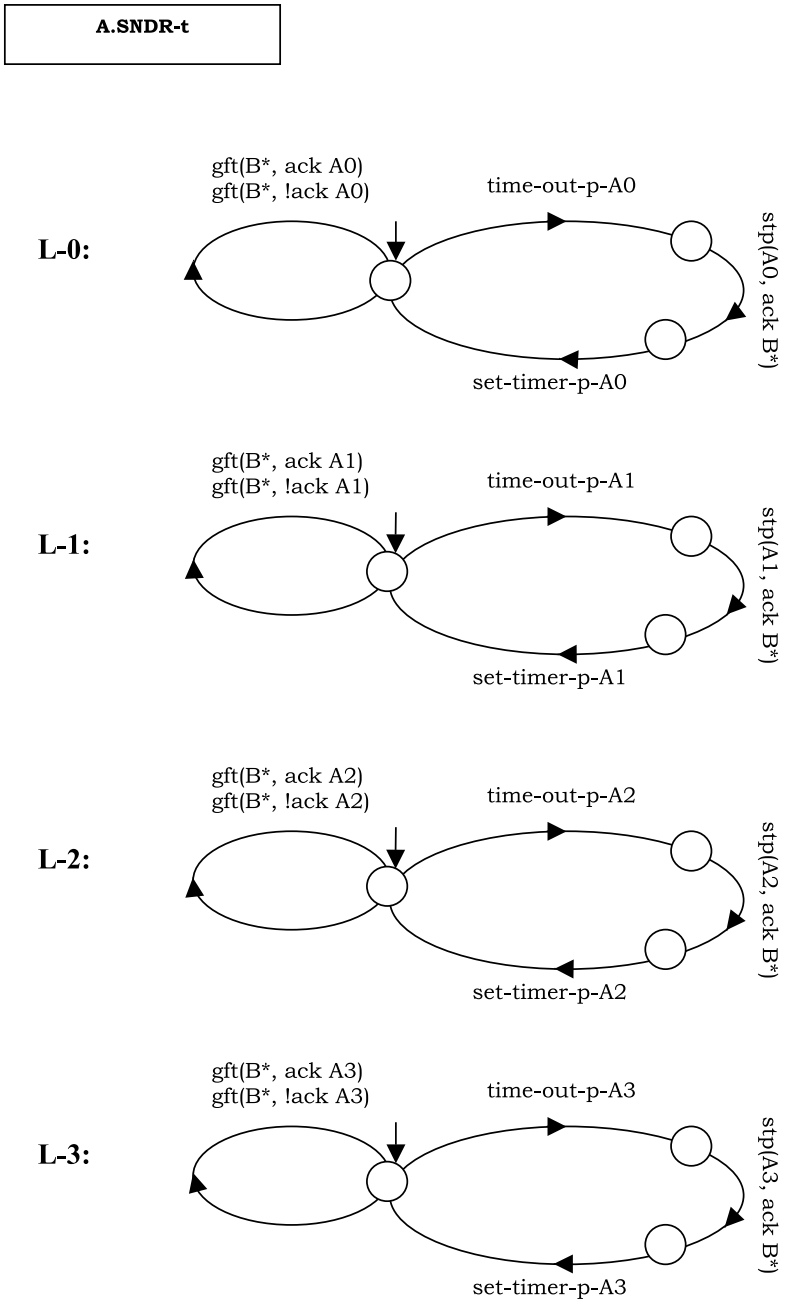


Figure 4.8: Automata *L-0*, *L-1*, *L-2*, and *L-3*, which are applied in automaton *A.SNDR-t*

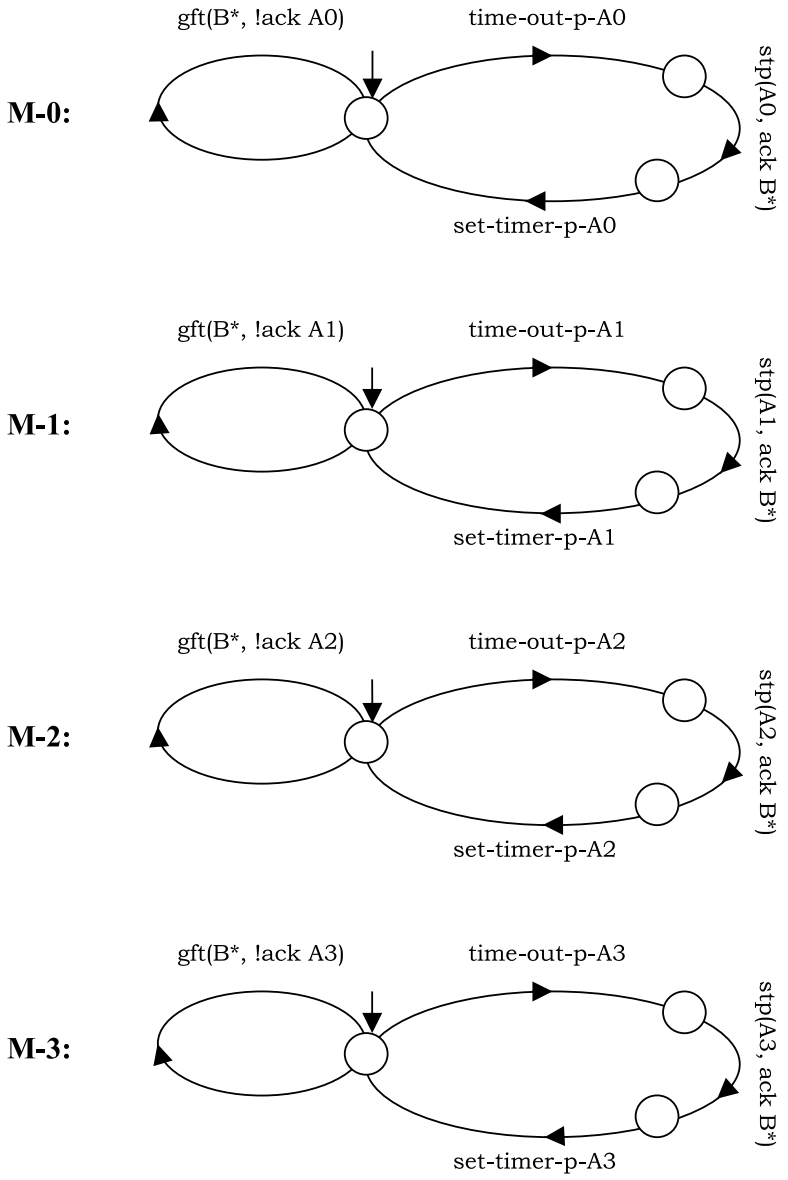


Figure 4.9: Automata *M-0*, *M-1*, *M-2*, and *M-3*, which are applied in automaton *A.SNDR-t*

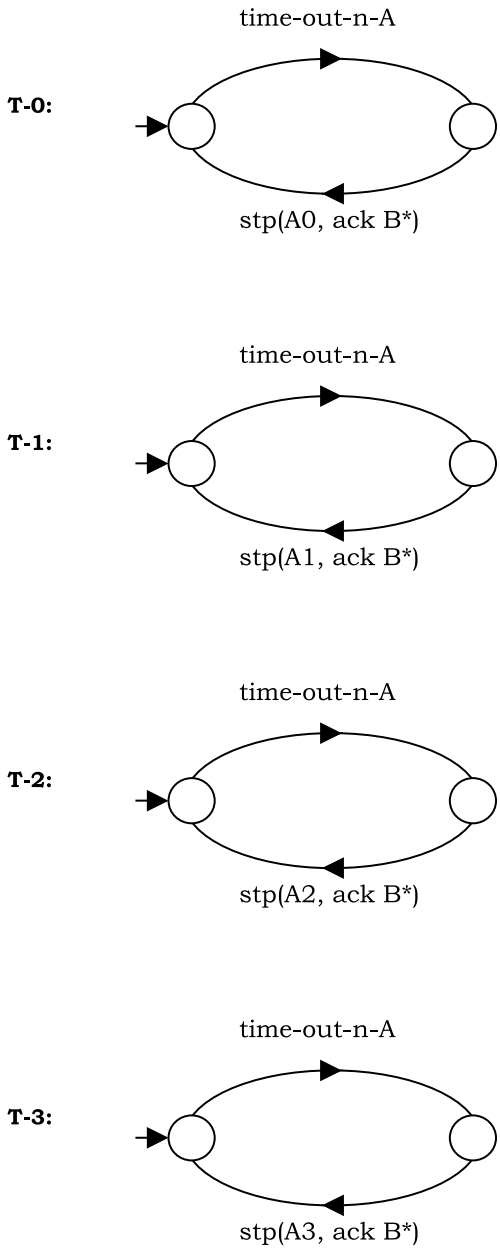


Figure 4.10: Automata *T-0*, *T-1*, *T-2*, and *T-3*, which are applied in automaton *A.SNDR-t*

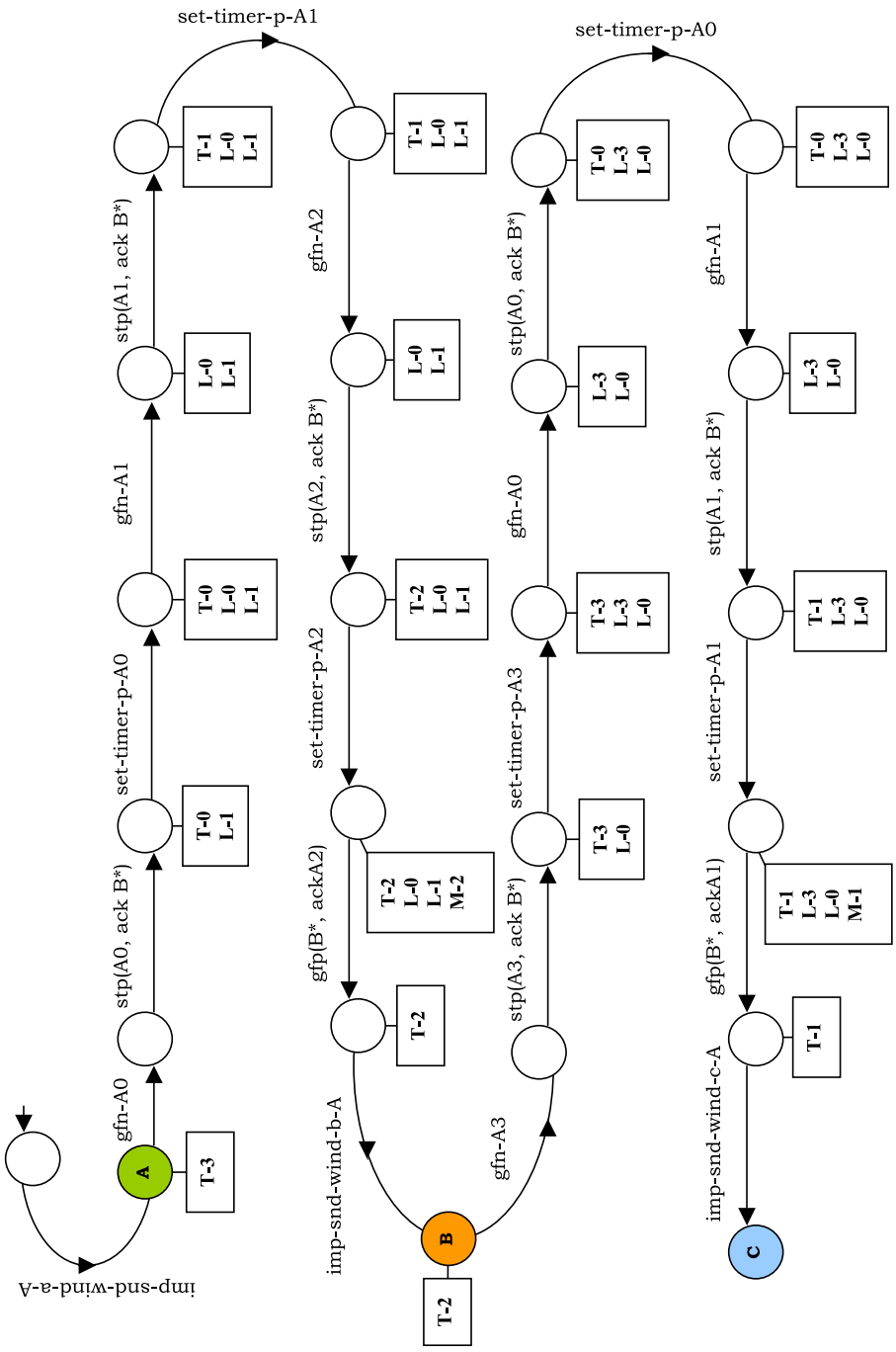


Figure 4.11: Automaton *A.SNDR-t*, page 1 of 2

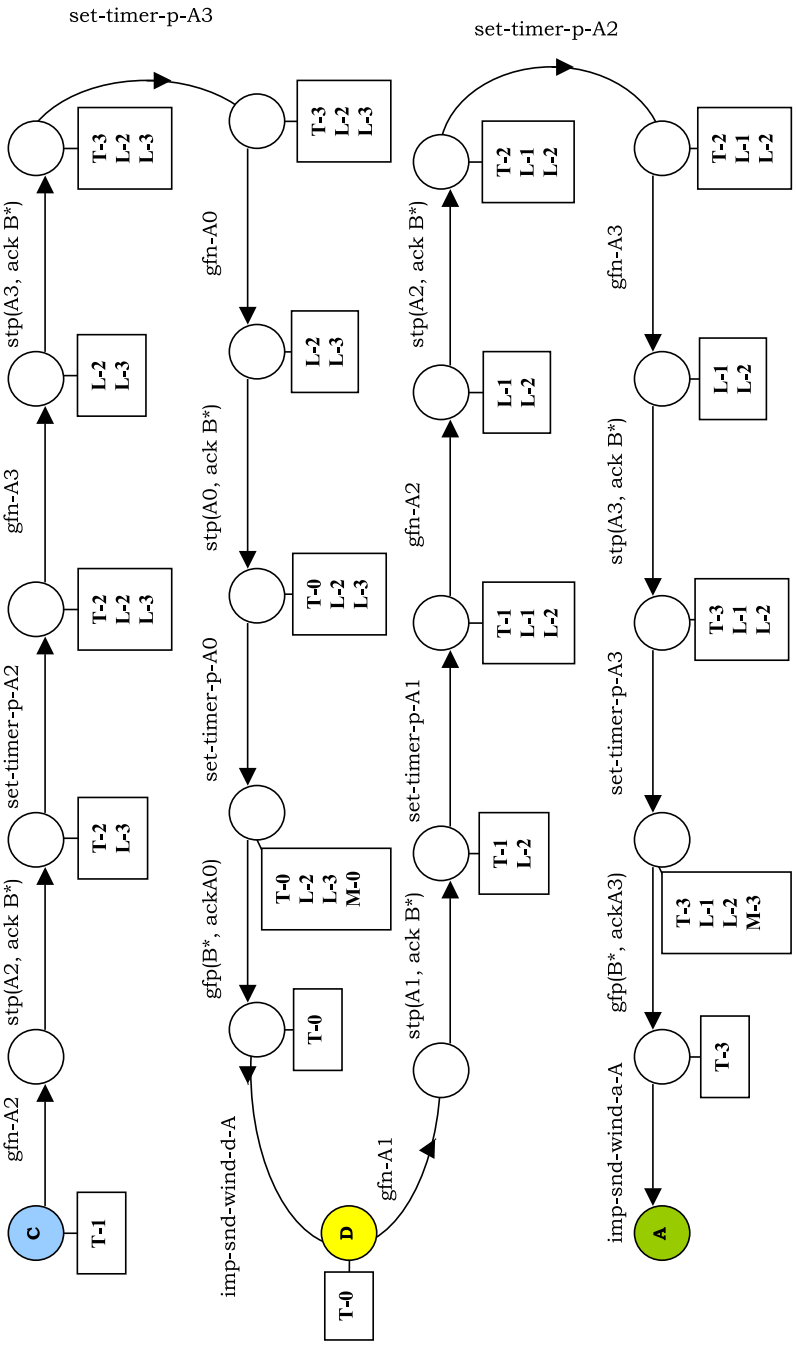


Figure 4.12: Automaton *A.SNDR-t*, page 2 of 2

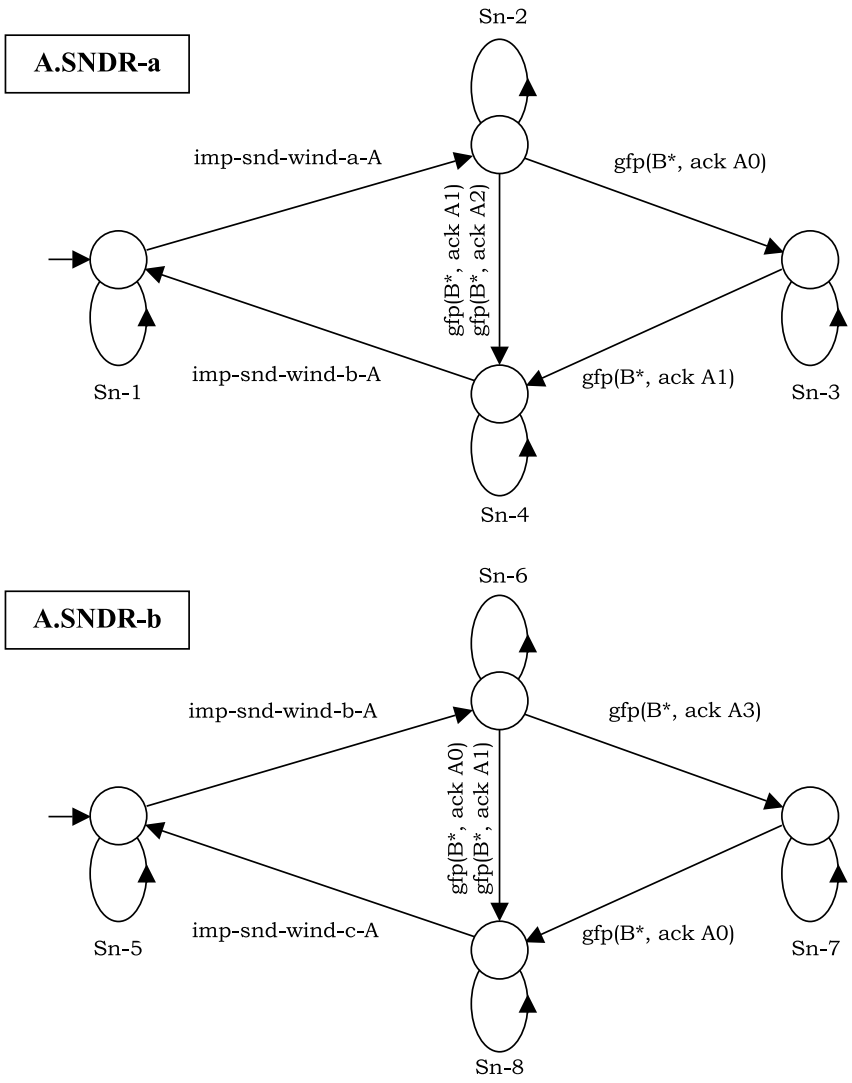


Figure 4.13: Automata *A.SNDR-a* and *A.SNDR-b*

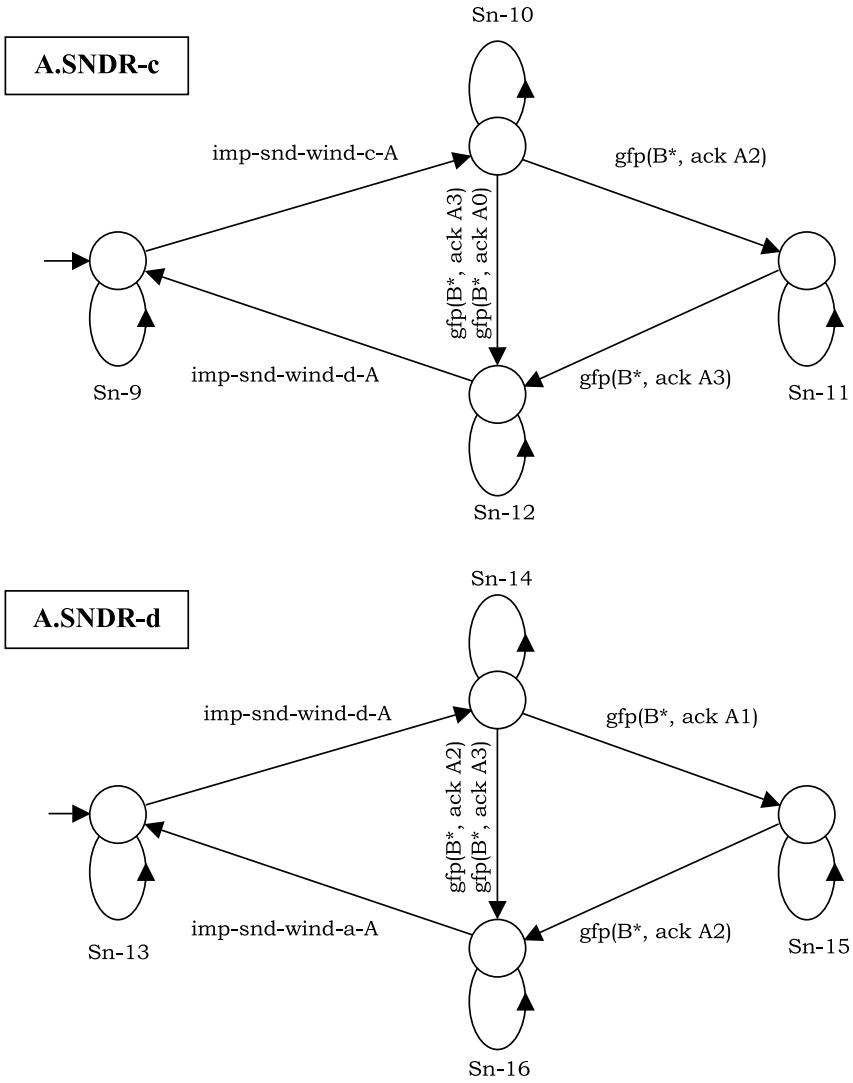


Figure 4.14: Automata *A.SNDR-c* and *A.SNDR-d*

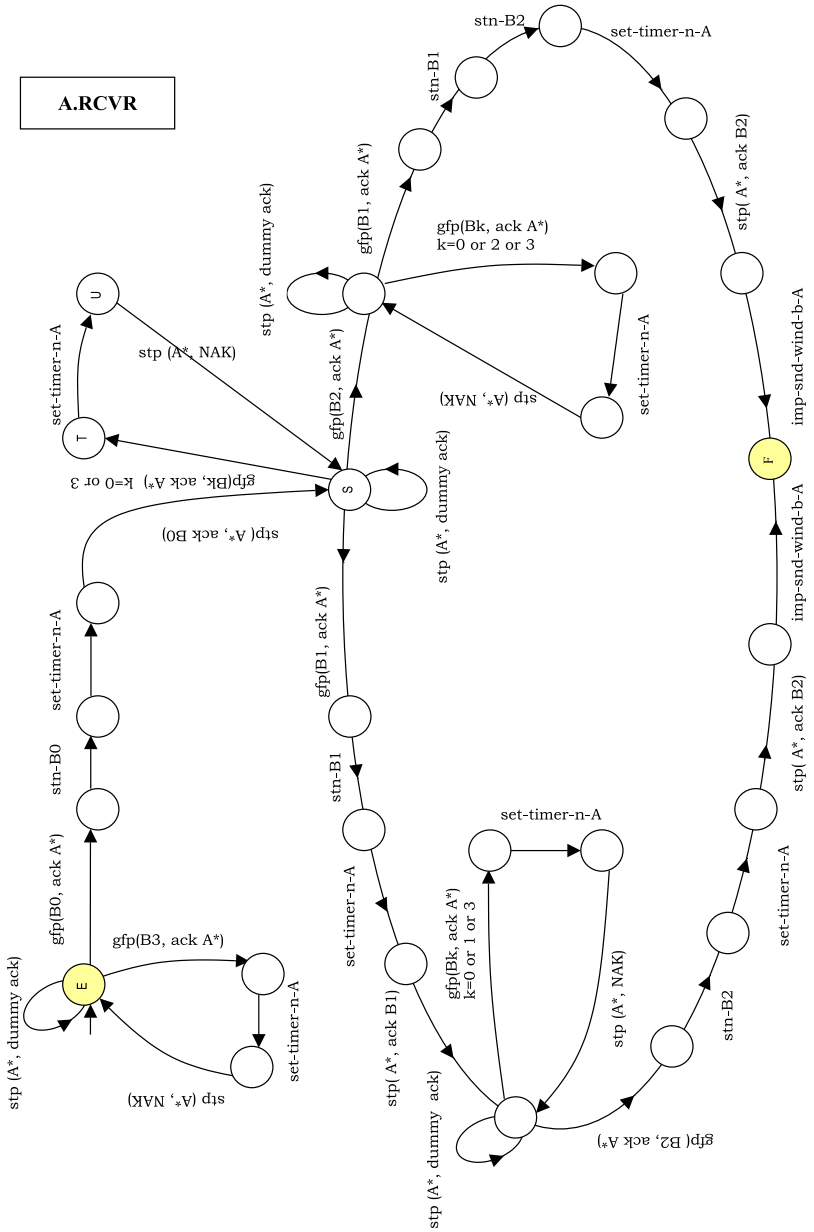


Figure 4.15: Automaton *A.RCVR*, page 1 of 12

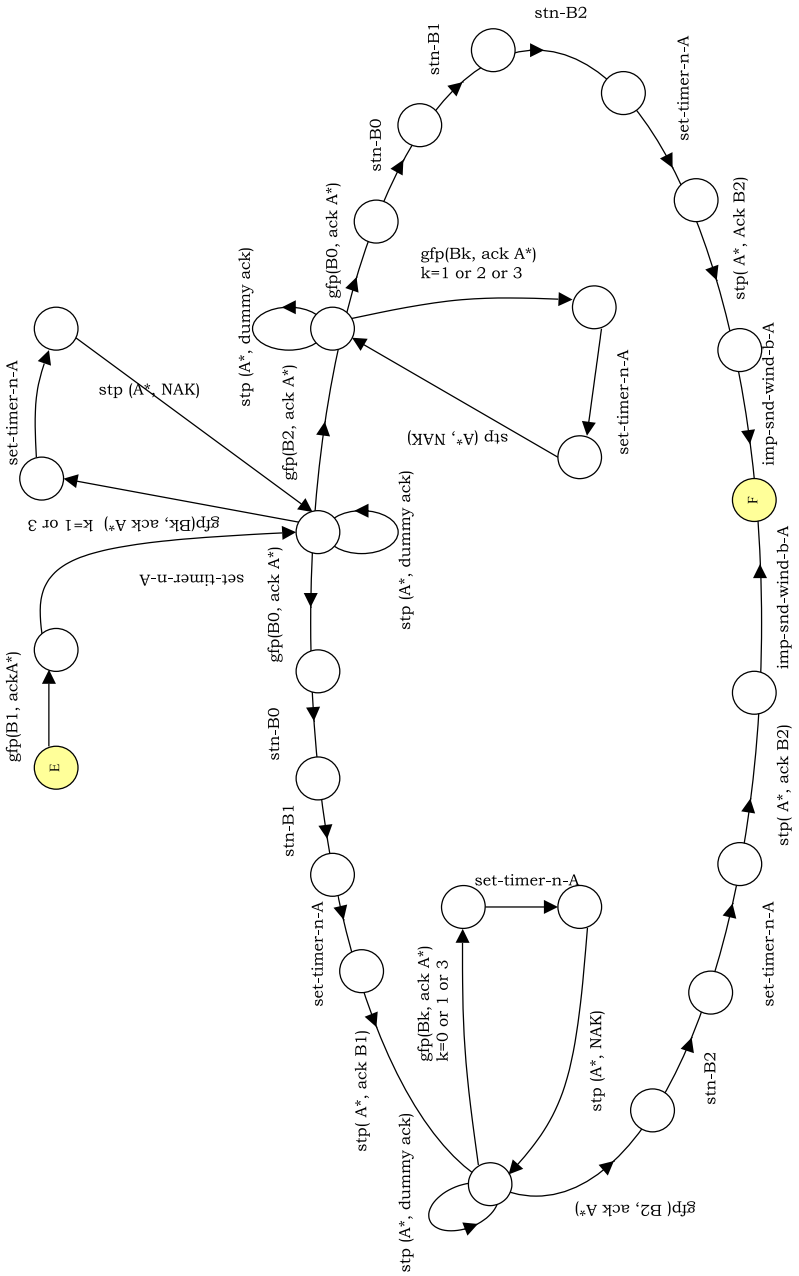


Figure 4.16: Automaton *A.RCVR*, page 2 of 12

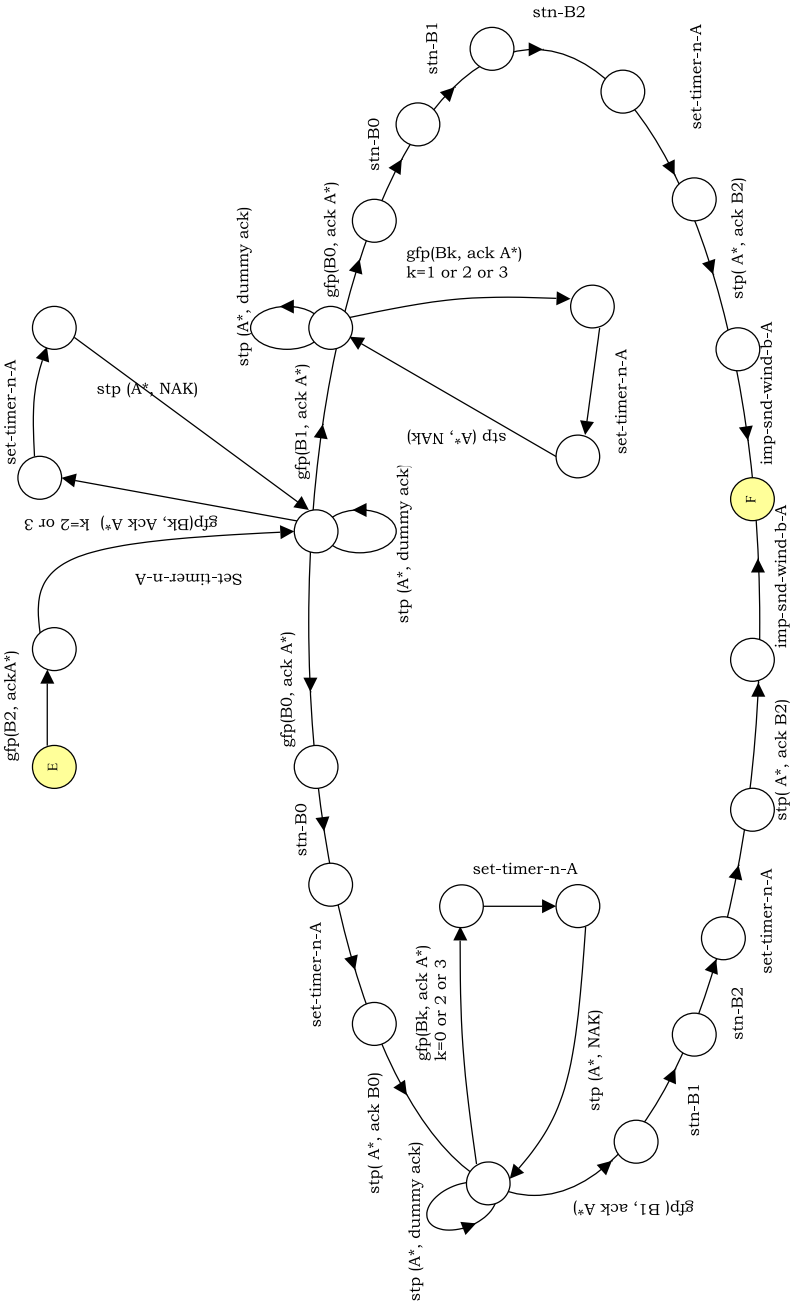


Figure 4.17: Automaton *A.RCVR*, page 3 of 12

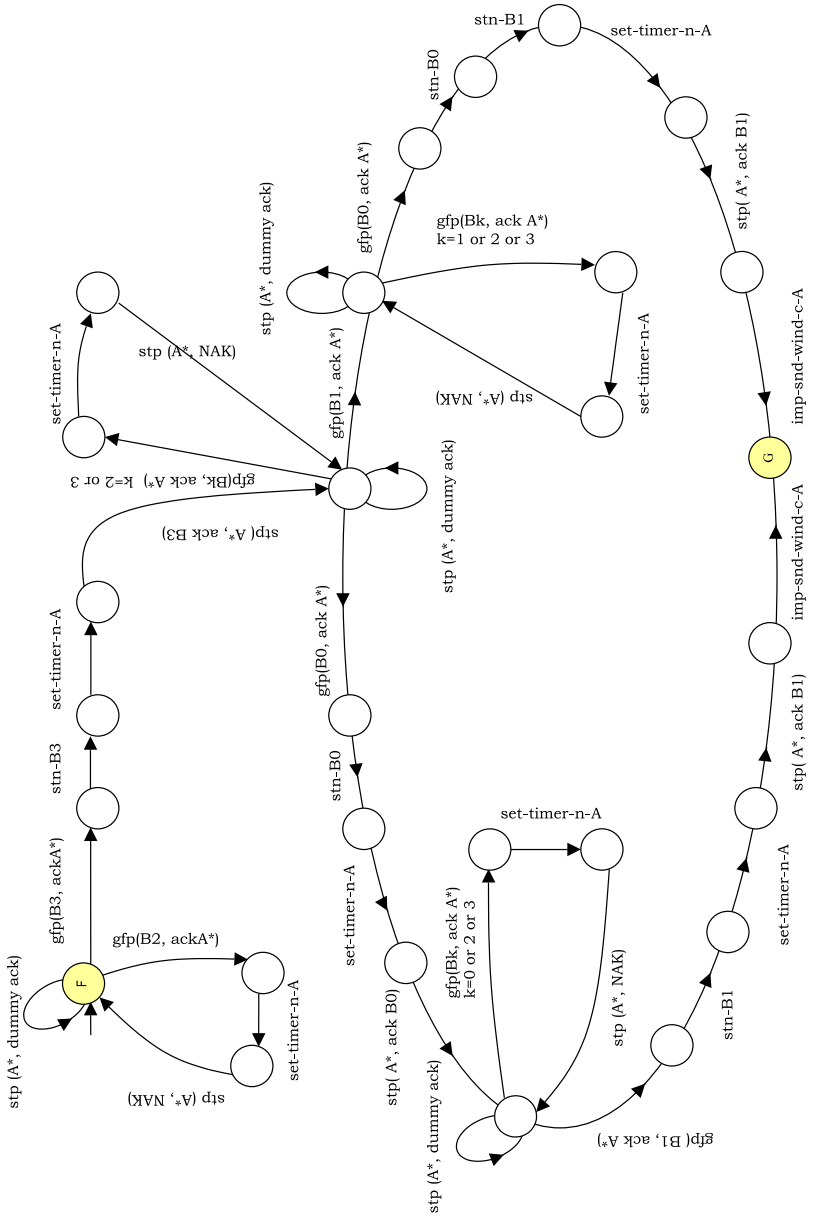


Figure 4.18: Automaton *A.RCVR*, page 4 of 12

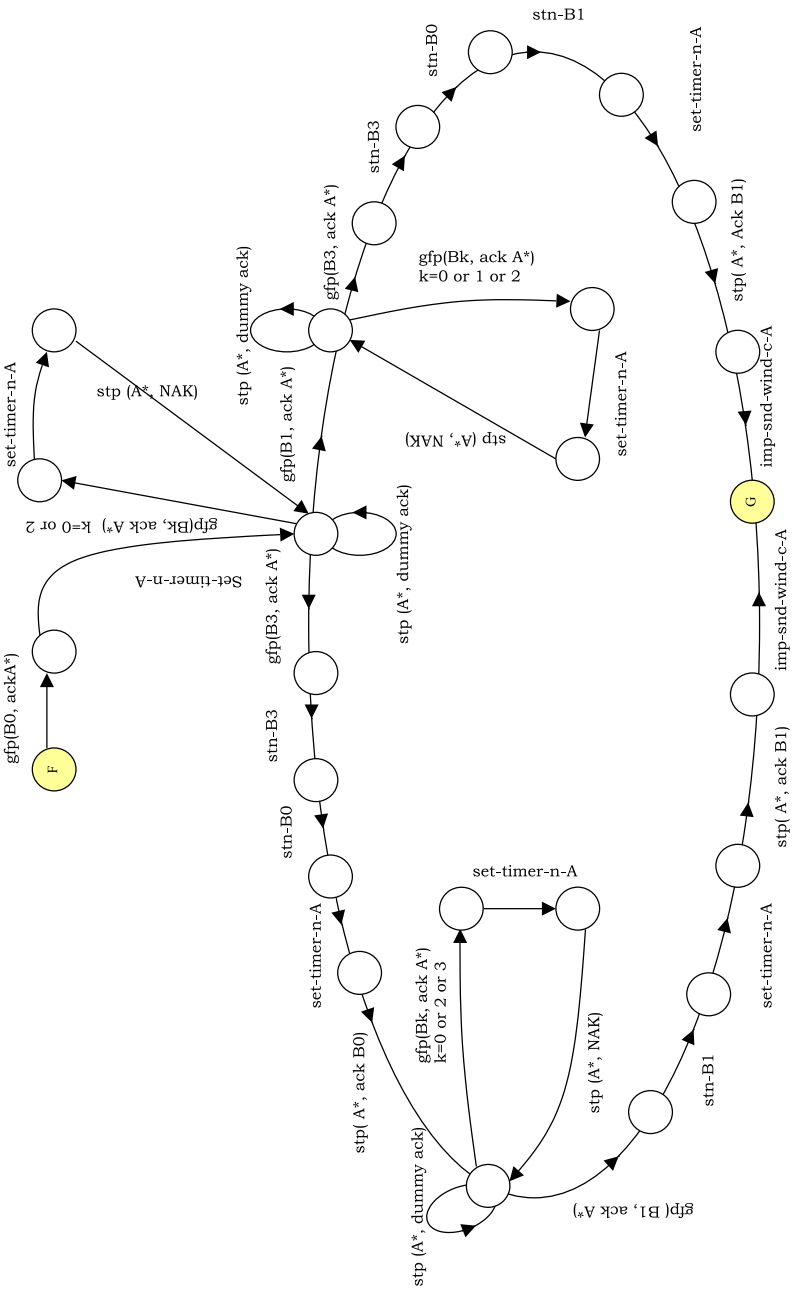


Figure 4.19: Automaton *A.RCVR*, page 5 of 12

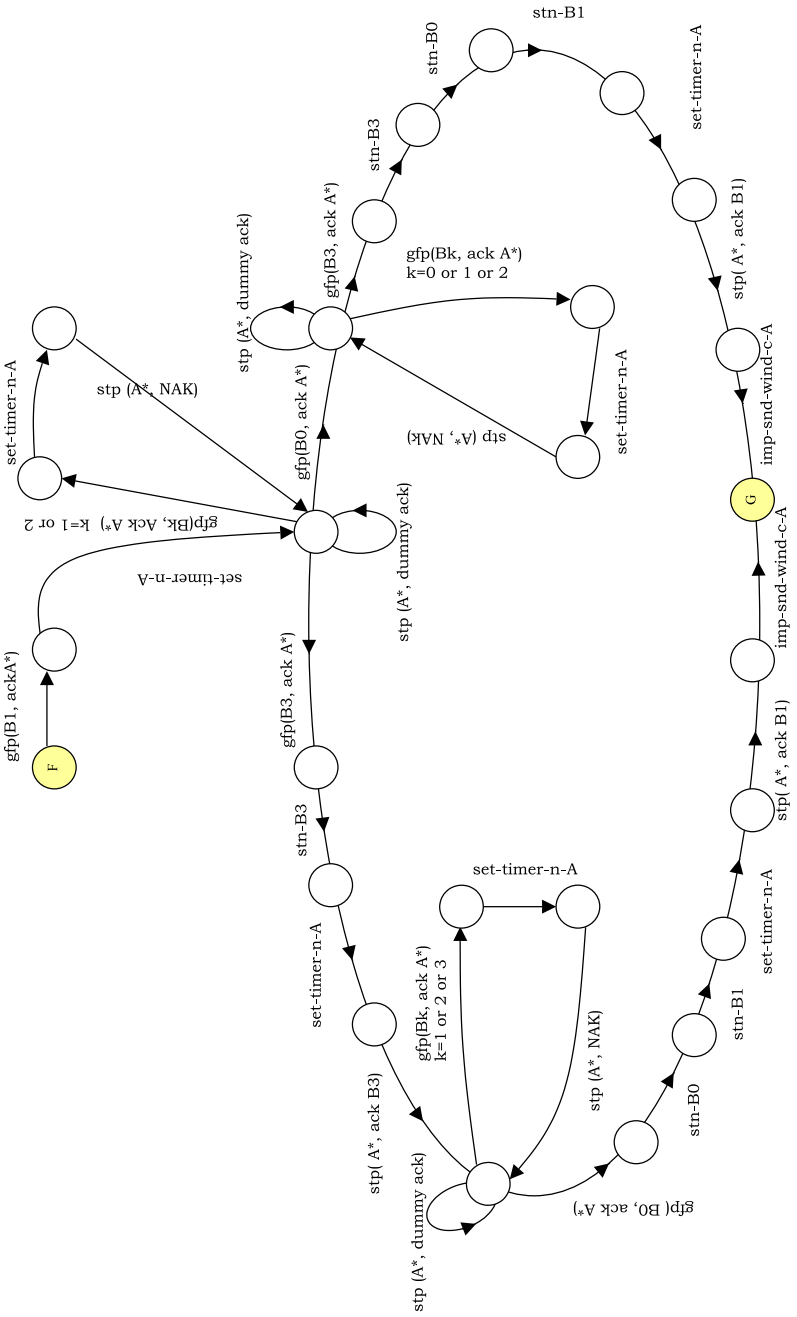


Figure 4.20: Automaton *A.RCVR*, page 6 of 12

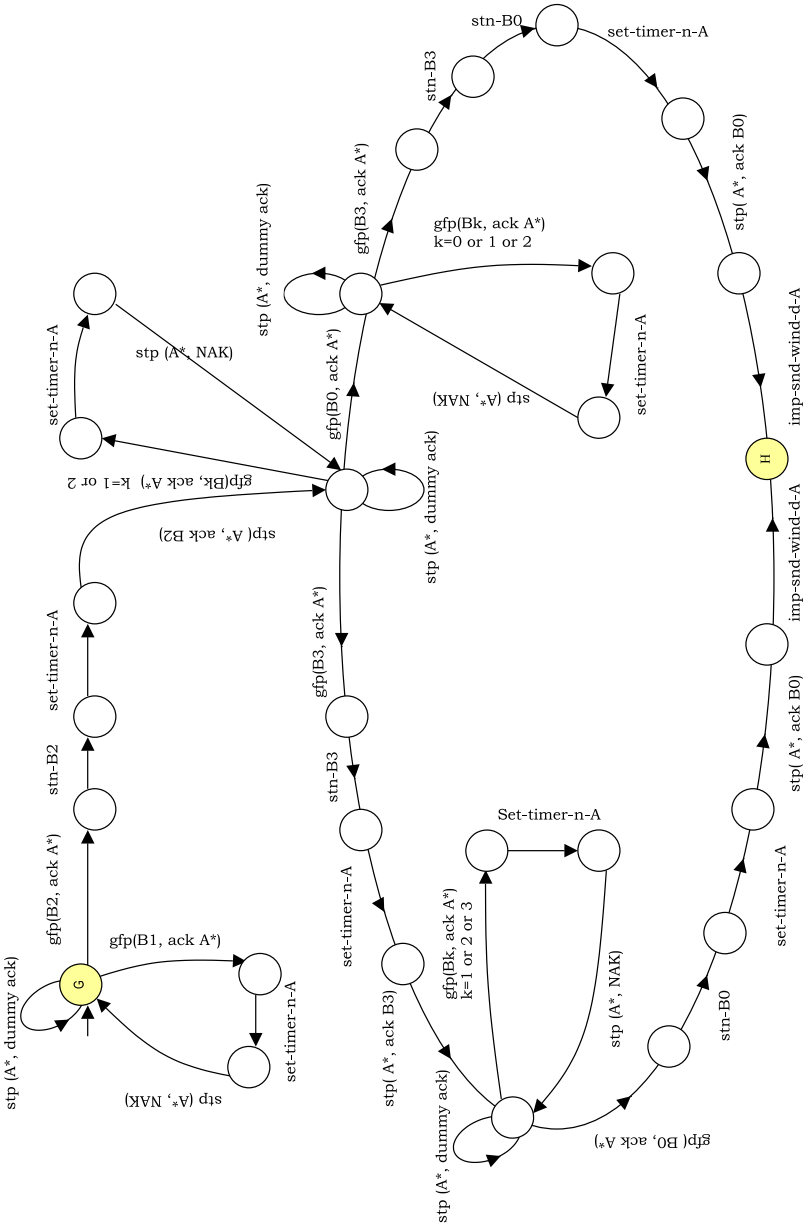


Figure 4.21: Automaton $A.RCVR$, page 7 of 12

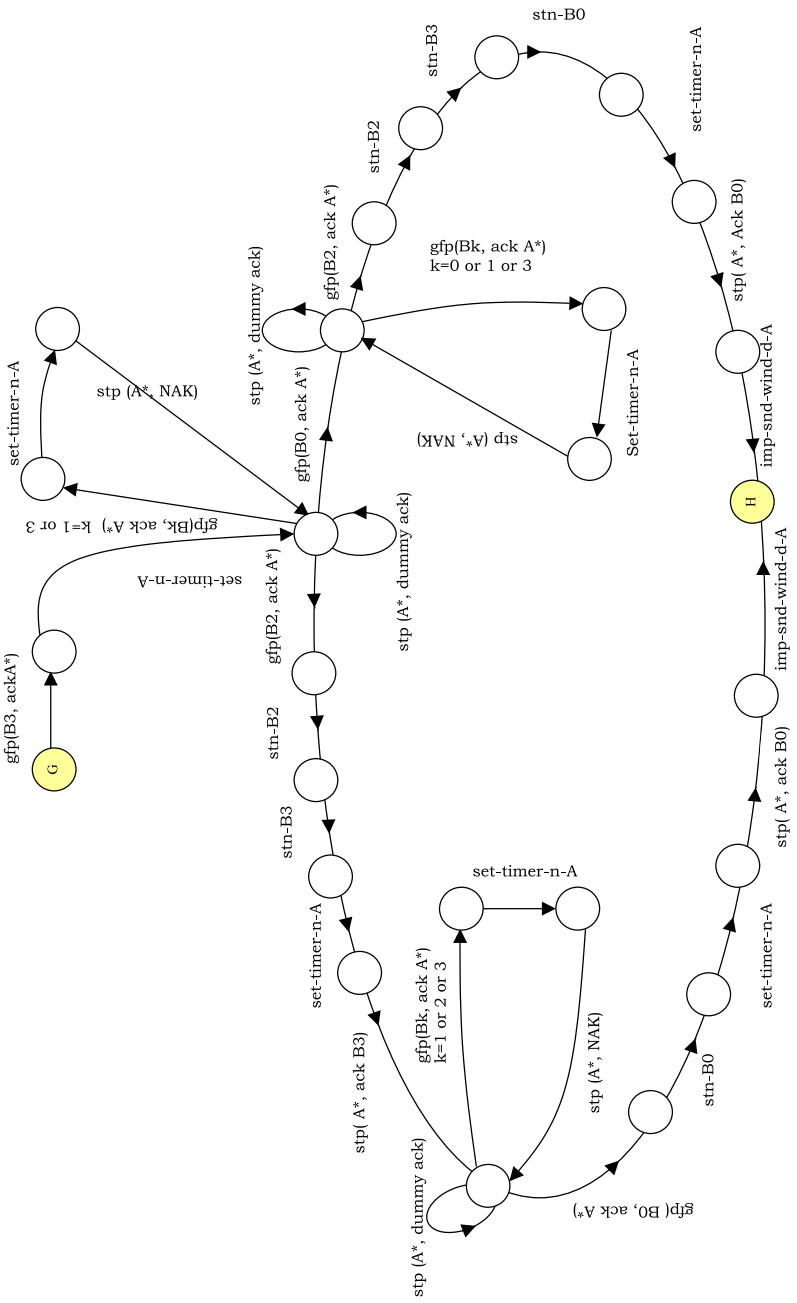


Figure 4.22: Automaton $A.RCVR$, page 8 of 12

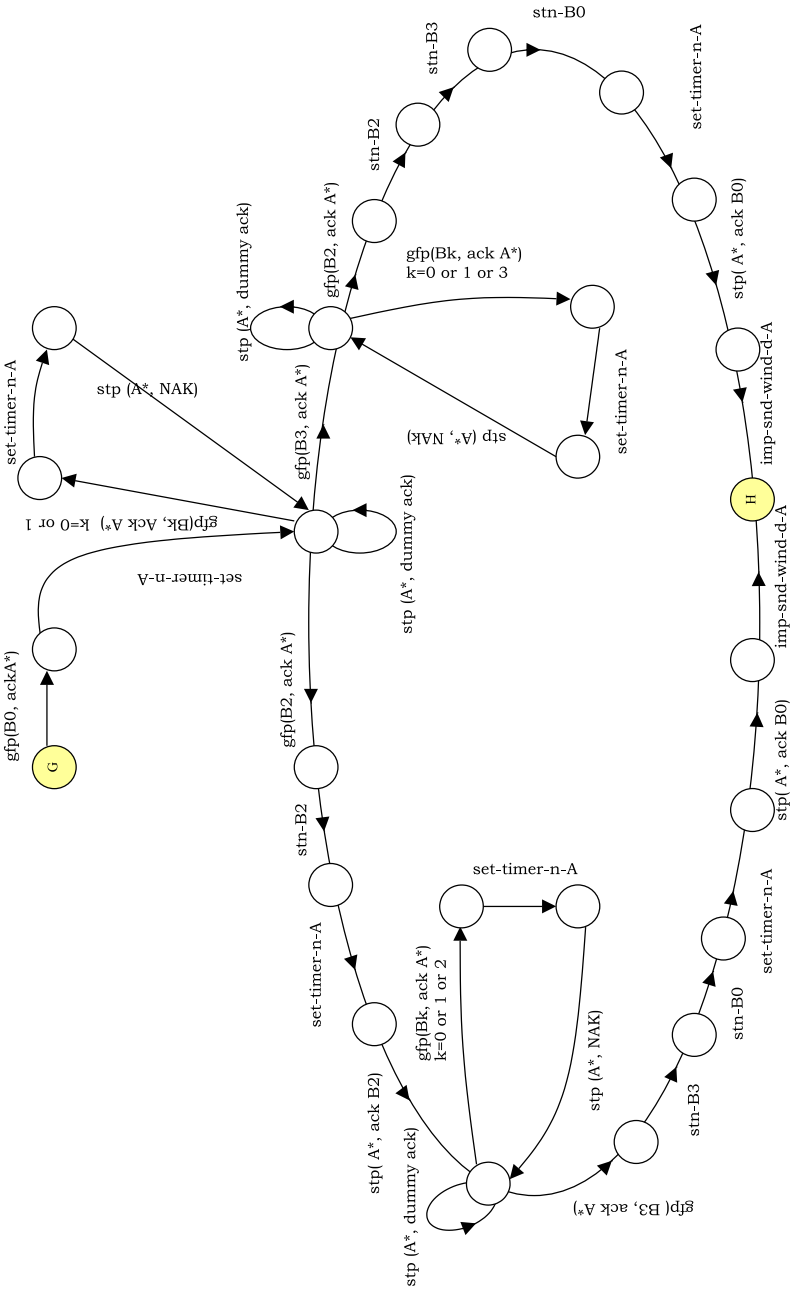


Figure 4.23: Automaton $A.RCVR$, page 9 of 12

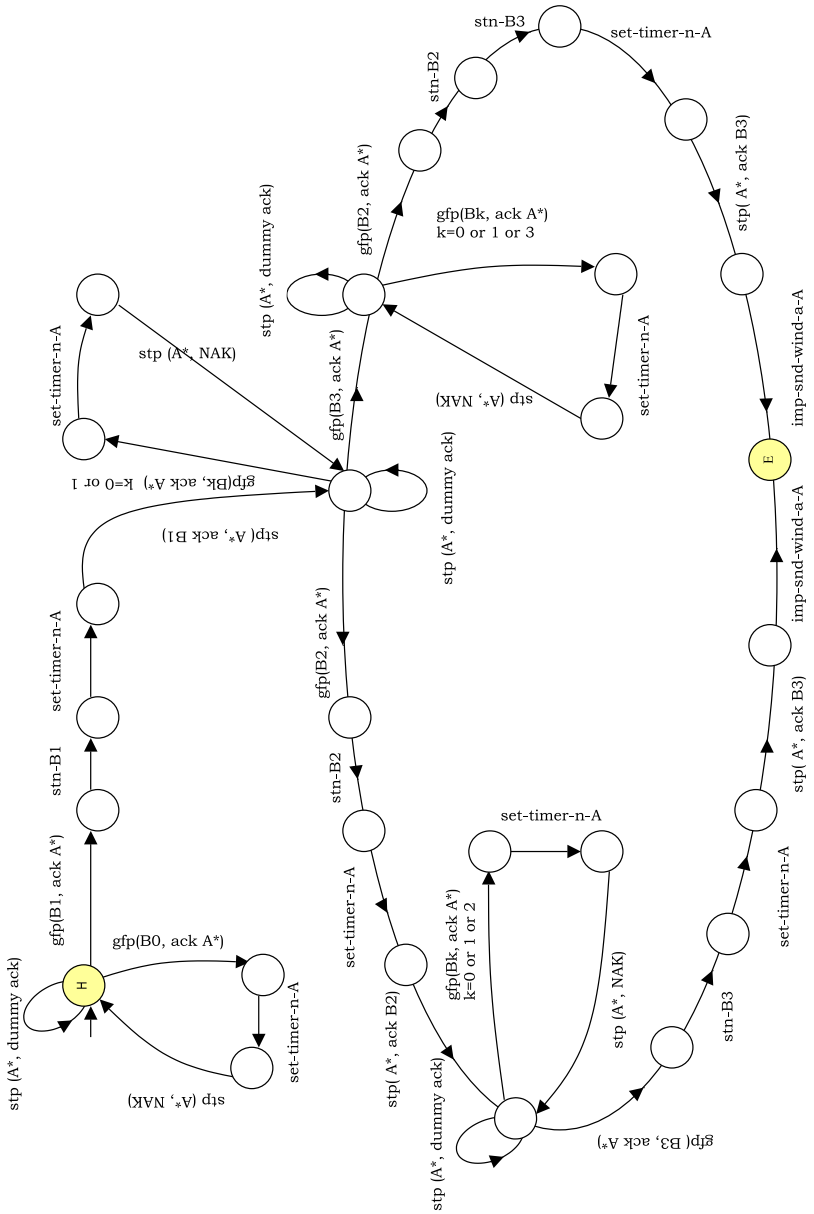


Figure 4.24: Automaton *A.RCVR*, page 10 of 12

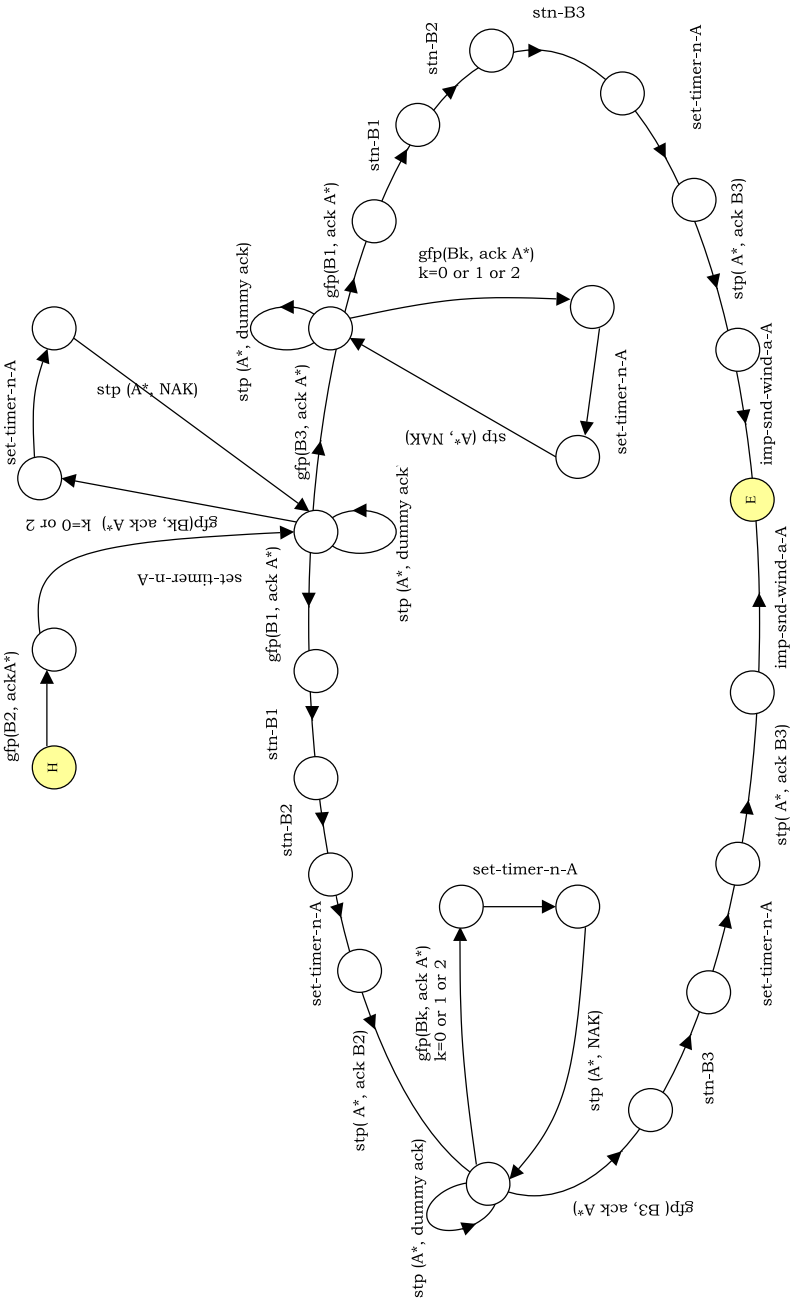


Figure 4.25: Automaton *A.RCVR*, page 11 of 12

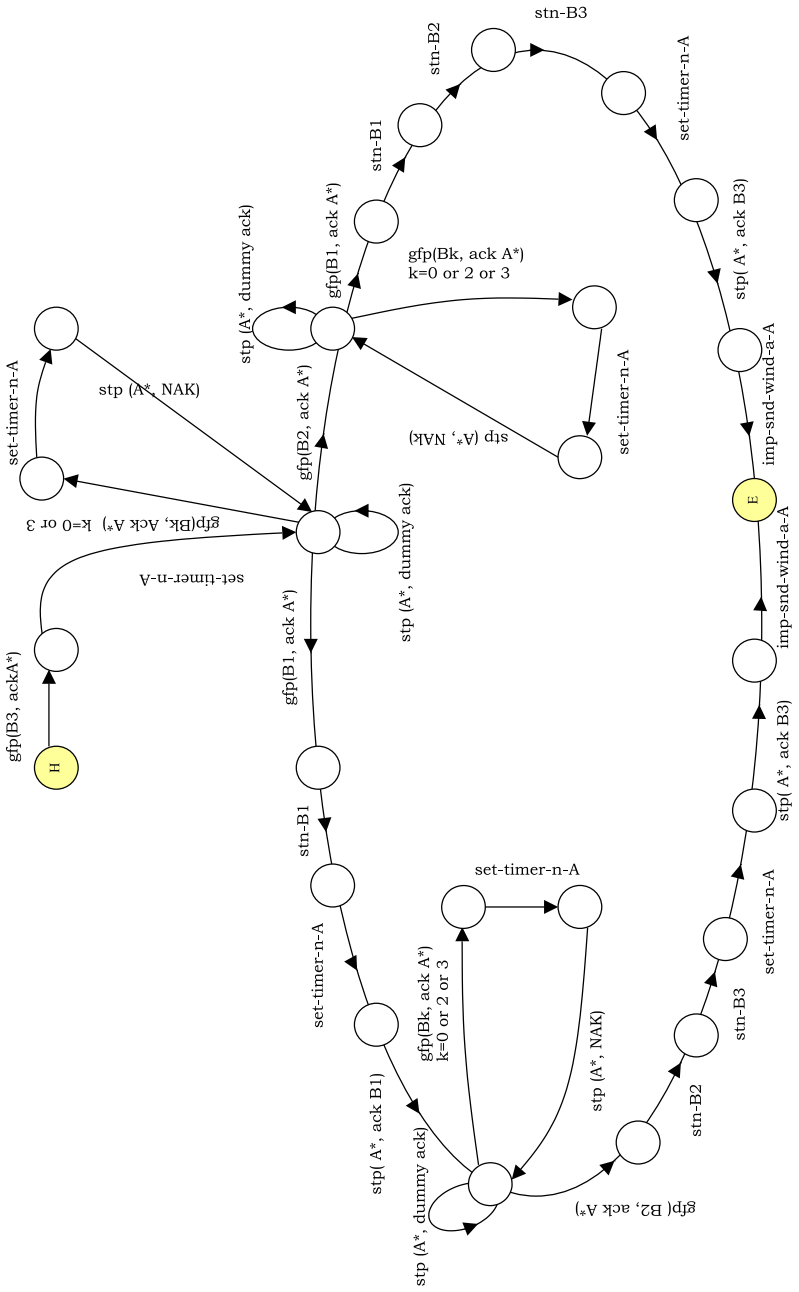


Figure 4.26: Automaton *A.RCVR*, page 12 of 12

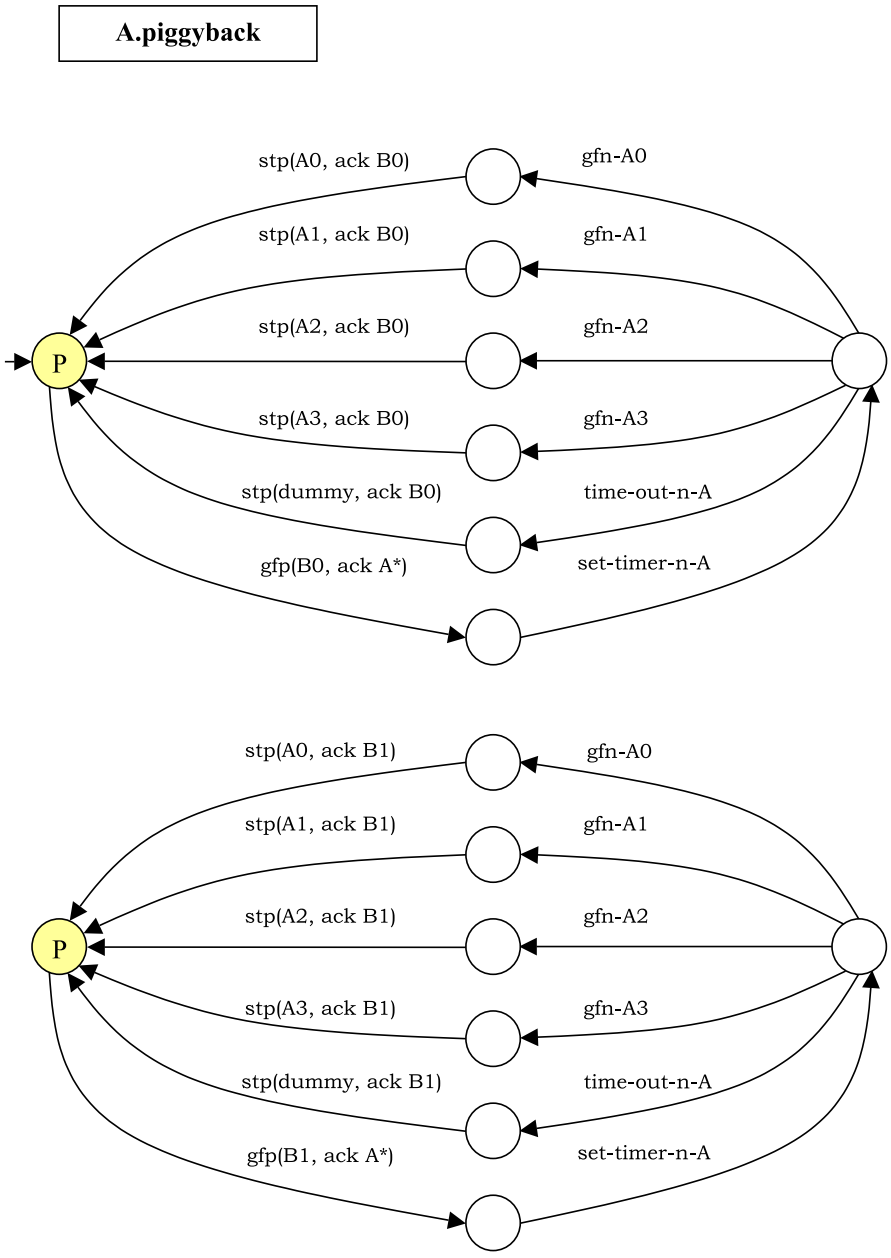


Figure 4.27: Automaton *A.piggyback*, page 1 of 3

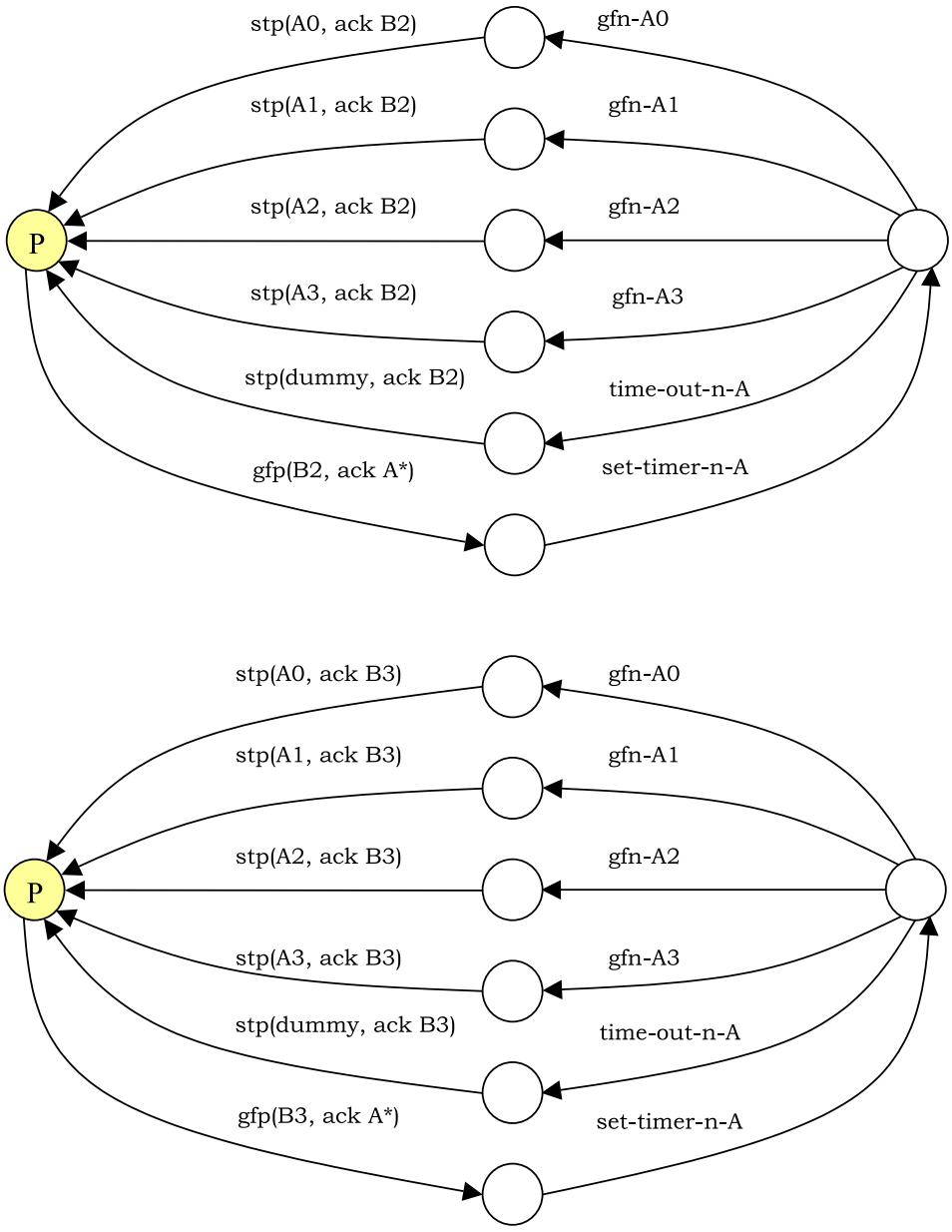


Figure 4.28: Automaton *A.piggyback*, page 2 of 3

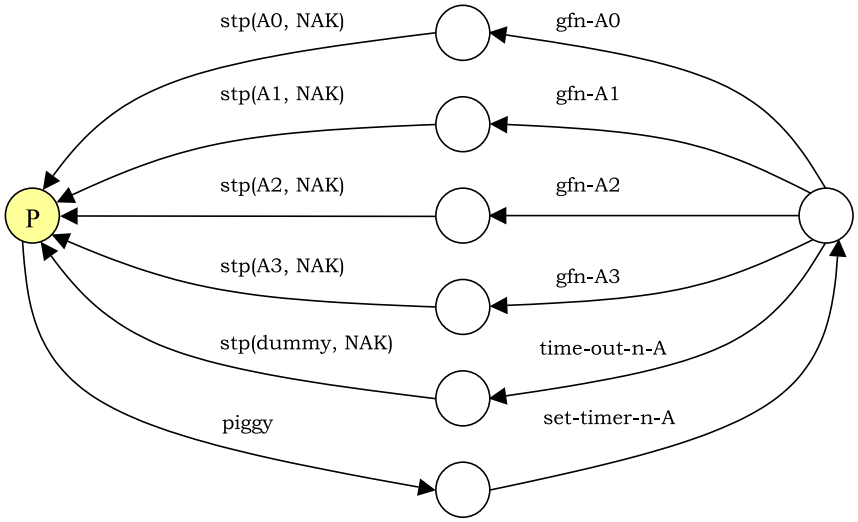


Figure 4.29: Automaton *A.piggyback*, page 3 of 3

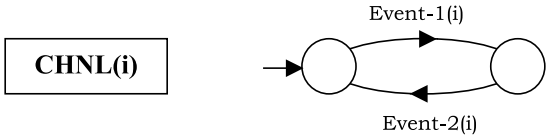


Figure 4.30: Automaton $CHNL(i), i = 1, 2, ..., 56$

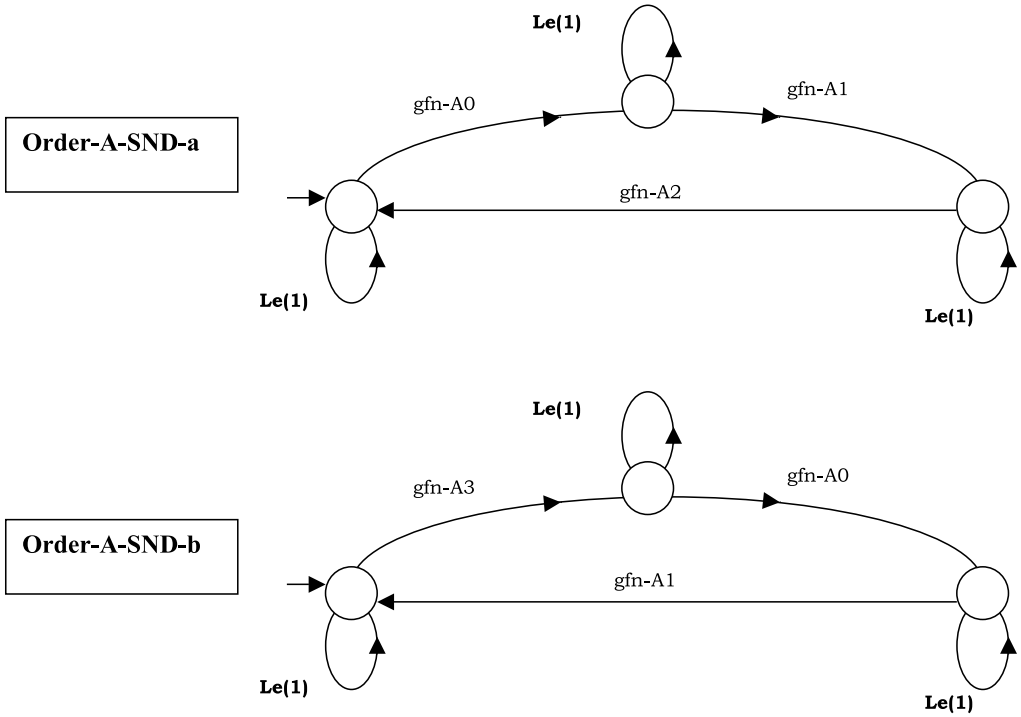


Figure 4.31: Automata *Order-A-SND-a* and *Order-A-SND-b*

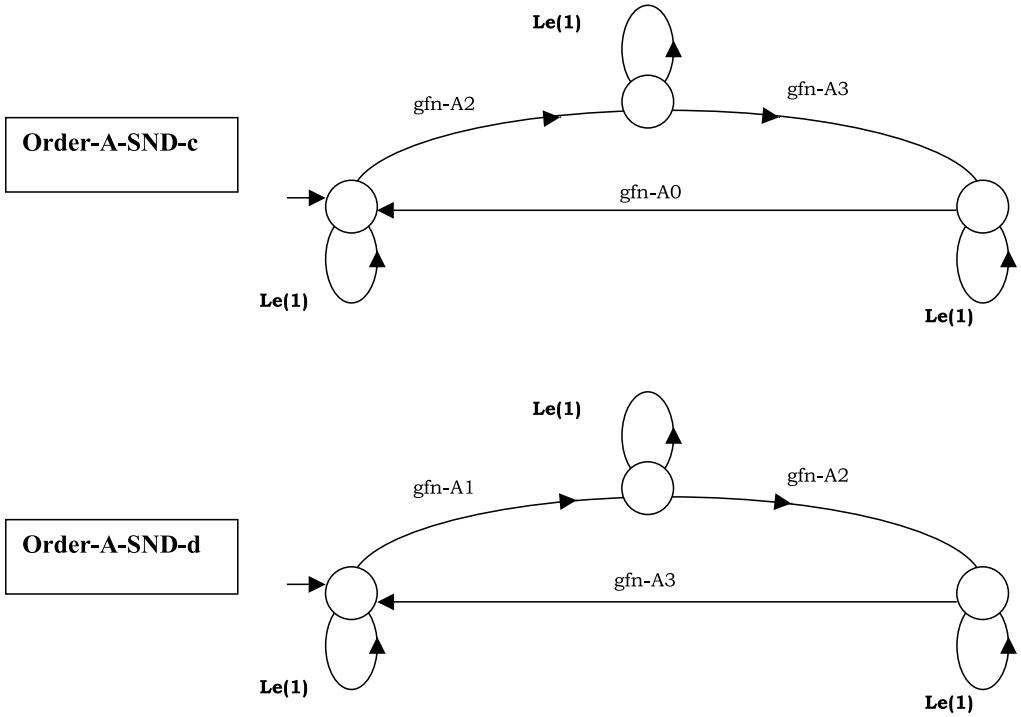


Figure 4.32: Automata *Order-A-SND-c* and *Order-A-SND-d*

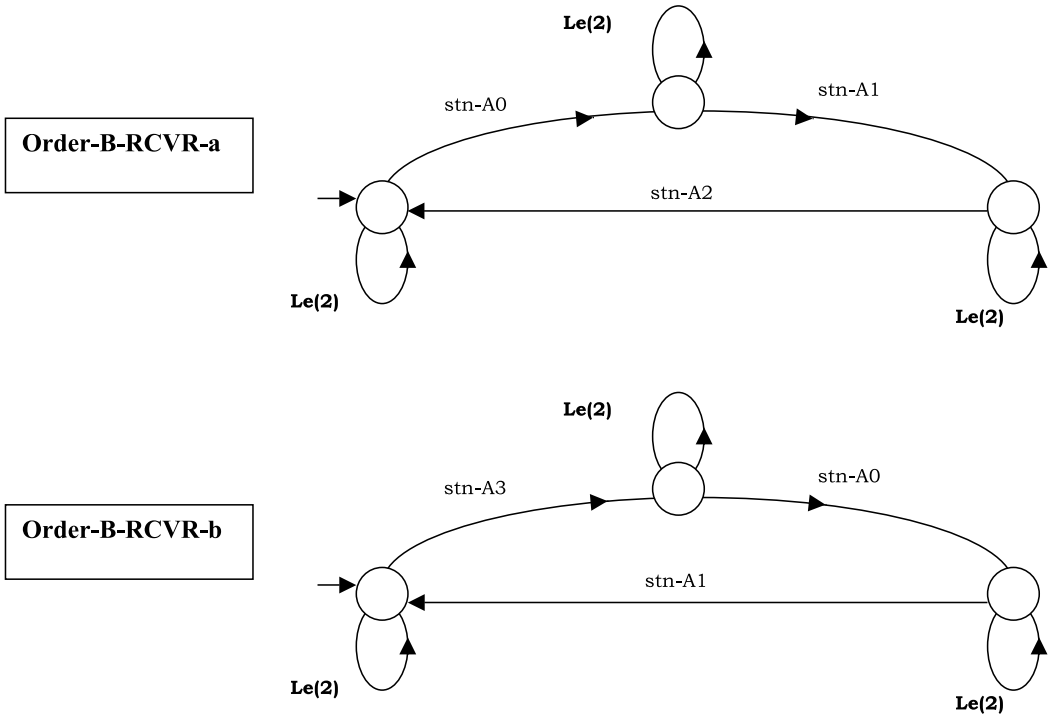


Figure 4.33: Automata *Order-B-RCVR-a* and *Order-B-RCVR-b*

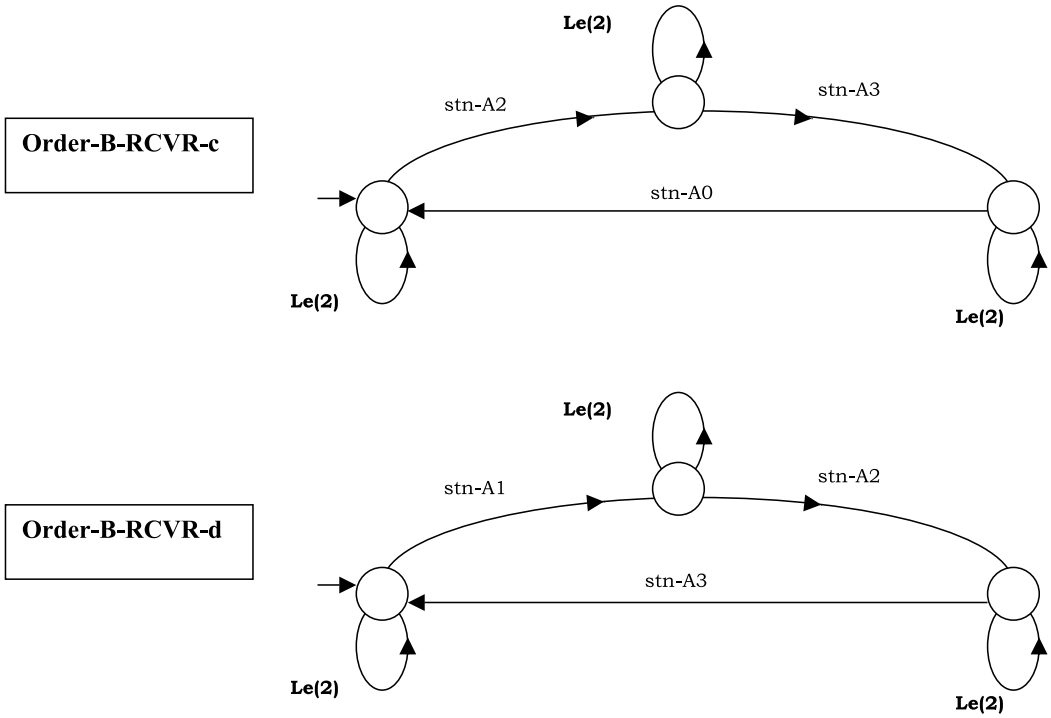


Figure 4.34: Automata *Order-B-RCVR-c* and *Order-B-RCVR-d*

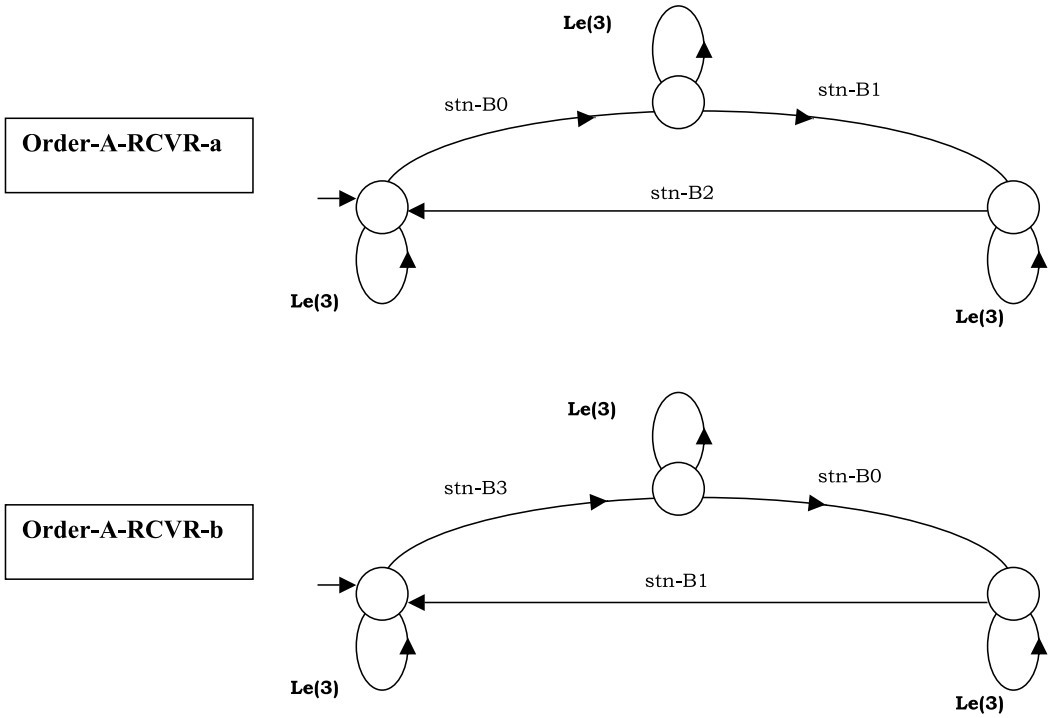


Figure 4.35: Automata *Order-A-RCVR-a* and *Order-A-RCVR-b*

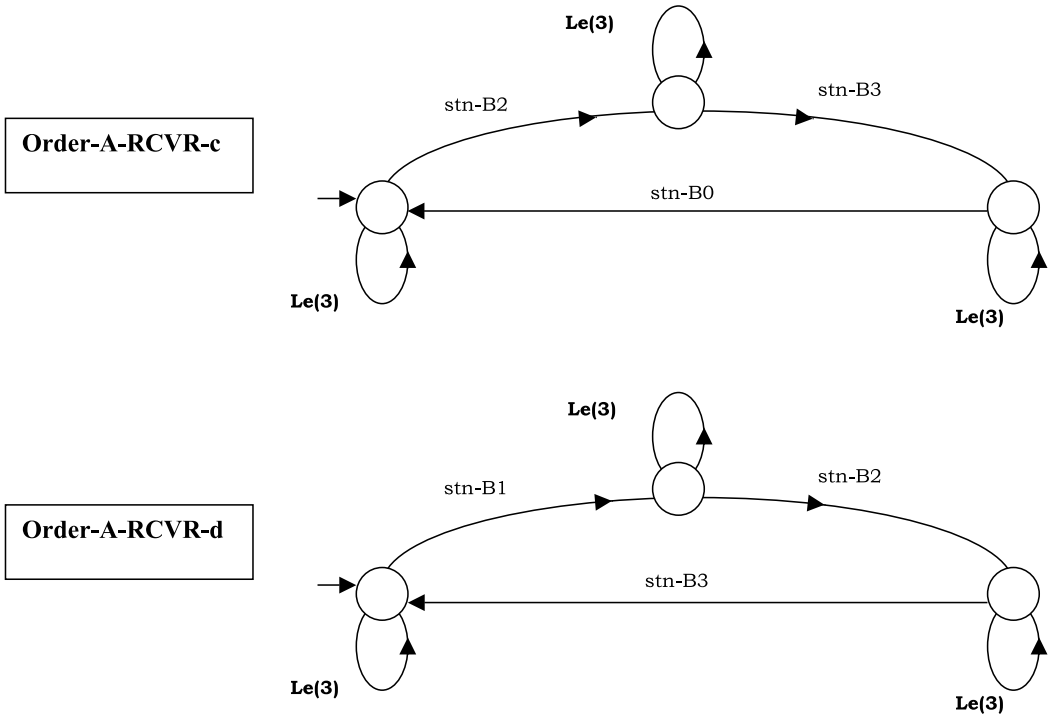


Figure 4.36: Automata *Order-A-RCVR-c* and *Order-A-RCVR-d*

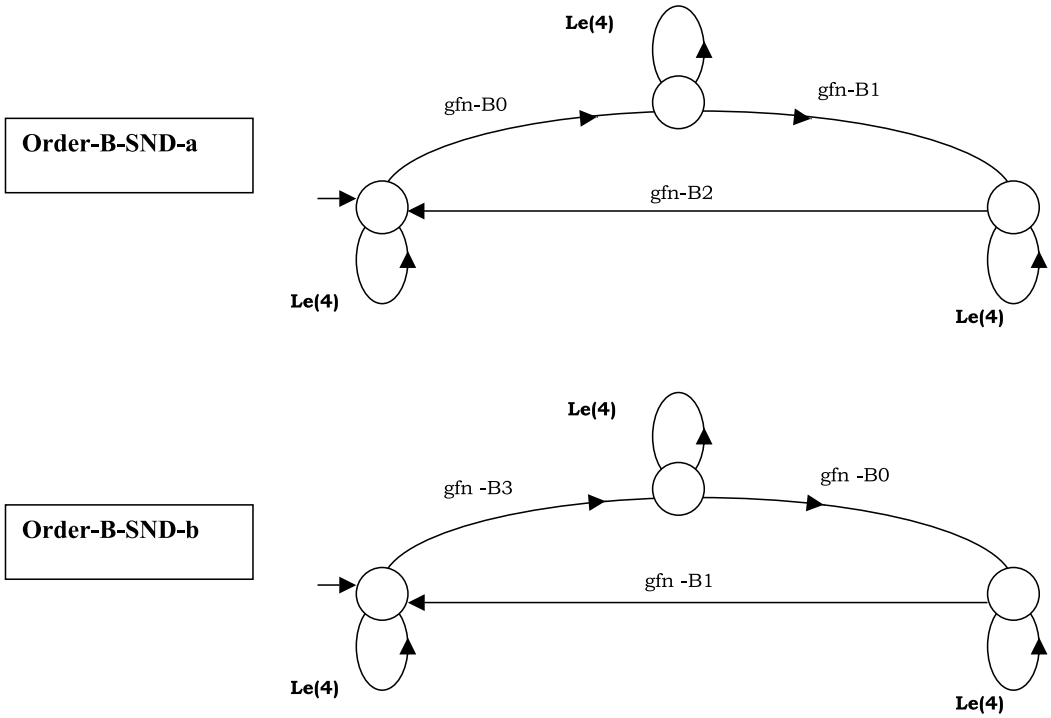


Figure 4.37: Automata *Order-B-SND-a* and *Order-B-SND-b*

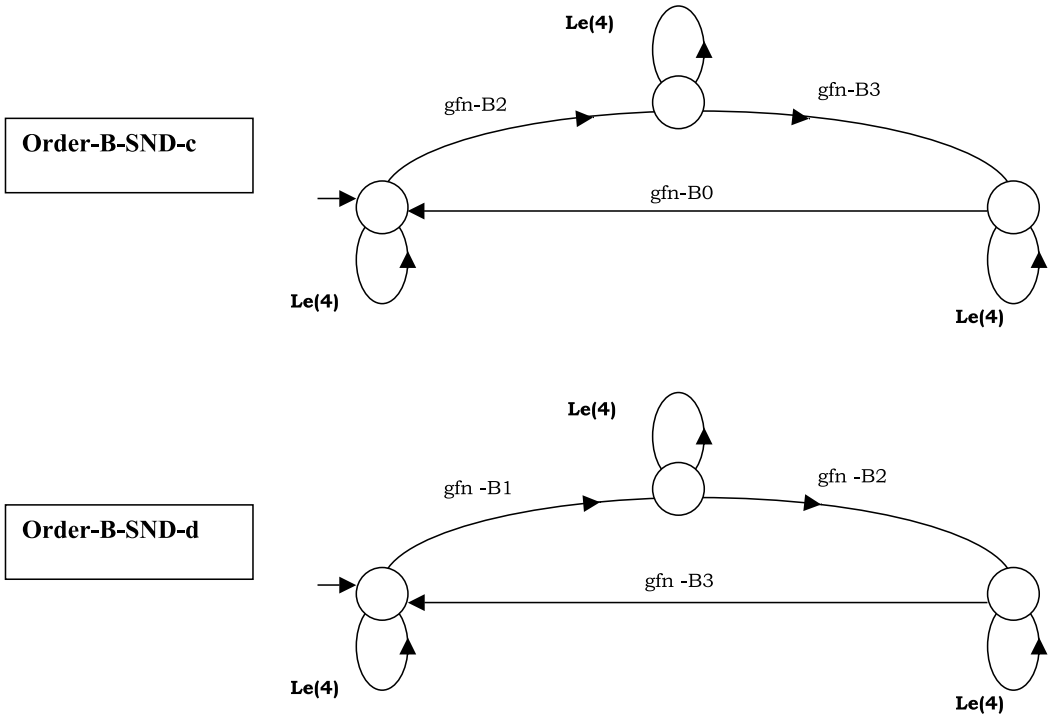


Figure 4.38: Automata *Order-B-SND-c* and *Order-B-SND-d*

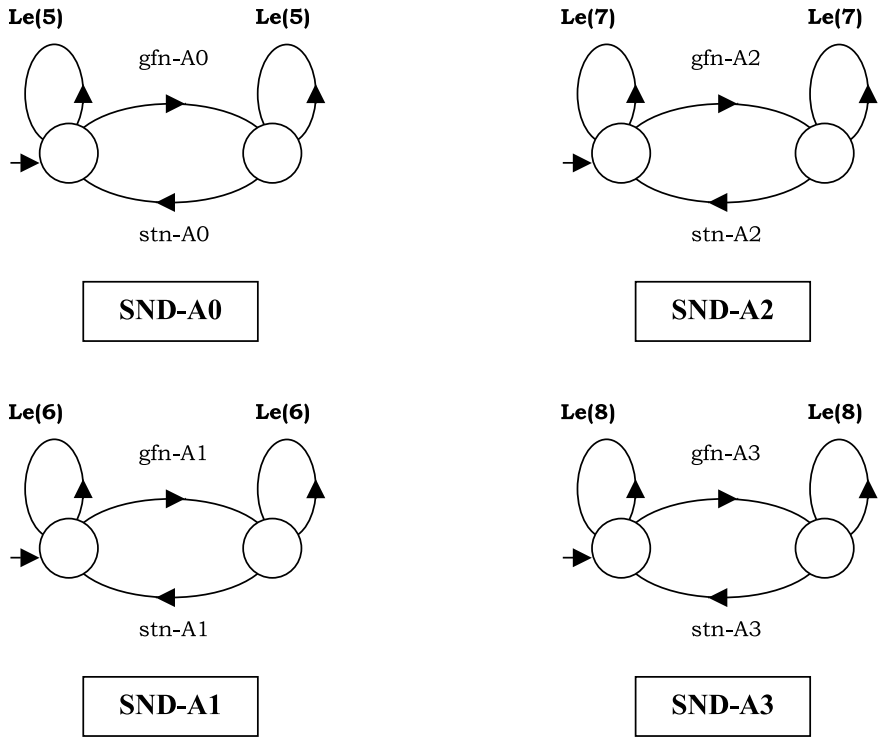


Figure 4.39: Automata *SND-A0*, *SND-A1*, *SND-A2*, and *SND-A3*

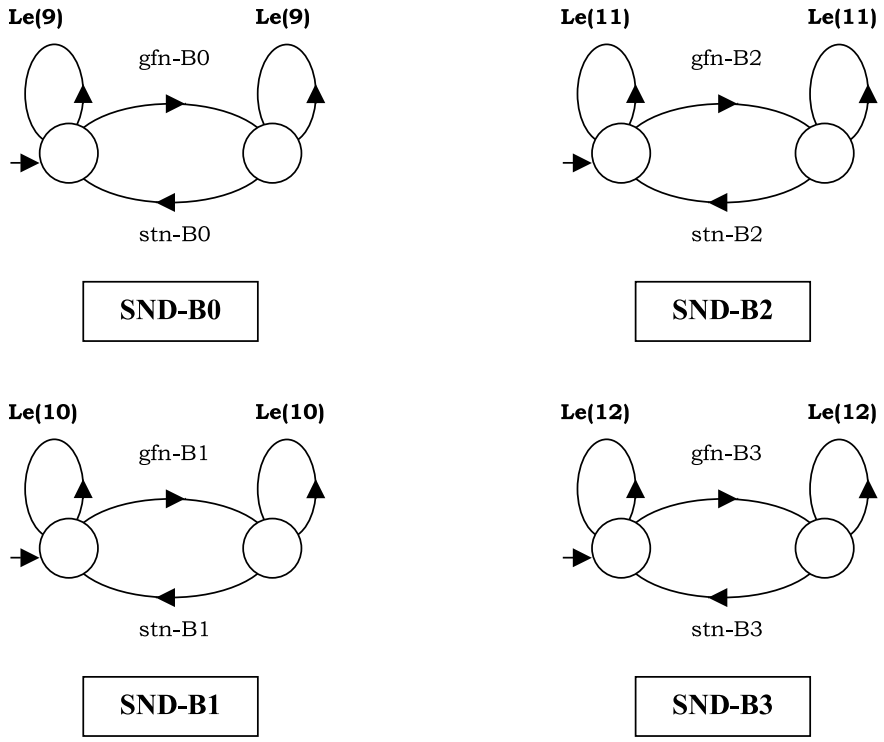


Figure 4.40: Automata *SND-B0*, *SND-B1*, *SND-B2*, and *SND-B3*

Problematic Sequences

Referring to the event table, we can recognize whether an event is in $\Sigma_{c,A}$, $\Sigma_{c,B}$, $\Sigma_{o,A}$, and/or $\Sigma_{o,B}$. Also, we have demonstrated how automata corresponding to the plant and the legal language can be found. For the protocol SW1, we can find two sequences of events, i.e., t and t', and an event σ that violate co-observability of the legal language with respect to the plant.

Event σ and the sequence t are given as follows:

σ = *stn-A0*

t = *imp-snd-wind-a-A, gfn-A0, stp(A0, dummy ack), set-timer-p-A0, gfp(A0, dummy), stn-A0, set-timer-n-B, imp-snd-wind-a-B, gfn-B0, stp(B0, ackA0), set-timer-p-B0, gfp(B0, !ackA0), stn-B0, set-timer-n-A, gfn-A1, stp(A1, ackB0),*

set-timer-p-A1, gfp(A1, ackB0), stn-A1, set-timer-n-B, gfn-B1, stp(B1, ackA1), set-timer-p-B1, gfp(B1, ackA1), stn-B1, set-timer-n-A, gfn-A2, stp(A2, ackB1), set-timer-p-A2, gfp(A2, ackB1), stn-A2, set-timer-n-B, gfn-B2, stp(B2, ackA2), set-timer-p-B2, imp-RCVR-wind-b-B, gfp(B2, ackA2), stn-B2, set-timer-n-A, time-out-n-A, stp(A2, ackB2), imp-RCVR-wind-b-A, imp-SND-b-A, gfp(A2, ackB2), set-timer-n-B, imp-snd-wind-b-B, gfn-B3, stp(B3, NAK), set-timer-p-B3, gfp(B3, NAK), stn-B3, set-timer-n-A, gfn-A3, stp(A3, ackB3), set-timer-p-A3, time-out-p-A0, stp(A0, ackB2), gfp(A3, ackB3), stn-A3, set-timer-n-B, gfp(A0, ackB2)

The sequence $t\sigma$ leads to the failure of the protocol. This is illustrated as follows. Suppose the window of the sender of agent A is advanced to **a**. The sender of agent A gets A0 from the network layer, sends A0 to the physical layer, and starts the timer of A0. Agent B receives A0 from the physical layer, sends A0 to the network layer, and sends the acknowledgment of A0 to the physical layer. However, the acknowledgment of A0 is damaged in the channel and agent A gets the damaged acknowledgment of A0. Agent A gets A1 and A2 from the network layer consecutively, sends them to the physical layer, and starts the timer of A1 and A2. Agent B receives A1 and A2, sends them to the network layer, and then sends an acknowledgment of A1 and A2 to the physical layer. Agent A receives the acknowledgment of A1 and A2. Agent B advances the window of its receiver to **b**. Agent A advances the window of its sender to **b**, gets A3 from the network layer, sends it to the physical layer, and starts the timer of A3. Agent B receives A3 and sends it to the network layer. The timer of A0 elapses and agent A resends A0 to the physical layer. Agent B receives A0 from the physical layer and sends A0 to the network layer, but the same data frame A0 has already been sent to the network layer. In fact, the receiver passes A0 to the network layer because it has assumed that A0 is a new frame to be sent to

the network layer, which is not true. This is why the protocol fails.

On the other hand, had a correct data transmission between the two agents happened instead, the following sequence t' would have occurred:

t' = *imp-snd-wind-a-A, gfn-A0, stp(A0, dummy ack), set-timer-p-A0, gfp(A0, dummy), stn-A0, set-timer-n-B, imp-snd-wind-a-B, gfn-B0, stp(B0, ackA0), set-timer-p-B0, gfp(B0, !ackA0), stn-B0, set-timer-n-A, gfn-A1, stp(A1, ackB0), set-timer-p-A1, gfp(A1, ackB0), stn-A1, set-timer-n-B, gfn-B1, stp(B1, ackA1), set-timer-p-B1, gfp(B1, ackA1), stn-B1, set-timer-n-A, gfn-A2, stp(A2, ackB1), set-timer-p-A2, gfp(A2, ackB1), stn-A2, set-timer-n-B, gfn-B2, stp(B2, ackA2), set-timer-p-B2, imp-RCVR-wind-b-B, gfp(B2, ackA2), stn-B2, set-timer-n-A, time-out-n-A, stp(A2, ackB2), imp-RCVR-wind-b-A, imp-snd-wind-b-A, gfp(A2, ackB2), set-timer-n-B, imp-snd-wind-b-B, gfn-B3, stp(B3, NAK), set-timer-p-B3, gfp(B3, NAK), stn-B3, set-timer-n-A, gfn-A3, stp(A3, ackB3), set-timer-p-A3, gfn-A0, stp(A0, dummy ack), gfp(A3, ackB3), stn-A3, set-timer-n-B, gfp(A0, dummy ack)*

The scenario captured by the sequence $t'\sigma$ is described as follows. Assume the window of the sender of agent A is advanced to **a**. The sender of agent A gets A0 from the network layer, sends A0 to the physical layer, and starts the timer of A0. Agent B receives A0 from the physical layer, sends A0 to the network layer, and sends the acknowledgment of A0 to the physical layer. Agent A receives acknowledgment of A0 from the physical layer. Then agent A gets A1 and A2 from the network layer consecutively, sends them to the physical layer, and starts the timer of A1 and A2. Agent B receives A1 and A2, sends them to the network layer, and then sends an acknowledgment of A1 and A2 to the physical layer. Agent A receives the acknowledgment of A1 and A2. Agent B advances the window of its receiver to **b**. Agent A advances the window of its sender to **b**, gets A3 from the network layer, sends it to the physical layer, and

starts the timer of A3. Agent B receives A3 and sends it to the network layer. Agent A gets new data A0 from the network layer and sends A0 to the physical layer. Agent B receives A0 from the physical layer and sends it to the network layer. The sequence $t'\sigma$ would not have led to a protocol failure.

We observe that $P_B(t) = P_B(t')$, both $t', t \in$ *Legal-language*, $t'\sigma \in$ *Legal-language*, $t\sigma \notin$ *Legal-language*, and sequences $t\sigma, t'\sigma \in$ *Plant*, where $\sigma \in \Sigma_{c,B} \backslash \Sigma_{c,A}$. Therefore, the pair $(t\sigma, t'\sigma)$ violates co-observability of *Legal-language* with respect to *Plant*. In other words, the sequence $t\sigma$ is a problematic sequence of the protocol, since it looks like the sequence $t'\sigma$, and $t'\sigma$ is a successful run of the protocol SW1.

4.2.2 SW2 Protocol

The SW2 protocol is a restriction of the SW1 protocol in the following sense:

- Agent A contains a sender and not a receiver.
- Agent B contains a receiver and not a sender.

Because of the previous two items, neither of the agents needs piggybacking to send a data frame or acknowledgment to the other agent. Actually, the existence of a dummy data frame or dummy acknowledgment is unnecessary, but we consider them in the protocol SW2.

As with the protocol SW1, we can construct the automaton *Plant* and the automaton *Legal-language* for the protocol SW2. We consider the sets $\Sigma_{c,A}$, $\Sigma_{c,B}$, $\Sigma_{o,A}$, and $\Sigma_{o,B}$ of this protocol and extract a pair of sequences of events, namely $(t\sigma, t'\sigma)$, that violate co-observability of *Legal-language* with respect to *Plant*. Consider the following event σ and sequences t, t':

$\sigma =$ *stn-A0*

t = imp-snd-wind-a-A, gfn-A0, stp A0, set-timer-p-A0, gfp A0, stn-A0, stp ackA0, gfp!ackA0, gfn-A1, stp A1, set-timer-p-A1, gfp A1, stn-A1, stp ackA1, gfp !ackA1, gfn-A2, stp A2, set-timer-p-A2, gfp A2, stn-A2, stp ackA2, gfp NAK, imp-RCVR-wind-b-B, time-out-p-A0, stp A0, set-timer-p-A0, gfp A0, stp ack dummy, gfp ackA2, imp-snd-wind–b-A, gfn-A3, stp A3, set-timer-p-A3, gfp A3, stn-A3

t′ = imp-snd-wind-a-A, gfn-A0, stp A0, set-timer-p-A0, gfp A0, stn-A0, stp ackA0, gfp ackA0, gfn-A1, stp A1, set-timer-p-A1, gfp A1, stn-A1, stp ackA1, gfp ackA1, gfn-A2, stp A2, set-timer-p-A2, gfp A2, stn-A2, stp ackA2, imp-RCVR-wind-b-B, gfp ackA2, imp-snd-wind-b-A, gfn-A3, stp A3, set-timer-p-A3, gfn-A0, stp A0, set-timer-p-A0, gfp A0, stp ack-dummy, gfp ackA3, gfp A3, stn-A3

The pair $(t\sigma, t'\sigma)$ cause failure of co-observability of *Legal-laguage* with respect to *Plant*, because $P_A(t) = P_A(t')$, where sequences $t', t \in$ *Legal-language*, $t'\sigma \in$ *Legal-language*, $t\sigma \notin$ *Legal-language*, sequences $t\sigma, t'\sigma \in$ *Plant*, and $\sigma \in \Sigma_{c,A} \backslash \Sigma_{c,B}$. Therefore the sequence $t\sigma$ is a problematic sequence of the protocol SW2.

4.2.3 SW3 Protocol

The protocol SW3 mainly works like the protocol SW1, but differs in the following respects:

- The sequence numbers of the frames are in $\{0, 1\}$.
- The window size of the senders and the receivers is one.

The two automata representing the plant and the legal language of the protocol SW3 can be constructed in the same way as those for the protocol SW1. We can find two sequences of events, $t\sigma$ and $t'\sigma$, that violate co-observability, as follows:

σ = *stn-B1*

t= gfn A0, stp(A0, ackB1), set-timer-p-A0, gfn B0, stp(B0, ackA1), set-timer-p-B0, gfp(A0, ackB1), stn A0, gfp(B0, ackA1), stn B0, set-timer-n-B, gfn B0, stp(B0, ackA0), set-timer-n-A, gfn A0, stp(A0, ackB0), set-timer-p-B0, gfp(B0, ackA0), stn B0, gfp(A0, ackB0), stn A0, set-timer-n-B, gfn B1, stp(B1, ackA0), set-timer-p-B1, gfp(B1, ack A0), stn B1, set-timer-n-A, gfn A1, stp(A1, ackB0), set-timer-p-A1, time-out-p-B1, stp(B1, dummy ack), gfp(B1, ackA1)

t' = *gfn-A0, stp(A0, ackB1), set-timer-p-A0, gfn-B0, stp(B0, ackA1), set-timer-p-B0, gfp(A0, ackB1), stn-A0, gfp(B0, ack A1), stn-B0, set-timer-n-B, gfn-B0, stp(B0, ack A0), set-timer-n-A, gfn-A0, stp(A0, ackB0), set-timer-p-B0, gfp(B0, ackA0), stn-B0, gfp(A0, ack B0), stn-A0, set-timer-n-B, gfn-B1, stp(B1, ackA0), gfp(B1, ackA0), stn-B1, set-timer-n-A, gfn-A1, stp(A1, ackB0), set-timer-p-A1, gfp(A1, ackB0), stn-A1, set-timer-n-B, gfn-B1, stp(B1, ackA1), set-timer-p-B1, gfp(B1, ackA1)*

The pair $(t\sigma, t'\sigma)$ violates co-observability of *Legal-laguage* with respect to *Plant*, because $P_A(t) = P_A(t')$, where sequences $t', t \in$ *Legal-language*, $t'\sigma \in$ *Legal-language*, $t\sigma \notin$ *Legal-language*, sequences $t\sigma, t'\sigma \in$ *Plant*, and $\sigma \in \Sigma_{c,A} \backslash \Sigma_{c,B}$. Therefore the sequence $t\sigma$ is a problematic sequence of the protocol SW3.

4.3 Verification of Telecommunication Protocol Examples

Theoretically, the program implemented using the some-paths method can be applied to verify the protocols mentioned in Section 4.2. However, since they are large examples, they need either an unreasonably long time or a large amount of memory due to the computational complexity of the program. Both time and memory are limited resources and we cannot expect to be able to run the

program for inputs of any size with limited memory or during a reasonable time.

For instance, our available memory is insufficient for the protocol in [4]. Because our memory is less than 8 GB, there is not enough room to execute the first step of the some-paths method, which is the construction of the automaton M. A depth-first search algorithm is used in the implementation of the algorithm introduced in [21] to construct automaton M. The program generates a long chain of consecutive states while constructing the automaton M of the above example. The number of consecutive states of the chain can be close to e^3g, where $e = 616$ and $g = 412$ are the number of states of the automaton E and the automaton G, respectively. If we consider the length of the chain and the amount of memory required to contain the information for each state of the chain, the available memory is not sufficient to provide results for this example using our program.

On the other hand, computing time, as opposed to memory, becomes the limiting factor in the data transmission protocol of [24]. To model the data transmission protocol of [24], we use the same automata *SNDR* and *RCVR* as those in [24] and use the automaton in Figure 4.41 to represent the *CHNL*. In this case, automaton E and G each contain 150 states. The program successfully constructs the automaton M for the example. The automata E and G are minimized, and then used to construct automaton M. Also, the automaton M is trimmed. Thus, the number of its states is decreased as much as possible. In the example, automaton M contains 54154 states. According to the second step of the some-paths method, the program would need to find the corresponding regular expression for the automaton M. The computational complexity of the second step is $O(m^3)$, where m is the number of states of the automaton M. We let the program run on a SUN ULTRA 1 machine for four days and terminated it because it could keep running for several months, due to the size of the problem

and the computational complexity of the method.

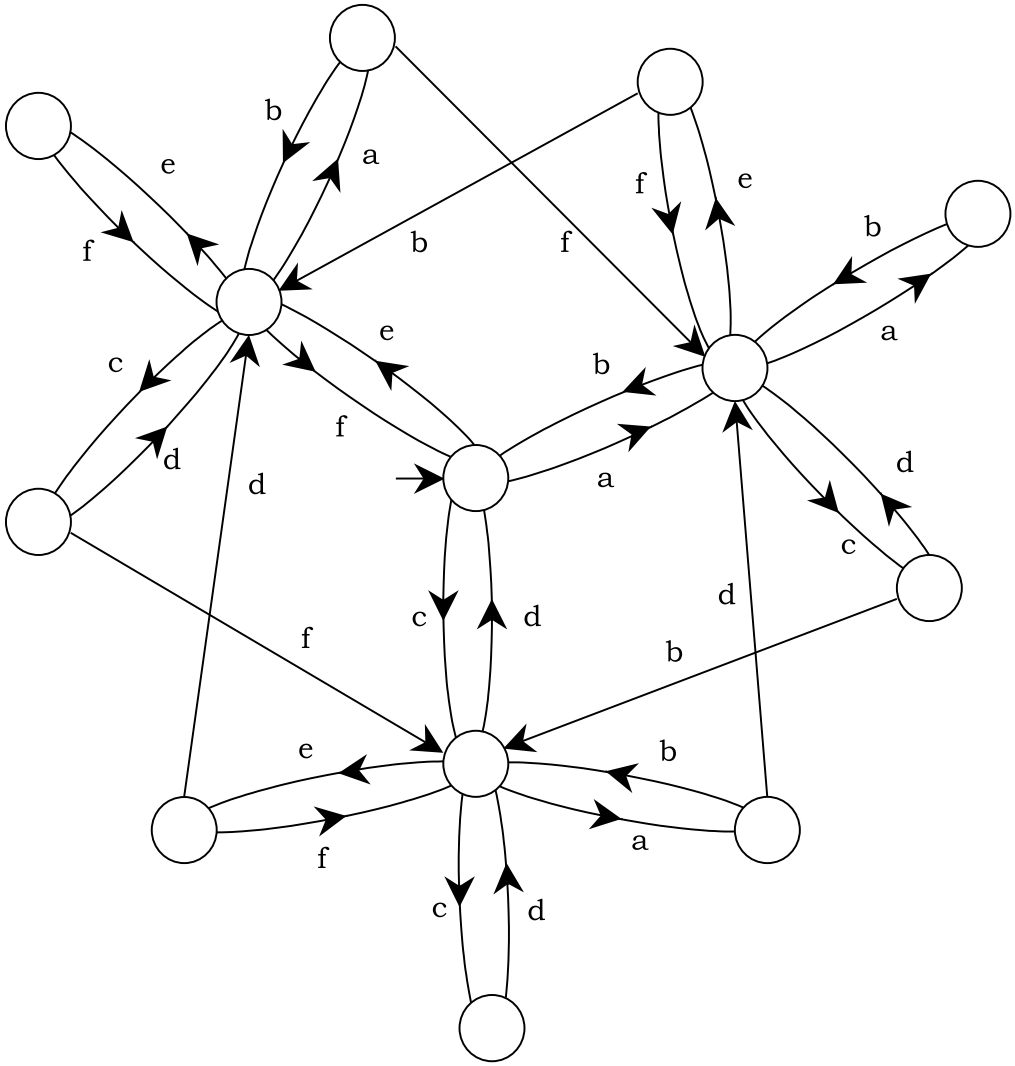


Figure 4.41: Automaton *CHNL* of [24], where $a \in \{send_0\}$, $b \in \{rcv_0, lose, cksumerr\}$, $c \in \{send_1\}$, $d \in \{rcv_1, lose, cksumerr\}$, $e \in \{sendack\}$, and $f \in \{rcvack, lose, cksumerrack\}$.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this work, we have studied an algorithm that checks whether a legal language is co-observable with respect to a plant. We proved that the algorithm also checks controllability of the legal language with respect to the plant.

Based on this algorithm, a method we have called the all-paths method was developed to find all the sequences of events that violate controllability or co-observability. Manipulations of the all-paths method introduces a second method, the some-paths method, which finds some of the sequences of events causing failure in co-observability or controllability. The some-paths method consists of fewer steps and is easier to implement than the all-paths method. The results obtained are a non-empty subset of the results of the all-paths method, when controllability or co-observability is violated. The computational complexity of the some-paths method is $O(n^{12})$. This can be shown that the computational complexity of the all-paths method is $O(n^{12})$ too.

We implemented the some-paths method to determine whether, for a given plant and a legal language, there were any sequences of events as a counter-

example to controllability or co-observability. Several small examples were verified using our computer program. Telecommunication protocols were also considered. These can be verified by checking controllability and co-observability of their legal behavior with respect to all possible behaviors of the system. Some data-link layer protocols, which are a class of telecommunication protocols, were modelled by Discrete-Event Systems concepts and verified manually. The protocols modelled were very large and therefore, while using the implemented program to verify the data link layer protocols is theoretically possible, it was not practically feasible.

5.2 Future Work

Given a fixed desired language, the all-paths method can be applied to find all tuples of sequences of events that violate co-observability or controllability of the desired language with respect to the plant. It would be helpful if an algorithm could be developed to use the results of the all-paths method to offer some modifications to the plant to get a new plant so that the desired language would be co-observable and controllable with respect to the new plant.

Bibliography

[1] G. R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed.* Addison-Wesley Publishing Company, 1999.

[2] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems.* Boston, Kluwer Academic, 1999.

[3] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation.* Addison-Wesley Publishing Company, 1988.

[4] M. Chang, "Verification of Go-Back-N protocol's modulus problem using discrete event system," Term-paper of ELEC 843, Queen's University, Kingston, Canada, December 14, 2001.

[5] R. Cieslak, C. Desclaux, A. S. Fawaz, and P. Varaiya, "Supervisory control of discrete-event processes with partial observations," *IEEE Transactions on Automatic Control*, Volume 33, Number 3, pages 249-260, 1988.

[6] W. H. J. Feijen and A. J. M. Van Gasteren, *On a Method of Multiprogramming (Monographs in Computer Science).* Springer Verlag, 1999.

[7] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 2nd edition, 2000.

[8] R. Kumar and M. Fabian, "On supervisory control of partial specification arising in protocol conversion," in *Proceedings of 1997 Annual Allerton Conference*, Urbana, IL, pp. 543-552, September 1997.

[9] R. Kumar, S. Nelvagal, and S. I. Marcus, "A discrete event systems approach for protocol conversion," *Discrete Event Dynamical Systems: Theory and Applications*, Volume 7, Number 3, pages 295-315, June 1997.

[10] S. Lafortune, *Software UM-DES for Discrete Event Systems,* `http://www.eecs.umich.edu/umdes`, 1999.

[11] F. Lin and W. M. Wonham, "On observability of discrete event systems," *Information Sciences*, Volume 44, Number 3, pages 173-198, 1988.

[12] M. T. Ozsu and P. Valduriez, *Principles of Distributed Database Systems.* Prentice Hall, 2nd Edition, 1999.

[13] A. Puri, S. Tripakis, and P. Varaiya, "Problems and examples of decentralized observation and control for discrete event systems", in *Synthesis and Control of Discrete Event Systems,* Kluwer Academic Publishers, Edited by B. Caillaud and P. Darondeau and L. Lavagno and X. Xie, pages 37-55, 2001.

[14] P. J. Ramadge and W. M. Wonham, "Supervision of discrete event processes," in *Proceedings of the 21st IEEE Conference on Decision and Control*, Volume 3, pages 1228-1229, December 1982.

[15] P. J. Ramadge and W. M. Wonham, "Supervisory control of a class of discrete-event processes," *SIAM Journal of Control and Optimization*, Volume 25, Number 1, pages 206-230, 1987.

[16] P. J. Ramadge and W. M. Wonham, "The control of discrete-event systems," in *Proceedings of the IEEE*, Volume 77, pages 81-98, 1989.

[17] S. L. Ricker, *Knowledge and Communication in Decentralized Discrete-Event Control,* PhD thesis, Queen's University, Kingston, Ontario, Canada, December 1999.

[18] K. Rudie, "Applications of discrete event system control theory to communication protocols," Internal Report, Bell-Northern Research, Toronto, August 12 , 1988.

[19] K. Rudie and J. C. Willems, "The computational complexity of decentralized discrete-event control problems," Technical Report from the IMA Preprint Series #1105, Institute for Mathematics and Its Applications, University of Minnesota, 1993.

[20] K. Rudie and J. C. Willems, "The computational complexity of decentralized discrete-event control problems," *IEEE Transactions on Automatic Control*, Volume 40, Number 7, pages 1313-1319, 1995.

[21] K. Rudie and J. C. Willems, "Decentralized discrete-event systems and computational complexity," *Discrete Event Systems, Manufacturing Systems, and Communication Networks.* Edited by P. R. Kumar and P. P. Varaiya, Volume 73 of the IMA Volumes in Mathematics and Its Applications Series, Springer-Verlag, pages 225-241, 1995.

[22] K. Rudie and W. M. Wonham, "Supervisory of communication processes," *Protocol Specification, Testing and Verification, X.* Edited by L. Logrippo and R. L. Probert and H. Ural, Elsevier Science Publishers (North-Holland), IFIP, 1990.

[23] K. Rudie and W. M. Wonham, "Think globally, act locally: decentralized supervisory control," *IEEE Transactions on Automatic Control*, Volume 37, Number 11, pages 1692-1708, November 1992.

[24] K. Rudie and W. M. Wonham, "Protocol Verification Using Discrete-Event Systems," in *Proceedings 31st IEEE Conference on Decision and Control (CDC)* , Tucson, AZ, December 16-18, pages 3770-3777, 1992.

[25] M. Sipser, *Introduction to the Theory of Computation.* Brooks Cole Publishing Company, 1996.

[26] W. Stallings, *Data and Computer Communications.* New Jersey: Prentice Hall, 6th Edition, 2000.

[27] A. S. Tanenbaum, *Modern Operating Systems.* New Jersey: Prentice Hall, 1992.

[28] A. S. Tanenbaum, *Computer Networks.* New Jersey: Prentice Hall, 4th Edition, 2002.

[29] A. S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms.* New Jersey: Prentice-Hall, 1st edition, 2002.

[30] A. S. Tanenbaum and S. Woodhull, *Operating Systems Design and Implementation.* New Jersey: Prentice-Hall, 1986.

[31] W. M. Wonham, *Software CTCT for Discrete Event Systems,* `http://www.control.utoronto.ca/cgi-bin/dlctct.cgi`, 2001.

[32] W. M. Wonham, *Notes on Discrete Event Systems,* `www.control.utotonto.ca`, 2002.

[33] W. M. Wonham and P. J. Ramadge, "On the supremal controllable sublanguage of a given language," *SIAM Journal of Control and Optimization,* Volume 25, Number 3, pages 637-659, 1987.

[34] W. M. Wonham and P. J. Ramadge, "Modular supervisory control of discrete-event systems," *Mathematics of Control, Signals, and Systems,* Volume 1, pages 13-30, 1988.

Appendix A

Regular Expression

A regular language is the language accepted by a finite-state automaton. Regular expressions are a notation system used to denote regular languages. Regular expressions describe languages composed of trivial basic components such as characters using the operations of union, concatenation, and Kleene star (repetition). They are represented using the following symbols:

- $+$ for union,
- $\cdot$ for concatenation, and
- $*$ for Kleene star (repetition).

Thus, regular expressions are defined by the following context-free grammar:

$$R \longrightarrow R + R | R \cdot R | R^* | (R) | \epsilon | symbol$$

Some examples of regular expressions are as follows:

$$b \cdot \epsilon \cdot (b^*) + (\epsilon)^* \tag{A.1}$$

$$((a + b)^* \cdot b^*)^* \tag{A.2}$$

$$b \cdot (b^*) + \epsilon \tag{A.3}$$

An infinite number of regular expressions may correspond to the same regular language. For instance, the regular expressions (A.1) and (A.3) are equivalent and both denote the same regular language. However, expression (A.3) is shorter than expression (A.1). The following collection of rules can be used to simplify a regular expression:

$$R \longleftarrow R \cdot \epsilon | \epsilon \cdot R | R + R$$

$$RR_1R_2 \longleftarrow R(R_1R_2)$$

$$R_1R_2R \longleftarrow (R_1R_2)R$$

$$\epsilon \longleftarrow \epsilon \cdot \epsilon | \epsilon + \epsilon | (\epsilon)^*$$

where R, R_1, and R_2 are arbitrary regular expressions and ϵ is a regular expression with length zero.

The above context-free grammar, examples, and simplification rules introduce in-order regular expressions. An in-order regular expression can be easily converted to its equivalent post-order or pre-order format.

Vita

Arezou Mohammadi

Education

2001-2003: Master of Science in Electrical and Computer Engineering, Queen's University at Kingston, Canada

1991-1995: Bachelor of Science in Computer Engineering, Isfahan University of Technology, Isfahan, Iran, 1995

Experience

September 2001- 2003: Teaching Assistant, Electrical and Computer Engineering Department, Queen's University at Kingston, Ontario, Canada

April 1998 -August 2000: Member of the Board, Sazegan Arvin Sepahan Company, Isfahan, Iran

June 1995 - April 1998: Design Engineer, I.S.I.E. Company, Isfahan, Iran

June 1995 - April 1998: January 1995 - June 1995: Software Engineer, Underwater Research Center, Isfahan University of Technology, Isfahan, Iran

January 1995 - August 2000: Part Time Instructor, High school for Gifted Students and Daneshgahi College, Isfahan, Iran