Kai Cai (cai@omu.ac.jp)

Invitation to Supervisory Control of Discrete-Event Systems

with Hands-On Python Software Tool

Kindle Direct Publishing
ISBN: 9798373331449

Book website:
`caikai.org/invitation-scdes`

ver. 1.0; October 15, 2024

To Murray Wonham

Prologue

Immediate after publishing our over 600-page book "Supervisory Control of Discrete-Event Systems" (Springer 2019), Murray Wonham suggested that we begin to prepare the second edition. As strange as it might have sounded, we did put much work towards that direction and the second edition was in its healthy growth, until Murray's sudden passing in a quiet Sunday morning (which happened to be the Father's Day) 2023. While the second edition will eventually be published (which reflects in part Murray's work between 2019 and 2023, until the very moment of his passing), I took a decision to delay the publication as a period for retrospection as well as prospection.

I have contemplated for quite a while how to introduce the supervisory control theory in a manner that is as plain as possible. For a newcomer student, it would be daunting for him or her to be referred to the over 600-page book as a starting point. I also heard from systems control colleagues who were interested in studying supervisory control, but was having hard time in following very different mathematical modeling and tools used. These have been my motivations to write up a bootcamp style book, which can be easily read in a nice Sunday afternoon. Still, essentials of supervisory control are toured for the reader, which can be stepping stones for preparing the reader to continue with more advanced materials.

In writing this book, therefore, I have kept in mind to maximize

the following ratio (often stressed by Murray as well)

$$\frac{\text{cybernetic insights}}{\text{mathematical technicalities}}$$

In particular, the emphasized links to standard control theory (based on differential equations and analysis) should help my colleagues see that supervisory control belongs to the same category but with a different clothes for a different scenario. A student, with or without a control background, should also find it easy to follow the content that requires a fairly moderate level of mathematics, but can appreciate the concept of 'feedback' central for all branches of control theory.

A key companion to the theoretical content is a python toolbox PyTCT, with a set of handy functions that assists modeling, visualization, analysis, control design and computation. The reader is invited to play with PyTCT to get hands-on experience with the theory.

What is supervisory control of discrete-event systems

The supervisory control theory originated in an attempt to craft a control theory for computer engineering. Hence the mathematical models are automata and formal languages (originated from theoretic computer science), while the theory is of control theoretic nature: analysis of system properties of controllability, observability, optimality, and synthesis of supervisory controllers based on the principle of feedback.

The perspective of systems in supervisory control is abstract and of high level. When you see a printer, you don't worry about how various mechatronic parts are interconnected under the hood, but rather think in terms of how the printer operates from mode to mode, what functions there are and how these functions can be used in certain orders. When you drive a car (or ride a bike), as another example, you never worry about how to solve an ordinary differential equation in order to determine the state of your car,

but rather which intersection ahead to make your next right turn, when should you merge to the right lane for the turn, and had you better switch your foot onto the break pedal to prepare for a light that is perhaps turning red soon.

This perspective leads to the type of dynamic systems termed *discrete-event systems* (DES), with a set of discrete states and event-driven dynamics. DES can be abstract models of real physical systems (like the printer and car), where our interest is in functions and high-level strategies instead of low-level minute mechanics or dynamics. DES are also natural models for software systems (no physics involved), like a computer program, a communication protocol, database control, and ChatGPT prompting. Supervisory control is about the control of DES, in terms of sequences of discrete control actions.

What are in this book

This book covers the basic modeling of DES and control design method. The six chapters are divided into two parts:

- Chapters 1, 2, 3: automaton model
- Chapters 4, 5, 6: control design

I have chosen to emphasize the 'state view', as this view seems more intuitive to many students, as well as to align with the same popular view of standard control. In each chapter, nevertheless, I have included a section (last but two) to describe the behavioral view (based on formal languages). These sections can be skipped with no affect to the rest. (As a side note, the behavioral view was also advocated in standard control to develop a model-independent control theory, and is recently revived in the context of data-driven control.) The last section of each chapter collects the PyTCT functions and codes that are relevant to the theoretic content of that chapter.

What is not in this book? Many. First of all, this book only covers the supervisory control theory, leaving out other theories of

DES such as petri nets and max-plus algebra. Neither does this book cover more advanced and important topics in supervisory control, such as partial observation, decentralized control, diagnosis, opacity, and timed DES. By leaving all these interesting topics out, I hope that this book is 'slim' enough to give you a swift start with the theory. At the same time, I hope that the included essential content can make you hungry to explore more. In the Epilogue, you can find a "Further Reading" part (for more beef).

Whom is this book written for

This book is written for anyone who is interested in getting a quick start with supervisory control of DES. In writing the book, I have three categories of audience in mind. First, students new to DES – this book can be a readable manual to get you started. Second, control colleagues who have found DES mathematically foreign – this book aligns with your state view and feedback control instinct. Third, and a little far reaching, lower-year students (freshmen and high schoolers) – I hope this book can be a first introductory course to certain central concepts and ideas in systems and control, without requiring knowledges of differential equations, analysis or linear algebra.

Get started with PyTCT

PyTCT is the Python extension of the supervisory control software package TCT (originated by Murray and his students in 1990s). Thus the core functions have been tested and debugged for three decades, and are extremely stable. PyTCT is freely downloadable from the GitHub link below. Install it and you are good to go to use it locally.

`https://omucai.github.io/PyTCT-docs`

To be more convenient, PyTCT can be used as long as you have a browser and Internet connection, thus with no hassle of downloading or installing the package locally. Thanks to my brilliant

students Masahiro Konishi, Shoma Matsui, and Hiromi Takahashi, we provide a Jupyter Hub where PyTCT is pre-installed. To access this service, go to the site below and follow instructions therein to register a free account. (Account registration requires an email address; not all emails are accepted. If your email doesn't work, either contact me directly or download PyTCT to use it locally.)

`https://jupyter.caikai.org`

Where to find additional material

Supplementary material (slides, codes) and updates to this book can be found on the website below:

`https://www.caikai.org/invitation-scdes`

Kai Cai
October 15, 2024

Contents

Chapter 1

Create An Automaton

1.1 Intuition

Automaton is an intuitive model for describing *dynamic systems* (which are systems whose internal states iteratively change). Let's start with simple examples to build up intuition about automaton.

Example 1.1 *Consider a printer commonly used in office and at home. A printer is initially at an "idle" state. Say we want to print a ten-page document. We can start printing the document by sending a printing job from a computer to the printer. Upon receiving this job, the printer will enter a "working" state: papers are being printed. In a normal situation, the printer will finish printing the ten-page document and automatically return to the "idle" state. It is not uncommon, however, we find some problem in the first few printed pages; say we intended to print the document double-sided but forgot to tick that option. In that case, we would manually stop the rest of the printing by cancelling the printing job; this action makes the printer return also to the "idle" state.*

More problematically, during the printing process papers may get jammed inside the printer (often happening with older printers). This causes the printer to enter a "broken" state (with blinking lights and error codes). Jammed papers

must be removed before the printer may be used again. If the jammed papers are successfully removed, the printer returns again to the "idle" state (ready to be used again).
The above (high-level) working mechanism of a printer can be described by an automaton, graphically displayed in Fig. 1.1. There are three states*:*

"IDLE", "WORKING", "BROKEN"

which are represented by circles and sequentially labeled 0, 1, 2. Between the states, there are five state transitions *(represented by directed edges):*

("IDLE", start*, "WORKING")*
("WORKING", auto_finish*, "IDLE")*
("WORKING", manual_stop*, "IDLE")*
("WORKING", breakdown*, "BROKEN")*
("BROKEN", fix*, "IDLE")*

Each of these transitions is written as a triple, where the first element is the state where the transition exists, the second element the event or action causing the occurrence of the transition, and the third element the state where the transition enters.
In addition, since a printer starts from the "IDLE" state, this state is the initial state *of the automaton. In Fig. 1.1, the initial state 0 is with an incoming edge without source. Finally, the "IDLE" and "WORKING" states are selected as* marker states*, which are 'good states' where the printer is working normally. These two marker states are represented by double-circles.*

In Example 1.1, the three states in the automaton represent three different *operational status* of a printer; thus transitions between states represent *change of status*. At this point the reader

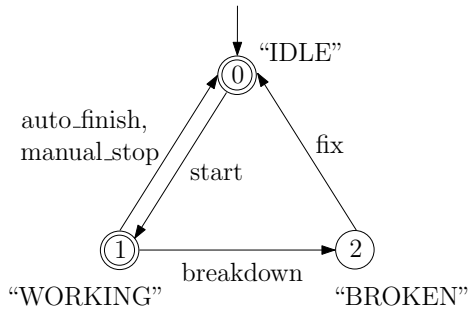


Figure 1.1: Automaton model of printer (the initial state is 0 denoted by a circle with an incoming edge without source; the marker states are 0 and 1 denoted by double-circles; here state 0 is both the initial state and a marker state)

is invited to use automaton to model a coffee machine, a vending machine, and an elevator (say serving a three-story building).

In the next example, we shall see that the states of automaton can also carry physical meanings.

Example 1.2 *Say you ordered this book on Amazon. In many of Amazon's warehouses, mobile robots are employed to automatically fetch ordered items from storage shelves. As displayed in Fig. 1.2, a robot is dispatched from the robot station (area labeled 0) to fetch the ordered book which is stored in a shelf in area 3. To get there, the robot may follow routes either in area 1 or in area 2 (depending on which shelf the ordered book is located). Once picking up the intended book from the shelf, the robot makes its way along another route to deliver the book to the drop-off destination in area 4. There is also a station for human workers (area 5), where the robot is not supposed to enter for safety reasons.*
The dynamics of robot navigation in the warehouse for item pickup and delivery can be modeled by an automaton in Fig. 1.3. There are six states, each representing a labeled

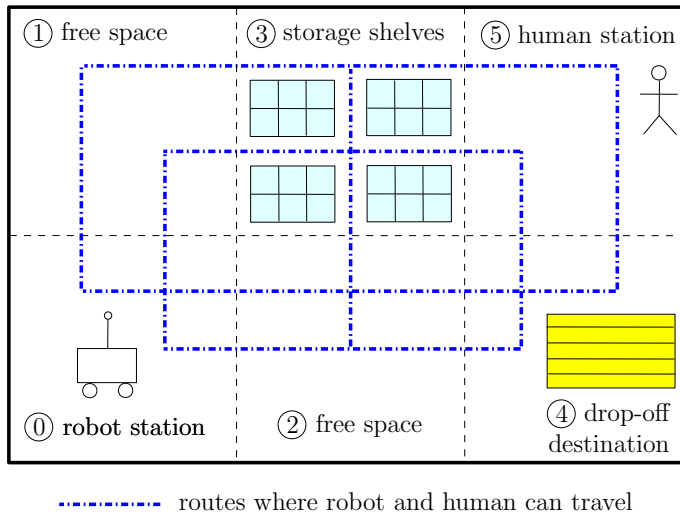


Figure 1.2: Warehouse logistic automation using robot for item pickup and delivery (blue lines represent routes along which robot and human can travel)

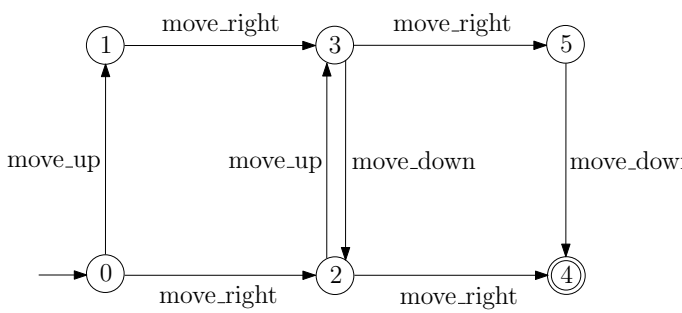


Figure 1.3: Automaton model of warehouse robot

physical area. State 0 of the robot station is the initial state of the automaton, as this is the area where the robot starts its navigation. State 4 of the delivery destination is the marker state, which signifies a successful pickup-delivery

operation. The transitions between these states represent physical movements of the robot from one area to another.

The reader is invited to consider using automaton to model a robot solving a maze (say a 3×3 grid).

1.2 Definition

In the two examples introduced in Section 1.1, we see that the automaton models have a few elements in common. There is a set of *states*, representing activities or locations that the dynamic system is involved. There is also a set of *state transitions*, whose occurrences cause state changes; these state transitions are labeled by *actions* or *events*. Additionally in the set of states, there is a (unique) *initial state* and a subset of *marker states*. These common elements constitute the mathematical definition of an automaton (below).

An *automaton* $\mathbf{A}$ is a tuple of five elements:

$$\mathbf{A} = (Q, \Sigma, \delta, q_0, Q_m) \tag{1.1}$$

We introduce these five elements in order.

- Q is the finite set of *states*. Each state represents an activity, a status, or a location involved in the dynamics of the system. We take the view that each state has duration in time.

- Σ is the finite set of *events*. Each event is a label of an action taken by the system or some external force acting on the system that causes a state transition. We view that each event occurs instantaneously, namely the time a state transition takes is negligible.

- δ is the finite set of *state transition triples*, i.e.

$$\delta \subseteq \{(q, \sigma, q') \mid q, q' \in Q, \sigma \in \Sigma\} \tag{1.2}$$

  For each element $(q, \sigma, q') \in \delta$, q is the current state where the transition exits, σ the event causing the transition, and q' the new state where the transition enters. This transition set δ can also be written as a function, as will be explained in more detail in Section 1.3.

- $q_0 \in Q$ is the (unique) *initial state*. If more than one initial state is needed, say q_{01} and q_{02}, this can be modeled by adding a dummy initial state q_0 and dummy state transitions from q_0 to q_{01} and q_{02}, respectively.

- $Q_m \subseteq Q$ is the subset of *marker states*, which represent those states considered as good or desired. For instance, completion of a task, reaching a goal location, or returning to home station. These states are where the system is desired to be able to visit.

In Example 1.1, the five elements of the automaton (Fig. 1.1) are as follows:

$$\begin{aligned} Q &= \{0, 1, 2\} \\ \Sigma &= \{\text{start}, \text{auto\_finish}, \text{manual\_stop}, \text{breakdown}, \text{fix}\} \\ \delta &= \{(0, \text{start}, 1), (1, \text{auto\_finish}, 0), (1, \text{manual\_stop}, 0), \\ &\qquad (1, \text{breakdown}, 2), (2, \text{fix}, 0)\} \\ q_0 &= 0 \\ Q_m &= \{0, 1\} \end{aligned}$$

In Example 1.2, the five elements of the automaton

(Fig. 1.3) are:

$$
\begin{aligned}
Q &= \{0, 1, 2, 3, 4, 5\} \\
\Sigma &= \{\text{move\_up}, \text{move\_down}, \text{move\_right}\} \\
\delta &= \{(0, \text{move\_up}, 1), (0, \text{move\_right}, 2), (1, \text{move\_right}, 3), \\
&\qquad (2, \text{move\_up}, 3), (2, \text{move\_right}, 4), (3, \text{move\_down}, 2), \\
&\qquad (3, \text{move\_right}, 5), (5, \text{move\_down}, 4)\} \\
q_0 &= 0 \\
Q_m &= \{4\}
\end{aligned}
$$

1.3 State transition function

The set δ of state transitions in (1.2) can be viewed as a function which maps a pair (q, σ) to a state q'. Thus $\delta : Q \times \Sigma \to Q$, where the domain is the *cartesian product* of the state set Q and the event set Σ:

$$Q \times \Sigma = \{(q, \sigma) \mid q \in Q, \sigma \in \Sigma\}$$

Hence $\delta(q, \sigma) = q'$ means that when event σ occurs at state q, state transitions to a new state q'. It is worth noting that in general δ is not defined for every pair (q, σ) in the domain $Q \times \Sigma$, so as a function δ is only *partial*. We say that $\delta : Q \times \Sigma \to Q$ is a partial *state transition function*, and write

$$\delta(q, \sigma)!$$

to mean that δ is defined for the pair (q, σ), while $\neg\delta(q, \sigma)!$ otherwise.

In Example 1.1, the state transition function is as follows:

$$\delta(0, \text{start})! \ \& \ \delta(0, \text{start}) = 1$$
$$\delta(1, \text{auto\_finish})! \ \& \ \delta(1, \text{auto\_finish}) = 0$$
$$\delta(1, \text{manual\_stop})! \ \& \ \delta(1, \text{manual\_stop}) = 0$$
$$\delta(1, \text{breakdown})! \ \& \ \delta(1, \text{breakdown}) = 2$$
$$\delta(2, \text{fix})! \ \& \ \delta(2, \text{fix}) = 0$$

On the other hand, the following are three examples of δ not being defined:

$$\neg\delta(0, \text{breakdown})!, \quad \neg\delta(1, \text{start})!, \quad \neg\delta(2, \text{auto\_finish})!$$

The partial state transition function in Example 1.2 is similar and left as an exercise to the reader.

Now that we know how to use the state transition function δ to describe *one-step* transitions, we can composite function δ to describe *multi-step* transitions. Say we have

$$\delta(q_1, \sigma_1)! \ \& \ \delta(q_1, \sigma_1) = q_2$$
$$\delta(q_2, \sigma_2)! \ \& \ \delta(q_2, \sigma_2) = q_3$$
$$\delta(q_3, \sigma_3)! \ \& \ \delta(q_3, \sigma_3) = q_4$$

Then the following is a three-step transition:

$$\delta(\delta(\delta(q_1, \sigma_1), \sigma_2), \sigma_3) = q_4$$

More compactly, it is convenient to have a transition function $\hat{\delta}$ to represent the above by writing

$$\hat{\delta}(q_1, \sigma_1\sigma_2\sigma_3)! \ \& \ \hat{\delta}(q_1, \sigma_1\sigma_2\sigma_3) = q_4$$

Here the sequence of events "$\sigma_1\sigma_2\sigma_3$" is called a *string*. In general, a string s on the event set Σ is a sequence of finite number of events in Σ. The *length* of string s, written $|s|$, is the number of constituent events. Thus the length of an arbitrary string is finite, though there is no bound for the length.

Now let's denote by Σ^+ the set of all strings on Σ, namely

$$\Sigma^+ = \{\sigma_1 \cdots \sigma_k \mid k \geq 1 \text{ is finite } \& \ (\forall i \in [1,k])\sigma_i \in \Sigma\}$$

Additionally bring in a special *empty string* ϵ which contains no event. Thus $\epsilon \notin \Sigma^+$ and $|s| = 0$. Define

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

Thus Σ^* consists of all strings including the empty string (and is known as the *kleene closure*).

Now let's extend the state transition function $\delta : Q \times \Sigma \to Q$ to

$$\hat{\delta} : Q \times \Sigma^* \to Q$$

First we need

$$(\forall q \in Q)\hat{\delta}(q,\epsilon)! \ \& \ \hat{\delta}(q,\epsilon) = q \tag{1.3}$$

This means that the empty string ϵ is defined at every state q; since ϵ physically means 'doing nothing', its occurrence trivially makes no state transition (you may imagine there is a virtual ϵ-selfloop at every state). Next, we require inductively

$$(\forall q \in Q, \forall s \in \Sigma^*, \forall \sigma \in \Sigma)\hat{\delta}(q,s)! \ \& \ \delta(\hat{\delta}(q,s),\sigma)! \Rightarrow \hat{\delta}(q,s\sigma)! \tag{1.4}$$

In words, whenever a string is defined at a state (i.e. $\hat{\delta}(q,s)!$) and an event is defined following that string (i.e. $\delta(\hat{\delta}(q,s),\sigma)!$), the new

string $s\sigma$ (*catenation* of s and σ) is defined at state q.

The above (1.3) and (1.4) define the extended transition function $\hat{\delta}$, which can be used to describe multi-step state transitions, including the trivial zero-step transition by the empty string ϵ. As being extended based on the partial transition function δ, here $\hat{\delta}$ is also partial and we write $\hat{\delta}(q, s)!$ to mean that $\hat{\delta}$ is defined for the pair (q, s) and $\neg\hat{\delta}(q, s)!$ otherwise. Henceforth for convenience, we will drop the $\hat{}$ from $\hat{\delta}$ and simply write $\delta : Q \times \Sigma^* \to Q$.

In Example 1.1, the following are instances of the extended state transition function:

$$\delta(0, \epsilon)! \ \& \ \delta(0, \epsilon) = 0$$
$$\delta(0, \text{start.auto\_finish})! \ \& \ \delta(0, \text{start.auto\_finish}) = 0$$
$$\delta(1, \text{breakdown.fix.start})! \ \& \ \delta(1, \text{breakdown.fix.start}) = 1$$
$$\delta(2, \text{fix})! \ \& \ \delta(2, \text{fix}) = 0$$

A few examples of δ not being defined are as follows:

$$\neg\delta(0, \text{manual\_stop.start})!$$
$$\neg\delta(1, \text{breakdown.start})!$$
$$\neg\delta(2, \text{breakdown})!$$

The reader is invited to write instances of the extended state transition function in Example 1.2.

1.4 Automaton as dynamic system

With the state set Q and the (extended) state transition function δ, an automaton is a dynamic system whose state changes in Q are described using δ.

Given an automaton $\mathbf{A} = (Q, \Sigma, \delta, q_0, Q_m)$, its working mech-

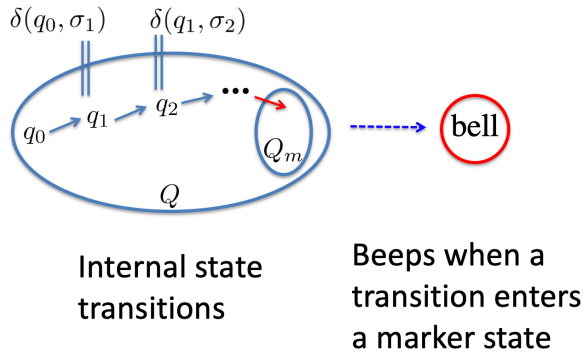


Figure 1.4: Automaton as dynamic system

anism can be visualized in Fig. 1.4. All dynamics starts from the initial state q_0. Upon an occurrence of an event $\sigma_1 \in \Sigma$ defined at q_0, the state transits to the next $q_1 = \delta(q_0, \sigma_1)$. Then at q_1, if another event $\sigma_2 \in \Sigma$ occurs, this causes another state transition to $q_2 = \delta(q_1, \sigma_2)$. Evolving in this fashion, a sequence of state transitions occurs, which can be identified by a string $s = \sigma_1\sigma_2 \cdots \in \Sigma^*$ that causes these transitions. This sequence of state transitions starting from the initial state q_0 is a *trajectory* of the automaton. The collection of all possible trajectories of the automaton is written

$$L(\mathbf{A}) := \{s \in \Sigma^* \mid \delta(q_0, s)!\} \tag{1.5}$$

We call $L(\mathbf{A})$ the *closed behavior* of $\mathbf{A}$, which includes all possible dynamics of the automaton.

If a trajectory ends at a marker state in Q_m, then this trajectory is considered as a good or desired one. We can imagine that there is a bell attached to the automaton, which beeps whenever a trajectory hits a marker state. The collection of all good

trajectories is written

$$L_m(\mathbf{A}) := \{s \in L(\mathbf{A}) \mid \delta(q_0, s) \in Q_m\} \tag{1.6}$$

Thus $L_m(\mathbf{A})$ is a subcollection of the closed behavior, and we call it the *marked behavior* of $\mathbf{A}$.

In Example 1.1, the automaton's trajectories are e.g.

ϵ, start, start.auto_finish, start.breakdown, start.breakdown.fix, start.breakdown.fix.start

All these trajectories are in the closed behavior of the automaton, but not all are in the marked behavior: the trajectory start.breakdown does not hit a marker state, so it is not in the marked behavior of the automaton.
For the automaton in Example 1.2, the reader is invited to write three trajectories which are in both the closed behavior and the marked behavior, while another three trajectories in the closed behavior but not in the marked behavior.

1.5 PyTCT

PyTCT is a python version of the TCT software package.[1] Compared to TCT, PyTCT allows OS-independent installation, writing scripts, and using vast libraries in python whenever needed.

In this section, we use PyTCT to create an automaton, display it, and simulate its trajectories. We start with the following two lines of codes, which should be included at the top of every script.

```
1 import pytct #import pytct package
2
3 pytct.init('xxx') #create a working folder
```

[1] https://www.control.utoronto.ca/~wonham/TCTX64_20210701.rar

The first line imports the PyTCT package. Then a working folder "xxx" (insert any name) is created by the **init** function to save all computation results. Optionally the following allows overwriting a created folder.

```
pytct.init('xxx', overwrite=True)
#allow overwriting a created working folder
```

In PyTCT, an automaton is created using the **create** function. To use this function, three elements need to be specified.

- State number: the number of states of the automaton to be created. Once a state number $n(\geq 1)$ is given, all the states are sequentially labeled from 0 to $n-1$. The initial state is labeled 0.
- Set of state stransitions: all the transition triples of the automaton. For the event labels, two formats are permitted: (i) strings like 'start', 'finish' (with single quotes); (ii) natural numbers like 0, 15 (without single quotes). Two formats are not allowed to be mixed in the same script.
- Set of marker states: a subset of the state set $\{0, \ldots, n-1\}$ that is marked.

Let's create the automaton of the printer in Fig. 1.1.

```
statenum=3 #number of states
#states are sequentially labeled 0,1,...,statenum
#initial state is labeled 0

trans=[(0,'start',1),
       (1,'auto_finish',0),
       (1,'manual_stop',0),
       (1,'breakdown',2),
       (2,'fix',0)] # set of transitions
#each triple is (exit state, event label, entering state)

marker = [0,1] #set of marker states

pytct.create('PRINTER', statenum, trans, marker)
```

```
#create automaton PRINTER
```

The last line of code above uses the **create** function, which has four inputs: (i) name of the automaton to be created (a string in single quotes), (ii) state number, (iii) set of transitions, and (iv) set of marker states. This function creates an automaton PRINTER, which is saved as "PRINTER.DES" in the folder "xxx". The automaton can be visualized using the **display_automaton** function below (for this display, the package *graphviz* is used).

```
pytct.display_automaton('PRINTER')
#plot PRINTER.DES
```

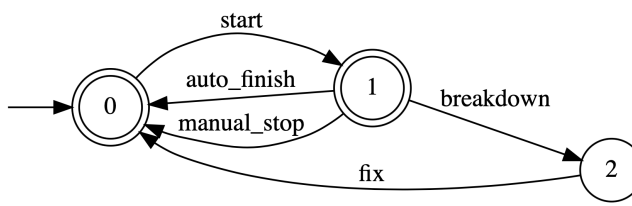


Figure 1.5: PyTCT plot of automaton

The above generates the plot of the automaton PRINTER in Fig. 1.5. It is easily inspected that this automaton is the same as the one in Fig. 1.1.

For a given automaton **A** (file "A.DES" in folder "xxx"), the following functions are handy to obtain the key information of the automaton.

```
pytct.statenum('A') #state number of A.DES

pytct.events('A') #events of A.DES

pytct.trans('A') #state transitions of A.DES

pytct.marker('A') #marker states of A.DES
```

Given an automaton **A** and a string $s \in L(\mathbf{A})$, the corresponding state trajectory can be generated by function **simu-**

late_automaton. The code for automaton PRINTER and a string start.breakdown.fix is as follows. The result is (0,1,2,0).

```
pytct.simulate_automaton('PRINTER',['start','breakdown','
    fix'])
#simulate PRINTER.DES with a string
```

One can also randomly sample a given automaton with a string of a specified length. The following code is to sample automaton PRINTER with a string of length 5. Each execution may generate a different state trajectory, as there is more than one string of length 5 from the initial state.

```
pytct.sample_automaton('PRINTER',5])
#sample PRINTER.DES with a string of length 5
```

The reader is invited to install PyTCT or log into our Jupyter Hub and try all the above introduced functions for the warehouse robot example in Fig. 1.3.

Chapter 2

Trim An Automaton

2.1 Reachable

Given an automaton **A**, we introduce *reachability* of its states from the initial state.

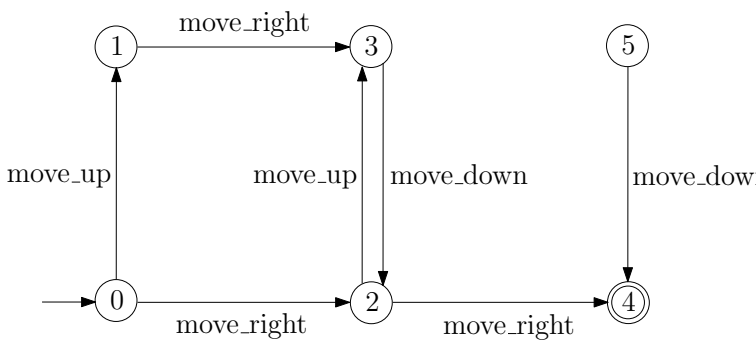


Figure 2.1: Warehouse robot revised: transition to human station removed

Example 2.1 *As displayed in Fig. 2.1, consider a revised version of the warehouse robot automaton, where a transition from state 3 to state 5 is removed. Namely, there is no movement of the robot from the storage shelf area to the human worker station (say blocked by a door). Let's inspect which states can be reached from the initial state 0, via a sequence of actions (i.e. a string).*

- *state 0: (trivially) reachable from 0 via the empty string* ϵ
- *state 1: reachable from 0 via string* move_up
- *state 2: reachable from 0 via e.g. string* move_right *or string* move_up.move_right.move_down
- *state 3: reachable from 0 via e.g. string* move_right. move_up *or string* move_up.move_right
- *state 4: reachable from 0 via e.g. string* move_right. move_right *or string* move_up.move_right. move_down.move_right
- *state 5:* not *reachable from 0*

Therefore in this revised robot automaton, except for state 5, all the other states are reachable from the initial state 0.

Given an automaton $\mathbf{A} = (Q, \Sigma, \delta, q_0, Q_m)$, we say that a state $q \in Q$ is *reachable* (from the initial state q_0) if there exists a string s defined at q_0 that reaches q, i.e.

$$(\exists s \in L(\mathbf{A}))\delta(q_0, s)! \ \& \ \delta(q_0, s) = q$$

Thus a reachable state is one that a trajectory of the automaton can visit. Now let

$$\mathcal{R}(Q) := \{q \in Q \mid q \text{ is reachable}\} \subseteq Q$$

be the subset of all reachable states. We say that the automaton $\mathbf{A}$ is *reachable* if $\mathcal{R}(Q) = Q$, i.e. every state in Q can be visited. Thus the automaton in Example 2.1 is *not* reachable, because $\mathcal{R}(Q) \subsetneqq Q$.

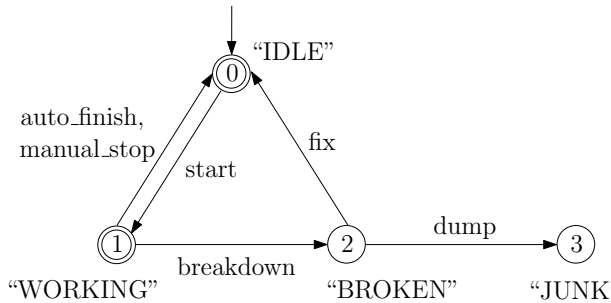


Figure 2.2: Printer revised: junk state added

Example 2.2 *Let's consider another example (Fig. 2.2), which is a revised version of the printer automaton with a new state 3 and transition (2,*dump*,3) added. Namely, the printer once (severely) broken may be directly dumped. In this automaton, inspect that every state is reachable (from the initial state 0). Hence the automaton itself is reachable.*

The reader is invited to work out the details of the above example (as done in Example 2.1) the reachability of every state to confirm that the revised printer automaton is indeed reachable.

2.2 Coreachable

A concept dual to reachability is *coreachability*. Given an automaton $\mathbf{A} = (Q, \Sigma, \delta, q_0, Q_m)$, we say that a state $q \in Q$ is *coreachable* (with respect to the marker state set Q_m) if there exists a string s defined at q that reaches a state in Q_m, i.e.

$$(\exists s \in \Sigma^*)\delta(q, s)! \ \& \ \delta(q, s) \in Q_m$$

Thus a coreachable state has at least one trajectory to reach a marker state. Let

$$\mathcal{CR}(Q) := \{q \in Q \mid q \text{ is coreachable}\} \subseteq Q$$

be the subset of all coreachable states. This is the set of states from which the marker state set Q_m can be visited. We say that the automaton $\mathbf{A}$ is *coreachable* if every state in Q is reachable, i.e. $\mathcal{CR}(Q) = Q$.

The revised printer automaton in Example 2.2 is *not* coreachable because $\mathcal{CR}(Q) \subsetneqq Q$. Let's inspect which states are coreachable and which are not, with respect to the marker state set $Q_m = \{0, 1\}$.

- state 0: coreachable via e.g. the empty string ϵ or string start
- state 1: coreachable via e.g. the empty string ϵ or string auto_finish or string manual_stop
- state 2: coreachable via e.g. string fix
- state 3: *not* coreachable

Since not all the states are coreachable, this automaton is *not* coreachable.

In Example 2.1, inspect that every state is coreachable, including the non-reachable state 5. Therefore this revised robot automaton is coreachable.

The reader is invited to work out the details of the above example on the coreachability of every state to confirm that the revised robot automaton is indeed coreachable.

2.3 Nonblocking

From systems design point of view, it is desired that every reachable state be coreachable, so that wherever the system visits there is always a path to a marker state. This leads to the definition of *nonblocking automaton.*

Given an automaton $\mathbf{A} = (Q, \Sigma, \delta, q_0, Q_m)$, we say that $\mathbf{A}$ is *nonblocking* if

$$(\forall q \in Q) q \text{ is reachable} \Rightarrow q \text{ is coreachable}$$

Thus in a nonblocking automaton, every reachable state is also coreachable. That is, $\mathcal{R}(Q) \subseteq \mathcal{CR}(Q)$ (reachable part of automaton $\mathbf{A}$ be coreachable). In the special case where $\mathbf{A}$ is reachable (i.e. $\mathcal{R}(Q) = Q$), then nonblocking is equivalent to coreachable (i.e. $\mathcal{CR}(Q) = Q$).

Note that the above definition is concerned only with reachable states. Thus a nonblocking automaton may contain non-reachable states. This is reasonable because if a state is not even reachable, it does not play any significant role whatsoever.

Note also that a coreachable automaton is nonblocking (trivially so because the right-hand side of the implication always holds). However a reachable automaton need not be nonblocking.

Finally, the definition of nonblocking is with regard to a given automaton, but not with regard to an individual state. This is different from previously introduced reachable/coreachbale states. Nevertheless one may define a *blocking state* as follows. For a given state $q \in Q$, we say that q is *blocking* if

$$q \text{ is reachable } \& \ q \text{ is not coreachable}$$

Such a blocking state q, whenever visited, has no hope of getting to any marker state. An automaton is *blocking* if it contains a blocking state. A special blocking state q is called a *deadlock state*

if no event is defined at q, i.e.

$$q \text{ is blocking } \& \ (\forall \sigma \in \Sigma) \neg \delta(q, \sigma)!$$

The revised robot automaton in Example 2.1 is nonblocking because it is coreachable.
On the other hand, the revised printer automaton in Example 2.2 is blocking because state 3 ("JUNK") is a blocking state (indeed a deadlock state).

2.4 Trim

A nonblocking automaton without nonreachable states is called a *trim automaton*. Given an automaton $\mathbf{A} = (Q, \Sigma, \delta, q_0, Q_m)$, we say that $\mathbf{A}$ is *trim* if

$$(\forall q \in Q) q \text{ is reachable } \& \ q \text{ is coreachable}$$

Namely in a trim automaton, every state is both reachable and coreachable, i.e. $\mathcal{R}(Q) = \mathcal{CR}(Q) = Q$.

By definition a trim automaton is nonblocking, but the reverse need not true. For the reverse to be true, reachability needs to be added. That is, if an automaton is nonblocking and reachable, then it is trim.

To transform a nontrim automaton into a trim automaton, one simply need to remove all nonreachable and noncoreachable states and the associated transitions. As a result, a trim automaton is both reachable and coreachable.

The revised robot automaton in Example 2.1 is nonblocking but not trim, for state 5 is not reachable. To make this automaton trim, one removes state 5 and the associated

transition (5, move_down, 4).
Neither is the revised printer automaton in Example 2.2 trim, because state 3 is not coreachable. To make this automaton trim, one removes state 3 and the associated transition (2, dump, 3).

2.5 Behavioral characterization

Recall from Section 1.4 that an automaton $\mathbf{A} = (Q, \Sigma, \delta, q_0, Q_m)$ (as a dynamic system) has two types of behaviors: Closed behavior

$$L(\mathbf{A}) := \{s \in \Sigma^* \mid \delta(q_0, s)!\}$$

is the set of all trajectories of $\mathbf{A}$ starting from the initial state q_0, and marked behavior

$$L_m(\mathbf{A}) := \{s \in L(\mathbf{A}) \mid \delta(q_0, s) \in Q_m\}$$

the subset of trajectories of $\mathbf{A}$ that can hit a marker state in Q_m. These two behaviors provide alternative descriptions of the properties introduced in the previous sections of this chapter: reachability, coreachability, nonblocking, and trim. For that, we bring in two additional concepts.

The first concept is an extension of $L(\mathbf{A})$ from regarding the initial state q_0 to regarding an arbitrary state $q \in Q$. For $q \in Q$ let

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

Thus $L(\mathbf{A}, q)$ is the set of all trajectories starting from state q.

The second concept is *closure of behavior*. Given a behavior L (i.e. language), let $s \in L$ be an arbitrary trajectory (i.e. string).

Write

$$\overline{s} := \{s' \mid (\exists t) s't = s\}$$

Namely $\overline{s}$ is the set of all (history) trajectories that can be extended to s itself. Note that the empty string ϵ and s itself belong to $\overline{s}$. Now define the closure of L by

$$\overline{L} := \{\overline{s} \mid s \in L\}$$

Thus $\overline{L}$ is the set of all history trajectories of its elements. By the definitions of the closed behaviors $L(\mathbf{A})$ and $L(\mathbf{A}, q)$, every trajectory s inside also has its history trajectories inside, hence

$$\overline{L(\mathbf{A})} = L(\mathbf{A}), \quad \overline{L(\mathbf{A}, q)} = L(\mathbf{A}, q)$$

This is not the case, however, for the marked behavior $L_m(\mathbf{A})$ because a trajectory hitting a marker state does not mean its history trajectories can also hit a marker state. Therefore

$$L_m(\mathbf{A}) \subseteq \overline{L_m(\mathbf{A})}$$

With the above two new concepts introduced, we present the behavioral characterizations of reachability, coreachability, nonblocking, and trim.

Automaton $\mathbf{A}$ is

- reachable if and only if for every state $q \in Q$,

$$(\exists s \in L(\mathbf{A}))q = \delta(q_0, s)$$

- coreachable if and only if for every state $q \in Q$,

$$(\exists q_m \in Q_m)(\exists s \in L(\mathbf{A}, q))q_m = \delta(q, s)$$

- nonblocking if and only if

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

(namely every trajectory in $L(\mathbf{A})$ can be extended to a trajectory to hit a marker state)

- trim if and only if

$$\mathbf{A} \text{ is reachable } \& \ \overline{L_m(\mathbf{A})} = L(\mathbf{A})$$

2.6 PyTCT

Let's create the revised printer automaton in Example 2.2. Call this automaton PRINTER_JUNK, created and displayed below.

```
import pytct #import pytct package

pytct.init('RevisedPrinter') #create a working folder

statenum=4 #number of states
#states are sequentially labeled 0,1,...,statenum
#initial state is labeled 0

trans=[(0,'start',1),
       (1,'auto_finish',0),
       (1,'manual_stop',0),
       (1,'breakdown',2),
       (2,'dump',3)
       (2,'fix',0)] # set of transitions
#each triple is (exit state, event label, entering state)

marker = [0,1] #set of marker states

pytct.create('PRINTER_JUNK', statenum, trans, marker)
#create automaton PRINTER_JUNK

pytct.display_automaton('PRINTER_JUNK')
```

```
23 #plot PRINTER_JUNK.DES
```

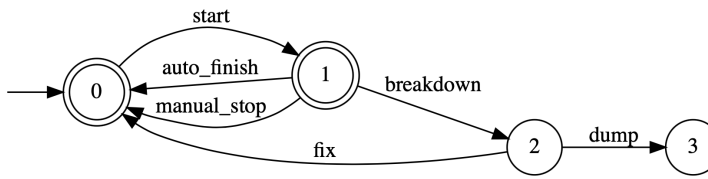


Figure 2.3: PyTCT plot of automaton PRINTER_JUNK

As is seen in Fig. 2.3, the added new state is 3 with the associated new transition (2,dump,3).

Now let's check the reachability, coreachability, nonblocking, and trim properties of automaton PRINTER_JUNK. First the function below checks reachability of the automaton. The result is *true*.

```
1 pytct.is_reachable('PRINTER_JUNK')
2 #check if PRINTER_JUNK is reachable
```

One can also check the reachability of an individual state. The following code checks if state 3 in PRINTER_JUNK is reachable. The result is *true*.

```
1 pytct.is_reachable('PRINTER_JUNK',3)
2 #check if state 3 in PRINTER_JUNK is reachable
```

In addition, if a state is reachable, a (shortest) string reaching the state from the initial state can be generated by the following function. The result is (start, breakdown, junk).

```
1 pytct.reachable_string('PRINTER_JUNK', 3)
2 #generate a string to the reachable state
```

The same set of the above three functions is also available for checking coreachability. First the function below checks coreachability of automaton PRINTER_JUNK. The result is *false*.

```
1 pytct.is_coreachable('PRINTER_JUNK')
2 #check if PRINTER_JUNK is coreachable
```

Coreachability of individual states can be checked as follows. For state 2, it is coreachable; while for state 3, it is not coreachable, and this is the reason why the automaton is not coreachable.

```
pytct.is_coreachable('PRINTER_JUNK',2)
#check if state 2 in PRINTER_JUNK is coreachable

pytct.is_coreachable('PRINTER_JUNK',3)
#check if state 3 in PRINTER_JUNK is coreachable
```

Finally, for a coreachable state, a (shortest) string from the state to a marker state can be generated by the following function. The result for state 2 is (fix).

```
pytct.coreachable_string('PRINTER_JUNK', 2)
#generate a string from the reachable state to a marker
    state
```

For a small automaton (with a handful of states), it is convenient to list out reachability and coreachability of all individual states. This can be done by the following code using a for-loop.

```
for i in range(statenum):
    print(f"state {i}: reachable -> {pytct.is_reachable('
    PRINTER_JUNK', i)}")
    print(f"state {i}: coreachable -> {pytct.is_coreachable
    ('PRINTER_JUNK', i)}")
#list reachability and coreachability of individual states
```

It is sometimes convenient to know a shortest string from an arbitrary state (which need not be the initial state) to another (reachable) state. The following function does so.

```
pytct.shortest_string('PRINTER_JUNK', 1, 3)
#generate a string from state 1 to state 3
```

Now let's move on to check nonblocking and trim properties. For nonblocking of automaton PRINTER_JUNK, the function below verifies that it is not, namely the automaton is blocking.

```
pytct.is_nonblocking('PRINTER_JUNK')
#check if PRINTER_JUNK is nonblocking
```

For a blocking automaton, the following function generates the set of all blocking states (which are reachable but not coreachable). For PRINTER_JUNK, the result is 3. That is, the newly added state is the blocking state, as from this (JUNK) state there is no path back to any marker state.

```
pytct.blocking_states('PRINTER_JUNK')
#generate all blocking states in PRINTER_JUNK
```

Finally, the following function checks if PRINTER_JUNK is trim. The result is *False*.

```
pytct.is_trim('PRINTER_JUNK')
#check if PRINTER_JUNK is trim
```

To convert a nontrim automaton to a trim automaton, the function below is used.

```
pytct.trim('PRINTER_TRIM','PRINTER_JUNK')
#trim an automaton

pytct.display_automaton('PRINTER_TRIM')
#plot PRINTER_TRIM.DES
```

The resulting trim automaton PRINTER_TRIM is displayed in Fig. 2.4. Observe that PRINTER_TRIM is the same automaton as PRINTER in the previous chapter, and resulted by removing from PRINTER_JUNK the newly added state and transition.

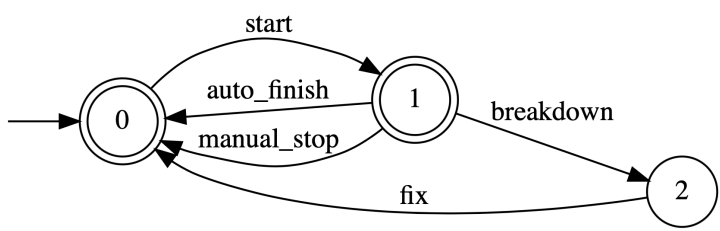


Figure 2.4: PyTCT plot of automaton PRINTER_TRIM

The reader is invited to work out all the above introduced functions for the revised robot automaton in Fig. 2.1.

Chapter 3

Multiply Two Automata

3.1 Intuition

A 'big' automaton may be formed by *product* of 'smaller' automata. Consider two automata $\mathbf{A}_1$ and $\mathbf{A}_2$, whose event sets are Σ_1 and Σ_2 respectively. In general, Σ_1 and Σ_2 can share some events in common (i.e. $\Sigma_1 \cap \Sigma_2 \neq \emptyset$), and also have distinct nonshared events (i.e. $\Sigma_1 \neq \Sigma_2$). The product of $\mathbf{A}_1$ and $\mathbf{A}_2$ should enforce synchronization on executing their shared events, while allow free execution of their distinct events. This is *synchronous product* of automata. Before the formal definition, let's look at an example.

Example 3.1 *Consider a printer automaton displayed on the left of Fig. 3.1 (the same one as that in Fig. 1.1). Also consider a user of this printer (think of your own experience). A user can start a printing job, manually stop the job in process, and when the job finishes successfully, take the printouts. This is modeled by the automaton USER displayed on the right of Fig. 3.1. As expressed in this automaton, once the event* auto_finish *occurs (the automaton reaches state 1), user only takes printouts (i.e. event* take_printouts*) before s/he can start or manually stop a printing job again.*

Observe that these two automata share the following events

in common: start, manual_stop, auto_finish. *On the other hand, the events* breakdown *and* fix *belong only to PRINTER, while the event* take_printouts *belong only to USER.*

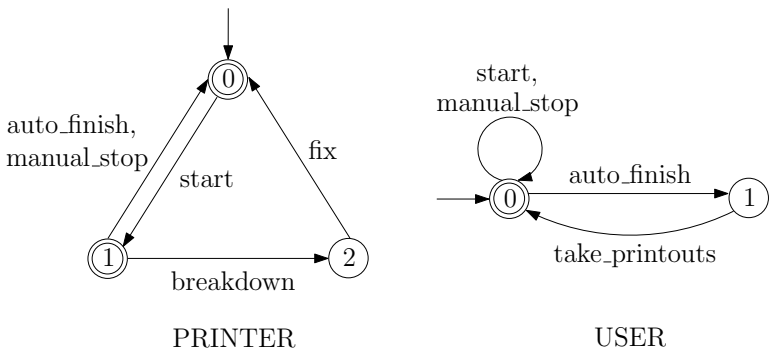


Figure 3.1: Automaton of a printer and automaton of a user

Now let's consider what events these two automata can execute together, while others independently. The rule is, a shared event must be executed together, while a nonshared event can be executed by the owner automaton alone. Let's start from the initial states of both automata, i.e. the state pair (0,0). PRINTER can execute only one event start, whereas USER can execute start, manual_stop, or auto_finish. Thus at (0,0), the shared event start is executed together, which leads to a new state pair (1,0) (because PRINTER transitions to state 1 and USER selfloops at state 0). This is depicted in Fig. 3.2.

Note that USER cannot execute two other events manual_stop or auto_finish, since these two events are shared by PRINTER and cannot be executed by PRINTER at its state 0. Thus what is depicted in Fig. 3.2 is all that can

occur at the initial state pair (0,0), and (1,0) is the sole new state pair.

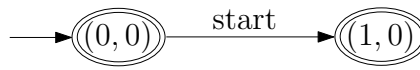


Figure 3.2: Synchronous product of PRINTER and USER: step 1

Next we consider the new state pair (1,0). By inspection we see that there are three events that can be executed at (1,0): manual_stop, auto_finish (shared and executable by PRINTER and USER), and breakdown (nonshared and executable by the owner PRINTER). Note that although the shared event start can be executed by USER, it cannot be executed by PRINTER, as a result start cannot occur at (1,0). The execution of manual_stop, auto_finish, or breakdown leads respectively to (0,0), (0,1), or (2,0); the latter two are new state pairs. This is displayed in Fig. 3.3.

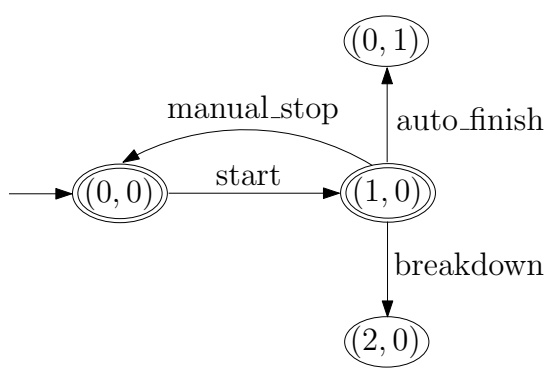


Figure 3.3: Synchronous product of PRINTER and USER: step 2

Now that there are two new state pairs (0,1) and (2,0), we consider them in order. First for (0,1), the only event that can occur is the nonshared event take_printouts (the shared event start cannot occur because it cannot be executed by USER). Execution of take_printouts returns to the initial (0,0). Second for (2,0), since PRINTER cannot execute any shared event, again no shared event can occur. The only event that can occur is the nonshared fix, which can be executed by PRINTER and returns to the initial (0,0). This step is displayed in Fig. 3.4.
Since no more new state pairs are obtained, this construction process ends. Note that among all possible six pairs, only four appear (and the rest two are nonreachable). We end this example by remarking that the marker state pairs of the resulting product automaton are (0,0) and (1,0), because the components are marker states of the respective automata PRINTER and USER.

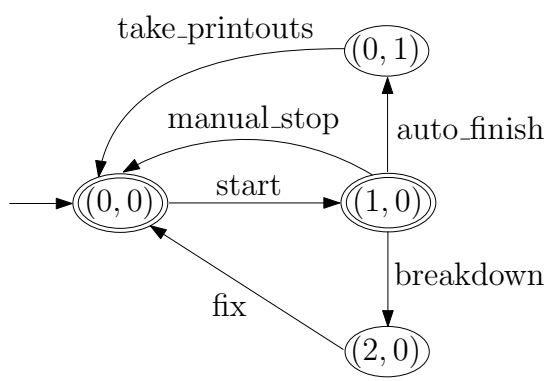


Figure 3.4: Synchronous product of PRINTER and USER: step 3

3.2 Definition

From the example in the preceding section, the synchronous product of two automata is another automaton, whose states are pairs of states of the component automata and whose transitions enforce synchronization on shared events, while allow occurrences of nonshared events. We present the mathematical definition below.

Let $\mathbf{A}_i := (Q_i, \Sigma_i, \delta_i, q_{0,i}, Q_{m,i})$, $i = 1, 2$, be two automata. Their *synchronous product* , written $\mathbf{A}_1||\mathbf{A}_2$, is a new automaton

$$\mathbf{A}_1||\mathbf{A}_2 = \mathbf{A} = (Q, \Sigma, \delta, q_0, Q_m)$$

whose elements are defined as follows.

- State set Q is the cartesian product of Q_1 and Q_2:

  $$Q = Q_1 \times Q_2 = \{(q_1, q_2) \mid q_1 \in Q_1, q_2 \in Q_2\}$$

  which consists of pairs of states from $\mathbf{A}_1$ and $\mathbf{A}_2$.

- Event set Σ is the union of that $\mathbf{A}_1$ and $\mathbf{A}_2$:

  $$\Sigma = \Sigma_1 \cup \Sigma_2$$

  Note that Σ_1 and Σ_2 do not have any special relations.

- State transition function $\delta : Q \times \Sigma \to Q$ is as follows: Let $q = (q_1, q_2) \in Q$ and $\sigma \in \Sigma_1 \cup \Sigma_2$. There are four cases.

Case 1 : if $\sigma \in \Sigma_1 \setminus \Sigma_2$ and $\delta_1(q_1, \sigma)!$ (i.e. σ is a nonshared event that can be executed by $\mathbf{A}_1$ at q_1), then

$$\delta(q, \sigma) = q' = (\delta_1(q_1, \sigma), q_2)$$

Namely $\mathbf{A}_1$ makes a state transition by executing σ while $\mathbf{A}_2$ stays put.

Case 2 : if $\sigma \in \Sigma_2 \setminus \Sigma_1$ and $\delta_2(q_1, \sigma)!$ (i.e. symmetric case to Case 1 where σ is a nonshared event that can be executed by $\mathbf{A}_2$ at q_2), then

$$\delta(q, \sigma) = q' = (q_1, \delta_2(q_2, \sigma))$$

Namely $\mathbf{A}_1$ stays put while $\mathbf{A}_2$ makes a state transition by executing σ.

Case 3 : if $\sigma \in \Sigma_1 \cap \Sigma_2$, $\delta_1(q_1, \sigma)!$, and $\delta_2(q_2, \sigma)!$ (i.e. σ is a shared event that can be executed by $\mathbf{A}_1$ at q_1 and by $\mathbf{A}_2$ at q_2), then

$$\delta(q, \sigma) = q' = (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma))$$

That is, both $\mathbf{A}_1$ and $\mathbf{A}_2$ make a state transition by simultaneously executing σ.

Case 4 : in all other cases (i.e. either σ is a shared event but cannot be executed together, or σ is a nonshared event but cannot be executed by the owner automaton), $\delta(q, \sigma)$ is not defined.

- Initial state $q_0 = (q_{0,1}, q_{0,2})$.
- Marker state set $Q_m = Q_{m,1} \times Q_{m,2} = \{(q_{m,1}, q_{m,2}) \in Q \mid q_{m,1} \in Q_{m,1}, q_{m,2} \in Q_{m,2}\}$.

In Example 3.1, the synchronous product of PRINTER and USER is an automaton $\mathbf{A} = (Q, \Sigma, \delta, q_0, Q_m)$ whose elements are:

- $Q = \{0, 1, 2\} \times \{0, 1\}$
- $\Sigma = \{\text{start}, \text{manual\_stop}, \text{auto\_finish}, \text{breakdown}, \text{fix}\} \cup \{\text{start}, \text{manual\_stop}, \text{auto\_finish}, \text{take\_printouts}\}$
- $\delta : Q \times \Sigma \to Q$

- $q_0 = (0, 0)$
- $Q_m = \{0, 1\} \times \{0\}$

Thus the automaton displayed in Fig. 3.4 is the trimmed version of **A** (by removing the two non-reachable states (1,1) and (2,1)).

Special case: $\Sigma_1 = \Sigma_2$

In this case, all events are shared, so Case 1 and Case 2 of the state transition function δ above do not exist. Let's see an example.

Example 3.2 *Consider the two automata displayed in Fig. 3.5. Note that* $\Sigma_1 = \Sigma_2 = \{\alpha\}$. *The reader is invited to work out their synchronous product.*

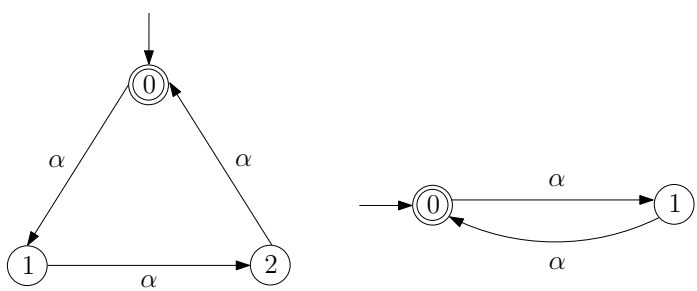


Figure 3.5: Synchronous product when all events are shared

In fact whenever $\Sigma_1 \neq \Sigma_2$ (which is generally the case), we can always 'add' those events only in Σ_2 but not in Σ_1 to $\mathbf{A}_1$ (and symmetrically 'add' those events only in Σ_1 but not in Σ_2 to $\mathbf{A}_2$),

such that the new $\mathbf{A}_1$ and $\mathbf{A}_2$ are both defined on the same event set $\Sigma_1 \cup \Sigma_2$.

But how to 'add' events in $\Sigma_2 \setminus \Sigma_1$ to $\mathbf{A}_1$ without affecting the rules of synchronous product? According to Case 2 of the definition, these events should be allowed anywhere in $\mathbf{A}_1$ and anytime occurrence. According to this, we add events in $\Sigma_2 \setminus \Sigma_1$ as *selfloops* at every state in $\mathbf{A}_1$. We denote the new automaton by

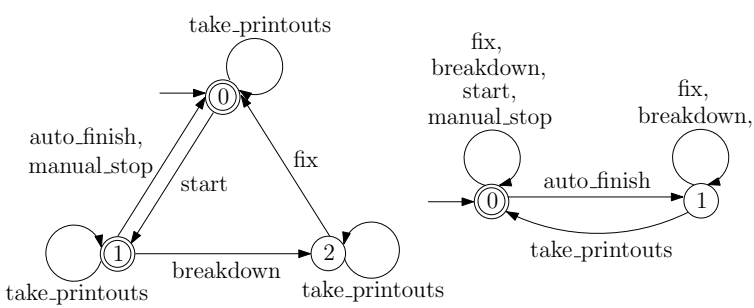
$$\mathbf{SA}_1 := \mathbf{selfloop}(\mathbf{A}_1, \Sigma_2 \setminus \Sigma_1)$$

Symmetrically, we selfloop all events in $\Sigma_1 \setminus \Sigma_2$ at every state in $\mathbf{A}_2$, and denote the outcome by

$$\mathbf{SA}_2 := \mathbf{selfloop}(\mathbf{A}_2, \Sigma_1 \setminus \Sigma_2)$$

Now both $\mathbf{SA}_1$ and $\mathbf{SA}_2$ are defined on the same event set $\Sigma \cup \Sigma_2$. Based on the defining rules of synchronous product, the following holds:

$$\mathbf{A}_1||\mathbf{A}_2 = \mathbf{SA}_1||\mathbf{SA}_2$$



selfloop(PRINTER, {*take_printouts*}) **selfloop**(USER, {*breakdown*, *fix*})

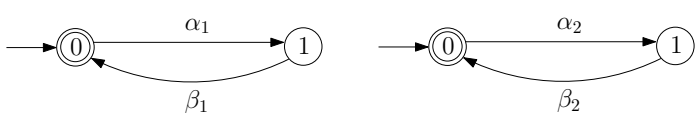
Figure 3.6: Selflooped automata of a printer and automaton of a user

In Example 3.1, the automata PRINTER and USER have different event sets. Selflooping the respective missing events, we obtain the selflooped automata as displayed in Fig. 3.6. The reader is invited to verify that the synchronous product of the two automata (after adding respective selfloops) is the same as the result in Fig. 3.4.

Special case: $\Sigma_1 \cap \Sigma_2 = \emptyset$

In this case, no event is shared, so Case 3 of the state transition function δ does not exist. Below is an example to illustrate this case.

Example 3.3 *Consider the two automata displayed in Fig. 3.7. Note that* Σ_1 *and* Σ_2 *have no event in common. The reader is invited to work out their synchronous product.*



$\mathbf{A}_1$ $\mathbf{A}_2$

Figure 3.7: Synchronous product when all events are nonshared

We end this section by noting that the synchronous product of three automata $\mathbf{A}_1$, $\mathbf{A}_2$, $\mathbf{A}_3$ can be done as $(\mathbf{A}_1||\mathbf{A}_2)||\mathbf{A}_3$, which is in fact the same as $\mathbf{A}_1||(\mathbf{A}_2||\mathbf{A}_3)$ (the reader is invited to check this fact). Namely the synchronous product is associative:

$$(\mathbf{A}_1||\mathbf{A}_2)||\mathbf{A}_3 = \mathbf{A}_1||(\mathbf{A}_2||\mathbf{A}_3)$$

Hence for convenience we write $\mathbf{A}_1||\mathbf{A}_2||\mathbf{A}_3$ without brackets. This extends to synchronous product of any finite number of automata.

3.3 Synchronously nonconflicting

The synchronous product of two nonblocking automata may be blocking, as illustrated in the example below.

Example 3.4 *Consider the two nonblocking automata* $\mathbf{A}_1$, $\mathbf{A}_2$ *displayed in Fig. 3.8. At the state pair (0,0), neither the shared event* β *nor* γ *can occur because they cannot be executed together. Hence the synchronous product* $\mathbf{A}_1||\mathbf{A}_2$ *has only the initial state (0,0), with the nonshared events* α_1, α_2 *defined (see Fig. 3.8). Note, however, that* $\mathbf{A}_1||\mathbf{A}_2$ *is blocking since the initial state (0,0) cannot reach any marker state (which does not exist in* $\mathbf{A}_1||\mathbf{A}_2$*).*

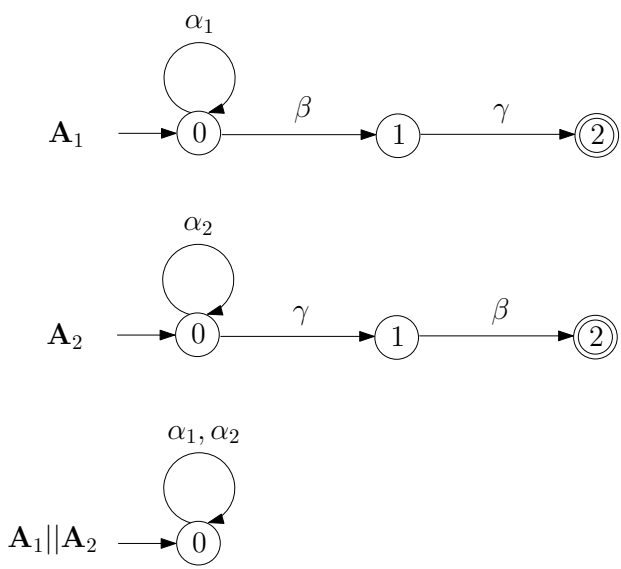


Figure 3.8: Synchronous product of two nonblocking automata fails to be nonblocking

In view of this example, we say that two (nonblocking) automata $\mathbf{A}_1$ and $\mathbf{A}_2$ are *synchronously nonconflicting* if their syn-

chronous product $\mathbf{A}_1||\mathbf{A}_2$ is nonblocking; otherwise we say that $\mathbf{A}_1$ and $\mathbf{A}_2$ are conflicting. Thus the two automata in Example 3.4 are conflicting, whereas the two in Example 3.1 are synchronously nonconflicting.

Note from Example 3.4 that synchronously conflicting occurs when two automata try to execute two (or more) shared events in different orders. Thus if two automata have no or only one event in common, then they are necessarily synchronously nonconflicting.

3.4 Behavior of synchronous product

In this section, we discuss what the behavior is the synchronous product $\mathbf{A} = \mathbf{A}_1||\mathbf{A}_2$ in terms of the behaviors of $\mathbf{A}_1$ and $\mathbf{A}_2$.

Let $\mathbf{A}_i := (Q_i, \Sigma_i, \delta_i, q_{0,i}, Q_{m,i})$, $i = 1, 2$, and their synchronous product $\mathbf{A} = (Q, \Sigma, \delta, q_0, Q_m)$. We start from the special case $\Sigma_1 = \Sigma_2$. Thus only Case 3 and Case 4 in the definition of synchronous product exist. By these two cases we have

$$\begin{aligned} L(\mathbf{A}) &= \{s \in \Sigma^* \mid \delta(q_0, s)!\} \\ &= \{s \in \Sigma^* \mid \delta_1(q_{0,1}, s)! \ \&\ \delta_2(q_{0,2}, s)!\} \\ &= \{s \in \Sigma^* \mid \delta_1(q_{0,1}, s)!\} \cap \{s \in \Sigma^* \mid \delta_2(q_{0,2}, s)!\} \\ &= L(\mathbf{A}_1) \cap L(\mathbf{A}_2) \end{aligned}$$

Moreover,

$$\begin{aligned} L_m(\mathbf{A}) &= \{s \in L(\mathbf{A}) \mid \delta(q_0, s) \in Q_m\} \\ &= \{s \in \Sigma^* \mid \delta_1(q_{0,1}, s) \in Q_{m,1} \ \&\ \delta_2(q_{0,2}, s) \in Q_{m,2}\} \\ &= \{s \in \Sigma^* \mid \delta_1(q_{0,1}, s) \in Q_{m,1}\} \cap \{s \in \Sigma^* \mid \delta_2(q_{0,2}, s) \in Q_{m,2}\} \\ &= L_m(\mathbf{A}_1) \cap L_m(\mathbf{A}_2) \end{aligned}$$

Consider again Example 3.2, where $\mathbf{A}_1$ and $\mathbf{A}_2$ are displayed in Fig. 3.5. Note that

$$L_m(\mathbf{A}_1) = \{\alpha^{3n} \mid n = 0, 1, \ldots\}$$
$$L_m(\mathbf{A}_2) = \{\alpha^{2n} \mid n = 0, 1, \ldots\}$$

Thus their synchronous product $\mathbf{A} = \mathbf{A}_1||\mathbf{A}_2$ satisfies

$$L_m(\mathbf{A}) = L_m(\mathbf{A}_1) \cap L_m(\mathbf{A_2}) = \{\alpha^{6n} \mid n = 0, 1, \ldots\}$$

Now consider the case $\Sigma_1 \neq \Sigma_2$. As in Section 3.2, we unify the event sets by selflooping the respectively missing events:

$$\mathbf{SA}_1 = \mathbf{selfloop}(\mathbf{A}_1, \Sigma_2 \setminus \Sigma_1)$$
$$\mathbf{SA}_2 = \mathbf{selfloop}(\mathbf{A}_2, \Sigma_1 \setminus \Sigma_2)$$

Now that the event sets of $\mathbf{SA}_1$ and $\mathbf{SA}_2$ are $\Sigma_1 \cup \Sigma_2$, the same conclusion as above applies. Namely

$$\begin{aligned} L(\mathbf{A}) &= L(\mathbf{A}_1||\mathbf{A}_2) \\ &= L(\mathbf{SA}_1||\mathbf{SA}_2) \\ &= L(\mathbf{SA}_1) \cap L(\mathbf{SA}_2) \end{aligned}$$

Similarly

$$L_m(\mathbf{A}) = L_m(\mathbf{SA}_1) \cap L_m(\mathbf{SA}_2)$$

Finally, $\mathbf{A}_1$ and $\mathbf{A}_2$ are synchronous nonconflicting if and only if $\mathbf{A}$ is nonblocking, i.e.

$$\overline{L_m(\mathbf{SA}_1) \cap L_m(\mathbf{SA}_2)} = L(\mathbf{SA}_1) \cap L(\mathbf{SA}_2)$$

The reader is invited to verify that in Example 3.4, the above condition does not hold.

3.5 PyTCT

Let's consider synchronous product of PRINTER and HUMAN. First create the automaton of the printer in Fig. 1.5.

```
statenum=3 #number of states
#states are sequentially labeled 0,1,...,statenum-1
#initial state is labeled 0

trans=[(0,'start',1),
       (1,'auto_finish',0),
       (1,'manual_stop',0),
       (1,'breakdown',2),
       (2,'fix',0)] # set of transitions
#each triple is (exit state, event label, entering state)

marker = [0,1] #set of marker states

pytct.create('PRINTER', statenum, trans, marker)
#create automaton PRINTER

pytct.display_automaton('PRINTER')
#plot PRINTER.DES
```

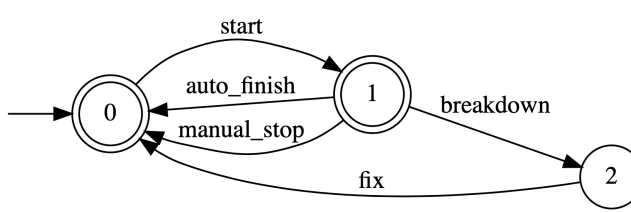


Figure 3.9: PyTCT plot of automaton PRINTER

Next, we create the automaton of the human in Fig. 3.10.

```
statenum=2 #number of states

trans=[(0,'start',0),
       (0,'auto_finish',1),
       (0,'manual_stop',0),
       (1,'take_printouts',0)] # set of transitions

marker = [0] #set of marker states

pytct.create('HUMAN', statenum, trans, marker)
#create automaton HUMAN

pytct.display_automaton('HUMAN')
#plot HUMAN.DES
```

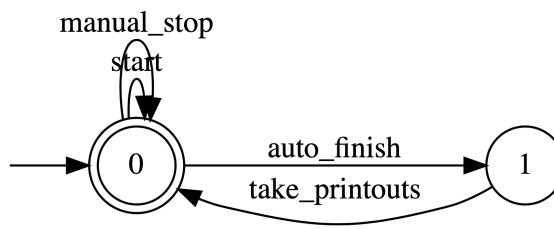


Figure 3.10: PyTCT plot of automaton HUMAN

With the two automata PRINTER and HUMAN created, the following function **sync** computes their synchronous product. The resulting automaton PH is displayed in Fig. 3.11.

```
pytct.sync('PH','PRINTER','HUMAN')
#synchronous product PH of PRINTER and HUMAN

pytct.display_automaton('PH')
#plot PH.DES
```

Note that PH in Fig. 3.11 is the same automaton as that in Fig. 3.4. However, the state numbers are recoded starting from the initial state 0 and continuing sequentially. To find out which of these states corresponds to which state pair, and display the resulting PH with state pairs, the following code may be used.

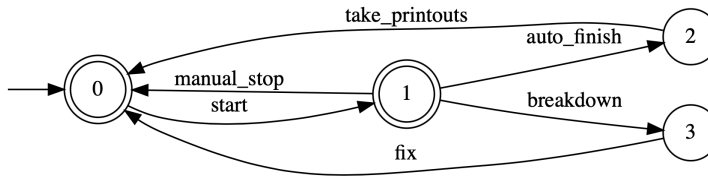


Figure 3.11: PyTCT sync of PRINTER and HUMAN

```
table = pytct.sync('PH','PRINTER','HUMAN',table=True,
    convert=True)
#variant of sync that returns table of state correspondence

print(table)
#print table of state correspondence

pytct.display_automaton('PH')
#plot PH.DES with state pairs
```

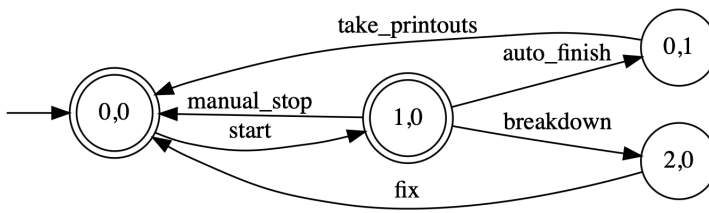


Figure 3.12: PyTCT sync of PRINTER and HUMAN, displaying state pairs

Now we unify the event sets of PRINTER and HUMAN by adding selfloops of the respectively distinct events. This is done by the **selfloop** function below. The results are displayed in Figs. 3.13 and 3.14

```
pytct.selfloop('PRINTER_SL','PRINTER',['take_printouts'])
#selfloop PRINTER with event take_printouts

pytct.display_automaton('PRINTER_SL')
#plot PRINTER_SL.DES
```

```
pytct.selfloop('HUMAN_SL','HUMAN',['breakdown,fix'])
#selfloop HUMAN with event breakdown, fix

pytct.display_automaton('HUMAN_SL')
#plot HUMAN_SL.DES
```

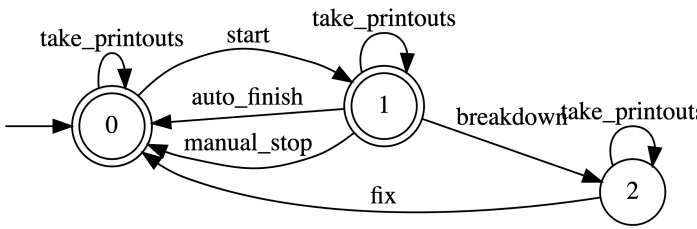


Figure 3.13: PyTCT selfloop for automaton PRINTER

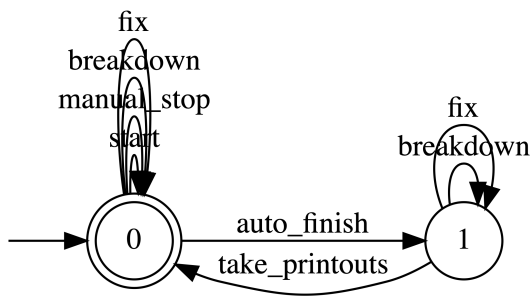


Figure 3.14: PyTCT selfloop for automaton HUMAN

The synchronous product of PRINTER_SL and HUMAN_SL is the same as the synchronous product of PRINTER and HUMAN – this is confirmed by the code below. In general, to verify if two automata are the same (modulo relabeling states and events), we use the function **isomorph**.

```
pytct.sync('PH_SL','PRINTER_SL','HUMAN_SL')
#synchronous product PH_SL of PRINTER_SL and HUMAN_SL

pytct.isomorph('PH_SL','PH')
```

```
#check if two automata are isomorphic
#(modulo relabeling of states and events)
```

Finally, to check if two automata are synchronously nonconflicting, first form their synchronous product and then check if the result is nonblocking. The following codes returns True, thereby confirming that PRINTER and HUMAN are synchronously nonconflicting.

```
pytct.is_nonblocking('PH')
#check if PH is nonblocking
```

Chapter 4

Close The Loop

4.1 Plant

From this chapter we start to talk about 'control'. It is customary to call the discrete-event system to be controlled the 'plant', which is modeled by an automaton[1]

$$\mathbf{P} = (X, \Sigma, \xi, x_0, X_m)$$

How the plant can be controlled depends on available technologies. In the (basic) supervisory control theory, the event set Σ is partitioned into a subset Σ_c of *controllable events* and a subset Σ_u of *uncontrollable events*:

$$\Sigma = \Sigma_c \cup \Sigma_u \quad (\Sigma_c \cap \Sigma_u = \emptyset)$$

Only the controllable events in Σ_c can be used as a means of control. Specifically, a controllable event may be enabled or disabled by an external controller, called *supervisor*, in order to achieve some desired behavior and/or to avoid some unwanted behavior. On the other hand, an uncontrollable event can never be prevented from occurring (due to lack of technology), and thus is treated as being always enabled.

[1] We consider the automaton nonblocking; if it is not, trim it. We also change notation for 'state' from Q, q_0, Q_m to X, x_0, X_m, in order to make appearance similar to standard control.

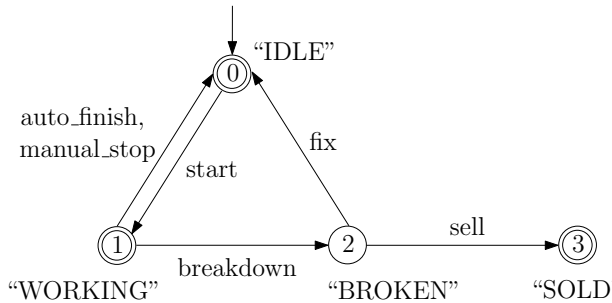


Figure 4.1: Printer with sold state

Example 4.1 *Let's consider the example of printer with a sold state (Fig. 4.1). In this model, the printer when broken down may be either fixed or sold. The sold state is a marker state. Commonsense suggests that the following events be controllable:*

$$\Sigma_c = \{\text{start}, \text{manual\_stop}, \text{fix}, \text{sell}\}$$

while the rest events be uncontrollable:

$$\Sigma_u = \{\text{auto\_finish}, \text{breakdown}\}$$

Example 4.2 *For the warehouse automation example, if the robot can move freely from one area to another, then all the events are controllable (as all movements can be enabled or disabled).*
It may also be a possible scenario where the robot's movement is controlled by traffic lights (say) installed between areas: The robot can move from one area to an adjacent one only when the corresponding light is green (Fig. 4.2). Thus in this scenario, an event is controllable if and only if

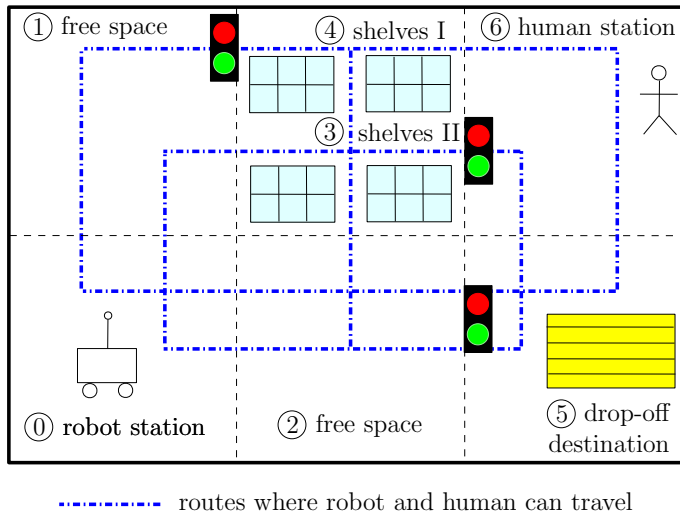


Figure 4.2: Warehouse automation with traffic lights for robot control

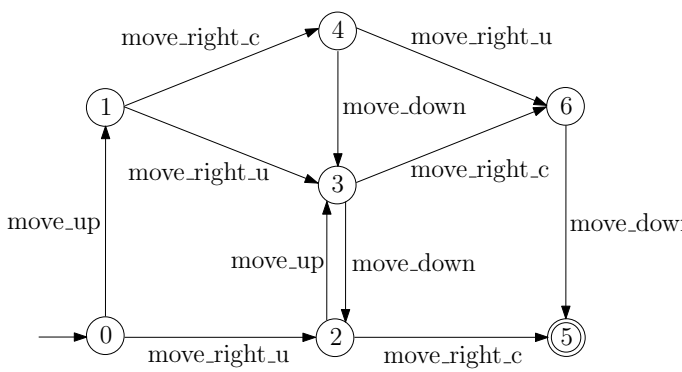


Figure 4.3: Automaton model of warehouse robot controlled by traffic lights

there is a traffic light for enabling/disabling the corresponding robot's movement. In Fig. 4.2, there are three traffic

lights installed on different segments on the routes where the robot (and the human) can travel. Also note that the storage shelf area is divided into two subareas to distinguish the two routes with traffic lights installed in different orders. Accordingly the robot movement can be modeled by the automaton displayed in Fig. 4.3. The only controllable events is

$$\Sigma_c = \{\text{move\_right\_c}\}$$

which appears in state 1, 2, and 3 corresponding the three traffic lights. Other events below with no corresponding traffic light are uncontrollable:

$$\Sigma_u = \{\text{move\_right\_u}, \text{move\_up}, \text{move\_down}\}$$

4.2 State-based specification

How to control a plant is dependent on what requirements are imposed on the behavior of the plant. Such control requirements are customarily called 'specification'.

In this section, we consider specifications on the states and state transitions of the plant. Such 'state-based specification' is also modeled by an automaton[2]

$$\mathbf{S} = (X_s, \Sigma_s, \xi_s, x_0, X_{s,m})$$

where $X_s \subseteq X$, $\Sigma_s \subseteq \Sigma$, $\xi_s \subseteq \xi$, and $X_{s,m} \subseteq X_m$. Namely state-based specification $\mathbf{S}$ is a *subautomaton* of the plant $\mathbf{P}$, obtained by removing some states and/or transitions which are deemed 'undesirable' or 'unsafe'. The initial state $\mathbf{S}$ is the same as that of $\mathbf{P}$,

[2]We consider the automaton nonblocking; if it is not, trim it.

which should not be removed since otherwise **S** would be empty.

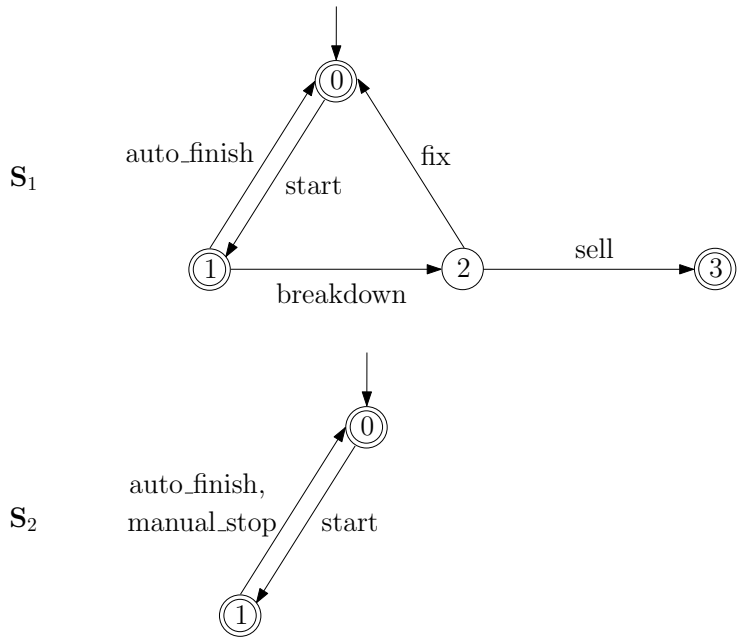


Figure 4.4: Examples of state-based specifications for printer with sold state

Let's consider two examples of state-based specifications on the printer in Fig. 4.1. The specification automata are displayed in Fig. 4.4.

- Specification $\mathbf{S}_1$ is the same as the printer except for removing a single state transition $(1, \text{manual\_stop}, 0)$. This means that it is desired that the printer is never stopped manually (which may cause some mechanical issues).
- Specification $\mathbf{S}_2$ is obtained from the printer by removing the state 2 ('BROKEN'); accordingly transi-

tions $(1, \text{breakdown}, 2)$, $(2, \text{fix}, 0)$, $(2, \text{sell}, 3)$, and state 3 ('SOLD') are all removed. This means that the printer is desired to never break down (!)

Commonsense suggests that while specification $\mathbf{S}_1$ is feasibly enforceable, $\mathbf{S}_2$ is unrealistic (at least there is no means of preventing the uncontrollable 'breakdown' event from occurring at state 1). Hence, although specifications can be imposed as one wishes, whether or not they can be enforced is case dependent.

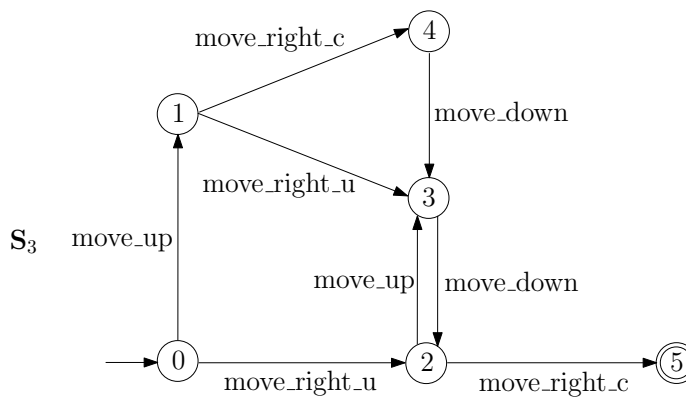


Figure 4.5: Example of state-based specification for warehouse robot with traffic lights

We also provide a state-based specification for the warehouse robot in Fig. 4.3. Since the states in the robot automaton carry information of physical locations, specifications can be imposed on the robot's locations. Consider the specification $\mathbf{S}_3$ displayed in Fig. 4.5. This automaton is obtained from the robot by removing the state 6 (human station), which is deemed unsafe and should be avoided.

Accordingly, all the transitions from or to state 6 are also removed.

4.3 State-feedback supervisory controller

Given a plant $\mathbf{P} = (X, \Sigma, \xi, x_0, X_m)$ with a subset of controllable events $\Sigma_c \subseteq \Sigma$, and a state-based specification $\mathbf{S} = (X_s, \Sigma_s, \xi_s, x_0, X_{s,m})$ (a subautomaton of $\mathbf{P}$), a *state-feedback supervisory controller* functions to enforce the specification on the behavior of the plant. Thus the controller is an entity that specifies which controllable events should be disabled at each state of the plant in order to satisfy the specification.

Let 2^{Σ_c} denote the set of all subsets of Σ_c (called *powerset*). We define *state-feedback* supervisory controller to be a function

$$C : X \to 2^{\Sigma_c}$$

which maps each state $x \in X$ to a subset $C(x)$ of controllable events to be *disabled* at x. Specifically which events should be disabled at x is dependent on the imposed specification $\mathbf{S}$. Note that by definition, the controller C cannot disable uncontrollable events.

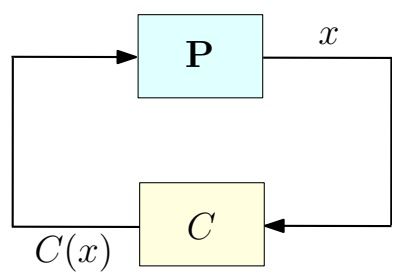


Figure 4.6: State feedback loop

This supervisory controller C is called 'state feedback' because, first it is a function on the state set X (domain), and second its evaluation $C(\cdot) \in 2^{\Sigma_c}$ (codomain) is fed back to influence the behavior of the plant. This feedback loop is displayed in Fig. 4.6, which operates as follows:

- The current state x of plant $\mathbf{P}$ is measured.
- Controller C evaluates x and issues a command of disabling a subset $C(x)$ of controllable events.
- The control command $C(x)$ is fed back to the plant $\mathbf{P}$, and accordingly $\mathbf{P}$ generates the next state x' by

$$x' = \xi(x, \sigma) \quad \text{if } \xi(x, \sigma)! \ \& \ \sigma \notin C(x). \tag{4.1}$$

  Namely x' is generated by an event σ that is defined at x in the plant and is not disabled by the controller.
- The process terminates if the condition in (4.1) is not satisfied, i.e. no more event can occur.

Starting from the initial state x_0, the above feedback loop generates a sequence of states:

$$x_0 \ x_1 \ x_2 \ldots$$

We say that the state sequence is generated by the *closed-loop system* $C/\mathbf{P}$, where the plant $\mathbf{P}$ is under the supervision of the controller C.

The closed-loop system $C/\mathbf{P}$ is itself an automaton (a subautomaton of $\mathbf{P}$), which can generate different state sequences from the initial state x_0, owing to generally multiple events that can occur at each state. Note that every state sequence generated by $C/\mathbf{P}$ can also be generated by the plant $\mathbf{P}$, but not vice versa.

The purpose of the state-feedback control is to ensure that the automaton $C/\mathbf{P}$ is nonblocking, and every state sequence gener-

ated by the closed-loop system $C/\mathbf{P}$ satisfies the specification $\mathbf{S}$, in the sense that the same state sequence can be generated by $\mathbf{S}$ from its initial state x_0. If $C/\mathbf{P}$ does satisfy $\mathbf{S}$, the closed-loop system can be represented as a subautomaton of $\mathbf{S}$.

Consider the printer with sold state in Fig. 4.1, and specification $\mathbf{S}_1$ in Fig. 4.4. Since $\mathbf{S}_1$ only requires removing the controllable event manual_stop at state 1, the following controller enforces $\mathbf{S}_1$:

$$C(x) = \begin{cases} \{\text{manual\_stop}\} & \text{if } x = 1 \\ \emptyset & \text{if } x = 0, 2, 3 \end{cases}$$

On the other hand, no controller exists to enforce specification $\mathbf{S}_2$ in Fig. 4.4, as controllers cannot disable uncontrollable events (in this case the uncontrollable event breakdown).

Consider the warehouse robot controlled by traffic lights in Fig. 4.3, and specification $\mathbf{S}_3$ in Fig. 4.5. This specification requires that the robot never enters the area of human station, which can be reached either from state 3 via move_right_c or from state 4 via move_right_u. While the event from state 3 is controllable and can be disabled (switching the corresponding traffic light to red), the event from state 4 is uncontrollable and thus cannot be prevented by the controller from occurring. This means that no controller exists to enforce $\mathbf{S}_3$.

You may have observed that the entrance to state 4 from state 1 is via the controllable event move_right_c, so if a controller can disable move_right_c at state 1 to prevent the robot from ever entering state 4. This in turn ensures that the robot never enters the area of human sta-

tion. Hence although there is no controller that can enforce $\mathbf{S}_3$, there does exist a controller that prevents the robot from entering an unsafe state. This observation is in fact the topic of *optimal supervisory control*, which will be introduced in Chapter 6 below.

4.4 Trajectory-feedback supervisory control

So far in this chapter we have focused on the perspective of *state*: state-based specification and state-feedback supervisory control. In this section we turn to an alternative, and in fact more general, setting which is in terms of system trajectories and behaviors.

4.4.1 Trajectory-based specificaton

Recall from Section **??** that the plant $\mathbf{P} = (X, \Sigma, \xi, x_0, X_m)$ (as a dynamic system) has two types of behaviors: Closed behavior

$$L(\mathbf{P}) := \{s \in \Sigma^* \mid \xi(x_0, s)!\}$$

is the set of all trajectories of $\mathbf{P}$ starting from the initial state x_0, and marked behavior

$$L_m(\mathbf{P}) := \{s \in L(\mathbf{P}) \mid \xi(x_0, s) \in X_m\}$$

the subset of trajectories of $\mathbf{P}$ that can hit a marker state in X_m.[3]

Since a control specification imposes requirements on the behavior of the plant, we consider a *trajectory-based specification*

$$S \subseteq L_m(\mathbf{P}).$$

[3]As in Section 4.1 we consider $\mathbf{P}$ a nonblocking automaton, so $\overline{L_m(\mathbf{P})} = L(\mathbf{P})$.

This means that among all trajectories in $L_m(\mathbf{P})$, those in S are desired ones (while those in $L_m(\mathbf{P}) \setminus S$ are unwanted).

How is such a trajectory-based specification S given? We consider that the desired behavior is described by an automaton $\mathbf{E}$, which is generally not a subautomaton of the plant $\mathbf{P}$. Thus $L_m(\mathbf{E}) \subseteq \Sigma^*$. Then let

$$S := L_m(\mathbf{E}) \cap L_m(\mathbf{P}) \subseteq L_m(\mathbf{P})$$

Given in this way, every trajectory in S is a requirement on the feasible behavior of the plant. Recall from Section 3.4 that S is indeed the behavior of the synchronous product of $\mathbf{E}$ and $\mathbf{P}$: namely letting $\mathbf{S} = \mathbf{P}||\mathbf{E}$, we have $S = L_m(\mathbf{S})$.

Note that any state-based specification $\mathbf{S}$ can always be expressed as a trajectory-based specification $S := L_m(\mathbf{S})$, but not the other way around.

Let's consider three examples of trajectory-based specifications on the printer in Fig. 4.1. These specifications are most conveniently described as languages of automata (Fig. 4.7).

- Specification $S_1 = L_m(\mathbf{E}_1) \cap L_m(\mathbf{P})$ means that if the printer breaks down once, then fix it; and if the printer breaks down for the second time, sell it. Note that this specification cannot be expressed as a state-based specification, as no subautomaton of the plant can describe this desired behavior.

- Specification $S_2 = L_m(\mathbf{E}_2) \cap L_m(\mathbf{P})$ means that no requirement is imposed on the printer, since all events are always enabled. This automaton is a generic specification that imposes no constraint on the behavior of the plant; in particular all controllable events are enabled at all times. This specification can be expressed

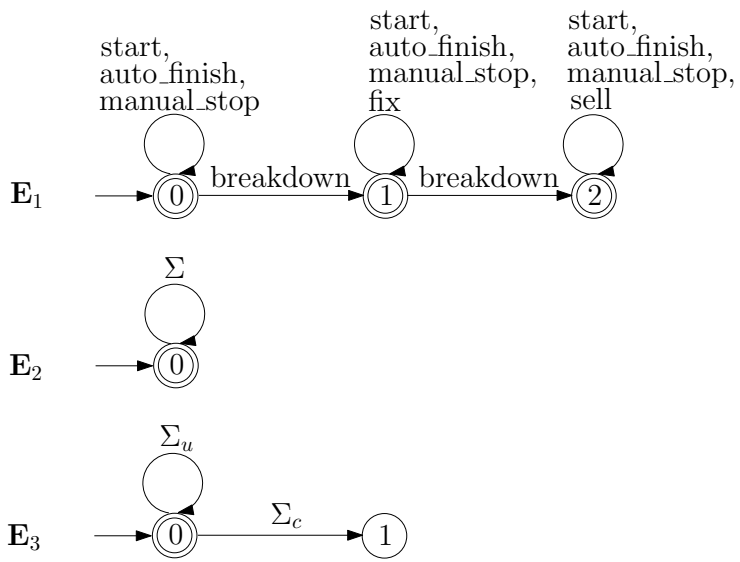


Figure 4.7: Examples of trajectory-based specifications for printer with sold state ($\Sigma = \{\text{start}, \text{manual\_stop}, \text{auto\_finish}, \text{breakdown}, \text{fix}, \text{sell}\}$, $\Sigma_c = \{\text{start}, \text{manual\_stop}, \text{fix}, \text{sell}\}$, $\Sigma_u = \{\text{auto\_finish}, \text{breakdown}\}$)

as a state-based automaton – simply the plant itself (no removal of any state or transition).

- Specification $S_3 = L_m(\mathbf{E}_3) \cap L_m(\mathbf{P})$ means that all controllable events are disabled, as execution of any controllable event leads to a blocking state. This automaton is also a generic specification that disables all controllable events in the plant at all times. This specification can be expressed as a state-based automaton – remove the transition $(0, \text{start}, 1)$ in the plant.

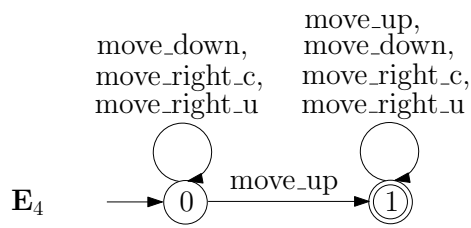


Figure 4.8: Example of trajectory-based specification for warehouse robot with traffic lights

We also provide a trajectory-based specification for the warehouse robot in Fig. 4.3. Note that the two generic specifications S_2 (enabling everything) and S_3 (disabling everything) in Fig. 4.7 apply here as well. Consider the specification $S_4 = L_m(\mathbf{E}_4) \cap L_m(\mathbf{P})$, where $\mathbf{E}_4$ is displayed in Fig. 4.8. Specification S_4 means that the event move_up should occur at least once, so that the robot can visit the shelve area (either state 3 or state 4); in other words, the robot should not go straight to the drop-off destination (state 5) without picking up anything from the shelf area. This specification cannot be expressed as a state-based specification.

4.4.2 Trajectory-feedback supervisory controller

To enforce $S \subseteq L_m(\mathbf{P})$, we define *trajectory-feedback* supervisory controller to be a function

$$C : L(\mathbf{P}) \to 2^{\Sigma_c}$$

which maps each trajectory $s \in L(\mathbf{P})$ (generated by the plant) to a subset $C(s)$ of controllable events to be *disabled* following s. Specifically which events should be disabled following s is depen-

dent on the imposed specification S. Note that by definition, the controller C cannot disable uncontrollable events. Note also that this trajectory-based controller is more general than state-based controller, inasmuch as with the trajectory information, we know not only *which* state the plant is at, but also *how* the plant arrives at this state.

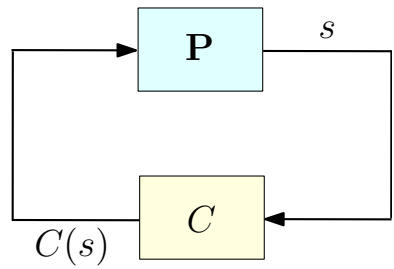


Figure 4.9: Trajectory feedback loop

This supervisory controller C is called 'trajectory feedback' because, first it is a function on the set of all possible trajectories $L(\mathbf{P})$ (domain), and second its evaluation $C(\cdot) \in 2^{\Sigma_c}$ (codomain) is fed back to influence the behavior of the plant. This feedback loop is displayed in Fig. 4.9, which operates as follows:

- The current trajectory s of plant $\mathbf{P}$ is measured.
- Controller C evaluates s and issues a command of disabling a subset $C(s)$ of controllable events.
- The control command $C(s)$ is fed back to the plant $\mathbf{P}$, and accordingly $\mathbf{P}$ generates the next event σ (so the next new trajectory $s\sigma$ is generated) if

$$s\sigma \in L(\mathbf{P}) \ \& \ \sigma \notin C(s). \tag{4.2}$$

  Namely σ is generated if it is defined after s in the plant and is not disabled by the controller.

- The process terminates if the condition in (4.2) is not satisfied, i.e. no more event can occur.

Starting from the initial empty string ϵ, the above feedback loop generates a trajectory:

$$\sigma_1 \ \sigma_2 \ \sigma_3 \ldots$$

We say that the trajectory is generated by the *closed-loop system* $C/\mathbf{P}$, where the plant $\mathbf{P}$ is under the supervision of the controller C. The closed-loop system $C/\mathbf{P}$ can generate different trajectories, owing to generally multiple events that can occur following each trajectory. To be precise, the set of all trajectories that can be generated by the closed-loop system $C/\mathbf{P}$ is $L(C/\mathbf{P})$ which satisfies $\epsilon \in L(C/\mathbf{P})$ and

$$s \in L(C/\mathbf{P}), s\sigma \in L(\mathbf{P}), \sigma \notin C(s) \Leftrightarrow s\sigma \in L(C/\mathbf{P})$$

In general,

$$\{\epsilon\} \subseteq L(C/\mathbf{P}) \subseteq L(\mathbf{P})$$

If $C(s) = \emptyset$ for every $s \in L(\mathbf{P})$ (i.e. no disabling action at all), then $L(C/\mathbf{P}) = L(\mathbf{P})$. We call $L(C/\mathbf{P})$ the *closed behavior* of the closed-loop system $C/\mathbf{P}$. The *marked behavior* of $C/\mathbf{P}$ is

$$L_m(C/\mathbf{P}) = L(C/\mathbf{P}) \cap S$$

Namely those trajectories generated by $C/\mathbf{P}$ that are also the desired ones as prescribed by the specification $S(\subseteq L_m(\mathbf{P}))$. We say that the closed-loop system $C/\mathbf{P}$ is nonblocking if

$$\overline{L_m(C/\mathbf{P})} = L(C/\mathbf{P})$$

The purpose of the trajectory-feedback control is to ensure that

the closed-loop system $C/\mathbf{P}$ is nonblocking, and every trajectory $s \in L_m(C/\mathbf{P})$ is in the specification S, i.e. $L_m(C/\mathbf{P}) \subseteq S$. The above implies that every trajectory $s \in L(C/\mathbf{P})$ generated by the closed-loop system is in the closure of the specification $\overline{S}$.

Consider the printer with sold state in Fig. 4.1, and specifications $S_i = L_m(\mathbf{E}_i) \cap L_m(\mathbf{P})$, where $i \in \{1, 2, 3\}$ and $\mathbf{E}_i$ are displayed in Fig. 4.7.

- S_1 requires fixing the printer if the printer breaks down once, whereas selling the printer for two breakdowns. The following trajectory-feedback controller enforces this specification:

$$C(s) = \begin{cases} \{\text{sell}\} & \text{if } s = s_1.\text{breakdown} \in L(\mathbf{P}) \\ \{\text{fix}\} & \text{if } s = s_2.\text{breakdown} \in L(\mathbf{P}) \\ \emptyset & \text{otherwise} \end{cases}$$

  Here string s_1 contains no breakdown event, while s_2 contains exactly one. Note that there is no state-feedback controller can enforce S_1, because the strings satisfy the first two conditions above hit the same state in the plant (state 2), but the required control actions are different. This shows that in order to enforce S_1, one does not only need to know *which* state the plant is at (in this case state 2), but also *how* the plant arrives at this state.

- S_2 imposes no requirement, so the supervisor needs to disable nothing:

$$(\forall s \in L(\mathbf{P}))\ C(s) = \emptyset$$

  This specification is conveniently used when the control purpose is solely to achieve nonblocking behavior

of the closed-loop system.

- S_3 requires disabling everything, which can be achieved by the following controller:

$$C(s) = \begin{cases} \{\text{start}\} & \text{if } s = \epsilon \\ \emptyset & \text{otherwise} \end{cases}$$

Finally consider the warehouse robot controlled by traffic lights in Fig. 4.3, and specification S_4 in Fig. 4.8. This specification requires that the robot executes move_up at least once (so as to enter the shelf area to perform an item pickup). This specification can be enforced by the following trajectory-feedback controller:

$$C(s) = \begin{cases} \{\text{move\_right\_c}\} & \text{if } s = \text{move\_right\_u} \\ \emptyset & \text{otherwise} \end{cases}$$

The reader is invited to convince himself/herself that this specification cannot be enforced by a state-feedback controller.

4.4.3 Automaton representation of closed-loop system

In the state-based case, the closed-loop system can be represented by a subautomaton of the specification. This is also the case for the trajectory-based closed-loop system, with the notable exception that the specification automaton **S** is now the synchronous product of the plant automaton **P** and the automaton **E**.

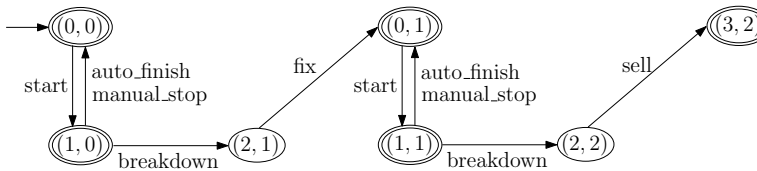


Figure 4.10: $\mathbf{S}_1 = \mathbf{P}||\mathbf{E}_1$

Consider again the printer with sold state in Fig. 4.1, and specification $S_1 = L_m(\mathbf{E}_1) \cap L_m(\mathbf{P})$ (where $\mathbf{E}_1$ is displayed in Fig. 4.7). Form the synchronous product $\mathbf{S}_1 = \mathbf{E}_1||\mathbf{P}$, whose structure is displayed in Fig. 4.10. Note that $S_1 = L_m(\mathbf{S}_1) \subseteq L_m(\mathbf{P})$, while S_1 is not a subautomaton of the plant $\mathbf{P}$. We can observe that the structure of $\mathbf{S}_1$ is sufficient to support the decision-making of the trajectory-feedback controller

$$C(s) = \begin{cases} \{\text{sell}\} & \text{if } s = s_1.\text{breakdown} \in L(\mathbf{P}) \\ \{\text{fix}\} & \text{if } s = s_2.\text{breakdown} \in L(\mathbf{P}) \\ \emptyset & \text{otherwise} \end{cases}$$

where string s_1 contains no breakdown event, while s_2 contains exactly one. Indeed, the state $(2,1)$ in $\mathbf{S}_1$ corresponds to $C(s) = \{\text{sell}\}$, and $(2,2)$ corresponds to $C(s) = \{\text{fix}\}$. Effectively, the synchronous product 'expands' the structure of the plant $\mathbf{P}$ so that the otherwise indistinguishable state 2 in $\mathbf{P}$ can be distinguished by $(2,1)$ and $(2,2)$ in $\mathbf{S}_1$ to reflect different numbers of occurrences of the breakdown event. For this reason, the closed-loop system can be represented based on $\mathbf{S}_1$ (in this particular case is $\mathbf{S}_1$ itself).

4.5 PyTCT

Let's create the plant automaton of PRINTER_SELL with controllable and uncontrollable events distinguished by red and green colors, respectively. The result is displayed in Fig. 4.11.

```
statenum=4 #number of states
#states are sequentially labeled 0,1,...,statenum-1
#initial state is labeled 0

trans=[(0,'start',1,'c'),
       (1,'auto_finish',0,'u'),
       (1,'manual_stop',0,'c'),
       (1,'breakdown',2,'u'),
       (2,'fixe',0,'c')
       (2,'sell',3,'c')] #set of transitions
#each triple is (exit state, event label, entering state)
#each event is either 'c' (controllable) or 'u' (
    uncontrollable)

marker = [0,1,3] #set of marker states

pytct.create('PRINTER_SELL', statenum, trans, marker)
#create automaton PRINTER_SELL

pytct.display_automaton('PRINTER_SELL',color=True)
#plot PRINTER_SELL.DES with color coding
```

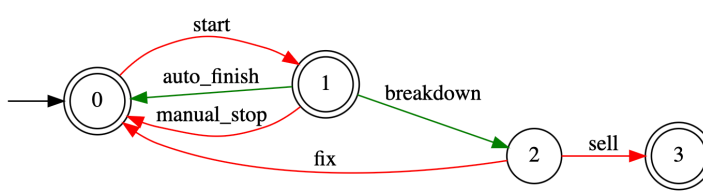


Figure 4.11: PyTCT create function with color coding of controllable events (red) and uncontrollable events (green)

An alternative way of creating a plant automaton with different color coding for controllable and uncontrollable events is to

use numbers (nonnegative integers) as labels for events. PyTCT recognizes (as default) that odd numbers are controllable events, whereas even numbers are uncontrollable events.[4] The following code shows how to use this alternative method; the result is in Fig. 4.12

```
statenum=4 #number of states
#states are sequentially labeled 0,1,...,statenum-1
#initial state is labeled 0

trans=[(0,11,1),
       (1,10,0),
       (1,13,0),
       (1,12,2),
       (2,15,0)] #set of transitions
#each triple is (exit state, event label, entering state)
#odd numbers for controllable events; even numbers for
    uncontrollable events

marker = [0,1,3] #set of marker states

pytct.create('PRINTER_SELL', statenum, trans, marker)
#create automaton PRINTER_SELL

pytct.display_automaton('PRINTER_SELL',color=True)
#plot PRINTER_SELL.DES with color coding
```

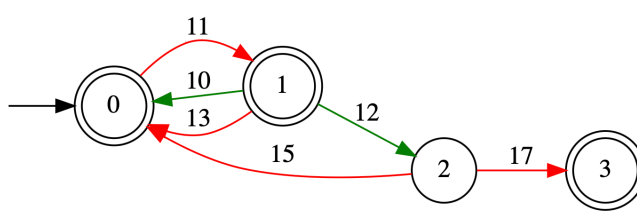


Figure 4.12: PyTCT create function with color coding of controllable events (odd numbers) and uncontrollable events (even numbers)

[4]Warning: PyTCT prohibits mixed use of numbers and strings (with single quotes) as event labels in the same script.

Next we create automaton models for specifications. We can directly use the **create** function. It may also be convenient to create a specification by removing certain states and/or transitions from the plant automaton. The following codes use the **subautomaton** function to create the two specification automata in Fig. 4.4. The results are displayed in Figs. 4.13 and 4.14, respectively.

```
pytct.subautomaton('S1','PRINTER_SELL',[],[(1,'manual_stop'
    ,0)])
#create subautomaton S1 by removing from PRINTER_SELL
# [state list] and [transition list]

pytct.display_automaton('S1',color=True)
#plot S1.DES with color coding
```

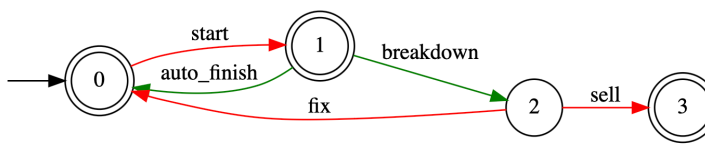


Figure 4.13: PyTCT subautomaton function by removing transitions

```
pytct.subautomaton('S2','PRINTER_SELL',[2],[])
#create subautomaton S2 by removing from PRINTER_SELL
# [state list] and [transition list]

pytct.display_automaton('S2',color=True)
#plot S2.DES with color coding
```

Note that the resulting automaton of the **subautomaton** function need not be trim (see Fig. 4.14). If a trim automaton is required, use the **trim** function.

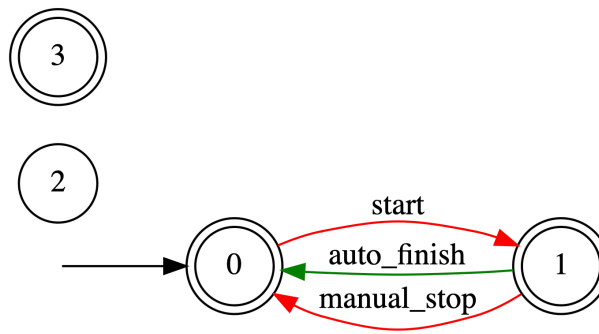


Figure 4.14: PyTCT subautomaton function by removing states (and the associated transitions)

Chapter 5

Analyze Controllability

5.1 State-based controllability

Given a plant to be controlled, and a specification that imposes requirements on the behavior of the plant, we ask:

> Does there exist a supervisory controller that enforces the specification on the plant?
>
> If so, can the controller be constructed?

In this chapter we address these questions by identifying a characterizing condition. We start with state-based feedback control in this section. Given a plant $\mathbf{P} = (X, \Sigma, \xi, x_0, X_m)$ with a subset of controllable events $\Sigma_c \subseteq \Sigma$, and a state-based specification $\mathbf{S} = (X_s, \Sigma_s, \xi_s, x_0, X_{s,m})$,[1] we aim to answer the question: *does there exist a state-feedback supervisory controller* $C : X \to 2^{\Sigma_c}$ *that enforces the specification on the behavior of the plant?*

In the preceding chapter, we have seen a few examples of positive and negative answers. The intuitive deciding factor is *whether or not the specification can be realized by disabling controllable events available at proper states of the plant.* Following this intuition, we can view the state set X_s as the 'safe region', while the complement $X \setminus X_s$ 'unsafe'. Thus the central question becomes:

[1] Both plant $\mathbf{P}$ and specification $\mathbf{S}$ are nonblocking; if they are not, trim them.

can we prevent the states in the safe region X_s from exiting to the unsafe $X \setminus X_s$ by disabling controllable events?

If the exiting is via a controllable event, we can prevent the exiting by disabling the controllable event. On the other hand, if the exiting is via an uncontrollable event, then we would have no means of keeping this exiting from happening. The above analysis leads to a central concept in the supervisory control theory: *controllability*.

We say that a state-based specification **S** is *controllable* with respect to a plant **P** if for every state $x \in X_s$ and every uncontrollable event $\sigma \in \Sigma_u$, if $\xi(x, \sigma_u)!$, then it must hold that $\xi_s(x, \sigma_u)!$ (i.e. $\xi_s(x, \sigma_u) \in X_s$). This means that every uncontrollable event allowed by the plant at every state in the safe region X_s must also be allowed by the specification (since there is no means of removing an uncontrollable event based on our assumed control technology). Writing this condition in one line is as follows:

$$(\forall x \in X_s, \forall \sigma \in \Sigma_u)\ \xi(x, \sigma_u)! \Rightarrow \xi_s(x, \sigma_u)! \tag{5.1}$$

If this controllability condition holds, then there is no danger caused by uncontrollable events and thus all we need to do is to properly disable controllable events. Under this condition, the following controller enforces specification **S** on the plant **P**:

$$C(x) = \begin{cases} \{\sigma \in \Sigma_c \mid \xi(x, \sigma)!\ \&\ \neg \xi_s(x, \sigma)!\} & \text{if } x \in X_s \\ \emptyset & \text{if } x \in X \setminus X_s \end{cases}$$

Namely C disables at each state in the safe region X_s all those controllable events that are allowed by the plant but not allowed by the specification. Note that C does not disable anything for any state that is already unsafe (these states are indeed irrelevant as the purpose of control is to keep states in the safe region X_s from going to the unsafe $X \setminus X_s$). The closed-loop system $C/\mathbf{P}$ is exactly **S**, namely it is nonblocking and the entire specification is

enforced.

On the flip side, if the controllability condition does not hold, then there exist a safe state $x \in X_s$ and an uncontrollable event $\sigma \in \Sigma_u$ such that although $\xi(x, \sigma)!$, we do not have $\xi_s(x, \sigma)!$. However, our assumed control technology cannot disable this uncontrollable event σ at x, and consequently no controller C can exist to enforce the specification on the plant.

The above analysis concludes that controllability is both sufficient and necessary for the existence of a state-based supervisory controller that realizes a given specification.

Example 5.1 *Let's reconsider the example of printer with a sold state (Fig. 5.1). In this model, the following events are controllable:*

$$\Sigma_c = \{\text{start}, \text{manual\_stop}, \text{fix}, \text{sell}\}$$

while the rest events are uncontrollable:

$$\Sigma_u = \{\text{auto\_finish}, \text{breakdown}\}$$

The two specifications $\mathbf{S}_1$ *and* $\mathbf{S}_2$ *are also displayed in Fig. 5.1.*

- *Specification* $\mathbf{S}_1$ *is controllable, since all uncontrollable transitions in the plant remain in* $\mathbf{S}_1$*. Indeed, as compared to the plant, only one controllable transition* $(1, \text{manual\_stop}, 0)$ *is removed. Thus the following controller enforces* $\mathbf{S}_1$*:*

$$C(x) = \begin{cases} \{\text{manual\_stop}\} & \text{if } x = 1 \\ \emptyset & \text{if } x = 0, 2, 3 \end{cases}$$

- *Specification* $\mathbf{S}_2$ *is not controllable: the uncontrollable*

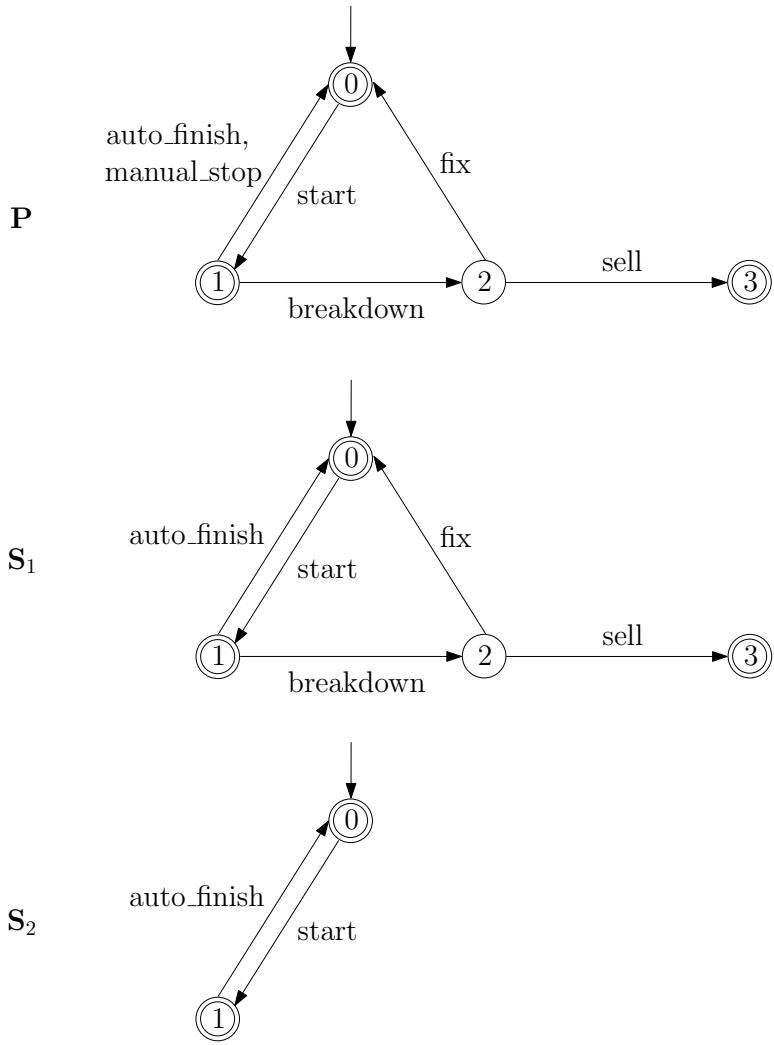


Figure 5.1: Printer with sold state and state-based specifications

event breakdown *allowed by the plant at state* 1 *in the safe region is not allowed by* $\mathbf{S}_2$. *Therefore there does*

not exist a controller that can enforce $\mathbf{S}_2$.

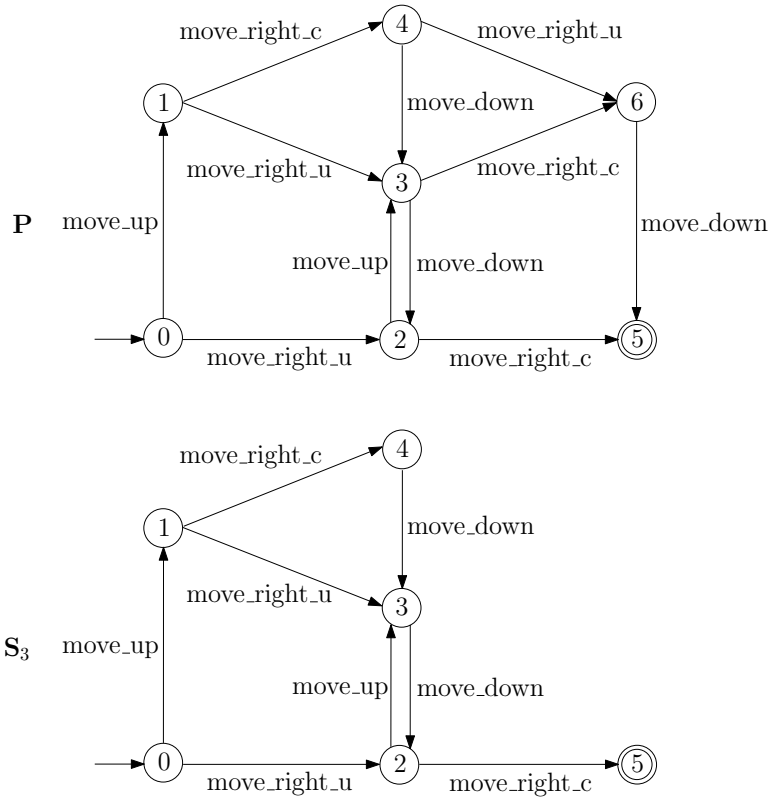


Figure 5.2: Warehouse robot controlled by traffic lights

Example 5.2 *Let's revisit the warehouse robot controlled by traffic light, where the plant* $\mathbf{P}$ *and specification* $\mathbf{S}_3$ *are displayed in Fig. 5.2. This specification* $\mathbf{S}_3$ *is not controllable, because the uncontrollable event* move_right_u *allowed by the plant at the state* 4 *in the safe region is not allowed by* $\mathbf{S}_3$. *Hence there is no controller that can enforce*

this specification.

5.2 Trajectory-based controllability

Consider a plant $\mathbf{P} = (X, \Sigma, \xi, x_0, X_m)$ with a subset of controllable events $\Sigma_c \subseteq \Sigma$, closed behavior $L(\mathbf{P})$, and marked behavior $L_m(\mathbf{P})$.[2] In this section we consider a trajectory-based specification $S \subseteq L_m(\mathbf{P})$. We view the closure $\overline{S}$ as the 'safe behavior', while $L(\mathbf{P}) \setminus \overline{S}$ 'unsafe', and aim to design a trajectory-based controller $C : L(\mathbf{P}) \to 2^{\Sigma_c}$ to prevent the trajectories in the safe behavior $\overline{S}$ from exiting to the unsafe $L(\mathbf{P}) \setminus \overline{S}$.

We say that a trajectory-based specification S is *controllable* with respect to a plant $\mathbf{P}$ if for every trajectory $s \in \overline{S}$ and every uncontrollable event $\sigma \in \Sigma_u$, if $s\sigma \in L(\mathbf{P})$ (i.e. allowed by the plant), then it must hold that $s\sigma \in \overline{S}$. This means that the safe behavior $\overline{S}$ must be 'invariant under uncontrollable flows': no uncontrollable event allowed by the plant can exit from the safe $\overline{S}$ (since otherwise there would be no control means of preventing the safe trajectories from entering the unsafe region). Writing this condition in one line is:

$$(\forall s \in \overline{S}, \forall \sigma \in \Sigma_u) s\sigma \in L(\mathbf{P}) \Rightarrow s\sigma \in \overline{S} \tag{5.2}$$

This controllability condition is both sufficient and necessary for the existence of a trajectory-based supervisory controller that enforces a given specification. To see sufficiency, consider that the controllability condition (5.2) holds; then there is no danger of existing $\overline{S}$ caused by uncontrollable events, and thus all we need to do is to properly disable controllable events. Under this condition,

[2] Consider a nonblocking $\mathbf{P}$ so that $\overline{L_m(\mathbf{P})} = L(\mathbf{P})$.

the following controller enforces specification S on the plant $\mathbf{P}$:

$$C(s) = \begin{cases} \{\sigma \in \Sigma_c \mid s\sigma \in L(\mathbf{P}) \ \&\ s\sigma \notin \overline{S}\} & \text{if } s \in \overline{S} \\ \emptyset & \text{if } s \in L(\mathbf{P}) \setminus \overline{S} \end{cases}$$

Namely C disables after each safe trajectory in $\overline{S}$ all those controllable events that are allowed by the plant but not allowed by the specification. Note that C does not disable anything for any trajectory that is already unsafe (these trajectories are indeed irrelevant as the purpose of control is to keep safe trajectories in $\overline{S}$ from going to the unsafe region). The closed-behavior $L(C/\mathbf{P})$ of the closed-loop system is exactly $\overline{S}$; hence

$$L_m(C/\mathbf{P}) = L(C/\mathbf{P}) \cap S = \overline{S} \cap S = S$$

This means that $\overline{L_m(C/\mathbf{P})} = L(C/\mathbf{P})$, i.e. the closed-loop system is nonblocking.

On the necessity, if the controllability condition does not hold, then there exist a safe trajectory $s \in \overline{S}$ and an uncontrollable event $\sigma \in \Sigma_u$ such that although $s\sigma \in L(\mathbf{P})$, we do not have $s\sigma \in \overline{S}$. However, our assumed control technology cannot disable this uncontrollable event σ after s, and consequently no trajectory-based controller C can exist to enforce the specification on the plant.

Let's use an example to illustrate the trajectory-based controllability. In this example, we also demonstrate that controllability can be verified based on the specification automaton $\mathbf{S}$ which is the synchronous product of the plant automaton $\mathbf{P}$ and the automaton $\mathbf{E}$.

Example 5.3 *Consider the printer with a sold state* $\mathbf{P}$ *and the trajectory-based specification* $S_1 = L_m(\mathbf{E}_1) \cap L_m(\mathbf{P})$ *(see Fig. 5.3). To verify if* S_1 *is controllable, form the syn-*

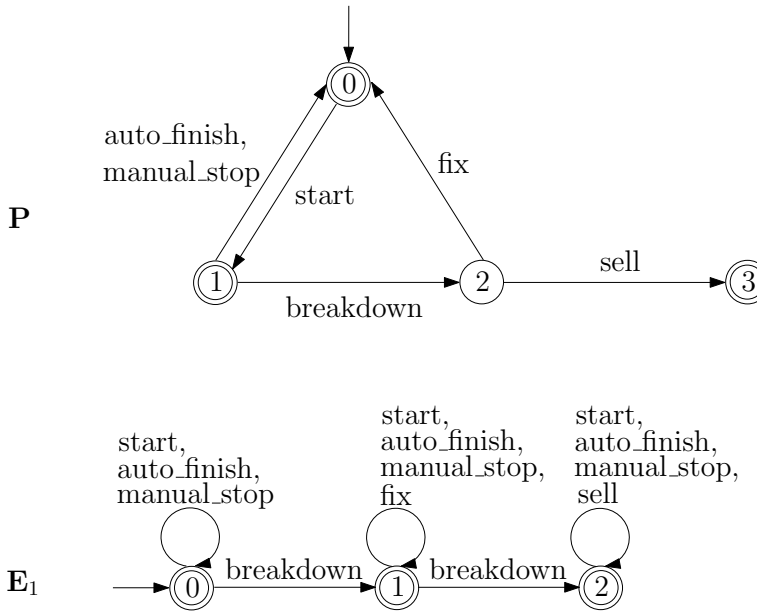


Figure 5.3: Printer with sold state and trajectory-based specification

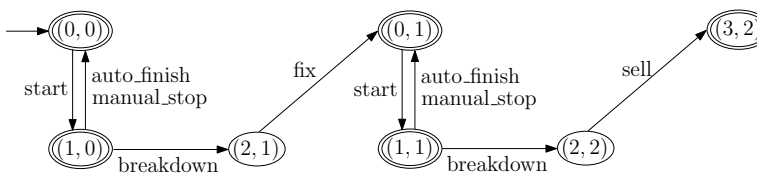


Figure 5.4: $\mathbf{S}_1 = \mathbf{P}||\mathbf{E}_1$

chronous product $\mathbf{S}_1 = \mathbf{P}||\mathbf{E}_1$*, whose structure is displayed in Fig. 5.4 (copied from Fig. 4.10 for convenience). The controllability condition (5.2) is checked by examining every state (pairs of plant and specification states) of* $\mathbf{S}_1$ *whether or not each uncontrollable event allowed by the first compo-*

nent (state of $\mathbf{P}$*) is also allowed by the second component (state of* $\mathbf{E}_1$*). In this example, it is inspected that this condition holds, which verifies that the specification* S_1 *is controllable. Indeed, the trajectory-feedback controller enforces* S_1*:*

$$C(s) = \begin{cases} \{\text{sell}\} & \text{if } s = s_1.\text{breakdown} \in L(\mathbf{P}) \\ \{\text{fix}\} & \text{if } s = s_2.\text{breakdown} \in L(\mathbf{P}) \\ \emptyset & \text{otherwise} \end{cases}$$

where string s_1 *contains no breakdown event, while* s_2 *contains exactly one.*

5.3 PyTCT

Controllability may be checked by the **is_controllable** function. Consider the plant PRINTER_SELL and specification S1 in Figs. 5.5 and 5.6 (created in the preceding chapter and reprinted here for convenience).

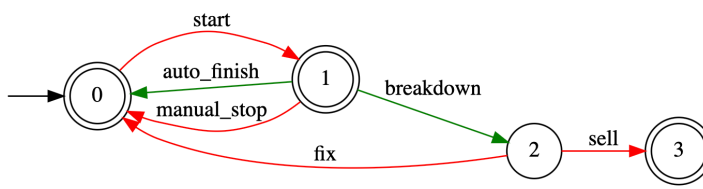


Figure 5.5: PRINTER_SELL plant

The following code shows how to use **is_controllable** to verify whether or not S1 is controllable with respect to PRINTER_SELL.

```
1 pytct.is_controllable('PRINTER_SELL','S1')
```

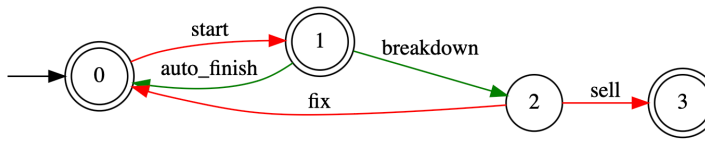


Figure 5.6: S1 specification

```
#check if S1 is controllable wrt. PRINTER_SELL
```

In this example, the above code returns "True", which confirms that **S1** is controllable with respect to **PRINTER_SELL**. Also consider specification **S2** (reprinted in Fig. 5.7).

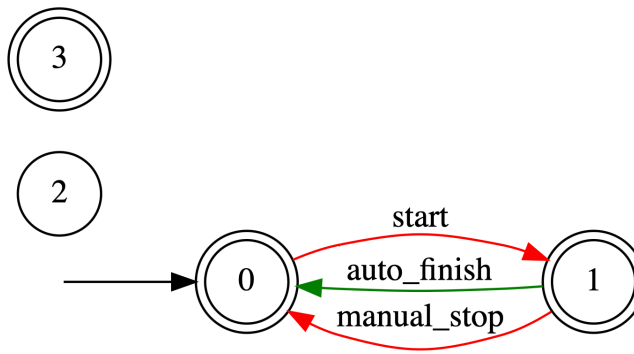


Figure 5.7: **S2** specification

The following code verifies that **S2** is *not* controllable with respect to **PRINTER_SELL**.

```
pytct.is_controllable('PRINTER_SELL','S2')
#check if S2 is controllable wrt. PRINTER_SELL
```

In this case, the following **uncontrollable_states** function may be used to compute the set of all uncontrollable states of the specification (i.e. those of **S2** that violate the controllability condition).

```
pytct.uncontrollable_states('PRINTER_SELL','S2')
```

```
#compute the set of uncontrollable states
```

The above code returns "[1]". Namely at state 1 of specification **S2**, the controllability condition does not hold. Indeed, on this state the uncontrollable event breakdown is allowed by plant **PRINTER_SELL** but is not allowed by **S2**.

The same two functions **is_controllable** and **uncontrollable_states** can be used for the trajectory-based controllability. Consider the trajectory-based specification $\mathbf{S_1}$ in Fig. 5.4. First let's create this specification, which is displayed as **SE1** in Fig. 5.8.

```
pytct.sync('SE1','PRINTER_SELL','E1')
#synchronous product

pytct.display_automaton('SE1',color=True)
#plot DES with color coding
```

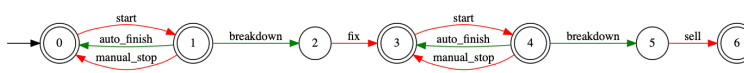


Figure 5.8: **SE1** specification

The following code verifies that **SE1** is controllable with respect to **PRINTER_SELL**.

```
pytct.is_controllable('PRINTER_SELL','SE1')
#check if SE1 is controllable wrt. PRINTER_SELL
```

Now remove from **SE1** the transition $(4, \text{breakdown}, 5)$. The resulting subautomaton **SE2** is displayed in Fig. 5.9.

```
pytct.subautomaton('SE2','SE1',[],[(4,'breakdown',5)])
#create subautomaton SE2 by removing from SE1
# [state list] and [transition list]

pytct.display_automaton('SE2',color=True)
#plot SE2.DES with color coding
```

Now confirm that **SE2** is *not* controllable with respect to the plant **PRINTER_SELL**. Namely the following code returns 'False'.

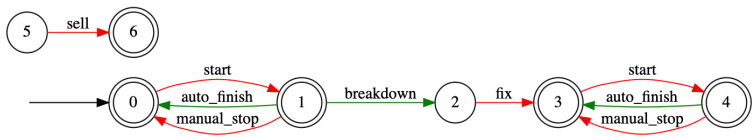


Figure 5.9: **SE2** specification

```
pytct.is_controllable('PRINTER_SELL','SE2')
#check if SE2 is controllable wrt. PRINTER_SELL
```

Finally the code below returns "[4]", which is the only uncontrollable state of specification **SE2**. Indeed, on this state the uncontrollable event breakdown is allowed by plant **PRINTER_SELL** but is not allowed by **SE2**.

```
pytct.uncontrollable_states('PRINTER_SELL','SE2')
#compute the set of uncontrollable states
```

Chapter 6

Design Optimal Controller

6.1 State-feedback optimal control

We have seen that controllability of a specification is the deciding property for the existence of a supervisory controller that enforces that specification. If controllability holds, we can construct a controller that realizes the entire specification.

Now what if controllability of a specification does not hold? Do we have to give up that specification all together? Well, possibly the imposed specification is overly strong to be realized fully on the plant, but part of it may still be realizable by a supervisory controller. Namely even when a specification turns out not controllable, a 'subspecification' may still be controllable and therefore realizable by a controller.

In this chapter, we address the problem of identifying and constructing controllable subspecifications of a given specification. Specifically, given a plant to be controlled and a specification that imposes requirements on the behavior of the plant, we ask:

> Does there exist a controllable subspecification of the given specification?
>
> If so, can the controllable subspecification be algorithmically constructed?

> Moreover if there exist multiple such controllable subspecificaitons, can we find the largest one?

We are interested in the *largest* controllable subspecification because such one would allow the maximal set of desired behaviors. Correspondingly, the controller that enforces the largest controllable subspecification is termed the 'optimal supervisory controller'.

We start our inquiry with state-feedback control in this section. Given a plant $\mathbf{P} = (X, \Sigma, \xi, x_0, X_m)$ with a subset of controllable events $\Sigma_c \subseteq \Sigma$, and a state-based specification $\mathbf{S} = (X_s, \Sigma_s, \xi_s, x_0, X_{s,m})$,[1] we aim to *find (if it exists) the optimal state-feedback supervisory controller* $C : X \to 2^{\Sigma_c}$ *that enforces the largest subspecification on the behavior of the plant.*

Let's build up intuition from the warehouse robot example.

Example 6.1 *Consider the example of warehouse robot controlled by traffic lights, where the plant* $\mathbf{P}$ *and specification* $\mathbf{S}$ *are reprinted in Fig. 6.1 for convenience. We have seen in Example 5.2 that this specification* $\mathbf{S}$ *is not controllable, since the uncontrollable event* move_right_u *allowed by the plant at the safe state* 4 *is not allowed by* $\mathbf{S}_3$*. Hence there is no controller that can enforce this specification in full.*

But can we find part of the specification that is realizable? In particular, it is only state 4 *that violates the controllability condition: Once the robot is allowed to move to state* 4*, there is no way of preventing it from entering the dangerous state* 6 *of the plant. In this sense state* 4 *should have been deemed unsafe. This observation suggests that we should act earlier so that the robot is prohibited from entering the problematic state* 4*. To this end, we further remove state* 4

[1] Both plant $\mathbf{P}$ and specification $\mathbf{S}$ are nonblocking; if they are not, trim them.

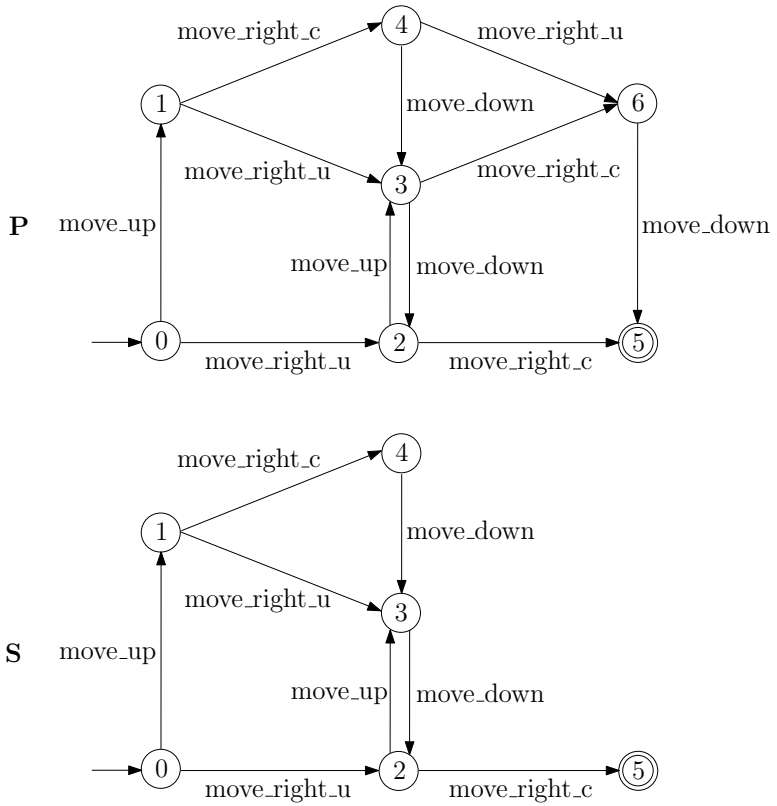


Figure 6.1: Warehouse robot controlled by traffic lights

to get the subspecification $\mathbf{S}^1$ *(displayed in Fig. 6.2). Now check that* $\mathbf{S}^1$ *is controllable. Hence* $\mathbf{S}^1$ *can be realized by a state feedback controller* $C : X \to 2^{\Sigma_c}$*. This controller is the following:*

$$C(x) = \begin{cases} \{\text{move\_right\_c}\} & \text{if } x = 1 \\ \emptyset & \text{otherwise} \end{cases}$$

The control decision is to disable move_right_c *at state* 1 *to*

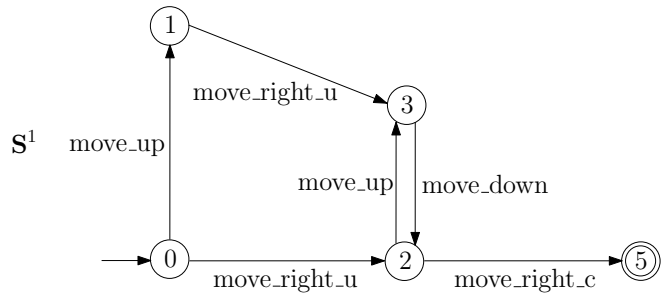


Figure 6.2: Largest controllable subspecification of warehouse robot

prevent the robot from ever entering state 4. *Note that this* C *is the optimal controller, because the subspecification* $\mathbf{S}^1$ *enforced by* C *is the largest possible (indeed only one state is removed which must be done to avoid uncontrollability). Also note that the closed-loop system* $C/\mathbf{P}$ *is represented by this automaton* $\mathbf{S}^1$ *and thus nonblocking.*

From the above example, we see that we need to identify and remove those states in the specification automaton $\mathbf{S}$ that violate the controllability condition. Is that all we need to do? There are in fact two important overlooks:

- Care for nonblocking
- Need of iteration

We illustrate these two points in the extended warehouse robot example below.

Example 6.2 *Consider a (slightly) extended warehouse robot (old state* 1 *is split into two new states* 1 *and* $1'$*), where the plant* $\mathbf{P}$ *and specification* $\mathbf{S}$ *are displayed in Fig. 6.3. Similar to Example 6.1, we remove state* 4 *that violates the*

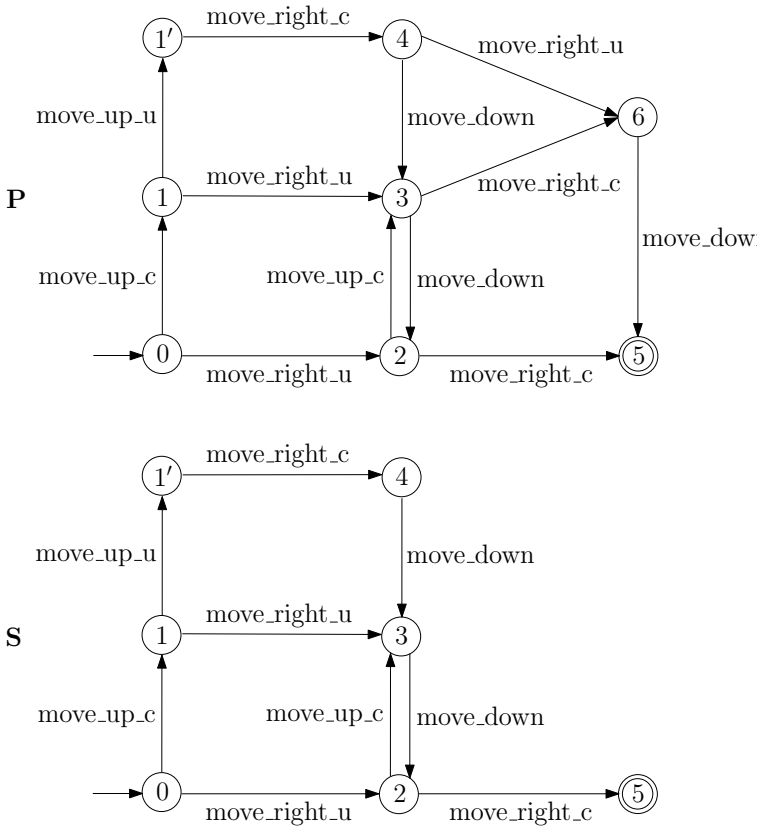


Figure 6.3: Extended warehouse robot controlled by traffic lights

controllability condition, and obtain the subautomaton $\mathbf{S}^1$ *(see Fig. 6.4). This* $\mathbf{S}^1$ *is, however, blocking since state* $1'$ *is not coreachable. If we stop here, although controllability holds, the closed-loop system would be blocking. This brings us to the first point: care for nonblocking. The resolution of this issue is straightforward: trim* $\mathbf{S}^1$ *(in this case removing the blocking state* $1'$*), and thereby we get* $\mathbf{S}^2$ *(Fig. 6.4). Are we done? Unfortunately, removing state* $1'$

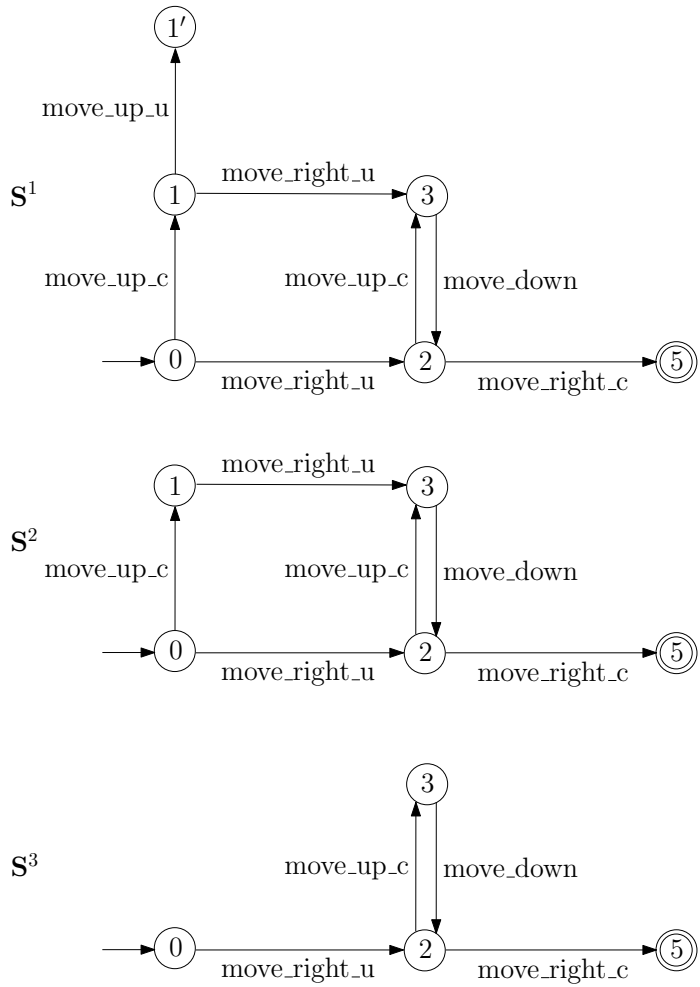


Figure 6.4: Iterated process to derive largest controllable subspecification of extended warehouse robot

makes $\mathbf{S}^2$ *not controllable again, because the uncontrollable event* move_up_u *allowed by the plant at the state* 1 *is not allowed by* $\mathbf{S}^2$*. This brings out the second point: need of*

iteration. Namely the two operations of ensuring controllability and ensuring nonblocking must be iterated until both properties hold. In this example, we continue to remove state 1 *and obtain* $\mathbf{S}^3$ *(Fig. 6.4). This* $\mathbf{S}^3$ *is checked to be both controllable and nonblocking, so a state-based feedback controller* $C : X \to 2^{\Sigma_c}$ *exists to enforce* $\mathbf{S}^3$*. This controller is the following:*

$$C(x) = \begin{cases} \{\text{move\_up\_c}\} & \text{if } x = 1 \\ \emptyset & \text{otherwise} \end{cases}$$

The control decision is to disable move_up_c *at the initial state* 0 *to prevent the robot from ever following the upper route. One may verify that this* C *is the optimal controller, because the subspecification* $\mathbf{S}^3$ *is the largest possible one that is both controllable and nonblocking. Accordingly the closed-loop system* $C/\mathbf{P}$ *represented by* $\mathbf{S}^3$ *is nonblocking.*

6.2 Optimal controller synthesis

From the preceding section, to synthesize a state-feedback supervisory controller for enforcing a subspecificaiton, there are two key operations: (1) identify and remove states that violate the controllability condition, and (2) trim the result to remove blocking states. These two operations should be applied alternatively as well as iteratively. In this section, we describe the synthesis *algorithm* (i.e. a series of steps of computation).

Consider a plant $\mathbf{P} = (X, \Sigma, \xi, x_0, X_m)$ with a subset of controllable events $\Sigma_c \subseteq \Sigma$, and a state-based specification $\mathbf{S} = (X_s, \Sigma_s, \xi_s, x_0, X_{s,m})$.[2] Let $U(\mathbf{S})$ denote the set of all states in $\mathbf{S}$ that do not

[2]Both plant $\mathbf{P}$ and specification $\mathbf{S}$ are nonblocking; if they are not, trim them.

satisfy the controllability condition (or simply the 'uncontrollable states'). Formally

$$U(\mathbf{S}) = \{x \in X_s \mid (\exists \sigma_u \in \Sigma_u)\ \xi(x, \sigma_u)! \ \&\ \neg \xi_s(x, \sigma_u)!\} \tag{6.1}$$

Note that to find $U(\mathbf{S})$, the plant $\mathbf{P}$ is needed (since the transition function ξ is involved).

In Example 6.1,

$$U(\mathbf{S}) = \{4\}, \quad U(\mathbf{S}^1) = \emptyset$$

In Example 6.2,

$$U(\mathbf{S}) = \{4\}, \quad U(\mathbf{S}^1) = \emptyset$$
$$U(\mathbf{S}^2) = \{1\}, \quad U(\mathbf{S}^3) = \emptyset$$

Now we describe the algorithm of synthesizing the optimal controller.

Algorithm 6.1 Optimal State-Feedback Controller Synthesis Algorithm

Input: Plant $\mathbf{P}$, specification $\mathbf{S}$
Output: Optimal controller $\mathbf{C}$
1: construct $U(\mathbf{S})$ as in (6.1)
2: **while** $U(\mathbf{S}) \neq \emptyset$ **do**
3: $\quad \mathbf{S}' \leftarrow$ remove $U(\mathbf{S})$ from $\mathbf{S}$
4: $\quad \mathbf{S} \leftarrow$ trim $\mathbf{S}'$
5: $\quad$ construct $U(\mathbf{S})$ as in (6.1)
6: **end while**
7: $\mathbf{C} \leftarrow \mathbf{S}$ and output $\mathbf{C}$

The two key operations are in lines 3 and 4 respectively. They are executed alternatively and iteratively until $\mathbf{S}$ is trim and $U(\mathbf{S})$ is empty. Thus the output $\mathbf{C}$ is controllable and nonblocking. Note that $\mathbf{C}$ may be empty, in case all the states are removed during the process. If $\mathbf{C}$ is empty, then we say that there does not exist

the optimal controller.

Why $\mathbf{C}$ is the largest controllable and nonblocking subspecification of the given $\mathbf{S}$? Suppose on the contrary that it is not. Thus there is another controllable and nonblocking subspecification $\mathbf{C}'$ (say), which has one more state x or one more transition (x, σ, x') than $\mathbf{C}$. Let's first consider the case of one more state x. Then x must have been removed during some iteration in either line 3 or line 4. This means respectively that x violates the controllability condition or is blocking. But this is a contradiction to the assumption that $\mathbf{C}'$ is both controllable and nonblocking. It remains to consider the case of one more transition (x, σ, x'). This transition must also have been removed during some iteration in either line 3 or line 4 when state x or state x' is removed. This means that either x or x' violates the controllability condition or is blocking. But this is again a contradiction to the assumption that $\mathbf{C}'$ is both controllable and nonblocking. Therefore, after all, $\mathbf{C}$ is the largest controllable and nonblocking subspecification of the given $\mathbf{S}$.

Finally, a state-feedback supervisory controller $C : X \to 2^{\Sigma_c}$ can be defined based on the controllable $\mathbf{C}$. The resulting closed-loop system $C/\mathbf{P}$ is nonblocking because $\mathbf{C}$ is so.

Applying Algorithm 6.1 to Example 6.1, we only need a single iteration:

line 1: $U(\mathbf{S}) = \{4\}$
line 2: $U(\mathbf{S}) \neq \emptyset$
line 3: $\mathbf{S}^1 \leftarrow$ remove $U(\mathbf{S})$ from $\mathbf{S}$
line 4: $\mathbf{S}^1 \leftarrow$ trim $\mathbf{S}^1$
line 5: $U(\mathbf{S}^1) = \emptyset$
line 6: **end while**
line 7: $\mathbf{C} \leftarrow \mathbf{S}^1$

The reader is invited to apply Algorithm 6.1 to Example 6.2, and convince him/herself that the output $\mathbf{C} = \mathbf{S}^3$.

6.3 Trajectory-feedback optimal control

Consider a plant $\mathbf{P} = (X, \Sigma, \xi, x_0, X_m)$ with a subset of controllable events $\Sigma_c \subseteq \Sigma$, closed behavior $L(\mathbf{P})$, and marked behavior $L_m(\mathbf{P})$.[3] In this section we consider a trajectory-based specification $S \subseteq L_m(\mathbf{P})$. Our goal is to find (if it exists) the optimal trajectory-feedback supervisory controller $C : L(\mathbf{P}) \to 2^{\Sigma_c}$ that enforces the largest subspecification of S on the behavior of the plant.

Let $T \subseteq S$, and recall that T is controllable (with respect to $\mathbf{P}$) if and only if (cf. (5.2))

$$(\forall s \in \overline{T}, \forall \sigma \in \Sigma_u) s\sigma \in L(\mathbf{P}) \Rightarrow s\sigma \in \overline{T}$$

Now consider the family of all controllable subspecifications of S:

$$\mathcal{C}(S) := \{T \subseteq S \mid T \text{ is controllable}\}$$

Let T_1, T_2 be two members of $\mathcal{C}(S)$. Then their union $T_1 \cup T_2$ is still a member of $\mathcal{C}(S)$. To see this, let $s \in \overline{T_1 \cup T_2}$, $\sigma \in \Sigma_u$, and $s\sigma \in L(\mathbf{P})$. Then $s \in \overline{T_1}$ or $s \in \overline{T_2}$. Consider the case $s \in \overline{T_1}$ (the other case is similar). Since $T_1 \in \mathcal{C}(S)$, it is controllable and $s\sigma \in \overline{T_1} \subseteq \overline{T_1 \cup T_2}$. This proves that $T_1 \cup T_2$ is controllable.

The above property of the family $\mathcal{C}(S)$ is called 'closure under set unions'. In fact, the union of any number of members in $\mathcal{C}(S)$ is still a member in $\mathcal{C}(S)$. This leads to the fact that the family $\mathcal{C}(S)$

[3] Consider a nonblocking $\mathbf{P}$ so that $\overline{L_m(\mathbf{P})} = L(\mathbf{P})$.

has a unique largest member, which is the union of all members:

$$\sup \mathcal{C}(S) := \bigcup_{T \in \mathcal{C}(S)} T$$

This largest member $\sup \mathcal{C}(S)$ is called the *supremal controllable subspecification* of S. Thus the trajectory-feedback controller $C : L(\mathbf{P}) \to 2^{\Sigma_c}$ that enforces $\sup \mathcal{C}(S)$ is the optimal one. That is, $L_m(C/\mathbf{P}) = \sup \mathcal{C}(S)$. Such an optimal controller exists if and only if $\sup \mathcal{C}(S) \neq \emptyset$. Note that if S is controllable itself, then $\sup \mathcal{C}(S) = S$.

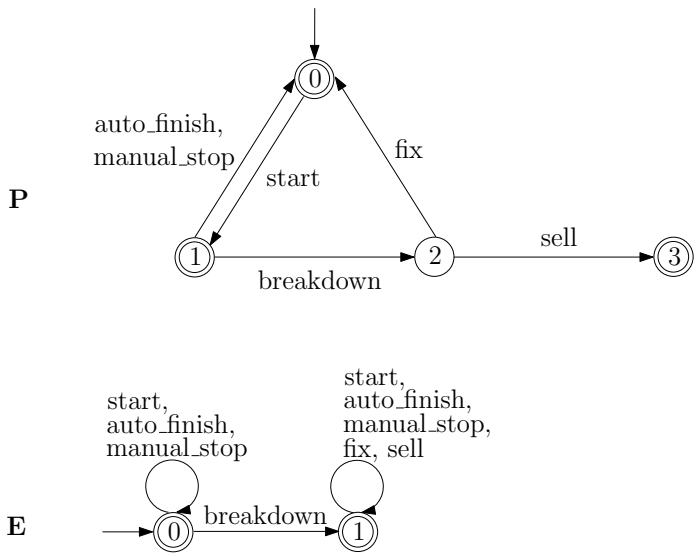


Figure 6.5: Printer with sold state and trajectory-based specification

Example 6.3 *Consider the printer with a sold state* **P** *(Fig. 6.5) and the trajectory-based specification* $S = L_m(\mathbf{P}) \cap L_m(\mathbf{E})$ *(Fig. 6.6). This specification requires that the printer be allowed to breakdown only once (!) The reader*

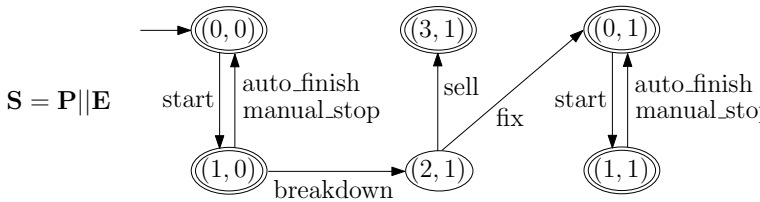
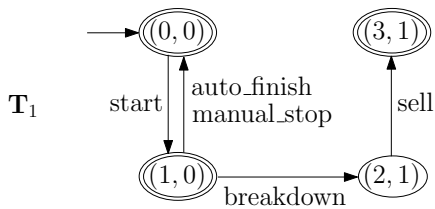


Figure 6.6: $\mathbf{S} = \mathbf{P}||\mathbf{E}$



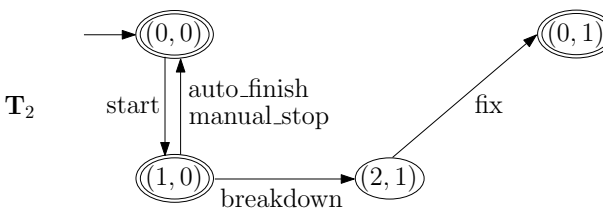


Figure 6.7: Controllable subspecifications

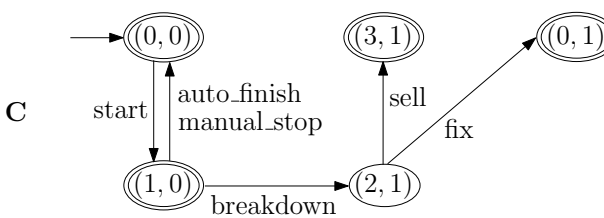


Figure 6.8: Supremal controllable subspecification

can verify that S is not controllable. Consider two subspec-

ifications of S (Fig. 6.7):

$$T_1 = L_m(\mathbf{T}_1), \quad T_2 = L_m(\mathbf{T}_2)$$

Verify that both T_1 and T_2 are controllable. Thus

$$T_1 \in \mathcal{C}(S), \quad T_2 \in \mathcal{C}(S)$$

The controller C_1 that enforces T_1 disables fix *after the first* breakdown *in order to avoid the second* breakdown*, whereas controller C_2 that enforces T_2 disables* sell *after the first* breakdown *and also disables* start *after the first* fix *in order to avoid the second* breakdown*.*
Their union is represented by the automaton $\mathbf{C}$ in Fig. 6.8. Namely $L_m(\mathbf{C}) = T_1 \cup T_2$. Verify that $T_1 \cup T_2$ is controllable, so $T_1 \cup T_2 \in \mathcal{C}(S)$. In fact for this example, we have

$$\mathcal{C}(S) = \{\emptyset, T_1, T_2, T_1 \cup T_2\}$$

and thus the supremal controllable subspecification is

$$\sup \mathcal{C}(S) = L_m(\mathbf{C})$$

The optimal controller C that enforces $\sup \mathcal{C}(S)$ only disables start *after the first* fix *to avoid the second* breakdown*. Thus C allows more behavior that either C_1 or C_2.*

Now that we know that the trajectory-feedback controller C enforcing the supremal controllable subspecification $\sup \mathcal{C}(S)$ is the optimal one, the question that remains is: How to compute $\sup \mathcal{C}(S)$? This is done by a similar algorithm in the preceding section (Algorithm 6.1), with the difference that the specification $\mathbf{S}$ is represented by the synchronous structure $\mathbf{P}||\mathbf{E}$. Next we describe this algorithm.

Consider a plant $\mathbf{P} = (X, \Sigma, \xi, x_0, X_m)$ with a subset of controllable events $\Sigma_c \subseteq \Sigma$, and another automaton $\mathbf{E} = (Y, \Sigma, \eta, y_0, Y_m)$ that represents a requirement.[4] Let the control specification $\mathbf{S}$ be the synchronous product of $\mathbf{P}$ and $\mathbf{E}$, i.e. $\mathbf{S} = \mathbf{P}||\mathbf{E}$.

Now define $U(\mathbf{S})$ for the set of all states in $\mathbf{S}$ that do not satisfy the trajectory-based controllability condition (or simply the 'uncontrollable states'). Such an uncontrollable state (pairs of plant and specification states) is one where there exists an uncontrollable event allowed by the first component (state of $\mathbf{P}$) but is not allowed by the second component (state of $\mathbf{E}$). Formally

$$U(\mathbf{S}) = \{(x, y) \in X \times Y \mid (\exists \sigma_u \in \Sigma_u)\ \xi(x, \sigma_u)! \ \&\ \neg\eta(y, \sigma_u)!\} \tag{6.2}$$

Algorithm 6.2 Optimal Trajectory-Feedback Controller Synthesis Algorithm

Input: Plant $\mathbf{P}$, requirement $\mathbf{E}$
Output: Optimal controller $\mathbf{C}$
1: $\mathbf{S} = \mathbf{P}||\mathbf{E}$
2: construct $U(\mathbf{S})$ as in (6.2)
3: **while** $U(\mathbf{S}) \neq \emptyset$ **do**
4: $\quad \mathbf{S}' \leftarrow$ remove $U(\mathbf{S})$ from $\mathbf{S}$
5: $\quad \mathbf{S} \leftarrow$ trim $\mathbf{S}'$
6: $\quad$ construct $U(\mathbf{S})$ as in (6.2)
7: **end while**
8: $\mathbf{C} \leftarrow \mathbf{S}$ and output $\mathbf{C}$

Like Algorithm 6.1, in Algorithm 6.2 the two key operations of removing uncontrollable states and blocking states are in lines 4 and 5 respectively. They are executed alternatively and iteratively until $\mathbf{S}$ is trim and $U(\mathbf{S})$ is empty. Different from Algorithm 6.1, here $\mathbf{S}$ is the synchronous product of plant $\mathbf{P}$ and requirement $\mathbf{E}$

[4] Both $\mathbf{P}$ and $\mathbf{E}$ are nonblocking; if they are not, trim them.

(line 1); thus

$$S = L_m(\mathbf{S}) = L_m(\mathbf{P}) \cap L_m(\mathbf{E}) \subseteq L_m(\mathbf{P})$$

When the algorithm terminates, the output $\mathbf{C}$ is both controllable and nonblocking. Note that $\mathbf{C}$ may be empty, in case all the states are removed during the process. If $\mathbf{C}$ is empty, then we say that there does not exist the optimal controller. Otherwise $\mathbf{C}$ is the largest controllable and nonblocking subspecification of $\mathbf{S}$ (for the same reason explained following Algorithm 6.1). Therefore $L_m(\mathbf{C}) = \sup \mathcal{C}(S)$. We illustrate Algorithm 6.2 by applying it to Example 6.3.

Consider again the printer $\mathbf{P}$ and requirement $\mathbf{E}$ in Fig. 6.5. Their synchronous product $\mathbf{S}$ is computed in line 1, which is displayed in Fig. 6.6. In line 2 we compute

$$U(S) = \{(1, 1)\}$$

This is because the uncontrollable event breakdown is allowed by the first component (plant $\mathbf{P}$) but is not allowed by the second component (requirement $\mathbf{E}$). Then the condition of line 3 is satisfied, and in line 4 the state $(1, 1)$ is removed from $\mathbf{S}$ together with all the three associated transitions. Since the resulting automaton is already trim, line 5 does not remove any state or transitions. Then in line 6, $U(\mathbf{S}) = \emptyset$; the while-loop ends in one iteration. The output $\mathbf{C}$ is exactly the one displayed in Fig. 6.8, which represents the supremal controllable subspecfication.

6.4 PyTCT

Let's consider the plant PRINTER_SELL in Fig. 6.9 (reprinted here for convenience), and create the requirement automaton **E** as follows. The automaton **E** is displayed in Fig. 6.10.

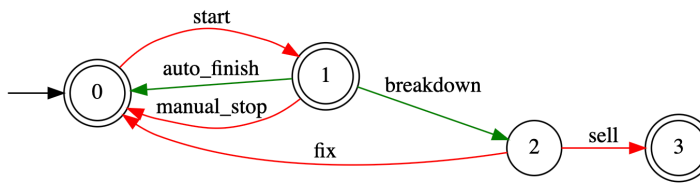


Figure 6.9: PRINTER_SELL plant

```
statenum=2 #number of states
#states are sequentially labeled 0,1,...,statenum
#initial state is labeled 0

trans=[(0,'start',0,'c'),
       (0,'auto_finish',0,'u'),
       (0,'manual_stop',0,'c'),
       (0,'breakdown',1,'u'),
       (1,'start',1,'c'),
       (1,'auto_finish',1,'u'),
       (1,'manual_stop',1,'c'),
       (1,'fix',1,'c'),
       (1,'sell',1,'c')] #set of transitions
#each triple is (exit state, event label, entering state)
#each event is either 'c' (controllable) or 'u' (
    uncontrollable)

marker = [0,1] #set of marker states

pytct.create('E', statenum, trans, marker)
#create automaton E

pytct.display_automaton('E',color=True)
#plot E.DES with color coding
```

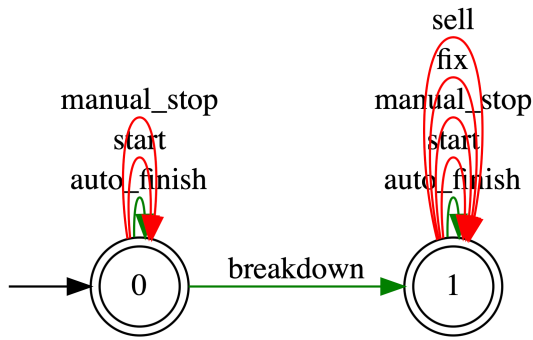


Figure 6.10: **E** requirement

Now let's compute the specification **S** by synchronous product. The result is displayed in Fig. 6.11.

```
pytct.sync('S','PRINTER_SELL','E')
#synchronous product

pytct.display_automaton('S',color=True)
#plot DES with color coding
```

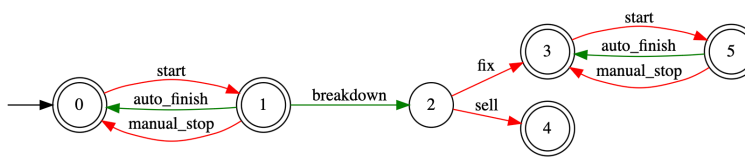


Figure 6.11: **S** specification

The function **supcon** in following code computes the optimal supervisor that achieves the supremal controllable subspecfication of **S**. The result is displayed in Fig. 6.12.

```
pytct.supcon('C','PRINTER_SELL','S')
#compute optimal supervisor

pytct.display_automaton('C', color=True) # display
    automaton
```

```
# red transition: controllable; green transition:
    uncontrollable
```

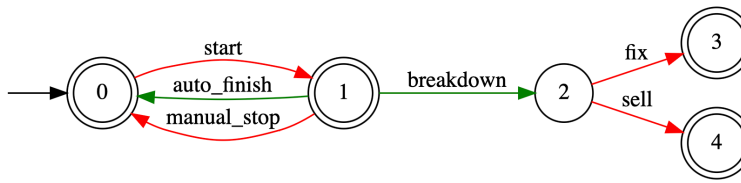


Figure 6.12: **C** optimal supervisor

Finally we can find out the control actions of the optimal supervisor **C** by the **conact** function.

```
table = pytct.conact('PRINTER_SELL','C')
#compute supervisor's control actions

print(table)
#print control actions
```

The above code returns that event start is disabled at state 3 of the optimal supervisor **C**.

Epilogue

I hope that I have included contents just enough to get you started with the supervisory control theory, and more importantly get you intrigued in wanting to know more. The following is a short list of suggested further reading (with my personal bias).

Further reading

• Appetizer: two encyclopedia articles

K. Cai and W.M. Wonham, "Supervisory control of discrete-event systems", Encyclopedia of Systems and Control, 2nd ed., Springer, 2020.

W.M. Wonham, "Supervisory control of discrete-event systems", Encyclopedia of Systems and Control, Springer, 2018.

• First course: two original papers that started the history

P.J. Ramadge and W.M. Wonham, "Supervisory control of a class of discrete event processes", SIAM Journal on Control and Optimization, vol. 25, no. 1, pp. 206–230, 1987.

W.M. Wonham and P.J. Ramadge, "On the supremal controllable sublanguage of a given language", SIAM Journal on Control and Optimization, vol. 25, no. 3, pp. 637–659, 1987.

• Second course: two meaty books

W.M. Wonham and K. Cai, "Supervisory Control of Discrete-Event Systems", Communications and Control Engineering, Springer, 2019.

C. Cassandras and S. Lafortune, "Introduction to Discrete Event Systems", 3rd ed., Springer, 2021.

• Dessert: two history papers

W.M. Wonham, K. Cai, and K. Rudie, “Supervisory control of discrete-event systems: a brief history”, Annual Reviews in Control, vol. 45, pp. 250–256, 2018.

W.M. Wonham, K. Cai, and K. Rudie, ”Supervisory control of discrete-event systems: a brief history – 1980-2015”, in Proceeding of the 20th IFAC World Congress, pp. 1827–1833, Toulouse, France, 2017.

Finally all the PyTCT functions introduced in the book are collected on the next page as a convenient reference.

PyTCT function collection (alphabetical order)

FUNCTION	WHAT IT DOES
blocking_states	find blocking states
conact	find control actions of supervisor
create	create automaton
display_automaton	display automaton
events	find events of automaton
init	initiate working folder
is_controllable	check controllability
is_coreachable	check coreachability
is_nonblocking	check nonblocking
is_reachable	check reachability
is_trim	check trim
isomorph	check isomorphism of automata
marker	find marker states of automaton
reachable_string	find string reaching given state
sample	sample string from automaton
selfloop	selfloop events on automaton
shortest_string	find shortest string
simulation_automaton	simulate string on automaton
statenum	find state number of automaton
subautomaton	find subautomaton
supcon	compute optimal supervisor
sync	compute synchronous product
trans	find transitions of automaton
trim	trim automaton
uncontrollable_states	find uncontrollable states

Index