A Manufacturing Process[†]

Tiziano Villa
Department of Computer Science
University of Verona, Italy

Matteo Zavatteri
Department of Computer Science
University of Verona, Italy

Part I

A manufacturing plant processes workpieces by employing two machines M_1, M_2 and a buffer B between them as shown in Figure 1. M_1 starts working a piece by taking it from an external infinite source. Once M_1 has started, either it finishes processing the piece or it breaks down. In the first case, M_1 places the piece into the buffer B, whose capacity is of one piece only, and returns ready to process other pieces. In the second case, M_1 discards the piece and waits for repair before starting processing pieces again. M_2 starts working a piece by taking it from B (if B contains any). After that, M_2 either finishes working the piece or breaks down by discarding it. In the first case, the piece is not put back into B as the process for such a piece is considered complete. In the second case, exactly as for M_1, M_2 waits for repair before processing other pieces. The model of the plant must take into consideration the following aspects:

1) Each machine has 3 states: *Idle* (initial and marked), *Active*, and *Down*.

2) Each machine has 4 transitions fired according to the following events:

   start : moving from *Idle* to *Active*.

   finish : moving from *Active* to *Idle*.

   break : moving from *Active* to *Down*.

   repair : moving from *Down* to *Idle*.

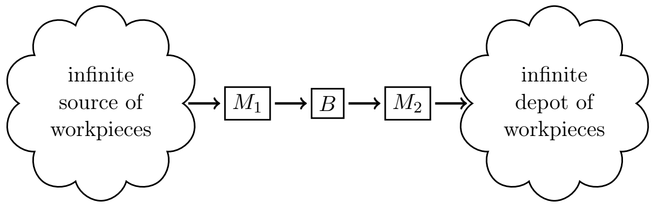


Figure 1: Manufacturing process setting.

3) The buffer B is either *Empty* (initial and marked) or *Full* and it is synchronized with some of the events of M_1 and M_2. When B is *Empty*, M_1 can finish processing a piece by placing it in the buffer. If this happens, then B becomes *Full* and M_1 cannot finish processing other pieces until B becomes *Empty* again. Likewise, when B is *Full*, M_2 can start processing a piece by taking it from B. If this happens, then B becomes *Empty* again. M_2 cannot start processing other pieces until B becomes *Full* again.

4) The events *finish* and *break* of each machine cannot be prevented.

Question 1. Build the plant automata M_1, M_2, B.

Answer 1. We introduce here (and use in the rest of the document) the following conventions to shorten the names of states and transitions. Let $i = 1, 2$. Then, I_i, A_i, and D_i shorten *Idle*, *Active*, and *Down* state names of machine i; s_i, f_i, b_i, and r_i shorten *start*, *finish*, *break*, and *repair* transition names of machine i, whereas E and F shorten *Empty* and *Full* state names of the buffer B. States are depicted as circles; the initial one is identified by an incoming "tail-less" arrow, whereas marked ones are double circled. Transitions are depicted as directed arrows la-

[*]This document is part of the reference material for *Discrete Event and Hybrid Systems* and *Systems Design Laboratory* offered by University of Verona in the Master Degree *Computer Engineering for Robotics and Smart Industry.*

[†]The original example discussed in Part I appears in the paper *P. J. Ramadge and W. M. Wonham. Supervisory Control of a Class of Discrete Event Processes. SIAM Journal on Control and Optimization 1987 25:1, 206-230.*

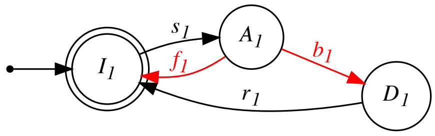


Figure 2: Plant automaton modeling M_1.

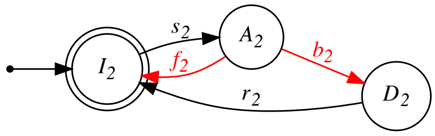


Figure 3: Plant automaton modeling M_2.

beled by the corresponding event. Controllable transitions are black; uncontrollable ones are red. Figure 2 and Figure 3 show the plant automata modeling the machines M_1 and M_2, respectively. Figure 4 shows the plant automaton modeling the buffer B.□

Question 2. Build the plant G as the parallel composition of M_1, M_2, and B. Is the result a shuffle?

Answer 2. Figure 5 shows $G := M_1 \| M_2 \| B$. A parallel composition is said a *shuffle* if the composed automata do not share events with each other. Thus, the result is not a shuffle since B shares f_1 with M_1 and s_2 with M_2. In other words, B is synchronized with M_1 and M_2 on those events. An example of a shuffle composition is $M_1 \| M_2$ (Figure 13). □

Question 3. Build the requirement automata to model the following requirements R_1, R_2, R_3, R_4.

R_1: M_1 can start working a piece only if B is empty.

R_2: M_2 can start working a piece only if B is full.

R_3: M_1 cannot start working a piece if M_2 is down.

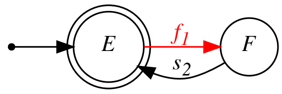


Figure 4: Plant automaton modeling B.

R_4: If M_1 and M_2 are down, then M_2 is repaired before M_1.

Answer 3. The general idea of modeling a requirement is to define an automaton R encoding it. Such an automaton will be taken as input along with the plant G by a synthesis algorithm to build a supervisor (if any) that restricts G according to R. When defining such an automaton we want to use as fewer states and events as possible. That is, we just want to encode no more than what we need (the synthesis operations coming next will do the rest of the job).

Since requirements are *constraints* that restrict the behavior of the plant G, the controllability of the events in R must always be consistent. That is, if an event is controllable (resp., uncontrollable in G), it is controllable (resp., uncontrollable) in R too. Such a restriction does not apply to states. Indeed, the states of R need not be related to the states of G and they generally have a specific interpretation that depends on the requirement itself. Moreover, we mark all states of R since in this example marking is property of the system and not of the requirements, and eventually we want to avoid undesired restrictions of G because of wrong marking in R.

To model R_1 we first need to track the state of the buffer B. This is easy if we start from a copy of the automaton B. To avoid confusion with the actual B, let us use in R_1 the state names B_E and B_F to track when the buffer is empty or full, respectively. To enforce that s_1 can be executed only if B is empty, we add a self loop transition labeled by s_1 on B_E. Figure 6 shows the resulting automaton R_1.

The requirement R_2 is already satisfied by G. Indeed, the automaton R_2 comes for free. Once again we need to track the state of B and thus we start with a copy of B (with the states renamed as be-

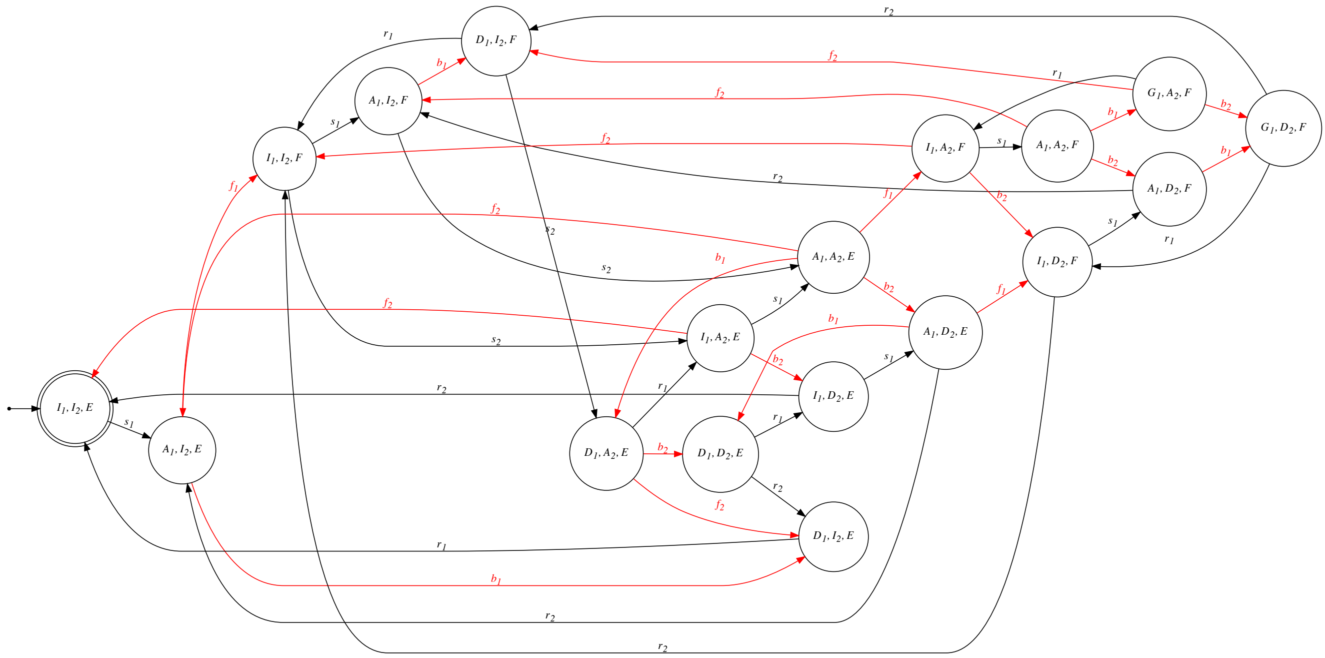


Figure 5: Whole plant automaton of the system $G := M_1 \| M_2 \| B$.

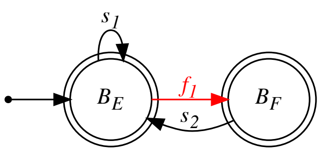


Figure 6: Requirement automaton modeling R_1.

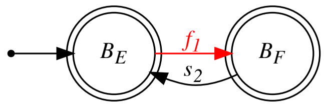


Figure 7: Requirement automaton modeling R_2.

fore). However, this time we do not need to add any transition since s_2 can only be executed from B_F. Therefore R_2 is basically identical to B with all states marked (Figure 7).

To model R_3 we first need to track the status of M_2. However, despite it would not be wrong, we do not need to start from a copy of M_2. We just need two states U_2 and D_2 to model "M_2 up" and "M_2 down", respectively. What about the events? Before thinking about M_2 we need to connect U_2 and D_2 in a way that this automaton correctly tracks when M_2 is up or down. Having a look at Figure 3, we know that there is only an event that takes M_2 down: the break event b_2. Therefore, we add a transition from U_2 to D_2 labeled by b_2. Likewise, there is a single event to take M_2 back up: the repair event r_2. Therefore, we add a transition from D_2 to U_2 labeled by r_2. Now that we have a tracking of M_2, we can enforce the requirement by adding a self loop transition on U_2 labeled by the event s_1. This way, if M_2 is down, M_1 cannot start. Figure 8 shows the requirement automaton R_3. To sum up, U_2 represent both I_2 and A_2 of automaton M_2. This is possible because all other events of M_2 are not relevant to build R_3.

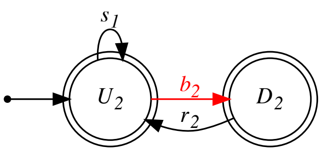


Figure 8: Requirement automaton modeling R_3.

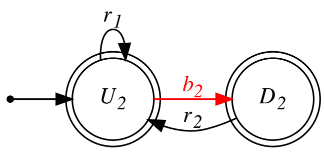


Figure 9: Requirement automaton modeling R_4.

To model R_4 we first need to track when both machines are down. However, instead of tracking all possible combinations of ups and downs of M_1 and M_2 (4 states), we start again from an automaton that only tracks the ups and downs of M_2 as we did for R_3 (2 states). Now, if we add a self loop labeled by r_1 on U_2 we have modeled the requirement. Figure 9 shows R_4. Note that M_1 can break down in both U_2 and D_2 since b_1 is not part of R_4. Therefore the possible 4 cases are the following:

1) If M_1 and M_2 are both up, then R_4 does not block them in any way.

2) If M_1 is down and M_2 is up, then R_4 is in U_2 and M_1 can be repaired because of the self loop r_1 on U_2.

3) If M_1 is up and M_2 is down, then R_4 is in D_2 and M_2 can be repaired because of the transition from D_2 to U_2 labeled by r_2.

4) If M_1 and M_2 are down, then R_4 is in D_2 and M_1 cannot be repaired until M_2 is not repaired. This is because r_1 can only be executed from U_2 and to get to U_2 we need to execute r_2 from D_2 which is equivalent to saying that when both are down M_2 is repaired first.

As a final note, considering the requirement R_4 in isolation, we point out that when both machines are down and M_2 is repaired, there is no obligation of repairing M_1 after that. Indeed, M_2 might break another time gaining repair priority again. The following sequence of events proves that: $i_1, f_1, i_2, i_1, f_1, i_1, b_1, b_2, r_2, i_2, b_2, r_2$.

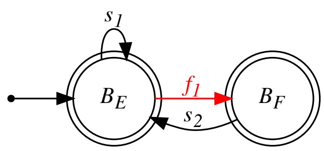


Figure 10: Requirement automaton $R_{1,2} := R_1 \| R_2$.

To conclude this answer we want to draw your attention to another important aspect. We built 4 automata: R_1, R_2, R_3, and R_4. However, all we did up to here can be further simplified in just 2 automata:

- $R_{1,2}$ merging R_1 and R_2 (Figure 10)
- $R_{3,4}$ merging R_3 and R_4 (Figure 11)

Therefore, using (R_1, R_2, R_3, R_4) or $(R_{1,2}, R_{3,4})$ is the same. But also taking $(R_1 \| R_2 \| R_3, R_4)$ or $(R_1, R_2 \| R_3 \| R_4)$ ... is the same. because when synthesizing a supervisor we will consider the parallel of *all* specifications. The reason why we do not provide a single requirement automaton $R_1 \| R_2 \| R_3 \| R_4$ is because it is an operation that we prefer to leave to the control synthesis algorithm. Our job is to construct a set of specifications which are "human-readable". Sometimes – as in this case – it might be convenient to merge some specifications (e.g., $R_{1,2}$ and $R_{3,4}$). We do this if it makes sense. It is up to us to choose the form and the number of the requirement automata. □

Question 4. Build a non-blocking supervisor S that restricts the plant G according to the requirements

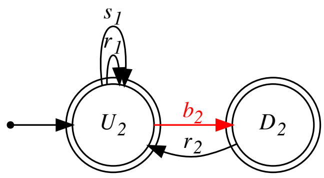


Figure 11: Requirement automaton $R_{3,4} := R_3 \| R_4$.

R_1, R_2, R_3, R_4. If S exists, then explain how S is deployed to restrict G and describe its control strategy.

Answer 4. In order to build a non-blocking supervisor S we proceed according to the following steps.

1) We set $S := G \| R_1 \| R_2 \| R_3 \| R_4 = G \| R_{1,2} \| R_{2,3}$. This is the initial, tentative, supervisor.

2) While there exists a state in S which *is* (or *has become*) *uncontrollable* or *non-co-accessible*, we remove that state from S. Note that a state can satisfy all these two properties simultaneously. Thus, there is no mutual-exclusiveness between these properties.

3) If (2) removed the initial state of (the current) S, then we are done: no supervisor exists. Otherwise, we set $S := \mathit{trim}(S)$ and we get the supervisor we are looking for[1].

Let us start with step (1). Figure 12 shows the parallel composition $S := G \| R_{1,2} \| R_{3,4}$, which is not a shuffle since we saw before that G was not a shuffle (and any composition with something not a shuffle remains not a shuffle).

We now move to step (2). A state of S has the form $(m_1, m_2, b, r_{1,2}, r_{3,4})$ thus belongs to the cross-product of the states of $M_1, M_2, B, R_{1,2}, R_{3,4}$. That is, $\mathit{States}(S) \subseteq \mathit{States}(M_1) \times \mathit{States}(M_2) \times \mathit{States}(B) \times \mathit{States}(R_{1,2}) \times \mathit{States}(R_{3,4})$ (in general, this subset relation is strict). Let $(m_1, m_2, b, r_{1,2}, r_{3,4})$ be any state of S. Then, we say that such a state is:

- *uncontrollable* if there exists an uncontrollable event e which is not executable in $(m_1, m_2, b, r_{1,2}, r_{3,4})$ but it is executable in the corresponding state (m_1, m_2, b) of G. In such a case, S would be trying to disable an uncontrollable event (a forbidden operation).

- *non-co-accessible* if there is no path from $(m_1, m_2, b, r_{1,2}, r_{3,4})$ to a marked state in S.

Also, we recall that any marked state is co-accessible since there exists a zero-length path to itself.

In the first iteration of the while loop, we discover that there are no uncontrollable states in the current S. Therefore, we proceed to look for the non-co-accessible states. It is clear that each state of S is co-accessible since the graph in Figure 12 is a strongly connected component (SCC)[2]. Therefore, since we do not remove any state of S we can exit (2).

Now we move to step (3) and take $S := \mathit{trim}(S)$ which again does not modify the current S, so S is the supervisor that we are looking for. Therefore, there exists a non-blocking supervisor S that restricts G according to the requirements R_1, R_2, R_3, R_4.

S is deployed in the system by means of a parallel composition with G. Since S was built from the parallel composition of G and the requirements, then S can track the events that G executes, and for each new event executed by G, can restrict the next set of controllable events that G is allowed to execute. As a result, S can prevent G from reaching some states that could be reachable without control. To sum up, the expected controlled behavior of G under the supervision of S (in symbols, S/G) is $S/G := S \| G$.

We are left to describe the strategy of S. First of all notice that the states that G reaches under the control of S are the states $\{(m_1, m_2, b) \mid$

[1] We recall that, given an automaton A, $\mathit{trim}(A)$ is A without the states that are non-accessible or non-co-accessible (and the corresponding transitions involving them).

[2] In a directed graph a strongly connected component (SCC) is a set of nodes such that for every two nodes in the set there exists a path from the first to the second (and of course there always exists a zero-length path from each node to itself).

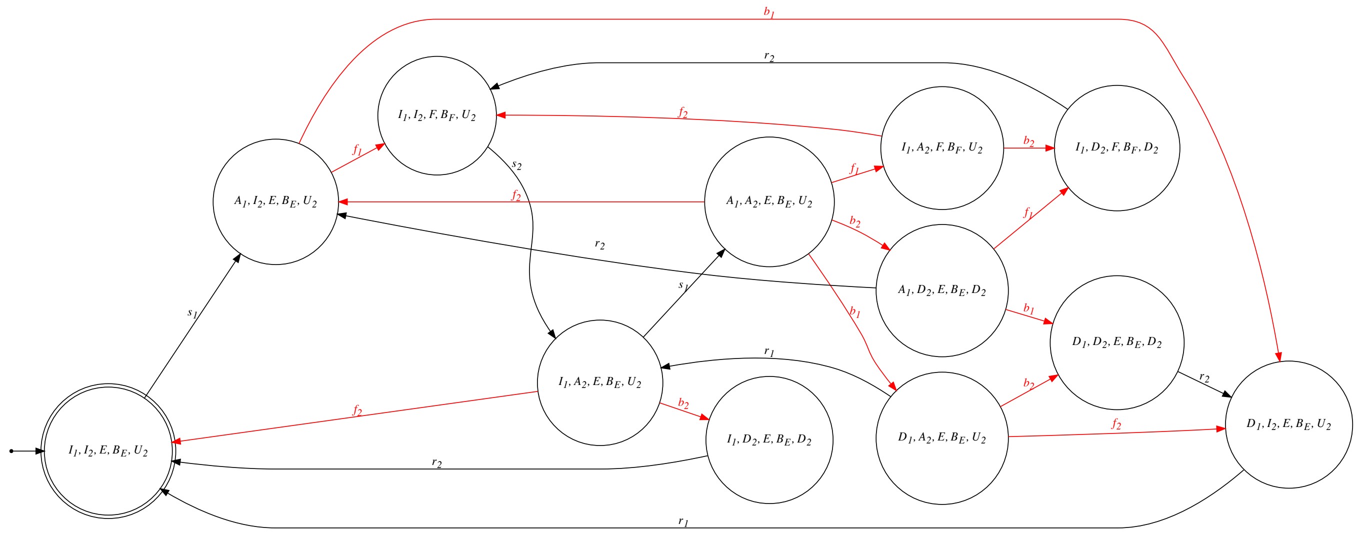


Figure 12: The initial, tentative, supervisor automaton $S := G \| R_{1,2} \| R_{3,4} = M_1 \| M_2 \| B \| R_1 \| R_2 \| R_3 \| R_4$.

$(m_1, m_2, b, r_{1,2}, r_{3,4}) \in \mathit{States}(S)\}$. After that, considering the events enabled by G in the various states, S restricts G by disabling the following events according to the state in which G is in.

- Whenever M_1 is in the initial state and the buffer is full, then S disables s_1 (Requirement R_1).
- Whenever M_2 is in the initial state and the buffer is empty, S does nothing (Requirement R_2 is already satisfied by G).
- Whenever M_1 is in the initial state and M_2 is broken, S disables s_1 (Requirement R_3).
- Whenever M_1 and M_2 are broken, S disables r_1 (Requirement R_4).

We also give a tabular view of this control strategy in Table 1 (to ease reading we focus on the state of the plant).

Part II

Consider M_1, M_2, and B of Part I.

Table 1: Tabular description of the strategy of S. We only show the events actually disabled by S with respect to the events that would be enabled by G.

State of G	**Events disabled by S**
(I_1, I_2, E)	$\emptyset$
(A_1, I_2, E)	$\emptyset$
(I_1, I_2, F)	$\{s_1\}$
(I_1, A_2, E)	$\emptyset$
(I_1, D_2, E)	$\{s_1\}$
(A_1, A_2, E)	$\emptyset$
(I_1, A_2, F)	$\{s_1\}$
(I_1, D_2, F)	$\{s_1\}$
(A_1, D_2, E)	$\emptyset$
(D_1, D_2, E)	$\{r_1\}$
(D_1, I_2, E)	$\emptyset$
(D_1, A_2, E)	$\emptyset$

Question 5. Build the plant G_1 as the parallel composition of M_1 and M_2 only. Is the result a shuffle?

Answer 5. Figure 13 shows $G_1 := M_1 \| M_2$. The result is a shuffle since M_1 and M_2 do not share any events. □

Let $K_1 \subseteq \mathcal{L}_m(G_1)$ be the language marked by G_1

restricted by the following requirement:

The buffer contains at most one piece.

Question 6. Build the automaton marking K_1.

Answer 6. We need to build an automaton H_1 such that $\mathcal{L}_m(H_1) = K_1$. We achieve this purpose by means of a parallel composition between G_1 (since $K_1 \subseteq \mathcal{L}_m(G_1)$) and another automaton that we do not have yet. We are looking for an automaton enforcing alternation between the events f_1 (of M_1) and s_2 (of M_2). In other words, such an automaton tracks the first occurrence of f_1, then disables f_1 up to the first occurrence of s_2, then disables s_2 up to the next occurrence of f_1, and repeats. But this is exactly the synchronization enforced by the buffer B in Part I. In this part, since B in is no longer part of the plant G_1, then such a synchronization is not guaranteed anymore by G_1 (which is not by chance a shuffle). Hence, such an automaton coincides with R_2 shown in Figure 7 (i.e., B with all states marked). Thus,

$$H_1 := G_1 \| R_2$$

Figure 14 shows the automaton H_1 marking K_1. Note that K_1 corresponds to the language marked by the plant G of Part I restricted to $R_{1,2}$, with a small difference: R_2 misses the self-loop labeled by s_1 at B_E. □

Question 7. Is K_1 prefix-closed?

Answer 7. A language L is prefix-closed (in symbols $L = \overline{L}$) if all prefixes of every string in L belong to L as well. A language $\mathcal{L}(A)$ generated by an automaton A is clearly prefix-closed since to generate any string in it, we need to generate all of its prefixes, incrementally. This is not guaranteed for marked languages. Indeed, for any automaton A, it holds that $\mathcal{L}_m(A) \subseteq \overline{\mathcal{L}_m(A)} \subseteq \mathcal{L}(A) = \overline{\mathcal{L}(A)}$, where all subset relations might, of course, be strict. Therefore, since $K_1 = \mathcal{L}_m(H_1)$, we need to check whether $\mathcal{L}_m(H_1) \subseteq \overline{\mathcal{L}_m(H_1)}$ is strict or not. Consider the automaton representation of H_1 in Figure 14. Starting from its initial state (I_1, I_2, B_E), it is easy to see that the string $s_1 f_1 \in K_1$ because by executing that sequence of events we end up in the state (I_1, I_2, B_F) which is marked. However, the prefix $s_1 \notin K_1$ because the state (A_1, I_2, B_E) is not marked. This is sufficient to conclude that K_1 is not prefix-closed since $K_1 = \mathcal{L}_m(H_1) \subset \overline{\mathcal{L}_m(H_1)} = \overline{K_1}$. □

Question 8. Is K_1 controllable? If so, describe the control strategy. Otherwise, compute $K_1^{\uparrow C}$.

Answer 8. To prove whether K_1 is controllable or not, we compute $K_1^{\uparrow C}$ and check if $K_1^{\uparrow C} \subseteq K_1$ is strict or not. In the former case K_1 is uncontrollable, whereas in the latter K_1 is controllable. To compute $K_1^{\uparrow C}$, we try to build a supervisor S_1 for G_1 with respect to the requirement H_1. If S_1 exists, then $\mathcal{L}_m(S_1) = K_1^{\uparrow C}$. After that, if the procedure discussed in Part I removes any state from S_1, then $K_1^{\uparrow C} \subset K_1$ and thus K_1 is not controllable. Otherwise, $K_1^{\uparrow C} = K_1$ and thus K_1 is controllable.

K_1 is not controllable. Initially, $S_1 := G_1 \| H_1$ (Figure 15a). Consider the state (A_1, I_2, h_6) of Figure 15a, where h_6 shortens the state (A_1, I_2, B_F) of H_1 (Figure 14). The event f_1 cannot be executed there. However, f_1 can be executed in the (corresponding) state (A_1, I_2) of G_1 in Figure 13. Therefore, $(A_1, I_2, h_6) = (A_1, I_2, A_1, I_2, B_F)$ is an uncontrollable state that we need to remove since the requirement automaton H_1 enforces the disabling of an uncontrollable event. Figure 15b shows the final S_1. Note that $\mathcal{L}_m(S_1) = K_1^{\uparrow C} \neq \emptyset$.

The control strategy of S_1 disables s_1 whenever M_1 is in its initial state. Indeed, the sequence of events s_1, f_1 (simulating the filling of the buffer) can be extended with, again, s_1, f_1 (simulating the overflow of the buffer). In other words, after any (sub)sequence s_1, f_1, s_1 we need to disable f_1. But this is a forbidden control action since f_1 is uncontrollable. Therefore, we need to prevent the second occurrence of s_1 whenever we observe a sequence s_1, f_1 until we see an occurrence of s_2. □

Let K_2 be the marked language of G_1 restricted to the requirement R_4 of Part I.

Question 9. Build an automaton marking K_2.

Answer 9. We need to build

$$H_2 := G_1 \| R_4$$

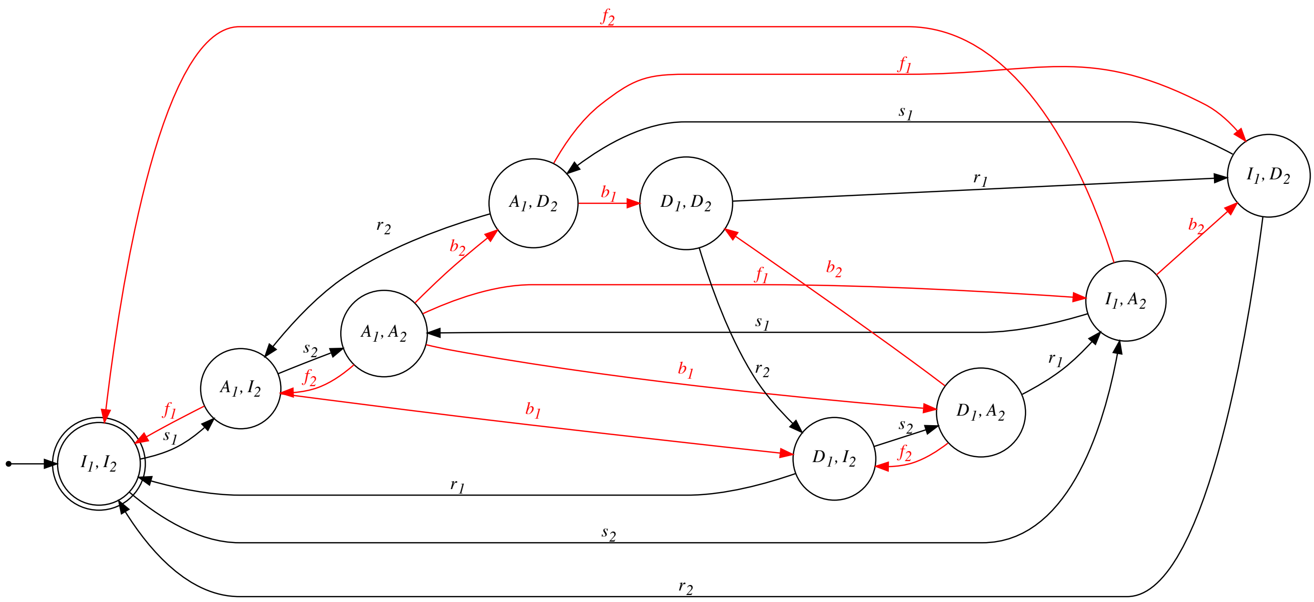


Figure 13: Plant automaton $G_1 := M_1 \| M_2$ (shuffle).

so that $K_2 = \mathcal{L}_m(H_2)$. Figure 16b shows H_2.

Question 10. Is K_2 prefix-closed?

Answer 10. Once again no. Consider the automaton representation of H_2 in Figure 16a. Starting from its initial state (I_1, I_2, U_2), it is easy to see that the string $s_2 f_2 \in K_2$ because by executing that sequence of events we get back to the initial state which is marked. However, the prefix $s_2 \notin K_2$ because the state (I_1, A_2, U_2) is not marked. This is sufficient to conclude that K_2 is not prefix-closed since $K_2 = \mathcal{L}_m(H_2) \subset \overline{\mathcal{L}_m(H_2)} = \overline{K_2}$. □

Question 11. Is K_2 controllable? If so, describe the control strategy. Otherwise, compute $K_2^{\uparrow C}$.

Answer 11. As we did for K_1, to prove whether or not K_2 is controllable, we try to build a supervisor S_2 for G_1 with respect to the requirement H_2 and see if we remove states in the process. We start by setting $S_2 := G_1 \| H_2$ which we show in Figure 16b. Since the algorithm does not remove any state, then $S_2 = G_1 \| H_2$ and K_2 is controllable. The control strategy is the following. If both M_1 and M_2 are down, r_1 is disabled. As soon as M_2 goes up again, r_1 becomes executable again. □

Question 12. Compare the languages $\overline{K_i}^{\uparrow C}$ and $\overline{K_i^{\uparrow C}}$ for each $i = 1, 2$.

Answer 12. We give the proof parametrized on i since its structure is the same. For each $i = 1, 2$, it holds that $\overline{K_i}^{\uparrow C} \supseteq \overline{K_i^{\uparrow C}}$. It remains to show if the inclusion is strict or not. To answer this question, we need to build two automata S_A and S_B such that $\mathcal{L}_m(S_A) = \overline{K_i}^{\uparrow C}$ and $\mathcal{L}_m(S_B) = \overline{K_i^{\uparrow C}}$ and test if they are equivalent. Let G_1' be G_1 with all states marked.

To build S_A we start from an automaton R_A such that $R_A := H_i$ but with all states marked. This way, $\mathcal{L}_m(R_A) = \overline{K_i}$. Then, we try to build a supervisor S_A for G_1' with respect to the requirement R_A as usual so that $\mathcal{L}_m(S_A) = \overline{K_i}^{\uparrow C}$. Initially, we set $S_A := G_1' \| R_A$.

To build S_B we set $S_B := S_i$ and mark all states so that $\mathcal{L}_m(S_B) = \overline{K_i^{\uparrow C}}$. By bisimulation...

We conclude with some variations of the questions we asked in this part. Let M_3 and M_4 be the same as

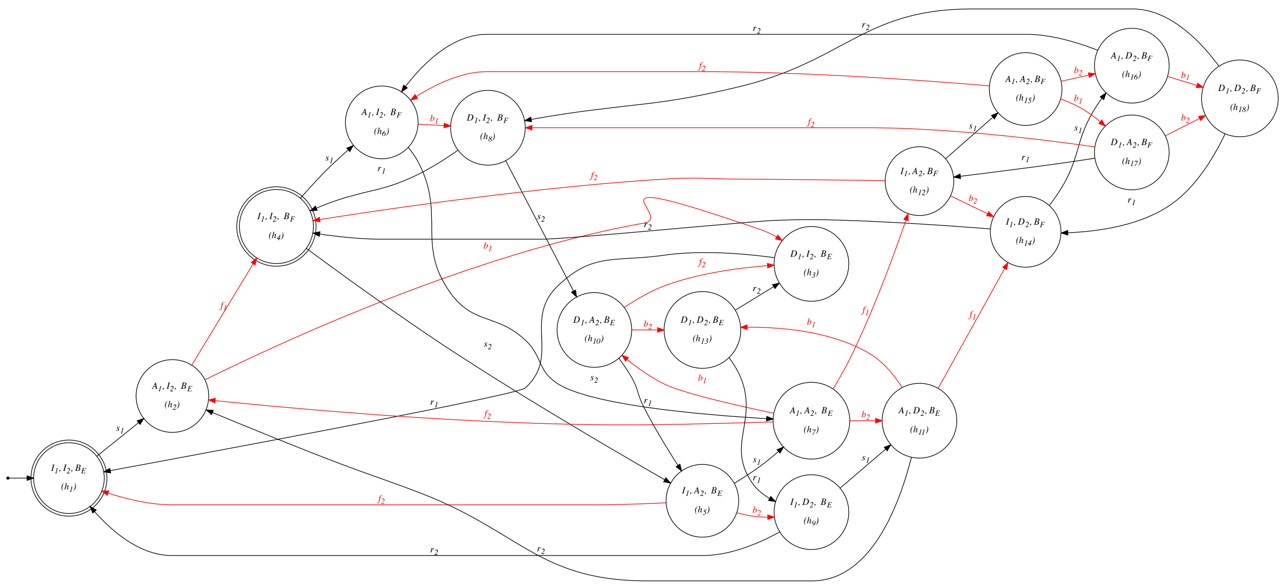


Figure 14: Automaton $H_1 := G_1 \| R_2$ marking K_1. We also shorten the label of each state as h_i for $i = 1 \ldots 18$.

M_1 and M_2, respectively, with the difference that the only marked states are those in which the machines are down.

Question 13. Build the automata M_3, M_4, and $G_2 := M_3 \| M_4$. Is G_2 a shuffle?

Answer 13. Figures 17a-17c show M_3, M_4, and $G_2 := M_3 \| M_4$. Once again, G_2 is not a shuffle since, as for M_1 and M_2 defined in Part I, M_3 and M_4 do not share any events. □

Let $K_3 \subseteq \mathcal{L}_m(G_2)$ be the marked language of G_1 restricted by the same requirement used to define K_1.

Question 14. Build the automaton for K_3.

Answer 14. The construction is identical to that given for K_1 but with respect to G_2. That is, we set $H_3 := G_2 \| R_2$ so that $\mathcal{L}_m(H_3) = K_3$ (Figure 18). □

Question 15. Is K_3 prefix-closed?

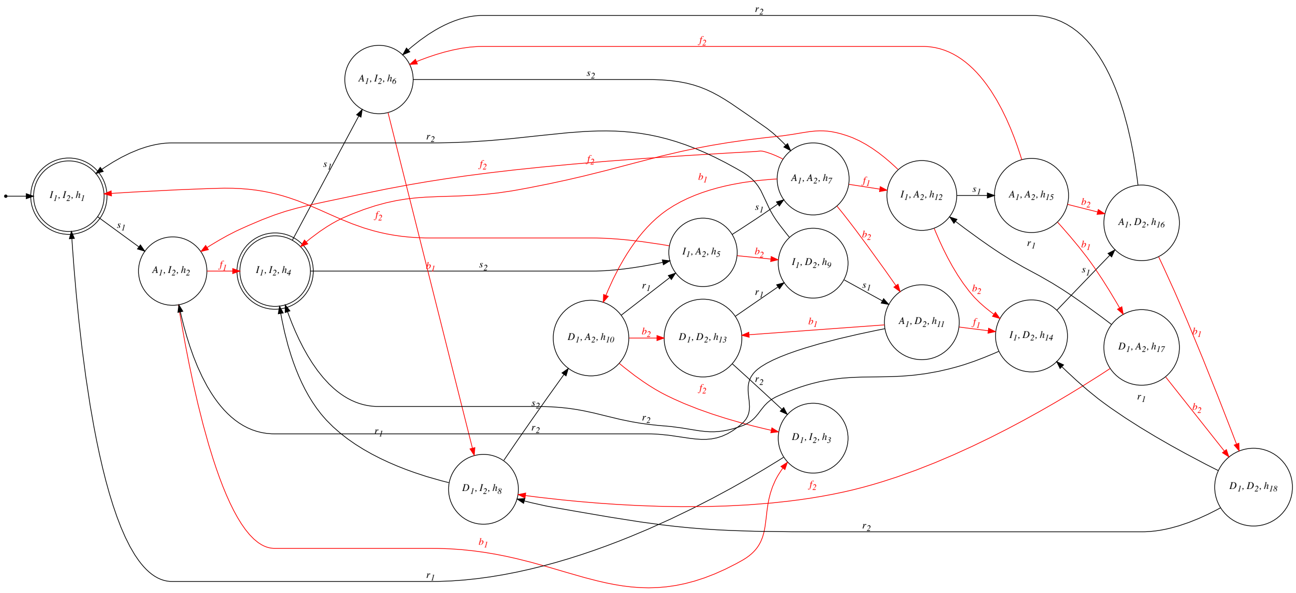
Answer 15. Still, no. Consider the automaton representation of H_3 in Figure 18. Starting from its initial state (I_1, I_2, B_E), it is easy to see that the string $s_1 f_1 s_1 b_1 s_2 b_2 \in K_3$ because by executing that sequence of events ends up in the state (D_1, D_2, B_E) which is marked. However, any (strict) prefix of such a string is not in K_3 because the states (I_1, I_2, B_E), (A_1, I_2, B_E), (I_1, I_2, B_F), (A_1, I_2, B_F), (D_1, I_2, B_F), and (D_1, A_2, B_E) are not marked. This is sufficient to conclude that K_3 is not prefix-closed since $K_3 = \mathcal{L}_m(H_3) \subset \overline{\mathcal{L}_m(H_3)} = \overline{K_3}$. □

Question 16. Is K_3 controllable? If so, describe the control strategy. Otherwise, compute $K_3^{\uparrow C}$.

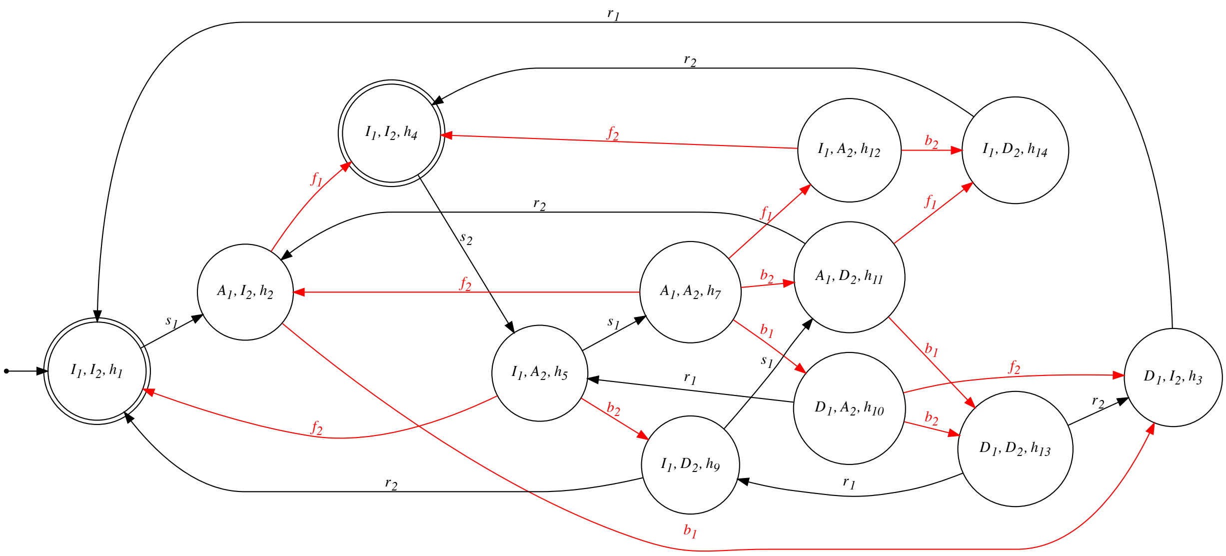
Answer 16. K_3 is not controllable. Same explanation as for K_1. Figure 19b shows the final S_3. Note that $\mathcal{L}_m(S_3) = K_3^{\uparrow C} \neq \emptyset$. The control strategy is the same as the one given for K_1 □

Question 17. Is $\overline{K_3}$ controllable?

Answer 17. No, a language K is controllable if and only if $\overline{K}$ is controllable. Since K_3 is not controllable, neither is $\overline{K_3}$.

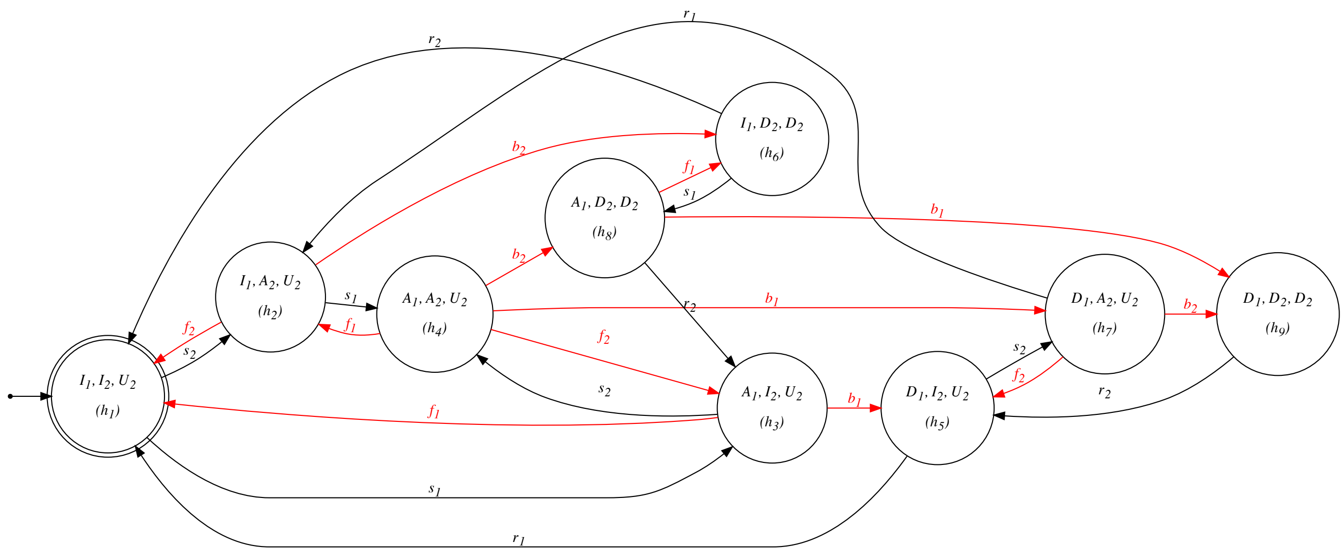


(a) Tentative (initial) supervisor $S_1 := G_1 \| H_1$.

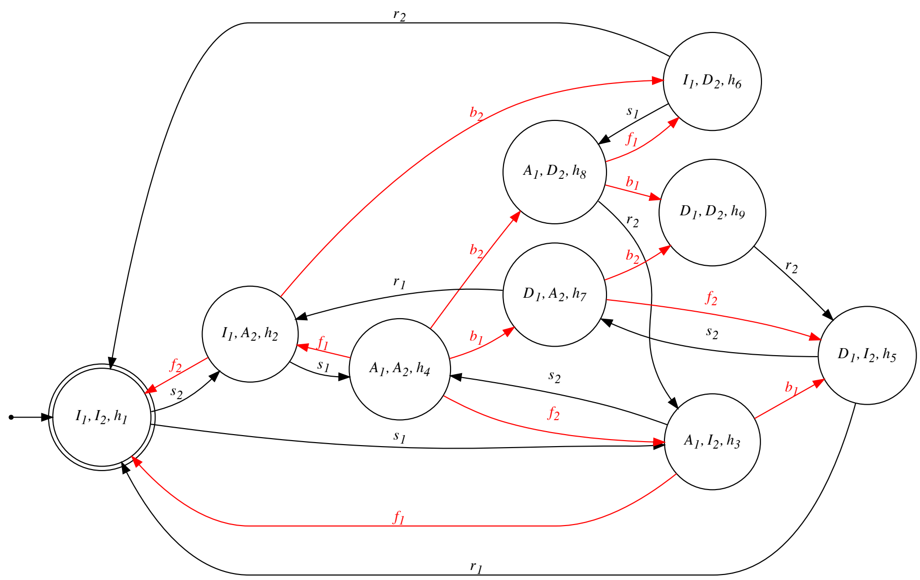


(b) Final supervisor S_1.

Figure 15: Initial and final status of the building of S_1.

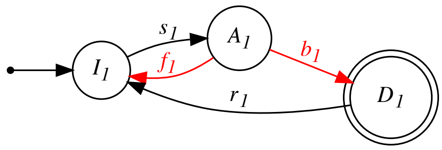


(a) Automaton $H_2 := G_1 \| R_4$ marking K_2. Once again, we also shorten the states as h_i for $i = 1 \ldots 9$.

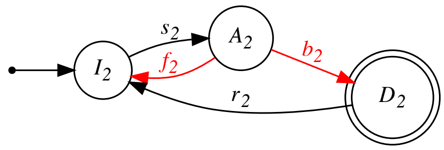


(b) Supervisor S_2.

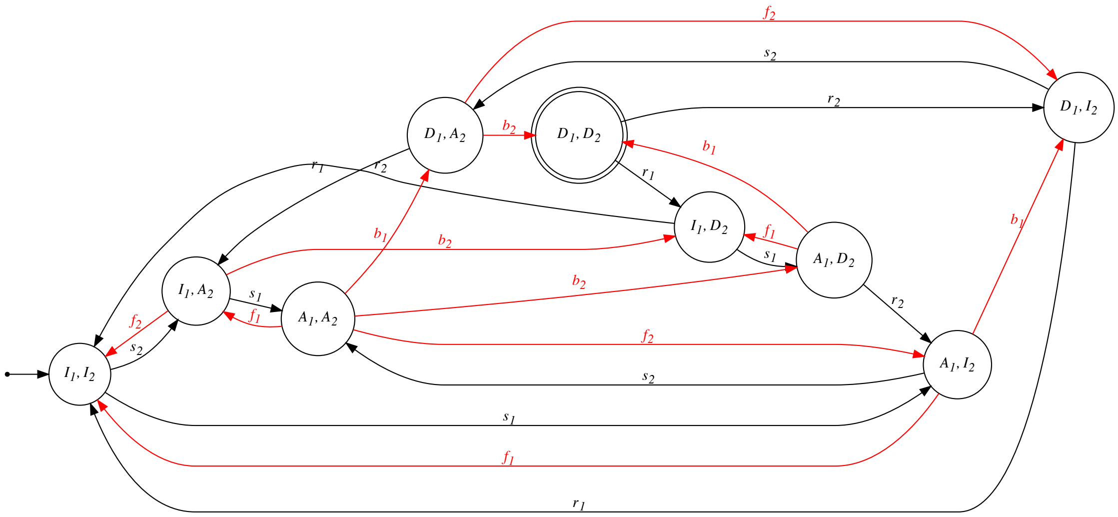
Figure 16: Automata H_2 and S_2.



(a) Automaton M_3



(b) Automaton M_4



(c) Automaton $G_2 := M_3 \| M_4$

Figure 17: Automata M_3, M_4, and G_2.

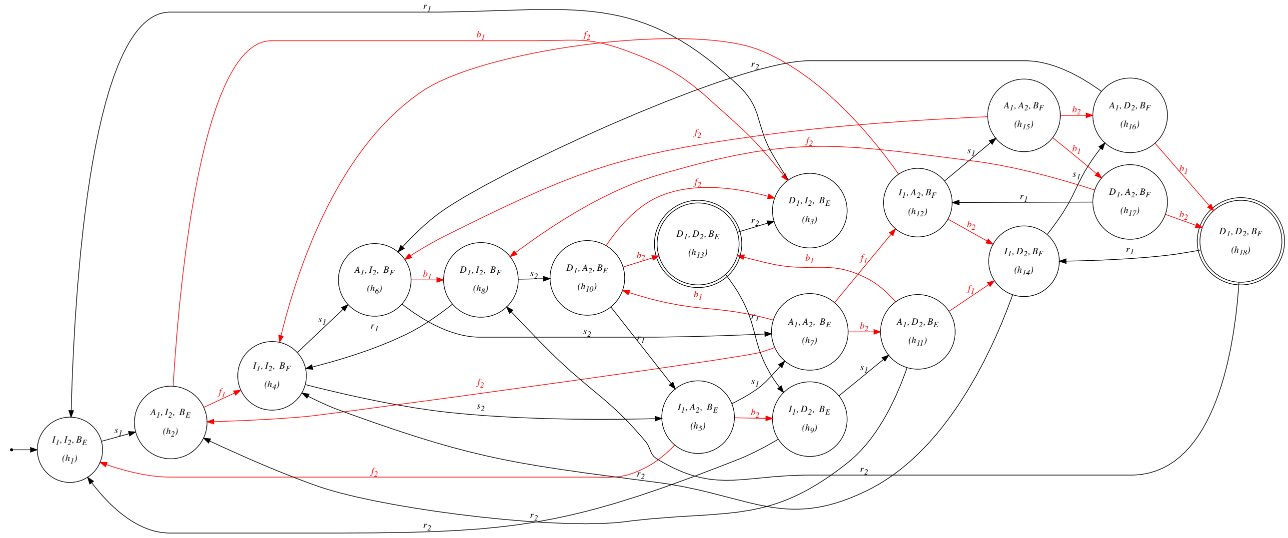
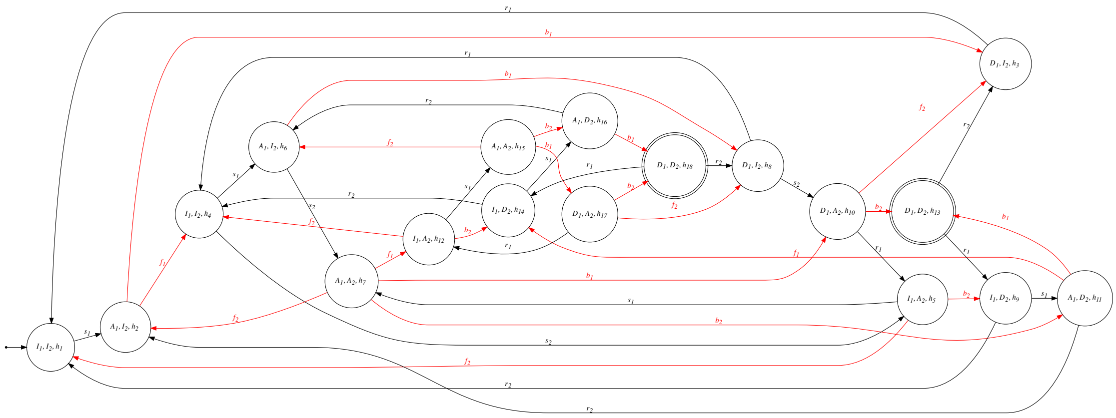
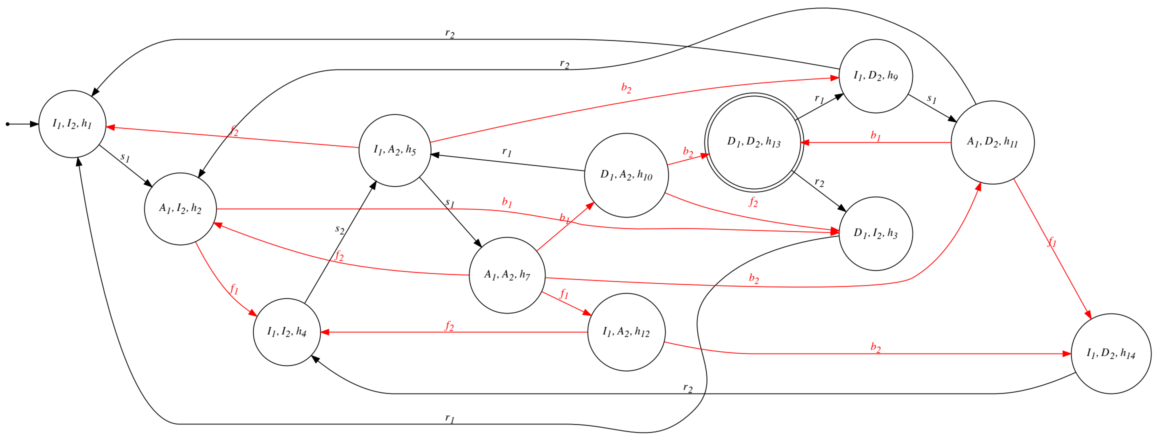


Figure 18: Automaton $H_3 := G_2 \| R_2$ marking K_3.



(a) Initial, tentative, supervisor $S_3 := G_2 \| H_3$.



(b) Final supervisor S_3.

Figure 19: Initial and final status of the building of S_3.