Parallel Planning in Asprilo

Hugo Robalino (799951), Maksym Nevar (800888), PhD François Laferrière
University of Potsdam

January 23th, 2023.

1 Introduction

Automated planning has been one of the first substantial applications of ASP. One example of that is Asprilo, in which ASP addresses automated planning with a sequential approach. Nevertheless, Its efficiency could be further improved by implementing parallel planning, ”since several independent operators (or actions) can be parallel at one time point, rendering unnecessary to consider all their total orderings during plan search, which is the case with sequential plans. Moreover, parallelism leads to decreased number of time points, which reduces the number of propositional variables and makes satisfiability testing more efficient” (Rintanen et al., 2006).

The structure of this report is as follows. In section 2 we present Asprilo, the framework where plan parallelization is to be implemented. In section 3 we introduce plan representation, which is the basis for our implementation of parallel planning. Next, in section 4 our parallel planning encoding for Asprilo is presented. Section 5 is where the experiments and their results are shown. Fiinally, section 6 concludes the report.

2 Asprilo Framework

Asprilo, which stands for Answer Set Programming for robotic intra-logistics, is a framework whose goal is to ”automatize warehouse operations by using robot vehicles that drive underneath mobile shelves and deliver them to picking stations” (Gebser et al., 2018). Within this framework, four main domains are used and several object types are described. However, we will only focus on one domain and the object type attributes pertinent to this research.

2.1 Object Types

Objects types are represented as facts that follow the format described below:

```
init(object(<object-type>, <object-id>), value(<attribute>, <value>)).
```

Below follows a pairing of current object types with their attributes, where the trailing represents the number of arguments of each attribute:

node	at/2
robot	at/2
shelf	at/2
pickingStation	at/2
product	on/1
order	line/2, pickingStation/1

Each node fact represents each position or ’square’ in the grid that can be occupied by a robot or a shelf. Robot facts represent the robots that will be moving through the grid carrying shelves to the picking stations. Shelves contain the products to be delivered to the picking stations. Picking stations are fixed and do not move, unlike robots and shelves. All of the aforementioned object types have the attribute `at` that indicates the node on the grid where they are located. Products' attribute `on` indicate in which shelf they are located. Finally, an order can consist of several instances of the line attribute detailing the needed product ID and the quantity required of that product, apart from taking the `pickingStation` attribute that indicates where all the product items in that order are to be delivered.

For instance, the facts below represent order number 1, which requires 20 units of product 3 delivered to picking station 2.

```
init(object(order,1),value(line,(3,20))).
init(object(order,1),value(pickingStation,2)).
```

2.2 Domain

Domain refers to the environment where the robot intra-logistics planning takes place. The general environment of the warehouse consists of its floor, represented as a two dimensional grid, multiple robots and shelves, which occupy one grid node each. Robots can move across the grid, occupy the same grid node as a shelf, as well as carry the shelves across the grid, relocate them and bring them to picking stations to deliver the products. Regarding products, multiple product quantities can be stored in shelves. The main goal is the completion of all orders, more specifically, an order is completed when all of its requested product units are delivered to its assigned picking station. The current domain used in the scope of this project disregards product quantities and considers an order to be delivered when a shelf containing the required product is positioned at a picking station. In other words, it corresponds to the original domain B.

3 Plan representation

Once the working framework (Asprilo) is established, we can now proceed to solve by means of plan representations. (Dimopoulos et al., 2019) provides different encodings for sequential and parallel representations of plans, which are given by the 4-tuple: $< \mathcal{F}, s0, s*, \mathcal{O} >$ where:

- $\mathcal{F}$ represents the *fluents* that have an associated finite domain of possible values.
- $s0$ represents the *initial state*. In other words, what value each fluent has at time step 0.
- $s*$ represents the *goal state*.
- $\mathcal{O}$ represents a finite set of operators or *actions*, where $a =< a^{prec}, a^{post} >$. In other words, each action consists of preconditions and postconditions

A successor state is obtained by applying an action to a state: $s' = o(a, s)$. Multiple actions can also be applied to obtain a successor state $s' = o(< a_1, ..., a_n >, s) = o(a_n, o(..., o(a_1, s)...))$. Following this, a successor state is said to be defined if $a^{prec}(x) = s(x)$ and $s'(x) = a^{post}(x)$ where x is a fluent. In other words, the preconditions of action a are fulfilled in the departing state s and the successor state s' includes the postconditions of action a and keeps any other fluents (apart from fluent x) from departing state s unchanged. In that sense, a sequential plan is given when a sequence of individual actions $< a_1, ..., a_n >$ applied to the initial state $s' = o(< a_1, ..., a_n >, s0)$ is defined and the successor state is the goal state $s' = s*$.

Once that the sequential plan representation is defined, (Dimopoulos et al., 2019) also proposes an array of parallel plan representations: $\forall$-step, $\exists$-step, and relaxed $\exists$-step, which are not covered in the current scope of this project, since they were not needed.

4 Encodings

In order to carry out our implementation, the plan representation of section 3 needs to be translated into the Asprilo framework, in other words, *fluents*, *initial states*, *goal states* and *actions* require counterparts in the Asprilo framework. Subsequently, they are represented as follows:

$\mathcal{F}$	*fluents*	`robot(R)` with finite values: `node(C)` and `carries(S)`
$s0$	*initial state*	`position(robot(R), node(C), carries(S), 0).`
$s*$	*goal state*	`position(robot(R), node(C), carries(S), T).`
$\mathcal{O}$	*actions*	`action(move(M); pickup(S); putdown(S); deliver(V)).`

`robot(R)` atoms take two values: `node(C)`, which encodes its position on the grid and `carries(S)`, which encodes whether or not and which shelf the robot carries. *Initial state* and *goal state* are encoded in the robot's position. The only difference between them is that `T` $= 0$ in the former whereas `T` $=$ `T` in the latter. It is important to note that the planning reaches its goal state when all deliveries are completed, which will be explained in section 4.3.

4.1 Object Representation

All the object types described in section 2.1 are represented as atoms given by `init` facts. For instance, a robot atom is instantiated as follows:

```
robot(R) :- init(object(robot, R), _).
```

Moreover, apart from the above mentioned object types, there is one more atom, namely the `carries(S)` atom, which shows whether a `robot(R)` is carrying a `shelf(S)`. It takes the value of 0 when no shelf is being carried: `carries(0)`.

4.2 Position Representation

The position of robots, shelves and picking stations on the grid is tracked by means of the position predicate with different number of arguments:

```
robot           position(robot(R), node(C), carries(S), T).
shelf           position(shelf(S), node(C), T).
pickingStation  position(station(P), node(C)).
```

For instance, the predicate `position\4` tracks where `robot(R)` is on the grid by means of the `node(C)` argument and whether it carries a shelf with the `carries(S)` argument at time step `T`.

4.3 Goal Encoding

The end goal is for all orders to be delivered at their corresponding picking station. For doing so, the auxiliary atom `link(V)` is instantiated, which connects order `O`, its corresponding shelf `S` where product `X` is and picking station `P` where order `O` is to be delivered. Once this auxiliary atom is set up, a constraint is instantiated where `link(V)` must occur. In other words, this constraint ensures that order `O` gets delivered.

```
link((order(O), product(X), shelf(S), station(P))).
:- link(V), not occurs_p( _ , action(deliver(V)), _ ).
```

4.4 Actions

In the current Asprilo domain, robots can take the following actions:

```
action(move((D,C))).
action(pickup(S)).
action(putdown(S)).
action(deliver(V)).
```

The `move` action takes two arguments: variable `D`, which is a tuple: `direction((X,Y))`, representing the direction of the movement in the grid. A robot can only move one node in the X or Y axis per time step; otherwise said: $|X+Y| = 1$. It also takes the variable `C`, which is also a tuple: `node((X',Y'))` and represents the robot's position before the action `move` takes place.

Actions `pickup` and `putdown` take only the variable `S` as an argument, representing the `shelf(S)` that will be either picked up or put down by the robot at the same position. Finally, action `deliver` takes the argument `V`, which represents the same set of variables of `V` in `link(V)` explained in the last section.

At this stage, once the rules that define actions are set, actions were then subsequently classified into two categories represented by the predicates: `isAction_m`, which only includes the `move` action and `isAction_p`, in which all other actions are included.

Once all actions were classified in these two categories, we processed each set of actions in a sequential manner. In the subsequent step, the two sets of actions were merged to form a parallelized answer.

4.5 Preconditions and Postconditions

Following (Dimopoulos et al., 2019) framework, sets of rules for preconditions and postconditions are established for each action:

```
prec(action(move((D, C))), robot(R), node(C)) :- ...
post(action(move((D, C))), robot(R), node(C')) :- ...
```

The first rule ensures that `robot(R)` is located at `node(C)` as a precondition for the `move` action to take place. The second rule ensures that `robot(R)` is located at `node(C')` at the next time step as a postcondition for the `move` action.

```
prec(action(pickup(S)), robot(R), carries(0)) :- ...
post(action(pickup(S)), robot(R), carries(S)) :- ...
prec(action(putdown(S)), robot(R), carries(S)) :- ...
post(action(putdown(S)), robot(R), carries(0)) :- ...
```

This set of rules ensures that `robot(R)` is not carrying any shelf, represented by the variable `carries(0)` as a precondition of the `pickup` action and it also ensures that `robot(R)` is carrying a shelf through the variable `carries(S)` as the postcondition for the `pickup` action. Conversely, it also ensures that a shelf is being carried as the precondition for the `putdown` action and that no shelf is being carried as the postcondition for the aforementioned action.

```
prec(action(deliver(V)), robot(R), carries(S)) :- ...
post(action(deliver(V)), robot(R), carries(S)) :- ...
```

Finally, this last set of rules ensures that a shelf is being carried as the precondition and postcondition of the `deliver` action by means of the `carries(S)` variable, since a robot needs to carry a shelf in order to deliver a product and will also be carrying that shelf once the delivery is done.

4.6 Auxiliary Rules

Now that preconditions and postconditions are established, some auxiliary rules are needed in order to guarantee the proper functioning of Asprilo's framework. Those auxiliary rules pertain the deliver action, shelf movement and superposition constraints.

Listing 1: Action deliver

```
1 timedLink(R,(O,X,S,P),C,T) :- link((O,X,S,P)),  position(R,C,_,T), position(S,C,T),
    position(P,C).
2 :- occurs_p(R,action(deliver(V)),T), position(P,C), not timedLink(R,V,C,T).
3 T1=T2 :- occurs_p(_,action(deliver(V)),T1), occurs_p(_,action(deliver(V)),T2).
```

Listing 1 makes sure that the `deliver` action happens only when robot `R`, shelf `S` and picking station `P` are located on the same node `C` at time step `T` by means of the auxiliary predicate `timedLink` on line

1 and the constraint on line 2. Moreover, it also ensures that no order is delivered twice in the constraint on line 3.

Listing 2: Shelf movement control

```
1 on(S,R,T) :- position(R,_,carries(S),T), S!=0.
2 {on(S,R,T)} :- occurs_p(R,action(putdown(S)),T).
3 position(S,C,T) :- position(R,C,_,T), on(S,R,T).
4 position(S,C,T) :- position(S,C,T-1), not on(S,_,T), T=1..h.
5 C=C' :- position(S,C,T-1), position(S,C',T), occurs_p(R,action(pickup(S)),T), not
      position(R,C,_,T-1).
6 :- on(S,R,T), occurs_p(R,action(putdown(S)),T), not occurs_m(R,_,T).
```

In order to make sure that shelf movement behaves as expected, the auxiliary atom `on` was implemented on lines 1 and 2 to track whether a shelf `S` is carried on robot `R` at time step `T`. Then, we couple the shelf position to the robot position if the shelf is located on the robot on line 3 and we implement shelf inertia on line 4 if the shelf is not on a robot. Finally, constraint on line 5 states that a shelf can only move when it is picked up by a robot and the constraint on line 6 makes sure there are no repeated solutions by deleting the answer where the atom `on(S,R,T)` exists after shelf `S` has been put down at time step `T` as, in fact, shelf `S` is not on robot `R`.

Listing 3: Superposition constraints

```
1 R1=R2 :- position(R1,C,_,T), position(R2,C,_,T).
2 R1=R2 :- position(R1,_,B,T), position(R2,_,B,T), B!=carries(0).
3 R1=R2 :- position(R1,C,_,T), position(R2,C',_,T), position(R1,C',_,T+1), position(R2
      ,C,_,T+1).
4 S1=S2 :- position(S1,C,T), position(S2,C,T).
```

Listing 3 takes care that only one robot is located at a certain node on line 1, in other words, two robots are not allowed to be located at the same node in the same time step. Moreover, on line 2, it is stated that a robot cannot carry more than one shelf per time step. Furthermore, constraint 3 prevents robot collision by prohibiting two robots moving towards the node the other is occupying. Finally, line 4 allows only one shelf per node.

4.7 Parallelization

Once the auxiliary rules are established, the parallel planning core can be implemented based on the plan representation described in section 3.

Listing 4: Parallelization part I

```
1 {position(R,C,B,T) : isNode(C), isCarry(B)} = 1 :- isRobot(R), T=1..h.
2 {occurs_m(R,A,T)} :- isRobot(R), isAction_m(A), T=1..h.
3 {occurs_p(R,A,T)} :- isRobot(R), isAction_p(A), T=1..h.
4 :- occurs_m(R,A,T), post(A,R,C), not position(R,C,_,T).
5 :- occurs_p(R,A,T), post(A,R,B), not position(R,_,B,T).
6 T=0:- position(R,C,_,T), not position(R,C,_,T-1), not occurs_m(R,A,T): post(A,R,C).
7 T=0:- position(R,_,B,T), not position(R,_,B,T-1), not occurs_p(R,A,T): post(A,R,B).
```

Listing 4 handles the matching of successor states (robot positions) to postconditions of applied actions. Choice rule on line 1 generates successor positions so that each robot is mapped to some value (`node(C), carries(B)`) in its domain. The next following lines is where the parallelization planning is implemented. To do so, each category of actions, as explained in section 4.4, is processed separately in a sequential manner so that two actions, one action from the m category (`isAction_m`) and one from the p category (`isAction_p`), can take place in the same time step. That is to say, `action(move(M))` could be combined with either `action(pickup(S))`, `action(putdown(S))` or `action(deliver(V))`.

The following choice rules and constraints are applied separately first to m actions and then to p actions. Lines 2 and 3 unconditionally pick up actions to apply, encoded in the `occurs_m` and `occurs_p` atoms, to obtain a successor state. Then, constraints in lines 4 and 5 assert that the relevant postcondition(s) hold in the respective successor positions. Finally, lines 6 and 7 make sure that robot positions unaffected by applied actions remain unchanged while establishing changed robot positions uniquely by means of applied actions.

Listing 5: Parallelization part II

```
1  :- occurs_m(R,A,T), prec(A,R,C), not position(R,C,_,T-1).
2  :- occurs_p(R,A,T), prec(A,R,B), not position(R,_,B,T-1).
3  A1=A2:- occurs_m(R,A1,T), occurs_m(R,A2,T).
4  A1=A2:- occurs_p(R,A1,T), occurs_p(R,A2,T).
```

Lines 1 and 2 in Listing 5 take care of that preconditions of applied actions hold in the robot position at time step `T-1`, while lines 3 and 4 deny multiple actions to be applied in parallel. Here, it is relevant to remark that line 3 denies multiple m actions to be applied in parallel while line 4 does so with p actions, in other words, it will never be the case that multiple actions from the same category (m or p) occur in the same time step.

Finally, in the last part of the parallelization encoding is where all the sequentially applied actions come together and merge under one atom. More specifically, atoms `occurs_m` and `occurs_p` merge under the atom `occurs` by means of the directive `#show` in order to have a proper parallelized answer.

5 Benchmarking

5.1 Foundational Experiments

To verify all possible parallelized action combinations work as expected, we created a minimal instance of 3 nodes, 1 robot, 1 shelf, 1 picking station, 1 product and 1 order as shown in figure 2 in the Appendix. To run the experiment, the number of time steps, or *horizon* `h`, needs to be established manually. On account of that, we set `h` to 3. Subsequently, we obtained 10 possible answers, confirming all the parallelized actions could actually take place as shown in table 2.

After proving parallelized actions worked as expected with a single robot and a shelf, a second instance was created to test if the encodings could handle multiple robots and shelves as shown in figure 3. In this case, `h` was set to 2 producing only one possible answer shown in table 3.

5.2 Experiments

Consequently, we proceeded to establish the number of orders as the main variable for the experiments and grid size as a second variable. Other variables were set as follows:

Robots	3 units
Shelves	3 units
Picking Stations	3 units
Products	7 kinds
Grid size	4×4, 5×5

Then, we proceeded to manually choose the smallest possible `h` value for each instance and record the run time to get the first answer for each instance and its corresponding `h` value as presented in table 1.

	Grid 4×4		Grid 5×5	
orders	*horizon*	seconds	*horizon*	seconds
1	5	0.01	7	0.03
2	9	0.41	11	5.08
3	9	0.54	12	3.11
5	10	0.82	13	5.52
10	13	7.27	17	102.64
12	15	32.92	18	141.59
15	16	285.54	20	1049.93
21	21	799.64	-	-

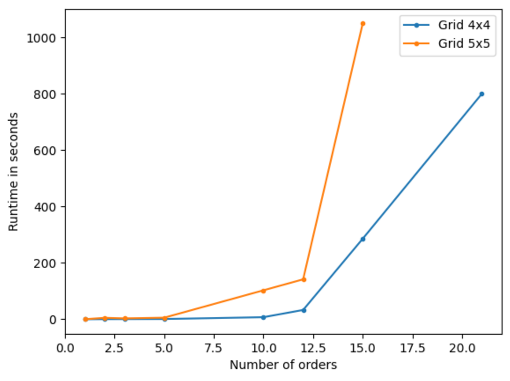
Table 1: Experiment results

With regard to the experiments layouts, every robot, shelf, picking station and product stayed in the same position while the number of orders changed. There is one layout for each grid size variable, which can be found in figures 4 and 5 in the Appendix.

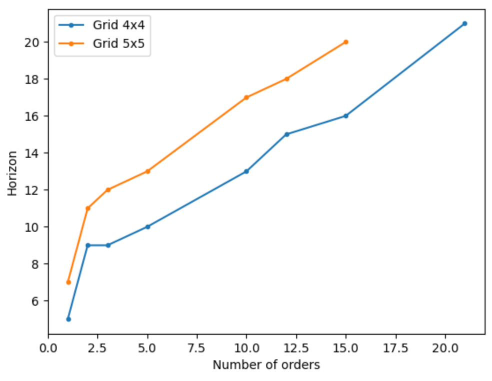
5.3 Results Analysis

It can be observed in figure 1 that both, horizon and runtime have a positive correlation with the number of orders. However, while horizon seems to have a linear relationship in subfigure 1b, runtime appears to have an exponential relationship in subfigure 1a. This is further proven if we take a look at table 1, where there is no *horizon* value or runtime value for the 21 orders row and grid 5×5 column due to a timeout.

Based on the graphs, it is sensible to conclude that this encoding is not scalable, since it times out at a relatively small grid size and number of orders. This might be due to the fact that it searches through all possible answers before returning the first answer.



(a) Number of orders - runtime



(b) Number of orders - horizon

Figure 1: Experiment results

5.4 Visualization

Since the Asprilo visualizer does not accept parallelized actions, we decided to slightly modify the answer by means of a clingo script, so that the answers can be visualized. To better clarify the modification, we could use answer 3 from minimal instance A as example.

Parallelized answer:

```
occurs(object(robot,1),action(move,(1,0)),1).
occurs(object(robot,1),action(pickup,()),1).
occurs(object(robot,1),action(move,(1,0)),2).
occurs(object(robot,1),action(deliver,(1,1)),3).
```

Modified answer:

```
occurs(object(robot,1),action(move,(1,0)),1).
occurs(object(robot,1),action(pickup,()),2).
occurs(object(robot,1),action(move,(1,0)),4).
occurs(object(robot,1),action(deliver,(1,1)),6).
```

In the modified answer, if two actions happen in consecutive time steps (such as `move...1` and `pickup...2` in the first two lines), that means they happen in parallel. Otherwise, if two actions happen in non-consecutive time steps, that means they happen in different time steps in the parallelized answer (such as `move...4` and `deliver...6` in the last two lines).

6 Conclusion

It is possible to achieve a partial parallelization when one `move` action happens simultaneously with another action (`pickup`, `putdown` or `deliver`) by handling each set of actions separately in a sequential manner and merging them afterwards. Nevertheless, the achieved encoding is not scalable as it quickly runs into timeouts at a relatively small grid size and with few number of orders. On possible reason of this lack of efficiency increase could be due to our encoding being sequential in nature and therefore not taking real advantage of parallel planning benefits. Future research is needed to improve the runtime and to achieve a truly scalable encoding.

References

Yannis Dimopoulos, Martin Gebser, Patrick Lühne, Javier Romero, and Torsten Schaub. plasp 3: Towards effective asp planning. *Theory and Practice of Logic Programming*, 19(3):477–504, 2019.

Martin Gebser, Philipp Obermeier, Thomas Otto, Torsten Schaub, Orkunt Sabuncu, Van Nguyen, and Tran Cao Son. Experimenting with robotic intra-logistics domains. *Theory and Practice of Logic Programming*, 18(3-4): 502–519, 2018.

Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence*, 170(12-13):1031–1080, 2006.

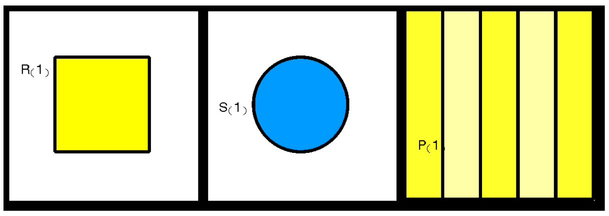


Figure 2: Minimal Instance A (Robot R, Shelf S, Picking station P)

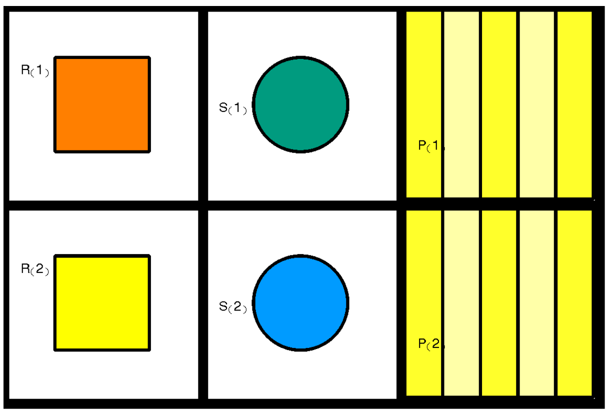


Figure 3: Minimal Instance B

Answer #	time step 1	time step 2	time step 3
Answer 1	$\Longrightarrow \upuparrows$	$\Longrightarrow \circ$	$\emptyset$
Answer 2	$\Longrightarrow$	$\upuparrows \Longrightarrow$	$\circ$
Answer 3	$\Longrightarrow \upuparrows$	$\Longrightarrow$	$\circ$
Answer 4	$\Longrightarrow$	$\upuparrows$	$\Longrightarrow \circ$
Answer 5	$\emptyset$	$\Longrightarrow \upuparrows$	$\Longrightarrow \circ$
Answer 6	$\Longrightarrow \upuparrows$	$\emptyset$	$\Longrightarrow \circ$
Answer 7	$\Longrightarrow \upuparrows$	$\Longrightarrow \circ$	$\downdownarrows$
Answer 8	$\Longrightarrow \upuparrows$	$\Longrightarrow \circ$	$\Longleftarrow$
Answer 9	$\Longrightarrow \upuparrows$	$\Longrightarrow \circ$	$\downdownarrows \Longleftarrow$
Answer 10	$\Longrightarrow \upuparrows$	$\Longrightarrow \circ$	$\Longleftarrow \downdownarrows$

$\Longrightarrow$ move right, $\Longleftarrow$ move left, $\upuparrows$ pick up, $\downdownarrows$ put down, $\circ$ deliver, $\emptyset$ no action

Table 2: Answers for Minimal Instance A in figure 2

Robot #	time step 1	time step 2
Robot 1	$\Longrightarrow \upuparrows$	$\Longrightarrow \circ$
Robot 2	$\Longrightarrow \upuparrows$	$\Longrightarrow \circ$

Table 3: Answer for Minimal Instance B in figure 3

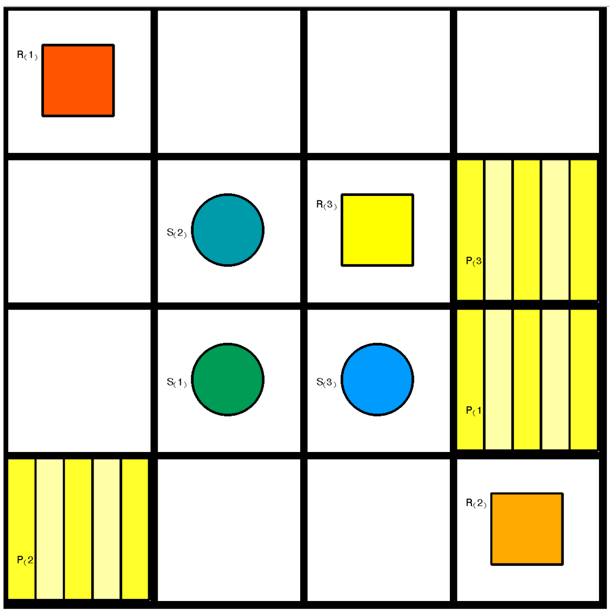


Figure 4: Grid size 4×4 layout

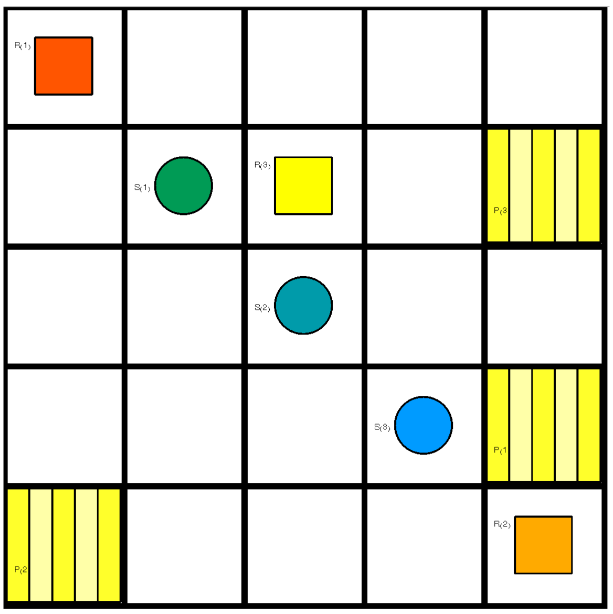


Figure 5: Grid size 5×5 layout