Simon György Szatmari

A Formalization of the Paradigm Coordination Language, applied to the n-Arm Robot Problem

Master's thesis

September 29th, 2016

Thesis supervisors: Dr. F.M. Spieksma, Dr. L. Groenewegen

Mathematisch Instituut, Universiteit Leiden

Abstract

A formalization for the Paradigm coordination language is introduced, in terms of graph products and Markov processes. The n-arm robot problem is presented, where a robot composed of several jointed arms tries to plan and to move. We find that the n-arm robot problem reduces to a concurrent competition for the space in which the arms evolve. This competition for space reduces to the classical reader/writers problem. Moving in space is equivalent to writing and sensing in space is equivalent to reading. Several coordination schemes for the arms are developed in Paradigm. Multiple arms can plan paths concurrently, but reserving space to move has to be done sequentially, and moving can be done concurrently. The solutions are simulated in Matlab, and the code is provided. Thereupon, the solutions are compared, and it is shown that not all coordination schemes are equivalent. Finally an outline of future work is provided.

Acknowledgements

I want to thank Dr. L. Groenewegen for introducing me to the exciting realm of concurrency and coordination, and Dr. F.M. Spieksma for introducing me to rigorous mathematical modeling. Thanks to the endless discussions we had, I have started to understand some fundamental aspects of coordination and of modeling. I want to thank Groenewegen for sharing with me, without holding any ideas back, the Paradigm coordination language of his own invention. Needless to say that none of this project would have been possible without Dr. Spieksma tireless suggestions and to-the-point corrections, which I believe have made me into a better modeler, and have greatly improved the quality of this work. I especially want to thank both of them for putting up with my wild ideas and weird approaches, patiently guiding me towards a rigorous exposition of what I had in mind.

Moreover, I need to thank the Mathematical institute, Leiden university, and its wonderful teachers, for their exciting courses, engaging discussions, and amazing cordiality. Needless to say that I thank my loving mother, who has supported me every step of this long journey. Finally, I need to give special thanks to all my friends who have given me moral stamina and emotional support, and a place to sleep when needed.

Contents

1 Introduction

The initial aim of this thesis was to develop an algorithm, method, or framework to solve the n-arm coordination problem, without using the PRM (probabilistic road-map method) method 3.1 (16), and using the Paradigm concurrent coordination language. The n-arm coordination problem is the problem of planning paths for n jointed arms sharing a common a space. In particular, the usual solution for this is to deploy a PRM-type method, considering all the arms as one super-robot, and to solve path-plannings at once for all the arms, as on one thread of execution. PRM-type path-planners and A*-type path-planners (local greedy search) are introduced in section 3 (p. 3), they are the most used algorithms in path-planning robotics.

If one is to consider the separate arms as separate entities, or agents, then a concurrent or multithreaded setting is entered. The separate arms, sharing a common space, can be thought as competing agents for the shared resource that is space. This organization necessitates a concurrent coordination scheme. Three coordination schemes are developed, the Super-Robot Model, the Round-Robin Solution and the Split and Non-Deterministic Solution, presented in section 5 (p. 39).

These schemes solve, at the multithreaded level, the organization of information flow for the different agents in the path-planning. These solutions are developed in the concurrent design language Paradigm, which is presented in section 4 (p. 39), developed by Dr. L. Groenewegen and Dr. E. de Vink.

An initial investigation into the probabilistic properties of these models is conducted in section 6 (p. 51). We use the standard assumption that each state of each agent takes a holding time characterized by an exponential distribution.

The probabilistic analysis given is limited, hence we resort to numerical simulations. In section 7 (p. 56), we present how we simulated all the concurrent coordination models. The simulation is presented in pseudo-code and the actual code is given in MATLAB code, in section 10 (p. 77).

The results of the simulations are presented in section 8 (p. 70). The results are finally compared in section 9 (p. 75). Moreover, in this last section the prospective next steps in this research are given.

2 Setup: World, Arm, Path-Planning, Moves

A common strategy to model space is to discretize it: sections of space are collapsed into vertices and edges are established between the vertices if the corresponding sections of space are connected [24, 30, 23]. If an object is in a certain section of space, then the object occupies the corresponding vertex. If there is an edge from that vertex to another vertex, then the object can, other constrains notwithstanding, move from the former section of space to the section of space represented by the latter vertex. We use the following grid discretization of space:

Definition 2.1 (World W, n-dim)**.** An n-dimensional world $W = (V(W), E(W))$ is an undirected graph, with $V(W)$ being the set of vertices and $E(W)$ being the set of edges, such that

$$V(W) \subseteq \mathbb{Z}^n \tag{2.1}$$

$$E(W) = \{(v_1, v_2) \,|\, 0 \leq \|v_1 - v_2\|_\infty \leq 1, v_1, v_2 \in V(W)\}. \tag{2.2}$$

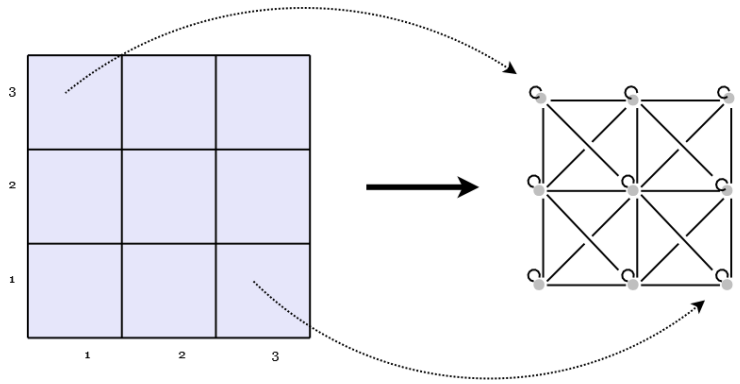

Figure 1: Each square becomes a vertex in the graph. Note the edge from each vertex to itself, representing null motion.

Robotic arms are usually made up of several limbs, connected together by joints. At the joints there are usually actuators [24, 30, 23]. There are several types of robotics arms, but in this work the focus is on the modeling of the most basic type: the limbs of the arm are rigid bodies and the arm is actuated at the joints, moreover the robot has a base fixed to a certain area in the world. In this modeling, a robotic arm is a simple graph where the vertices represent the solid rigid bodies that are the limbs and the edges represent the joints and actuators between them. Moreover, the structure of an arm does not change in time, for instance the arm cannot be cut, a limb cannot be detached and then reattached, etc.

Definition 2.2 (Arm)**.** An *arm* A is an undirected graph in which any two vertices are connected by exactly one path; such an undirected graph is also called a tree.

Definition 2.3 (Limb)**.** A *limb* L of an arm A is a vertex of A: $L \in V(A)$.

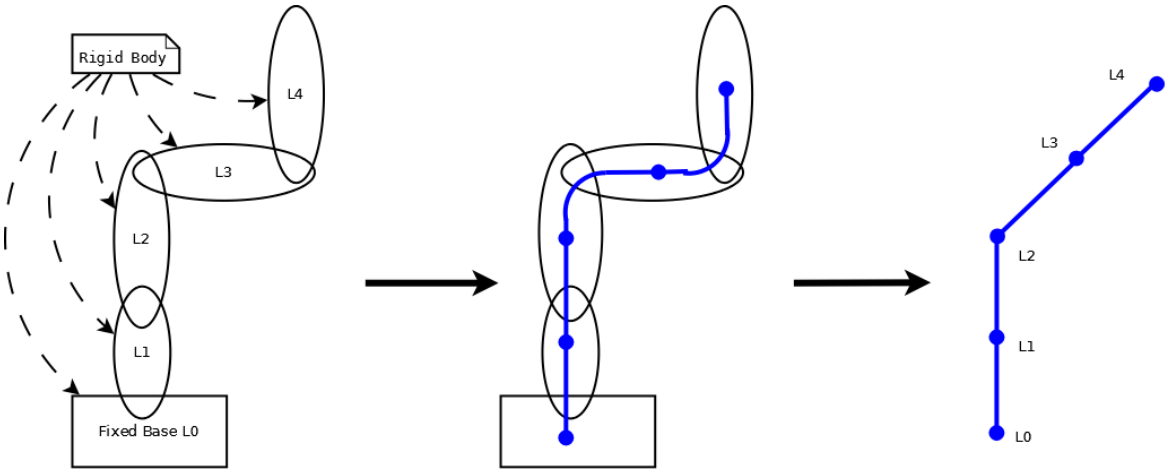

Figure 2: One limb L per rigid body in the arm A.

It is reasonable to think of a limb L of an arm A as occupying a certain section of space in a world W, in other terms the vertex $L \in V(A)$ is on a certain vertex of W. Similarly, we have to place the arm A into the world W. In robotics, this is usually referred to as the pose of a robot. We have to be careful however, the structure of A has to be preserved when it is placed in W and two vertices of A cannot occupy the same vertex in W: the vertices of A represent limbs which are rigid bodies and thus they are not allowed to overlap. This placement is precisely what a graph homomorphism is:

Definition 2.4 (Graph Homomorphism)**.** A *graph homomorphism* h from a graph A to a graph W is a mapping $h : V(A) \to V(W)$ such that $(u, v) \in E(A)$ implies $(h(u), h(v)) \in E(W)$.

Homomorphisms are injective, so that two limbs cannot be mapped to the same vertex in the world, and graph homomorphisms are structure preserving, so that the integrity of an arm is preserved in the world.

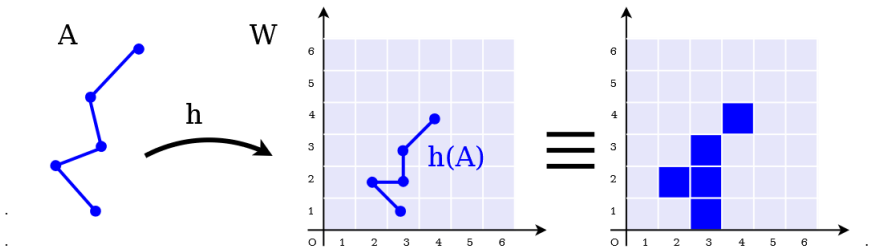

Figure 3: Arm A in world W via homomorphism h. The third picture is for visualization.

Definition 2.5 (Hom)**.** The space of graph homomorphisms from A to W is denoted $\mathrm{Hom}(A,W)$.

Definition 2.6 (Pose)**.** A *pose* of an arm A in a world W via a graph homomorphism $h : V(A) \to V(W)$ is $h(A) \subset W$.

A common task in robotics is to plan a trajectory from an initial robot pose to a desired goal pose [24, 30, 23]. The aim of path-planning can be to find a trajectory from the initial pose to *any* pose where limb L touches a certain specific vertex in W, or to find a trajectory from the initial pose to a specific pose. We will cover this in section 3 (p. 14). We are interested in collision-free trajectories only.

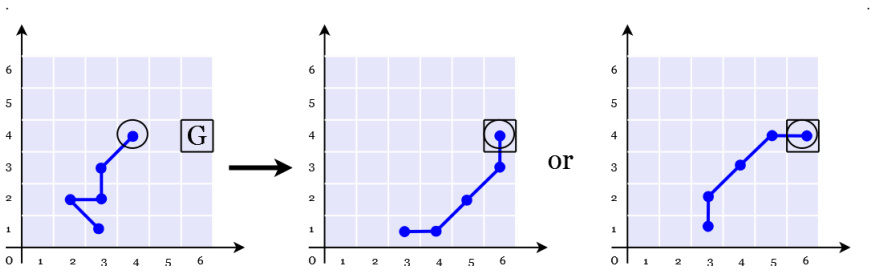

Figure 4: Initial configuration on the left, in the middle and on the right two acceptable goal configurations.

The path-planning algorithm has to specify how to move from the initial arm configuration $h^0(A)$ (initial pose) to the desired goal configuration $h^T(A)$ (goal pose), or to an intermediate configuration $h^t(A)$ (intermediate pose). As an analogy, think of a movie of the robot arm moving from $h^0(A)$ to $h^T(A)$, where $h^0(A)$ would be the beginning of the movie and $h^T(A)$ the end, then each movie frame would correspond to a certain $h^t(A)$. Given $h_0(A)$, the path-planning algorithm will have to output a sequence of graph homomorphisms $(h^1,\ldots,h^T)$, $T<\infty$.The exact value of T is not crucial. However, it is imperative for homomor-

phism h^t to be compatible with homomorphism h^{t-1}: at each time step, any limb $L \in V(A)$ can only move using only one edge of W. This warrants the following.

Definition 2.7 (Move)**.** A *move* M is a function $M : \mathrm{Hom}(A,W) \to \mathrm{Hom}(A,W)$, such that for homomorphism $h \in \mathrm{Hom}(A,W)$ and for any vertex $v \in V(A)$, we have that the edge $((M(h))(v), h(v))$ is in $E(W)$.

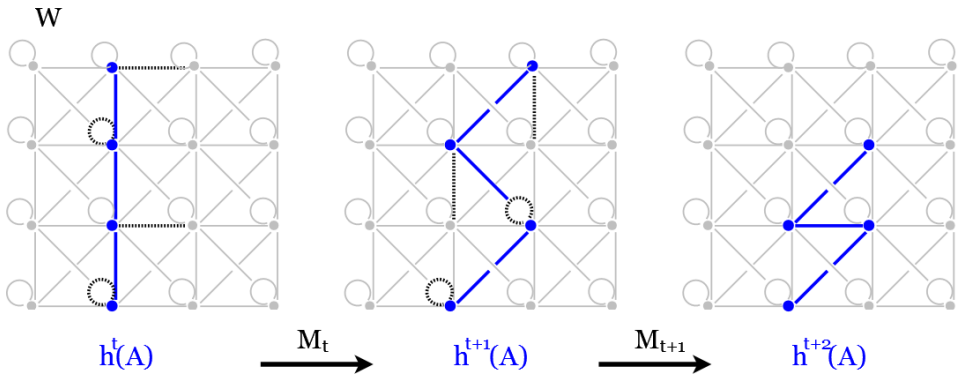

Figure 5: The edges in dashed black are $(M(h)(v), h(v)) \in E(W)$. Note that it would be possible to go from pose $h^t(A)$ to pose $h^{t+2}(A)$ in one move.

Definition 2.8 (Path-Planning)**.** A sequence of homomorphisms $(h^1, \ldots, h^T)$, $T < \infty$ is a *path-planning* of an arm A in a world W if there exists a sequence of moves $(M_1, \ldots, M_{T-1})$ such that $h^{t+1} = M_t(h^t)$ for any $t = 1, \ldots, T-1$.

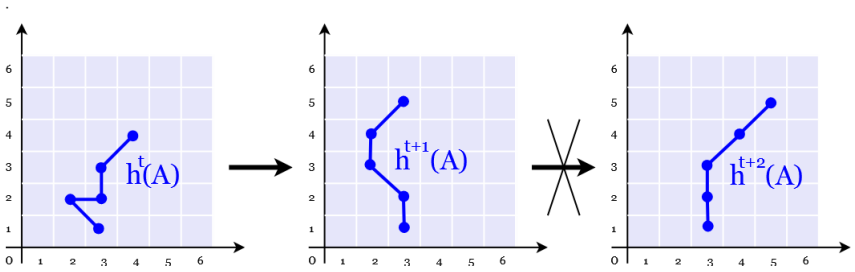

Figure 6: The first transition is allowed, the second it not: each limb can only move to a neighboring vertex! Note that one move allows the displacement of several vertices.

2.1 The n-Arms Robot Problem

The n-arm coordination problem addresses the asynchronous parallelism as arising in the collaboration of n robotic arms $A_1, \ldots, A_n$ sharing a common, but otherwise empty, world W. We first present the n-arm coordination problem, when only one arm is moving. We follow

up by presenting the n-arm coordination problem, when all the arms are moving simultaneously and asynchronously. Note that the goals of each arm may or may not be independent of each other, however the solutions to achieve these goals are dependent.

2.2 One Active Arm

The arms $A_1, \ldots, A_n$ are initially placed into W via homomorphisms $h_i^0(A_i)$, $h_i^0 \in \mathrm{Hom}(A_i, W)$. Say that only arm A_i is active and that the others are resting. Clearly arm A_i cannot occupy the space occupied by any of the other arms, so that no trajectory of A_i can intersect with any of the other arms. The organization of the different arms in this case is trivial: the information $\bigcup_{j\neq i} h_j^0(A_j)$ has to be provided to A_i, one way or another. In implementation, this message passing figures as the reading of a shared matrix representing the world W. Because only arm A_i is active, there are no concurrency problems arising from the sharing of this matrix. The planning of a collision-free trajectory can subsequently take place, given that an objective has been specified, using any collision-avoiding path-planning algorithm.

The set of obstacles arm A_i has to avoid when doing the path-planning and then moving has to be defined both on the vertex set $V(W)$ of and on the edge set $E(W)$ of W. On the vertex set, the set of obstacles the arm has to avoid is $\bigcup_{j\neq i} h_j^0(V(A_j)) \subset V(W)$. On the level of the edge set, the arm has to avoid more than just $\bigcup_{j\neq i} h_j^0(E(A_j)) \subset E(W)$.

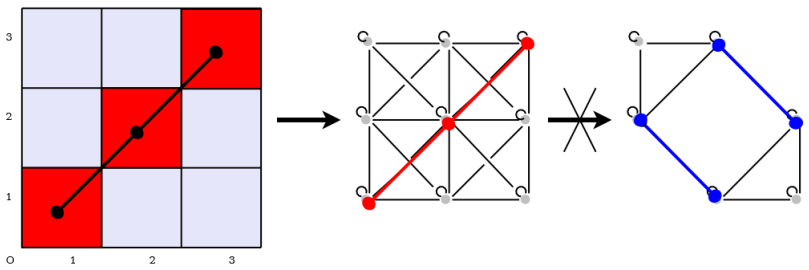

Figure 7: On the left, an arm $h^0(A)$, in red, is in the space W. On the right is the result of $W \backslash h^0(A)$, with the classical graph subtraction. The blue edges should should not be allowed, since an arm should not be allowed to move diagonally across another arm.

In particular, if an arm A_j has two adjacent vertices on a diagonal edge in W, then the cross-diagonal has to be disallowed as well. This leads to the following definition.

Definition 2.9 (Difference operation \ on world W)**.** For a world W of dimension n and a

subgraph $G = \{V(G), E(G)\}$, the graph $W \backslash G$ is defined by

$$V(W \backslash G) = \{v \in V(W) | v \notin V(G)\}, \tag{2.3}$$

and

$$\begin{aligned} E(W \backslash G) = \{(v, v') \in E(W) | v, v' \in V(W \backslash G) \text{ and if } \|v - v'\|_1 = i, i = 2, \ldots, n \\ \text{then } \nexists (w, w') \in E(G) \text{ s.t.} \\ w, w' \in B_{1,i-1}(v) \cap B_{1,i-1}(v') \text{ and } \|w - w'\|_1 = i.\} \end{aligned} \tag{2.4}$$

where $B_{1,i}(v)$ is the ball of vertices around v of radius i using $\|.\|_1$.

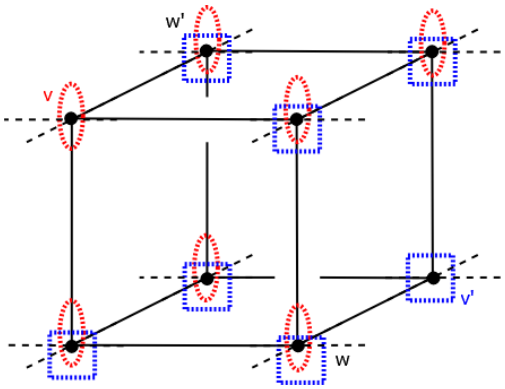

Figure 8: Note that $\|v - v'\|_1 = 3$. The neighborhood $B_{1,2}(v)$ is shown by the dashed red ellipses, whereas the neighborhood $B_{1,2}(v')$ is denoted by the dashed blue boxes. Edge $(v, v') \in E(W \backslash G)$ is not allowed if there is an edge w, w' in $E(G)$ such that w, w' is in the intersection of the two neighborhoods and $\|w - w'\|_1 = 3$.

Definition 2.10 (Path-planning with obstacle)**.** A sequence of homomorphisms $(h^1, \ldots, h^T)$, $T < \infty$ is a *path-planning* of an arm A in a world W with obstacle set $O \subset W$ if there exists a sequence of moves $(M_1, \ldots, M_{T-1})$, $M_t : \text{Hom}(A, W \backslash O) \to \text{Hom}(A, W \backslash O)$, using the graph subtraction of the definition above, such that $h^{t+1} = M_t(h^t)$ for any $t = 1, \ldots, T-1$.

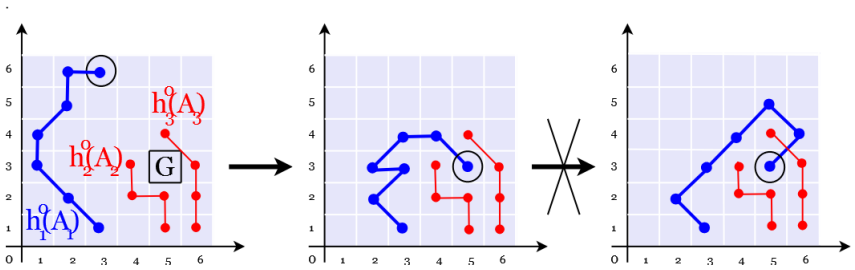

Figure 9: The case on the right cannot happen, because edge $((5,3),(6,4))$ is disallowed, since it is crossing edge $((6,3),(5,4))$.

Once the path-planning step is complete, the arm A_i moves from the initial configuration $h_i^0(A_i)$ to the goal configuration $h_i^T(A_i)$, by moving from pose to pose using the homomorphisms of the path-planning. In implementation, this moving figures as the writing into the shared matrix representing the world W.

Once the moving is complete, the arm A_i can restart the above procedure: once an objective has been specified, get the set of obstacles by reading the world matrix, then do a path-planning and finally take the sequence of moves specified by the path-planning.

Procedure.

1: Get objective.

2: Get obstacle set.

3: Do path-planning.

4: Do moving.

2.3 All Arms Active

The n-arm coordination problem addresses the asynchronous parallelism as arising in the collaboration of n robotic arms $A_1,\ldots,A_n$ sharing a common, but otherwise empty, world W. As we just saw, if only arm A_i is moving, then we might consider the other arms as static obstacles. If all the arms need to move, then the situation is more difficult, but not intractable. The heart of the matter is that the space W is a finite resource for which the arms compete [8, 1, 16]. This competition forces a collaboration of agents, where the agents are the arms. Two arms cannot be in the same area at the same time. As each arm strives to achieve its objective through path-planning and motion, two arms *will not* be able to be in the same area in the future. Therefore, there needs to be an organization to the path-planning processes

of the different arms. For any arm A_i and path-planning $(h_i^1, \ldots, h_i^{T_i})$, its current and future position corresponds to

$$\bigcup_{t=0,\ldots,T_i} h_i^t(A_i) \subset W. \tag{2.5}$$

Therefore, the obstacle set for arm A_j is

$$\bigcup_{i \neq j} \bigcup_{t=0,\ldots,T_i} h_i^t(A_i) \subset W. \tag{2.6}$$

On one hand, there needs to be a path-planning algorithm that solves path-planning problems for the arms. On the other hand, an additional construct (procedure) is needed to manage the arms and to manage their path-planning algorithms. This procedure organizes the separate dynamics of the arms, by defining transitions on the product (space) of the separate dynamics. The procedure is constructed in Paradigm, a concurrent coordination language [15, 16, 17, 3]. The Super-Robot Strategy, the Round-Robin Strategy and the Partial Reservation Strategy, solve this problem and are presented below. The Paradigm versions of the Super-Robot procedure, of the Round-Robin procedure and of the Critical Section procedure, are given in subsections 5.1 (p. 39), 5.2 (p. 40), and 5.3 (p. 43).

2.3.1 The Super-Robot Strategy

The first go-to procedure consists of thinking of the arms $A_1, \ldots, A_n$ as of one super-robot, and to do the path-planning for that one super-robot, such as on one thread of execution. After a more costly path-planning stage, all the arms will be able to move at the same time.

Procedure. 1: Get objectives for all $A_1, \ldots, A_n$.

2: Do path-planning for the robot "$A_1 + \ldots + A_n$".

3: Do moving for all of the arms $A_1, \ldots, A_n$.

2.3.2 The Round-Robin Strategy

The second go-to procedure is to let only one arm move at a time, implementing the strategy mentioned in subsection 2.2 (p. 6) for each arm, in a round-robin fashion. One arm gets one batch. The easiest way to path-plan in this strategy is to let the path-planning of arm A_i take place when it is its turn. Depending on the path-planning algorithms used, the path-planning of arm A_i, if done during the motion of another arm, would have to be partially or entirely recomputed, see section 3 (p. 14). This is a consequence of the obstacle set changing for an arm, as another arm moves.

Procedure.
1: Initialize index $i = 0$.
2: **while** TRUE **do**
3: Get objectives for $A_1, \ldots, A_i$.
4: **if** A_i has an objective **then**
5: Get obstacle set for A_i.
6: Do path-planning for A_i.
7: Do moving for A_i.
8: Remove objective for A_i.
9: **end if**
10: $i = (i+1) \mod n$.
11: **end while**

2.3.3 The Partial Reservation Strategy

The procedure that we propose lies between these two extremes. The approach taken is to let each arm plan and move simultaneously as much as possible, but as separate entities. In particular, all arms are allowed to do path-planning simultaneously, but only one arm is allowed to reserve space in W at a time. Say arm A_i is the first one to finish the path-planning process, giving a path-planning $(h_i^1, \ldots, h_i^T)$, it then has to tell the other arms it needs $\bigcup_{t=0,\ldots,T} h_i^t(A_i) \subset W$, while beforehand the other arms only know that arm A_i is in $h_i^0(A_i) \subset W$. In implementation, this passing of information figures as the writing into a shared matrix representing the world W, which by abuse of notation we also call W. As soon as this reservation is completed, arm A_i is allowed to start moving in it. Meanwhile, as soon as this reservation is completed, the other arms have to be interrupted and forced to update their knowledge of W and then to correct their path-planning solutions. Chunks of space they were using in their path-planning might, now or in the future, be occupied by A_i. In implementation, this updating translates as a rereading of W. Finally, as the arm A_i finishes to move, by taking the final pose $h_i^T(A_i)$, it frees up $\bigcup_{t=0,\ldots,T-1} h_i^t(A_i)$ in W, since it does not need that space anymore. Each arm completes the moves along a path-planning, and the size of the path-planning can be controlled by the designer.

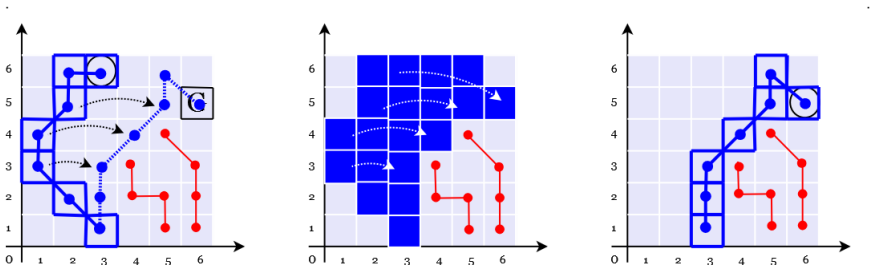

Figure 10: On the left initial path-planning only, in the middle space reservation and then motion, on the right clean-up.

This passing of information gives rise to the classical reader/writer problem: to avoid race conditions and uncertainty, the shared memory resource W needs protection [36, 10, 36]. Indeed, if two arms are allowed to write into W simultaneously, then they might both think that they have reserved the same area of space and both might start moving in the same area, resulting in a collision. If an arm is reading while another is writing, then the one reading might get an outdated set of obstacles, which might cause a path-planning on a collision course with another arm. The usual response to this problem is to deploy a critical section solution [40], [35, 22, 37, 16], where only one agent is allowed to write at a time, and some mechanism that ensures the correct updating for the other agents.

The detailed presentation of the extra procedure used in this work, solving the path-planning problem and the critical section problem, is delayed to section 5 (p. 39). Note that the critical section problem is also solved by procedures 2.3.1 (p. 9) and 2.3.2 (p. 10). However, these two procedures are in some circumstances worse than the approach preconized in this work. In section 7 (p. 56), a simulation on one thread is developed. In section 8 (p. 70), numerical simulations compare the performances of these procedures. Finally, in the next section, we give a description of the detailed possible behaviors, encapsulated into states, of each arm A_i when considered individually, as if unconstrained by the existence of the other arms.

2.4 Detailed Behavior Explanation

We give a detailed description of the possible behaviors, encapsulated into states, of each arm A_i when considered individually, as if unconstrained by the existence of the other arms. From now on we refer to arm A_i as Arm_i when considered in the view of the collaboration solution between arms, in view of section 4 (p. 25). We will write A_i when considered as a

simple graph, in view of definition 2.2 (p. 3).

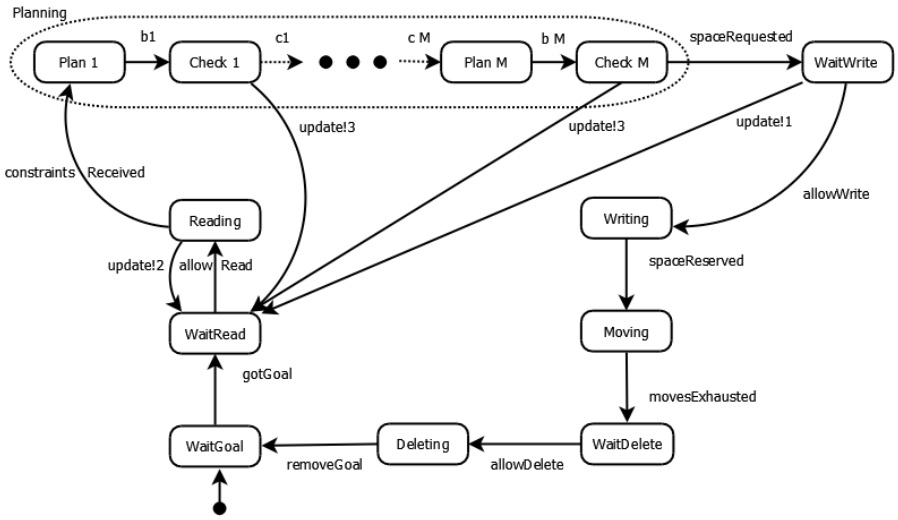

Figure 11: Each state encapsulates blocks of code, whereas the arrows illustrate the flow between the blocks.

Let $\mathsf{Arm}_1,\ldots,\mathsf{Arm}_n$ evolve in an world W. Any component Arm_i first starts by getting an objective, otherwise it has no reason to path-plan or to move. The behavior of getting an objective is encapsulated as state WaitGoal, to signify that the arm will wait until an objective is received from an exterior source. Thus WaitGoal is the initial state. As an objective is obtained, Arm_i proceeds to the state WaitRead, a bookkeeping state before state Reading, mainly important for the Split and Non-Deterministic Solution of subsection 5.3 (p. 43). Subsequently, Arm_i proceeds to read W, this is encapsulated in state Reading, loading W into local memory. As it gains the knowledge of W and of the constraints imposed by the other arms, Arm_i proceeds to path-planning, this is encapsulated in the sequence of states $\mathsf{Plan\ 1},\mathsf{Check\ 1},\ldots,\mathsf{Plan\ M},\mathsf{Check\ M}$, where the path-planning algorithm is sliced up into M jobs, and M is specified by the designer. Most path-planning algorithms solve the path-planning problem in cycles (for-loop or while-loop), so that this is doable in practice. The states $\mathsf{Plan\ 1},\ldots,\mathsf{Plan\ M}$ are actually responsible for the path-planning, and the states $\mathsf{Check\ 1},\ldots,\mathsf{Check\ M}$ are effectively breaking conditions, responsible for potentially suspending the computation. Whenever convenient, we will denote the sequence $(\mathsf{Plan\ 1},\mathsf{Check\ 1},\ldots,\mathsf{Plan\ M},\mathsf{Check\ M})$ as state Planning. As a path-planning is obtained from the path-planning algorithm, Arm_i proceeds to state WaitWrite, in which it will wait for the critical section permission to write into W.

Since it might be that during states Reading, Planning or WaitWrite another arm got the critical section permission to write into W, Arm_i might transition back to WaitRead to get an update. This is illustrated by arrows update1!, update2!, update3!. The update sends the arms into state WaitRead, which can only proceed to state Reading, thus ensuring that the knowledge of W will be updated.

As Arm_i obtains the critical section, it proceeds to write into W: this is encapsulated by state Writing and state Deleting, as deleting is also a form of writing. As soon as the space reservation is completed, Arm_i is free to safely start its motion, this is encapsulated in state Moving. As the moving draws to an end and the arm becomes still, Arm_i does not need all the space reserved in the previous state, and thus it cleans up after itself by freeing space, but not before asking for the critical section again in state WaitDelete. If the arm is already in possession of the critical section, then the permission to delete is automatically granted, otherwise the arm has to wait to get the critical section anew. Therefore, an arm will need the critical section at most twice per planning: once to reserve the space to move and once to free the space previously reserved. This freeing of space is encapsulated in state Deleting. As the space is freed, Arm_i will have attained its objective (that it initially got from WaitGoal), removes that objective via action removeGoal. Subsequently, the arm waits in state WaitGoal for the arrival of a new objective.

3 A Visualization of Path-Planning Coordination

A plethora of different methods have been developed over the years, trying different combinations of theories, approaches, heuristics, and sub-methods. The fields of influence range from animal socio-biology to mathematical group-theoretic topology [24, 27, 9, 34, 14, 23]. This menagerie of algorithms makes it is necessary to have a structured way of thinking about the similarities and differences in these methods.

A robot has a certain set of actuators. An actuator is a component responsible for moving parts of a system. In section 2 (p. 2), a robot A moves from pose $h^i(A)$ to pose $h^{i+1}(A)$ in a world W. In reality, such a move would be the result of actuators being actuated. We confound actuators and edges of a robot as one concept, even though in hardware they might differ.

Moreover, many path-planning algorithms consider a robot as one unit, as a robot with no joints, for instance such as a car occupying one square at a time on a map. In such a definition a robot is then just one node $A = \{a\}$ injected into W.

Definition 3.1 (Robot)**.** A robot A is a disjoint union of arms, such a graph is also called a forest.

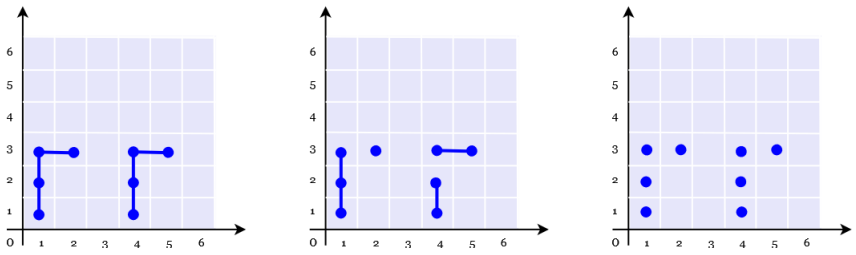

Figure 12: Different coordination setups corresponding to different path-planning organizations.

In a robot A, the number of joints is $|E(A)|$ and the number of limbs is $|V(A)|$. Most path-planning algorithms plan on $E(A)$, if $|E(A)| \geq 1$ and on $V(A)$, if $|V(A)| = 1$. However on one hand, it is possible to consider many unitary robots as a graph with no edges, where $|V(A)| > 1$, and it is possible to consider one complex arm as made up of many robot units, where the units have to respect some juxtaposition constraints.

We visualize path-planning algorithms by looking at how different threads are responsible for different sub-robots. The literature often refers to *coupled* planning for the case

where the path-planning sits on one thread of execution, and to *decoupled* planning where the path-planning uses several threads of execution [14, 33]. First, put all the vertices of a robot into a vector, on a level corresponding for time t_0. Put one such vector per time slice t_i, at most a countable number of them. On each level, the elements of the vector are first colored with a tag associated to at least one thread of execution. Subsequently, each element of the vector is associated a color P (for planning) or M (for moving) per thread tag.

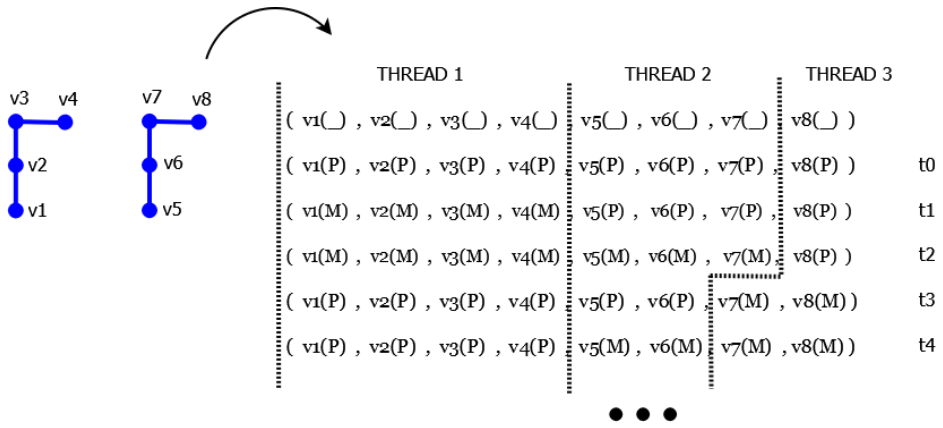

Figure 13: Different threads manage different vertices. In theory, more than one thread could manage one vertex, and the vertices that are managed by one thread could change over time.

Moreover, almost all path-planning strategies come in two flavors: *dynamic* and *non-dynamic*. By a dynamic path-planning algorithm, we mean an algorithm that does not have to recompute the complete path-planning for a local change in W. By non-dynamic algorithm, we mean an algorithm that has to recompute the complete path-planning solution for a local change in W. This dynamicity can be happen both for vertices of color P or of color M. For instance, see subsection 3.1 (p. 16).

As shown in the previous section 2 (p. 2), communication between different threads gives rise to coordination problems, because the matrix representing space has to be shared. In the literature, the coordination structure of path-planning algorithms is not usually declared, since it is assumed that the robot is totally coupled or totally decoupled. However, many more combinations are possible, but there needs to be a coordination scheme coordinating the different threads, which is usually not easy to do. For this reason, the coordination language Paradigm is presented in section 4 (p. 25), and some solutions are developed in section 5 (p. 39).

In the remainder of this section, a short literature review of state-of-the-art path-planning

algorithms is presented, classifying them with our strategy.

3.1 PRM-Type Planners

Let A be an arm, evolving in a world W. The original Probabilistic Road-map Method or PRM [21], uses the a two stage strategy for finding a path-planning (sequence of poses) between two robot poses.

- A preprocessing or sampling stage of the configuration space $\mathbb{C}(A,W)$, in which a road-map is constructed.
- A query phase or path computation phase, for given start and end poses.

This type of methods best represents the Super-Robot Strategy of subsection 2.3 (p. 8). First, the algorithm samples the configuration space $\mathbb{C}(A,W)$ of a robot for feasible poses. The set $\mathbb{C}(A,W)$ is the space created by the set of actuators of a robot [30, 14, 24, 21]. For an arm, in the context of section 2 (p. 2), each joint gets at least one dimension (the same number of dimensions as the dimension of the Lie group associated with the joint [30]), and the configuration space would be the space created by the juxtaposition of these different joints (for instance a product of intervals $(0,2\pi)$). A feasible pose is a pose that does not violate the structure of the arm: for instance, the arm does not intersect itself and does not collide with any obstacle present in W. After $\mathbb{C}(A,W)$ is sampled, a road-map is constructed. A road-map is a graph $(V(\mathbb{C}(A,W)), E(\mathbb{C}(A,W)))$ such that $v \in V(\mathbb{C}(A,W))$ corresponds to a feasible pose in the configuration space. There is an edge $e = (v,v') \in E(\mathbb{C}(A,W))$ iff a local, deterministic and fast path-planner is able to find a path-planning between the poses represented by v and v'.

3.1.1 Pseudo-Code

Here is the algorithm of the original PRM for the construction of the road-map [21, 14].

Procedure. *Stage 1: Construct Road-Map*

Input: W, A, $V \leftarrow \emptyset$, $E \leftarrow \emptyset$.

Output: V, E.

1: **while** (V, E) is not satisfactory **do**
2: $c \leftarrow$ A feasible sample from $\mathbb{C}(A, W)$. ◊ Determine whether feasible with W.
3: $V \leftarrow V \cup \{c\}$.
4: $N(c) \leftarrow$ a set of neighbors of c from V. ◊ Neighbor can be defined in any way.
5: **for** $c' \in N(c)$ **do**
6: **if** $(c', c) \notin E$ **then**
7: **if** the local deterministic planner finds a path between c' and c **then**
8: $E = E \cup \{(c, c')\}$.
9: **end if**
10: **end if**
11: **end for**
12: **end while**

The road-map is a graph that compresses a continuous set of path-plannings into a discrete number of path-planning sections. It is like a random partitioning of the configuration space. Each node represents all the poses that are "close" to it, given a certain distance function. Each edge represents how to go from one representative to the other.

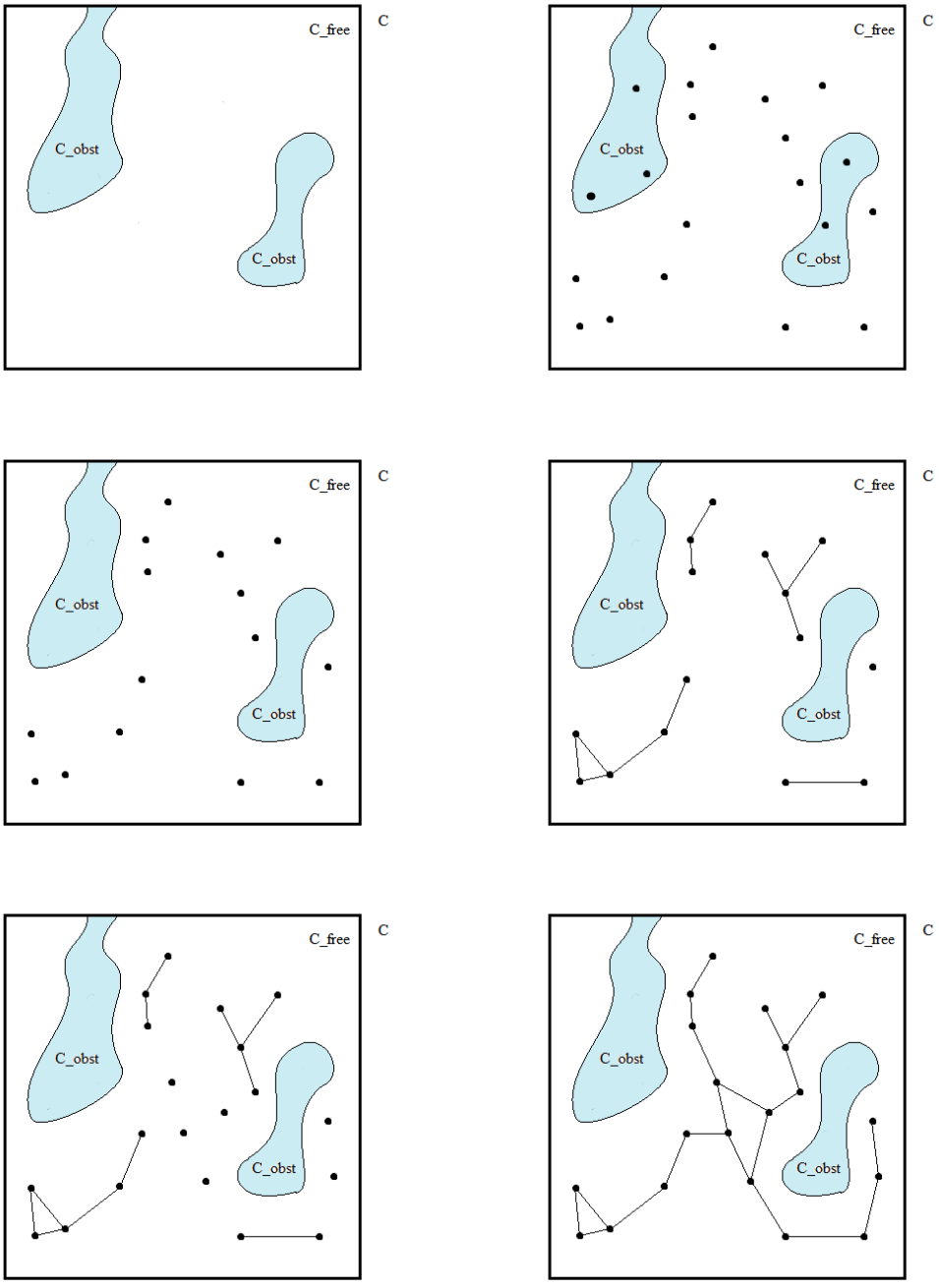

Figure 14: Construction of road-map on a two dimensional configuration space. There is no bound on the dimension of the configuration space, even if the robot lives in a 3 dimensional world.

The second phase is the query phase, where it is possible to use any graph path-planning method, such as Dijkstra, to find a path of between a start pose and an end pose. Once the road-map is constructed, and a graph path-planning method has given a sequence of pose representatives (vertices), then via the same (local, deterministic, and fast) path-planner, as used in the construction of the road-map, it is possible to recompute the precise sequence of poses and actuator actions.

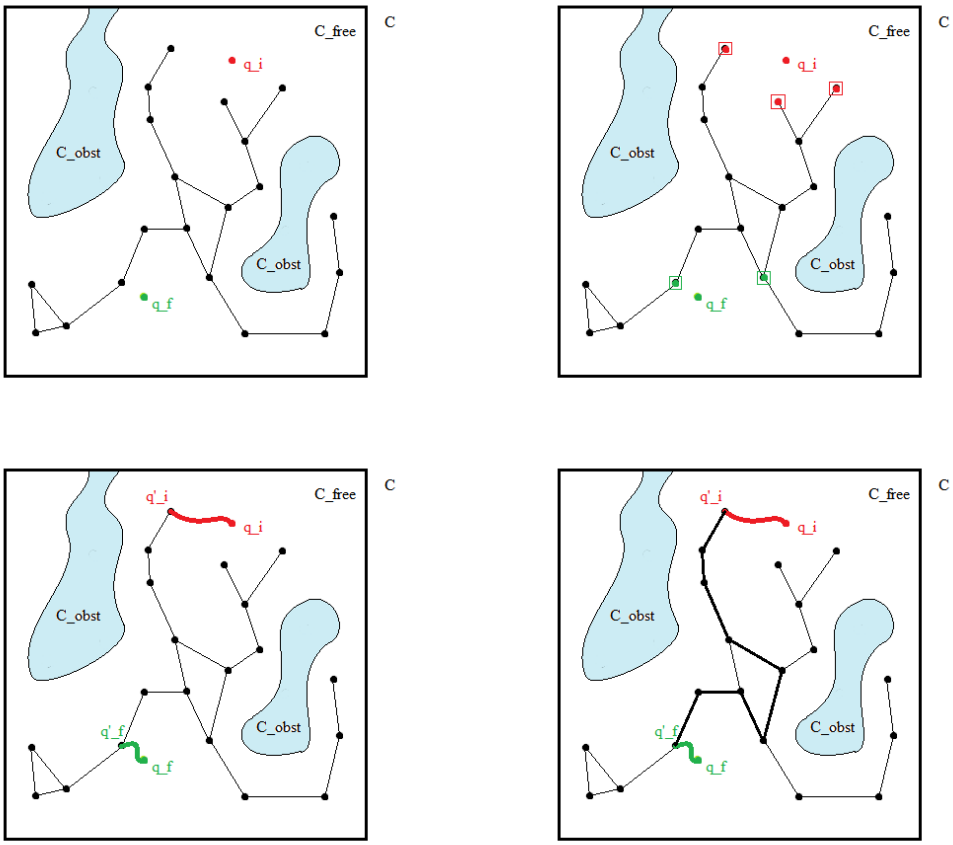

Figure 15: A possible query, with some smoothing.

3.1.2 Visualization

Probabilistic road-map methods are global methods by construction: they act on all of the joints of an arm, or on all of the degrees of freedom of a robot, at once. As such, the probabilistic road-map methods sit on one thread of execution.

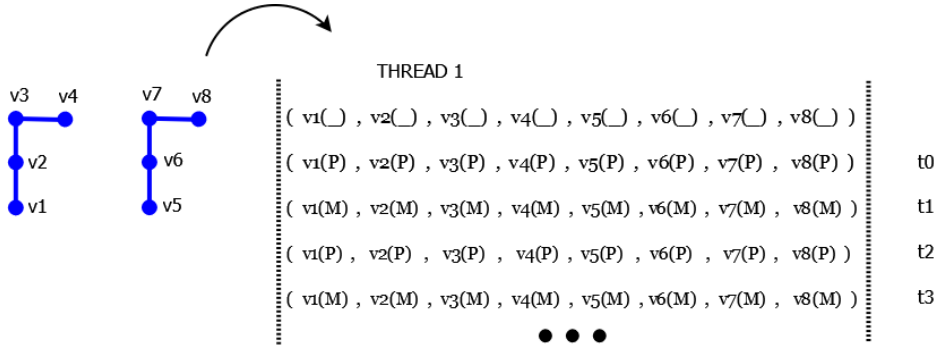

Figure 16: In the PRM method, usually everything is handled by one thread.

In particular, the PRM plans for all of the joints, hence for all of the limbs, at the same time. Thereupon, the limbs are all moved at the same time as well.

3.1.3 Variants

Several variants of the probabilistic road-maps have been developed. On one hand, one can use the same technique on W instead of $\mathbb{C}(A,W)$, i.e. directly on the world, using unitary robots, instead of the configuration space of a robot. It is also possible to use PRM on only a subset of the configuration space, and to use another method on another subset.

Moreover, the configuration space can become very high dimensional very fast. This causes the curse of dimensionality. As the number of dimensions go up, it becomes more and more difficult to sample the space. Therefore, several techniques have been developed about the sampling of $\mathbb{C}(A,W)$ [6, 44, 20].

Furthermore, a possible area of research is to work on the collision checking mechanism. As the points of the road-map are sampled from $\mathbb{C}(A,W)$, it is necessary to determine whether the point is feasible [38, 19]. This step is also an active area of research.

As per usual, it is interesting to create dynamic versions of existing algorithms. In a PRM-type method, the dynamicity occurs at the level of the road-map (and not at the level of the query). It is an interesting problem to understand how to update the road-map when a new obstacle appears or disappears, without recomputing the whole graph [4, 28, 25].

For a more thorough comparison of PRM-type methods, see [14, 24].

3.2 A*-Type Planners

Let $A = \{a\}$ be a unitary robot evolving in a world W. The A* method is a local planner. It evaluates successively each neighboring vertices in the world, and gives the most rewarding vertex as next position for the robot. Recall that the world $W = (V(W), E(W))$ is a graph, where $V(W)$ represent locations and $E(W)$ represent possible transitions between the locations. Planning a path for navigation can then be cast as a search problem on this graph [13]. Therefore, A* is in essence a local greedy search on W, toward a certain goal. However, this greedy search is guided by a heuristic function h, which greatly diminishes the amount of necessary computations, and helps to return in some situations optimal paths.

The following is from [12, 13]. A* plans a path from an initial state $s_{start} \in V(W)$ to a goal state $s_{goal} \in V(W)$. It stores an estimate $g(s)$ of the path cost from the initial state to each state s. Initially, $g(s) = 1$ for all states $s \in V(W)$. The algorithm begins by updating the path cost of the start state to be 0, then places this state onto a *priority queue* known as the *OPEN* list. Each element s in this queue is ordered according to the sum of its current path cost from the start, $g(s)$, and a heuristic estimate of its path cost to the goal, $h(s, s_{goal})$. The state with the minimum such sum is at the front of the priority queue. The heuristic $h(s, s_{goal})$ typically underestimates the cost of the optimal path from s to s_{goal} and is used to focus the search. The algorithm then pops the state s at the front of the queue and updates the cost of all states reachable from this state through a direct edge: if the cost of state s, $g(s)$, plus the cost of the edge between s and a neighboring state s', $c(s, s')$, is less than the current cost of state s', then the cost of s' is set to this new, lower value. If the cost of a neighboring state s' changes, it is placed on the *OPEN* list. The algorithm continues popping states off the queue until it pops off the goal state.

3.2.1 Pseudo-Code

Procedure. *Sub-Algorithm: Compute Shortest Path*

Input: *OPEN*, h, g

Output: *OPEN*

1: **while** $\text{argmin}_{s \in OPEN}(g(s) + h(s, s_{goal})) \neq s_{goal}$) **do**
2: remove state s from the front of *OPEN*.
3: **for** $s' \in Succ(s)$ **do**
4: **if** $g(s') > g(s) + c(s, s')$ **then**
5: $g(s') = g(s) + c(s, s')$.
6: insert s' into *OPEN* with value $(g(s') + h(s', s_{goal}))$.
7: **end if**
8: **end for**
9: **end while**

Procedure. *A**

Input: *OPEN*, h, g, s_{start}, s_{goal}.

Output: V, E.

1: **for** $s \in V(W)$ **do**
2: $g(s) \leftarrow \infty$.
3: **end for**
4: $g(s_{start}) \leftarrow 0$.
5: $OPEN \leftarrow \emptyset$.
6: insert s_{start} into *OPEN* with value $g(s_{start} + h(s_{start}, s_{goal}))$.
7: Compute Shortest Path.

3.2.2 Visualization

A* is a method that works on one unitary robot at a time, on one thread. It can be easily adapted to multiple unitary robots, but then one needs multiple threads. A whole arm can follow a unitary leader robot, as if a chain would be dragged by one of its ends. However, only one unitary robot can start moving at a point in time. This is close to the Critical-Section Strategy of section 2 (p. 2) , and already motivates the Split and Non-Deterministic Solution

of section 5 (p. 39). In other words, not more than one unitary leader robot may transition from a planning stage (color P) to a moving stage (color M).

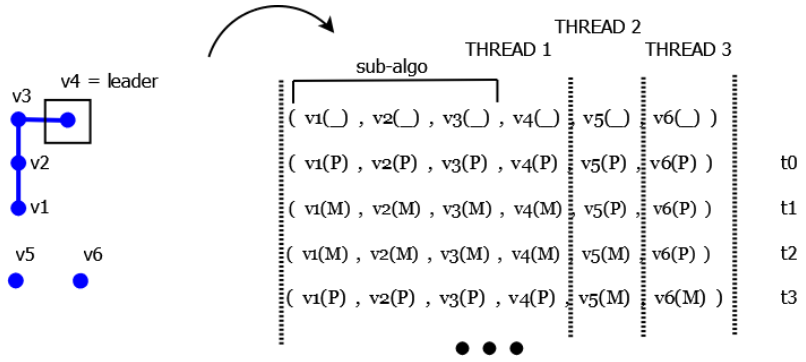

Figure 17: In A*, only one unitary robot is handled. Here vertices v_5 and v_6 represent unitary robots. Vertex v_4 is the leader of an arm (v_1, v_2, v_3, v_4). If v_4 is managed by an A*-type algorithm, then vertices v_1, v_2, v_3.

3.2.3 Variants

There are many algorithms that have almost the same characteristics as A*. Note that Greedy Search and Dijkstra Search are in this category: they both can plan for only a unitary robot on a space W represented by a graph, in a greedy fashion.

Different heuristics give rise to different versions of A*, each being better adapted to the problems they solve. The heuristic corresponds to the problem at hand [13].

Moreover, A* methods often have a forward and a backward version. The one presented above is a forward method, because it looks at the different vertices of W from s_{start} to s_{goal}. The backward version looks at the different vertices of W from s_{goal} to s_{start} [13].

As per usual, it is interesting to create dynamic versions of existing algorithms. In a A*-type method, the dynamicity occurs at the level of the computed path. As the world W changes, it is important to be able to recompute part of the path-planning that came close to the changes in W. However, the path-planning situated far enough from the change in W should not be affected, otherwise one would have to recompute the path-planning at any change of W [42, 12, 39, 11].

3.3 Combination of Path-Planners

Different path-planning methods, such as A* or PRM, but also others, can be combined together as long as there is a coordination model between them [26, 43]. Visualizing the coordi-

nations, each path-planning method has a certain type of activity transfer constraint on each vertex, from color P to color M. As long as all the methods and sub-methods used respect the transition constraints, it is possible to combine them using Paradigm, at the level of the threads of the different agents, i.e. different threads.

4 Paradigm Coordination Language

In the subsection 2.3 (p. 8), the need for a collaboration between different agents was motivated. For the description of such a collaboration, we may use the language Paradigm. The name Paradigm is an abbreviation for PARallelism, its Analysis, Design and Implementation by a General Method [15, 16, 17, 3]. Paradigm's subject of interest is parallelism as arising in collaborations, where agents have to collaborate in order to achieve their respective and possibly interleaved goals. The fundamental unit is the agent, whose possible behaviors can be precisely described by an STD or state transition diagram.

4.1 Definitions and Concepts

Definition 4.1 (STD)**.** An state-transition diagram (STD) of an agent is a directed graph[1] $Z = (V(Z), E(Z))$,

$$V(Z) = \{ \text{ set of states of the agent } \}, \tag{4.1}$$

$$E(Z) = \{ \text{ set of allowed transitions between states } \}. \tag{4.2}$$

The agent is always in exactly one state at time, and the transitions occur instantaneously. We think of an agent as spending some time in a state, and then transitioning to a next state if possible.

For two agents A_1 and A_2, with associated STD Z_1 and STD Z_2, it might be that some of the states are interdependent. For instance, if $z_1 \in Z_1$ and $z_2 \in Z_2$, it might be that A_1 cannot be in z_1 while A_2 is in z_2, etc. Out of this arises the need of a coordination language, such as Paradigm, ACP, CSP or CCS, responsible of establishing a collaborative discipline between different agents.

In Paradigm, the interdependence relations between agents are declared on an aggregated level of the phases and traps, not at the detailed level. Subgraphs of STDs are formed into *phases*, and the agents interact at the level of the phases. If $z_1, z_1' \in Z_1$ establish the same constraints on $z_2 \in Z_2$, then z_1 and z_1' can be put into a same phase of Z_1.

Definition 4.2 (Phase)**.** A *phase* S of an STD Z is a subgraph $S \subseteq Z$, i.e. $V(S) \subseteq V(Z)$ and $E(S) \subseteq E(Z)$.

[1] Paradigm can be readily generalized to multigraphs, i.e. directed graphs where there can be multiple edges between two vertices, see [bib].

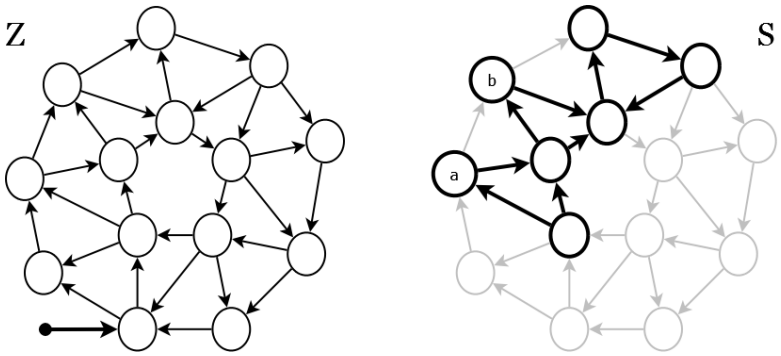

Figure 18: An arbitrary detailed STD on the left, with the initial state being indicated by the dotted arrow. On the left is an arbitrary phase restriction. Note that not all arrows between two vertices need to be included into phase, even if both vertices are in the phase, e.g. for vertices a and b.

As an example, depending on the construction of the phases, if agent A_1 has current phase $S_1 \subseteq Z_1$ and agent A_2 has current phase $S_2 \subseteq Z_2$, and $s_1 \in S_1$ and $s_2 \in S_2$ are mutually exclusive (A_1 cannot be in s_1 when $A2$ is in s_2 and vice-versa), then S_1 and S_2 have to be mutually exclusive as well (so that the above situation is impossible).

Write $\mathscr{C}$ for the whole undisciplined collaboration of agents $A_1,\ldots,A_n$, with associated STDs $Z_1,\ldots,Z_n$. The collaboration $\mathscr{C}$ corresponds to a product space of $Z_1,\ldots,Z_n$, in the sense of graph products [18].

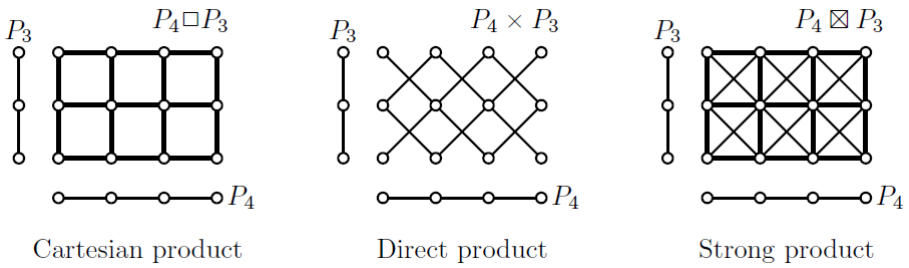

Figure 19: Simple examples of graph products.

Definition 4.3 (Graph Cartesian Product)**.** The Cartesian product of graphs G and H is a graph, denoted $G\square H$, whose vertex set is the Cartesian product $V(G) \times V(H)$ of sets. Two vertices (g,h) and (g',h') are adjacent precisely if $g = g'$ and $(h,h') \in E(H)$ or $(g,g') \in E(G)$

and $h = h'$. Thus

$$V(G\square H) = \{(g,h)|g \in V(G) \text{ and } h \in V(H)\}, \tag{4.3}$$

$$E(G\square H) = \{((g,h),(g',h'))|g = g', (h,h') \in E(H), \text{ or } (g,g') \in E(G), h = h'\}. \tag{4.4}$$

Definition 4.4 (Graph Direct Product)**.** The direct product of G and H is the graph, denoted as $G \times H$, whose vertex set is $V(G) \times V(H)$, and for which vertices (g,h) and (g',h') are adjacent precisely if $(g,g') \in E(G)$ and $(h,h') \in E(H)$. Thus

$$V(G \times H) = \{(g,h)|g \in V(G) \text{ and } h \in V(H)\}, \tag{4.5}$$

$$E(G \times H) = \{((g,h),(g',h'))|(g,g') \in E(G) \text{ and } (h,h') \in E(H)\}. \tag{4.6}$$

Definition 4.5 (Graph Strong Product)**.** The strong product of G and H is the graph denoted as $G \boxtimes H$, and defined by

$$V(G \boxtimes H) = \{(g,h)|g \in V(G) \text{ and } h \in V(H)\}, \tag{4.7}$$

$$E(G \boxtimes H) = E(G\square H) \cup E(G \times H). \tag{4.8}$$

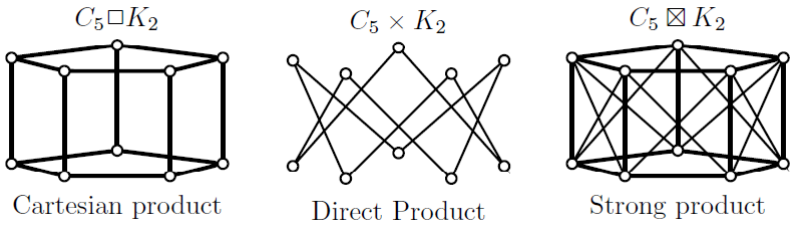

Figure 20: Other examples of graph products. Note that the definitions work with directed graphs as well.

Note that if agents A_1 and A_n have detailed STDs Z_1 and Z_2, then they are always in a certain state $z_1 \in Z_1$ and $z_2 \in Z_2$. Now in order to view these two agents as one system, the first reflex is to put them in a vector $\vec{z} = (z_1, z_2)$. The graph that describes $\vec{z}$ is given by $Z_1 \boxtimes Z_2$.

With these definitions in mind, the graph that represents the complete unconstrained collaboration $\mathscr{C}$ of agents $A_1, \ldots, A_n$ is

$$\mathscr{C} = Z_1 \boxtimes \cdots \boxtimes Z_n. \tag{4.9}$$

However, not all of C is feasible. For instance, if $z_1 \in Z_1$ and $z_2 \in Z_2$ are mutually exclusive, then the point $(z_1, z_2, \ldots) \in V(\mathscr{C})$ is not feasible, since by definition A_1 cannot be in z_1 while

A_2 is in z_2 and vice-versa. As this interdependence of agents is defined at the aggregated level, it useful to see what is a phase S_i of Z_i on $\mathscr{C}$, denoted $\mathscr{S}_i$.

$$\mathscr{S}_i = Z_1 \boxtimes \cdots \boxtimes Z_{i-1} \boxtimes S_i \boxtimes Z_{i+1} \boxtimes \cdots \boxtimes Z_n., \tag{4.10}$$

$$S_i = \text{proj}_i(\mathscr{S}_i). \tag{4.11}$$

Within a collaboration, agents behave simultaneously but asynchronously, for instance in a partially random fashion. It is sufficient to declare some coordination rules at the level of the phases, guiding the agents in a beneficial cooperation. Phases are a useful generalization: to make coordination rules more detailed, one can simply declare smaller phases. These coordination rules allow for phase changes based on sufficient progress made within the current phases of different agents. Progress within a phase is recorded whenever an agent enters a specific *absorbing* subset of states of a phase, i.e. a *trap*. Traps represent irreversible progress within a phase. However, an agent can (potentially) leave a trap of a previous phase once a new phase is imposed.

Definition 4.6 (Trap)**.** A *trap* T of a phase S is an absorbing subset of $V(S)$, i.e. $T \subseteq V(S)$ and for any $z \in T$ there is no $(z, z') \in E(S)$ such that $z' \notin T$. If $T = V(S)$, the trap is called triv (for trivial). Finally, we add a trap unknown, which corresponds to no knowledge of any trap entered.

Trap unknown acts as a trivial trap. Since trap messages will be sent from agent to protocol, see definition 4.12 (p. 32), trap unknown corresponds to no confirmation, being received by the protocol, about a phase change. Thus trap unknown corresponds to the protocol's lack of knowledge. Trap triv corresponds to a confirmation, being received by the protocol, about a phase change.

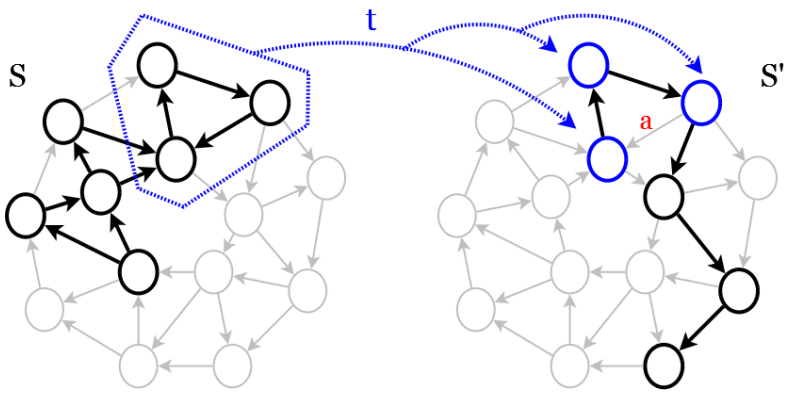

Figure 21: Two phases, on the left an aribitrary trap is specified in dashed blue, connecting the phase on the left with the phase on the right. Note that a trap is a subset of the set of states (vertices) and not of the set of actions (arrows), thus it is not a typo to not include action a (in red) into phase S'.

On $\mathscr{C}$, a trap T of a phase S_i of an STD Z_i of an agent A_i, denoted $\mathscr{T}$, is

$$\mathscr{T} = V(Z_1) \times \cdots \times V(Z_{i-1}) \times T_i \times V(Z_{i+1}) \times \cdots \times V(Z_n), \tag{4.12}$$

$$T_i = \text{proj}_i(\mathscr{T}). \tag{4.13}$$

A trap T of phase S of STD Z *connects* phase S to a phase S' of Z if $T \subseteq V(S')$. Such trap-based connectivity between two phases of Z is called a *phase transfer* $S \xrightarrow{T} S'$. For an agent with a given STD, the set of phases and traps, called a *partition*[2], determines the set of possible global behaviors, which are summarized in a directed graph called the *role* STD. Note that it is possible to have different roles on the same agent, describing different global behaviors corresponding to different point of views.

Definition 4.7 (Partition)**.** A *partition* $\pi = \{(S_k, T_{j_k}) | k \in K, j_k \in J_k\}$ of an STD Z, where K and J_k are non-empty index sets, is a set of pairs (S_k, T_{j_k}) consisting of a phase S_k of Z and of a trap T_{j_k} of S_k. Define $\mathbb{T}(S_k) = \{T_{j_k} | k \in K, j_k \in J_k\}$ to be the set of traps of S_k.

Note that a trap is an absorbing subset of a phase. $\mathbb{T}(S)$ does not represent all possible absorbing subsets of a phase, but only the ones chosen by the modeler. In particular, we set the trap triv and the trap unknown to be always in $\mathbb{T}(S)$.

[2]This definition of partition has nothing to do with a partition induced by an equivalence relation.

Definition 4.8 (Role)**.** A *role* $\pi(Z)$ of a corresponding partition $\pi = \{(S_i, T_{j_i}) | i \in I, j_i \in J_i\}$ is the directed graph $\pi(Z) = (V(\pi(Z)), E(\pi(Z)))$, with

$$V(\pi(Z)) = \{S_i | i \in I\}, \tag{4.14}$$

$$E(\pi(Z)) = \{(S_i, T_{j_i}, S') | S_i, S' \in V(\pi(Z)), T_{j_i} \in \mathbb{T}(S_i), \text{ s.t. } \exists S_i \xrightarrow{T_{j_i}} S', i \in I, j_i \in J_i\}. \tag{4.15}$$

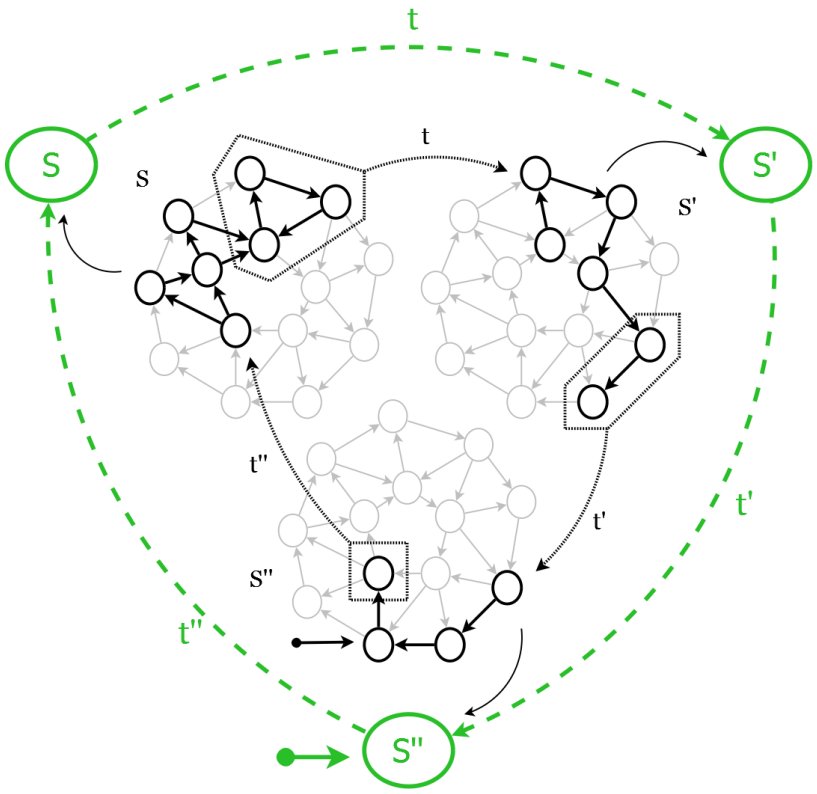

Figure 22: The three phases S, S' and S'' are collapsed into vertices. The arrows between them are induced by the traps. In green is the role STD of the partition consisting of phases $\{S, S', S''\}$ and traps $\{T, T', T''\}$. We have $T \in \mathbb{T}(S)$, $T' \in \mathbb{T}(S')$, $T'' \in \mathbb{T}(S'')$.

Note that for any phase S, it is always the case that $S \xrightarrow{\text{triv}} S$. We chose to omit this transition from the drawings. Moreover, if a phase transition is not used at all in a collaboration, then we choose to omit it as well from the drawings.

Definition 4.9 (Refined Role)**.** Let Z be the detailed STD of some agent A in a collaboration, with role $\pi(Z)$. Moreover, for a phase $S \in V(\pi(Z))$, define $\widehat{\mathbb{T}}(S)$ to be the set of all possible trap intersections of the phase S, with the rule that $T \cap \text{unknown} = T$, for any T. The refined role

is the directed graph $\widehat{\pi}(Z) = (V(\widehat{\pi}(Z)), E(\widehat{\pi}(Z)))$, with

$$V(\widehat{\pi}(Z)) = \left\{(S,T) \,|\, S \in V(\pi(Z)), T \in \widehat{\mathbb{T}}(S)\right\}, \tag{4.16}$$

$$E(\widehat{\pi}(Z)) = \{((S,T),(S',T'))|(S,T),(S',T') \in V(\widehat{\pi}(Z)), T \in \widehat{\mathbb{T}}(S), \tag{4.17}$$
$$T' = \left(\bigcap_{x \in \widehat{\mathbb{T}}(S'), T \subset x} x\right) \in \widehat{\mathbb{T}}(S')\}.$$

The vertices of this directed graph are pairs, where the first element is a phase S of the role $\pi(Z)$ and the second element is a refined trap T of the phase S, with possibly the trap unknown. All the traps of S are in $\widehat{\mathbb{T}}(S)$, as well as the trap unknown, and then all possible trap intersections. The edges of this directed graph are between two vertices (S,T) and (S',T') such that T' is the smallest trap in $\widehat{\mathbb{T}}(S')$ containing T (smallest in the sense of intersections). Say that an agent is, at the refined level, in trap $T' = T_1 \cap T_2$, then being in trap T' implies that the agent is both in T_1 and in T_2. Thus there is no loss of information.

We are finally ready to define a coordination rule and then a protocol. In Paradigm, the coordination rules, called *consistency rules*, are defined with the help phases and traps, on the aggregated level. A consistency rule weaves together, in a well-designed fashion, the different roles of the different agents. Therefore, it is natural to provide a definition that uses the product space viewpoint of the different roles.

Definition 4.10 (Consistency Rule)**.** Let $\{A_j | j = 1,\dots,n\}$ be the agents of the collaboration $\mathscr{C} = Z_1 \boxtimes \cdots \boxtimes Z_n$, where Z_j is the detailed STD of agent A_j. Let $\pi(Z_j)$ denote the directed graph representing the role for the corresponding agent, moreover suppose that each agent has only one role. Then a *consistency rule* is an edge (Ψ, Ψ') in the graph $\pi(\mathscr{C}) = \pi(Z_1) \boxtimes \cdots \boxtimes \pi(Z_n)$.

Definition 4.11 (Refined Consistency Rule)**.** Let $\{A_j | j = 1,\dots,n\}$ be the agents of the collaboration $\mathscr{C} = Z_1 \boxtimes \cdots \boxtimes Z_n$, where Z_j is the detailed STD of agent A_j. Let $\widehat{\pi}(Z_j)$ denote the directed graph representing the refined role for the corresponding agent, moreover suppose that each agent has only one refined role. Then a *refined consistency rule* is an edge $(\widehat{\Psi}, \widehat{\Psi'})$ in the graph $\widehat{\pi}(\mathscr{C}) = \widehat{\pi}(Z_1) \boxtimes \cdots \boxtimes \widehat{\pi}(Z_n)$.

Since a consistency rule is an edge linking two vertices, which correspond to two different aggregated states of all the agents present in the collaboration, one can concatenate these edges in order to obtain a path. Such a path characterizes one realization of the collaboration at the aggregated level, whereas the set of all possible paths characterizes the dynamics of the collaboration. This leads to the following definition.

Definition 4.12 (Protocol)**.** Let $A_1, \ldots, A_n$ be the agents of a collaboration $\mathscr{C}$. The *protocol* $\mathscr{P}$ is the set of all possible paths on the directed graph $\pi(\mathscr{C}) = \pi(Z_1) \boxtimes \cdots \boxtimes \pi(Z_n)$, where the *protocol steps* are the refined consistency rules, in view of definition 4.10.

In practice, for a collaboration $\mathscr{C}$, with agents $A_1, \ldots, A_n$, corresponding detailed STDs $Z_1, \ldots, Z_n$, roles $\pi(Z_1), \ldots, \pi(Z_n)$, a consistency rule $(\Psi, \Psi') \in E(\pi(\mathscr{C}))$ corresponds to phase transitions $(S_1 \xrightarrow{T_1} S_1', \ldots, S_1 \xrightarrow{T_1} S_1')$, with $S_i \in V(\pi(Z_i))$, $T_i \in E(\pi(Z_i))$, which we write as

$$* A_1(\pi(Z_1)) : S_1 \xrightarrow{T_1} S_1', \ldots, A_n(\pi(Z_n)) : S_n \xrightarrow{T_n} S_n',$$

Definition 4.13 (Refined Protocol)**.** Let $A_1, \ldots, A_n$ be the agents of a collaboration $\mathscr{C}$. The *refined protocol* $\widehat{\mathscr{P}}$ is the set of all possible paths on the directed graph $\widehat{\pi}(\mathscr{C}) = \widehat{\pi}(Z_1) \boxtimes \cdots \boxtimes \widehat{\pi}(Z_n)$, where the *refined protocol steps* are the consistency rules, in view of definition 4.11.

4.2 Explanation

4.2.1 Need for Strong Product

Let's justify the need of taking strong product of graphs, see definition 4.5 (p. 27). Two arbitrary agents A_1 and A_2, with STDs Z_1 and Z_2, can each only be in one state $z_1 \in Z_1$ and $z_2 \in Z_2$ at once. Now suppose that A_1 goes to a next state $z_1' \in Z_1$ and A_2 goes to a next state $z_2' \in Z_2$: either A_1 makes the transition first and then A_2 makes the transition, or vice-versa, or both agents make the transition simultaneously.

Suppose that A_1 makes the transition first, then putting the states of the agents in a vector, we have the transition $(z_1, z_2) \to (z_1', z_2)$ and then we have the transition $(z_1', z_2) \to (z_1', z_2')$. This corresponds precisely to the graph box product. Therefore, for agents $A_1, \ldots, A_n$ with detailed STDs $Z_1, \ldots, Z_n$, the graph box product $Z_1 \square \cdots \square Z_n$ corresponds to the situation where no two agents transition simultaneously. Following the same reasoning on the aggregated level, $\pi(Z_1) \square \cdots \square \pi(Z_n)$ corresponds to the situation where no two consistency rules are applied simultaneously.

Suppose now that A_1 makes the transition exactly at the same time as A_2, then we have the transition $(z_1, z_2) \to (z_1', z_2')$. This corresponds precisely to the graph direct product. Therefore, for agents $A_1, \ldots, A_n$, the graph direct product $Z_1 \times \cdots \times Z_n$ corresponds to the situation where all the agents transition simultaneously. Following the same reasoning on the aggregated level, $\pi(Z_1) \times \cdots \times \pi(Z_n)$ corresponds to the situation where each agent has been prescribed a phase transition.

Finally, $Z_1 \boxtimes \cdots \boxtimes Z_n$ corresponds to the situation the agents may transition in a sequence or simultaneously. Similarly, on the aggregated level, $\pi(Z_1) \boxtimes \cdots \boxtimes \pi(Z_n)$ corresponds to the situation where consistency rules may be applied in a sequence or simultaneously. Note that the exactly the same train of thought applies to refined consistency rules, with the description $\widehat{\pi}(\mathscr{C}) = \widehat{\pi}(Z_1) \boxtimes \cdots \boxtimes \widehat{\pi}(Z_n)$.

Note that if for any state $z \in V(Z_i)$ there would exist a self-arrow $(z,z) \in E(Z_i)$, then it would be the case that $Z_1 \times \cdots \times Z_n = Z_1 \boxtimes \cdots \boxtimes Z_n$, but it would still be the case that $Z_1 \square \cdots \square Z_n \neq Z_1 \boxtimes \cdots \boxtimes Z_n$.

4.2.2 Need for Refined Consistency Rules

Let's justify the need of having roles and refined roles, and consistency rules and refined consistency rules. In any Paradigm model, the agents $\{A_i\}_1^n$ have STDs $\{Z_i\}_1^n$ and have current phases $\{S_i\}_1^n$. The agent A_i is allowed to progress within the current phase S_i. After a certain amount of time spent in a current state $z_i \in S_i$, the agent will transition to a next state $z_i' \in S_i$, only if $(z_i, z_i') \in E(S_i)$. The current phase S_i has a set of associated traps $\mathbb{T}(S_i)$. When A_i enters a trap $T_i \in \mathbb{T}(S_i)$, a corresponding trap commit is fired. The information is relayed to the protocol $\mathscr{P}$, and this takes some time.

$$A_i \xrightarrow{\text{send trap commit } T_i} \mathscr{P}, \text{ arrives at time } t_{T_i}.$$

The protocol has a set of consistency rules, for instance say it has the following consistency rule.

$$* A_1(\pi(Z_1)) : S_1 \xrightarrow{T_1} S_1', \ldots, A_n(\pi(Z_n)) : S_n \xrightarrow{T_n} S_n',$$

which means that once the protocol will have received all the traps commits $T_1, \ldots, T_n$ from the agents $A_1, \ldots, A_n$, it will impose the new constraints $S_1', \ldots, S_n'$. This information is relayed back to the agents.

$$\begin{aligned} &\mathscr{P} \xrightarrow{\text{sends phase constraints } S_1'} A_1, \text{ arrives at time } t_{S_1}, \\ &\quad \vdots \\ &\mathscr{P} \xrightarrow{\text{sends phase constraints } S_n'} A_n, \text{ arrives at time } t_{S_n}. \end{aligned}$$

Once an agent A_i receives the new phase constraint, it is allowed to make a phase transition $S_i \xrightarrow{T_i} S_i'$, by simply continuing its progress at the detailed level, but now subject to the

constraint S'_i. This procedure can be seen as

$$A_1 \xrightarrow{\text{send trap commit } T_1} \mathscr{P}, \text{ at } \mathscr{P} \text{ at time } t_{T_1}, \tag{I}$$

$$\vdots$$

$$A_n \xrightarrow{\text{send trap commit } T_n} \mathscr{P}, \text{ at } \mathscr{P} \text{ at time } t_{T_n},$$

$$*A_1(\pi(Z_1)) : S_1 \xrightarrow{T_1} S'_1, \dots, A_n(\pi(Z_n)) : S_n \xrightarrow{T_n} S'_n, \text{ enabled at time } \max(t_{T_1}, \dots, t_{T_n}), \tag{II}$$

$$\mathscr{P} \xrightarrow{\text{sends phase constraint } S'_1} A_1, \text{ at } A_1 \text{ at time } t_{S'_1}, \tag{III}$$

$$\vdots$$

$$\mathscr{P} \xrightarrow{\text{sends phase constraint } S'_n} A_n, \text{ at } A_n \text{ at time } t_{S'_n}.$$

The consistency rule (II) accurately describes an action (arrow) that $\mathscr{P}$ can take. However, the decision of which consistency rule to pick might hinge on the order of arrival of the trap commits $T_1, \dots, T_n$, and maybe of some other trap commits $T'_1, \dots, T'_n$, since another rule could have been applied instead of rule (II). In other words, the decision of which consistency rule to choose depends, for instance, on $t_{T_1}, \dots, t_{T_n}, t_{T'_1}, \dots, t_{T'_n}$, and more generally on all possible trap commit arrivals.

The situation becomes more apparent when an agent A_i is in a phase S_i with overlapping traps. For instance, triv, $T_i \in \mathbb{T}(S_i)$, and triv $\cap T_i = T_i$. The implication is that A_i can be both in triv and in T_i simultaneously. This information is relayed to the protocol $\mathscr{P}$.

$$A_i \xrightarrow{\text{send trap commit } triv} \mathscr{P}, \text{ arrives at time } t_{\mathsf{triv}}$$

$$A_i \xrightarrow{\text{send trap commit } T_i} \mathscr{P}, \text{ arrives at time } t_{T_i}.$$

The protocol $\mathscr{P}$ may receive in any order the trap commits triv and T_i, i.e. $t_{\mathsf{triv}} < t_{T_i}$ or $t_{\mathsf{triv}} > t_{T_i}$ or $t_{\mathsf{triv}} = t_{T_i}$. Therefore, the states that can influence the decision making of $\mathscr{P}$ can be either unknown, for no information received yet, triv, T_i, or triv $\cap T_i$. Moreover this applies to each agent A_i of the collaboration. In our terminology, the states that can influence the decision making of $\mathscr{P}$, before (II) is applied, are $\widehat{\mathbb{T}}(S_1) \times \cdots \times \widehat{\mathbb{T}}(S_n)$, for instance $(\mathsf{triv}, \mathsf{triv}, \dots, \mathsf{triv}, T_n)$. Moreover, since it is ambiguous which triv belongs to which phase, the phases of the agents are also included, so that we write $((S_1, \mathsf{triv}), (S_2, \mathsf{triv}), \dots, (S_{n-1}, \mathsf{triv}), (S_n, T_n))$ instead.

A protocol $\mathscr{P}$ represents the rules (phase transitions) that apply to the agents, but the decision making process itself, based on the state of the collaboration at the global level, is

represented by the refined protocol $\widehat{\mathscr{P}}$. The protocol $\mathscr{P}$ represents how the phase impositions trickle down from a coordinator to the agents, and the refined protocol $\widehat{\mathscr{P}}$ represent how the trap commits bubble up from the agents to the coordinator.

In practice, we will only define consistency rules for a protocol $\mathscr{P}$, and assume the existence of a refined protocol $\widehat{\mathscr{P}}$, corresponding to $\mathscr{P}$.

4.2.3 Generalization To Multiple Roles

In the preceding subsections, Paradigm definitions were developed for the situation where each agent A_i, with detailed STD Z_i, has only one role $\pi(Z_i)$. For now, say that in a collaboration the agents are $A_1,\dots,A_n$, with detailed STDs $Z_1,\dots,Z_n$, and roles $\pi(Z_1),\dots,\pi(Z_n)$. Each agent is at all times in some state $z_i \in Z_i$ and in some current phase S_i, so that $z_i \in S_i$. Putting all the states of all the agents into a vector $\vec{z} = (z_1,\dots,z_n)$, the behavior of all the agents together lives on the collaboration $\mathscr{C} = Z_1 \boxtimes \cdots \boxtimes Z_n$. This collaboration is structured or disciplined with the help of a refined protocol, and the set of associated consistency rules. For instance, suppose that a new consistency rule has been applied and that now the agents $A_1,\dots,A_n$ are in phases $S_1,\dots,S_n$. The current phases establish temporary constraints on the agents, which means that now $\vec{z} \in \mathscr{S} = S_1 \boxtimes \cdots \boxtimes S_n$. This constraint will remain in place until a next consistency rule is applied, and the phase impositions arrive to the agents. Moreover, a consistency rule

$$* A_1(\pi(Z_1)) : S_1 \xrightarrow{T_1} S_1', \dots, A_n(\pi(Z_n)) : S_n \xrightarrow{T_n} S_n',$$

becomes applicable only when each agent has entered its respective trap, i.e. when $\vec{z} \in \mathscr{T} = T_1 \boxtimes \cdots \boxtimes T_n$. Thus we have a transition $S_1 \boxtimes \cdots \boxtimes S_n \xrightarrow{T_1 \boxtimes \cdots \boxtimes T_n} S_1' \boxtimes \cdots \boxtimes S_n'$. Writing $\mathscr{S}' = S_1' \boxtimes \cdots \boxtimes S_n'$, this reads as the transition $\mathscr{S} \xrightarrow{\mathscr{T}} \mathscr{S}'$.

In Paradigm, a natural generalization of this chapter is to consider agents that have multiple roles. Without loss of generality, say that each agent A_i has two roles $\pi_X(Z_i)$ and $\pi_Y(Z_i)$. That means that each A_i has two distinct dynamics in view of the collaboration. Since agents have phases imposed, where an phase is a vertex of a role, to have two roles concurrently affecting the same agent means that there are two phases currently imposed, constraining the agent in the collaboration. Say that A_i, in the view of role $\pi_X(Z_i)$, is constrained to the phase X_i, and in the view of role $\pi_Y(Z_i)$, constrained to Y_i. At the detailed level, if A_i is in state z_i, then $z_i \in X_i \cap Y_i$. Note that necessarily $X_i \cap Y_i \neq \emptyset$.

For instance, a person might be both a student and a friend, two point of views that both

impose some restrictions on the possible behaviors of a person, given a certain phase. For instance, a person might be, as a student, in a phase InClass, and as a friend, in a phase InRoom. In that case, the person's behavior will be restricted by both phases at the same time. This illustrates the fact that in a collaboration, a role is really like a point of view, usually with respect to some other agents.

In Paradigm, for one agent, only one role step per role is allowed in one consistency rule. One consistency rule can hold two role steps for the same agent, but on two different roles. For instance, if there are two agents A_1 and A_2, with detailed STDs Z_1 and Z_2, and corresponding two roles $\pi_X(Z_i)$ and $\pi_Y(Z_i)$, then the following is **not** allowed,

$$*A_1(\pi_X(Z_1)) : X_1 \xrightarrow{T_1} X_1', A_1(\pi_X(Z_1)) : X_1'' \xrightarrow{T_1'} X_1''', \text{(even if } X_1 \equiv X_1'').$$

However, the following **is** allowed.

$$*A_1(\pi_X(Z_1)) : X_1 \xrightarrow{T_1} X_1', A_1(\pi_Y(Z_1)) : Y_1 \xrightarrow{T_1'} Y_1', A_2(\pi_X(Z_2)) : X_2 \xrightarrow{T_2} X_2', A_2(\pi_Y(Z_1)) : Y_2 \xrightarrow{T_2'} Y_2'.$$

The definition 4.8 (p. 29) of roles, the definition 4.9 (p. 30) of refined roles, the definition 4.10 (p. 31) of consistency rules and the definition 4.11 (p. 31) of refined consistency rules, do not need to change. However, the definition 4.12 (p. 32) of protocols and the definition 4.13 (p. 32) of refined protocols can be extended to multiple roles.

Definition 4.14 (General Protocol)**.** Let $A_1, \ldots, A_n$ be the agents of a collaboration $\mathscr{C}$, with associated detailed STDs $Z_1, \ldots, Z_n$ and associated m roles $\{\pi_j(Z_i)\}_{j=1}^m$, where i is the index of the agent and j is the index of the role. To each role π_j there is a corresponding protocol $\mathscr{P}_j$, and the *general protocol* $\mathscr{P}$ is the set of all possible paths on the directed graph $\mathscr{P}_1 \boxtimes \cdots \boxtimes \mathscr{P}_m$, where a step is a vector $(\rho_1, \ldots, \rho_m)$ of consistency rules, in view of definition 4.10 (31), with ρ_i being a protocol step in $\mathscr{P}_i$, with some protocol steps being possibly trivial.

Note that it is the responsibility of the modeler to ensure that the different protocols are compatible. Badly designed protocols can easily lead to deadlocks or starvation in the collaboration.

Let's inspect a general protocol $\mathscr{P} = \mathscr{P}_1 \boxtimes \cdots \boxtimes \mathscr{P}_m$. For a role j, remember that a way of writing a consistency rule is as the transition $(S_1, \ldots, S_n) \xrightarrow{(T_1, \ldots, T_n)} (S_1', \ldots, S_n')$, with $S_i, S_i' \in \pi_j(Z_i)$, where i is the index of an agent. Let $S_i^j, S_i^{j\prime} \in \pi_j(Z_i)$, then an edge Ξ in $\mathscr{P} = \mathscr{P}_1 \boxtimes \cdots \boxtimes$

$\mathscr{P}_m$ acts as a stack of consistency rules:

$$\Xi = \begin{pmatrix} (S_1^1,\dots,S_n^1) \xrightarrow{(T_1^1,\dots,T_n^1)} (S_1^{1\prime},\dots,S_n^{1\prime}) \\ (S_1^2,\dots,S_n^2) \xrightarrow{(T_1^2,\dots,T_n^2)} (S_1^{2\prime},\dots,S_n^{2\prime}) \\ \dots \\ (S_1^m,\dots,S_n^m) \xrightarrow{(T_1^m,\dots,T_n^m)} (S_1^{m\prime},\dots,S_1^{m\prime}) \end{pmatrix}$$

Above, row j corresponds to a consistency rule belonging to protocol $\mathscr{P}_j$, and a "column" $(S_i^1, S_i^2, \dots, S_i^m)$ corresponds to all the phases being currently imposed on agent A_i. At the detailed level, this means that the state z_i of an agent A_i is constrained concurrently to all of the phases, i.e. $z_i \in \bigcap_{j=1}^m S_i^j$. As we argued a few paragraphs above, if $\vec{z} = (z_1,\dots,z_m)$ represents the detailed state of all the agents at once, before the application of the rule Ξ, $\vec{z}$ is constrained to $(\bigcap_{j=1}^m S_1^j) \boxtimes \dots \boxtimes (\bigcap_{j=1}^m S_n^j)$. As soon as each z_i belongs to the traps $\{T_i^j\}_{j=1}^m$, where j is the index of the role and i is the index of the agent, the consistency rule Ξ is ready to be applied. The new set of phase impositions, one per role, will constrain the agents to $(\bigcap_{j=1}^m S_1^{j\prime}) \boxtimes \dots \boxtimes (\bigcap_{j=1}^m S_n^{j\prime})$.

The problem as to how to design general protocol is an area of active research. We do not expand on refined general protocols, but the treatment is essentially the same.

Definition 4.15 (Refined General Protocol)**.** Let $A_1,\dots,A_n$ be the agents of a collaboration $\mathscr{C}$, with associated detailed STDs $Z_1,\dots,Z_n$ and associated m roles $\{\pi_j(Z_i)\}_{j=1}^m$, where i is the index of the agent and j is the index of the role. To each role π_j there is a corresponding refined protocol $\widehat{\mathscr{P}}_j$, and the *refined general protocol* $\mathscr{P}$ is the set of all possible paths on the directed graph $\widehat{\mathscr{P}}_1 \boxtimes \dots \boxtimes \widehat{\mathscr{P}}_m$, where a step is a vector $(\rho_1,\dots,\rho_m)$ of refined consistency rules, with ρ_i being a protocol step in $\mathscr{P}_i$, with some protocol steps being possibly trivial,in view of the definition 4.11 (p. 31).

4.2.4 Generalization to Multigraphs

Another generalization can be done at the detailed level. Until now, the definition of a state transition diagram Z (STD) is that of a directed graph. Everything in this chapter naturally expands to multigraphs, where the edges have an identity, i.e. directed graphs where there can be more than one arrow between two vertices, and the arrows are named. We give the definition of a multigraph here, but do not go further. Note that the definitions of graph products do not need to change.

Definition 4.16 (Multigraph)**.** A multigraph Z is a 4-tuple $Z = (V(Z), E(Z), s_Z, t_Z)$ with

- $V(Z)$ being a set of vertices,
- $E(Z)$ being a set of arrows,
- $s_Z : E(Z) \to V(Z)$, a function assigning to each arrow its source vertex,
- $t_Z : E(Z) \to V(Z)$, a function assigning to each edge its target vertex.

The reason why we avoid to talk of multigraphs is simply that in this work only directed graphs are needed, and that multigraphs would just add confusion to an already rather cumbersome notation. Finally, note that role graphs are actually multigraphs.

4.3 Paradigm Applied to the n-Arms Robot Problem

In the end of the first section, the need for a concurrent coordination mechanism was shown. In the following sections, different coordination mechanisms that solve the n-Arms Robot Problem are given in Paradigm. In the first section, three strategies to solve the n-Arms Robot Problem were presented: the Super-Robot Strategy, the Round-Robin Strategy and the Partial Reservation Strategy. Correspondingly, three Paradigm models are given: the Super-Robot Model, the Round-Robin Solution, and the Split and Non-Deterministic Solution. In particular, the agents are $\mathrm{Arm}_1, \ldots, \mathrm{Arm}_n$ and they all have the same detailed STD, corresponding to the given models. For the Super-Robot model and for the Round-Robin solution the detailed STD is given in Figure 23 (p. 39), and for the Split and Non-Deterministic solution the detailed STD is given in Figure 11 (p. 12).

5 Paradigm Coordination Models

5.1 The Super-Robot Model

In the Super-Robot Strategy from subsection 2.3 (p. 8), all the arms are considered as one super-robot or as one bigger and more complex arm. In the case of a super-robot there is no real need for coordination, since there is only one agent, i.e. the super-robot itself. This gives rise to the rather trivial Super-Robot Model, where there is only one detailed STD (with one phase, being the same as the detailed STD, the agent never leaving that phase).

Here is the detailed STD of the robot, see also Figure 11 (p. 12).

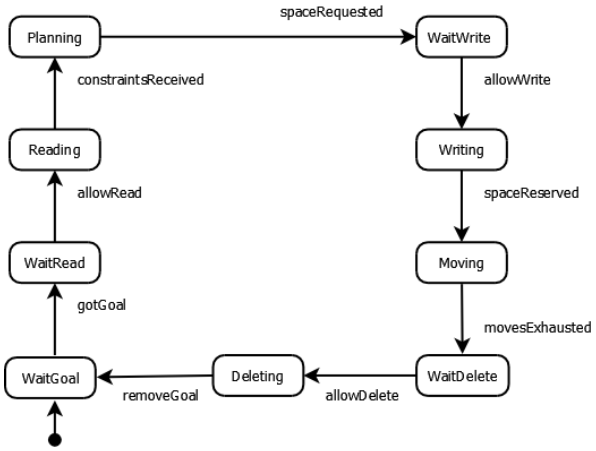

Figure 23: Note that the arrows update1!,update2! and update3! are not present.

The update arrows update1!, update2! and update3! have been disallowed. This signifies that the super-robot does not need to update its path-planning solution. From the first chapter, the modeling is focused on the robot itself, so that there are no dynamic obstacles in the world, outside of the robot. As soon as the robot is considered as made up of separate entities, there is a need for actual coordination, such as in the Round-Robin Model.

5.2 Critical-Section, Round-Robin Solution

The following Paradigm solution formalizes the Round-Robin Strategy of subsection 2.3 (p. 8). This is a collaboration that is an choreography, with participants Arm_1, ..., Arm_n and no conductor. The arms share a common, but otherwise empty world W. Since W is a finite resource, the arms cannot all move and plan at the same time: they compete for it. Reading simultaneously can also be risky, since reading exactly while another agent is writing can result in a data-race. The Round-Robin Strategy effectively lets only one arm to be alive: at any point in time, at most one arm can read, plan, reserve space, move or free space. However, different arms can request simultaneously to read, to plan, to reserve, to move and to free space. The arms Arm_i compete for a service turn, to be allowed by the other arms. On arm Arm_i exclusively has the service turn at a time. If an arm gets the service turn, then it is allowed to read, to plan, to reserve, to move and then to free space. Once that is done, another arm gets to read, to plan, to reserve, to move and then to free space.

5.2.1 Participant Arm Detailed STD

The detailed STD for the Round-Robin model is the same as for the Super-Robot model, see Figure 23 (p. 39).

The detailed STD of any Arm_i starts in state WaitGoal, in which state the arm is not competing for W. As soon as it reaches state WaitRead, by taking action gotGoal, the arm starts to wait for a service turn. By taking action allowRead, it reaches in order the states Reading, Planning, WaitWrite, Writing, Moving, WaitDelete, and then Deleting spending its service turn, until it takes action removeGoal. Using action removeGoal, it transitions into state WaitGoal, deleting the current objective and letting another arm take the service turn. After some progress, it reaches again state WaitRead.

5.2.2 Phases and Traps

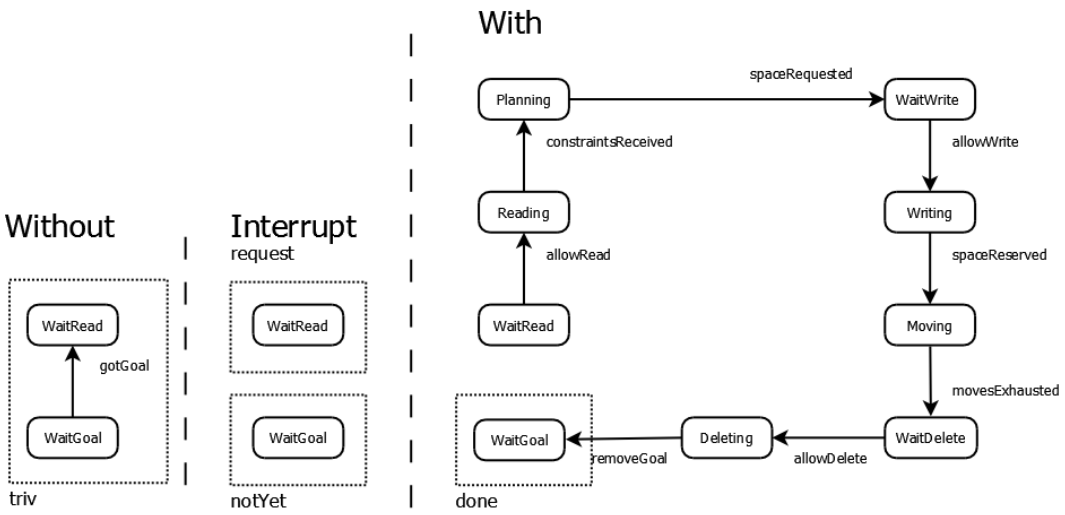

Figure 24: Partition for the role CS.

Phase Without specifies the behavioral freedom any Arm_i has when not having the service turn. Phase With specifies the behavioral freedom any Arm_i has when having the service turn. Phase Interrupt specifies the behavioral dynamic of Arm_i when it is being interrupted, being checked for a potential service turn. Note that in all phases, the arm is prevented updating, as actions update1!, update2! and update3! are not present.

Without Phase Without prohibits Arm_i to be in states Reading, Planning, WaitWrite, Writing, Moving, WaitDelete, and Deleting keeping Arm_i out of the critical behaviors. It also prevents the arm from updating, as actions update1!, update2! and update3! are not present in the detailed STD for this model. These actions are not needed since arms do not need to update in this model, which is a natural consequence of Reading, Planning, Writing, Moving, Deleting all being part of phase With.

Phase Without has trap triv, the trivial trap of it. It expresses trivial progress within the phase towards a next phase to be imposed. In other words, while being within phase Without and hence not having the service turn, every progress is good enough for Arm_i, even no progress, as the progress is no so much towards being fit for the service turn, but only towards being checked for being fit.

With Contrarily, phase With allows going to states Reading, Planning, WaitWrite, Writing, Moving, WaitDelete and Deleting.

Phase With has trap done: as Arm_i gets the service turn, it progresses towards finishing up the motion, which definitively happens in state WaitGoal. Trap done indicates that the arm no longer needs the service turn, and that the next arm is free to get the service turn.

Interrupt The phase Interrupt, intermediate between Without and With, is an interrupted form of Without, as action gotGoal cannot be taken (however being in state WaitRead is allowed).

Phase Interrupt has two traps notYet and request, indicating different kinds of progress towards being fit for the service turn: trap notYet for not enough progress yet and trap request for being fit indeed.

5.2.3 Role CS

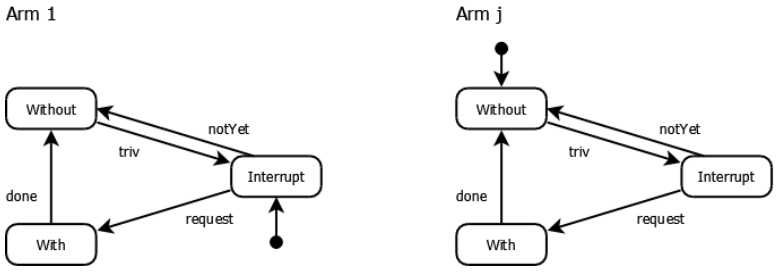

Figure 25: Partition for the role CS of the Round-Robin solution. Arm_1 starts in phase Interrupt, all the other arms starts in phase Without.

Note that the following is similar to the Split and Non-Deterministic solution, since parts of the mechanics are the same. To enable a smooth consecutive imposition of phases on a Arm_i, the following connectivity of traps is needed. Trap triv is connecting from Without to Interrupt, so phase transfer Without $\xrightarrow{\mathsf{triv}}$ Interrupt is well-defined. It actually means that once Without is constraining Arm_i, the phase transfer from Without to Interrupt can occur unconditionally, at any moment, as trap triv means that every progress within Without can be interrupted. The phase transfers Interrupt $\xrightarrow{\mathsf{notYet}}$ Without and Interrupt $\xrightarrow{\mathsf{request}}$ With are well-defined, and correspond to the two different connecting traps of Interrupt. Similarly, phase transfer With $\xrightarrow{\mathsf{done}}$ Without is well-defined and it only occurs after necessary progress has been made.

5.2.4 RoRo Protocol

$$*\mathsf{Arm}_i(\mathsf{CS}) : \mathsf{Interrupt} \xrightarrow{\mathsf{notYet}} \mathsf{Without}, \mathsf{Arm}_{(i+1 \mod n)+1}(\mathsf{CS}) : \mathsf{Without} \xrightarrow{\mathsf{triv}} \mathsf{Interrupt} \tag{5.1}$$

$$*\mathsf{Arm}_i(\mathsf{CS}) : \mathsf{Interrupt} \xrightarrow{\mathsf{request}} \mathsf{With} \tag{5.2}$$

$$*\mathsf{Arm}_i(\mathsf{CS}) : \mathsf{With} \xrightarrow{\mathsf{done}} \mathsf{Without}, \mathsf{Arm}_{(i+1 \mod n)+1}(\mathsf{CS}) : \mathsf{Without} \xrightarrow{\mathsf{triv}} \mathsf{Interrupt} \tag{5.3}$$

The above three rules define the Round-Robin collaboration. Agent Arm_1 starts in phase Interrupt and all the other agents start in phase Without. The First rule to be applied is rule 5.1, as at the very beginning the agent in Interrupt will have made no progress yet. The agent Arm_1 starts in phase Without and transitions right away to phase Interrupt, using the trivial trap triv.

In rules 5.1 and 5.2, the rules consider whether the arm is ready to get the service turn. The choice is being determined by the two disjoint connecting traps notYet and request of phase Interrupt of Arm_i. Depending on which trap has been entered, either rule 5.1 or 5.2 is applied. Rule 5.2 grants the service turn to Arm_i, transitioning the arm from phase Interrupt to phase With, whereas rule 5.1 denies the service turn, transitioning the arm from phase Interrupt to phase Without and transitioning the next arm from phase Without to phase Interrupt. Henceforth, rule 5.3 is eventually applied after rule 5.2, and rule 5.1 or rule 5.2 is eventually applied after rule 5.3.

After Arm_i has done enough progress in phase With to enter trap done, rule 5.3 is ready to be applied. Since trap done starts right after state Deleting, in state WaitGoal, the consistency rule will come into effect right after Arm_i has finished deleting. Actually, Reading, Planning, Writing, Moving, Deleting are what Arm_i needs the service turn for. As the consistency rule is being applied, it imposes the phase Without on arm Arm_i and the phase Interrupt on arm $\mathsf{Arm}_{(i+1 \mod n)+1}$. The consistency rule does not need to wait long for commits from these agents, since phase Without injects trivially into phase Interrupt using trap triv. Henceforth, rule 5.1 or rule 5.2 is eventually applied.

5.3 Critical-Section, Split and Non-Deterministic Solution

The following Paradigm solution formalizes the Partial Reservation Strategy of subsection 2.3 (p. 8). This is a collaboration that is a choreography, with participants Arm_1, …, Arm_n and no conductor. The arms share a common, but otherwise empty world W presented by

a matrix. Since W is a finite resource, the arms have to compete for it, especially they need to reserve some sections of space. This reservation translates as a writing operation in W. Access to W is a critical section. Arms get to write into W on a turn by turn basis. The arms simultaneously want to reserve space in W, which can be thought of as a service turn, to be given to exclusively one agent at a time, and agreed by the other agents, implementing a mutual exclusion for the critical section. Once an arm has reserved some space in W, the other arms need to update their set of obstacles and possibly recompute their path-planning.

5.3.1 Participant Arm Detailed STD

The detailed STD is the same in as the section Detailed Behavior Explanation, see Figure 11 (p. 12).

The following explanation is similar to the one for the Round-Robin Model, subsection 5.2 (p. 40), but it is not exactly the same. The detailed STD of any Arm_i starts in state WaitGoal, in which state the arm is not competing for W. It is still not competing for the critical section in states WaitRead, Reading and Planning. After some progress, by taking action spaceRequested, it reaches state WaitWrite: the arm now starts to compete for W by waiting for the critical section (service turn). By taking action allowWrite, it reaches the state Writing, where it spends its critical section, until it takes action spaceReserved. Using action spaceReserved, it transitions into state Moving, letting another arm take the critical section. The arm will need the critical section once more for state Deleting, to free the space it has reserved in Writing. Thus the arm requests the service turn again in state WaitDelete. By taking action allowDelete, it reaches the state Deleting, where it spends its critical section, until it takes action removeGoal. Then it lets another arm take the critical section.

After some progress, it reaches again state WaitWrite. If while being in states Reading, Planning or WaitWrite, another arm gets the critical section, then by taking action update1!, update2! or update3! Arm_i is forced to go back to WaitRead. Since WaitRead only leads to Reading, this forces an update for Arm_i.

5.3.2 Phases and Traps

Phase Without specifies the behavioral freedom any Arm_i has when not having the permission to enter the critical section. Phase With specifies the behavioral freedom any Arm_i has when having the permission to be in the critical section. Phase Interrupt and phase Update

specify the behavioral dynamic of Arm_i when it is being interrupted or when it is preemptively updating.

Without Phase Without prohibits Arm_i to be in state Writing or in state Deleting (both are a forms of writing), as well as to take the action allowWrite and the action allowDelete, thus keeping Arm_i out of the critical section. It also prevents the arm from updating, as actions update1!, update2! and update3! are prohibited as well.

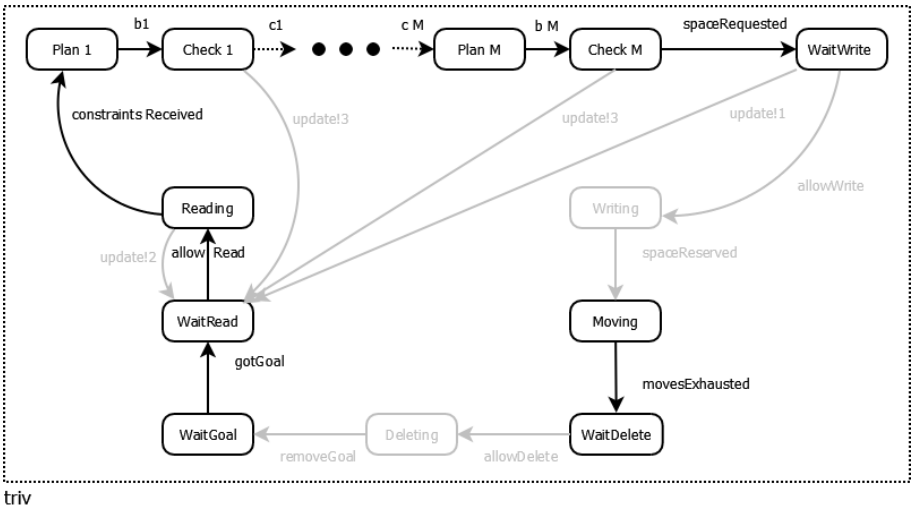

Figure 26: Phase Without with trap triv connecting to phase Interrupt.

Phase Without has trap triv, the trivial trap of it. It expresses trivial progress within the phase towards a next phase to be imposed. In other words, while being within phase Without, every progress is good enough for Arm_i, even no progress, as the progress is no so much towards being fit for the critical section, but only towards being checked for being fit.

Interrupt The phase Interrupt, intermediate between Without and With, is an interrupted form of Without, as action spaceRequested cannot be taken (however being in state WaitWrite is allowed).

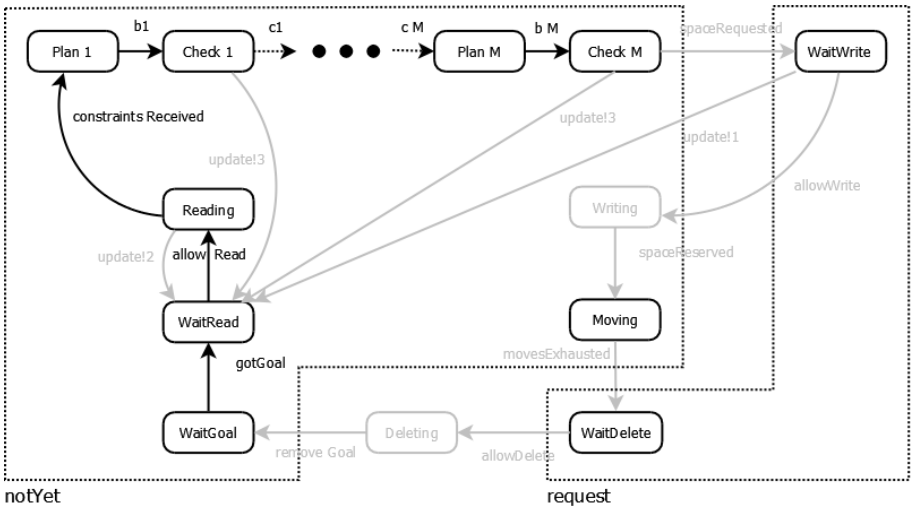

Figure 27: Phase Interrupt with trap notYet connecting to phase Without and the trap request connecting to phase With.

Phase Interrupt has two traps: notYet and the trap request, indicating different kinds of progress towards being fit for the critical section: trap notYet for not enough progress yet and trap request for being fit indeed.

With Contrarily, phase With allows going to state Writing and to state Deleting, staying there and leaving, all this only once.

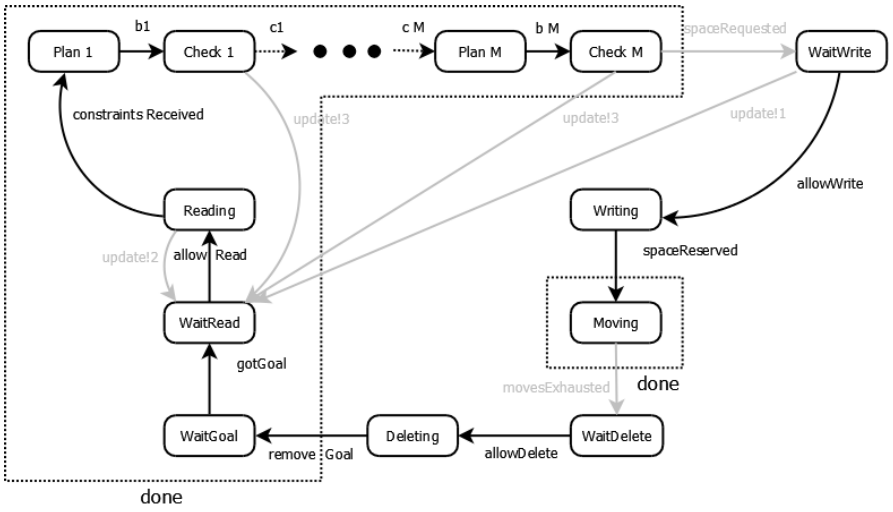

Figure 28: Phase With with the trap done connecting to phase Without.

Phase With has the trap done; as Arm_i is permitted to enter the critical section, it progresses towards giving up the privileged access to W as fast as possible. Trap done in-

dicates being fit for granting the permission to another Arm_j, $j \neq i$, thus withdrawing the permission from Arm_i.

Update Finally, phase Update is a preempted form of Without, where the arm is forced to make an update before continuing progress. In Update, actions spaceRequested, constraintsReceived, allowRead and allowWrite, spaceReserved are not allowed to be taken, but actions update1!, update2! and update3! are allowed, thus forcing an update of the arm. Indeed, these last three actions only lead to state WaitRead. The update then occurs later when the phase constraint Update is lifted and replaced by the less constraining phase Without.

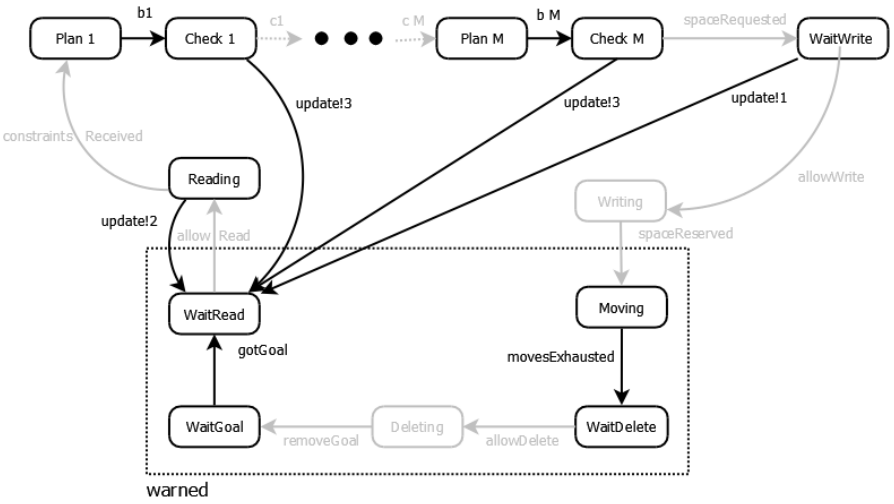

Figure 29: Phase Update with trap warned connecting to phase Without.

Phase Update has trap warned, expressing the guarantee that either Arm_i has not read W yet, if trap warned is entered through states Moving, WaitDelete, WaitGoal or WaitRead; or that it will read it again, if trap warned is entered coming from states Reading, Check i (part of Planning) or WaitWrite. The arm thus updates its knowledge of W.

5.3.3 Role Split-CS

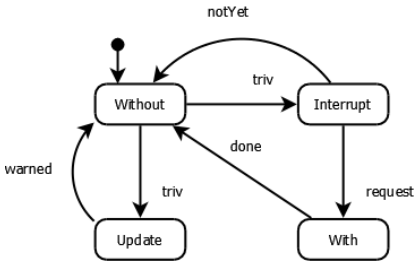

Figure 30: Role STD $\mathsf{Arm}_i(\mathsf{CS})$ of Arm_i.

To enable a smooth consecutive imposition of phases on a Arm_i, the following connectivity of traps is needed. Trap triv is connecting from Without to Interrupt, so phase transfer Without $\xrightarrow{\text{triv}}$ Interrupt is well-defined. It actually means that once Without is constraining Arm_i, the phase transfer from Without to Interrupt can occur unconditionally, at any moment, as trap triv means that every progress within Without can be interrupted. In exactly the same way, trap triv is connecting from Without to Update, so phase transfer Without $\xrightarrow{\text{triv}}$ Interrupt is well-defined. Similarly, two phase transfers, Interrupt $\xrightarrow{notYet}$ Without and Interrupt $\xrightarrow{request}$ With, are well-defined and correspond to the two different connecting traps of Interrupt. Phase transfer With $\xrightarrow{done}$ Without is well-defined and it only occurs after necessary progress has been made. Finally, phase transfer Update $\xrightarrow{warned}$ Without is well-defined and it only occurs once the corresponding arm is guaranteed to recheck the world W.

5.3.4 Split-CS Protocol

$$*\,\mathsf{Arm}_i(\mathsf{CS}) : \mathsf{Interrupt} \xrightarrow{\mathsf{notYet}} \mathsf{Without} \tag{5.4}$$

$$*\,\mathsf{Arm}_i(\mathsf{CS}) : \mathsf{Interrupt} \xrightarrow{\mathsf{request}} \mathsf{With} \tag{5.5}$$

$$*\,\mathsf{Arm}_i(\mathsf{CS}) : \mathsf{Without} \xrightarrow{\mathsf{triv}} \mathsf{Interrupt}, \mathsf{Arm}_{\forall j \neq i}(\mathsf{CS}) : \mathsf{Without} \xrightarrow{\mathsf{triv}} \mathsf{Without} \tag{5.6}$$

$$*\,\mathsf{Arm}_i(\mathsf{CS}) : \mathsf{With} \xrightarrow{\mathsf{done}} \mathsf{Without}, \mathsf{Arm}_{\forall j \neq i}(\mathsf{CS}) : \mathsf{Without} \xrightarrow{\mathsf{triv}} \mathsf{Update} \tag{5.7}$$

$$*\,\mathsf{Arm}_j(\mathsf{CS}) : \mathsf{Update} \xrightarrow{\mathsf{warned}} \mathsf{Without} \tag{5.8}$$

The above five rules define the Critical Section collaboration. The collaboration begins in rule 5.6, where all arms start in phase Without. Since the collaboration has just begun, the protocol will transition a certain Arm_i into phase Interrupt, to check whether the arm needs

the critical section. Most likely Arm_i will be in trap notYet, rather than in trap request, so that rule 5.4 is likely to be applied next. In phase Without, the arm can still get its objectives, then read W and finally plan a path; however, it wont be able to reserve the space for that path-planning.

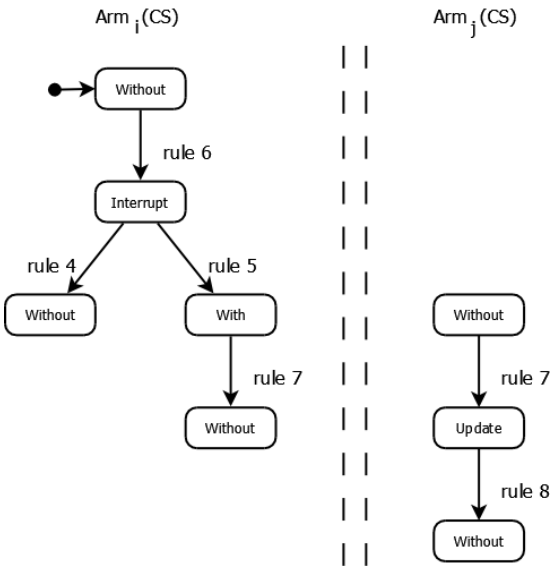

Figure 31: The two lanes, one for Arm_i and the other for Arm_j are executed in parallel.

In rules 5.4 and 5.5, the agent Arm_i has already been interrupted. If the agent is in trap notYet, that means the agent did not make enough progress yet to ask for the critical section, and then rule 5.4 is applied. If the agent is in trap request, that means the agent did make enough progress, and that the critical section is necessary for further progress, and then rule 5.5 is applied.

In rule 5.7, once the agent has used its critical section, it enters trap done of phase With, signifying that it is ready to relinquish the critical section. Since trap done starts right after state Writing or after state Deleting (which is a form of writing), the system will preempt the other agents right after Arm_i has finished writing into W (or deleting something from W). Actually that is all Arm_i needs the critical section for. As the system transitions Arm_i into state Without, it imposes the phase Update on all the arms Arm_j, for $j \neq i$. The system does not need to wait for commits from these agents, since phase Without injects trivially into phase Update using trap triv.

Subsequently, each arm in phase Update will be able to transition to phase Without using rule 5.8. As long as any agent is still in Update, rule 5.6 cannot be applied anew. Therefore, the system will wait for all the arms $\mathsf{Arm}_j, j \neq i$ to be updated, before interrupting another

agent $\mathrm{Arm}_{i'}$ for the critical section, using rule 5.6.

6 Probabilistic Take on Paradigm

This section justifies the section Simulation 7 (p. 56). A Paradigm model can be analyzed either at the global level or at the detailed level. Subsection 6.1 (p. 51) shows why a Paradigm model is hard to analyze at the global level. Subsection 6.2 (p. 55) shows how a Paradigm model would be analyzed at the detailed level and points to relevant literature.

6.1 Probabilistic Take at the Global Level

Any state of an STD might take different amounts of time to complete the task(s) assigned to it. For instance, this might be caused by hardware fluctuations or because the scheduler prioritizes another process. For this reason, most algorithms are given in big-O and a small-o running times, indicating the worst case scenario and the best case scenario. Another approach consist in viewing the running time of each state as a random variable.

In Paradigm, an agent A is always in a current state $z \in Z$, where Z is an associated detailed STD, and in a current phase $S \in \pi(Z)$, where $\pi(Z)$ is an associated role. If for any state in S, the running time is exponentially distributed, then the agent behaves as a *continuous-time Markov process*, or CTMP. A CTMP is a stochastic process with a discrete state space in which the time between transitions follows exponential distributions. Moreover, the lifetime before entering a trap t of a phase S follows a *phase-type distribution*, or PHD. In what follows, we first introduce CTMPs as much as it is necessary to talk about PHDs, from [31, 32, 5, 7], which we introduce subsequently.

6.1.1 Continuous-Time Markov Process

Let $V(S)$ denote a countable set of states (states of STD S), and let $\{A(t)\}_{t\geq 0}^{\infty}$ (A for agent) be a stochastic process with state space $V(S)$, with $|V(S)| \in \mathbb{N} \cup \{\infty\}$ being the size of the state space. Denote the states by an indexing $k = 1, \ldots, |V(S)|$, for states $s_1, \ldots, s_{|V(S)|}$. Moreover, assume bounded rates for the sojourn times associated to the states.

$\{A(t)\}_{t\geq 0}^{\infty}$ is a continuous-time Markov process, if it is characterized by the Markov property

$$P(A(t_{k+1}) = k+1 | A(t_k) = k, \ldots, A(t_0) = 0) = P(A(t_{k+1}) = k+1 | A(t_k) = k), \tag{6.1}$$

for any $0 \leq t_0 \leq t_1 \leq \ldots \leq t_k \leq t_{k+1}$ and $s_1, \ldots, s_k \in V(S)$.

The process is time homogeneous if for all $w \geq 0$, and there is a matrix $p_t(i, j)$, $1 \leq i, j \leq n$

such that

$$P(A(t+w) = s_j | A(w) = s_i) = P(A(t) = s_j | A(0) = s_i) = p_t(i,j). \tag{6.2}$$

The probability that an agent is in state s_i at time t is denoted by $p_t(i) = P(A(t) = s_i)$. Maybe it is not exactly known in which state the agent starts: denote by $\boldsymbol{p}(0) = (p_0(1), p_0(2), \ldots)$ the initial probability vector of the CTMP. The agent stays in state s_i for an exponentially distributed amount of time, before transitioning to another state. To represent these holding times, associate to state s_i the exponential random variable T_i with parameter $\lambda_i \in [0, \infty)$, with $\lambda_i = 0$ if i is an absorbing state.

$$P(T_i \leq t) = 1 - e^{-\lambda_i t}, \tag{6.3}$$

$$\tag{6.4}$$

The agent jumps into state s_j from state s_i with probability $p(i,j)$, which gives the transition rate $\lambda_{i,j} = p(i,j)\lambda_i$. Therefore λ_i gives the total rate for leaving the state s_i. This behavior can be summarized in the $n \times n$ infinitesimal generator matrix[3] $\boldsymbol{Q}$ [7], capturing the structure of the STD

$$\boldsymbol{Q}(i,j) = \begin{cases} -\lambda_i, & \text{if } i = j, \\ \lambda_{i,j}, & \text{if } i \neq j. \end{cases} \tag{6.5}$$

Note that the diagonal elements of $\boldsymbol{Q}$ are non-positive. If it is possible to transition from state s_i to state s_j, then $\boldsymbol{Q}(i,j) = \lambda_{i,j} > 0$, otherwise it is $\boldsymbol{Q}(i,j) = \lambda_{i,j} = 0$.

This treatment of Markov processes is sufficient to tackle Markov processes with absorbing states. A absorbing state is a state that once entered, cannot be left. Therefore, such a state is also a trap. For convenience, suppose that a trap T of an STD S consists of a single state s_T, and let $V(S) \backslash s_T = \{s_1, \ldots, s_n\}$ (so for now n is **not** the number of agents in the system). Order the state of the CTCM with the n non-absorbing states first, and the absorbing state s_T at the $n+1$-th position. Then the infinitesimal generator matrix $\boldsymbol{Q}$ can be written as

$$\boldsymbol{Q} = \left[\begin{array}{c|c} \boldsymbol{D}_0 & \boldsymbol{d}_1 \\ \hline \boldsymbol{0} & 0 \end{array}\right] \tag{6.6}$$

All the non-absorbing state transitions are in the $n \times n$ submatrix $\boldsymbol{D}_0$, describing transitions only between non-absorbing states. The $n \times 1$ matrix $\boldsymbol{d}_1$ describes the transitions from non-absorbing states to the absorbing state. The $1 \times n$ matrix $\boldsymbol{0}$ consists only of 0's, since there can

[3] This matrix is usually called infinitesimal generator matrix, generator matrix or transition rate matrix.

be no transition from the absorbing state to the non-absorbing state. The last 1×1 zero gives the transition rate out of the absorbing state, i.e. 0. See [31, 32, 5, 7] for more detail.

6.1.2 Phase-Type Distributions

Definition 6.1 (Phase-Type Distribution)**.** A *phase-type distribution* (PHD) is defined as the distribution of the hitting time X, i.e. to enter an absorbing state from the set of non-absorbing states of an absorbing continuous time Markov process $\{A(t)\}_{t\geq 0}^{\infty}$.

A PHD with n non-absorbing states is said to have order n. The background CTMP $\{A(t)\}_{t\geq 0}^{\infty}$ has an initial probability vector $(\boldsymbol{p}, p_T)$ where $\boldsymbol{p}$ is a $1 \times n$ vector and p_T is the initial probability of the absorbing state (trap) T. Since $\boldsymbol{Q}$ is the infinitesimal generator, and by the definition of the rates $\sum_j \boldsymbol{Q}(i,j) = 0$, it holds that

$$\boldsymbol{D}_0 \mathbf{1} + \boldsymbol{d}_1 = \mathbf{0}, \tag{6.7}$$

where $\mathbf{1}$ is a $n \times 1$ vector of 1's, and $\mathbf{0}$ is a $n \times 1$ vector of 0's.

The agent starts in an arbitrary state in $V(S)$, with the probabilities being given by $(\boldsymbol{p}, p_T)$, therefore $\boldsymbol{p}\mathbf{1} + p_T = 1$. In particular, we do not assume that $p_T = 0$. That means we do not assume that $A(t)$ is strictly positive: if the agent begins already in a trap, then $A(t) = 0$.

Since $\{A(t)\}_{t\geq 0}^{\infty}$ is a CTMP, the holding time of each state s_i is exponentially distributed with parameter $-\boldsymbol{D}_0(i,i)$, as

$$\boldsymbol{D}_0(i,i) = -\left(\sum_{i \neq j} \boldsymbol{D}_0(i,j) + \boldsymbol{d}_1(i)\right), \tag{6.8}$$

since the rate to leave s_i is the rate of transfer into the trap, i.e. $\boldsymbol{d}_1(i)$, plus the sum of the rates of transfers into any other state, i.e. $\sum_{j\neq i} \boldsymbol{D}_0(i,j)$.

The hitting time X describing the time until absorption is said to be of phase-type with representation $(\boldsymbol{p}, \boldsymbol{D}_0)$. The vector $\boldsymbol{d}_1$ and the probability p_T are implicitly determined by $\boldsymbol{D}_0$ and $\boldsymbol{p}$.

The distribution function of the hitting time of X is given by

$$F_X(t) = 1 - \boldsymbol{p} e^{\boldsymbol{D}_0 t} \mathbf{1}, \text{ for } t \geq 0. \tag{6.9}$$

Finally, as the state $s_1, \ldots, s_n$ are non-absorbing, the matrix $\boldsymbol{D}_0$ is invertible [7], and the value $(-\boldsymbol{D}_0)^{-1}(i,j)$ gives the expected total time spent in state s_j before absorption, given

that the initial state is s_i. The ith moment μ_i of a PHD is derived from the moment matrix $M = (-\boldsymbol{D}_0)^{-1}$ as

$$\mu_i = E[X^i] = i!\boldsymbol{p}\boldsymbol{M}^i\mathbf{1}. \tag{6.10}$$

In practice, given a matrix $\boldsymbol{D}_0$, it is surprisingly easy to find the exact form of the associated distributions and densities. Simply use a symbolic algebra software package, such as MATLAB, which are powerful enough to give the analytic form of matrix exponentials [29].

6.1.3 Communication Between Agents

Recall from section 4 (p. 25), that agents enter traps as they progress in their respective current phases. As they arrive into the traps, which serve as commits, this information is relayed to the protocol. A consistency rule is ready to be applied only when the corresponding traps have been entered, and the information has been relayed. For simplicity, suppose that we have two agents A_1 and A_2 in a collaboration. They have initial detailed STD Z and phases $\{S_1, S_2, S_3\}$ and traps $\{T_1', T_1, T_2\}$ such that $S_1 \xrightarrow{T_1'} S_1$, $S_1 \xrightarrow{T_1} S_2$ and $S_2 \xrightarrow{T_2} S_3$, therefore they have the role $\pi(Z) = \{\{S_1, S_2, S_3\}, \{T_1', T_1, T_2\}\}$. Moreover, say the collaboration has two consistency rules

$$*A_1(\pi(Z)) : S_1 \xrightarrow{T_1} S_2,\ A_2(\pi(Z)) : S_1 \xrightarrow{T_1'} S_1, \tag{I}$$

and

$$*A_1(\pi(Z)) : S_2 \xrightarrow{T_2} S_3,\ A_2(\pi(Z)) : S_1 \xrightarrow{T_1'} S_1. \tag{II}$$

and suppose that rule (I) is applied first, and rule (II) is applied second.

Say the rule (I) is applied first, agent A_1 has to reach trap T_1 and agent A_2 has to reach trap T_2. This information has to be relayed to the protocol, and this trap passing takes a certain amount of time. Thereupon, the rule can be processed, which takes a certain amount of time, and then phase S_2 will be sent to agent A_1 and phase S_1 will be sent to agent A_2. This phase passing again takes a certain amount of time.

More precisely, to each state in Z, associate an independent exponentially distributed holding time. These distributions do not have to be identical from state to state. With this assumption, the agents behave as CTMPs, with state space Z. As a consequence, as we mentioned in subsection 6.1.2 (p. 53), the time taken by A_i to arrive into a certain trap T follows a PHD, call this H_i^T. This random variable has representation $\boldsymbol{p}$ and $\boldsymbol{D_0}$, where $\boldsymbol{p}$ is the initial probability vector and $\boldsymbol{D_0}$ is the infinitesimal generator.

$$F_{H_i^T}(t) = 1 - \boldsymbol{p}e^{\boldsymbol{D}_0 t}\mathbf{1}, \text{ for } t \geq 0.$$

When can rule 1 be applied? Only when the trap commit T_1 from agent A_1 and the trap commit T_1' from agent A_2 have both reached the protocol. If δ_i is the traveling time from agent A_i to the protocol, this happens exactly at

$$\max(H_1^{T_1} + \delta_1, H_2^{T_1'} + \delta_2),$$

moreover, say the processing of a rule takes δ_P, so that gives

$$\max(H_1^{T_1} + \delta_1, H_2^{T_1'} + \delta_2) + \delta_P$$

and finally, to send a phase imposition from the protocol to agent A_i takes δ_i', which gives the phase arrival-time

$$\Delta_i^{(I)} = \max(H_1^{T_1} + \delta_1, H_2^{T_1'} + \delta_2) + \delta_3 + \delta_i' \tag{6.11}$$

for agent A_i and rule (I). The reasoning is similar for rule (II).

Even with simple phases, for instance with Coxian or an Erlang structures, it is becomes very rapidly a hard task to compute anything out at the global level. The situation is exacerbated for general Paradigm models where an agent might have multiple roles and the phases can have arbitrary structures.

6.2 Probabilistic Take at the Detailed Level

An alternate way of analyzing concurrency models is to concatenate the different detailed STDs of the agents involved. For agents $A_1, \ldots, A_n$, with detailed STDs $Z_1, \ldots, Z_n$ and with detailed protocol $\widehat{\mathscr{P}}$, consider the whole system on $\mathscr{C} = Z_1 \boxtimes \cdots \boxtimes Z_n \boxtimes \widehat{\mathscr{P}}$. The evolution of the concurrent model is described by the evolution of the vector $(z_1, \ldots, z_n, \rho) \in \mathscr{C}$.

With probability one, only one agent transitions at a time:

$$(z_1, \ldots, z_i, \ldots, z_n, \rho) \xrightarrow{\lambda_{z_i \to z_i'}} (z_1, \ldots, z_i', \ldots, z_n, \rho)$$

where $\lambda_{z_i \to z_i'}$ is the rate associated with the arrow from state $z_i \in Z_i$ to state $z_i' \in Z_i$. This includes the transitions for the protocol, for instance

$$(z_1, \ldots, z_i, \ldots, z_n, \rho) \xrightarrow{\lambda_{\rho \to \rho'}} (z_1, \ldots, z_i', \ldots, z_n, \rho'),$$

where $\rho, \rho' \in \widehat{\mathscr{P}}$.

7 Simulation of Simple Paradigm Models

Instead of doing a probabilistic analysis on either the global level or the detailed level, we construct simulations of the Paradigm models, and proceed to analyze them statistically.

7.0.1 Setup

In this section, we show how to simulate a Paradigm model with one role: the steps are given in pseudo-code and the Round-Robin Solution is used to exemplify how the simulation works. MATLAB code is given in the appendix for the Split and Non-Deterministic Solution.

We shy away from most optimization tricks on purpose: our concern is a low-level and transparent implementation, from which statistics can be extracted with ease. As such, the simulation is script-like, on one thread and without protection. We show how to simulate the collaboration, but not how to implement the states of the detailed STDs themselves. For instance, state Planning in Figure 11 (p. 12) is not implemented, but rather it is left as a dummy state. These dummy states take random amounts of time to execute, characterized by exponential distributions, as in section 6 (p. 51). When a word is written in this `font`, it means that it is a data-structure in the implementation.

Let $A_1, \ldots, A_n$ be the agents in a collaboration, with detailed STDs $Z_1, \ldots, Z_n$, and roles $\pi(Z_1), \ldots, \pi(Z_n)$. Let the agents have current phases $S_1, \ldots, S_n, S_i \in V(\pi(Z_i))$ and initial states $z_1, \ldots, z_n, z_i \in S_i$ (initial state in phase S_i). We concentrate on how to make one consistency rule transition

$$* A_1(\pi(Z_1)) : S_1 \xrightarrow{T_1} S_1', \ldots, A_J(\pi(Z_J)) : S_J \xrightarrow{T_J} S_J',$$

for a subset $\{1, \ldots, J\}$ of the agents, and for some traps $T_1, \ldots, T_J$.

7.0.2 Assumptions

In the following treatment, we make the following simplifying assumptions.

• All the agents are always active, no agent is at rest.

• All the agents have the same detailed STD structure.

• Every agent has only one role.

• Every phase has at most three traps, trap unknown, trap triv and maybe another non-trivial trap.

• In every phase, an agent can visit a state only once, unless that state is a leaf state in the phase.

• In every phase, an agent can transition from a state to only one other state.

• For a given set of current phases, only one consistency rule can be applied.

7.0.3 Constant Variables

For the whole simulation, we have the following global constants.

• A vector `order` that indexes all the possible states of a detailed STD.

• A vector `mu` that indexes the parameters of the exponential distributions, corresponding to the states in `order`.

• A vector `agent2protocol` that indexes the parameters of the exponential distributions representing the traveling times, for the trap commits from an agent to the protocol.

• A parameter `protocol` for the exponential distribution representing the processing time of consistency rules, assuming that each processing is described by the same parameter.

• A vector `protocol2agent` that indexes the parameters of the exponential distributions representing the traveling times, for the phase impositions from the protocol to the agents.

7.0.4 Simulation Data-Structures

The time span between the application of two consecutive consistency rules is referred to as a *cycle*. Within each cycle, we keep a bookkeeping on the following for each agent.

• A vector `times`, where $\texttt{times}^c$ is the starting time for cycle c.

• A table `states`, where $\texttt{states}_i^c$ is the initial state for agent A_i in cycle c.

• A table `phases`, where $\texttt{phases}_i^c$ is the phase name for agent A_i in cycle c.

• A table `changes`, where $\texttt{changes}_i^c$ is 1 if agent A_i receives phase in cycle c and 0 otherwise.

• A vector `rules`, where $\texttt{rules}^c$ represents the rule that was applied in cycle c (this data-structure is for debugging purposes).

• A vector `focus`, where $\texttt{focus}^c$ represents the value of i (present in the consistency rules) in cycle c.

• A table `tableAdjW`, where $\texttt{tableAdjW}_i^c$ is $\texttt{Adj}_{\texttt{W}}^c(S_i)$. Adjacency matrices $\texttt{Adj}_{\texttt{W}}^c(S_i)$, where $\texttt{Adj}_{\texttt{W}}^c(S_i)_{j,j}$ is the execution time of state j for agent A_i, in cycle c and in phase S_i.

• A table `tableAdjΣ`, where $\texttt{tableAjdΣ}_i^c$ is $\texttt{Adj}_{\Sigma}^c(S_i)$. Adjacency matrices $\texttt{Adj}_{\Sigma}^c(S_i)$, where $\texttt{Adj}_{\Sigma}^c(S_i)_{j,j}$ is the completion time of state j for agent A_i, in cycle c and in phase S_i.

• A table `tableTrap`, where $\texttt{tableTrap}_k^c = ph\&trp_{A_k}$ is a matrix, for agent A_k and cycle c, with $ph\&tr_{A_k,i,j}$ being the time at which trap j of phase i was entered, based on $\texttt{Adj}_{\Sigma}^c(S_i)$

and $\texttt{Adj}_W^c(S_i)$.

A phase S is an STD, thus a directed graph $(V(S), E(S))$. One usual representation of directed graphs is in terms of an *adjacency matrix* $Adj(S)$. The set of traps of a phase will make it's appearance later. For states $x, y \in V(S)$, provides the respective positions of x and y in the adjacency matrix, say i and j. Then

$$Ajd(S)_{i,j} = -2, \text{ if } (x,y) \in E(S)$$

$$Ajd(S)_{i,i} = -1, \text{ if } x \in V(S)$$

$$Ajd(S)_{i,i} = 0, \text{ otherwise.}$$

There is a "-2" in $Adj(S)_{i,j}$ if there is an arrow from the state at position i to the state at position j, a "-1" to mark whether a state is present or not, and "0" otherwise. The sign of the digits in the adjacency matrix can be used for some logical checks.

7.1 Strategy

Simulating a Paradigm model amounts to some intricate bookkeeping. Looking at the outputs of each method and starting in cycle c, method Time-Work computes the processing times $\texttt{tableAdjW}_i^c$, for each A_i in phase S_i. Thereupon, the method Completion-Time computes the finishing times of the states into $\texttt{tableAjd}\Sigma_i^c$, for each A_i in phase S_i cycle c. The methods Time-Work, Completion-Time, Trap-Commit, Try-Rule, and Weave are submethods of the method Simulation.

In method Simulation, time adjustments are computed into $\texttt{tableAjd}\Sigma_i^c$ for each A_i. Then method Trap-Commit computes the trap commit arrivals into $\texttt{tableTrap}_i^c$ for each A_i. Back in method Simulation, times adjustments are computed into $\texttt{tableTrap}_i^c$ for each A_i, to take into account the trap traveling times, based on the parameters `agent2protocol`.

Subsequently, method Try-Rule computes the following:

- which next rule will be applied, stored into $\texttt{rules}^c$;
- which is the next agent focus (if any), stored into $\texttt{focus}^{c+1}$;
- which are the phase impositions, stored into $\texttt{phases}^{c+1}$;
- which agents have new phase impositions (boolean), stored into $\texttt{changes}^{c+1}$;
- what is the time at which the new rule is enabled, i.e. when all the necessary trap commits reach the protocol; stored into $\texttt{times}^{c+1}$.

Again, method Simulation adds a processing time into $\texttt{times}^{c+1}$, based on the parameter `protocol`. Subsequently, method Simulation adds the travel times of the new phase imposi-

tions into $\texttt{times}^{c+1}$, based on the parameters $\texttt{protocol2agent}$. Method Weave determines the initial states in which each agent will begin in the next turn, and stores this information into $\texttt{states}^{c+1}$.

7.1.1 Using the Memoryless Property

The memoryless property is used between the methods Weave and Work-Time. The method Weave outputs for an agent the initial state of the next phase. The method Work-Time computes the holding times for each states. If the agent reaches a leaf state z_i in the previous phase S_i, then the agent restarts the execution of that leaf state until a new phase S_i' arrives at time $t_{S_i'}$. In particular, with probability 1, this means that $t_{S_i'} < t_{z_i}$, where t_{z_i} is the completion time of state z_i. Instead of setting the holding time of z_i in the new phase S_i' to be $t_{z_i} - t_{S_i'}$, we simply recompute it in method Work-Time. This is acceptable because of the memoryless property.

7.2 Method Work-Time

7.2.1 Aim

The behavior of each agent A_i is recorded as it progresses within one phase S_i and one cycle c, so within one adjacency matrix $\texttt{Adj}_W^c(S_i)$, being initially a copy of $Adj(S_i)$, as if the agent would be allowed to progress to the end of its phase. A consistency rule might be applied before the agent reaches the end of its current phase. Given an initial state $z_i \in S_i$, the simulation produces exponential holding times w_i for each state, where the parameter is given by $\texttt{mu}_{z_i}$.

7.2.2 Pseudo-Code

Procedure. *Work-Time for A_i in cycle c.*

Input: $\texttt{mu}$, $\texttt{states}_i^c$, $\texttt{Adj}_W^c(S_i)$.

Output: $\texttt{Adj}_W^c(S_i)$.

1: $z_i \leftarrow \texttt{states}_i^c$.

2: **if** $\texttt{Adj}_W^c(S_i)_{z_i,z_i} = -1$ **then**

3: $\quad \texttt{Adj}_W^c(S_i)_{z_i,z_i} \leftarrow X \sim \text{Exp}(\texttt{mu}_{z_i})$. $\quad \Diamond$ X is a sample from $\text{Exp}(\texttt{mu}_{z_i})$.

4: **end if**

5: $z_i \leftarrow$ next state in S_i. $\quad \Diamond$ deduced from $\texttt{Adj}_W^c$

6: **while** z_i is a non-empty state in S_i **do**

7: $\quad$ **if** $\texttt{Adj}_W^c(S_i)_{z_i,z_i} = -1$ **then**

8: $\quad\quad \texttt{Adj}_W^c(S_i)_{z_i,z_i} \leftarrow X \sim \text{Exp}(\texttt{mu}_{z_i})$. $\quad \Diamond$ X is a sample from $\text{Exp}(\texttt{mu}_{z_i})$.

9: $\quad$ **end if**

10: $\quad z_i \leftarrow$ next state in S_i. $\quad \Diamond$ deduced from $\texttt{Adj}_W^c$

11: **end while**

The above procedure populates the diagonal of the adjacency matrix with the respective holding times of the associated states, on lines 3 and 8. The agent starts in state $\texttt{states}_i^c$, on line 1, and then iterates through the phase on lines 5 and 10. The while-loop stops when there is no possible next state, i.e. when the next state is empty. Note that $\texttt{Adj}_W^c(S_i)_{z_i,z_i} = -1$ if the state z_i has not been reached by the agent, and $\texttt{Adj}_W^c(S_i)_{z_i,z_i} \geq 0$ otherwise, as each $w_i \geq 0$.

7.3 Method Completion-Time

7.3.1 Aim

Now for agent A_i and current phase S_i, we have the holding times of each state that can be visited. However, this does not directly give the time at which the agent finishes the execution of a certain state (the time at which a certain state is entered), information that is necessary for the simulation. Let $\texttt{Adj}_\Sigma^c(S_i)$, be a copy of $Adj(S_i)$. Given an initial state $\texttt{states}_i^c \in S_i$ and initial time $t = 0$, $\texttt{Adj}_\Sigma^c(S_i)_{z_i,z_i}$ will represent the cumulative time taken by agent A_i to finish a certain state z_i, in cycle c. Method Simulation will adjust for the initial times.

7.3.2 Pseudo-Code

Procedure. *Completion Times for A_i in cycle c.*

Input: $\texttt{states}_i^c$, $\texttt{Adj}_W^c(S_i)$.

Output: $\texttt{Adj}_\Sigma^c(S_i)$.

1: $z_i \leftarrow \texttt{states}_i^c$.

2: $t \leftarrow \texttt{times}^c$.

3: $\Sigma \leftarrow \texttt{Adj}_W^c(S_i)_{z_i,z_i}$.

4: $\texttt{Adj}_\Sigma^c(S_i)_{z_i,z_i} \leftarrow (\Sigma + t)$.

5: $z_i \leftarrow$ next state in S_i. ◊ deduced from $\texttt{Adj}_\Sigma^c(S_i)$

6: **while** z_i is a non-empty state in S_i **do**

7: $\quad \Sigma_2 \leftarrow \texttt{Adj}_W^c(S_i)_{z_i,z_i}$.

8: $\quad \Sigma \leftarrow (\Sigma + \Sigma_2)$.

9: $\quad \texttt{Adj}_\Sigma^c(S_i)_{z_i,z_i} \leftarrow (\Sigma + t)$.

10: $\quad z_i \leftarrow$ next state in S_i. ◊ deduced from $\texttt{Adj}_\Sigma^c(S_i)$

11: **end while**

The above procedure populates the diagonal of the adjacency matrix with the respective cumulative times of the associated states, on lines 4 and 9. The agent starts in state z_i, on line 1, and then iterates through the phase on lines 5 and 10. The while-loop stops when there is no possible next state, i.e. when the next state is empty. Note that $\texttt{Adj}_\Sigma^c(S_i)_{z_i,z_i} = -1$ if the state z_i has not been reached by the agent, and $\texttt{Adj}_\Sigma^c(S_i)_{z_i,z_i} \geq 0$ otherwise: the value on that diagonal is the total time taken by the agent to finish state z_i, given that it started in state $\texttt{states}_i^c$ at time t.

Note that for any agent A_i, with an initial state $\texttt{states}_i^c$ and phase S_i, one can use $\texttt{Adj}_W^c(S_i)$ and $\texttt{Adj}_\Sigma^c(S_i)$ to find time at which the agent has reached a certain state z_i. Namely, $\texttt{Adj}_\Sigma^c(S_i)_{z_i,z_i} - \texttt{Adj}_W^c(S_i)_{z_i,z_i}$ gives the time at which a state z_i has been reached (after time adjustment).

7.4 Method Trap-Commit

7.4.1 Aim

A simulation of a Paradigm model records the evolution of the different agents as time progresses. In each cycle, progress is marked by the application of a consistency rule. This

progress is being stored in the data-structures introduced earlier.

Consistency rules feed on trap commits, which depend on arrival times, so that for any agent A_k, it is sufficient to know $\texttt{Adj}^c_\Sigma(S_k)$ (after time adjustment). We implement a procedure that uses a bookkeeping based on decision-tables. Since there is nothing really interesting about this method, only the code is given in the appendix. The procedure takes as inputs $\texttt{phase}^c_k$, $\texttt{state}^c_k$, $\texttt{Adj}^c_{\texttt{W}}(S_k)$, and $\texttt{Adj}^c_\Sigma(S_k)$, and that outputs a matrix $ph\&tr_k$ of size $|V(\pi(Z_k))| \times |E(\pi(Z_k))|$, with

$$ph\&tr_k(i,j) = \text{ time at which trap } j\text{, in phase } i\text{, was entered by agent } A_k,$$

and $ph\&tr_k(i,j) = -1$ otherwise. For instance, $ph\&tr_k(\mathsf{Interrupt}, \mathsf{triv}) = -1$ means that phase Interrupt has not been entered by agent A_k this cycle, whereas $ph\&tr_k(\mathsf{Interrupt}, \mathsf{notYet}) = 5{,}14$ would mean that agent A_k has entered trap notYet in phase Interrupt at time 5,14. $ph\&tr_k(\mathsf{Without}, \mathsf{triv}) = 0$ would mean that agent A_k has entered trap triv in phase Without at time 0.

7.4.2 Input/Output

For an agent A_k, the inputs are some encoding of the trap/phase structure of $\pi(Z_k)$, $\texttt{phases}^c_k$, $\texttt{states}^c_k$, $\texttt{Adj}^c_{\texttt{W}}(S_k)$, and $\texttt{Adj}^c_\Sigma(S_k)$. The output is $ph\&tr_k$.

7.5 Method Try-Rule

7.5.1 Aim

First compute $ph\&tr_{A_1}, \ldots, ph\&tr_{A_n}$: from this information, it is possible to decide which consistency rule will be applied next, and when it will be possible to apply the consistency rule. From the setup, remember that we assume that only one consistency rule is applicable at a time. Therefore, it is sufficient to try each consistency rule, using `tableTrap` on each $ph\&tr_{A_1}, \ldots, ph\&tr_{A_n}$, and catch the one that works.

7.5.2 Pseudo-Code

Procedure. *Try Rule* $\quad *A_1(\pi(Z_1)) : S_1 \xrightarrow{T_1} S'_1, \ldots, A_J(\pi(Z_J)) : S_J \xrightarrow{T_J} S'_J.$

Input: $\texttt{states}^c, \texttt{phases}^c, \texttt{tableTrap}^c, \texttt{focus}^c.$

Output: $\texttt{phases}^{c+1}, \texttt{times}^{c+1}, \texttt{changes}^{c+1}, \texttt{focus}^{c+1}, \texttt{rule}^c.$

1: $\texttt{times}^{c+1} \leftarrow -1.$ ◊ Value -1 is useful for debugging.

2: $\texttt{phases}^{c+1} \leftarrow \texttt{phases}^c.$

3: $\texttt{changes}^{c+1} \leftarrow 0.$ ◊ No change by default, for any agent.

4: $\texttt{focus}^{c+1} \leftarrow \texttt{focus}^c.$

5: $\texttt{rule}^{c+1} \leftarrow -1.$

6: **if** $(ph\&tr_1(S_1, T_1) \geq 0) \wedge \cdots \wedge (ph\&tr_J(S_J, T_J) \geq 0)$ **then**

7: $\quad \texttt{times}^{c+1} \leftarrow \max(ph\&tr_1(S_1, T_1), \ldots, ph\&tr_J(S_J, T_J)).$

8: $\quad \texttt{phases}_1^{c+1} \leftarrow S'_1, \ldots, \texttt{phases}_J^{c+1} \leftarrow S'_J.$

9: $\quad \texttt{changes}_1^{c+1} \leftarrow 1, \ldots, \texttt{changes}_J^{c+1} \leftarrow 1.$ ◊ Boolean 1 signifies a phase change.

10: $\quad \texttt{focus}^{c+1} \leftarrow$ Update($\texttt{focus}^c$). ◊ Update is an small helper method given by the designer.

11: $\quad \texttt{rule}^{c+1} \leftarrow RULE\ ID.$

12: **end if**

For instance, the second rule of the Round-Robin collaboration takes the following form.

Procedure. $Try*Arm_i(RoRo) : Interrupt \xrightarrow{notYet} Without, Arm_{(i+1 \mod n)+1}(RoRo) : Without \xrightarrow{triv} Interrupt$.

Input: $\texttt{states}^c$, $\texttt{phases}^c$, $\texttt{tableTrap}^c$, $\texttt{focus}^c$.

Output: $\texttt{phases}^{c+1}$, $\texttt{times}^{c+1}$, $\texttt{changes}^{c+1}$, $\texttt{focus}^c$, $\texttt{rule}^c$.

1: $\texttt{times}^{c+1} \leftarrow -1$.

2: $\texttt{phases}^{c+1} \leftarrow \texttt{phases}^c$.

3: $\texttt{changes}^{c+1} \leftarrow 0$.

4: $\texttt{rule}^{c+1} \leftarrow -1$.

5: $\texttt{focus}^{c+1} \leftarrow \texttt{focus}^c$.

6: **if** $(ph\&tr_i(\mathsf{Interrupt}, \mathsf{notYet}) \geq 0) \wedge (ph\&tr_{(i+1 \mod n)+1}(\mathsf{Without}, \mathsf{triv}) \geq 0)$ **then**

7: $\quad \texttt{times}^{c+1} \leftarrow \max(ph\&tr_i(\mathsf{Interrupt}, \mathsf{notYet}), ph\&tr_{(i+1 \mod n)+1}(\mathsf{Without}, \mathsf{triv}))$.

8: $\quad \texttt{phases}_i^{c+1} \leftarrow \mathsf{Without}, \texttt{phases}_{(i+1 \mod n)+1}^{c+1} \leftarrow \mathsf{Interrupt}$.

9: $\quad \texttt{changes}_i^{c+1} \leftarrow 1, \texttt{changes}_{(i+1 \mod n)+1}^{c+1} \leftarrow 1$.

10: $\quad \texttt{focus}^{c+1} \leftarrow (i+1 \mod n)$.

11: $\quad \texttt{rule}^{c+1} \leftarrow 2$.

12: **end if**

The variable $\texttt{times}^{c+1}$ stores the time at which the new phase impositions arrive to the agents in the collaboration. The vector $\texttt{phases}^{c+1}$ stores the set of phases in the new cycle. The final stage in the simulation of a Paradigm model consists in weaving together the old phases and the new phases of the agents. The vector $\texttt{changes}^{c+1}$ stores which agents get a new phase in the next cycle $c+1$. Into $\texttt{rule}^c$ is stored the tag of the rule that has just been applied (so still in cycle c).

7.6 Method Weave

7.6.1 Aim

The arrival into a trap does not prevent an agent to continue its progress within its current phase. This is one of the advantages of a Paradigm model, the phase/trap construction allows for a certain level of progress smoothness. There are several cases.

• An agent has not yet reached a leaf state in its current phase when a new phase imposition arrives.

• An agent has reached a leaf state in its current phase when a new phase imposition arrives,

but it has not yet finished the execution of the leaf state.

• An agent has reached a leaf state in its current phase when a new phase imposition arrives, but it has already finished the execution of the leaf state.

In Paradigm modeling, an agent is never let to rest. If an agent has reached a leaf state in its phase, then it will simply restart the execution of that state. Specifically, suppose that agent A_i arrives into state $z_i \in S_i$, which is a leaf state, and finished the execution at time t_{z_i}. Suppose that there is a phase transition $S_i \xrightarrow{T_i} S_i'$, and that the new phase imposition arrives at time $t_{S_i'}$. If $t_{z_i} < t_{S_i'}$ then the agent restarts z_i until $t_{z_i} \geq t_{S_i'}$. See subsection [using memoryless property]

For an agent A_i, let the time at which the execution of z_i is completed be t_{z_i}, and the time at which the phase S_i' is imposed be $t_{S_i'}$. Because the holding times in each state are memoryless, there is no need to store the value $t_{z_i} - t_{S_i}$. The memoryless property means that probabilistically, the value $t_{z_i} - t_{S_i}$, given t_{S_i}, is the same as the distribution of the holding time of z_i. As such, if $t_{z_i} \geq t_{S_i'}$, then in the next cycle, it is acceptable to recompute t_{z_i}.

7.6.2 Pseudo-Code

Procedure. *Weave for agent A_i in cycle c.*

Input: $\texttt{states}_i^c$, $\texttt{times}_i^c$, and $\texttt{tableAdj}\Sigma_i^c$.

Output: $\texttt{states}_i^{c+1}$.

1: $t_{S_i'} \leftarrow \texttt{times}_i^c$.
2: $z_i \leftarrow \texttt{states}_i^c$.
3: $t_{z_i} \leftarrow \texttt{Adj}_\Sigma^c(S_i)_{z_i,z_i}$.
4: **if** z_i is a leaf in S_i **then**
5: $\quad \texttt{states}_i^{c+1} \leftarrow z_i$.
6: **end if**
7: **while** z_i is not a leaf in S_i **do**
8: $\quad$ **if** $t_{z_i} \geq t_{S_i'}$ **then**
9: $\quad\quad \texttt{states}_i^{c+1} \leftarrow z_i$.
10: $\quad\quad$ **return**
11: $\quad$ **else**
12: $\quad\quad z_i \leftarrow$ next state in S_i.
13: $\quad\quad t_{z_i} \leftarrow \texttt{Adj}_\Sigma^c(S_i)_{z_i,z_i}$.
14: $\quad\quad$ **if** z_i is a leaf in S_i **then**
15: $\quad\quad\quad \texttt{states}_i^{c+1} \leftarrow z_i$.
16: $\quad\quad$ **end if**
17: $\quad$ **end if**
18: **end while**

From lines 1 to 3, the weaving procedure is initialized. On line 1, t_{S_i} represents the time

at which the new phase imposition S_i has arrived. On line 2, z_i represents the initial state z_i of agent A_i in cycle c, thus the initial state in phase S_i. On line 3, t_{z_i} presents the time stamp at which agent A_i completes the execution of state z_i.

From lines 4 to 18, the weaving is done. The aim of the procedure is to find next initial state of agent A_i, in the next phase S_i', in the next cycle $c+1$.

First, from line 4 to 6, the algorithm asks whether the current state z_i is a leaf in the phase: if it is, then necessarily z_i will be the initial state of the next phase.

Note that the code from lines 12 to 16 is identical with the code from line 2 to 6. As the algorithm iterates over the phase, searching for the initial state of the next phase, if the finishing time t_{z_i} of the state z_i is less than t_{S_i}, then the algorithm steps one state forward and asks whether the new state z_i is a leaf in the phase, just as on lines 2 to 6. Otherwise, from lines 8 to 10, if the finishing time t_{z_i} of the state z_i is greater than t_{S_i}, then it must be the first one to be so, and thus it will be the initial state of the next phase.

7.7 Method Simulation

7.7.1 Aim

Using the assumption that no agent is at rest, the agent will keep restarting leaf states in a phase, until a new phase is imposed. The distribution of the time taken to finish the work-load associated to a state given the arrival-time of a phase is the same as the distribution of the time taken to finish the work-load, because of the memoryless property.

1. To emulate a Paradigm model, remains to put the different procedures together. In each cycle, method Work-Time and method Completion-Time is applied first to all of the agents with new phases, thus computing $\texttt{tableAdjW}^c$ and $\texttt{tableAdj}\Sigma^c$, using the parameters in `mu`.

2. The times in $\texttt{tableAdj}\Sigma^c$ are updated by method Simulation, using the initial times in $\texttt{times}^c$.

3. Henceforth, the trap arrivals are computed with the help of method Trap-Commit, again for the agents with new phases, and stored into $\texttt{tableTraps}^c$.

4. The times in $\texttt{tableTraps}^c$ are updated by method Simulation, using the parameters in `agent2protocol`.

5. Subsequently, each consistency rule is tried and the one that works is caught, updating the next set of phases $\texttt{phases}^{c+1}$, the next initial time $\texttt{times}^{c+1}$, the next set of agents with new phases $\texttt{changes}^{c+1}$, the current rule applied $\texttt{rule}^{c}$, and finally the next agent on which the simulation is focused $\texttt{focus}^{c+1}$ if any.

6. The times in $\texttt{times}^{c+1}$ are updated by method Simulation, using the parameters in `simulation` and in `protocol2agent`.

7. Using this information, procedure Weave is applied on each agent, computing $\texttt{states}^{c+1}$. and initializing $\texttt{tableAdjW}^{c+1}$, and then the process restarts at the next cycle.

7.7.2 Pseudo-Code

Procedure. *Simulation*

Input: parameters, $\texttt{states}^1$, $\texttt{phases}^1$, $\texttt{changes}^1$, $\texttt{rules}^1$, $\texttt{focus}^1$, $\texttt{times}^1$, and M.

Output: $\texttt{states}$, $\texttt{phases}$, $\texttt{changes}$, $\texttt{rules}$, $\texttt{focus}$, $\texttt{times}$, $\texttt{tableAdjW}$, $\texttt{tableAdj}\Sigma$, and $\texttt{tableTrap}$.

Create empty $\texttt{tableAdjW}$, and $\texttt{tableAdj}\Sigma$ and initialize parameters.
for each agent A_i from A_1 to A_n **do**
    $\texttt{tableAdjW}_i^1 \leftarrow \texttt{phases}_i^1$.
end for
for each cycle c from 1 to M **do**
    for each agent A_i from A_1 to A_n **do**
        if $\texttt{changes}_i^c = 1$ **then**
            $\texttt{tableAdjW}_i^c \leftarrow$ Work-Time($\texttt{mu}$, $\texttt{states}_i^c$, $\texttt{tableAdjW}_i^c$).
            $\texttt{tableAdj}\Sigma_i^c \leftarrow$ Completion-Time($\texttt{states}_i^c$, $\texttt{times}^c$, $\texttt{tableAdjW}_i^c$, $\texttt{tableAdj}\Sigma_i^c$).
            $\texttt{tableAdj}\Sigma_i^c \leftarrow$ adjust with initial times.
            $\texttt{tableTrap}_i^c \leftarrow$ Trap-Commit($\texttt{phases}_i^c$, $\texttt{states}_i^c$, $\texttt{tableAdj}\Sigma_i^c$)
            $\texttt{tableTrap}_i^c \leftarrow$ adjust with traveling times.
        else
            $\texttt{tableAdjW}_i^c \leftarrow \texttt{tableAdjW}_i^{c-1}$.
            $\texttt{tableAdj}\Sigma_i^c \leftarrow \texttt{tableAdj}\Sigma_i^{c-1}$.
            $\texttt{tableTrap}_i^c \leftarrow \texttt{tableTrap}_i^{c-1}$.
        end if
    end for
    for each rule R **do**
        $\texttt{phases}^{c+1}$, $\texttt{times}^{c+1}$, $\texttt{changes}^{c+1}$, $\texttt{focus}^{c+1}$, $\texttt{rule}^c \leftarrow$ Try-Rule($Rule$, $\texttt{phases}^c$, $\texttt{tableTrap}^c$, $\texttt{changes}^c$, $\texttt{focus}^c$).
    end for
    $\texttt{times}^{c+1} \leftarrow$ adjust with protocol processing time. ◇ using $\texttt{protocol}$.
    $\texttt{times}^{c+1} \leftarrow$ adjust with traveling times. ◇ using $\texttt{protocol2agent}$.
    for each agent A_i from A_1 to A_n **do**
        $\texttt{states}_i^{c+1} \leftarrow$ Weave($\texttt{states}_i^c$,$\texttt{times}_i^c$,$\texttt{tableAdj}\Sigma_i^c$).
    end for
end for

From lines 1 to 4, the simulation starts by initializing the table of holding time adjacency matrices $\texttt{tableAdjW}^1$, where each agent has to have the right phase structure, given by $\texttt{phases}^1$. From line 5 to 27, the simulation executes a complete cycle, where one cycle corresponds to the application of only one consistency rule. From lines 19 to 21, every consistency rule is tried, but by design only one is applicable at a time. Consistency rules feed on trap commits, and so from lines 6 to 18, the holding times $\texttt{tableAdjW}^c$, arrival times $\texttt{tableAdj}\Sigma^c$ and corresponding trap commits $\texttt{tableTrap}^c$ are computed for each agent. Finally, once a consistency rule is chosen, the phase arrival time is set, and the cycle initial states of the new cycle are computed. This happens from lines 22 to 26. The traveling times for the traps commits and for the phase impositions, plus the processing times for the protocol, are incorporated on lines 12, 22, and 23 respectively.

8 Numerical Exploration

In section 7 (p. 56), was shown one way of simulating the Paradigm models of section 5 (p. 39). In this section, one important statistic of concurrency models is investigated, namely the *feed* of an agent with respect to a state z.

For a system with agents $A_1, \ldots, A_n$, feed can be defined for a subset of the agents, for instance it can be defined for either only one agent A_i or for all the agents at once.

Definition 8.1 (Feed)**.** For an agent A with detailed STD Z, the long run expected feed $\bar{\mathbb{F}}(A, z)$ with respect to one state $z \in Z$ is

$$\bar{\mathbb{F}}(A, z) = \mathbb{E}\left(\lim_{T \to \infty} \frac{\int_0^T \mathbb{1}_{A(t)=z} dt}{T} \right). \tag{8.1}$$

Given that the aim of an agent is to be in a certain state, the long run expected feed of an agent is useful to determine how well the concurrency model lets the agent achieve that aim. In particular, in The Super-Robot Model 5.1 (p. 39), in the Round-Robin Solution 5.2 (p. 40) and in the Split and Non-Deterministic Solution 5.3 (p. 43), the aim of all the agents is to be in state Moving.

Therefore, for a system with n arms $\mathsf{Arm}_1, \ldots, \mathsf{Arm}_n$, the long run expected feed for the whole system is

$$\bar{\mathbb{F}}(\mathsf{Arm}_1, \ldots, \mathsf{Arm}_n, \mathsf{Moving}) = \sum_{i=1}^{n} \mathbb{E}\left(\lim_{T \to \infty} \frac{\int_0^T \mathbb{1}_{A_i(t)=z} dt}{T} \right). \tag{8.2}$$

The feed is a relevant statistic for the performance of a coordination scheme. In our robotic context, it is better to have the highest possible feed for state Moving. A high feed corresponds to simultaneous motion for the different arms of the robot. A high feed corresponds to a fluid coordination scheme. A low feed would mean that some agents are denied access of a certain state, meaning that those agents have more difficulty in making progress. Therefore, a high feed is preferable.

As discussed in the section 3 about path-planning (p 14), there are many different path-planning methods, with many different running times, and many different implementations. Moreover, some of the path-planning methods are trivially parallelizable [2, 41]. Therefore it is difficult to say how to cut up, or what execution time to expect from the state Planning in Figure 11 (p. 12) and in Figure 23 (p. 39).

We attempt a small initial exploration of state space to compare different models, checking whether they behave differently to solve the task at hand. The state space is the set of

parameters of the exponential holding times of the planning state(s) and of the moving state. For each point, in the state space, the simulation of section 7 (p. 56) is run 10 times, with 1000 cycles.

8.1 Exploration of the Super-Robot Model

For the Super-Robot model, the holding times of every state, except for states Planning and Moving, is kept at $\mu = 1$. This includes the traveling times from the protocol to the agents, from the agents to the protocol, and the working time of the protocol itself, as discussed in section 6.1.3 (p. 54).

For states Moving and Planning, the holding times range take the value $\mu_{\text{Moving}} \in \{1,2,\ldots,20\}$ and $\mu_{\text{Planning}} \in \{1,2,\ldots,20\} * 3$, to account for an increase in planning time. Therefore, 400 points are explored. Finally, the feed value is multiplied by 3, since there are 3 agents, to account for each arm moving simultaneously.

Recall that in this model there is only one robot, the super-model. The coordination is the trivial no coordination. In the subsequent models, the coordination is explored for 3 agents, and there the state Planning will take holding times in $\{1,2,\ldots,20\}$.

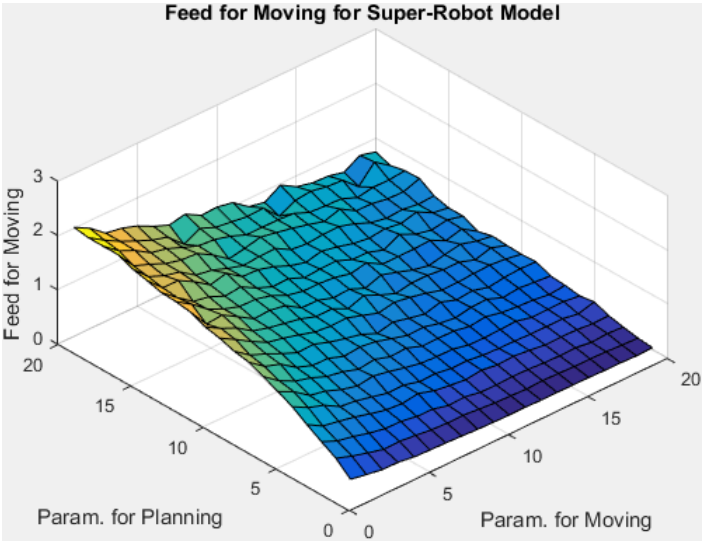

Figure 32: Super-Robot Model

8.2 Exploration of the Round-Robin Solution

For the Round-Robin solution, the holding times of every state, except for states Planning and Moving, is kept at $\mu = 1$. This includes the traveling times from the protocol to the agents,

from the agents to the protocol, and the working time of the protocol itself.

For states Moving and Planning, the holding times range take the value $\mu_{\text{Moving}} \in \{1,2,\ldots,20\}$ and $\mu_{\text{Planning}} \in \{1,2,\ldots,20\}$. Therefore, 400 points are explored.

The coordination is explored for 3 agents.

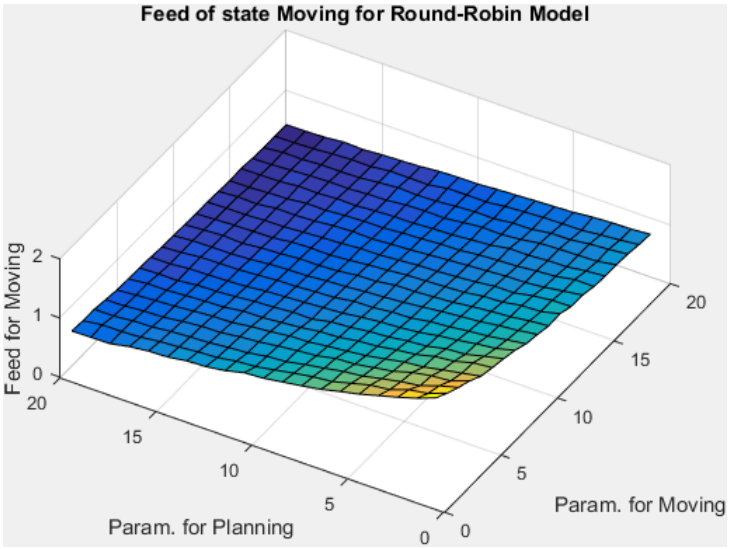

Figure 33: Round-Robin

8.3 Exploration of the Split and Non-Deterministic Solution

For the Split and Non-Deterministic solution, the holding times of every state, except for states Plan1, Plan2, Plan3, and Moving, is kept at $\mu = 1$. The path-planning is also split into 3. This is arbitrary. This includes the traveling times from the protocol to the agents, from the agents to the protocol, and the working time of the protocol itself.

For state Moving the holding times take the values $\mu_{\text{Moving}} \in \{1,2,\ldots,20\}$ and for states Plan1, Plan2, and Plan3, the holding times take the value $\mu_{\text{Plan}} \in \{1,2,\ldots,20\}/3$ (so that the first parameter is 1/3 instead of 1), such that Plan1, Plan2, and Plan3 always take the same parameters. Therefore, 400 points are explored.

Thus coordination is explored for 3 agents and 3 splits for the planning state.

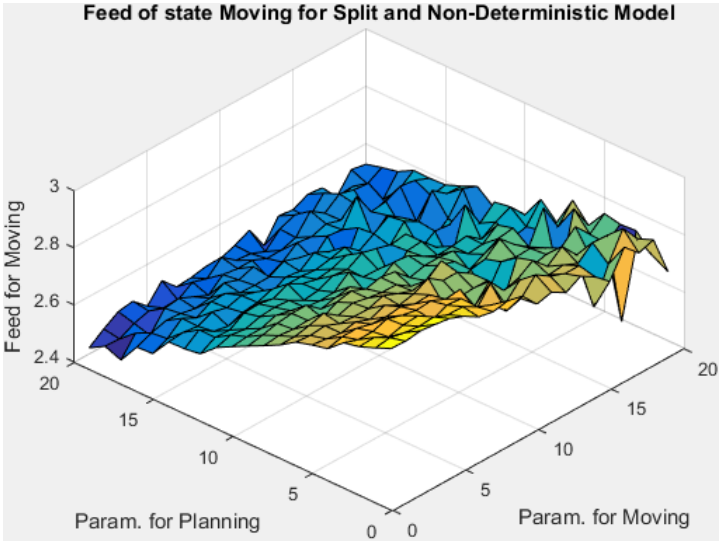

Figure 34: Split Model

8.4 Comparison of the Models

In order to draw some qualitative conclusions about the different coordination models, the different models are compared.

First, here is the difference between $\bar{\mathbb{F}}(\text{Moving})_{Split}$ and $\bar{\mathbb{F}}(\text{Moving})_{Robin}$.

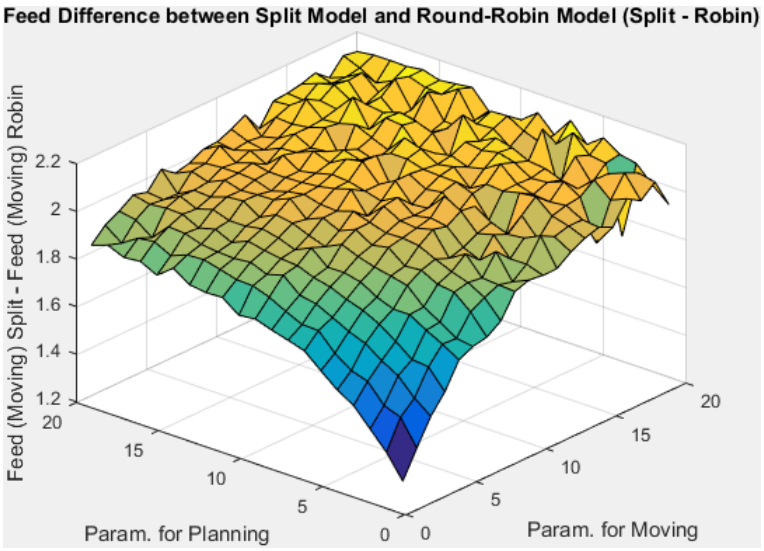

Figure 35: Graph of $\bar{\mathbb{F}}(\text{Moving})_{Split} - \bar{\mathbb{F}}(\text{Moving})_{Robin}$.

Note that in $\bar{\mathbb{F}}(\text{Moving})_{Split}$ the agents can move concurrently, but that in $\bar{\mathbb{F}}(\text{Moving})_{Robin}$ the agents do not move concurrently, but rather sequentially.

Second, here is the difference between $\bar{\mathbb{F}}(\text{Moving})_{Split}$ and $\bar{\mathbb{F}}(\text{Moving})_{Super}$.

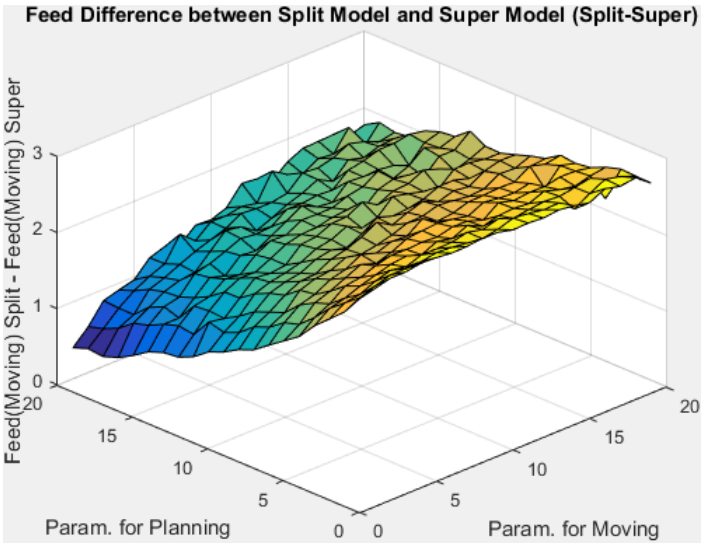

Figure 36: Graph of $\bar{\mathbb{F}}(\text{Moving})_{Split} - \bar{\mathbb{F}}(\text{Moving})_{Super}$.

Note that in $\bar{\mathbb{F}}(\text{Moving})_{Split}$ as well as in $\bar{\mathbb{F}}(\text{Moving})_{Super}$ the agents can move concurrently.

9 Conclusion

9.1 Analysis of Results

The results of the simulation seem counter-intuitive. In Figure 32 (p. 71), representing the Super-Robot model, it seems that the more the super-robot spends time in state Planning, the more the super-robot is fed with state Moving, which seems incorrect. However, after careful examination of the code, we could not find a mistake.

The simulations for the Round-Robin solution, in Figure 33 (p. 33), and for the Split and Non-Deterministic solution, in Figure 34 (p. 73), seem coherent since they have roughly the same behavior. For the Round-Robin solution, the feed seems to go up the less the robot spends time in state Moving. For the Split and Non-Deterministic solution, the feed seems to go up the more the robot spends time in state Moving, as long as the time spent for planning is big enough.

When comparing the Round-Robin solution and the Split and Non-Deterministic solution, in Figure 35 (p. 73), the Split and Non-Deterministic solution outperforms the Round-Robin solution as expected.

Moreover, comparing the Super-Robot solution and the Split and Non-Deterministic solution, in Figure 36 (p. 74), the Split and Non-Deterministic soltion also outperforms the Super-Robot solution.

These results indicate that the different models indeed behave very differently, however we must acknowledge the fact that some of these results seem odd. Moreover, note that only the case for 3 agents was tested.

9.2 Future Work

Each simulation takes under this implementation too long to execute. It is necessary to remedy this situation by finding a way to parallelize Paradigm simulations, in a safe and scallable way, both in the number of agents and in the size of the detailed STDs.

The models developed in this work should be rechecked using another programming approach, to see whether a mistake was made using this approach. Also it is necessary to see how the models respond to the number of arms, i.e. agents, present in the coordination solutions.

Moreover, it is necessary to find a canonical way of converting Paradigm models into

actual concurrent code, such as in Python, Java or C++.

On another hand, this research starts what we believe is a new kind of research in path-planning, where coordination models and path-planning methods are united in one path-planning coordination scheme. An interesting avenue of research is to see how different threads, responsible for arm path-planning and motion, can be coordinated with the help of Paradigm concurrency models for the best possible outcome.

10 Appendix

10.1 Code for Simulation

```
%Paradigm simulation for the Critical-Section Split Critical Section Solution, with 3 splits in state Planning

%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%THE INITIALIZATION%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

clear all

%Paradigm simulation for the Critical-Section Round-Robin Solution
%Simulation Parameters
nbCycles = 100; nbAgents = 3;  nbStates = 14;  nbPhases = 4;

%Indexing of the states
WaitGoal=1;     WaitRead=2;     Reading=3;      Plan1=4;
Check1=5;       Plan2=6;        Check2=7;       Plan3=8;
Check3=9;       WaitWrite=10;   Writing=11;     Moving=12;
WaitDelete=13;  Deleting=14;

%Associated exponential distribution parameters
mu = [1 1 0.001 0.001 0.001 0.001 0.001 0.001 0.001 0.001 0.001 0.001 0.001 0.001];
agent2protocol = [0.001 0.001 0.001];
protocol2agent = [0.001 0.001 0.001];
protocol = 0.001;

%Indexing of the phases
Without = 1;    Interrupt = 2;  With = 3; Update = 4;

%Indexing of the traps
triv = 1;       nY = 2;     req = 3;    dn = 4;    wrn = 5;

%Phase / Trap dependenciesc
phase_state_to_trap = [
    0,0,0,0,0,0,0,0,0,0,0,0,0,0;
    nY,nY,nY,nY,nY,nY,nY,nY,nY,req,0,nY,req,0;
    dn,dn,dn,dn,dn,dn,dn,dn,dn,0,0,dn,0,0;
    wrn,wrn,0,0,0,0,0,0,0,0,0,wrn,wrn,0;
];
```

```
38
39 %phases
40 adj_With = [-1, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
41 0, -1, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
42 0, 0, -1, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
43 0, 0, 0, -1, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0;
44 0, 0, 0, 0, -1, -2, 0, 0, 0, 0, 0, 0, 0, 0;
45 0, 0, 0, 0, 0, -1, -2, 0, 0, 0, 0, 0, 0, 0;
46 0, 0, 0, 0, 0, 0, -1, -2, 0, 0, 0, 0, 0, 0;
47 0, 0, 0, 0, 0, 0, 0, -1, -2, 0, 0, 0, 0, 0;
48 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0;
49 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -2, 0, 0, 0;
50 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -2, 0, 0;
51 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0;
52 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -2;
53 -2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1];
54
55 adj_Without = [-1, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
56 0, -1, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
57 0, 0, -1, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
58 0, 0, 0, -1, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0;
59 0, 0, 0, 0, -1, -2, 0, 0, 0, 0, 0, 0, 0, 0;
60 0, 0, 0, 0, 0, -1, -2, 0, 0, 0, 0, 0, 0, 0;
61 0, 0, 0, 0, 0, 0, -1, -2, 0, 0, 0, 0, 0, 0;
62 0, 0, 0, 0, 0, 0, 0, -1, -2, 0, 0, 0, 0, 0;
63 0, 0, 0, 0, 0, 0, 0, 0, -1, -2, 0, 0, 0, 0;
64 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0;
65 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
66 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -2, 0;
67 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0;
68 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
69
70 adj_Interrupt = [-1, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
71 0, -1, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
72 0, 0, -1, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
73 0, 0, 0, -1, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0;
74 0, 0, 0, 0, -1, -2, 0, 0, 0, 0, 0, 0, 0, 0;
75 0, 0, 0, 0, 0, -1, -2, 0, 0, 0, 0, 0, 0, 0;
76 0, 0, 0, 0, 0, 0, -1, -2, 0, 0, 0, 0, 0, 0;
77 0, 0, 0, 0, 0, 0, 0, -1, -2, 0, 0, 0, 0, 0;
78 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0;
79 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0;
80 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
81 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0;
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0;
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];

adj_Update = [-1, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
0, -2, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
0, 0, 0, -1, -2, 0, 0, 0, 0, 0, 0, 0, 0, 0;
0, -2, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0;
0, 0, 0, 0, 0, -1, -2, 0, 0, 0, 0, 0, 0, 0;
0, -2, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0;
0, 0, 0, 0, 0, 0, 0, -1, -2, 0, 0, 0, 0, 0;
0, -2, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0, 0;
0, -2, 0, 0, 0, 0, 0, 0, 0, -1, 0, 0, 0, 0;
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, -2, 0;
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0;
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];

%Creating the data-structures
%Tables
table_AdjW = cell(nbAgents, nbCycles);
table_AdjS = cell(nbAgents, nbCycles);
table_Traps = cell(nbAgents, nbCycles);
%Matrices
states = zeros(nbAgents, nbCycles);
phases = zeros(nbAgents, nbCycles);
times = zeros(nbAgents, nbCycles);
%encoding what has changed, and not what is going to change
changes = ones(nbAgents, nbCycles);
focus = zeros(1, nbCycles);
focus(1) = 1;
rules = zeros(1, nbCycles);

%Initialization of data-structures
%The initial state labels in the first cycle
states(1,1) = WaitRead; states(2,1) = WaitGoal; states(3,1) =
    WaitGoal;
%The initial phase labels in the first cycle
phases(1,1) = With; phases(2,1) = Without; phases(3,1) =
   Without;
%The initial phase adjacency matrices for holding times
table_AdjW{1,1} = adj_With; table_AdjW{2,1} = adj_Without;
   table_AdjW{3,1} = adj_Without;
table_AdjS{1,1} = adj_With; table_AdjS{2,1} = adj_Without;
```

```
    table_AdjS{3,1} = adj_Without;
123
124 %Start at agent 1
125 current_agent = 1;

1 %%%%%%%%%%%%%%%%%%%%%%%%
2 %%%%%THE SIMULATION%%%%%%
3 %%%%%%%%%%%%%%%%%%%%%%%%
4 for c =1:(nbCycles-1)
5     for Ai = 1:nbAgents;
6         %we do not want to recompute everything
7         if changes(Ai,c) == 1
8             table_AdjW{Ai,c} = WorkTime(mu, states(Ai, c),
                  table_AdjW{Ai, c});
9             table_AdjS{Ai,c} = CompletionTime(states(Ai,c),
                  table_AdjW{Ai, c}, table_AdjS{Ai, c});
10            ind1 = (table_AdjS{Ai, c} > 0);
11            table_AdjS{Ai,c} = table_AdjS{Ai, c} + times(Ai,c
                  )*ind1;
12            %Compute the arrivals into the traps
13            table_Traps{Ai,c} = TrapCommitSplit(phases(Ai, c)
                  , states(Ai, c), table_AdjS{Ai, c}, table_AdjW
                  {Ai,c});
14            %Add the sending times for the trap commits
15            ind2 = (table_Traps{Ai,c} >= 0); %indices of
                  positive elements
16            travel = exprnd(agent2protocol(Ai), size(ind2));
                  %different sending time for different traps
17            table_Traps{Ai, c} = table_Traps{Ai, c} + travel
                  .*ind2; %only the positive elements are
                  affected
18        else
19            table_AdjW{Ai,c} = table_AdjW{Ai,c-1};
20            table_AdjS{Ai,c} = table_AdjS{Ai,c-1};
21            table_Traps{Ai,c} = table_Traps{Ai, c-1};
22        end
23    end
24
25    %METHOD TRY-RULE
26    trap_Slice = cell(nbAgents,1);
27    for Ai=1:nbAgents
28        trap_Slice{Ai}=table_Traps{Ai,c};
29    end
30    [phases(:,c+1), times(:,c+1), changes(:,c+1), focus(c+1),
          rules(c)] =  TryRule(phases(:,c), trap_Slice,
```

```
        changes(:,c), focus(c));
31
32      %times(:,c+1), the whole column set to one value
33      %adjust the processing time
34      times(:,c+1) = times(:,c+1) + exprnd(protocol);
35
36      %adjust the sending times of each phase imposition
37      for Ai = 1:nbAgents
38          times(Ai,c+1)= times(Ai,c+1) + exprnd(protocol2agent(
                Ai));
39      end
40
41      %Fill table_AdjW and table_AdjS with the newly obtained
            phases.
42      for Ai=1:nbAgents
43          if phases(Ai,c+1) == Without
44              table_AdjW{Ai,c+1} = adj_Without;
45              table_AdjS{Ai,c+1} = adj_Without;
46          end
47          if phases(Ai,c+1) == Interrupt
48              table_AdjW{Ai,c+1} = adj_Interrupt;
49              table_AdjS{Ai,c+1} = adj_Interrupt;
50          end
51          if phases(Ai,c+1) == With
52              table_AdjW{Ai,c+1} = adj_With;
53              table_AdjS{Ai,c+1} = adj_With;
54          end
55          if phases(Ai,c+1) == Update
56              table_AdjW{Ai,c+1} = adj_Update;
57              table_AdjS{Ai,c+1} = adj_Update;
58          end
59      end
60
61      %METHOD WEAVE
62      for Ai = 1:nbAgents
63          this_adjW = table_AdjW{Ai, c};
64          next_adjW = table_AdjW{Ai, c+1};
65          this_adjS = table_AdjS{Ai, c};
66          states(Ai, c+1) = Weave3(states(Ai,c),times(Ai, c+1),
                this_adjS);
67      end
68  end
```

10.2 Code for Work-Time

```
function [ adjW ] = WorkTime(mu, state, adjW)
    next_state = state;
    if(adjW(next_state,next_state) < 0)
        %add  mu(next_state)/1000000000 to avoid zeros
        adjW(next_state,next_state) = exprnd(mu(next_state))
           + mu(next_state)/1000000000;
    end
    next_state = next(adjW, next_state);
    while (isempty(next_state) == false)
        if(adjW(next_state,next_state) < 0)
            %add  mu(next_state)/1000000000 to avoid zeros
            adjW(next_state,next_state) = exprnd(mu(
               next_state)) + mu(next_state)/1000000000;
        end
        next_state = next(adjW, next_state);
    end
end
```

10.3 Code for Completion-Time

```
function [ adjS ] = CompletionTime(state,adjW,adjS)
    %Function definition is different from the pseudo-code,
       it is easier to pass adjS for modification, than to
       declare it inside this function.
    next_state = state;
    sum = adjW(next_state, next_state);
    adjS(next_state, next_state) = sum;
    next_state = next(adjW, next_state);
    while (isempty(next_state)==false)
       sum2 = adjW(next_state, next_state);
       sum = sum + sum2;
       adjS(next_state, next_state) = sum;
       next_state = next(adjW, next_state);
    end
end
```

10.4 Code for Trap-Commit

```
function [ phaseNtrap ] = TrapCommit(phase, state, adjS)
    %Indexing the traps.
    triv = 1; notYet = 2; request = 3; done = 4;
    nbStates = 9;

    phaseNstate = [
        0,0,0,0,0,0,0,0,0; %phase without has trap triv
        notYet,request,0,0,0,0,0,0,0; %phase interrupt has
           traps notYet and request
        done,0,0,0,0,0,0,0,0; %phase done has trap done
    ];

    %col1: triv, col2: notYet, col3: request, col4: done
    phaseNtrap = [
        -1,-1,-1,-1; %phase 1
        -1,-1,-1,-1; %phase 2
        -1,-1,-1,-1; %phase 3
    ];

    phaseNtrap(phase, triv) = adjS(state, state);

    for(node = 1:nbStates)
        trap = phaseNstate(phase, node);
        if(trap > 0)
            phaseNtrap(phase, trap) = adjS(node, node);
        end
    end
end
```

10.5 Code for Try-Rule

```
function [ next_phases, next_time, next_change, focus, rule ]
    = TryRule(sim_phase, trap_slice, next_change, focus)
    nbAgents = 3;

    next_phases = sim_phase;
    next_time = -1;
    next_change = zeros(size(next_change)); %put a one if
       there has been a change
    rule= -1;

    Without = 1;
    Interrupt = 2;
    With = 3;
    Update = 4;

    triv = 1;
    notYet = 2;
    request = 3;
    done = 4;
    warned = 5;

    %Rule 1
    if sim_phase(focus) == Interrupt && trap_slice{focus}(
       Interrupt, notYet) >= 0
       next_time = trap_slice{focus}(Interrupt, notYet);

       next_phases(focus) = Without;
       next_change(focus) = 1;
       if focus == 3
           focus =1;
       else
           focus = focus+1;
       end
       rule = 1;
       return
    end

    %Rule 2
    if sim_phase(focus) == Interrupt && trap_slice{focus}(
       Interrupt, request) >= 0
       next_time = trap_slice{focus}(Interrupt, request);

```

```
39          next_phases(focus) = With;
40          next_change(focus) = 1;
41          rule = 2;
42          return
43      end
44
45      %Rule 3
46      cond3 = sim_phase(focus) == Without;
47      time3 = trap_slice{focus}(Without,triv);
48      for Ai = 1:nbAgents
49              cond3 = cond3 && (sim_phase(Ai) == Without);
50              time3 = max(time3, trap_slice{Ai}(Without, triv));
51      end
52
53      if cond3 == true
54          next_time = time3;
55
56          next_phases(focus) = Interrupt;
57          next_change(focus) = 1;
58          rule = 3;
59          return
60      end
61
62      cond4 = sim_phase(focus) == With;
63      time4 = trap_slice{focus}(With,done);
64      for Ai = 1:nbAgents
65          if(Ai ~= focus)
66              cond4 = cond4 && (sim_phase(Ai) == Without);
67              time4 = max(time4, trap_slice{Ai}(Without, triv));
68          end
69      end
70
71      if cond4 == true
72          next_time = time4;
73
74          next_phases(focus) = Without;
75          next_change(focus) = 1;
76          for Ai = 1:nbAgents
77              if(Ai ~= focus)
78                  next_phases(Ai) = Update;
79                  next_change(Ai) = 1;
80              end
81          end
82          rule = 4;
```

```
83          return
84      end
85
86
87      cond5 = true;
88      time5 = -1;
89      for Ai = 1:nbAgents
90          if(Ai ~= focus)
91              cond5 = cond5 && (sim_phase(Ai) == Update);
92              time5 = max(time5, trap_slice{Ai}(Update, warned));
93          end
94      end
95
96      if cond5 == true
97          next_time = time5;
98
99          for Ai = 1:nbAgents
100             if(Ai ~= focus)
101                 next_phases(Ai) = Without;
102                 next_change(Ai) = 1;
103             end
104         end
105         if focus == 3
106             focus = 1;
107         else
108             focus = focus+1;
109         end
110         rule = 5;
111         return
112     end
113 end
```

10.6 Code for Weave

```
function [ next_state ] = Weave(state, time, this_adjS )
    this_state = state;
    finish = this_adjS(this_state, this_state);

    if islast(this_adjS, this_state) == true
        next_state = this_state;
    end

    while islast(this_adjS, this_state) == false
        if finish >= time
            next_state = this_state;
            break; %to the next agent
        else
            this_state = next(this_adjS, this_state);
            finish = this_adjS(this_state, this_state);
            if islast(this_adjS, this_state) == true
                next_state = this_state;
            end
        end
    end
end
```

10.7 Code for Helper Functions

10.7.1 Method next

```
function [ next_state ] = next(phase, state)
    next_state = find(phase(state, :) == -2);
end
```

10.7.2 Method islast

```
function [ bool ] = islast( phase, state )
    %   given a phase matrix, this function returns whether
        the state is the
    %   a leaf or not, returns yes for leaf, no for not a
        leaf
    next_state = next(phase,state);
    if isempty(next_state)==false
        bool = false;
    else
        bool = true;
    end
end
```

References

[1] G. A. Agha. Actors: A model of concurrent computation in distributed systems. Technical report, DTIC Document, 1985.

[2] N. M. Amato and L. K. Dale. Probabilistic roadmap methods are embarrassingly parallel. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 1, pages 688–694. IEEE, 1999.

[3] S. Andova, L. Groenewegen, and E. P. de Vink. Dynamic consistency in process algebra: From paradigm to acp. *Science of Computer Programming*, 76(8):711–735, 2011.

[4] O. B. Bayazit, J.-M. Lien, and N. M. Amato. Probabilistic roadmap motion planning for deformable objects. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 2, pages 2126–2133. IEEE, 2002.

[5] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.

[6] V. Boor, M. H. Overmars, and A. F. van der Stappen. The gaussian sampling strategy for probabilistic roadmap planners. In *Robotics and automation, 1999. proceedings. 1999 ieee international conference on*, volume 2, pages 1018–1023. IEEE, 1999.

[7] P. Buchholz, J. Kriege, and I. Felko. *Input Modeling with Phase-Type Distributions and Markov Models: Theory and Applications*. Springer, 2014.

[8] K. M. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):632–646, 1984.

[9] H. Choset. Coverage for robotics–a survey of recent results. *Annals of mathematics and artificial intelligence*, 31(1-4):113–126, 2001.

[10] P.-J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, 1971.

[11] A. Elfes. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22(6):46–57, 1989.

[12] D. Ferguson and A. Stentz. Using interpolation to improve path planning: The field d* algorithm. *Journal of Field Robotics*, 23(2):79–101, 2006.

[13] D. Ferguson, M. Likhachev, and A. Stentz. A guide to heuristic-based path planning. In *Proceedings of the international workshop on planning under uncertainty for autonomous systems, international conference on automated planning and scheduling (ICAPS)*, pages 9–18, 2005.

[14] R. Geraerts and M. H. Overmars. A comparative study of probabilistic roadmap planners. In *Algorithmic Foundations of Robotics V*, pages 43–57. Springer, 2004.

[15] L. Groenewegen and E. De Vink. Operational semantics for coordination in paradigm. In *International Conference on Coordination Languages and Models*, pages 191–206. Springer, 2002.

[16] L. Groenewegen and E. De Vink. Evolution on-the-fly with paradigm. In *International Conference on Coordination Languages and Models*, pages 97–112. Springer, 2006.

[17] L. Groenewegen, N. van Kampenhout, and E. de Vink. Delegation modeling with paradigm. In *International Conference on Coordination Languages and Models*, pages 94–108. Springer, 2005.

[18] R. Hammack, W. Imrich, and S. Klavžar. *Handbook of product graphs*. CRC press, 2011.

[19] D. Hsu, L. E. Kavraki, J.-C. Latombe, R. Motwani, S. Sorkin, et al. On finding narrow passages with probabilistic roadmap planners. In *Robotics: The Algorithmic Perspective: 1998 Workshop on the Algorithmic Foundations of Robotics*, pages 141–154, 1998.

[20] D. Hsu, T. Jiang, J. Reif, and Z. Sun. The bridge test for sampling narrow passages with probabilistic roadmap planners. In *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, volume 3, pages 4420–4426. IEEE, 2003.

[21] L. E. Kavraki, P. Svestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4):566–580, 1996.

[22] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS)*, 5(1):1–11, 1987.

[23] J.-C. Latombe. *Robot motion planning*, volume 124. Springer Science & Business Media, 2012.

[24] S. M. LaValle. *Planning algorithms*. Cambridge university press, 2006.

[25] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. *The International Journal of Robotics Research*, 20(5):378–400, 2001.

[26] M. S. S. B. Luan, Silveira; Renan Q. and associates. Space d*, a path-planning algorithm for multiple robots in unknown environments. *Brazilian Computer Society*, 2012.

[27] J.-A. Meyer and D. Filliat. Map-based navigation in mobile robots:: Ii. a review of map-learning and path-planning strategies. *Cognitive Systems Research*, 4(4):283–317, 2003.

[28] P. E. Missiuro and N. Roy. Adapting probabilistic roadmaps to handle uncertain maps. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 1261–1267. IEEE, 2006.

[29] C. Moler and C. Van Loan. Nineteen dubious ways to compute the exponential of a matrix. *SIAM review*, 20(4):801–836, 1978.

[30] R. M. Murray, Z. Li, S. S. Sastry, and S. S. Sastry. *A mathematical introduction to robotic manipulation*. CRC press, 1994.

[31] M. F. Neuts. *Probability distributions of phase type*. Purdue University. Department of Statistics, 1974.

[32] M. F. Neuts. Models based on the markovian arrival process. *IEICE Transactions on Communications*, 75(12):1255–1265, 1992.

[33] P. A. O'Donnell and T. Lozano-Pérez. Deadlock-free and collision-free coordination of two robot manipulators. In *Proc. of the IEEE International Conference on Robotics & Automation (ICRA)*, 1989.

[34] L. E. Parker. Current state of the art in distributed autonomous mobile robotics. In *Distributed Autonomous Robotic Systems 4*, pages 3–12. Springer, 2000.

[35] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[36] J. Protic, M. Tomasevic, and V. Milutinović. *Distributed shared memory: Concepts and systems*, volume 21. John Wiley & Sons, 1998.

[37] M. Raynal. Algorithms for mutual exclusion. 1986.

[38] G. Sánchez and J.-C. Latombe. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In *Robotics Research*, pages 403–417. Springer, 2003.

[39] A. Stentz. Optimal and efficient path planning for partially-known environments. In *Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on*, pages 3310–3317. IEEE, 1994.

[40] G. Taubenfeld. *Synchronization algorithms and concurrent programming*. Pearson Education, 2006.

[41] C.-C. Tsai, H.-C. Huang, and C.-K. Chan. Parallel elite genetic algorithm and its application to global path planning for autonomous robot navigation. *IEEE Transactions on Industrial Electronics*, 58(10):4813–4821, 2011.

[42] J. Van Den Berg, D. Ferguson, and J. Kuffner. Anytime path planning and replanning in dynamic environments. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 2366–2371. IEEE, 2006.

[43] J. Van Den Berg, S. J. Guy, M. Lin, and D. Manocha. Reciprocal n-body collision avoidance. In *Robotics research*, pages 3–19. Springer, 2011.

[44] S. A. Wilmarth, N. M. Amato, and P. F. Stiller. Maprm: A probabilistic roadmap planner with sampling on the medial axis of the free space. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 2, pages 1024–1031. IEEE, 1999.