

Multi-Robot Systems in Cognitive Factories:
Representation, Reasoning, Execution and Monitoring

by

Kadir Haspalamutgil

Submitted to the Graduate School of Sabancı University
in partial fulfillment of the requirements for the degree of
Master of Science

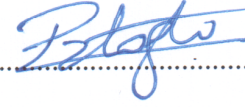
Sabancı University

August, 2011

Multi-Robot Systems in Cognitive Factories:
Representation, Reasoning, Execution And Monitoring

APPROVED BY:

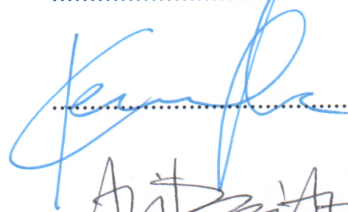
Assist. Prof. Dr. Volkan Patođlu
(Thesis Co-Advisor)



Assist. Prof. Dr. Esra Erdem
(Thesis Co-Advisor)



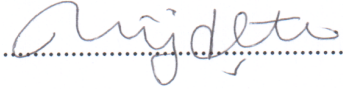
Prof. Dr. Kemal İnan



Prof. Dr. Ali Rana Atılgan



Assist. Prof. Dr. Mijdat etin



DATE OF APPROVAL:



© Kadir Haspalamutgil, 2011
All Rights Reserved

Multi-Robot Systems in Cognitive Factories: Representation, Reasoning, Execution and Monitoring

Kadir Haspalamutgil

ME, Master of Science, 2011

Thesis Co-Advisors: Assist. Prof. Dr. Volkan Patoğlu

Assist. Prof. Dr. Esra Erdem

Keywords: Manipulation Planning, Execution And Monitoring, Diagnosis, Decoupled Planning, Cognitive Factory.

Abstract

We propose the use of causality-based formal representation and automated reasoning methods from artificial intelligence to endow multiple teams of robots in a factory, with high-level cognitive capabilities, such as optimal planning and diagnostic reasoning. We present a framework that features bilateral interaction between task and motion planning, and embeds geometric reasoning in causal reasoning. We embed this planning framework inside an execution and monitoring framework and show its applicability on multi-robot systems. In particular, we focus on two domains that are relevant to cognitive factories: i) a manipulation domain with multiple robots working concurrently / co-operatively to achieve a common goal and ii) a factory domain with multiple teams of robots utilizing shared resources.

In the manipulation domain two pantograph robots perform a complex task that requires true concurrency. The monitoring framework checks plan execution for two sorts of failures: collisions with unknown obstacles and change of the world due to human interventions. Depending on the cause of the failures, recovery is done by calling the motion planner (to find a different trajectory) or the causal reasoner (to find a new task plan). Therefore, recovery relies on not only motion planning but also causal reasoning.

We extend our planning and monitoring framework for the factory domain with multiple teams of robots by introducing algorithms for finding optimal decoupled plans and diagnosing the cause of a failure/discrepancy (e.g., robots may get broken or tasks may get reassigned to teams). We show the applicability of these algorithms on an intelligent factory scenario through dynamic simulations and physical experiments.

Bilişsel Fabrikalarda Çoklu-Robot Sistemleri: Gösterim, Akıl Yürütme, İcra ve Takibi

Kadir Haspalamutgil

ME, Yüksek Lisans Tezi, 2011

Tez Eş Danışmanları: Yrd. Doç. Dr. Volkan Patoğlu

Yrd. Doç. Dr. Esra Erdem

Anahtar kelimeler: Manipülasyon Planlama, İcra ve Takibi, Teşhis, Ayrışmış Planlama, Bilişsel Fabrika.

Abstract

Bu tezde, bir fabrikada bulunan birden fazla robot takımlarının yapay zekası, nedensellik tabanlı şekillendirme gösteriminin kullanımı ve otomatik akıl yürütme yöntemleri ile, eniyilenmiş planlama ve teşhissel akıl yürütme gibi yüksek seviye bilişsel becerileri kazandırmak üzere ele alınmıştır. Sunduğumuz mimari, görev ve hareket planlama arasında iki-yönlü etkileşim sağlamak ve geometrik akıl yürütmeyi mantıksal akıl yürütme ile birleştirmektedir. Bu planlama mimarisini, icra ve takip mimarisinin içine yerleştirip, uygulanabilirliğini çoklu robot sistemleri üzerinde gösteriyoruz. Özellikle, bilişsel fabrikalarla ilgili iki problem üzerinde odaklanıyoruz: i) çoklu robotların eş zamanlı olarak / işbirliği yaparak ortak görev için çalıştığı bir manipülasyon problemi ve ii) birden fazla robot takımının ortak kaynak kullanımını değerlendirdiği bir fabrika problemi.

Manipülasyon probleminde, iki pantograf robot, gerçek eşzamanlılık isteyen karışık bir görevi gerçekleştirmektedirler. Takip mimarisi planın icrasını iki tip hata için kontrol etmektedir: bilinmeyen engellerle çarpışma ve dünyanın harici bir müdahale (örneğin insan müdahalesi) sonucu değişmesi. Hatanın sebebine dayanarak, hareket planlama (farklı bir yörünge bulmak için) veya mantıksal akıl yürütücü (yeni bir görev planı bulmak için) çağrılarak kurtarma yapılmaktadır. Bu sebeple, kurtarma hem hareket planlamaya hem de mantıksal akıl yürütmeye dayanmaktadır.

Önerdiğimiz planlama ve takip etme mimarisini, eniyilenmiş ayrışmış planlar ve hatanın/çelişkinin (örneğin, robotların bozulması veya takımlar için yeni görevler belirlenmesi) sebebini bulmak adına; birden fazla robot takımlarına sahip

fabrika problemleri için genelleştiriyoruz. Bu algoritmaların uygulanabilirliğini, zeki bir fabrika senaryosu üzerinde dinamik benzetimler ve fiziksel deneyler üzerinde gösteriyoruz.

Acknowledgements

It is a great pleasure to extend my gratitude to my thesis advisor Assist. Prof. Dr. Volkan Patođlu and Assist. Prof. Dr. Esra Erdem for his precious guidance and support. I would gratefully thank Prof. Dr. Kemal İnan, Prof. Dr. Ali Rana Atılgan and Assist. Prof. Dr. Mijdat etin for their feedbacks and spending their valuable time to serve as my jurors.

I would like to acknowledge the financial support provided by Sabancı University through my Master education under Internally-funded Research Projects scholarship.

Many thanks to my friends, Can Palaz, Tansel Uras, Ozan Tokatlı, Aykut Cihan Satıcı, Ahmetcan Erdođan, Melda Ulusoy, Alper Ergin, Elif Hocaođlu and many other friends for their help and friendship during my study.

Finally, I would like to thank my family for all their love and support throughout my life.

Contents

1	Introduction	1
1.1	Contributions	8
1.2	Outline	9
2	Representation and Reasoning for a Dynamic Domain	10
2.1	Action Description Language $\mathcal{C}+$	10
2.2	CCALC	14
2.3	Why $\mathcal{C}+$ and CCALC?	19
3	Assembly Planning With Multiple Robots	22
3.1	A Planning and Execution Monitoring Framework for Multiple Robots	23
3.1.1	Execution Monitoring	27
3.2	Example: Two Robots and Multiple Payloads	28
3.3	Embedding Geometric Reasoning in Action Domain Descriptions	29
3.3.1	Representing Actions and Change	30
3.3.2	Embedding Geometric Reasoning in Causal Laws	31

3.4	Bilateral Interaction between Causal Reasoning and Motion Planning	33
3.5	Case Studies	38
3.6	Discussion	42
4	Cognitive Factories With Multiple Teams Of Robots	52
4.1	A Planning and Execution Monitoring Framework For Multiple Teams of Robots	53
4.2	A Cognitive Painting Factory Scenario	54
4.3	High Level Representation of the Factory	56
4.4	Causality-Based Reasoning for Finding Optimal Decoupled Plans for Multiple Teams of Robots	60
4.5	Diagnostic Reasoning in a Cognitive Factory	61
4.5.1	Modifying the Domain Description: Exceptions	62
4.5.2	Finding Minimal Diagnosis	62
4.5.3	Repairing as part of Replanning	65
4.6	Embedding Diagnostic Reasoning in an Execution Monitoring Framework	66
4.6.1	Dynamic Simulation of the Cognitive Painting Factory	71
5	Conclusions	74
A	The Task Plan for Query 2 of the Manipulation the Planning Problem	76

List of Figures

2.1.1 The suitcase domain described in C+	13
2.2.1 The suitcase domain described in the language of CCALC.	16
3.1.1 Classical 3-layer robot control architecture for planning, execution and monitoring.	23
3.1.2 Proposed framework for planning, execution and monitoring.	25
3.4.1 (a) presents the initial state, while (b)–(f) illustrate the execution of Plan 2. Colors red and blue are associated with Robots 1 and 2. Circles indicate the positions of the robot end-effectors and circles' labels denote the time steps. Solid red and blue lines denote the trajectories of robot end-effectors. Green, red and magenta lines denote Payloads 1–3. Black disks represent the obstacles. For instance, at Step 3, end-effectors of Robots 1 and 2 are located at $(3, 5)$ and $(8, 5)$ respectively, and robots hold Payload 2. At Step 4, end-effectors of Robots 1 and 2 are located at $(6, 4)$ and $(10, 7)$, still holding Payload 2. Trajectory of Payload 2 moving from Step 3 to 4 is depicted in brown.	38

3.5.1	Snapshots of execution of Plan 2. Note that the camera is rotated 90° counterclockwise with respect to the trajectory plots in Fig. 3.4.1 for a less occluded view.	47
3.5.2	(a)–(e) present execution of Plan 2 from step 11, where there is an unknown obstacle (green disk) just outside the workspace, next to the points $(0, 6)$ and $(0, 7)$, interfering with the motion of the robots. Colors red and blue are associated with Robots 1 and 2. Circles indicate the positions of the robot end-effectors and circles' labels denote the time steps. Solid red and blue lines denote the trajectories of robot end-effectors. Green, red and magenta lines denote Payloads 1–3. Black disks are the known obstacles, while the green disk represents the unknown obstacle. At Step 14, there is an action failure because the payload collides with the unknown obstacle (Fig. 3.5.2 (a)). In (b) the robots identify the cause of failure and backtrack to a safe state. A new motion plan is found and executed in (c). The execution of rest of the plan is presented in (d)–(e).	48
3.5.3	Snapshots of execution of Plan 2, starting from time step 14. At time step 14, the payload collides with the obstacle. Note that the camera is rotated 90° counterclockwise with respect to the trajectory plots in Fig. 3.5.2 for a less occluded view.	49

3.5.4 (a)–(d) present execution of Plan 2 from step 22. Colors red and blue are associated with Robots 1 and 2. Circles indicate the positions of the robot end-effectors and circles' labels denote the time steps. Solid red and blue lines denote the trajectories of robot end-effectors. Green, red and magenta denote Payloads 1–3. Black disks represent known obstacles. At Step 24, there is a human intervention and the position of Payload 3 is changed (Fig. 3.5.4 (a)). In (b) the robots identify the cause of failure and replan accordingly. A new task plan and corresponding trajectories are found and executed in (c)–(d).	50
3.5.5 Snapshots of execution of Plan 2, starting from time step 23. At time step 24, there is a human intervention: someone changes the position of Payload 3. After putting Payload 2 to a safe position, the robots continue with the execution of the new task plan, Plan 3. Note that the camera is rotated 90° counterclockwise with respect to the trajectory plots in Fig. 3.5.4 for a less occluded view.	51
4.1.1 Decoupled planning, execution and monitoring framework	53
4.6.1 Snapshots for three consecutive states (Steps 16–18) of plan execution for workspaces. The pink assembly line is along the north wall; the pit stop area is marked by P; the squares are the boxes (1: unprocessed, 2: painted and wet, 3: painted and dry, 4: waxed, 5: stamped); the large robot is the carrier, it can be attached (a) or unattached (u); the triangles are the workers, which can have different end-effectors (2: painter, 4: waxer, 5: stamper); the benched robots are either white (ready to be given to another team) or black (still in the transportation delay).	68

4.6.2 Cognitive painting factory with multiple teams of KUKA youBots	71
4.6.3 KUKA youBot holonomic mobile manipulator	72
4.6.4 Execution of an example plan for one team in dynamic simulation	
Gazebo	73

List of Algorithms

1	TASK&MOTION_PLAN	44
2	MOTION_PLAN	45
3	EXECUTE&MONITOR	46
4	DIAGNOSE	63
5	EXECUTE&MONITOR WITH DIAGNOSIS	67

List of Tables

4.6.1 List of actions for all teams	69
---	----

Chapter 1

Introduction

New approaches for automated fabrication of customized products become highly demanded since conventional manufacturing and assembly systems fall short of responding to ever increasing market demands for customized and variant-rich products in a cost effective manner. The cognitive factory concept [1,2] is such a paradigm shift that promises significant advantages over conventional manufacturing by balancing the efficiency and flexibility demands in automation, while simultaneously achieving a high-degree of reliability. In particular, cognitive factories aim to endow manufacturing system with high-level reasoning capabilities in the style of cognitive robotics, such that these systems become capable of planning their own actions, reconfiguring themselves to allow fabrication of a wide range of parts and to react to change in demands, detecting failures during execution, diagnosing the cause of these failures and recovering from such failures. Since cognitive factories can plan their own actions and self-reconfigure, they can rapidly respond to changing customer needs and customization requests, demonstrating the necessary flexibility, while maintaining cost-effectiveness compared

to human workshops. Moreover, thanks to fault-awareness, diagnostic reasoning and failure recovery features of cognitive factories, these systems enable a high-degree of reliability comparable to those of mass production systems.

Manipulation planning, automatic generation of robot motion sequences for manipulation of movable objects among obstacles to achieve a desired goal configuration, is an indispensable process in realizing cognitive factories. Manipulation planning problems involve objects that can only move when picked up by robots and the order of pick-and-place operations for manipulation may matter to obtain a feasible kinematic solution [3]. Therefore, geometric reasoning and motion planning alone are not sufficient to solve these problems; planning of actions such as the pick-and-place operations need to be integrated with the motion planning problem.

On the other hand, once a collision-free plan with its trajectory is computed with such a hybrid approach, its execution in a dynamic world may lead to failures due to incomplete knowledge (e.g., unknown obstacles) or uncertainties (e.g., human interventions). For a successful execution of an hybrid plan in a dynamic world, detecting failures and deciding how to recover from such failures are necessary.

To this end, we present a generic and modular planning and execution framework for a cognitive factory that involves complex manipulation tasks performed concurrently / cooperatively by multiple robots. The monitoring framework checks plan execution and depending on the cause of the failures, recovery is done by calling the motion planner (to find a different trajectory) or the causal reasoner (to find a new task plan). Therefore, recovery relies on not only motion planning but also causal reasoning. Then, we extend this framework for a cognitive factory

that involves complex tasks performed by *multiple teams of robots* by efficiently using the shared resources. We focus on two crucial aspects of a cognitive factory setting with multiple teams of robots: planning and decision-making for reconfigurable networked robots for efficient use of shared resources (e.g., workforce, time, product components) and execution monitoring and diagnostic reasoning for fault-tolerance (e.g., preventing action failures and recovering from failures when they occur).

We propose the use of a causality-based formal representation (e.g., action language $\mathcal{C}+$ [4]) and automated reasoning methods and tools (e.g., the causal reasoner CCALC [5]) from artificial intelligence to endow multi-robot systems in a factory, with such high-level cognitive capabilities. In particular, we introduce novel algorithms for finding optimal decoupled plans and execution monitoring framework for recovering from failures due to uncertainties or incomplete knowledge, and for diagnosing the cause of a failure/discrepancy (e.g., robots may get broken or tasks may get reassigned to teams). To be able to coordinate networked robots and diagnose and handle failures in a dynamic setting, we embed these algorithms modularly in a generic execution and monitoring framework that allows reusability of computed plans in case of failures. The proposed generic framework is applied, in particular, to cognitive factory scenarios.

Our cognitive factory framework possesses core characteristics of a reconfigurable manufacturing system. In particular, the approach not only increases the speed of responsiveness of manufacturing systems to unpredicted events, such as sudden market demand changes or unexpected machine failures, but also facilitates a quick production launch of new products and flexible customization of existing products in the product family. Reasoning about its resources, the proposed

approach adjusts itself to provide exactly the functionality and production capacity needed, maximizing the system productivity with the available resources. In that sense, our work plays an important role towards Cognitive Technical Systems (CTS)—systems “that know what they are doing” [6].

To the best of our knowledge, there is no framework comparable to ours as a whole in the context of cognitive factories. However, there are contributions in the literature that are relevant to sub-components of our framework. For instance, in [7] a method is presented that integrates a global planning system with the domain specific machining planning system of [8] and an ad-hoc perception mechanism. In particular, the manufacturing domain is represented in PDDL and a classical planner is utilized to find a sequence of actions to reach the goal. The main role of the global planner is to augment the local machining plans with transportation and handling operations. The perception system is used for inspection of machined parts and triggers re-planning of the global planner when faulty parts are detected. Unlike our framework, this approach does not consider optimal planning for multiple teams of robots, cooperation of robots/teams, efficient use of resources, or diagnostic reasoning. Furthermore, in our study we utilize a highly expressive causality-based representation and reasoning formalism (action language $\mathcal{C}+$) that can handle concurrent actions by multiple agents, nondeterministic effects, multi-valued actions/fluent, additive fluents for reasoning about shared resources, state/transition constraints, and changes that do not involve actions (e.g., ramifications of actions), defaults, etc. Along with the capabilities of the causal reasoner (CCALC), our approach substantially extends the classes of problems that can be solved. In particular, we can solve not only planning but also prediction/postdiction and diagnosis problems using the same domain description.

There are recent contributions in the literature that are relevant to cognitive factories but not covered within the scope of this study. The readers are referred to the special issue of Advanced Engineering Informatics [9] on cognitive factories and the review article [10] for aspects of cognitive factories that complements our framework. The approaches for generative CNC machining planning using shape grammars and for automated fixture design to enable autonomous fabrication of customized part geometries [8], methods that enable human-robot cooperation in a cognitive factory setting [11], and the model-based approach that computes success probabilities of plans utilizing online observations [12] can be listed as representatives of these interesting contributions.

Execution monitoring framework for manipulation planning is an important part of cognitive factories, and there are several studies within this scope, even though they are not generalized to cognitive factories. In most plan execution monitoring systems in robotics and AI, the idea is to detect plan failures by comparing the measured outputs with the estimated outputs of the system, and to recover from these failures. Consider, for instance, the execution monitoring system PLANEX [13] used in SHAKEY, the procedural reasoning system (PRS) [14] used in the mobile robot FLAKEY, the reactive action packages system [15] used in the mobile robot CHIP, Livingstone control architecture [16] used in NASA's first New Millennium spacecraft, the execution monitoring system [17] on board with the mobile robot XAVIER [18], and the rationale-based monitoring system [19] applied in various domains. (See [20] for a survey of execution monitoring for robotics.)

Some of the more recent systems, such as [17,21–27], follow an alternative approach by lifting execution monitoring to a higher-level where monitoring condi-

tions are specified declaratively, failures are detected and classified/diagnosed by a reasoner, and a recovery from such failures is done, possibly by the same reasoner, considering temporal constraints. In particular, some of them describe execution monitoring in a logic-based framework for reasoning about actions, in which a planning problem can be formulated and solved. For instance, [25] and [27] describe execution monitoring in the situation calculus [28], which makes them applicable to Golog programs [29]. In [24], the authors describe execution monitoring in the fluent calculus [30] for FLUX programs [31]. [23] studies monitoring in a general action representation framework in second-order propositional logic, applicable in various reasoning systems such as CCALC. [21] and [26] describe monitoring conditions in a temporal logic formalism and check their correctness by a progression algorithm or model checker.

Some of these systems focus more on detecting faults (e.g., [23]), while some of them focus more on plan repair or failure recovery (e.g., [22, 32–35]). Some systems monitor the execution of the whole plan and some monitor some parts of the plan (e.g., PRS monitors “intentions” that can be viewed as subplans). Some systems check the observed real-world state with the estimated real-world state at every time step, whereas some systems perform this check from time to time and some systems monitor some world states only (e.g., the rationale-based monitoring system) and in particular undesirable behavior (as in [17]).

As for failure recovery, some of these systems use a set of precompiled recovery procedures to recover from failures (as in PRS), some of them replan from the current state to reach the goal, and some of them backtrack to an earlier state (e.g., “point of failure”) that caused the failure and do replan from that point on (e.g., [22]). For instance, in case of an execution failure, PLANEX continues to

execute the parts of the plan that are independent; if there is no such independent part, a new plan is generated. IPEM [32] integrates partial-order planning with plan execution: it starts with an incomplete plan and tries to reduce the “flaws” (i.e., a list of things to be done, such as “unexecuted action”, to complete the plan) at each step. XRFM [36] provides the continual modification of plans during their execution, using a rich collection of failure models and plan repair strategies: it projects a default plan into its possible executions, diagnoses failures of these projected plans by classifying them into a taxonomy of predefined failures, and then revises the default plan by following the pointers from the predefined failures to predefined plan repair strategies.

Finally, several alternative approaches have been proposed for modeling of cognitive manufacturing systems. In particular, in [37] the use of structured reactive controllers and transformational modeling are advocated, while in [38] hierarchical hybrid modeling and control are proposed as a viable solution. Note that, none of these approaches are comparable to our system, since they attack different challenges and emphasize different aspects of modeling of cognitive factories.

1.1 Contributions

We have introduced a new approach to manipulation planning and execution monitoring for multiple teams of robots that aim to complete a common goal in an optimal way. The contributions of this thesis can be summarized as follows:

- *Execution monitoring for a team of robots.* We have introduced a novel planning and execution monitoring framework for robotic manipulation planning. This framework can handle failures due to unknown obstacles or human intervention: when the plan fails, the failure is diagnosed using the sensor information; depending on the cause of the failure, the execution monitoring agent modifies the planning problem and asks the causal reasoner to find a different plan (e.g., that does not collide with a recently discovered obstacle). This execution monitoring framework has been implemented on a physical setup with two pantograph robots. The robots have to work together to complete the manipulation task.
- *Embedding high-level diagnostic reasoning in execution monitoring.* We have integrated high-level diagnostic reasoning in the framework above to recover from failures (e.g., broken robots) which can not be detected by sensors.
- *Decoupled planning for multiple teams of robots.* In a cognitive factory, each team has its own (manipulation) task. To facilitate the use of shared resources (e.g., robots), we have introduced a decoupled planning algorithm to find an overall optimal plan so that all the tasks of the teams are completed in minimum number steps. In this partially distributed algorithm, every

team plans for its own task, and a central agent communicates with every team orchestrating a more efficient use of shared resources.

- *Execution monitoring for multiple teams of robots.* We have extended the execution monitoring framework above to multiple teams of robots, where an optimal decoupled plan is found by the help of a central agent as described above. According to this extension, when a team fails executing its own plan, then the overall decoupled plan is modified by the central agent accordingly. We have used the physics simulator Gazebo to implement cognitive painting factory environment with multiple teams of robots.

1.2 Outline

In this thesis, a new approach for execution and monitoring framework is suggested for multiple teams of robots. Chapter 2 explains our high level representation formalism $\mathcal{C}+$ and casual reasoner CCALC. In chapter 3, we present our execution and monitoring framework for a single team of robots. Then we applied this framework on an assembly planning problem. In chapter 4 we extend our approach for multiple teams of robots and introduce a diagnosis algorithm for errors that we can not identify by sensor information. Last chapter concludes the thesis and suggests future work to develop this approach.

Chapter 2

Representation and Reasoning for a Dynamic Domain

We describe dynamic domains using the high-level representation formalism $\mathcal{C}+$ [4], and perform reasoning tasks in this domain using the causal reasoner CCALC [5].

2.1 Action Description Language $\mathcal{C}+$

We describe action domains in the action description language $\mathcal{C}+$, by “causal laws.” Let us give a brief description of the syntax and the semantics of $\mathcal{C}+$; we refer the reader to [4] for a comprehensive description.

We start with a (*multi-valued propositional*) *signature* that consists of a set σ of *constants* of two sorts, along with a nonempty finite set $Dom(c)$ of *value names*, disjoint from σ , assigned to each constant c . An *atom* of σ is an expression of the form $c = v$ (“the value of c is v ”) where $c \in \sigma$ and $v \in Dom(c)$. A *formula* of σ is a propositional combination of atoms. If c is a Boolean constant, we will use c

(resp. $\neg c$) as shorthand for the atom $c = True$ (resp. $c = False$).

A signature consists of two sorts of constants: *fluent constants* and *action constants*. Intuitively, fluent constants denote “fluents” characterizing a state; action constants denote “actions” characterizing an event leading from one state to another. A *fluent formula* is a formula such that all constants occurring in it are fluent constants. An *action formula* is a formula that contains at least one action constant and no fluent constants.

An *action description* is a set of *causal laws* of three sorts. *Static laws* are of the form

$$\mathbf{caused\ } F \mathbf{\ if\ } G \tag{2.1}$$

where F and G are fluent formulas. *Action dynamic laws* are of the form (2.1)

where F is an action formula and G is a formula. *Fluent dynamic laws* are of the form

$$\mathbf{caused\ } F \mathbf{\ if\ } G \mathbf{\ after\ } H \tag{2.2}$$

where F and G are as above, and H is a fluent formula. In (2.1) and (2.2) the part **if G** can be dropped if G is *True*.

The meaning of an action description can be represented by a “transition system”, which can be thought of as a labeled directed graph whose nodes correspond to states of the world and edges to transitions between states. Every state is represented by a vertex labeled with a function from fluent constants to their values. Every transition is a triple $\langle s, A, s' \rangle$ that characterizes change from state s to state s' by execution of a set A of primitive actions.

While describing action domains, we can use some abbreviations. For instance, we can describe the (conditional) direct effects of actions using expres-

sions of the form

$$c \text{ causes } F \text{ if } G \quad (2.3)$$

which abbreviates the fluent dynamic law

$$\text{caused } F \text{ if } True \text{ after } c \wedge G$$

expressing that “executing c at a state where G holds, causes F .”

We can formalize that F is a precondition of c by the expression

$$\text{nonexecutable } c \text{ if } \neg F \quad (2.4)$$

which stands for the fluent dynamic law

$$\text{caused } False \text{ if } True \text{ after } c \wedge \neg F.$$

Similarly, we can prevent the execution of two actions c and c' by the expression

$$\text{nonexecutable } c \wedge c'.$$

Similarly, we can express that F holds by default by the abbreviation

$$\text{default } F.$$

We can represent that the value of a fluent F remains to be true unless it is caused to be false, by the abbreviation

$$\text{inertial } F.$$

Notation: L ranges over $\{l1, l2\}$ and a ranges over action constants.

Action constants: Domains:
 $toggle(L)$ Boolean

Fluent constants: Domains:
 $up(L)$ Boolean
 $open$ Boolean

Causal laws:
 $toggle(L)$ **causes** $up(L)$ **if** $\neg up(L)$
 $toggle(L)$ **causes** $\neg up(L)$ **if** $up(L)$

caused $open$ **if** $up(l1) \wedge up(l2)$

inertial $open, \neg open$

inertial $up(L), \neg up(L)$

exogenous a

Figure 2.1.1: The suitcase domain described in C+.

Example – Suitcase domain

Consider, for instance, the suitcase domain introduced in [39]: there is a suitcase with two latches $l1$ and $l2$; when these two latches are up then the suitcase automatically opens. There are three propositional fluents: $up(L)$, where L is $l1$ or $l2$, and $open$; $up(L)$ holds iff latch L is up, $open$ holds iff the suitcase is open. There is an action of toggling a latch L denoted by $toggle(L)$. If a latch is down (resp. up) then it becomes up (resp. down) after toggling it. We can describe this domain in the action description language $\mathcal{C}+$ by the causal laws presented in Fig. 2.1.1. The first two lines of causal laws describe the direct effects of toggling the latches. The third line describes the indirect effects of toggling. The commonsense law of inertia is expressed by the next two lines. The last line expresses that actions are exogenous.

2.2 CCALC

The Causal Calculator (CCALC) [5] is a reasoning system, that performs reasoning tasks over an action domain description represented in a fragment of $\mathcal{C}+$ described above. To present formulas to CCALC, conjunctions \wedge , disjunctions \vee , implications \supset , negations \neg are replaced with the symbols $\&$ (or $\&\&$), $++$, $->$, and $-$ respectively. In most of the action descriptions, fluents are inertial and actions are exogenous; therefore, CCALC allows us to include this information at the very beginning of the action description while declaring fluent constants and action constants. For instance, the suitcase domain represented in $\mathcal{C}+$ in Fig. 2.1.1 is presented to CCALC as in Fig. 2.2.1.

In addition CCALC provides two facilities to be used in action domain descriptions: external predicates/functions and action attributes. External predicates/functions are not part of the signature of the domain description (i.e., they are not declared as fluents or actions). They are implemented as functions in some programming language of the user's choice, such as C++, and embedded in casual laws. External predicates take as input not only some parameters from the action domain description (e.g., the locations of robots) but also detailed information that is not a part of not the action domain description (e.g., geometric models). They are used to check externally some conditions under which the causal laws apply, or compute externally some value of a variable/fluent/action. For instance, suppose that the external predicate `collision(X, Y, X1, Y1)` (implemented in C++) checks whether the path between (X, Y) and $(X1, Y1)$ collides with an obstacle. Then we can express that there is no state at which the endpoints of a payload are located at (X, Y) and $(X1, Y1)$ where `collision(X, Y, X1, Y1)` holds:

```
caused false
  if xpos(r1)=X1 & ypos(r1)=Y1 &
    xpos(r2)=X2 & ypos(r2)=Y2
  where collision(X1,Y1,X2,Y2).
```

In addition, an external predicate can accomplish some other tasks as “side-effects.” For instance, while checking whether a robot located at (X, Y) collides with another robot at $(X1, Y1)$, the external predicate `collision(X, Y, X1, Y1)` can form a database keeping which locations lead to a collision and which locations do not. Then this database can be reused in the future.

Another useful feature of CCALC is its ability to represent “attributes of actions” that allows us to talk about various special cases of actions. Consider, for instance, the action of “a robot R picking a payload”. We can enforce that a robot R cannot pick a payload while moving by a causal law like

```
nonexecutable move(R) & pick(R).
```

However, when we want to specify some effects of these actions, we need to consider special cases of them. For instance, to express the effect of picking a payload, it may be useful to consider where the robot is picking the payload at. To express the effect of moving, it may be useful to consider in which direction and by what number of steps the robot is moving the payload. To denote these special cases of actions, we declare their attributes. For instance, for `pick`, we introduce an attribute `pickpoint` (as a function that returns which endpoint) after we declare `pick` as an exogenous action:

```
pick(robot)      :: exogenousAction;
```

```

:- sorts
latch.

:- objects
l1, l2 :: latch.

:- variables
L :: latch.

:- constants
up(latch), open :: inertialFluent;
toggle(latch)    :: exogenousAction.

% effects of toggling
toggle(L) causes up(L) if -up(L).
toggle(L) causes -up(L) if up(L).

% suitcase is open if
%   both of the latches are open
caused open if up(l1) & up(l2).

```

Figure 2.2.1: The suitcase domain described in the language of CCALC.

```

pickpoint(robot) ::
    attribute(endpoint) of pick;

```

and describe its effect (“robot R is holding a payload at its endpoint P”):

```

pick(R) causes holding(R,P)
    if pickpoint(R)=P.

```

By this way, additional special cases of an action can be defined without having to modify the definitions of more general actions. Note that such a representation of actions by special cases is a form of “hierarchical” representation of actions.

Once such an action domain description is given, we can perform various reasoning tasks via "queries" in an action query language, like the variation of the action query language \mathcal{Q} [40]. For instance we can present a query to CCALC as follows:

```
:- query
maxstep :: 2;
0: -up(l1), -up(l2), -open;
maxstep: open.
```

The query above describes the initial state at time step 0, and the last line describes the goal condition at time step $\text{maxstep}=2$.

Given a domain description and a query, CCALC checks whether the query is satisfied by the domain description (in the sense of satisfiability planning of [41]) as follows:

1. it transforms the causal laws into a propositional theory Γ_D , via "causal logic" [4],
2. it transforms the query into a propositional theory Γ_P ,
3. it checks whether $\Gamma_D \cup \Gamma_P$ is satisfiable;
4. if $\Gamma_D \cup \Gamma_P$ is unsatisfiable, it returns No;
5. otherwise, it returns Yes and presents an example extracted from a satisfying interpretation for $\Gamma_D \cup \Gamma_P$.

The transformations in the first two steps are different: the one in 1) is based on literal completion, whereas the one in 2) is based on a simpler procedure (see [4])

for a detailed description). Such a difference allows one to check the satisfiability of other queries (for instance, for replanning) without executing the first step again. Step 3) is done automatically by a state-of-the-art SAT solver, such as MINISAT [42] or its parallel variant MANYSAT [43].

CCALC allows us to compute shortest plan as well. To find a shortest plan, we modify the query above (let us label the modified query as ‘Query 1’):

```
:- query
label :: 1; % Query 1
maxstep :: 0..infinity;
0: -up(l1), -up(l2), -open;
maxstep: open.
```

With this query, CCALC successively tries to find a plan of length $\text{maxstep}=0, 1, \dots, \text{infinity}$. For Query 1, CCALC finds a shortest plan for $\text{maxstep}=1$ where both latches are toggled at time step 0:

```
0:
ACTIONS: toggle(l1) toggle(l2)
1:
```

CCALC can also show the complete history of this plan, including state information, if prompted:

```
0: -up(l1) -up(l2) -open
ACTIONS: toggle(l1) toggle(l2)
1: up(l1) up(l2) open
```

We can add some constraints to a planning problem specified as a query. For instance, we can ensure that CCALC finds the shortest plan, as shown in the following query:

```
:- query
label :: 2; % Query 2
maxstep :: 0..infinity;
0: -up(l1), -up(l2), -open;
maxstep: open;
[/\T | T<maxstep ->>
  (- (T: up(l2)) ->> - (T: toggle(l1)))].
```

With this modification, CCALC finds the following shortest plan instead:

```
0: -up(l1) -up(l2) -open
ACTIONS: toggle(l2)
1: up(l2) -up(l1) -open
ACTIONS: toggle(l1)
2: up(l1) up(l2) open
```

2.3 Why $\mathcal{C}+$ and CCALC?

We have decided to use the action description language $\mathcal{C}+$ to describe action domains due to its expressivity: we can formalize not only effects and preconditions of actions, but also state/transition constraints and changes that do not directly involve actions; we can represent not only deterministic effects but also

nondeterministic effects of actions. Also $\mathcal{C}+$ allows concurrency, unless specified otherwise via nonexecutability constraints.

We envision agents (robots) in a framework that has the capability of solving not only planning problems but also other reasoning tasks; since we aim endowing agents (robots) with various kinds of high-level reasoning mechanisms (such as prediction, postdiction, diagnosis, reasoning about shared resources, etc.) in the sense of cognitive robotics [44]. The action description language $\mathcal{C}+$, with the query language defined above, provides a common language for all these reasoning tasks, and thus allows us to setup such a framework.

CCALC can answer queries about a domain description represented in $\mathcal{C}+$ with respect to a reasoning task described in the query language above. Therefore, it allows us to solve different sorts of reasoning problems mentioned above (possibly with temporal constraints). Being able to add domain-specific temporal constraints as part of queries, for instance, allow us to do intelligent replanning to find different and “better” plans, as explained in the following sections.

Due to its modular structure and generic implementation, CCALC allows us to use various kinds of search engines to answer queries: CCALC supports SAT solvers such as MINISAT and the parallel SAT solvers like MANYSAT; the user can choose which search engine to use for answering which query. Due to well-studied relations between action languages and Answer Set Programming (ASP) [45], as in [46], we can also use efficient ASP solvers instead of SAT solvers, as in [47].

CCALC also supports external predicates/functions that can be implemented in some programming language of the user’s choice. These predicates/functions are important, for instance, in embedding low-level geometric reasoning in high-level reasoning, as explained in the following sections.

CCALC has other useful features/utilities as well: it supports additive fluents (to talk about the total effect of concurrently executing actions on numeric-valued fluents that denote shared resources), macros (to define complex notions succinctly, in some ways similar to “derived predicates”), attributes (to talk about special cases of actions). For more information about CCALC, we refer the reader to [4].

Chapter 3

Assembly Planning With Multiple Robots

Many existing planning and execution monitoring frameworks for robotic manipulation consider the classical 3-layer robot control architecture (Fig. 3.1.1) based on the observe-plan-act cycle, as discussed in Introduction. The idea is to compute a discrete task plan, and then to find continuous trajectories for the task plan, and then to execute these trajectories. If the execution of the plan fails, then the task planner is asked to replan. We extend this 3-layer architecture by making use of high-level causal reasoning at each level of the architecture as much as possible.

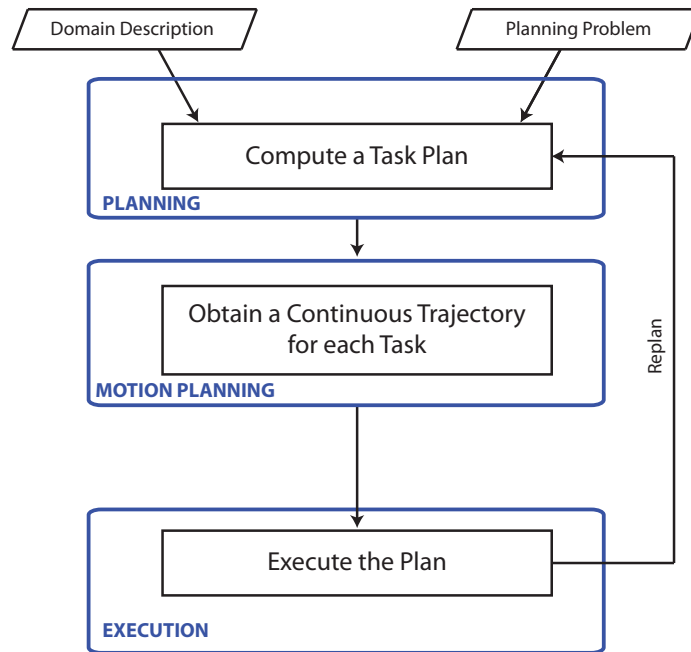


Figure 3.1.1: Classical 3-layer robot control architecture for planning, execution and monitoring.

3.1 A Planning and Execution Monitoring Framework for Multiple Robots

We consider robotic manipulation problems with multiple robots. Our aim eventually is to compute a complete continuous trajectory for each robot to reach a common goal in an optimal way, considering the possibility of concurrent executions of actions by multiple robots; and to ensure that the robots execute this plan robustly, considering the possibility of failures due to incomplete knowledge (such as unknown obstacles) or uncertainty (such as human intervention). We embed knowledge representation and automated reasoning in each level of the classical 3-layer robot control architecture (Fig. 3.1.2), in such a way as to tightly

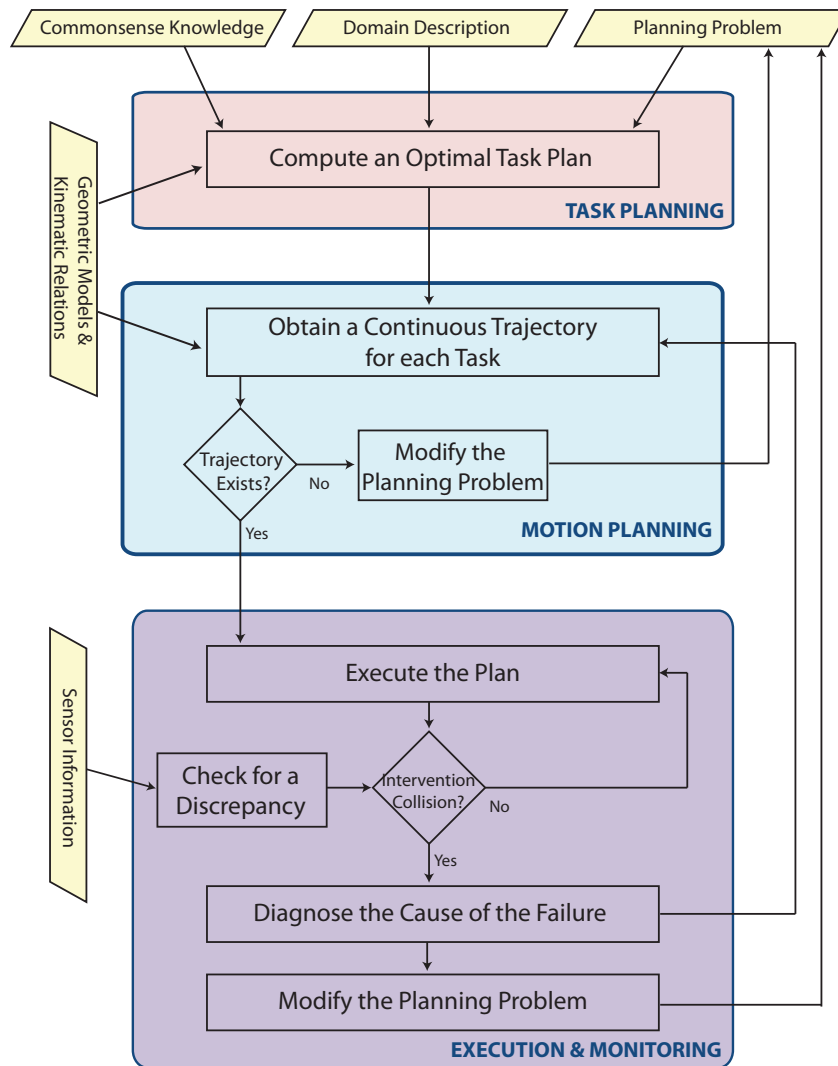


Figure 3.1.2: Proposed framework for planning, execution and monitoring.

integrate these layers.

Let us describe our execution and monitoring framework shown in Fig. 3.1.2 in more detail. We start with an action domain description and a planning problem description in the input language of CICALC, geometric models in VRML, and

kinematic relations as a C++ program.

- A description of geometric models include specifications of the geometry of robots, payloads, and other objects (e.g., obstacles) in the environment.
- Kinematic relations between robot end-effector configurations and robot joint configurations are implemented as functions in C++.
- An action domain description is a set of causal laws that express preconditions and direct effects of actions of robots, causal relations that do not involve these actions directly (e.g., ramifications), and state and transition constraints. These causal laws may include external predicates expressing conditions that involve geometric reasoning (as shown in Section 2) so that geometric models are also taken into account while a task plan is computed.
- A planning problem description is a set of formulas that express an initial state (or a set of initial states, in case of uncertainty), goal conditions, and temporal constraints.

Given an action domain description and a planning problem, first we compute an optimal plan (a sequence of actions) $\langle A_0, \dots, A_n \rangle$ and its complete history (including intermediate states) $\langle S_0, A_0, S_1, \dots, S_n, A_n, S_{n+1} \rangle$ using CCALC. The computed plan may involve concurrent execution of actions by multiple robots; so each A_i is a set of primitive actions. The optimality of a plan can be defined in terms of its length (the value of n) or the total cost of actions; in the following, we consider the former. Next, given a discrete plan and its history, and considering the given geometric models and kinematic relations, a collision-free trajectory T for each robot (if one exists) is computed by our motion planner, based on Rapidly

exploring Random Trees (RRT) [48]. Initially T is empty. For each transition $\langle S_i, A_i, S_{i+1} \rangle$ in the given history, the motion planner tries to compute a continuous trajectory T_i . If such a transition T_i is found then it is appended to the end of T .

If the motion planner fails to find a continuous trajectory for a transition, we identify the cause of that failure. There may be various kinds of failure with different causes. Depending on the cause of the failure, we modify the planning problem by adding domain-specific temporal constraints to avoid similar sorts of failure, as shown in Section 2. Afterwards, the modified planning problem is solved by CCALC, generating a different optimal task plan.

It is important to emphasize here two types of relations between different kinds of problem solving. First, the bilateral interaction between causality-based reasoning and motion planning: the causal reasoner guides the motion planner by finding an optimal task-plan; if there is no feasible kinematic solution for that task-plan then the motion planner guides the causal reasoner by modifying the planning problem with new temporal constraints. Second, the embedding of geometric reasoning in causal reasoning: while computing a task-plan, the causal reasoner takes into account geometric models and kinematic relations by means of external predicates implemented for geometric reasoning (e.g., to check some collisions); in that sense the geometric reasoner guides the causal reasoner to find feasible kinematic solutions. In the following, we illustrate these two aspects of our approach in more detail.

Once such a trajectory is computed, we can ensure that the robots follow this trajectory robustly by monitoring its execution.

3.1.1 Execution Monitoring

To ensure that a plan is executed safely without failure in a dynamic setting, we need to monitor its execution in systematic way, like in Algorithm 3.

According to this algorithm, if there is a plan failure, then first the cause of the failure is identified. Afterwards, depending on the cause of the failure, a recovery from the failure is achieved. For instance, suppose that a failure is detected during the execution of an action A_i at state S_i at time step i of a plan with the history $\langle S_0, A_0, S_1, \dots, A_n, S_{n+1} \rangle$. There are basically three type of failure:

- **Unknown obstacles:** If this failure is due to some unknown obstacle, then the robots shall go to a “safe” state (possibly the immediate previous state, State S_i), call the motion planner with a new configuration to obtain a feasible trajectory Π for the rest of the plan (with the history $\langle S_i, A_i, S_{i+1}, \dots, A_n, S_{n+1} \rangle$), and, if there is such a feasible trajectory computed by the motion planner, continue with the execution of this trajectory.
- **Intervention:** If the observed state is different than the expected state, then the robots shall go to a safe state and ask for a new hybrid plan from this state.
- **Diagnosis:** If the states are not changing as expected, and the failure can not be detected based on sensor information, then the cause of failure is detected by reasoning with the diagnosis algorithm (see Algorithm 4).

Note that these failures are generic, and our approach can be extended to solve any similar failures.

3.2 Example: Two Robots and Multiple Payloads

Consider two robots and multiple payloads on a platform. The payloads can be manipulated by the end effectors of the robots. In particular, the end-effector of each robot can pick (hold and elevate) or drop (release) the payload at one of its end points. None of the robots can carry the payload alone; the robots have to pick the payload at opposite ends. Since a payload is elevated from the platform when the robots are holding it, the payload can not collide with the other payloads. However, collisions between payloads may occur if a payload is dropped on top of another one and such collisions are not permitted. Similarly, other types of collisions (robot-robot, payload-obstacle and robot-obstacle) are not permitted either.

Initially, a configuration of the payloads on the platform is given (e.g., as in Fig. 3.4.1(a)). The goal is to reconfigure the payloads in an optimal manner (by minimum number of steps). This problem requires payloads to be picked and placed a number of times before they can be positioned into their final configuration. Due to the constraint that a payload can be carried by two robots only and due to the optimality of the plan, this problem requires true concurrency. Another challenge, meanwhile, is to avoid collisions of the payloads with each other. Also, other sorts of failure may occur while executing the plan, due to incomplete knowledge (such as unknown obstacles) or uncertainty (such as human intervention); in such cases, the robots should be able to recover from the failure if possible.

3.3 Embedding Geometric Reasoning in Action Domain Descriptions

Let us first describe the action domain of the example above, in the input language of CCALC (i.e., the action description language $\mathcal{C}+$).

We view the platform as a grid. We represent the robots by the constants r_1 and r_2 . We denote the payloads by nonnegative integers, and denote the endpoints of a payload i by the nonnegative integers $2i$ and $2i - 1$.

We characterize each robot by its end-effector, and describe its position by a grid point: the location (X, Y) of a robot R is specified by two functional fluents, $x_{pos}(R) = X$ and $y_{pos}(R) = Y$. Similarly, the location (X, Y) of an end point P_1 of the payload is specified by two fluents, $x_{pay}(P_1) = X$ and $y_{pay}(P_1) = Y$. Movements of a robot R in some direction D are described by actions of the form $move(R, D)$. Each such action has an attribute that specifies the number of steps to be taken by the robot. In addition, we denote the actions of picking and dropping a payload by $pick(R)$ and $drop(R)$; the former action has an attribute that specifies at which endpoint the robot picks the payload.

In the following, suppose that R denotes a robot, P_1 and P_2 denote the endpoints of a payload, N and N_1 range over nonnegative integers $1, \dots, \max N$, and D and D_1 range over all directions, *up*, *down*, *right*, *left*. Also suppose that X_1, X_2, Y_1, Y_2 range over integers $1, \dots, \max XY$.

3.3.1 Representing Actions and Change

We describe the preconditions and the effects of the actions as in [49]. For instance, we describe the direct effect of a robot's picking a payload, by the causal

laws

```
pick(R) causes holding(R,P)
  if pickpoint(R)=P.
```

These causal laws express that, after a robot R picks a payload at its endpoint P , the robot is holding it at its endpoint P .

One of the preconditions of the action of a robot R 's picking a payload at its endpoint P is that “ R should not be holding P ”; otherwise, the action is not executable. This is expressed by the causal laws

```
nonexecutable pick(R) if holding(R,P) .
```

We can also express conditions on the concurrent executability of actions. For instance, two robots R and $R1$ cannot pick a payload at the same endpoint:

```
nonexecutable pick(R) & pick(R1)
  if pickpoint(R)=pickpoint(R1) & R@<R1 .
```

A robot R cannot pick a payload while moving:

```
nonexecutable move(R,D) & pick(R) .
```

In addition to causal relations that involve actions, as in the preconditions and direct effects of actions above, we can also express causal relations that do not involve actions directly. For instance, if a robot R is holding a payload $P1$, then the location of the payload is the same as the location $(X1, Y1)$ of the robot.

```

caused xpay(P1)=X1
    if holding(R,P1) & xpos(R)=X1.
caused ypay(P1)=Y1
    if holding(R,P1) & ypos(R)=Y1.

```

Such causal laws allow us to reason about ramifications of actions without describing them: whenever a robot moves then the payload it holds moves as its indirect effect.

Finally, we can add state/transition constraints to ensure some conditions. For instance, we can prohibit states where the robots hold different payloads:

```

caused false
    if holding(R1,P1) &
        holding(R2,P2) &
        R1@<R2 & P1\=P2
    where -samePayload(P1,P2).

```

where `samePayload(P1,P2)` describes that the endpoints `P1` and `P2` belong to the same payload. Such causal laws allow us to reason about qualifications of actions without describing them: the robots cannot pick different payloads.

3.3.2 Embedding Geometric Reasoning in Causal Laws

We can embed geometric reasoning in such an action domain description in two ways: using state constraints and using external predicates.

Let us first consider collisions of payloads with each other. We can identify the conditions under which payloads collide with each other, provided that the orientations and the lengths of the payloads, as well as the positions of their leftmost

bottom endpoints are given. For instance, consider two payloads $K1$ and $K2$ of length $lengthP$ on the board, whose orientations are vertical. Suppose that the left bottom endpoints of these payloads are $(minXP(K1), minYP(K1))$ and $(minXP(K2), minYP(K2))$. Then these payloads collide with each other if $abs(minYP(K1) - minYP(K2)) \leq lengthP$. Once such collision conditions are identified, we can prevent them:

```

caused false
  if orientationP(K1)=v &
    orientationP(K2)=v & K1@<K2 &
    -beingCarried(K1) &
    -beingCarried(K2) &
    minXP(K1)=minXP(K2) &
    abs(minYP(K1)-minYP(K2))<lengthP.

```

Similarly, we can identify conditions for collisions between payloads with different orientations, and add causal laws to prevent such cases.

Next let us consider collisions between the robots, or between a robot and obstacles. To detect this sort of collisions, we need to know the geometric models and kinematic relations; however, such detailed information is not represented at the high-level (otherwise, if we could represent it, the domain description and thus the planning problem would be too large for the causal reasoner). Fortunately, CCALC supports external predicates (as explained in Section 2). For instance, we check whether a robot located at $(X1, Y1)$ collides with another robot at $(X2, Y2)$, by an external predicate `collision(X1, Y1, X2, Y2)` implemented as a C++ program. Then we can add causal laws to ensure that the robots do not collide with each other:

```

caused false
  if xpos(r1)=X1 & ypos(r1)=Y1 &
     xpos(r2)=X2 & ypos(r2)=Y2
  where collision(X1,Y1,X2,Y2).

```

While checking whether a robot located at $(X1, Y1)$ collides with another robot at $(X2, Y2)$, the external predicate `collision(X1, Y1, X2, Y2)` forms a database keeping which locations lead to a collision and which locations do not. This database can be reused in the future.

3.4 Bilateral Interaction between Causal Reasoning and Motion Planning

With the action domain description and the external predicate above, CCALC combines causal reasoning with geometric reasoning to compute task plans without robot-robot or robot-obstacle collisions. For instance, consider the environment in Fig. 3.4.1. Suppose that initially the robots `r1` and `r2` are at $(1, 1)$ and $(9, 9)$ respectively; the first payload is located at $(3, 2)$ and $(8, 2)$; the second payload is located at $(8, 5)$ and $(3, 5)$; the third payload is located at $(9, 3)$ and $(9, 8)$. The goal is to move the payloads to the following locations: first payload to $(3, 2)$ and $(3, 7)$; the second payload to $(6, 7)$ and $(6, 2)$; the third payload to $(9, 8)$ and $(9, 3)$. This planning problem can be described in the language of CCALC by means of a “query” as follows:

```

:- query % Query 1
maxstep:: 0 ..infinity;

```



```

0: % Initial state
% robot 1
xpos(r1)=1, ypos(r1)=1,
% robot 2
xpos(r2)=9, ypos(r2)=9,
% endpoints (1 and 2) of payload 1
xpay(1)=3, ypay(1)=2,
xpay(2)=8, ypay(2)=2,
% endpoints (3 and 4) of payload 2
xpay(3)=8, ypay(3)=5,
xpay(4)=3, ypay(4)=5,
% endpoints (5 and 6) of payload 3
xpay(5)=9, ypay(5)=3,
xpay(6)=9, ypay(6)=8;
maxstep: % Goal conditions
% endpoints of payload 1
xpay(1)=3, ypay(1)=2,
xpay(2)=3, ypay(2)=7,
% endpoints of payload 2
xpay(3)=6, ypay(3)=7,
xpay(4)=6, ypay(4)=2,
% endpoints of payload 3
xpay(5)=9, ypay(5)=8,
xpay(6)=9, ypay(6)=3,
% robots must not be holding any payloads

```

```
[/\R /\P | -holding(R,P)].
```

This query, Query 1, asks for a plan with the minimum number of time steps. CCALC then computes the following plan (Plan 1) of length 27 for this problem:

```
0:  move(r1, up, steps=3)
    move(r1, left, steps=1)
    move(r2, down, steps=4)
...
13: pick(r1, pickpoint=1)
    pick(r2, pickpoint=2)
14: move(r1, down, steps=2)
    move(r1, right, steps=2)
    move(r2, up, steps=3)
    move(r2, left, steps=2)
...
26: drop(r1)
    drop(r2)
```

Note that each step of the plan involves concurrent execution of a set of primitive actions. For instance, at time step 13, both robots pick the opposite endpoints 1 and 2 of the payload 1 at the same time. At time step 14, the robot *r1* moves down by two units and right by two units at the same time (note that this concurrent action essentially describes a diagonal move of the robot); meanwhile, the robot *r2* moves up by three units and left by two units.

For instance, consider the transition $\langle S_3, A_3, S_4 \rangle$ from the history of Plan 1:

```

3: holding(r1,4), holding(r2,3),
   xpos(r1)=3, ypos(r1)=5,
   xpos(r2)=8, ypos(r2)=5,
   move(r1, down, steps=3)
   move(r1, right, steps=1);
4: xpos(r1)=4, ypos(r1)=2,
   xpos(r2)=8, ypos(r2)=5,...;

```

where the end-points of the second payload (that the robots are holding) are at (3, 5) and (8, 5). The motion planner cannot find a continuous trajectory for this transition since the payload collides with an obstacle at Step 4 (third kind of failure). Then, Query 1 is modified by adding a constraint as follows:

```

:- query % Query 2
maxstep :: 0..infinity;
0: ...; % Initial states
maxstep: ...; % Goal conditions
% Constraints
T<maxstep ->>
  -((T: holding(r1,4)) &&
    (T: holding(r2,3)) &&
    (T: xpos(r1)=4) && (T: xpos(r2)=8) &&
    (T: ypos(r1)=2) && (T: ypos(r2)=5)).

```

to ensure that CCALC does not consider S_4 as a possible state. Then, after two more tries, CCALC computes a different plan (Plan 2) without such a failure. The

complete task plan is presented in Appendix A.

```
0: move(r1, up, steps=1)
    move(r1, right, steps=1)
    move(r2, down, steps=2)
    move(r2, right, steps=1)
1: move(r1, up, steps=3)
    move(r1, right, steps=1)
    move(r2, down, steps=2)
    move(r2, left, steps=2)
2: pick(r1, pickpoint=4)
    pick(r2, pickpoint=3)
3: move(r1, down, steps=1)
    move(r1, right, steps=3)
    move(r2, up, steps=2)
    move(r2, right, steps=2)
...
26: drop(r1)
    drop(r2)
```

Thus, for each action of this plan, the motion planner can find a continuous collision-free trajectory (Fig. 3.4.1).

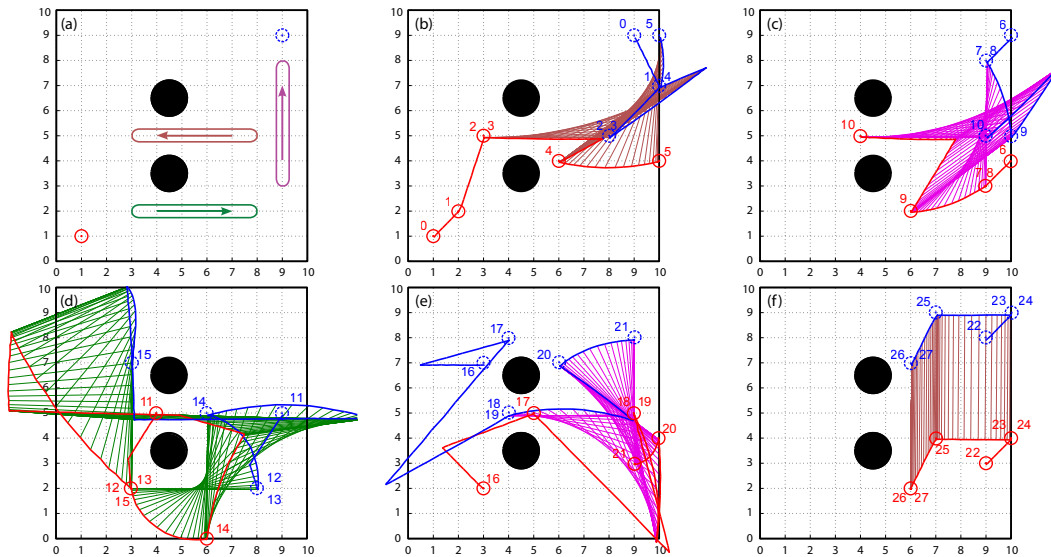


Figure 3.4.1: (a) presents the initial state, while (b)–(f) illustrate the execution of Plan 2. Colors red and blue are associated with Robots 1 and 2. Circles indicate the positions of the robot end-effectors and circles’ labels denote the time steps. Solid red and blue lines denote the trajectories of robot end-effectors. Green, red and magenta lines denote Payloads 1–3. Black disks represent the obstacles. For instance, at Step 3, end-effectors of Robots 1 and 2 are located at $(3, 5)$ and $(8, 5)$ respectively, and robots hold Payload 2. At Step 4, end-effectors of Robots 1 and 2 are located at $(6, 4)$ and $(10, 7)$, still holding Payload 2. Trajectory of Payload 2 moving from Step 3 to 4 is depicted in brown.

3.5 Case Studies

We have tested the applicability and effectiveness of our planning and monitoring framework using two pantograph robots (two degrees-of-freedom planar parallel manipulators) to perform a complex manipulation task that requires concurrent execution of actions. In particular, we used symmetric wooden sticks with metal tips as payloads. To enable pick and drop actions, we equipped the end-effectors

of the pantograph robots with linear servo motors with magnetic tips acting out of plane. The magnetic tip of the linear servo motors can pick (hold and elevate) a payload at one of its end points. Similarly, the payload can be dropped by retracting the magnetic tip inside its case. The robots were closed loop controlled at 100 Hz to ensure robust trajectory tracking of their end-effectors. The controllers of the robots have been implemented on a PC-based control architecture, that comprises of a PCI I/O card and a workstation, simultaneously running RTX real-time operating system and Windows XP SP2.

In the following, we present three sample scenarios in this setting. Scenario 1 involves no failures due to incomplete knowledge (such as unknown obstacles) or uncertainty (such as human interventions). Scenario 2 involves a collision of a payload (while being carried) with an unknown obstacles. Scenario 3 involves a human intervention that changes the position of a payload.

Scenario 1 – No surprises

Consider the example from the previous section, where Plan 2 is a collision-free plan. Fig. 3.4.1 depicts the experimentally recorded trajectories of the robots during this plan execution, Fig. 3.5.1 shows the snapshots of the plan execution. Fig. 3.4.1(a) depicts the initial configuration while the first six steps of the plan execution are shown in Fig. 3.4.1(b). Observe that the successful completion of this plan necessitates payloads to be picked and dropped a number of times before they can be arranged to their final configuration. For instance, the robots first pick Payload 2 and drop it to an intermediate location (Fig. 3.4.1(b)), then move Payload 3 to an intermediate location (Fig. 3.4.1(c)), then place Payloads 1 and 3 into their goal positions (Fig. 3.4.1(d) and (e)), and finally move Payload 2 to its final

position (Fig. 3.4.1(f)).

Scenario 2 – Unknown obstacles

Suppose that there is an obstacle outside of the grid but just next to the points $(0, 6)$ and $(0, 7)$, and that the robots do not know about the presence of this obstacle. Suppose also that the robots are executing Plan 2. Experimentally recorded trajectories of the robots is given in Fig. 3.5.2 and the snapshots of plan execution are presented in Fig. 3.5.3. At time step 14, there is an action failure because the payload collides with the unknown obstacle (Fig. 3.5.2(a)). Following Algorithm 3, the robots identify the cause of this failure as “unknown obstacles”, goes back to a safe state, in this case the previous state S_{14} , and calls the motion planner with an updated configuration (Fig. 3.5.2(b)). The motion planner finds a different trajectory for the transition $\langle S_{14}, A_{14}, S_{15} \rangle$ that does not collide with the unknown obstacle. After the robots follow this new trajectory they continue with execution of the rest of the task plan as computed earlier (Fig. 3.5.2(c)–(e)).

Scenario 3 – Human interventions

Suppose that the robots are executing Plan 2 and at time step 24, someone changes the position of Payload 3. The experimentally recorded trajectories for this scenario are shown in Fig. 3.5.4, the snapshots of plan execution are given in Fig. 3.5.5. In particular, when intervention takes place at time step 24 (Fig. 3.5.4 (a)), following Algorithm 3, the robots identify the cause of this failure as “human intervention”, goes back to a safe state, in this case the previous state S_{23} , and calls the task planner with a new initial state (Fig. 3.5.4 (b)). The task planner finds a different

task plan, Plan 3, (and its corresponding trajectory) from that point on to reach the goal:

```
0: move(r1, left, steps=3)
    move(r2, left, steps=3)
1: move(r1, down, steps=1)
    move(r1, left, steps=1)
    move(r2, down, steps=1)
    move(r2, left, steps=1)
2: pick(r1, pickpoint=6)
    pick(r2, pickpoint=5)
3: move(r1, right, steps=3)
    move(r2, right, steps=3)
4: drop(r1)
    drop(r2)
5: move(r1, up, steps=1)
    move(r1, right, steps=1)
    move(r2, up, steps=1)
    move(r2, right, steps=1)
6: pick(r1, pickpoint=4)
    pick(r2, pickpoint=3)
7: move(r1, down, steps=3)
    move(r1, left, steps=2)
    move(r2, down, steps=3)
    move(r2, left, steps=2)
8: move(r1, up, steps=1)
```



```
move(r1, left, steps=2)
move(r2, up, steps=1)
move(r2, left, steps=2)
9: drop(r1)
   drop(r2)
```

The robots follow Plan 3 and its corresponding trajectory to reach the goal state (Fig. 3.5.4 (c)-(d)).

3.6 Discussion

In this chapter we have presented a modular framework to monitor the execution of a plan computed with hybrid planning approach. The main contributions of this chapter can be summarized as follows:

Hybrid plan repair/recovery relative to failures. The planning and monitoring framework introduced in this chapter considers two sorts of failures that may take place in robotic manipulations, and decides for a recovery or plan repair based on the kind of failure. Unlike many existing approaches to monitoring, plan repair or recovery from failures does not only rely on motion planning but also causal reasoning as well.

Temporal monitoring conditions. Due to the use of a causal reasoner, the execution monitoring framework can specify some monitoring conditions as temporal constraints while asking for a new task plan once human intervention is detected. In that sense, execution monitoring is lifted to high-level.

We have illustrated the usefulness of this approach with a physical implementation of a problem that involves two robots working for a common goal. We have considered three scenarios of plan execution:

No surprises. The robots have a complete knowledge of the environment and nothing unexpected occurs in the domain while a plan is being executed.

Collisions with unknown obstacles. The robots do not have a complete knowledge of the obstacles in the environment and thus collide with such an unknown while executing a plan.

Human interventions. While the robots are executing a plan, a human intervenes and change the locations of some payloads.

In each scenario, we have shown how the robots find and execute a plan using our plan execution and monitoring algorithm.

Algorithm 1 TASK&MOTION_PLAN

Input: Action domain description \mathcal{D} , and planning problem \mathcal{P} with the initial state(s) S and the goal G

while true do

$plan, P \leftarrow$ Compute a shortest task plan P of length n (within a history $H = \langle S_0, A_0, S_1, \dots, S_n, A_n, S_{n+1} \rangle$, where $S_0 = S$ and $S_{n+1} = G$) using CCALC with \mathcal{D} and \mathcal{P} (if there is such a plan);

if $\neg plan$ **then**

return *false*;

$T := \langle \rangle$; // Initially the trajectory is empty

$trajectoryFound := true$;

$i := 0$;

while $trajectoryFound$ **do**

$\langle S_i, A_i, S_{i+1} \rangle \leftarrow$ Extract from H the next transition;

 // Compute a trajectory π for $\langle S_i, A_i, S_{i+1} \rangle$, if one exists

$trajectoryFound, \pi \leftarrow$ MOTION_PLAN(S_i, S_{i+1});

if $\neg trajectoryFound$ **then**

$\mathcal{P} \leftarrow$ Identify the cause of the failure and modify the planning problem \mathcal{P} accordingly;

else

$T \leftarrow$ Append π to T ;

if A_i is the last action **then**

return *true, P, H, T*;

$i++$;

Algorithm 2 MOTION_PLAN

Input: Initial state S and goal state G

// C, t, d denote the configuration space, a specified timeout and a specified distance

$s \leftarrow$ Find initial configuration in C that corresponds to S ;

$g \leftarrow$ Find goal configuration in C that corresponds to G ;

$V := \{\langle s, 1 \rangle, \langle g, 2 \rangle\}$; // The roots of Tree 1 and Tree 2

$E := \emptyset$; // Empty set of undirected edges in these trees

$connected := false$;

while $\neg connected$ and the timeout t is not exceeded **do**

$p \leftarrow$ Sample a random point in C ;

if p is collision-free **then**

$p_1 \leftarrow$ Find the closest point to p in V with label 1;

$p_2 \leftarrow$ Find the closest point to p in V with label 2;

for $i = 1, 2$ **do**

$q_i \leftarrow$ Find the point in V with a distance of d from p_i in the direction of p ;

if the path connecting p_i and q_i is collision-free **then**

$V := V \cup \{\langle q_i, i \rangle\}$; // Expand Tree i with q_i

$E := E \cup \{\{p_i, q_i\}\}$;

if both the path connecting p and p_1 , and the path connecting p and p_2 are collision-free **then**

$V := V \cup \{\langle p, _ \rangle\}$; // Add p with an arbitrary label

$E := E \cup \{\{p, p_1\}, \{p, p_2\}\}$; // Connect the two trees

$connected := true$;

if $connected$ **then**

$\pi \leftarrow$ Extract the trajectory from $\langle V, E \rangle$;

return $true, \pi$;

else

return $false$; // No trajectory

Algorithm 3 EXECUTE&MONITOR

Input: An action domain description \mathcal{D} , a planning problem \mathcal{P} with the initial state(s) S and the goal G
 // Find a task plan P with history H and trajectory Π
 $planFound, P, H, \Pi \leftarrow \text{TASK\&MOTION_PLAN}(\mathcal{D}, \mathcal{P});$
if $\neg planFound$ **then**
 Abort; // no plan to execute
 $executionCompleted := false;$
while $\neg executionCompleted$ **do**
 $\langle S, A, S' \rangle \leftarrow$ Extract from H the next transition;
 $\pi_A \leftarrow$ Extract from Π the trajectory for $\langle S, A, S' \rangle;$
 $failureDetected \leftarrow$ Execute A by following π_A at the current state S , and
 meanwhile check for any interventions or unknown obstacles;
 if $\neg failureDetected$ **then**
 if A is the last action of P **then**
 $executionCompleted := true;$
 else
 $I \leftarrow$ Find a set of possible safe states close to $S;$
 if $I == \emptyset$ **then**
 Abort; // no solution
 else if $S \in I$ **then**
 Go back to state $S;$
 $failureCause \leftarrow$ Identify the cause of the failure;
 if $failureCause$ is “Unknown Obstacle” **then**
 $trajectoryFound, \Pi \leftarrow$ Find a new trajectory for the rest of the plan P
 starting from S
 if $\neg trajectoryFound$ **then**
 $\mathcal{P} \leftarrow$ Identify the cause of the failure and modify the planning
 problem \mathcal{P} accordingly by adding temporal constraints;
 EXECUTE&MONITOR(\mathcal{D}, \mathcal{P});
 else if $failureCause$ is “Human Intervention” **then**
 $\mathcal{P} \leftarrow$ Modify the planning problem \mathcal{P} accordingly by adding tem-
 poral constraints;
 EXECUTE&MONITOR(\mathcal{D}, \mathcal{P});
 else
 Pick a safe state S'' in $I;$
 $\mathcal{P} \leftarrow$ Modify the planning problem \mathcal{P} accordingly by adding temporal
 constraints and setting the initial state as $S'';$
 EXECUTE&MONITOR(\mathcal{D}, \mathcal{P});
 return ; // plan successfully executed

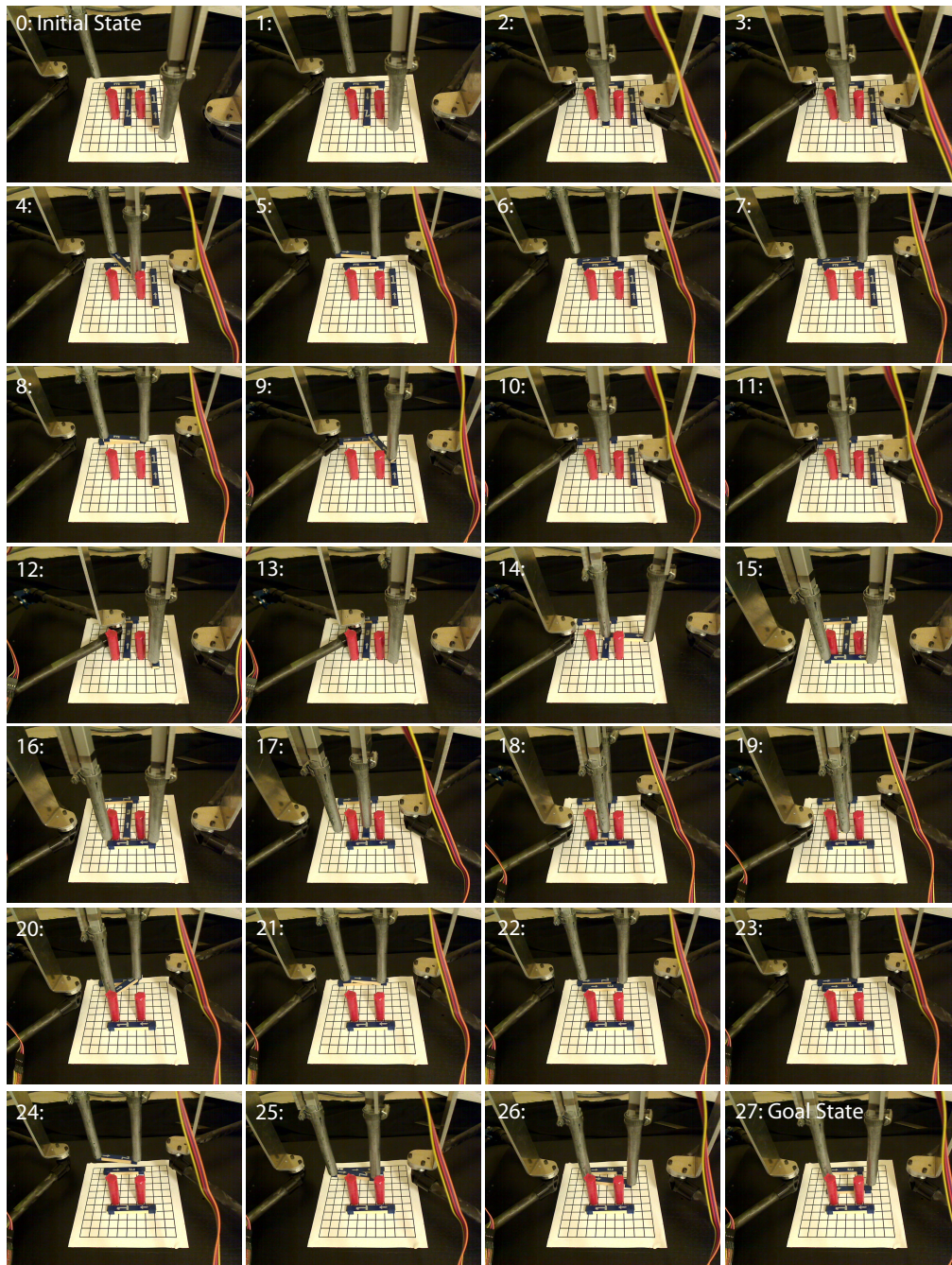


Figure 3.5.1: Snapshots of execution of Plan 2. Note that the camera is rotated 90° counterclockwise with respect to the trajectory plots in Fig. 3.4.1 for a less occluded view.

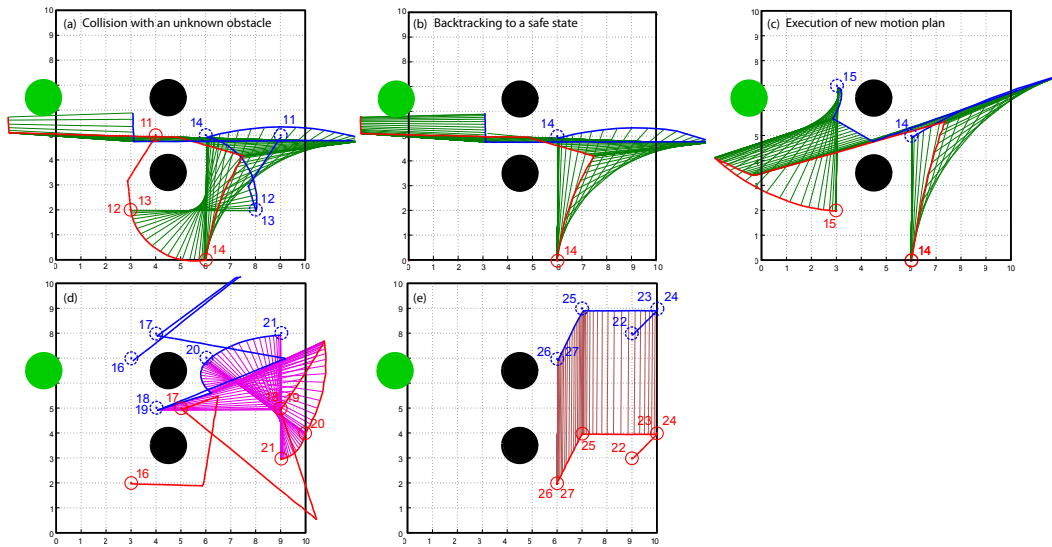


Figure 3.5.2: (a)–(e) present execution of Plan 2 from step 11, where there is an unknown obstacle (green disk) just outside the workspace, next to the points $(0, 6)$ and $(0, 7)$, interfering with the motion of the robots. Colors red and blue are associated with Robots 1 and 2. Circles indicate the positions of the robot end-effectors and circles' labels denote the time steps. Solid red and blue lines denote the trajectories of robot end-effectors. Green, red and magenta lines denote Payloads 1–3. Black disks are the known obstacles, while the green disk represents the unknown obstacle. At Step 14, there is an action failure because the payload collides with the unknown obstacle (Fig. 3.5.2 (a)). In (b) the robots identify the cause of failure and backtrack to a safe state. A new motion plan is found and executed in (c). The execution of rest of the plan is presented in (d)–(e).

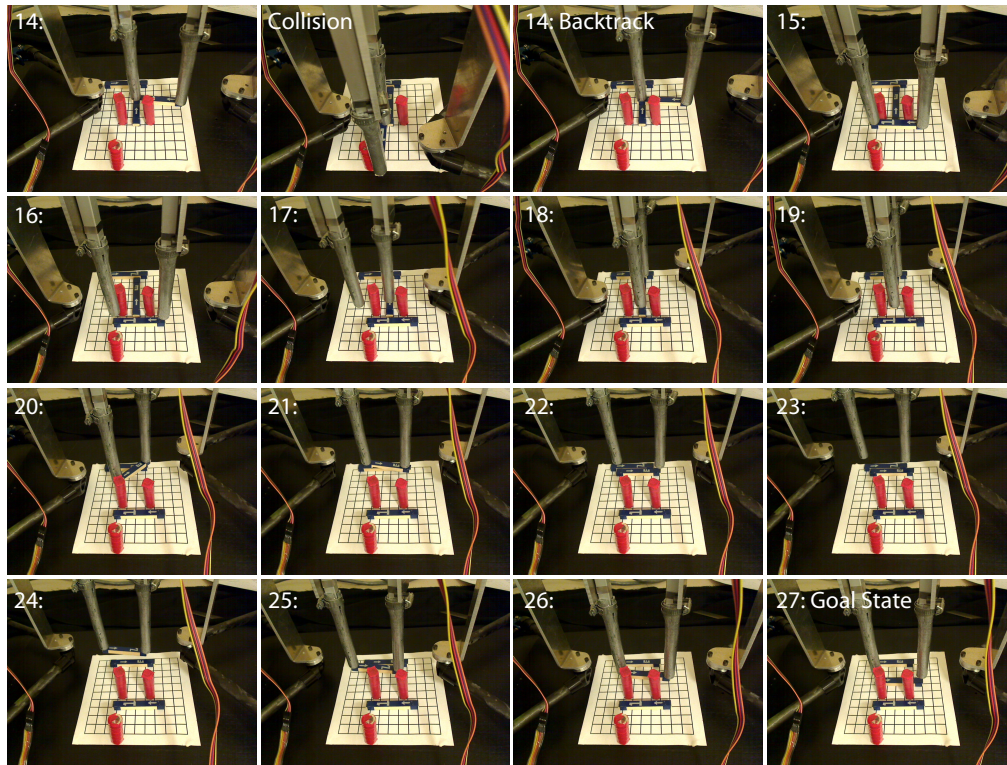


Figure 3.5.3: Snapshots of execution of Plan 2, starting from time step 14. At time step 14, the payload collides with the obstacle. Note that the camera is rotated 90° counterclockwise with respect to the trajectory plots in Fig. 3.5.2 for a less occluded view.

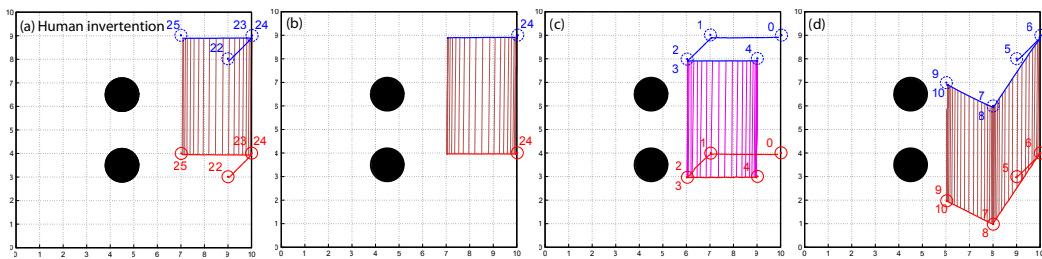


Figure 3.5.4: (a)–(d) present execution of Plan 2 from step 22. Colors red and blue are associated with Robots 1 and 2. Circles indicate the positions of the robot end-effectors and circles’ labels denote the time steps. Solid red and blue lines denote the trajectories of robot end-effectors. Green, red and magenta denote Payloads 1–3. Black disks represent known obstacles. At Step 24, there is a human intervention and the position of Payload 3 is changed (Fig. 3.5.4 (a)). In (b) the robots identify the cause of failure and replan accordingly. A new task plan and corresponding trajectories are found and executed in (c)–(d).

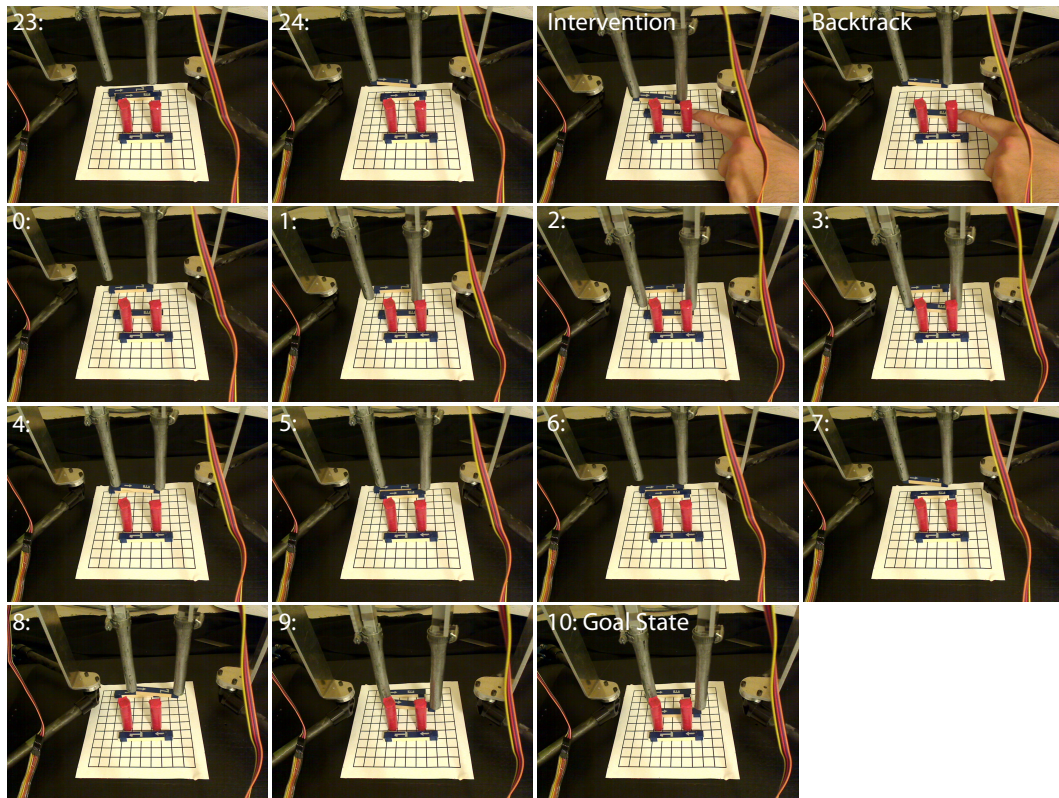


Figure 3.5.5: Snapshots of execution of Plan 2, starting from time step 23. At time step 24, there is a human intervention: someone changes the position of Payload 3. After putting Payload 2 to a safe position, the robots continue with the execution of the new task plan, Plan 3. Note that the camera is rotated 90° counterclockwise with respect to the trajectory plots in Fig. 3.5.4 for a less occluded view.

Chapter 4

Cognitive Factories With Multiple Teams Of Robots

So far, we have discussed execution and monitoring of a plan by a single team of robots. Now, consider multiple teams of robots, possibly sharing some resources, trying to achieve a common goal. This problem is more challenging: On the one hand, each team tries to complete its task as early as possible. On the other hand, all the teams try to use the shared resources as efficiently as possible so that all the tasks for all teams are completed as soon as possible. One approach to solve such a problem would be to model the whole domain (including all the workspaces of all teams) as a single planning problem and employ a planner to find optimal plans. Then, the execution and monitoring framework presented in the previous section can be utilized without any modifications. However, as the problem size gets larger, solving the planning problem and monitoring the execution of the overall plan becomes computationally intractable.

4.1 A Planning and Execution Monitoring Framework For Multiple Teams of Robots

We introduce a novel framework, shown in Fig. 4.1.1, where each team autonomously finds its own plan and monitors its execution, and where a central agent ensures optimality of the overall plan by orchestrating efficient use of shared resources through communication with the teams.

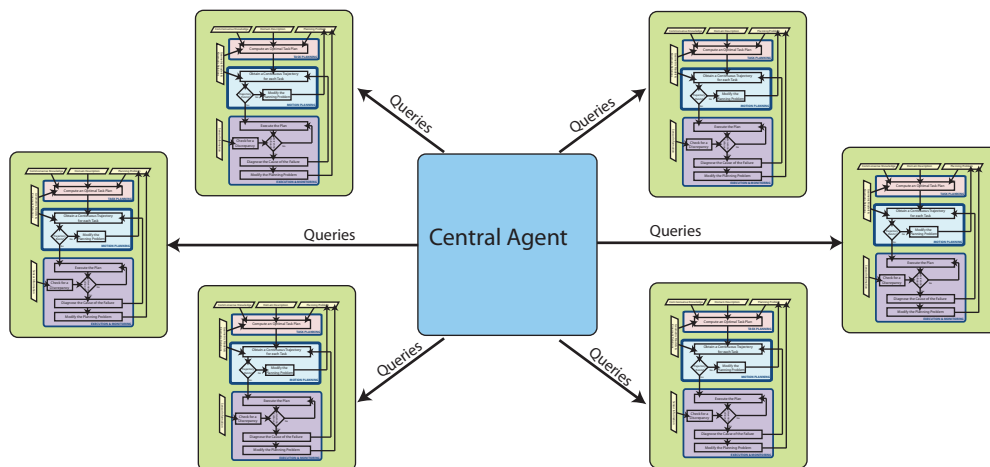


Figure 4.1.1: Decoupled planning, execution and monitoring framework

Suppose that the goal is to complete all the assigned tasks of all the teams in a minimum number of steps, under the assumption that teams can exchange robots. For a plan length k , a team is a lender if it can complete its task on its own in k steps; a borrower if it can not complete its task on its own in k steps. Assuming that a team can not lend a robot and borrow a robot in a plan and that a team can not lend or borrow more than one robot, we design an intelligent algorithm that finds an optimal decoupled plan by efficiently using the shared resources. This

algorithm relies on the following three sorts of queries (asked to each team) to decouple plans and to orchestrate robot exchanges among the teams:

- Can the goal be achieved in k steps, without any robot exchanges?
- Can the goal be achieved in k steps, while lending a robot before step k_0 ?
- Can the goal be achieved in k steps, while borrowing a robot after step k_0 ?

In decoupled planning, the goal is to match each borrowing team with a different lending team so that the matched teams agree on lend/borrow times. Let $lend_i$ denote the time step at which Team i can lend a robot (i.e., Team i answers the first query above affirmatively for $k = lend_i$). Let $borrow_i$ denote the time step at which Team j can borrow a robot (i.e., Team j answers the second query above affirmatively for $k = borrow_i$). Let $transport$ denote the transportation delay. Then, there is a matching between the lending team and the borrowing team, if $lend_i + transport \leq borrow_j$. Based on this observation, the optimal decoupled planning algorithm uses binary search for each team to find valid lend/borrow times. Details of this algorithm are discussed in [50].

Once a decoupled plan is found, each team starts its own plan and monitors its execution. When a failure is detected, then the central agent is asked to coordinate replanning.

4.2 A Cognitive Painting Factory Scenario

Consider a Painting Factory with multiple teams of robots, where each team is located in separate workspace and each workspace produces different colored boxes. To complete the task each box should be painted, waxed and stamped in every

workspace. The teams are heterogeneous. Each team is composed of two types robots with different capabilities: worker robots and a single carrier robot.

Worker robots can move horizontally and change their end-effectors to complete different tasks such as painting, waxing or stamping. Carrier robots can move both horizontally and vertically, and push or pull the worker robots to make them move vertically. Each workspace is depicted as a grid and contains an assembly line along one of the walls to carry the boxes and a pit stop area where worker robots can change their end-effectors. To make more efficient use of shared resources, teams can exchange robots: a team can give a worker robot at any step through their pit stop and another team can receive it through its pit stop after a transportation delay. While executing the task, the robots may be broken, or a new task (e.g., the number of boxes may be increased due to a new order) may be assigned to teams.

The teams act as autonomous cognitive agents; therefore, each team makes its own plan to complete its own designated task. Given the initial state of each workspace and the designated tasks for each team (e.g., how many boxes of which colors to paint), the goal is for all the teams to complete these tasks in a minimum number of steps. Since teams can lend or receive worker robots from other teams and each team makes its own plan, we use the proposed optimal decoupled planning algorithm to reach this goal.

During a plan execution, if any unexpected event occurs, as mentioned above, the goal is to diagnose the cause of the failure or discrepancy, and find an optimal (decoupled) plan for recovery. We propose to reach this goal by means of a diagnosis algorithm.

4.3 High Level Representation of the Factory

We represent a cognitive factory in the high-level action language $\mathcal{C}+$ [4], which is a logic-based formalism based on causality. We can describe preconditions and (conditional) effects of actions, as well as indirect effects of actions and static/dynamic constraints with this language. Also we can describe true concurrency (where actions cannot be serialized) and nondeterministic effects of actions (where we are not sure about the outcome of an action).

Consider, for instance, a painting factory where each workspace is viewed as a grid. Let us represent the carrier robot by the constant $c1$; n worker robots by the constants $w1, w2 \dots wn$; and each one of b boxes with a distinct number in $\{1, \dots, b\}$.

We consider the following fluents to represent this factory:

- The robots are supposed to be located at grid squares; therefore, the location of a robot R is specified by two functional fluents, $xpos(R)=X$ and $ypos(R)=Y$.
- The boxes are supposed to be located in some order on the assembly line; therefore, the location L of a box B on the assembly line is specified by a single fluent $linePos(B)=L$.
- The status of a box B is denoted by the functional fluent $workDone(B)=WS$ where WS stands for a work stage: 0–unprocessed, 1–painted, 2–waxed, 3–stamped.
- A newly painted box is wet and it has to be left to dry before it can be waxed; to formalize this transition constraint, we need a relational fluent

`wetpaint(B)` to show that box B is wet.

- The functional fluent `endEffector(W)=E` denotes that the worker robot W has the end-effector E ; the value of E denotes the role of the worker relative to the work stage: 1–painter, 2–waxer, 3–stamper.
- The carrier robot needs to attach to and detach from the worker robots, to be able to push or pull them along the vertical axis. Therefore, we need a relational fluent `attached(C,W)` to express that the carrier robot C is attached to the worker robot W .
- For robot exchanges between teams, we consider a relational fluent `bench(W)` to describe that a worker robot W is at the bench area and ready for an exchange.

We also consider the following actions:

- A robot R (which may be a worker W or carrier C) can move in the direction D by one unit; we denote this action by `move(R,D)`.
- A worker robot W can perform the following actions: `swapEndEffector(W,E)` – changing its end-effector to E , `workOn(W,B)` – working on a box B to proceed to the next work stage.
- A carrier robot can perform the following actions: `attach(C,W)` – attaching to a worker robot W , `detach(C)` – detaching from the worker robot it is attached to, `push(C)` – pushing the worker robot (it is attached to) vertically by one unit, `pull(C)` – pulling the worker robot (its is attached to) vertically by one unit.

- In addition to the actions of robots in a team, there is also the action of shifting the assembly line, denoted by `lineShift`.
- For robot exchanges between teams, we introduce two actions, namely `giveRobot(W)` which puts `W` to the bench (`W` needs to be in the pit stop area first) and `takeRobot(W, X, Y)` which takes `W` from the bench and puts it at `(X, Y)` in the pit stop area.

In $\mathcal{C}+$, we describe actions and change by “causal laws.” Consider, for instance, the action `workOn(W, B)`. We formalize by the following causal law that this action, as its direct effect, increments the work stage `WS` of a box `B` if the worker robot `W` is working on `B`:

```
workOn(W, B) causes workDone(B)=WS
  if workDone(B)=WS-1.
```

The action `workOn(W, B)` denotes painting if the current work stage is 0, i.e., `workDone(B)=0`. Therefore, we formalize that painting a box `B` causes the box to have wet paint, by the causal law

```
workOn(W, B) causes wetpaint(B)
  if workDone(B)=0.
```

We can describe change that does not directly involve an action of a robot. For instance, we formalize that a box with wet paint gets dry, by the causal law

```
caused -wetpaint(B) after wetpaint(B).
```

We describe preconditions of actions by causal laws as well. For instance, we describe that a robot `W` cannot work on a box `B` that still has wet paint, as follows:

`nonexecutable workOn(W,B) if wetpaint(B) .`

and that a worker robot W can work on a box B only if the worker has the appropriate end-effector for the next work stage, as follows:

```
nonexecutable workOn(W,B)
  if endEffector(W)=WS & workDone(B)\=WS-1 .
```

Similarly, we can express that a worker robot can work on a box if it is right next to the assembly line and it is aligned with the box, that the worker robots do not move vertically, that a worker robot swap its end-effector only if it is in the pit area, and other preconditions of the worker's actions. We can also formalize the preconditions of a carrier's actions: A carrier attaches to a worker only if it is right on top of it; a carrier can not push/pull a robot it is not attached to it.

Concurrent actions are allowed unless specified otherwise. For instance, we can express that a carrier cannot detach and pull at the same time by the causal law

```
nonexecutable detach(C) if pull(C) .
```

Similarly, we prevent the following concurrent actions: A worker cannot work on a box while the line is shifting; the pushing/pulling carrier robot and the pushed/pulled worker robot cannot be involved in any other action; and a moving robot cannot attach or detach or work on a box.

Once an action domain is described by a set of causal laws as shown above, we can present it to CCALC and ask CCALC "queries" about the existence of plans or the causes of observed failures. CCALC transforms the given domain description and the query into a set of propositional formulas, calls a SAT solver

(e.g., MANYSAT [43]) to find a model of these formulas, and extracts an answer to the given query from this model. For more detailed information about CCALC, we refer the reader to [4, 5].

4.4 Causality-Based Reasoning for Finding Optimal Decoupled Plans for Multiple Teams of Robots

One of the reasoning tasks that each team of robots should be able to perform is planning. A *plan* of length k is a sequence $\langle A_0, \dots, A_{k-1} \rangle$ of (concurrent) actions. Here each A_i is a set of elementary actions. If A_i is empty then it corresponds to “waiting”; otherwise, A_i characterizes the concurrent execution of the elementary actions it contains.

Planning Problem. Given an action domain description \mathcal{D} , an initial state s , a goal g , and a nonnegative integer k , find a plan of length at most k .

We can present a planning problem to CCALC by means of “queries”. For instance, the following query asks for a plan whose length is at most 100, for a team with one worker (`w1`) and one carrier (`c1`):

```
:- query
maxstep :: 100;
0: % INITIAL STATE
% no robot is attached to another
[/\C /\W | -attached(C,W)],
% no block has wetpaint
[/\B | -wetpaint(B)=0],
% worker is at (1,3)
```

```

xpos(w1)=1, ypos(w1)=4,
% carrier is at (1,1)
xpos(c1)=1, ypos(c1)=1,
% boxes are not yet processed
[/\B | linePos(B)=B+lineLength],
[/\B | workDone(B)=0];
maxstep: %GOAL
% boxes are painted
linePos(maxBox)=0,
[/\B | workDone(B)=3].

```

If we replace `maxstep :: 100` with `maxstep :: 18..100`, then the query asks for a shortest plan whose length is at least 18 and at most 100. In that case, with the domain description whose parts are briefly explained above in the previous section, CCALC finds a shortest plan of length 29 in 10 CPU seconds using the parallel SAT solver MANYSAT.

4.5 Diagnostic Reasoning in a Cognitive Factory

In the painting factory domain, a robot may get broken and thus may not succeed attaching to another robot or working on the boxes. We assume that a global sensor can detect if robots are attached/detached (but cannot detect which robot is broken), and the work stage. Once such discrepancies (“exceptions”) are noticed, the goal is to find their possible causes (i.e., which robots can be broken) so that an external agent (e.g., human or some other robot) can inspect the possibly broken robots and repair them. To take into account such discrepancies, we modify the domain description, and introduce an algorithm that computes minimal diagnoses

by means of asking CCALC prediction queries with temporal constraints over the modified domain description.

4.5.1 Modifying the Domain Description: Exceptions

We introduce a new fluent `broken(R)` to describe that a robot `R` may get broken at any step:

```
caused broken(R)
  if broken(R) after -broken(R).
```

and then we modify the causal laws describing the effect of `attach(C,W)` as follows:

```
attach(C,W) causes attached(C,W)
  if -broken(C) & -broken(W).
```

Similarly, we modify the causal laws describing the effects of `detach(C)` and `workOn(W,B)`.

4.5.2 Finding Minimal Diagnosis

Suppose that after a sequence $\langle A_0, \dots, A_n \rangle$ of (concurrent) actions is executed at a state S , a discrepancy is observed between the expected state (with respect to the domain description) and the observed state S' (obtained by sensors). In the painting domain, the causes of such discrepancies are due to broken robots; therefore, we can identify possible diagnoses by sets of possibly broken robots. We can find the set C of all minimal sets of at most k broken robots by Algorithm 4.

Algorithm 4 DIAGNOSE

Input: A state s , a sequence A_0, \dots, A_n of actions executed at s , an observed state s' , a nonnegative integer k

Output: Current state for a team, potentially updated by robot breakdown information

$R \leftarrow$ set of unbroken robots at state s ;

$C \leftarrow$ set of sets of at most k candidate robots to explain the anomalies, empty initially;

$m := n$; $holds := false$;

while $m \geq 0$ and $C = \emptyset$ **do**

 // Find minimal sets of at most k possibly broken robots, assuming that actions A_0, \dots, A_m are completely executed

$i := 1$;

while $i \leq k$ and $C = \emptyset$ **do**

for all set r of i robots in R **do**

$s_r \leftarrow$ modify s' by making robots in r broken;

$holds \leftarrow$ check if executing A_0, \dots, A_m (and possibly a subset of each action in A_{m+1}, \dots, A_n) at s results in s_r ;

if $holds$ **then**

$C := C \cup \{r\}$;

$i++$;

$m--$;

$x \leftarrow$ inspect the sets of robots in C to find the broken robots, and modify the state s' by specifying the broken robots;

return x ;

In the presence of qualification constraints, failure of some actions may prevent the execution of upcoming actions, leading to an unexpected state. For instance, if a carrier fails to attach to a worker (since the carrier is broken), the carrier cannot push the worker to the assembly line. Therefore, to be able to find a diagnosis for a discrepancy, the outer while loop of Algorithm 4 considers cases where only the first $m = n, n - 1, \dots, 1$ actions are completely executed. Then, the inner while loop of Algorithm 4 checks (through a query to CCALC) for every subset r of at most $i = 1, 2, \dots, k$ robots whether the execution of A_0, \dots, A_m of actions at S leads to S' where the robots in r are broken. If CCALC returns a positive answer to the query, then two important information becomes available: 1) an ex-

planation as to when the robots in r may have got broken during the execution of these actions, 2) which actions in A_{m+1}, \dots, A_n are executed.

For instance, consider the execution of a single concurrent action `attach(c1, w3)`, `workOn(w3, 2)` at a state where `c1` is not attached to `w3` and Box 2 is waxed (i.e., stage is 2). After this action is executed, we observe an unexpected state where `c1` is not attached to `w3` and Box 2 is stamped (i.e., stage is 3). To find a minimal diagnosis, Algorithm 4 checks whether a single robot might be broken ($n = 1$, $k = 1$) by a CCALC query. For the carrier robot `c1`, the query looks like:

```
maxstep :: 1;
0: -attached(c1, w3), workDone(2)=2,
    ...;
0: attach(c1, w3), workOn(w3, 2),
    -[\ / V_A | V_A\=attach(c1, w3) &
      V_A\=workOn(w3, 2)];
maxstep: % observed state
broken(c1), -attached(c1, w3),
workDone(2)=3 ... .
```

Here the two lines before the observed state description express that no other action `V_A` is executed in addition to `attach(c1, w3)` and `workOn(w3, 2)`. CCALC returns a positive answer with an explanation that the robot `c1` might be broken initially. Note that, to a similar query for the worker robot `w3`, CCALC returns a negative answer (otherwise work stage would not have changed); therefore, `w3` is not broken.

It is important to note here that, since broken robots are viewed and formulated in the domain description as “exceptions”, specifying these exceptions in queries

does not lead to inconsistencies, due to the nonmonotonic semantics of \mathcal{C}^+ .

Here is another example. Consider the execution of `attach(c1, w3)` and `lineShift` followed by `push(c1)` at a state where `c1` is not attached to `w3`. After the execution, it is observed that `c1` is still not attached to `w3`. Algorithm 4 finds a minimal diagnosis of $k = 1$ robot, by the query

```
maxstep :: 2;
0: -attached(c1, w3), ...;
0: attach(c1, w3), lineShift,
    -[\ / V_A | V_A \=attach(c1, w3) &
      V_A \=lineshift];
1: -[\ / V_A | V_A \= push(c1)];
maxstep: % current observed state
        broken(c1), -attached(c1, w3), ... .
```

for which CCALC returns a scenario that not only hints that `c1` is broken but also shows that `push(c1)` is not executed.

Note that in a workspace with x robots, DIAGNOSE checks $O(n \times x^k)$ such queries.

4.5.3 Repairing as part of Replanning

Once broken robots are identified by Algorithm 4 as part of our execution and monitoring algorithm, an external agent can repair them. For that, we modify the domain description further by introducing a new action `repair(R)` to repair broken robots `R`. The preconditions and effects of this action are described by causal laws, including the following


```
repair(R) causes -broken(R) .  
nonexecutable repair(R) if -broken(R) .
```

We also add causal laws to ensure that a robot does not perform any other actions while being repaired, or when it is broken.

With the modified domain description, the execution and monitoring algorithm can ask CCALC to compute a plan to reach the goal from the observed state S' where discrepancy is detected. Here, we also specify the broken robots as part of S' so that they get repaired as part of replanning.

4.6 Embedding Diagnostic Reasoning in an Execution Monitoring Framework

Let us demonstrate how our diagnosis algorithm and optimal decoupled planning algorithm can be embedded effectively in an execution and monitoring framework.

Algorithm 5 presents the overall execution and monitoring algorithm. First, for each team i , INIT tries to compute a plan $P_X[i]$ of length at most k_X ; if such a plan is computed (i.e., the team may be able to spare a robot) then the team is designated as a lender; otherwise, it is designated as a borrower. After that, Algorithm 5 tries to find an overall plan with minimum number of steps, calling FIND_OPTIMAL_PLAN; if such a plan is found, then it is executed by the teams. During the execution, if a discrepancy is detected between the observed state of the world and the expected state, then Algorithm 5 tries to diagnose the cause of the discrepancy in terms of minimum number of broken robots, calling DIAGNOSE.

If one of the robots is found as broken, then Algorithm 5 asks CCALC for a new plan that may include repair of the broken robot. Otherwise, the algorithm asks CCALC for a new plan from the observed state.

Algorithm 5 EXECUTE&MONITOR WITH DIAGNOSIS

Input: An action domain description \mathcal{D} , a nonnegative integer k , n planning problems $\mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_n$ (one for each team) with initial states s_1, s_2, \dots, s_n and goal states g_1, g_2, \dots, g_n , and a transportation delay t_d

Output: Achieve the goals of all teams in minimum time steps

// Let X be a tuple consisting of a plan length k_X ; and, for each team i , a plan $P_X[i]$ of length k_X , team role $role_X[i]$, and lower and upper bounds, $l_X[i]$ and $u_X[i]$, on the earliest/latest lend/borrow times.

$k_X := k$;

for all teams i **do**

$P_X[i], role_X[i], l_X[i], u_X[i] \leftarrow \text{INIT}(\mathcal{D}, k_X, \mathcal{P}_X[i]);$

while $k_X > 0$ **do**

$X \leftarrow \text{FIND_OPTIMAL_PLAN}(\mathcal{D}, \mathcal{P}_1, \mathcal{P}_2, \dots, \mathcal{P}_n, t_d, X);$

$replan := false$;

while $\neg replan \wedge k_X > 0$ **do**

$k_X := k_X - 1$;

for all teams i **do**

$A_i, c_i, e_i, o_i \leftarrow$ extract from $P_X[i]$ the actions to be executed, the current state, the expected state after A_i and the observed state after A_i ;

$l_X[i], u_X[i], role_X[i] \leftarrow$ update the bounds and roles;

$updated := false$;

if $o_i \neq e_i$ **then**

$updated := true$;

$s_i \leftarrow \text{DIAGNOSE}(\mathcal{D}, \mathcal{P}_i, c_i, o_i)$;

if new order of boxes **then**

$updated := true$;

$\mathcal{P}_i \leftarrow$ modify the planning problem;

if $updated$ **then**

$replan := true$;

$s_i \leftarrow$ obtain the current state

$P_X[i], role_X[i], l_X[i], u_X[i] \leftarrow \text{INIT}(\mathcal{D}, k_X, \mathcal{P}_i)$;

Consider, for instance, four teams: Team 1 has 1 worker and has to paint 4 boxes, Team 2 has 3 workers and has to paint 5 boxes, Team 3 has 2 workers and

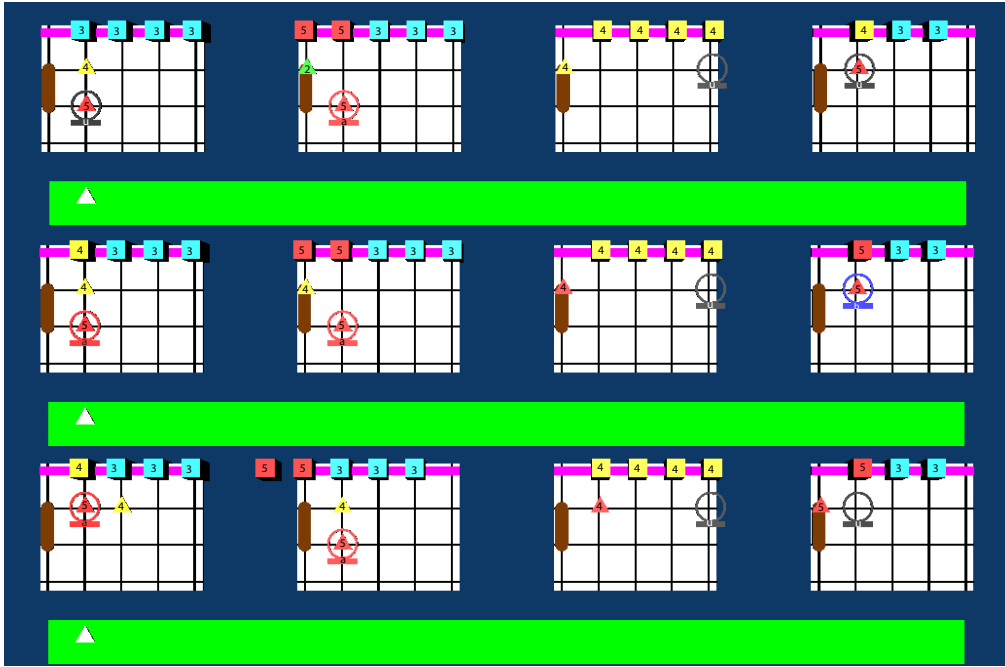


Figure 4.6.1: Snapshots for three consecutive states (Steps 16–18) of plan execution for workspaces. The pink assembly line is along the north wall; the pit stop area is marked by P ; the squares are the boxes (1: unprocessed, 2: painted and wet, 3: painted and dry, 4: waxed, 5: stamped); the large robot is the carrier, it can be attached (a) or unattached (u); the triangles are the workers, which can have different end-effectors (2: painter, 4: waxer, 5: stamper); the benched robots are either white (ready to be given to another team) or black (still in the transportation delay).

has to paint 4 boxes, and Team 4 has 1 worker and has to paint 3 boxes. For $k_X = k = 1$, no plan is found for any of the teams (by INIT). Then FIND_OPTIMAL_PLAN finds an optimal decoupled plan of length 27. According to this plan, Team 1 completes its tasks in 27 steps with a help received at Step 13. While this plan of length 27 is being executed, at Step 16, the carrier of Team 4 (c_1) tries to attach to the worker w_1 , but fails. This discrepancy is detected at Step 17 (i.e., $attached(c_1, w_1)$ is not observed). DIAGNOSE (with $m = 26$ and $k = 1$) infers that c_1 is broken (as explained in the first example of Section 4.5.2), and returns the observed state s_2 by adding the information about the broken robot.

Since a discrepancy is detected during the execution Team 2’s plan, the current states of other teams are also obtained for the purpose of decoupled replanning. During replanning, no plan is found for Team 2 to complete the remaining job in 10 steps, causing it to become a borrowing team. The other 3 teams on the other hand are lenders: Teams 2 and 3 were lenders in the original plan, and they continue to be lenders; Team 1 was a borrower in the original plan, however after borrowing a robot at Step 13, it became self-sufficient and is also treated as a lender for the replan. There is a free robot sent to the bench by Team 3 before the failure. Under these conditions, an overall decoupled plan of length 10 is found. According to this plan, Team 4 repairs the carrier immediately (Step 17) and receives the robot from the bench next (Step 18). This plan is executed without any unexpected events; therefore, all tasks of the teams are completed at Step 27. Table 4.6.1 and Figure 4.6.1 show some of these actions and their executions. A video of this sample run is provided also at <http://krr.sabanciuniv.edu/cogrobo/demos/cogfactory>.

Examples in which the new orders arrive for teams are handled via replanning in Algorithm 5. Such examples are not included in text due to space restrictions.

Table 4.6.1: List of actions for all teams

Table 4.6.1

	Team 1	Team 2	Team 3	Team 4
0	move(w1,right) lineShift	move(w1,right) move(w3,right) move(c1,right) lineShift	move(w1,right) move(w2,right) move(c1,right) lineShift	move(w1,right) lineShift
1	move(w1,right) lineShift	move(w1,right) move(w2,right) lineShift attach(c1,w3)	move(w1,right) move(w2,right) move(c1,right)	lineShift
2	lineShift	move(w2,right) push(c1) lineShift	move(w1,right) move(c1,right) lineShift	move(w1,right) lineShift

Continued on Next Page...

Table 4.6.1 – Continued

	Team 1	Team 2	Team 3	Team 4
3	workOn(w1,1)	push(c1) workOn(w1,1)	workOn(w1,1)	move(w1,right) lineShift
4	move(w1,right) lineShift	lineShift detach(c1)	move(w2,right) move(c1,right) lineShift	workOn(w1,3)
5	workOn(w1,3)	move(w2,left) move(c1,down) workOn(w1,2)	move(w1,right) move(w2,right) move(c1,up) lineShift	move(w1,left) move(c1,up)
6	move(w1,right)	move(w1,right) workOn(w3,1) attach(c1,w2)	workOn(w1,4) attach(c1,w2)	workOn(w1,2)
7		move(w3,right) push(c1) workOn(w1,3)	move(w1,left) push(c1)	move(w1,left)
8	move(c1,right) workOn(w1,4)	move(w1,right) workOn(w2,1) workOn(w3,2)	workOn(w1,3) workOn(w2,4) detach(c1)	workOn(w1,1)
9	move(w1,left)	move(w3,left) pull(c1) workOn(w1,4)	move(w1,left) move(w2,left)	move(w1,left) move(c1,right)
10	move(w1,left)	move(w3,left) push(c1) lineShift	workOn(w1,2) workOn(w2,3)	swapEndEffector(w1,2)
11	workOn(w1,2)	workOn(w1,5) workOn(w2,2) giveRobot(w3)	move(w1,left) move(w2,left)	move(w1,right)
12	move(w1,left)	move(w1,left)	move(w1,left) workOn(w2,2)	workOn(w1,1)
13	move(w1,left) takeRobot(w2,1,2)	move(w1,left) pull(c1)	move(w2,left) giveRobot(w1)	move(w1,left)
14	move(c1,up) swapEndEffector(w1,2)	move(w1,left)	workOn(w2,1)	swapEndEffector(w1,3)
15	move(w1,right) move(w2,right)	move(w1,left)	move(w2,left)	move(w1,right) move(c1,up)
16	workOn(w1,1) attach(c1,w2)	swapEndEffector(w1,2)	swapEndEffector(w2,3)	workOn(w1,1) attach(c1,w1)
17	move(w1,right) push(c1)	move(w1,right) lineShift	move(w2,right)	move(w1,left) repair(c1)
18	workOn(w1,2) workOn(w2,1)	workOn(w1,3)	workOn(w2,1)	move(c1,down) lineShift takeRobot(w2,1,2) swapEndEffector(w1,2)
19	lineShift	move(w1,right) push(c1)	lineShift	move(w1,right) move(w2,right)
20	workOn(w1,3) workOn(w2,2)	workOn(w1,4) workOn(w2,3)	workOn(w2,2)	workOn(w1,2) attach(c1,w2)
21	lineShift	lineShift	lineShift	move(w1,right)

Continued on Next Page...

Table 4.6.1 – Continued

	Team 1	Team 2	Team 3	Team 4
				push(c1)
22	workOn(w1,4) workOn(w2,3)	workOn(w1,5) workOn(w2,4)	workOn(w2,3)	workOn(w1,3) workOn(w2,2)
23	lineShift	lineShift	lineShift	lineShift
24	workOn(w2,4)	workOn(w2,5)	workOn(w2,4)	workOn(w2,3) detach(c1)
25	lineShift detach(c1)	lineShift detach(c1)	lineShift	lineShift
26	lineShift	lineShift	lineShift	lineShift

4.6.1 Dynamic Simulation of the Cognitive Painting Factory

We have implemented a simulation of the cognitive painting factory domain in the dynamic simulation environment Gazebo. The robots and decoupled planning algorithms are implemented in Robot Operating System (ROS), utilizing custom codes written in C++ and Python codes. The simulated cognitive factory domain is presented in Figure 4.6.2.

For the dynamic simulation, we have replaced all the robots with KUKA youBot holonomic mobile manipulator depicted in Figure 4.6.3. Since all robots can move along horizontal and vertical direction, no carrier robots are employed. However, to preserve the challenges of the previously studied cognitive painting factory domain, in each workspace we have assigned a different role to one of the mobile platforms. In particular, one robot in each team is designated as a charging station and all other robots (workers) are required to meet and dock with the charger robot as their batteries become drained. Hence, in the new scenario the carrier robot is replaced with a charger robot, while the attach / detach actions are replaced with dock / undock actions. An example execution of a plan in the dynamic simulator is presented in Fig. 4.6.4

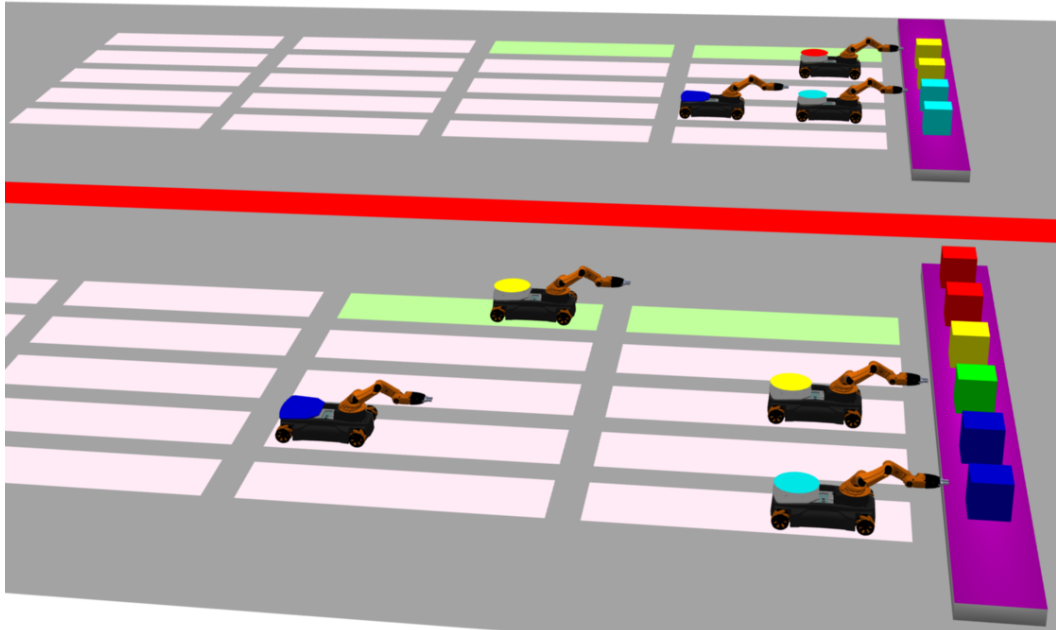


Figure 4.6.2: Cognitive painting factory with multiple teams of KUKA youBots

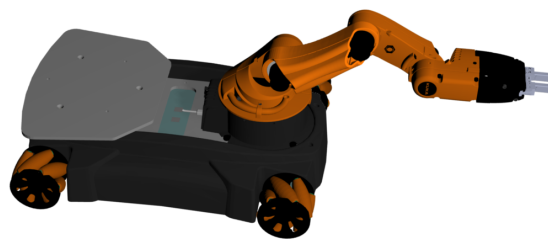


Figure 4.6.3: KUKA youBot holonomic mobile manipulator

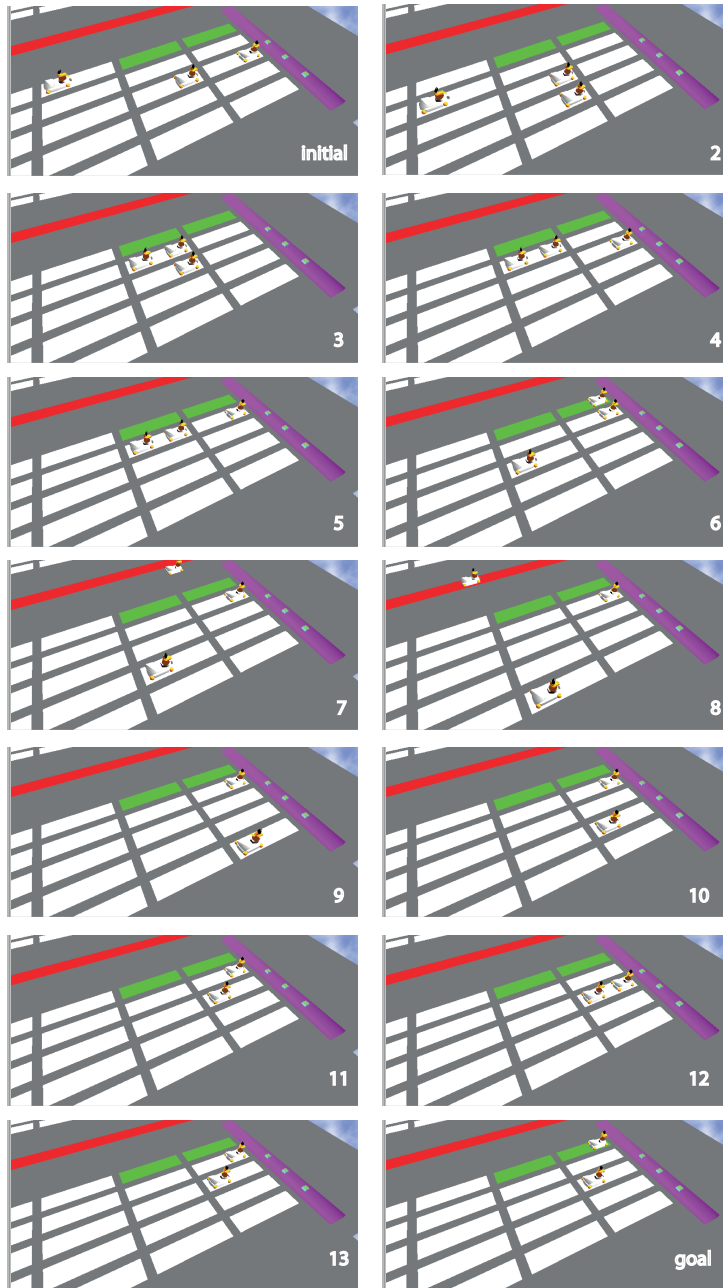


Figure 4.6.4: Execution of an example plan for one team in dynamic simulation Gazebo

Chapter 5

Conclusions

We introduced a new approach for execution and monitoring of a manipulation planning problem and have illustrated the usefulness of this approach with a physical implementation of a problem that involves two robots working for a common goal. We have considered three scenarios of plan execution:

No surprises. The robots have a complete knowledge of the environment and nothing unexpected occurs in the domain while a plan is being executed.

Collisions with unknown obstacles. The robots do not have a complete knowledge of the obstacles in the environment and thus collide with such an unknown while executing a plan.

Human interventions. While the robots are executing a plan, a human intervenes and change the locations of some payloads.

In each scenario, we have shown how the robots find and execute a plan using our plan execution and monitoring algorithm.

Then we expand this approach for multiple teams of robots and introduced two algorithms utilizing the formalisms, methods and tools of causal reasoning to endow multiple teams of self-reconfigurable robots with high-level reasoning capabilities. We introduce a system which is capable of decoupled planning and execution monitoring of multiple teams of robots. In addition this system can diagnose the errors which can not be detected by sensors, and computes minimal diagnoses for failures in terms of number of broken robots.

We have shown the applicability and usefulness of these algorithms, embedded in a generic execution and monitoring framework, in a cognitive painting factory scenario, which provides a good case study towards future intelligent factories.

As a part of future work, we plan to extend our planning and monitoring framework to handle probabilistic data obtained by sensor fusion. We may also develop the domain description to handle probabilistic occasions to make the framework more useful for unstructured environments.

Appendix A

The Task Plan for Query 2 of the Manipulation the Planning Problem

```
0: move(r1, up, steps=1)
    move(r1, right, steps=1)
    move(r2, down, steps=2)
    move(r2, right, steps=1)
1: move(r1, up, steps=3)
    move(r1, right, steps=1)
    move(r2, down, steps=2)
    move(r2, left, steps=2)
2: pick(r1, pickpoint=4)
    pick(r2, pickpoint=3)
3: move(r1, down, steps=1)
    move(r1, right, steps=3)
    move(r2, up, steps=2)
```

```
    move(r2, right, steps=2)
4: move(r1, right, steps=4)
    move(r2, up, steps=2)
5: drop(r1)
    drop(r2)
6: move(r1, down, steps=1)
    move(r1, left, steps=1)
    move(r2, down, steps=1)
    move(r2, left, steps=1)
7: pick(r1, pickpoint=5)
    pick(r2, pickpoint=6)
8: move(r1, down, steps=1)
    move(r1, left, steps=3)
    move(r2, down, steps=3)
    move(r2, right, steps=1)
9: move(r1, up, steps=3)
    move(r1, left, steps=2)
    move(r2, left, steps=1)
10: drop(r1)
    drop(r2)
11: move(r1, down, steps=3)
    move(r1, left, steps=1)
    move(r2, down, steps=3)
    move(r2, left, steps=1)
12: pick(r1, pickpoint=1)
```

```
    pick(r2, pickpoint=2)
13: move(r1, down, steps=2)
    move(r1, right, steps=3)
    move(r2, up, steps=3)
    move(r2, left, steps=2)
14: move(r1, up, steps=2)
    move(r1, left, steps=3)
    move(r2, up, steps=2)
    move(r2, left, steps=3)
15: drop(r1) drop(r2)
16: move(r1, up, steps=3)
    move(r1, right, steps=2)
    move(r2, up, steps=1)
    move(r2, right, steps=1)
17: move(r1, right, steps=4)
    move(r2, down, steps=3)
18: pick(r1, pickpoint=6)
    pick(r2, pickpoint=5)
19: move(r1, down, steps=1)
    move(r1, right, steps=1)
    move(r2, up, steps=2)
    move(r2, right, steps=2)
20: move(r1, down, steps=1)
    move(r1, left, steps=1)
    move(r2, up, steps=1)
```

```
    move(r2, right, steps=3)
21: drop(r1)
    drop(r2)
22: move(r1, up, steps=1)
    move(r1, right, steps=1)
    move(r2, up, steps=1)
    move(r2, right, steps=1)
23: pick(r1, pickpoint=4)
    pick(r2, pickpoint=3)
24: move(r1, left, steps=3)
    move(r2, left, steps=3)
25: move(r1, down, steps=2)
    move(r1, left, steps=1)
    move(r2, down, steps=2)
    move(r2, left, steps=1)
26: drop(r1)
    drop(r2)
```

Bibliography

- [1] M.F. Zaeh, M. Beetz, K. Shea, G. Reinhart, K. Bender, C. Lau, M. Ostgathe, W. Vogl, M. Wiesbeck, M. Engelhard, C. Ertelt, T. Rühr, M. Friedrich, and S. Herle. The cognitive factory. In *Changeable and Reconf. Manufacturing Systems*, pages 355–371. 2009.
- [2] Michael Beetz, Martin Buss, and Dirk Wollherr. CTS - What is the role of artificial intelligence? In *Proc. of KI*, pages 19–42, 2007.
- [3] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic, Dordrecht, 1991.
- [4] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *AIJ*, 153:49–104, 2004.
- [5] Norman McCain and Hudson Turner. Causal theories of action and change. In *Proc. of AAAI/IAAI*, pages 460–465, 1997.
- [6] Ronald J. Brachman. Systems that know what they’re doing. *IEEE Intelligent Systems*, 17(6):67–71, 2002.

- [7] C. Ertelt, K. Shea, D. Pangercic, T. Ruhr, and M. Beetz. Integration of perception, global planning and local planning in the manufacturing domain. In *Proc. of ETFA*, pages 1–8, 2009.
- [8] Kristina Shea, Christoph Ertelt, Thomas Gmeiner, and Farhad Ameri. Design-to-fabrication automation for the cognitive machine shop. *Advanced Engineering Informatics*, 24(3):251–268, 2010. The Cognitive Factory.
- [9] Kristina Shea, editor. *Special Issue in Cognitive Robotics*, volume 24. Advanced Engineering Informatics, Elsevier, 2010.
- [10] A. Bannat, T. Bautze, M. Beetz, J. Blume, K. Diepold, C. Ertelt, F. Geiger, T. Gmeiner, T. Gyger, A. Knoll, C. Lau, C. Lenz, M. Ostgathe, G. Reinhardt, W. Roesel, T. Ruehr, A. Schuboe, K. Shea, I. Stork genannt Wersborg, S. Stork, W. Tekouo, F. Wallhoff, M. Wiesbeck, and M.F. Zaeh. Artificial cognition in production systems. *IEEE Transactions on Automation Science and Engineering*, 8(1):148–174, 2011.
- [11] C. Lenz, S. Nair, M. Rickert, A. Knoll, W. Rosel, J. Gast, A. Bannat, and F. Wallhoff. Joint-action for humans and industrial robots for assembly tasks. In *Proc. of ROMAN*, pages 130–135, 2008.
- [12] Paul Maier, Martin Sachenbacher, Thomas Rühr, and Lukas Kuhn. Automated plan assessment in cognitive manufacturing. *Advanced Engineering Informatics*, 24(3):308–319, 2010. The Cognitive Factory.
- [13] Richard Fikes. Monitored execution of robot plans produced by *trips*. In *Proc. of IFIP Congress*, pages 189–194, 1971.

- [14] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In *Proc. of AAAI*, pages 677–682, 1987.
- [15] R. James Firby. An investigation into reactive planning in complex domains. In *Proc. of AAAI*, pages 202–206, 1987.
- [16] Brian C. Williams and P. Pandurang Nayak. A model-based approach to reactive self-configuring systems. In *Proc. of AAAI/IAAI*, pages 971–978, 1996.
- [17] J.L. Fernandez and R.G. Simmons. Robust execution monitoring for navigation plans. In *Proc. of IROS*, pages 551–557, 1998.
- [18] R. Simmons, J.L. Fernandez, R. Goodwin, S. Koenig, and J. O’Sullivan. Lessons learned from xavier. *Robotics Automation Magazine, IEEE*, 7(2):33–39, 2000.
- [19] Manuela M. Veloso, Martha E. Pollack, and Michael T. Cox. Rationale-based monitoring for planning in dynamic environments. In *Proc. of AIPS*, pages 171–180, 1998.
- [20] Ola Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2):73–88, 2005.
- [21] Patrick Doherty, Jonas Kvarnström, and Fredrik Heintz. A temporal logic-based planning and execution monitoring framework for unmanned aircraft systems. *Autonomous Agents and Multi-Agent Systems*, 19(3):332–377, 2009.

- [22] Thomas Eiter, Esra Erdem, and Wolfgang Faber. Undoing the effects of action sequences. *Journal Applied Logic*, 6(3):380–415, 2008.
- [23] Thomas Eiter, Esra Erdem, Wolfgang Faber, and Ján Senko. A logic-based approach to finding explanations for discrepancies in optimistic plan execution. *Fundamenta Informaticae*, 79(1–2):25–69, 2007.
- [24] Matthias Fichtner, Axel Großmann, and Michael Thielscher. Intelligent execution monitoring in dynamic environments. *Fundamenta Informaticae*, 57(2–4):371–392, 2003.
- [25] Mikhail Soutchanski. High-level robot programming and program execution. In *Proc. of ICAPS Workshop on Plan Execution*, 2003.
- [26] Khaled Ben Lamine and Froduald Kabanza. Reasoning about robot actions: A model checking approach. In *Advances in Plan-Based Control of Robotic Agents*, pages 123–139, 2001.
- [27] Giuseppe De Giacomo, Raymond Reiter, and Mikhail Soutchanski. Execution monitoring of high-level robot programs. In *Proc. of KR*, pages 453–465, 1998.
- [28] Raymond Reiter. *Knowledge in action: Logical Foundations for specifying and implementing dynamical systems*. MIT Press, 2001.
- [29] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.

- [30] Michael Thielscher. The concurrent, continuous Fluent Calculus. *Studia Logica*, 67(3):315–331, 2001.
- [31] Michael Thielscher. FLUX: A logic programming method for reasoning agents. *Theory and Practice of Logic Programming*, 5(4-5):533–565, 2005.
- [32] Jose A. Ambros-Ingerson and Sam Steel. Integrating planning, execution and monitoring. In *Proc. of AAAI*, pages 83–88, 1988.
- [33] Karen Zita Haigh and Manuela M. Veloso. Planning, execution and learning in a robotic agent. In *Proc. of AIPS*, pages 120–127, 1998.
- [34] Karen L. Myers. CPEF: A continuous planning and execution framework. *AI Magazine*, 20(4):63–69, 1999.
- [35] Solange Lemai and Félix Ingrand. Interleaving temporal planning and execution in robotics domains. In *Proc. of AAAI*, pages 617–622, 2004.
- [36] Michael Beetz and Drew McDermott. Improving robot plans during their execution. In *Proc. of AIPS*, pages 3–12, 1994.
- [37] T. Ruhr, D. Pangercic, and M. Beetz. Structured reactive controllers and transformational planning for manufacturing. In *Proc. of ETFA*, pages 97–104, 2008.
- [38] M. Rungger, H. Ding, T. Paschedag, and O. Stursberg. Hierarchical hybrid modeling and control of cognitive manufacturing systems. In *International Workshop on Cognition for Technical Systems*, 2008.
- [39] Fangzhen Lin. Embracing causality in specifying the indirect effects of actions. In *Proc. of IJCAI*, pages 1985–1991, 1995.

- [40] Michael Gelfond and Vladimir Lifschitz. Action languages. *Electronic Transactions on Artificial Intelligence*, 2:193–210, 1998.
- [41] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proc. of ECAI*, pages 359–363, 1992.
- [42] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Proc. of SAT*, pages 502–518, 2003.
- [43] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Control-based clause sharing in parallel sat solving. In *Proc. of IJCAI*, pages 499–504, 2009.
- [44] Hector Levesque and Gerhard Lakemeyer. Cognitive robotics. In *Handbook of Knowledge Representation*. Elsevier, 2007.
- [45] Vladimir Lifschitz. What is answer set programming? In *Proc. of AAAI*, 2008.
- [46] Vladimir Lifschitz and Hudson Turner. Representing transition systems by logic programs. In *Logic Programming and Non-monotonic Reasoning: Proc. Fifth Int’l Conf. (Lecture Notes in Artificial Intelligence 1730)*, pages 92–106, 1999.
- [47] Vladimir Lifschitz. Action languages, answer sets and planning. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 357–373. Springer Verlag, 1999.
- [48] Steven M. Lavelle. Rapidly-exploring random trees: A new tool for path planning. Technical report, 1998.

- [49] Ozan Caldiran, Kadir Haspalamutgil, Abdullah Ok, Can Palaz, Esra Erdem, and Volkan Patoglu. Bridging the gap between high-level reasoning and low-level control. In *Proc. of LPNMR*, 2009.
- [50] Tansel Uras. Applications of ai planning in genome rearrangement and in multi-robot systems, 2011.