Coordination for Agents with Freedom of Choice

Henk J. Pijper

Coordination for Agents with Freedom of Choice

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Henk J. Pijper
born in Dordrecht, The Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, The Netherlands
http://www.ewi.tudelft.nl/

Coordination for Agents with Freedom of Choice

Author: Henk J. Pijper
Student id: 1174738
Email: H.J.Pijper@student.tudelft.nl

Abstract

Autonomous, self-interested agents have to construct plans for their activities. Together, these plans form a joint plan in the multi-agent system the agents are part of. Merging the plans of these agents is not guaranteed to be successful as their plans might conflict.

Hence, we require some form of coordination that enables agents to plan for their tasks autonomously. We will present a *pre-planning coordination approach*. Our research focusses on coordination of tasks with preconditions and effects, which we define as *instantiation-coordination*. It is based on an existing approach, called *plan-coordination*. Verifying instantiation-coordination is coNP-complete. Ensuring instantiation-coordination is even harder: Σ_2^p-complete.

We apply our approach to a classical planning domain: Logistics. Based on these experiments, we evaluate the cost involved with our coordination approach for autonomous agents.

Thesis Committee:

Chair:	prof. dr. C. Witteveen, Faculty EEMCS, TU Delft
Committee Member:	dr. T.B. Klos, Faculty EEMCS, TU Delft
Committee Member:	dr. P.G. Kluit, Faculty EEMCS, TU Delft
University Supervisor:	ir. J.R. Steenhuisen, Faculty EEMCS, TU Delft
University Supervisor:	C. Yadati Narasimha MSc., Faculty EEMCS, TU Delft

Preface

More than once, I was asked when my thesis work would be finished. Well, to all of them: here it is! After several months of hard work, I am pleased with the result. Coordination for agents with freedom of choice; at the start of my thesis work not the title I imagined for my report. However, I think it covers an interesting extension to impressive previous work on coordination.

During my thesis work, I was supported by many others. Without their help, I doubt whether my work would have reached this level. First of all, I would like to thank my daily supervisors Renze Steenhuisen and Chetan Yadati Narasimha. With Renze, I had very valuable discussions on ideas and the theory underlying my thesis work. Chetan often provided me with new theoretical insights and could often provide a clear context for the theory.

I would like to thank Cees Witteveen for being my supervisor. During our meetings, he was able to swiftly come up with new ideas and to assess mine. His remarks allowed me to considerably enhance this report. Peter Kluit and Thomas Klos, thanks in advance for reading and assessing my work. As I spend quiet some time on the 7^{th} floor, I would like to thank all members of the Algorithmics group for our lunches and discussions at the coffee machine.

Finally, I would like to thank the people closest to me; the ones more distant from my thesis work. Foremost, I am grateful for the thorough support of my girlfriend. It must have been dozens of hours that I troubled her with my issues. I would like to thank my brother for providing me with his opinion. Last, but not least, I would like to thank my parents for their continuous support during the period of my thesis and the many years that led me to this point.

This report's title and preface alone might give you a little sense on the contents of this report, so I would encourage you to read on!

Henk J. Pijper
Delft, The Netherlands
January, 2010

Contents

List of Figures

Nomenclature

$\mathcal{T}^a$	Abstract complex task
α	Action
A	Set of actions
a	Agent
$\mathcal{A}$	Set of agents
$\mathcal{C}_e$	Set of effects for a set of tasks
$\mathcal{C}_p$	Set of preconditions for a set of tasks
$\mathcal{T}$	Complex task
$\precsim$	Condition dependency
$\precsim_{\text{inter}}$	Inter-agent condition dependency
$\precsim_{\text{intra}}$	Intra-agent condition dependency
c	Condition
C	Set of conditions
Δ	Coordination set
c_e	Effect
C_e	Set of effects
ϕ	Boolean formula
Π	Joint plan
π	Plan

c_p	Precondition
C_p	Set of preconditions
t	Task
f_{ta}	Task allocation function
$\mathcal{P}$	Environment of tasks
I	Set of task instantiations
$\mathbf{T}$	Task partitioning
$\prec$	Precedence relation for tasks
$\prec_{\text{inter}}$	Inter-agent precedence relation for tasks
$\prec_{\text{intra}}$	Intra-agent precedence relation for tasks
T	Set of tasks

Chapter 1

Introduction

A soccer match, between robots... It is the main focus of the yearly international research and education initiative: *RoboCup*.[1] In various leagues, teams from all over the world compete with their robotic teams. Without intervention of their creators, the robots play soccer matches on a small indoor field.

Except for the smallest, robots have to decide what to do themselves. They should *plan* which actions they will perform in the near future. They are *autonomous* in deciding what actions to perform and when to execute them. On the other hand, they should *cooperate* to win a match. Winning a match is equivalent to achieving two simple *goals*: to score goals and to prevent goals against. Robots should construct plans such that these two goals are achieved. Together, their plans form a *joint plan* for the team. Just like in 'ordinary' soccer, it is the *team* that wins a match, not the individual. Hence, *planning* plays a key role in these soccer robots. The better robots *coordinate* for their plans, the better the overall joint plan will be.

Robots in various leagues have to rely on their sensors to explore their environment. Each robot has to identify, among other facts, where the ball is, where teammates and opponents are, and its location on the field. Suppose two robots both process the information they get from their sensors. One of them constructs and executes a plan to defend, the other to attack. This would lead to a low quality joint plan. It hurts the effectiveness of the robots as a team. By *communicating* during the match, they can coordinate decisions such as to defend or attack as a team.

Not every decision is to be communicated during the game. It would be far too time consuming to coordinate each and every act. Positions as attacker, defender, or keeper can be easily assigned to robots *before* the match starts. Similarly, assuming all robots are identical, it saves communication effort to decide upfront which robot to take the corner kicks. These coordination measures are defined *prior to* the match and hence before the robots start planning.

[1] The official site of the RoboCup: `http://www.robocup.org/`.

A soccer robot in the previous example can be seen as an *agent*. An agent can be almost any concept or thing that has some form of *autonomy*. Together, the soccer robots constitute a *multi-agent system*. In a multi-agent system, multiple agents have to cooperate and coordinate their activities to achieve a goal. These systems are not only used for modelling robots that play soccer. Examples of such systems are disaster response teams [38], organisations in supply-chains [11], and examples from nature, like ant colonies [27].

Planning plays a key role in multi-agent systems. It involves reasoning about the effects of actions one will perform, before actually executing them. We will study multi-agent systems in which a problem is decomposed in subproblems. Subproblems are assigned to agents, which have to construct plans for them. After planning, we merge agents' plans into a *joint plan*. While each agents constructed its plan in isolation, their plans might *conflict* when merging them.

We would like to prevent these conflicts. We will be particularly interested in *coordination* of agents' planning processes, *prior to planning*. This type of coordination *prevents* conflicts when merging plans.

1.1 Coordinating Agents

Let's revisit our soccer robots example. Assume, for simplicity, we have two robots in our team; robots r_1 and r_2. Suppose r_1 is assigned the *task* t_1 of taking a corner kick. To *achieve* this task, r_1 has to move to the corner flag, position itself, and shoot the ball. In this position, it is impossible for r_1 to score a goal. Hence, another task is required to score. Robot r_2 is assigned the task t_2 to shoot on target when obtaining the ball.

Robot r_1 might pass the ball close to the goal, or further away. Task t_1 has a so-called *disjunctive effect*; either one of the *effects* of it will hold. Therefore, robot r_2 should position itself either close to the goal, or further from the goal. Its position *depends* on the effect of task t_1. We define the position of r_2 to be the *precondition* of t_2. We will call this relation a *condition dependency* between an effect and a precondition.

When robot r_1 kicks the ball, it is probably too late for robot r_2 to position itself. Hence, robots r_1 and r_2 should *coordinate* their planning activities to agree on a single position. Whether it is a successful goal attempt or not, the robots will probably loose the ball as a team. There default action is therefore to return to their own half and defend. We will call these tasks t_3 and t_4 for respectively robot r_1 and r_2. While there is no clear dependency between the effects of tasks t_1, t_2 and the preconditions of t_3, t_4 we can simply *order* them by a *precedence relation*.

1.2 Our Research

We highlighted various terms in the soccer robot example. In our research, we focus on achieving coordination for the planning activities of autonomous agents. Pre-

vious work, by Valk [35], addressed coordination for *tasks* and *precedences* between tasks. It provides a framework for analysing coordination as well.

In our work, we will extend: (i) the existing framework for analysing coordination and (ii) the existing coordination mechanism for ensuring coordination. We study *pre-planning* coordination. By ensuring pre-planning coordination, each plan constructed by an agent merges without conflicts into a feasible joint plan. We will analyse the impact of *enriching* the task coordination framework on coordination of autonomous planning agents. Our enrichments are twofold: (i) we extend tasks to have preconditions and effects and (ii) we add a dependency relation between effects and preconditions. The extended tasks are *abstract tasks*.

We formulate the central research question for this thesis as:

> How to ensure coordination of autonomous, self-interested planning agents for dependent abstract tasks?

To answer this research question, we will extend the coordination framework and mechanism as presented in Valk [35].[2] To distinguish between our work and previous work, we will use the term *plan-coordination* for the previous work. Our work extends coordination with the notion of *task instantiation*; we therefore term it *instantiated plan-coordination*.

1.2.1 Contributions

We identify three contributions in our work with respect to the research question.

- We formally *extend* the *task coordination framework*, which has been used in previous work for the analysis of plan-coordination. We will use the enriched framework to analyse instantiated plan-coordination.
- We state the problem of *instantiated plan-coordination* in terms of the enriched task coordination framework. We show that with our enrichments the problems of verifying and ensuring coordination remain in the *same complexity class*. On the other hand, we show that our enrichments in isolation are of the same complexity classes as well. Hence, the enrichments are not trivial extensions.
- We analyse the *applicability* of our coordination approach by applying it to the Logistics problem.
  - We extend the number of Logistics instances we can coordinate.
  - We run various planners on instances for this problem to identify the cost of autonomy for agents.

[2] Although interesting, we will not consider decomposition [4] and temporal extensions [32] to this approach.

1.3 Outline

Our contributions are in Chapters 3 and 4. Some preliminary discussion is required before introducing instantiated plan-coordination. In Chapter 2, we will therefore first present the field of multi-agent coordination and the existing task coordination framework.

Subsequently, we will introduce instantiated plan-coordination in Chapter 3. There, we discuss the extensions to the task coordination framework and the complexity results we have obtained.

In Chapter 4, we discuss instantiated plan-coordination for the Logistics problem in a multi-agent setting. We ran domain-independent planners on these instances in a central and multi-agent setting. Both settings are compared in terms of plan length and run-time to identify the cost of autonomy for the agents in this specific Logistics problem.

Finally, we will finalise our work in Chapter 5. It concludes this report and features an outlook for potential future work.

Chapter 2

Planning and Coordination Preliminaries

Our discussion on instantiation-coordination requires several concepts to be clear. In this chapter, we introduce these concepts. Additionally, we will position our research in the field and identify research that is related to our work.

First, in Section 2.1, we will sketch the field of multi-agent planning and coordination. Next, we will discuss the main foundation for our work, the *task coordination framework* in Section 2.2. In Section 2.3, we will conclude this chapter with the incitement for our work.

2.1 Multi-agent Planning

We have a set of *agents*, grouped into a single *system*. An agent can be virtually anything; from a human being to an automated software agent.[1] What matters is that these agents form a system that is able to perform collective tasks. To perform these tasks, agents have to reason about the activities of other agents to coordinate for their own activity [24].

Here, we will study coordination for *planning by* and *for* a multi-agent system.[2] During planning, agents reason about the activities they want to *execute*. They construct plans that form a blue-print for their activities. These activities result from the tasks assigned to agents during an *allocation* process. Finally, we identify the *scheduling* phase in which resources are allocated to agents and the activities scheduled in time. In Figure 2.1, the overall process is depicted. This process need not be sequential. Often, phases are repeated with knowledge inferred from subsequent phases.

Agents each have their own planning, scheduling, and execution phases. If their activities would not interfere, it would be senseless to study them in a multi-agent setting. We therefore *do* assume their activities to interfere and require *co-*

[1] Our discussion will not limit itself to a certain type of agent.

[2] Plans can be constructed *by* agents and *for* agents. Here, we have both.

Figure 2.1: Activity-related process.

ordination to deal with this interference. By coordinating the activities of agents, possible conflicts can be avoided, resolved, or repaired.

Focussing on planning, we define conflicts between agents as conflicting plans. Plans conflict if they cannot be *merged* into a feasible joint plan. We identify three moments at which coordination can be achieved to deal with conflicting plans [7]:

Before planning When achieving coordination prior to planning, agents do *not* have to reason about other agents' planning processes. By constructing plans that adhere to the coordination constraints, coordination is *ensured* and conflicts are *avoided*. An active topic in coordination research is that of *social laws* [30]. Social laws are rules for all agents in the multi-agent system. When agents obey these rules, conflicts will be avoided. Social laws restrict the actions of agents, depending on the state of the world. An example of social laws in practise are the rules defined for road traffic.

Another approach is the pre-planning coordination approach introduced by Valk [35]. Coordination is defined at the level of tasks and precedences between them. By *decoupling* [31] for plans cyclic dependencies between agents are avoided.[3] Agents are coordinated by adding dependencies between tasks. It differs from social laws in that the set of dependencies added to the tasks is defined on a per-agent basis.

During planning Ensuring coordination during planning requires agents to *communicate*. By communicating, agents can harmonise their activities. An influential approach is Generalized Partial Global Planning (GPGP) [23]. Agents in this approach have a limited view of the multi-agent system they operate in. By communicating about task relations and partial-beliefs of agents, coordination is achieved.

After planning Finally, we can coordinate the activities of agents after planning. First, each agent constructs a plan for its local activities. After planning, these plans are merged. Conflicts that arise during merging might either be resolved by repairing the joint plan [6] or by re-planning [26].

In the work of Valk [35], the relation between coordination before and after planning is acknowledged. A pre-planning coordination approach might coordinate up to the extent that coordination after planning is trivial. We will focus our attention on coordination *before* planning.

[3] Decoupling is a process in which coordination is achieved such that agents do not have to reason about interactions between their subproblems.

2.1.1 Agents and Assumptions

Planning and coordination can be applied in various multi-agent systems. Agents play a key role in such systems. Because we study coordination for these agents, we have to identify some of their characteristics.

Autonomous Agents require at least some *autonomy* in constructing their own plans. Autonomy is the ability of agents to decide on their own activities. Autonomy is an ambiguous term in literature on multi-agent systems [2]. For our work, we will assume that agents do not or cannot communicate at any moment of the planning phase. Hence, we will have to coordinate for the agents' activities.

Self-interested We distinguish between *cooperative* and *self-interested* agents. Cooperative agents have the overall goal as their highest goal. They are benevolent to each other in achieving this. On the other hand, self-interested agents pursue only their own goals. We model agents to be self-interested.

To conclude, we will study coordination for multi-agent systems with autonomous, self-interested agents. We assume these agents to:

(i) *not* take the planning process of other agents into account,

(ii) *not* revise their plans after planning,

(iii) *not* communicate during planning.

2.1.2 Actions and Classical Planning

Agents perform their activities by executing *actions*. A partially-ordered set of actions is a plan for the activity. Instead of 'activity', the terms 'problem' and 'task' are used in the field of *classical planning* [16]. This active field of planning research is based on the model of a *state transition system*. In a state transition system, an action $\alpha_{i,j}$ in a set of actions A triggers a transition between states s_i and s_j from a set of states S.

Actions have preconditions and effects. An action $\alpha \in A$ is said to be applicable in a state $s \in S$ if s satisfies the preconditions of α. Applying α in s triggers a state transition.

A plan is constructed from an initial state $s_0 \in S$ to one of a set of goal states $S_g \subseteq S$. We require *agents* to be able to construct such plans at the level of actions. We will not reason about such plans, but concern ourselves with *coordination* only.

To ensure coordination, we could identify the dependencies between actions as Dimopoulos and Moraitis [8] did. In this approach, however, coordination is interleaved with planning by *cooperative* agents. We rather abstract from the planning process and its details. To do so, we will define our coordination approach on a higher level of abstraction.

2.1.3 Tasks

In *Hierarchical Task Network* (HTN) planning [9], a hierarchical network of *tasks* is used to construct a plan at the level of actions. It starts by decomposing high-level tasks into subtasks, until the level of actions is reached. We will also use *tasks* to abstract away from the level of actions. Our notion of a task corresponds to a subproblem, that has to be solved by a planner.

In HTN planning, tasks are defined at various levels. On the other hand, in classical planning literature [10, 18, 20], the entire planning problem is often referred to as a 'planning task'. We will abstract away from actions such that the dependencies captured in these actions are preserved in the tasks. A task is therefore the highest level of abstraction at which all dependencies, *relevant for coordination*, are captured. We term these tasks *elementary tasks* as they are elementary for analysing coordination. An elementary task will often be simply referred to as a task and will be represented by a symbol t.

2.2 Task Coordination Framework

We will turn our attention to pre-planning coordination for self-interested, autonomous agents in a multi-agent setting. In this section, we formalise the concepts required for analysing this type of coordination in the *task coordination framework*. This framework has been introduced by Valk [35]. Our discussion is similar to Steenhuisen et al. [33]. Various extensions like a decomposition relation [4] and synchronisation constraints [32] have been proposed, but will not be discussed here.[4]

2.2.1 Tasks and Relations

First of all, we assume tasks to be *elementary*. An elementary task t is defined as in Section 2.1.3. That is, all relations between tasks relevant for coordination are captured at this level and all planning details are omitted. We consider all tasks to be adequately defined with respect to these criteria.

The only relation we identify between tasks is the *precedence relation* ($\prec$). It defines a partial order over the set of tasks. If task t has to *end* before task t' *starts*, we denote this as $t \prec t'$. We call this relation a *precedence relation*. An agent planning for these tasks has to ensure that t is completed before t' starts. All actions required for executing t will therefore precede the actions required for achieving t'.[5]

Together, a set of tasks T and a precedence relation $\prec$ over T constitute a planning problem. We define a planning problem in our framework as a *complex task* $\mathcal{T}$.

[4] In the initial work of Valk [35], capabilities for agents were defined as well. Capabilities are not relevant for our work and therefore not discussed.

[5] We define the precedence relation to be *transitively closed*. That is, if $t \prec t'$ and $t' \prec t''$, then we have $t \prec t''$ as well.

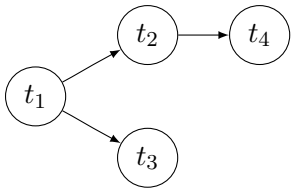


Figure 2.2: Complex task.

DEFINITION 2.1 (Complex Task)**.** *A* complex task *is a tuple* $\mathcal{T} = (T, \prec)$*, for which* T *is a set of* tasks *and* $\prec \subseteq T \times T$ *specifies a partial order over* T*.*

EXAMPLE 2.1. In Figure 2.2, a graphical representation for a complex task is shown. Each plan for this complex task will first have to achieve task t_1. Then, it can start with either t_2, t_3, or both tasks simultaneously. Task t_4 has to go after t_2.

In Example 2.1, we identified how a planner for the complex task could order the tasks in a plan. We will call this more restrictive ordering a *refinement* of the complex task. Refinements will be used for the discussion on plans, later on.

DEFINITION 2.2 (Refinement)**.** *A complex task* $\mathcal{T}' = (T', \prec')$ *is a* refinement *of another complex task* $\mathcal{T} = (T, \prec)$*, if and only if:*

$$T = T'$$
$$\prec \subseteq \prec'$$

2.2.2 Task Allocation

We define the set of agents as $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$. Before these agents can plan, tasks need to be *allocated* to them, as shown in Figure 2.1. For us, only the result of allocation is relevant; the task *assignment*. Hence, we assume an assignment to be given by some *task allocation function*:

$$f_{ta} : T \to \mathcal{A}. \tag{2.1}$$

The task allocation function f_{ta} induces a partitioning $\mathbf{T} = \{T_i\}_{i=1}^{n}$ of T into *subtasks*. Each subtask $T_i = \{t \in T \mid f_{ta}(t) = a_i\}$ is the set of tasks assigned to agent a_i. With the tasks assigned to an agent, the agent inherits the precedence relationships associated with these tasks. An agent is assigned a subtask of a complex task.

DEFINITION 2.3 (Complex Subtask)**.** *Given a complex task* $\mathcal{T} = (T, \prec)$*, a set of agents* $\mathcal{A}$*, and a task allocation function* f_{ta}*, a* complex subtask *for agent* $a_i \in \mathcal{A}$ *is a complex task* $\mathcal{T}_i = (T_i, \prec_i)$*, for which:*

$$T_i = \{t \in T \mid f_{ta}(t) = a_i\}$$
$$\prec_i = \prec \cap (T_i \times T_i)$$

All precedences *within* the subtask of an agent, $\prec_i$, will be taken into account. These are *intra-agent* precedences, $\prec_{\text{intra}}$. Precedences between tasks of different agents are omitted from these subtasks. These are *inter-agent* precedences. So, we have that: $\prec_{\text{intra}} \cap \prec_{\text{inter}} = \emptyset$ and $\prec = \prec_{\text{intra}} \cup \prec_{\text{inter}}$.

DEFINITION 2.4 (Inter-agent Precedences). *Given a complex task $\mathcal{T} = (T, \prec)$, a partitioning $\mathbf{T} = \{T_i\}_{i=1}^n$ of T, and a set of agents $\mathcal{A}$, with each T_i allocated to $a_i \in \mathcal{A}$, the set of* inter-agent precedences $\prec_{inter}$ *is defined by:*

$$\prec_{inter} = \prec \setminus \bigcup_{i=1}^{n} (T_i \times T_i)$$

Assigning tasks to agents results in a partitioning. We represent such a *partitioned complex task* for the complex task $(T, \prec)$ as:

$$(\{T_i\}_{i=1}^n, \{\prec_i\}_{i=1}^n \cup \prec_{\text{inter}}), \text{ or} \tag{2.2}$$

$$(\{T_i\}_{i=1}^n, \prec). \tag{2.3}$$

In Equation 2.3, we did not explicitly state the precedences that are projected onto an agent. These can be inferred from the task partitioning $\{T_i\}_{i=1}^n$.

2.2.3 Plans

During planning for its complex subtask $\mathcal{T}_i = (T_i, \prec_i)$, agent a_i plans for its complex task. It constructs a *refinement* for it. So, a *plan* for $\mathcal{T}$ is a partial order over T.

DEFINITION 2.5 (Plan). *Given a complex task $\mathcal{T} = (T, \prec)$, a* plan π *for $\mathcal{T}$ is a tuple $(T', \prec')$, which is a* refinement *of $\mathcal{T}$.*

Remark. A plan for a complex task $(T, \prec)$ is an *abstraction* of the concrete plan, at the level of actions. A concrete plan is a (partially) ordered set of *actions*, represented by: $(A, <)$.[6] An action α precedes α' if and only if $\alpha < \alpha'$. We are given a function that maps a task to the actions in the plan corresponding to it: $f_\alpha : T \to 2^A$.[7] Using this function, we can transform the concrete plan $(A, <)$ into the plan $\pi = (T, \prec')$. In π, we have $t \prec' t'$ if and only if $\forall \alpha \in f_\alpha(t), \forall \alpha' \in f_\alpha(t') : \alpha < \alpha'$.

Each agent constructs a plan for its own complex subtask. Consequently, to get a *joint plan* for the complex task, we have to *merge* the agents' plans.

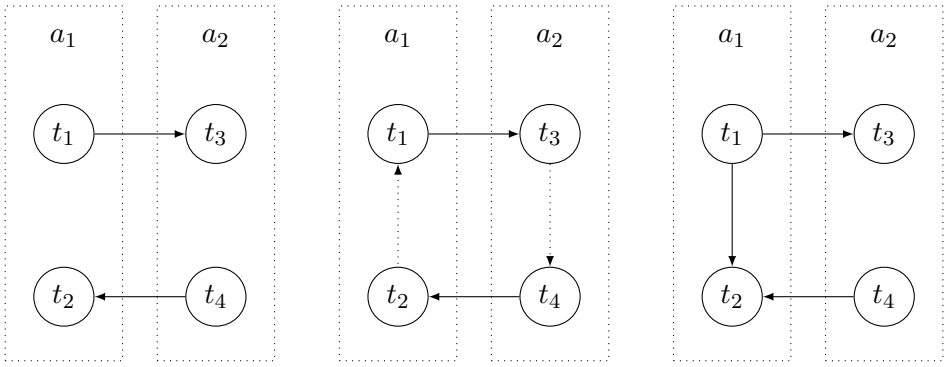
DEFINITION 2.6 (Joint Plan). *Given a partitioned complex task $\mathcal{T} = (\{T_i\}_{i=1}^n, \prec)$ and a plan $\pi_i = (T_i', \prec_i')$ for each subtask $(T_i, \prec_i)$, a* joint plan $\Pi = (T', \prec')$ *for $\mathcal{T}$ is defined by:*

$$T' = T = T_1' \cup T_2' \cup \cdots \cup T_n'$$
$$\prec' = \prec_{inter} \cup (\prec_1' \cup \prec_2' \cup \cdots \cup \prec_n').$$

If a plan or joint plan is *acyclic*, we say it is *feasible*. Otherwise, the plan is *in*feasible.

[6] A concrete plan transforms an initial state to a goal state for a state transition system. Each action in the plan triggers a state transition.

[7] 2^A is the power set of A.



(a) Partitioned complex task.

(b) Cyclic joint plan.

(c) Plan-coordinated.

Figure 2.3: Precedences between tasks of two agents.

2.2.4 Coordination

The autonomous agents we have defined are unwilling to revise the plans for their complex subtasks afterwards. So, when we merge their *feasible* plans into a joint plan, we have to ensure that the joint plan is feasible as well. Agents reason about their intra-agent precedences, but not about inter-agent precedences. These precedences pose a threat to the construction of a feasible joint plan, as we will see in the next example.

EXAMPLE 2.2. In Figure 2.3(a), a partitioned complex task $(\{T_i\}_{i=1}^{n}, \prec)$, assigned to two agents a_1 and a_2 is shown. Agent a_1 is assigned tasks t_1, t_2 and a_2 is assigned t_3, t_4. For this complex task, the set of intra-agent precedences is empty $\prec_{\text{intra}} = \emptyset$.

The two inter-agent precedences $t_1 \prec t_3$ and $t_4 \prec t_2$ pose a threat to the construction of a feasible joint plan. When allowing each agent to plan for its complex subtask, agent a_1 might come up with the plan $\pi_1 = (\{t_1, t_2\}, \{t_2 \prec t_1\})$ and a_2 with $\pi_2 = (\{t_3, t_4\}, \{t_3 \prec t_4\})$. This situation leads to a cyclic joint plan, as shown in Figure 2.3(b).

So, what could we do to ensure that each joint plan is feasible? We could prevent either of the precedence refinements $\{t_2 \prec t_1\}$ or $\{t_3 \prec t_4\}$. To achieve this, we add a single intra-agent precedence relation $t_1 \prec t_2$, as shown in Figure 2.3(c). Now, both agents can construct any plan without having a potential infeasible joint plan. Hence, the agents are coordinated with respect to their plans.

Coordination is required to ensure a feasible joint plan after merging. In Example 2.2, we saw that we can coordinate the agents by adding one intra-agent precedence. This is exactly the idea that we will use for *ensuring* coordination. Because we coordinate for the plans of agents, we call it *plan-coordination*.

First, we should know when a partitioned complex task is coordinated. We have to verify whether it is *impossible* for agents to come up with plans for their

complex subtasks, that merge into an infeasible joint plan.[8] This problem is the *plan-coordination verification problem*.

DEFINITION 2.7 (Plan-Coordination Verification Problem (PCV))**.** *Given a partitioned complex task* $(\{T_i\}_{i=1}^n, \prec)$*, does it hold for all* feasible plans $\pi_i = (T_i, \prec_i')$ *that the* joint plan *is acyclic?*

Each *yes*-instance of PCV is *plan-coordinated*. With respect to Example 2.2, we can conclude that the partitioned complex task of Figure 2.3(a) is *not* plan-coordinated. There exists a cyclic plan for this complex task. On the other hand, the refined complex task in Figure 2.3(c) *is* plan-coordinated.

In Example 2.2, the additional precedence limits agent a_1 in its planning freedom, while the complex subtask of agent a_2 is unaffected. While we will ensure plan-coordination by adding intra-agent precedences, our aim is to add a *minimal* number of such precedences. The set of precedences we add is referred to as the *coordination set* Δ.

When we ensure plan-coordination, we enable the agents to plan autonomously for their complex subtasks. They are able to plan independently from each other. So, we effectively *decouple* the agents with respect to plans. We call this *plan-decoupling*.

DEFINITION 2.8 (Plan-Decoupling Problem (PD))**.** *Given a partitioned complex task* $(\{T_i\}_{i=1}^n, \prec)$ *and an integer* $K > 0$*,*[9] *does there exist a* coordination set $\Delta = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$*, with* $\Delta_i \subset T_i \times T_i$*, such that:*

(i) $(\{T_i\}_{i=1}^n, \prec \cup \Delta)$ *is a yes-instance of PCV,*

(ii) $|\Delta| \leq K$*?*

Note that we achieve plan-decoupling by adding *intra-agent* precedences between tasks. We could have defined the plan-decoupling problem in various ways. Plan-coordination could be achieved by adding inter-agent precedences as well or by removing tasks.

Verifying whether a partitioned complex task is plan-coordinated and achieving plan-decoupling for a partitioned complex task are computationally hard problems. The PCV problem is coNP-complete, while the plan-decoupling problem is Σ_p^2-complete [35].

2.3 Instantiating Tasks

In Section 2.2.1, we assumed tasks to be *elementary*. In practise, however, tasks are an abstraction of actions that are based on preconditions and effect conditions. In

[8] We have to check whether the joint plan is always *acyclic*.

[9] For $K = 0$, PD is equal to PCV.

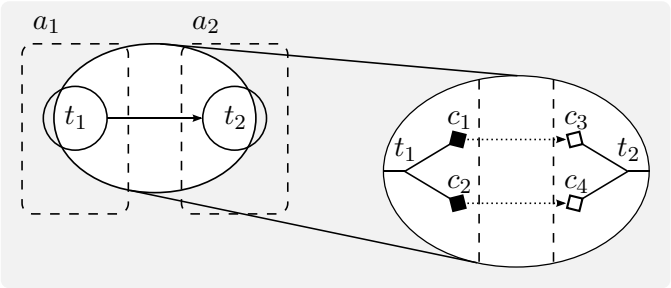


Figure 2.4: Dependencies between effects and preconditions of tasks.

classical planning, e.g., a planning task is represented by initial conditions, goal conditions, and a set of actions.

Because tasks correspond to subproblems, a task has initial and goal conditions as well. We will call these *preconditions* and *effect conditions*, short: effects. Preconditions indicate the conditions required to hold before the actions of a task can be executed. Effect conditions are achieved when the actions of the task have been executed.

EXAMPLE 2.3. Suppose we are given the complex task $\mathcal{T} = (\{t_1, t_2\}, \{t_1 \prec t_2\})$, with task t_1 assigned to agent a_1 and task t_2 to agent a_2. In the left hand side of Figure 2.4, the subtasks of $\mathcal{T}$ are shown.

In the right hand side of Figure 2.4, we zoom in on the tasks. We see that task t_1 has two possible effects c_1, c_2 and t_2 has two possible preconditions c_3, c_4. Either one of the effects could be instantiated during planning. Similarly, for task t_2, one of its preconditions will be instantiated. Because both tasks are assigned to different agents, the instantiations are constructed independently of each other. If either conditions c_1, c_4 or c_2, c_3 are instantiated, a_1 and a_2 are not coordinated with respect to their tasks.

The complex task of Figure 2.4 could e.g. model that soccer robot a_1 passes the ball (t_1) either forward (c_1) or backwards (c_2) to robot a_2. Robot a_2's task depends on this pass and on the direction of it. It should either move forward (c_3) or backwards (c_4).

As long as it is known what the *exact* preconditions and effects of tasks are, agents can be plan-decoupled. However, plan-coordination does *not* imply that these tasks are coordinated with respect to their instantiations. In practise, the outcome of tasks depends on the planning process as we saw in Example 2.3. Tasks could achieve one out of multiple effects, or require only some preconditions. These preconditions and effects are *instantiated* during planning. From Example 2.3, we learn that we have to coordinate for these instantiations of tasks as well. In the subsequent chapters, we will therefore address *coordination for instantiated plans*.

Chapter 3

Instantiation Coordination

We would like to coordinate for the instantiations agents can construct. Plan-coordination is insufficient for modelling this issue, as discussed at the end of Chapter 2. We should come up with a new approach for modelling this type of coordination. We will base this approach on plan-coordination and the task coordination framework. Consequently, we will enrich the task coordination framework in this chapter to facilitate *instantiated plan-coordination*.

Tasks will be enriched with preconditions and effects. Preconditions are conditions required to hold before the task is executed. Effects are conditions that are achieved by executing the task. Each task can have multiple preconditions and multiple effects that are *disjunctive*. Either one of the conditions is selected to be respectively required or achieved during executing the task. By selecting conditions for a task, the task is said to be *instantiated*.

In this chapter, we will focus on *coordination* for the instantiation of tasks that have *disjunctive* preconditions and effects. We will start in Section 3.1 by outlining the coordination problem. To reason about the problem more formally, in Section 3.2, we will discuss enrichments of the task coordination framework. Next, we will focus on conditions and the dependencies between them in Section 3.3. How to coordinate for tasks with disjunctive conditions will be the topic of Section 3.4. Finally, we will conclude this chapter in Section 3.5.

3.1 Disjunctive Conditions

In the task coordination framework, as discussed in Section 2.2, a task was modelled as a single unit of work. Although this representation is sufficient to coordinate the tasks with respect to precedences, a task is essentially more than that. For a task be executable, it is likely that some conditions on its surroundings have to hold. We say a task has a *precondition*, a condition required to hold right before the task is executed. By executing the task, it changes the environment in which it is

executed. We denote the new conditions it achieves by the task's *effect condition*, or short: *effect*. A *plan* for a task transforms its precondition into its effect condition.

When the precondition and effect condition of a task are known before planning, we say the task is *concrete*. On the other hand, we define *abstract* tasks as tasks for which the precondition and effect condition are not set before planning. For many tasks, one out of multiple preconditions should hold to execute the task. We call such a precondition a *disjunctive condition*. Both preconditions and effect conditions can be disjunctive. So, an abstract task has a disjunctive precondition or a disjunctive effect condition.

Tasks depend on each other by their preconditions and effects. A task's precondition is likely to be achieved by another task's effect. The task that achieves the effect has to be executed prior to the task that requires it. Such an ordering can be modelled by the *precedence relations* of the task coordination framework. This ordering alone is sufficient for concrete tasks, *not* for tasks with a disjunctive effect condition.

Suppose that two tasks depend on each other by a disjunctive condition with two elements. If the first task chooses to provide the first condition, the plan for the dependent task might still require the second condition to be achieved. Although the ordering between these tasks is sufficient, the tasks are not *coordinated*.

When we allow *autonomous agents* to plan for tasks with disjunctive conditions, they would require *communication* during planning agree on which condition to select. To assure autonomy of the agents, this coordination issue has to be resolved prior to planning. Therefore, we will address the following problem in this chapter:

> How to ensure that instantiated plans, constructed by autonomous, self-interested planning agents, can be merged into a feasible joint instantiated plan?

To analyse the coordination problem, we will first enrich the task coordination framework discussed in Section 2.2.

3.1.1 Assumptions

Related to classical planning, one might expect an *initial world state* and a *goal state* to be part of the planning problem. However, we only defined tasks. Implicitly, tasks incorporate the notion of an initial and goal state. All tasks that have no prerequisite tasks, which do not depend on the execution of other tasks, depend on the initial state. On the other hand, a goal state is modelled such that all tasks executed at the end, deliver the goal effects.

A task might have a disjunctive precondition and a disjunctive effect. During planning, one precondition and one effect will be selected. We do *not* model *dependencies* between preconditions and effects within a task. This would require reasoning about a plan for it. Having elementary tasks, we assume that for each precondition each effect can be achieved for a task.

3.2 Enriching the Task Coordination Framework

A *complex task* $\mathcal{T} = (T, \prec)$ is central to the task coordination framework, as discussed in Section 2.2. Along with a set of agents $\mathcal{A}$ and a task allocation function $f_{ta} : T \to \mathcal{A}$, plan-coordination between autonomous agents can be analysed. Here, we will extend this framework to make it suitable for coordination for tasks with disjunctive preconditions and effects.

For the notion of tasks with disjunctive preconditions and disjunctive effect conditions, the task coordination framework lacks detail about tasks. Modelling them as *elementary tasks* is not sufficient. Moreover, relations between these preconditions and effects cannot be represented in the framework. Therefore, we will incorporate both of these features in the framework such that we can reason about coordination for *abstract* tasks.

3.2.1 Tasks

To start with, we extend the definition of a task with disjunctive preconditions and effects.

DEFINITION 3.1 (Task). *A* task *t is a single unit of work, represented by a tuple (C_p, C_e), for which C_p is a set of preconditions and C_e a set of effect conditions, such that:*

- *right before the execution of t at least one condition $c_p \in C_p$ holds,*
- *right after the execution of t at least one condition $c_e \in C_e$ holds.*

We could represent the preconditions $C_p = \{c_{p_1}, c_{p_2}, \ldots, c_{p_n}\}$ of a task as a *disjunction* of preconditions: $c_{p_1} \vee c_{p_2} \vee \cdots \vee c_{p_n}$. Similarly, the set of effects $C_e = \{c_{e_1}, c_{e_2}, \ldots, c_{e_m}\}$ is a disjunction of the effects: $c_{e_1} \vee c_{e_2} \vee \cdots \vee c_{e_m}$. Within a disjunction, one or more conditions might hold to fulfil it.

A condition is represented by a boolean formula. To uniquely identify conditions, we use constant symbols from a formal language $\mathcal{L}$. Hence, a condition is a tuple (γ, ϕ) with γ a symbol from $\mathcal{L}$ and ϕ a boolean formula. In our discussion, we will only represent conditions by a tuple when it is appropriate.

Each *effect* condition can be identified by its symbol γ in $\mathcal{L}$. We require each effect to be identifiable in terms of the boolean formula as well. Hence, after executing an effect c_e we assume a literal l_γ to hold that can *only* be achieved by executing c_e and cannot be falsified.[1] Condition $c = (\gamma, \phi)$ is *achieved* and *holds* when its formula ϕ evaluates to true.

We will often use a convenient notation to denote the preconditions and effects of a task. For a task $t_i = (C_{p_i}, C_{e_i})$, we define $C_p(t_i) = C_{p_i}$ and $C_e(t_i) = C_{e_i}$. Both notations are interchangeable.

For a set of tasks T, we define $\mathcal{C}_p(T)$ to be the union of preconditions of $t \in T$: $\mathcal{C}_p(T) = \{c_p \mid c_p \in C_p, (C_p, C_e) \in T\}$. Similarly, $\mathcal{C}_e(T)$ is the union of the effects of all tasks: $\mathcal{C}_e(T) = \{c_e \mid c_e \in C_e, (C_p, C_e) \in T\}$.

[1] We require this, to define dependencies between conditions later on.

(a) Concrete task. (b) Abstract task.

Figure 3.1: Representation of tasks with preconditions and effects.

Concrete, Abstract, and Instantiated Tasks

We distinguish between three types of tasks: *abstract*, *concrete*, and *instantiated* tasks. An *abstract task* is a task with more than one precondition or more than one effect; $|C_p| > 1$ or $|C_e| > 1$. A *concrete task* is a task with one precondition and one effect condition, i.e., $|C_p| = 1$ and $|C_e| = 1$. Before executing tasks, they have to be *instantiated*. For such a task, we require one precondition and one effect to be chosen for execution.[2] We will use the notation: $t = (c_p, c_e)$.

Concrete tasks correspond to the tasks used in the *original coordination framework*, as discussed in Section 2.2. Essentially, tasks in the original framework have a single precondition and effect.

Remark. One might wonder why we define an instantiated task to have a single precondition and a single effect. In Definition 3.1, we defined *at least* one precondition to hold and *at least* one effect during execution. Hence, potentially, we could assume more conditions to hold.

However, it is up to a planner to decide whether *more than one* condition is made to hold. Our framework strives to abstract away from these planning details. Therefore, we assume an instantiated task to have a single precondition and effect.

In Figure 3.1, a concrete and an abstract task are shown. An open diamond represents a *precondition*, a solid one an *effect*. The line between preconditions and effects represents a plan for the task. While multiple plans might exist for a given task, the line does not represent a certain or single plan.

Instantiating Tasks

By pruning conditions from abstract tasks, we can *reduce* them into concrete tasks. For an abstract task t, we define all tasks it can be reduced to as a set of tasks:

$$\mathcal{P}(t) = 2^{C_p(t)} \times 2^{C_e(t)}. \tag{3.1}$$

We denote the set of concrete tasks by $\mathcal{P}_c(t)$ and the set of abstract tasks by $\mathcal{P}_a(t)$. These disjoint sets constitute $\mathcal{P}(t)$: $\mathcal{P}(t) = \mathcal{P}_c(t) \cup \mathcal{P}_a(t)$. For concrete task t we have that: $\mathcal{P}_c(t) = \{t\}, \mathcal{P}_a(t) = \emptyset$.

DEFINITION 3.2 (Concrete Task Instantiation)**.** *Given a task $t = (C_p, C_e)$, its* instantiation *is an* instantiated *task $t' = (c'_p, c'_e)$, for which $c'_p \in C_p$ and $c'_e \in C_e$.*

[2]Concrete and instantiated tasks are virtually the same. We classify them to be different to ease our discussion.

Given the set of tasks $\mathcal{P}_c(t)$ an abstract task t can be reduced to, we define the set of instantiations $I(t)$ for task t to be:

$$I(t) = \{(c_p, c_e) \mid c_p \in C_p, c_e \in C_e, (C_p, C_e) \in \mathcal{P}_c(t)\}. \tag{3.2}$$

Finally, we assume each instantiated task $t' \in I(t)$ to be *executable*. It implies that each precondition $c_p \in C_p(t)$ and effect $c_e \in C_e(t)$ of task t can form an instantiated task $(c_p, c_e) \in I(t)$.

3.2.2 Relations

Passing a ball between two soccer robots involves two tasks. The robot possessing the ball has to pass the ball. Its *effect* is the ball being at the other robot. The other robot has to take the pass to prevent the ball from bouncing away. Its *precondition* is to have the ball at its disposal. We say that the precondition of taking the pass *depends* on the effect of passing.

For conditions that depend on each other, we do not require their boolean formulas to be identical. Moreover, several effects might be required to fulfil a single precondition. We denote the *fulfilment* of precondition $c_p = (\gamma_{c_p}, \phi_{c_p})$ by effect condition $c_e = (\gamma_{c_e}, \phi_{c_e})$ by: $c_e \models c_p$. It corresponds to the tautological implication of their respective formulas: $\phi_{c_e} \models \phi_{c_p}$. Similarly, for a set of effect conditions $C_e = \{c_{e_1}, c_{e_2}, \ldots, c_{e_n}\}$ required to fulfil precondition c_p, we write $\bigwedge_{i=1}^{n} c_{e_i} \models c_p$. We say that precondition c_p *depends* on the effects in C_e.

EXAMPLE 3.1. Suppose we have two tasks $t_1 = (c_{p_1}, c_{e_1}), t_2 = (c_{p_2}, c_{e_2})$ with $c_{e_1} = (\gamma_1, \phi)$ and $c_{e_2} = (\gamma_2, \phi)$. It would imply that upon completion of task t_1 formula ϕ is satisfied. But then, condition c_{e_2} is satisfied as well without executing task t_2.

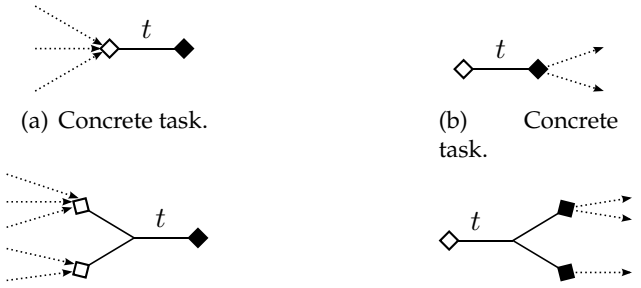
Situations, as sketched in Example 3.1, cannot occur in our framework. We assumed the boolean formula ϕ of each effect $c_e = (\gamma, \phi)$ to have a unique literal l_γ that holds when c_e is achieved. Hence, ϕ in Example 3.1 cannot represent a formula for two conditions.

DEFINITION 3.3 (Condition Dependency Relation)**.** *Given the set of all effect conditions $C_e = \{c_{e_1}, c_{e_2}, \ldots, c_{e_n}\}$ a precondition $c_p = (\gamma_{c_p}, \phi_{c_p})$* depends *on, with $c_{e_i} = (\gamma_i, \phi_i), 1 \leq i \leq n$, a* condition dependency $c_{e_i} \precsim c_p$[3] *exists if and only if:*

$$\bigwedge_{i=1}^{n} \phi_i \models \phi_{c_p} \textit{ and}$$

$$\forall C'_e \subset C_e : \bigwedge_{j=1}^{m} \phi_j \in C'_e \not\models \phi_{c_p}.$$[4]

[3]The condition dependency relation $\precsim$ is a set of (c_e, c_p)-tuples: $\precsim \subseteq \mathcal{C}_e \times \mathcal{C}_p$. For readability, we write $c_e \precsim c_p$.



(a) Concrete task. (b) Concrete task. (c) Abstract task. (d) Abstract task.

Figure 3.2: Condition dependencies for concrete and abstract tasks.

In Definition 3.3, we defined the *condition dependency relation* $\precsim$. If achieving a set of effects C_e is required for fulfilling precondition c_p on the level of formula, we defined that c_p depends on these effects C_e. All effects in C_e are required to hold for achieving c_p.

If a precondition $c_p = (\gamma_i, \phi_i)$ depends on an effect $c_e = (\gamma_j, \phi_j)$, we assume literal l_{γ_j} to be required to satisfy ϕ_i. In other words, a precondition that depends on an effect depends on the literal associated with achieving this effect.

Remark. A condition dependency $c_e \precsim c_p$ between an effect c_e and a precondition c_p models that achieving c_e 'enables' c_p. This relation is similar to the 'enables' relation used in the TÆMS-framework [21]. This relation, however, is defined over tasks, not over preconditions and effects.

Concrete and Abstract Tasks

In Figure 3.2, tasks are shown with condition dependencies depicted by dotted arrows. For *concrete tasks*, a precondition might require one or more effects to hold. In Figure 3.2(a), this is represented by multiple incoming arrows. All of these condition dependencies should hold for a successful execution of the task. On the other hand, in Figure 3.2(b), the two outgoing arrows represent two preconditions that depend on this task's effect.

In Figure 3.2(c), an *abstract task* is shown. By instantiating this task, at least one precondition is to be fulfilled. All effects the instantiated precondition depends on have to be achieved. For the preconditions not instantiated, it is irrelevant whether the effects they depend on are instantiated. For the task shown in Figure 3.2(d), at least one effect will be instantiated. Preconditions that depend on the instantiated effect can be used in an instantiation as well. However, for all effects not instantiated, we cannot ensure the dependencies to hold. Hence, all preconditions that depend on these effects cannot be used in an instantiation.

So far, we extended the plan coordination framework with (i) abstract tasks and (ii) dependencies between their conditions . Note that these extensions add modelling power to the framework. Between preconditions or effects of the same

task, we identified a *disjunctive relation*. Either one of the preconditions and effects can be instantiated. On the other hand, all dependencies related to a condition are *conjunctively* related. If an effect holds, all 'outgoing' condition dependencies hold. For a precondition to be fulfilled, all 'incoming' dependencies are required to hold.

Remark. Our discussion so far leads to a contradiction. On the one hand, we instantiate a single precondition and single effect per task. On the other hand, we allow these conditions to be involved in various dependencies.

Suppose we have tasks $t_i = (\{c_{p_i}\}, \{c_{e_{i,1}}, c_{e_{i,2}}\})$ and $t_j = (\{c_{p_j}\}, \{c_{e_{j,1}}, c_{e_{j,2}}\})$, with $c_{e_{i,1}} \precsim c_{p_j}$ and $c_{e_{i,2}} \precsim c_{p_j}$. Now, an instantiation for t_i should contain both $c_{e_{i,1}}$ and $c_{e_{i,2}}$ to fulfil c_{p_j}. However, we do not allow for such instantiations.

Hence, we require in our framework for any effects $c_{e_1}, c_{e_2} \in C_e(t)$ of any task t that we do *not* have $c_{e_1} \precsim c_p$ and $c_{e_2} \precsim c_p$ for any precondition c_p.

Precedence Relation

Not only tasks are affected by the extensions to the framework. The precedence relation $\prec$ is affected as well. Recall that it specifies a partial order over a set of tasks T. In our extended framework, the set T might contain *abstract tasks*.

After reducing and instantiating abstract tasks, precedences should be retained. If a precedence orders abstract tasks $t_1 \prec t_2$, their respective instantiations should be ordered as well: $t_1' \prec t_2'$. Precedences *propagate* under reduction and instantiation of tasks. Formally, if $t_1 \prec t_2$, then:

$$\forall t_1^{\mathcal{P}} \in \mathcal{P}(t_1), t_2^{\mathcal{P}} \in \mathcal{P}(t_2) : t_1^{\mathcal{P}} \prec t_2^{\mathcal{P}} \text{ and}$$
$$\forall t_1^{I} \in I(t_1), t_2^{I} \in I(t_2) : t_1^{I} \prec t_2^{I}.$$

Dependency and Precedence Relation

A dependency defines an *order* over effects and preconditions. By ordering these conditions, the tasks they constitute are implicitly ordered as well. For ordering tasks, we use *precedence relations*. One might think that a dependency between a task t's effect and a task t''s precondition implies an ordering $t \prec t'$. While true for *concrete tasks*, this does not necessarily hold for *abstract tasks*.

EXAMPLE 3.2 (Multiple Dependencies)**.** Suppose that we have three abstract tasks $t_1 = (\{c_{p_1}^1\}, \{c_{e_1}^1, c_{e_2}^1\})$, $t_2 = (\{c_{p_1}^2, c_{p_2}^2\}, \{c_{e_1}^2\})$, and $t_3 = (\{c_{p_1}^3, c_{p_2}^3\}, \{c_{e_1}^3\})$. For these tasks, two dependencies are defined: $c_{e_1}^1 \precsim c_{p_1}^2, c_{e_2}^1 \precsim c_{p_1}^3$. In Figure 3.3, these dependencies are shown. After planning for t, it becomes an instantiated task t'. This concrete task can be either $t' = (\{c_{p_1}^1\}, \{c_{e_1}^1\})$ or $t' = (\{c_{p_1}^1\}, \{c_{e_2}^1\})$. So, either one of the dependencies will become unnecessary. When we would have added the precedence relations $t_1 \prec t_2, t_1 \prec t_3$, it turns out that these would have been too restrictive.

From Example 3.2, we learn that a condition dependency between conditions of two tasks does not necessarily impose a precedence relation between these tasks.

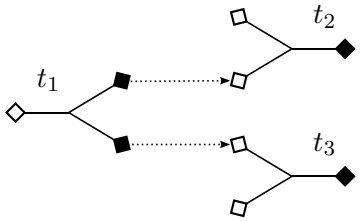
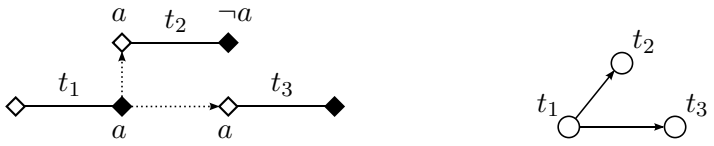


Figure 3.3: Dependency relations between three tasks.



(a) Condition dependencies. (b) Precedences.

Figure 3.4: Conflicting effects between tasks.

Therefore, by adding condition dependencies between tasks' conditions, we *increase* the representational power of the coordination framework.

Violated Conditions

Definition 3.3 poses a problem. A condition dependency $c_{e_i} \precsim c_{p_j}$ does not preclude other tasks from being executed between task t_i achieving c_{e_i} and task t_j requiring c_{p_j}. These intermediate tasks might falsify c_{e_i}, as we will see in the next example.

EXAMPLE 3.3. Suppose we are given the tasks and dependencies shown in Figure 3.4(a). It shows tasks $t_1 = (\{c_{p_1}\}, \{c_{e_1}\}), t_2 = (\{c_{p_2}\}, \{c_{e_2}\}), t_3 = (\{c_{p_3}\}, \{c_{e_3}\})$. For each effect we have $c_{e_i} = (\gamma_i, \phi_i^{c_e})$, for each precondition: $c_{p_i} = (\gamma_i, \phi_i^{c_p})$. Preconditions c_{p_2} and c_{p_3} depend on c_{e_1}. In terms of the associated boolean formulas, we have that $\phi_1^{c_e} = \phi_2^{c_p} = \phi_3^{c_p} = a \wedge l_{\gamma_1}$.

Dependencies impose an order over tasks, as shown in Figure 3.4(b). This partial order leaves two total orders: $\langle t_1, t_2, t_3 \rangle$ or $\langle t_1, t_3, t_2 \rangle$. In total order $\langle t_1, t_2, t_3 \rangle$, t_2 is executed before t_3. Hence, literal $\neg a$ will hold and $\neg a \wedge l_{\gamma_1} \not\models \phi_3^{c_p}$. In other words, t_3 cannot be executed.

In Example 3.3, task t_3 could not be executed in the plan $\langle t_1, t_2, t_3 \rangle$. However, recall from Section 2.1.2 that a task is performed by executing actions [33]. These actions achieve the preconditions and effects of tasks. So, condition a is achieved by some action. Like in the task coordination framework, we will assume that all actions are *reversible*.[5] For Example 3.3, this would imply that despite task t_2 being

[5]Each effect generated by a reversible action, can be reverted.

planned before t_3, the condition a can be achieved by a set of additional actions such that task t_2 is executable.

Abstract Complex Task

Having defined abstract tasks and condition dependencies, we are ready to extend the notion of a complex task. Recall, from Chapter 2, that a complex task is a tuple $\mathcal{T} = (T, \prec)$ of tasks and precedences between these tasks.

DEFINITION 3.4 (Abstract Complex Task)**.** *An* abstract complex task *is a 3-tuple* $\mathcal{T}^a = (T, \prec, \precsim)$*, for which the precedence relation* $\prec$ *specifies a partial order over* T*, a task* $t \in T$ *is a tuple* (C_p, C_e)*, and* $\precsim$ *is a set of condition dependency relations* $\precsim \subseteq \mathcal{C}_e(T) \times \mathcal{C}_p(T)$*.*

Remark. The way we model precedences and dependencies is similar to Partial-Order Causal Link (POCL) planning [39], also referred to as plan-space planning [16]. Planners like UCPOP [28], select actions, order them, and add causal links. Ordering actions in POCL is similar to the ordering of tasks by the precedence relation. Causal links represent condition dependencies between actions.

POCL planning differs from task-based planning as discussed here. Tasks are defined at an abstract level in our framework, whereas concrete actions are used in POCL.

3.2.3 Agents

Before a set of agents $\mathcal{A}$ can plan for an abstract complex task $\mathcal{T}^a = (T, \prec, \precsim)$, tasks need to be allocated to them. We assume there exists a task allocation function:

$$f_{ta} : T \to \mathcal{A} \tag{3.3}$$

Using this function, the set of tasks T is *partitioned* into subsets $T_1, T_2, \ldots, T_n$, with each T_i allocated to agent $a_i \in \mathcal{A}$. The task allocation function implies a task partitioning $\mathbf{T} = \{T_i\}_{i=1}^n$. A set of tasks T_i induces a set of precedence relations $\prec_i$ and condition dependencies $\precsim_i$. This induced problem is an *abstract complex subtask* for an agent.

DEFINITION 3.5 (Abstract Complex Subtask for an Agent)**.** *Given an* abstract complex task $\mathcal{T}^a = (T, \prec, \precsim)$ *and a task allocation function* f_{ta} *the* abstract complex subtask $\mathcal{T}_i^a = (T_i, \prec_i, \precsim_i)$ *an agent* a_i *has to plan for, is defined by:*

$$\begin{aligned} T_i &= \{t \mid t \in T, f_{ta}(t) = a_i\} \\ \prec_i &= \prec \cap (T_i \times T_i) \\ \precsim_i &= \precsim \cap (\mathcal{C}_e(T_i) \times \mathcal{C}_p(T_i)) \end{aligned}$$

Each task $t \in T$ of the abstract complex task $(T, \prec, \precsim)$ is allocated to exactly one agent. This allocation induces precedence relations and condition dependencies to

be allocated as well. However, not all relations are allocated to an agent. In Section 2.2, we identified the precedence relations not assigned to agents to be *inter-agent* precedence relations. They represent an order between tasks of different agents. We define *inter-agent condition dependencies* to be the dependencies between conditions of tasks that belong to different agents.

DEFINITION 3.6 (Inter-agent Condition Dependencies)**.** *Given an abstract complex task $(T, \prec, \precsim)$, a partitioning $\mathbf{T} = \{T_i\}_{i=1}^n$ of T, and a set of agents $\mathcal{A}$, with each T_i allocated to $a_i \in \mathcal{A}$, the set of* inter-agent dependencies *$\precsim_{inter}$ is defined by:*

$$\precsim_{inter} = \precsim \setminus \bigcup_{i=1}^{n} (\mathcal{C}_e(T_i) \times \mathcal{C}_p(T_i))$$

We represent the result of partitioning an abstract complex task $(T, \prec, \precsim)$ into n partitions, by: $(\{(T_i, \prec_i, \precsim_i)\}_{i=1}^n, \prec_{\text{inter}}, \precsim_{\text{inter}})$. However, a task partitioning induces a partitioning of precedence and condition dependency relations. Therefore, we will use the more concise notations $(\{T_i\}_{i=1}^n, \prec, \precsim)$ and $(\mathbf{T}, \prec, \precsim)$ to denote a *partitioned abstract complex task*.

3.3 Instantiating Tasks

In this section, we will focus on our extension to the plan coordination framework. Precedences can be *propagated* during instantiation, as discussed in Section 3.2.2. Precedences neither affect condition dependencies nor abstract tasks. Therefore, we will ignore them in this section. In Section 3.3.3, we will incorporate the *abstract complex task*.

So, instead of an abstract complex task $\mathcal{T}^a = (T, \prec, \precsim)$, we will reason about the *dependencies instance* $(T, \precsim)$ obtained by ignoring $\prec$.[6] Instead of a plan, we construct a *dependencies instantiation* for a *dependencies instance*. An instantiation is a tuple of a set of instantiated tasks T' and a set of condition dependencies $\precsim'$ between conditions of tasks in T'. We represent *all* possible instantiations of a set of tasks T by:

$$I(T) = \left\{ \{t'_1, t'_2, \ldots, t'_n\} \mid t'_i \in I(t)\,(1 \leq i \leq n)\,, t \in T \right\}. \tag{3.4}$$

Each $T' \in I(T)$ is a set of instantiated tasks that contains an instantiated task $t'_i \in T'$ for each task $t_i \in T$.

Instantiating the *tasks* of dependencies instance $(T, \precsim)$ implicitly *instantiates condition dependencies*. Dependencies between conditions that are not instantiated become irrelevant. If an effect is not instantiated, there cannot be a dependency on this effect. Similarly, if a precondition is not instantiated, it will not be dependent on any effect.

[6] Similarly, we define a partitioned dependencies instance as $(\{T_i\}_{i=1}^n, \precsim)$ or $(\mathbf{T}, \precsim)$.

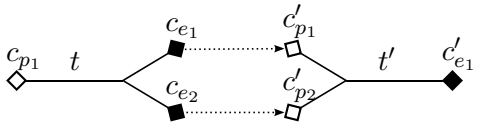


Figure 3.5: Dependencies instance with two feasible instantiations.

DEFINITION 3.7 (Dependencies Instantiation)**.** *Given a* dependencies instance $(T, \precsim)$*, a* dependencies instantiation $(T', \precsim')$ *for it, is defined by:*

$$\begin{aligned} T' &\in I(T) \\ \precsim' &= \precsim \cap \big(C_e(T') \times C_p(T')\big) \end{aligned}$$

A dependencies instantiation, as defined in Definition 3.7, need *not* be executable. The condition dependencies in an instantiation are inferred from the instantiations of tasks T', as defined by the second equation in Definition 3.7. Hence, preconditions might depend on effects that are not instantiated.

EXAMPLE 3.4. In Figure 3.5, a dependencies instance is shown with two tasks $t = (\{c_{p_1}\}, \{c_{e_1}, c_{e_2}\})$, $t' = (\{c'_{p_1}, c'_{p_2}\}, \{c'_{e_1}\})$ and condition dependencies $c_{e_1} \precsim c'_{p_1}, c_{e_2} \precsim c'_{p_2}$. Not all instantiations for this dependencies instance are executable. A feasible plan for t and t' would contain concrete tasks with either both c_{e_1} and c'_{p_1} instantiated, or both c_{e_2} and c_{p_2}. All other instantiations of t and t' would violate the dependencies.

In Example 3.4, we informally identified the *correctness* property of a dependencies instantiation. A dependencies instantiation $(T', \precsim')$ for a dependencies instance $(T, \precsim)$ is said to be *correct* if all condition dependencies are instantiated that the preconditions of tasks in T' depend on. Otherwise, it is *in*correct.

DEFINITION 3.8 (Correct Dependencies Instantiation)**.** *Given a dependencies instance* $(T, \precsim)$*, an instantiation* $(T', \precsim')$ *for it is* correct *if and only if* $\forall c_e \precsim c'_p$ *it holds that:*

$$(c'_p, c'_e) \in T' \rightarrow (c_p, c_e) \in T'$$

Furthermore, to be executable, the dependencies instantiation should be *acyclic*. Cyclic instantiations cannot be executed. Now, we can define a *feasible* dependencies instantiation. Such an instantiation $(T', \precsim')$ of a dependencies instance $(T, \precsim)$ should satisfy the two properties:

(i) $(T', \precsim')$ is *correct* with respect to $(T, \precsim)$,

(ii) $(T', \precsim')$ is *acyclic*.

DEFINITION 3.9 (Feasible Dependencies Instantiation)**.** *Given a dependencies instance* $(T, \precsim)$*, an instantiation* $(T', \precsim')$ *for it is* feasible *if and only if:*

(i) $(T', \precsim')$ *is* correct,

(ii) *after defining for all tasks* $(c_p, c_e) \in T'$ *a dependency* $c_p \precsim' c_e$*; the condition dependency relation* $\precsim'$ *is* acyclic*, with respect to* T'*.*[7]

An incorrect or cyclic dependencies instantiation is *in*feasible.

Merging Instantiations

Instead of a complex subtask for an agent, each agent a_i is allocated a sub-instance $(T_i, \precsim_i)$. This instance is defined like a complex subtask, ignoring the precedence relation. For a dependencies instance $(T, \precsim)$, a task-partitioning $\mathbf{T} = \{T_i\}_{i=1}^n$ induces the *partitioned dependencies instance* $(\{T_i\}_{i=1}^n, \precsim)$.

The next step is to merge those instantiations into a *joint dependencies instantiation*.

DEFINITION 3.10 (Joint Dependencies Instantiation)**.** *Given a partitioned dependencies instance* $(\{T_i\}_{i=1}^n, \precsim)$ *and an instantiation* $(T_i', \precsim_i{}')$ *for each sub-instance* $(T_i, \precsim_i)$*, a* joint instantiation $(T', \precsim')$ *for* $(\{T_i\}_{i=1}^n, \precsim)$ *is defined by:*

$$\begin{aligned} T' &= T_1' \cup T_2' \cup \cdots \cup T_n' \\ \precsim' &= \precsim_1{}' \cup \precsim_2{}' \cup \ldots \cup \precsim_n{}' \cup \left(\precsim_{\mathit{inter}} \cap \left(C_e(T') \times C_p(T')\right)\right) \end{aligned}$$

Just like for a dependencies instantiation, we require a *feasible* joint dependencies instantiation to be:

(i) *correct*,

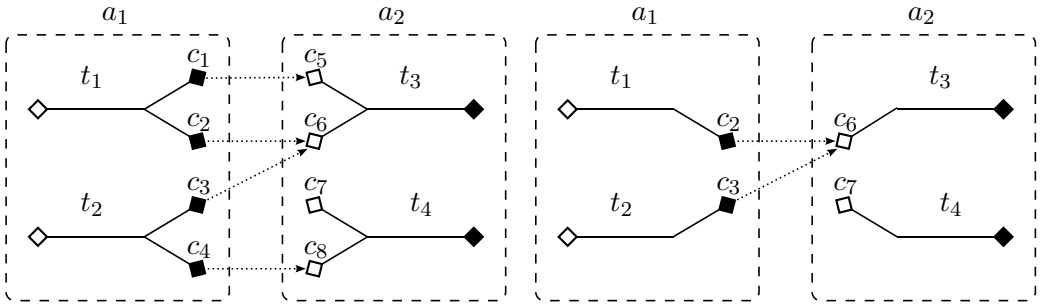
(ii) *acyclic*.

Merging instantiations into a joint instantiation is not trivial. For a set of feasible instantiations for sub-instances, the joint instantiation need *not* be feasible. We illustrate this with an example.

EXAMPLE 3.5. In Figure 3.6(a), a partitioned dependencies instance is shown. Each instantiation for an agent's sub-instances is feasible as the dependencies instance defines only inter-agent dependencies. A joint instantiation is feasible only when it is correct. If agent a_2 instantiates precondition c_6, it requires agent a_1 to instantiate effects c_2, c_3. Would a_1 instantiate effect c_1 or c_4, then the joint instantiation is infeasible.

In Section 3.4, we will turn our attention to this problem. There, we will discuss how to *coordinate* agents' instantiations.

[7] Note that we can define an effect to depend on a precondition. The dependency relation is defined over conditions and both preconditions and effects are conditions.



(a) A dependencies instance.

(b) A joint instantiation for the dependencies instance.

Figure 3.6: Two agents instantiating a dependencies instance.

3.3.1 Selecting and Pruning Dependencies

In Example 3.5, we saw that along with instantiating a precondition for a task by one agent, a condition of another agent should be instantiated as well. Moreover, agent a_2 itself is indirectly affected; instantiating c_6 imposes instantiating c_7 as well.

In this section, we will introduce the concept of *selecting* and *pruning* conditions and condition dependencies. By selecting conditions and effects, we *select* them to be part of an instantiation. Similarly, by *pruning* them, they will *not* be part of an instantiation. Essentially, we *reduce* abstract tasks when we prune conditions and dependencies from a dependencies instance.

Conditions and dependencies are always pruned or selected with respect to a dependencies instance.[8] After pruning or selecting, the dependencies instance is altered. We say it has been *reduced*. When a dependencies instance $(T, \precsim)$ is reduced until all tasks are *concrete*, the instance $(T, \precsim)$ can be trivially transformed into an *instantiation*.

DEFINITION 3.11 (Condition Selection)**.** *Given a dependencies instance $(T, \precsim)$, a condition $c \in (\mathcal{C}_p(T) \cup \mathcal{C}_e(T))$ is* selected *if for all instantiations $(T', \precsim')$ for $(T, \precsim)$, it holds that:*

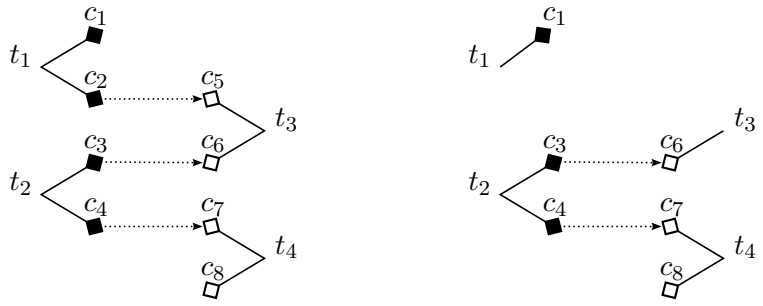
$$c \in (\mathcal{C}_p(T') \cup \mathcal{C}_e(T')).$$

Selecting a condition for a dependencies instance implies its use in *every instantiation* for the reduced dependencies instance. Similarly, *pruning* a condition implies that it is *not* used in any instantiation.

DEFINITION 3.12 (Condition Dependency Selection)**.** *Given a dependencies instance $(T, \precsim)$, a dependency $c_i \precsim c_j$ is* selected *if for all instantiations $(T', \precsim')$ for $(T, \precsim)$, it holds that:*

$$c_i \precsim' c_j.$$

[8] In fact, conditions are pruned from *abstract tasks*. While these tasks are defined for a dependencies instance, we say that we prune this instance.



(a) Original instance.

(b) Reduced instance (c_1 selected).

Figure 3.7: A dependencies instance.

Selecting a condition dependency $c_e \precsim c_p$ implies the selection of both conditions c_p and c_e. Similarly, *pruning* a condition dependency $c'_e \precsim c'_p$ implies that c'_p is *not* used in any instantiation.

In practise, one would often like to ensure that a dependency is selected, irrespective of whether the dependent precondition is selected. This is an *enabled* condition dependency.

DEFINITION 3.13 (Enabled Condition Dependency)**.** *Given a dependencies instance $(T, \precsim)$, a dependency $c_i \precsim c_j$ is* enabled *if for all instantiations $(T', \precsim')$ for $(T, \precsim)$, it holds that:*

$$c_i \in \mathcal{C}_e(T').$$

Propagation of Selecting and Pruning

Selecting an effect $c_e \in C_e$, with $|C_e| > 1$, makes the other effects in C_e irrelevant. Therefore, we say that the other effects are *pruned* as a result of selecting c_e. The selection of a condition *propagates* through a dependencies instance.

EXAMPLE 3.6. In Figure 3.7(a), a dependencies instance is shown.[9] Selecting effect c_1 would imply that effect c_2 is pruned. Consequently, the condition dependency $c_2 \precsim c_5$ cannot be made to hold and has to be pruned as well. This, in turn, implies that dependency $c_2 \precsim c_5$ cannot be made to hold and precondition c_5 has to be pruned as well. The reduced dependencies instance after selecting c_1 is shown in Figure 3.7(b).[10]

From Example 3.6, we learn that selecting or pruning a single condition potentially has a lot of impact on the dependencies instance. The selection or pruning of a dependency might propagate through an instance. In Table 3.1, an overview

[9] We use a concise representation for the relevant preconditions and effects of tasks only.

[10] In fact, the instance can be reduced even further. This will be shown in Example 3.7.

	Operation	Result of propagation (single step)	
Precondition $c_p \in C_p$	Select	Selected	$c_e \precsim c_p$
		Pruned	$C_p \setminus \{c_p\}$
	Prune	Selected	if $C_p = \{c_p, c'_p\}$ then c'_p, otherwise none
		Pruned	$c_e \precsim c_p$
Effect $c_e \in C_e$	Select	Selected	-
		Enabled	$c_e \precsim c_p$
		Pruned	$C_e \setminus \{c_e\}$
	Prune	Selected	if $C_e = \{c_e, c'_e\}$ then c'_e, otherwise none
		Pruned	$c_e \precsim c_p$
Dependency $c_e \precsim c_p$, $c_e \in C_e$, $c_p \in C_p$	Select	Selected	c_e, c_p
		Pruned	-
	Enable	Selected	c_e
		Pruned	-
	Prune	Selected	-
		Pruned	c_p

Table 3.1: Operations to reduce a dependencies instance.

of the different types of propagation is shown. Next, we will discuss each of these into more detail.

Precondition *Selecting* a precondition results in the selection of all condition dependencies it depends on. On the other hand, all other preconditions in the disjunctive precondition will be pruned.

Pruning a precondition results in a selection of another precondition only if it is part of a disjunctive precondition of size two. Because this precondition is removed, the other has to be instantiated. All dependencies it has can be pruned, as the pruned condition will not depend on effects.

Effect *Selecting* an effect prunes all other effects in the disjunctive precondition. All dependencies on the selected effect will be enabled.

Pruning an effect results in selecting the other effect only if it is part of a disjunctive effect of size two. Removing this effect requires the other effect to be

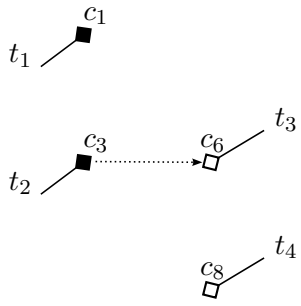


Figure 3.8: A reduced dependencies instance.

instantiated. All dependencies that depend on this effect will be pruned.

Dependency *Selecting* a dependency requires both its effect and precondition to be part of each instantiation. Therefore, both the effect and precondition are selected.

Enabling a dependency requires the effect related to it to be part of each instantiation. It is irrelevant whether the dependent precondition holds.

Pruning a dependency prevents the dependent precondition to hold. Therefore, it is pruned. The effect is not affected as it is still allowed to hold in an instantiation.

A dependencies instance can be reduced to contain concrete tasks only by repeatedly selecting and pruning conditions. An instance with concrete tasks can be converted into an instantiation. Hence, an instantiation can be achieved by repeatedly selecting, pruning, and propagating.

EXAMPLE 3.7. In Example 3.6, the dependencies instance of Figure 3.7(a) was reduced to the instance of Figure 3.7(b). We can propagate even further for this instance. It is clear that the dependency $c_3 \precsim c_6$ has to be selected. Otherwise, the task t_3 cannot be instantiated. As a result, effect c_3 will be selected and c_4 is pruned. Eventually, $c_4 \precsim c_7$ is pruned along with c_7. This leads to the instance shown in Figure 3.8. All its tasks are concrete and therefore the instance can be trivially instantiated.

Cyclic Dependencies

A cyclic instantiation of a dependencies instance cannot be executed. Therefore, we required *instantiations* to be *acyclic*. On a higher level, a dependencies *instance* might be cyclic.

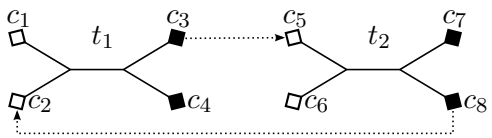


Figure 3.9: Cyclic dependency conditions between tasks.

DEFINITION 3.14 (Cyclic Dependencies Instance). *A dependencies instance* $(T, \precsim)$ *is* cyclic *if and only if* $\precsim$ *is cyclic with respect to* T*, after adding the dependencies:*

$$\precsim = \precsim \cup \{c_p \precsim c_e \mid c_p \in C_p(t), c_e \in C_e(t), t \in T\}.$$

A dependencies instance is cyclic if adding the dependencies of a task's effects on its preconditions, creates a cycle in the dependencies on conditions. Instantiating a cyclic dependencies instance might result in a cyclic instantiation. However, a cyclic dependencies instance does not preclude an *acyclic* instantiation for it, as we will see in the next example.

EXAMPLE 3.8. In Figure 3.9, a cyclic dependencies instance is shown with tasks t_1, t_2 and dependencies $c_3 \precsim c_5, c_8 \precsim c_2$. Clearly, instantiations $t'_1 = (c_2, c_3)$ and $t'_2 = (c_5, c_8)$ result in a infeasible plan, with cycle $t'_1 - t'_2 - t'_1$. However, all other instantiations of t_1 and t_2 are acyclic.[11]

An acyclic instantiation exists for the dependencies instance of Example 3.8. Hence, it would have been too constraining to prevent cycles in the dependencies instance. Therefore, we will *not* require dependencies instances to be acyclic.

3.3.2 Consistent and Minimal Dependencies Instances

A prerequisite for a feasible joint instantiation is the *existence* of a feasible instantiation for a given dependencies instance. If there exists a feasible instantiation for a dependencies instance, we call such an instance *consistent*. Otherwise, the instance is *in*consistent. Note that requiring a dependencies instance to be consistent is weaker than requiring it to be acyclic.

EXAMPLE 3.9. In Figure 3.10(a), three tasks are shown. Task t_1 has a disjunctive effect, while both t_2 and t_3 have a single precondition. In a feasible instantiation, t_1 should be instantiated with effect condition c_1. Instantiating t_1 with c_2 would invalidate $c_1 \precsim c_3$ and would therefore prevent task t_2 from execution. While there exists a feasible instantiation for this instance, it is consistent.

On the other hand, observe the instance of Figure 3.10(b). Here, both condition dependencies are required for tasks t_2 and t_3. However, we can at most instantiate a single effect of t_1. Either $c_1 \precsim c_3$ is violated or $c_2 \precsim c_4$ is. While there exists no feasible instantiation for this instance, it is *in*consistent.

(a) Consistent. (b) Inconsistent.

Figure 3.10: Consistency of a dependencies instance.

Checking consistency for the instances of Example 3.9 seems to be trivial. However, checking whether any given dependencies instance is consistent turns out to be hard.

DEFINITION 3.15 (Consistent Dependencies Problem (CD))**.** *Given a dependencies instance* $(T, \precsim)$*, does there exist a* feasible instantiation $(T', \precsim')$ *for it?*

We will prove CD to be NP-complete. First, we will show that CD is in NP. Thereafter, we reduce 3-SAT to CD to prove it to be NP-hard.

PROPOSITION 3.1. *CD is in* NP*.*

Proof. Given an instantiation $(T', \precsim')$ of the dependencies instance $(T, \precsim)$, we have to check whether it is *feasible*. An instantiation is feasible if it is *correct* and *acyclic*. Both properties can be checked in polynomial time.

Correct Each task $t' \in T'$ is concrete and therefore has a single precondition and effect. For each precondition c_p, it is checked whether for all effects c_e it depends on ($c_e \precsim c_p$), $c_e \precsim' c_p$ holds. This can be checked in polynomial time.

Acyclic To check whether an instantiation is cyclic, we can represent the instance as a directed graph $G = (V, A)$. For each task $t'_i \in T'$ we construct a vertex $v_i \in V$. For each condition dependency $c_j \precsim' c_k, c_j \in \mathcal{C}_e(t'_j), c_k \in \mathcal{C}_p(t'_k), j \neq k$ we create an arc $(v_j, v_k) \in A$. For each vertex $v \in V$ of G we can determine in polynomial time whether it is reachable from itself. If so, G is cyclic. Otherwise, it is acyclic.

□

Proving CD to be in NP does not prove CD to be difficult to solve. By proving the next proposition, we will show CD to be intractable for any given input, unless P = NP.

PROPOSITION 3.2. *CD is* NP*-hard.*

[11]Note that not all other instantiations are correct, e.g. $t'_1 = (c_2, c_4), t'_2 = (c_5, c_7)$.

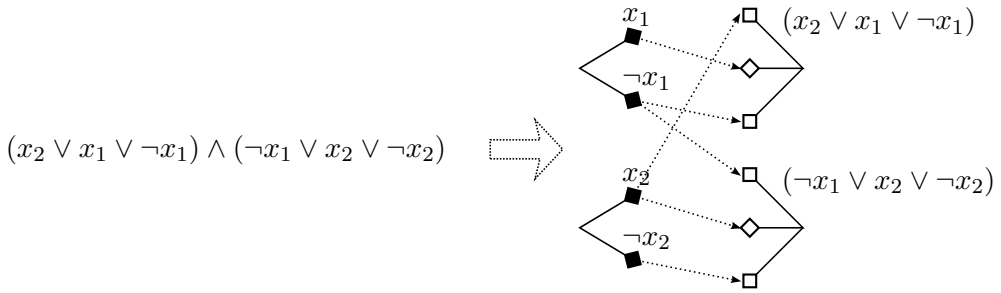


Figure 3.11: Transformation from 3-SAT to CD.

Proof. We will prove that CD is NP-hard by reducing 3-SAT to CD. More specifically, we reduce 3-SAT to the problem of determining whether a *correct* dependencies instantiation exists for a dependencies instance.

A 3-SAT instance consists of a set of clauses $\mathcal{C} = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$ over a set of variables $X = \{x_1, x_2, \ldots, x_m\}$. Each clause contains three literals; $\forall \gamma \in \mathcal{C} : |\gamma| = 3$.

First, we transform a 3-SAT instance to a dependencies instance. For each variable $x_i \in X$ we define a task $t_i = (c_{p_i}, \{c_{e_i}, c'_{e_i}\})$ with a disjunctive effect of size two. Effect c_{e_i} corresponds to the literal x_i and effect c'_{e_i} corresponds to literal $\neg x_i$.

A clause $\gamma_j \in \mathcal{C}$ is of the form $(x_q \vee x_r \vee x_s)$.[12] For each clause, we construct a task $t_j = (\{c^1_{p_j}, c^2_{p_j}, c^3_{p_j}\}, c_{e_j})$ with $c^1_{p_j}$ corresponding to x_q, $c^2_{p_j}$ to x_r, and $c^3_{p_j}$ to x_s. If the first literal in γ_j is positive (x_q), a condition dependency $c_{e_q} \precsim c^1_{p_j}$ is created. Otherwise, for a negative literal $\neg x_q$, we create a dependency $c'_{e_q} \precsim c^1_{p_j}$. Similarly, we add dependencies for literals x_r or $\neg x_r$ and x_s or $\neg x_s$. In Figure 3.11, an example of this process is shown.

The tasks defined for *variables* will be denoted by the set T_x. All tasks defined for *clauses* are represented by the set T_γ. For a dependencies instance $(T, \precsim)$, $T = T_x \cup T_\gamma$.

A yes-instance of 3-SAT should transform into a yes-instance of CD under a valid reduction. A certificate for 3-SAT is a truth assignment to the variables in X. It is a set of literals $L = \{l_1, l_2, \ldots, l_m\}$. For each literal $l_i \in L$ either $l_i = x_i$ or $l_i = \neg x_i$ evaluates to true.

While L is a certificate for the 3-SAT instance, it makes each clause $\gamma \in \mathcal{C}$ evaluate to true. At least one literal per clause evaluates to true, which corresponds to at least one precondition $c_p \in C_p(t_\gamma)$ for each task $t_\gamma \in T_\gamma$ to hold. Each variable $x \in X$ is assigned a truth value by the certificate. Hence, one effect $c_e \in C_e(t_x)$ per task $t_x \in T_x$ holds. In short, for a yes-instance of 3-SAT each task $t \in T$ is correctly instantiated.

Recall from Definition 3.8 that for all instantiated preconditions, the effects they depend on should be instantiated as well. We defined a condition dependency

[12] Literals in a clause can be negative as well.

from the effect corresponding to a literal to preconditions corresponding to the occurrence of that literal. Each precondition depends on a single effect. As the precondition corresponds to the literal, the effect will hold as well.

Given a yes-instance for CD, under the transformation from 3-SAT to CD, it should correspond to a yes-instance for 3-SAT. A certificate for CD is a *feasible* dependencies instantiation $(T', \precsim')$. Each task $t' \in T'$ is an instantiated task. For each task $t_i \in T_\gamma$ corresponding to clause $\gamma_i \in \mathcal{C}$, we have that $|C_p(t_i)| = 1$. This corresponds to a single literal in γ_i to evaluate to true and thus γ_i to evaluate to true. On the other hand, for each task $t_j \in T_x$ we have that either c_{e_j} or c'_{e_j} holds. These effects correspond to respectively literals x_j and x'_j.

The dependencies instantiation $(T', \precsim')$ is *correct*. Each effect an instantiated precondition depends on is therefore instantiated as well. A condition dependency $c_{e_q} \precsim c_p$, with $c_e \in C_e(t_q), t_q \in T_x$ and $c_p \in C_p(t_r), t_r \in T_\gamma$, was created when both c_e and c_p correspond to the same literal. No other dependencies than these have been constructed by our reduction.

Given a feasible dependencies instantiation, we have correspondingly: (i) a truth-assignment L to all variables in X, (ii) each clause $\gamma \in \mathcal{C}$ is satisfied. Therefore, a yes-instance of 3-SAT corresponds to a yes-instance of CD under the reduction.[13] □

PROPOSITION 3.3. *CD is* NP*-complete.*

The NP-completeness of CD follows directly from Propositions 3.1 and 3.2.

While it is possible to construct inconsistent dependencies instances, we will assume in subsequent sections that all dependencies instances given are consistent. When appropriate, we will state this in each of our definitions.

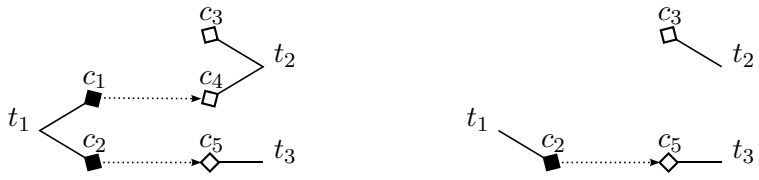
Minimal Dependencies Instance

Consistency of a dependencies instance implies that *at least one* feasible instantiation exists for it. Suppose a single instantiation is feasible for an instance. If the instance contains any disjunctive preconditions or effects, only one condition per disjunctive condition can be feasibly instantiated. All other conditions are *redundant*.

EXAMPLE 3.10. The dependencies instance of Figure 3.12(a) is consistent. An instantiation of tasks that contains conditions c_2, c_5, c_3 is feasible. For this instance, condition c_1 will not be used in any instantiation because c_2 is required for precondition c_5. It is therefore redundant. As a result, precondition c_4 is redundant as well. In Figure 3.12(b), the dependencies instance without redundant conditions is shown.[14]

[13] In our proof, we did not check for acyclicity as the reduction always yields an acyclic dependencies instance.

[14] Note that this is an instantiation for the instance of Figure 3.12(a) as well.



(a) Non-minimal. (b) Minimal.

Figure 3.12: Consistent dependencies instances.

By pruning redundant preconditions and effects, a new dependencies instance is created. If *no* preconditions and effects can be pruned from a dependencies instance $(T, \precsim)$, without reducing the set of feasible instantiations, the dependencies instance is called *minimal*. That is, for the set of conditions $C = \mathcal{C}_p(T) \cup \mathcal{C}_e(T)$ the number of conditions $|C|$ is minimal.

DEFINITION 3.16 (Minimal Dependencies Instance)**.** *A dependencies instance* $(T, \precsim)$ *is* minimal *if and only if:*

(i) *for each precondition* $c_p \in \mathcal{C}_p(T)$*, a feasible instantiation* $(T', \precsim')$ *for* $(T, \precsim)$ *exists, for which* $c_p \in \mathcal{C}_p(T')$*,*

(ii) *for each effect* $c_e \in \mathcal{C}_e(T)$*, a feasible instantiation* $(T', \precsim')$ *for* $(T, \precsim)$ *exists, for which* $c_e \in \mathcal{C}_e(T')$*.*

3.3.3 Instantiation and the Abstract Complex Task

So far, we discussed the implications of our extension in isolation. We reasoned about a dependencies instance and instantiation. This allowed for a clean analysis of their properties. Now, we would like to discuss our extensions with respect to the *abstract complex task*, defined in Definition 3.4.

Incorporating precedences in the previous definitions, related to dependencies instances and instantiations, is often straightforward. Therefore, we would like to focus on two aspects of the abstract complex task: (i) *consistency* and (ii) the relation between *instantiations and plans*.

Consistency

Similar to a dependencies instance, we define an abstract complex task to be *consistent* if and only if at least one *feasible plan* exists for it. We will reuse the notion of consistency for a dependencies instance. To do so, we transform the abstract complex task into a dependencies instance. If and only if this dependencies instance is consistent, the abstract complex task is consistent.

Transformation of an abstract complex task into a dependencies instance can be performed by Algorithm 3.1. We define a dummy task $t_{i,j}$ for each precedence between tasks $t_i \prec t_j$. Each precondition of $t_{i,j}$ depends on a single effect of t_i.

Algorithm 3.1 Transformation of an abstract complex task into a dependencies instance.

Input: Abstract complex task $(T, \prec, \precsim)$.
Output: Dependencies instance $(T^\star, \precsim^\star)$.

```
T* = T
≾* = ≾
for all t_i ≺ t_j do
  define task t_{i,j} = (C_{p_{i,j}}, c_{e_{i,j}})
  T* = T* ∪ {t_{i,j}}
  for all c_e ∈ C_e(t_i) do
    define a corresponding precondition c_{p_{i,j}} ∈ C_{p_{i,j}}
    define c_e ≾* c_{p_{i,j}}
  end for
  for all c_p ∈ C_p(t_j) do
    define c_{e_{i,j}} ≾* c_p
  end for
end for
```

Each precondition of task t_j depends on the single effect of $t_{i,j}$. This way, no matter which instantiation is constructed, t precedes t'.

To test if an abstract complex task $(T, \prec, \precsim)$ is consistent, we transform it into a dependencies instance $(T^\star, \precsim^\star)$ by applying Algorithm 3.1. We can check whether $(T^\star, \precsim^\star)$ is consistent by checking for CD. If it is consistent, there exists a *feasible* dependencies instantiation $(T', \precsim')$ for it. This instantiation is both correct and acyclic.

DEFINITION 3.17 (Consistent Abstract Complex Task)**.** *Given an abstract complex task $(T, \prec, \precsim)$, it is* consistent *if and only if given $(T, \prec, \precsim)$ as input to Algorithm 3.1, Algorithm 3.1's output $(T^\star, \precsim^\star)$ is a yes-instance of CD.*

Instantiations and Plans

Having defined consistency for a complex task, one might wonder what the result of *planning* for an abstract complex task will be. In the original framework, a *plan* $(T', \prec')$ is constructed for a complex task $(T, \prec)$. In our extension, a *dependencies instantiation* $(T', \precsim')$ is constructed for a dependencies instance $(T, \precsim)$. We will merge both concepts in the *instantiated plan*.

DEFINITION 3.18 (Instantiated Plan)**.** *Given an abstract complex task $\mathcal{T}^a = (T, \prec, \precsim)$, an* instantiated plan $\pi' = (T', \prec', \precsim')$ *for $\mathcal{T}^a$ is defined by:*

(i) $(T', \prec')$ *is a refinement of* $(T, \prec)$*,*

(ii) $(T', \precsim')$ *is an instantiation of* $(T, \precsim)$*.*

An instantiated plan is *feasible* if and only if $(T', \precsim')$ is *correct* with respect to $(T, \precsim)$ and $(T', \prec', \precsim')$ is *acyclic* under the transformation of Algorithm 3.2. Algo-

Algorithm 3.2 Transformation of a dependencies instantiation into a complex task.

Input: Instantiated plan $(T', \prec', \precsim')$.
Output: Plan $(T, \prec)$.
  for all $t'_i \in T'$ **do**
    define $t_i \in T$
  end for
  for all $t'_i \prec' t'_j; t'_i, t'_j \in T'$ **do**
    define $t_i \prec t_j; t_i, t_j \in T$
  end for
  for all $c_{e_{t'_i}} \precsim c_{p_{t'_j}}; t'_i, t'_j \in T'$ **do**
    define $t_i \prec t_j; t_i, t_j \in T$
  end for

rithm 3.2 converts an instantiated plan into a plan for a complex task. Each concrete task is transformed into a task for the complex task. All dependencies between a precondition of task $t'_i \in T'$ and effect of $t'_j \in T'$ are converted into a precedence $t_i \prec t_j$. Recall from Section 3.2.2 that precedences model the order imposed by dependencies between concrete tasks.

DEFINITION 3.19 (Joint Instantiated Plan)**.** *Given a partitioned abstract complex task* $\mathcal{T}^a = (\{T_i\}_{i=1}^n, \prec, \precsim)$ *and an instantiated plan* $(T'_i, \prec'_i, \precsim'_i)$ *for each subtask* $(T_i, \prec, \precsim)$*, a* joint instantiated plan $\Pi' = (T', \prec', \precsim')$ *for* $\mathcal{T}^a$ *is defined by:*

$$
\begin{aligned}
T' &= T'_1 \cup T'_2 \cup \cdots \cup T'_n \\
\prec' &= \prec'_1 \cup \prec'_2 \cup \cdots \cup \prec'_n \cup \prec_{\mathit{inter}} \\
\precsim' &= \precsim'_1 \cup \precsim'_2 \cup \cdots \cup \precsim'_n \cup \left(\precsim_{\mathit{inter}} \cap \left(\mathcal{C}_e(T') \times \mathcal{C}_p(T')\right)\right)
\end{aligned}
$$

A joint instantiated plan is simply an instantiated plan, but for a partitioned abstract complex task. Hence, a joint instantiated plan is *feasible* if and only if $(T', \precsim')$ is *correct* with respect to $(\{T_i\}_{i=1}^n, \precsim)$ and $(T', \prec', \precsim')$ is *acyclic* under the transformation of Algorithm 3.2.

3.4 Coordination

In Example 3.5, agent a_1 was affected by a_2's selection of preconditions. Not *every* instantiation constructed by a_1 can be merged with *every* instantiation of a_2's tasks. Both agents have to *coordinate* their instantiations, such that their instantiations can be merged.

In Section 2.2.4, we identified the problem of plan-coordination with respect to precedences between tasks. By adding a coordination set of precedences Δ, complex tasks could be *decoupled*, by which *coordination* is ensured. In this section, we will extend the notions of coordination and decoupling for the *enriched framework* to include the notion of dependencies instances.

Partitioned Abstract Complex Task

We will define coordination and decoupling in this section for a partitioned abstract complex task $(\{T_i\}_{i=1}^n, \prec, \precsim)$. Each agent $a_i \in \mathcal{A}$ is allocated its abstract complex subtask $(T_i, \prec_i, \precsim_i)$. During planning, it constructs an instantiated plan $\pi'_i = (T'_i, \prec'_i, \precsim'_i)$ for it. After planning, agents' plans are merged into the joint instantiated plan $\Pi = (T', \prec', \precsim')$.

An instantiated plan π'_i for subtask $(T_i, \prec_i, \precsim_i)$ is assumed to be *feasible*. That is, it is executable by the agent that constructed it. However, the joint instantiated plan Π' might be *in*feasible. Inter-agent precedences and dependencies are introduced during plan merging and pose two threats to the joint plan: (i) the joint instantiated plan is *cyclic* or (ii) the joint instantiated plan is *incorrect*.

3.4.1 Coordination Verification

Suppose a *consistent* abstract complex task is allocated to a set of autonomous agents. We would like to know whether these agents are allowed to construct *each feasible instantiated plan* for their subtask. Can all feasible instantiated plans for these sub-instances be merged into a *feasible joint instantiated plan*?

In other words, given a partitioned consistent abstract complex task, we would like to verify if it is *impossible* for agents to construct a feasible instantiated plan for their subtasks, which merge into an *in*feasible joint instantiated plan. This is the *instantiation-coordination verification problem*.

DEFINITION 3.20 (Instantiated Plan-Coordination Verification Problem (IPCV))**.** *Given a* consistent *partitioned abstract complex task* $(\{T_i\}_{i=1}^n, \prec, \precsim)$*, does it hold for* all feasible instantiated plans $(T'_i, \prec'_i, \precsim'_i)$ *that the joint instantiated plan* $(T', \prec', \precsim')$ *is* feasible*?*

IPCV is a generalisation of the PCV problem, defined in Definition 2.7. If we provide IPCV with a partitioned complex task with only concrete tasks and no dependencies, the input reads $(\{T_i\}_{i=1}^n, \prec, \emptyset)$. This input equals the input to PCV. A *feasible* instantiated plan is then equal to an *acyclic* plan.

Because PCV is coNP-complete, we know that IPCV is coNP-hard. However, we might tighten this result to IPCV being coNP-complete as well. It would imply that adding the additional representation of abstract tasks and dependencies to the coordination framework does *not* harden the complexity of coordination verification for it.

PROPOSITION 3.4. *IPCV is in* coNP*.*

Proof. To prove that IPCV is in coNP, we have to show that a *no*-instance of ICV can be verified in polynomial time. A no-instance of IPCV is a set of feasible instantiated plans $\{(T'_i, \prec'_i, \precsim'_i)\}_{i=1}^n$ for which the joint plan Π' is *not* feasible. Given such a set of plans, we can construct in polynomial time the *joint plan* $\Pi' = (T', \prec', \precsim')$ for

it. The joint plan is feasible if $(T', \precsim')$ is a *correct* instantiation for the dependencies instance $(\{T_i\}_{i=1}^n, \precsim)$ and Π' is acyclic.

A joint instantiated plan that is cyclic or incorrect is *in*feasible. Hence, we have verified a no-instance of IPCV. □

Adding the notion of abstract tasks and dependencies does not harden the result for coordination verification. We can, however, not conclude that these enrichments are trivial. By proving the following propositions, we will show that verifying coordination for partitioned dependencies instances is not trivial.[15] It turns out to be coNP-complete as well.

DEFINITION 3.21 (Instantiation-Coordination Verification Problem (ICV))**.** *Given a* consistent *partitioned dependencies instance* $(\{T_i\}_{i=1}^n, \precsim)$*, does it hold for* all feasible instantiations $(T_i', \precsim_i')$ *that the joint instantiation* $(T', \precsim')$ *is* feasible*?*

We will show that ICV is coNP-hard by reducing an NP-complete problem to the complementary problem of ICV. This is the Instantiation-Coordination Contradiction Problem (ICC): does a set of instantiations exist for which the joint instantiation is *in*feasible?

For this reduction, we use the NP-complete *path with forbidden pairs problem* (PWFP) [12]. We will adapt this problem from Garey and Johnson [13].

DEFINITION 3.22 (Path with Forbidden Pairs Problem (PWFP))**.** *Given a directed acyclic graph* $G = (V, A)$*, two vertices* $s, t \in V$*, and a set of forbidden pairs* $\mathcal{F} = \{\{a_1, a_1'\}, \{a_2, a_2'\}, \ldots, \{a_n, a_n'\}\}$*, does a path exist from* s *to* t *that contains at most one arc from each pair* $\{a_i, a_i'\} \in \mathcal{F}$*?*

PROPOSITION 3.5. *ICV is* coNP*-hard.*

The proof of coNP-hardness of ICV is based on the proofs of coNP-completeness of the plan-coordination verification problem, that is defined in Ter Mors [34] and Valk [35].

Proof. We will represent an instance of PWFP by the tuple $(G, \mathcal{F}, s, t)$. Given an instance of PWFP, we have to transform it into an instance of ICC; a partitioned dependencies instance $(\mathbf{T}, \precsim)$.

First, we define how to transform vertices and the arcs that are *not* involved in forbidden pairs. We construct for each of the n vertices $v_i \in V$ a task partition $T_i = \{t_i\}$. Task t_i is a *concrete* task and has therefore a single precondition c_p^i and a single effect c_e^i. An arc $(v_i, v_j) \in A$, that is not involved in a forbidden pair, is converted into a condition dependency $c_e^i \precsim c_p^j$.

Second, we have to transform the arcs in the forbidden pairs to the ICC instance. For a forbidden pair $p = \{(v_a, v_b), (v_c, v_d)\}$, not both arcs (v_a, v_b) and (v_c, v_d) can be in the path $s - t$. To represent this, we will construct a task partition for which a cyclic instantiation exists. We transform a forbidden pair $p_j = \{(v_a, v_b), (v_c, v_d)\}$

[15] A dependencies instance is an abstract complex task without precedences; $(T, \emptyset, \precsim)$.

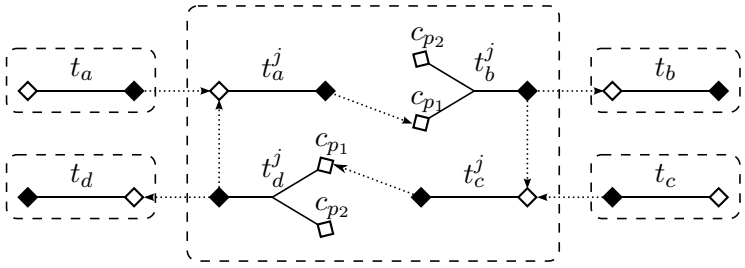


Figure 3.13: Transformation from PWFP forbidden pair to a task partition in ICC.

into a task partition $T_{n+j} = \{t_a^j, t_b^j, t_c^j, t_d^j\}$. Tasks $t_a^j = (c_p^{a_j}, c_e^{a_j})$ and $t_c^j = (c_p^{c_j}, c_e^{c_j})$ are concrete, $t_b^j = (\{c_{p_1}^{c_j}, c_{p_2}^{c_j}\}, c_e^{c_j})$ and $t_d^j = (\{c_{p_1}^{d_j}, c_{p_2}^{d_j}\}, c_e^{d_j})$ have a disjunctive precondition of size two and a single effect. In Figure 3.13, the tasks corresponding to a forbidden pair are shown.

For forbidden pair $p_j = \{(v_a, v_b), (v_c, v_d)\}$, either of the arcs cannot be used in a path from s to t. We want to model this in the transformation of a forbidden pair as well. Therefore, we add the condition dependencies $c_e^{a_j} \precsim c_{p_1}^{b_j}, c_e^{b_j} \precsim c_p^{c_j}, c_e^{c_j} \precsim c_{p_1}^{d_j}, c_e^{d_j} \precsim c_p^{a_j}$. In a feasible instantiation for this sub-instance, either $c_e^{a_j} \precsim c_{p_1}^{b_j}$ or $c_e^{c_j} \precsim c_{p_1}^{d_j}$ will not be selected. In Figure 3.13, these condition dependencies are shown.

By reducing PWFP to ICC, we will let a path from s to t for PWFP correspond to a cyclic joint instantiation for ICC. We therefore add dependencies $c_e(t_a) \precsim c_p^{a_j}$, $c_e^{b_j} \precsim c_p(t_b)$, $c_e(t_c) \precsim c_p^{c_j}$, and $c_e^{d_j} \precsim c_p(t_d)$. If a path from s to t contains an arc of the forbidden pair, there is a corresponding path through the instantiation of the dependencies instance.

Finally, we will have to represent s and t in the ICC instance. For both vertices, a corresponding task partition has already been constructed with a single task. Given a path from s to t, we would like to have a cyclic ICC instance. Suppose s has been converted into task $t_s = (c_p^s, c_e^s)$ and t in $t_t = (c_p^t, c_e^t)$. We could simply add the dependency $c_e^t \precsim c_p^s$. A path from s to t would then correspond with a cyclic joint instantiation. However, suppose a trivial path $s-t$ exists in the form of an arc (s, t). This would create a cyclic instantiation. However, it would imply as well that the dependencies instance is *inconsistent*. There is no instantiation for this instance that is *acyclic*. To accommodate this issue, we construct a task partition $T_{n+k+1} = \{t'_s, t'_t\}$. Task $t'_t = (c_p^{t'}, c_e^{t'})$ is a concrete task, $t'_s = (\{c_{p_1}^{s'}, c_{p_2}^{s'}\}, c_e^{s'})$ is an abstract task with two disjunctive preconditions. We add dependencies $c_e^{t'} \precsim c_{p_1}^{s'}$, $c_e^t \precsim c_p^{t'}$, and $c_e^{s'} \precsim c_p^s$ to have a consistent instance for which a cyclic instantiation can be constructed.[16] In Figure 3.14, an example transformation is shown from PWFP to ICC.

[16]By e.g. selecting all preconditions that do not depend on an effect, a feasible instantiation is constructed. No cycles within task partitions exist and there is no path through partition T_{n+k+1}.

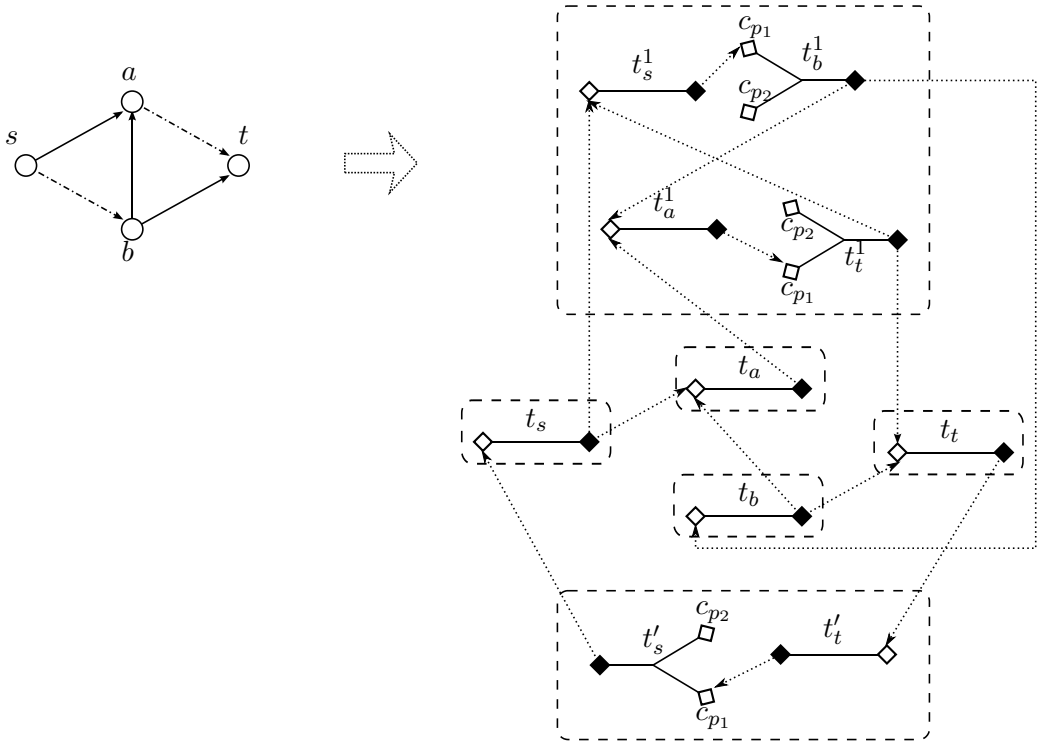


Figure 3.14: Transformation of a PWFP instance to a ICC instance.

A yes-instance for PWFP should correspond to a yes-instance for the constructed ICC. A yes-certificate for PWFP is a path from s to t that contains at most one arc from each forbidden pair $\{(v_a, v_b), (v_c, v_d)\}$. From the path $s-t$, all arcs that are in a forbidden pair can be extracted. This is the set A_p. We identify two cases; A_p is either empty or not.

$A_p = \emptyset$ No forbidden pair is used in the path $s-t$. Therefore, no cyclic instantiation is created in a task partition that corresponds to a forbidden pair. However, we can trivially create a cycle by selecting dependency $c_e^{t_{t'}} \precsim c_{p_1}^{t_{s'}}$. This is a yes-instance of ICC.

$A_p \neq \emptyset$ At least one arc from a forbidden pair is used in the path $s - t$. Per task partition, at most one arc from the forbidden pair is used. The corresponding instantiation for the agent is therefore always acyclic. When we, again, select $c_e^{t_{t'}} \precsim c_{p_1}^{t_{s'}}$, the joint instantiation is clearly cyclic. We have a yes-instance of ICC.

A yes-instance for ICC is an instantiation for the dependencies instance $(T, \precsim)$ that is *not* feasible. Such an instantiation can be incorrect or cyclic.

Note that the constructed instances cannot yield an incorrect instantiation. Each task has a single effect condition and is therefore always instantiated. Consequently, all condition dependencies are selected. No precondition can be selected such that it requires some other effect to hold.

To construct an infeasible instantiation, it has to be *cyclic*. Cyclic instantiations can be constructed in two ways. Either local to an agent in a task partition corresponding to a forbidden pair or by instantiating $c_e^{t_{t'}} \precsim c_{p_1}^{t_{s'}}$. An instantiation for a task partition that corresponds to a forbidden pair cannot be cyclic because we assumed agents to construct a feasible instantiation. Therefore, $c_e^{t_{t'}} \precsim c_{p_1}^{t_{s'}}$ has to be selected in every cyclic instantiation.

A cyclic instantiation has to contain a cycle that involves multiple agents. While the graph G of the PWFP instance is acyclic, the constructed ICC instance is acyclic as well.[17] Therefore, a path from t_s to t_t has to exist to create a cycle. If such a path runs through a task partition for a forbidden pair, only one of the dependencies that correspond to the forbidden pair will be selected. Otherwise, it would imply a cyclic instantiation for the partition. □

PROPOSITION 3.6. *ICV is* coNP*-complete.*

The coNP-completeness of ICV follows directly from Propositions 3.4 and 3.5.

For a yes-instance of ICV, the agents can construct instantiations for their dependencies instance independently of each other. It allows for a *concurrent* instantiation of all sub-instances. Agents can only construct instantiations independent of each other when they know which inter-agent dependencies will be *selected* and which will be *pruned*. Hence, concurrency requires all inter-agent condition dependencies to be either selected or pruned, prior to planning. For each selected dependency $c_{e_i} \precsim_{\text{inter}} c_{p_j}$, c_{e_i} has been selected. For each pruned dependency $c_{e_k} \precsim_{\text{inter}} c_{p_l}$, c_{p_l} has been pruned.

Minimal Dependencies Instance

Even for a *minimal* dependencies instance, ICV turns out to be coNP-complete. With a minor restriction to the PWFP problem, we can prove this proposition.

PROPOSITION 3.7. *ICV is* coNP*-complete for a* minimal *dependencies instance.*

Proof. Recall that for a minimal dependencies instance $(T, \precsim)$ each precondition $c_p \in \mathcal{C}_p(T)$ and effect $c_e \in \mathcal{C}_e(T)$ is part of at least one feasible instantiation. While all tasks have a single effect, these are always instantiated. Hence, we only have to show that when each *precondition* is part of a feasible instantiation, ICV is still coNP-hard.

For the task partitions that correspond to a forbidden pair, it is clear that there exists a feasible dependencies instantiation for each of the disjunctive preconditions. In such an instantiation, we should *avoid* selecting the dependency $c_e^{t'} \precsim c_{p_1}^{s'}$.

[17] If we assume feasible instantiations for the task partitions corresponding to forbidden pairs.

If we *do* select this dependency, a trivial path $s - t$ would prevent minimality for the dependencies instance. Precondition $c_{p_1}^{s'}$ then, cannot be part of a feasible instantiation. But, if we require each path $s - t$ to contain *at least one arc* from a forbidden pair, we *do* have a minimal instance. If an instantiation contains precondition $c_{p_1}^{s'}$, we select for all other task partitions the preconditions that do not depend on an effect.

In Valk [35], it is shown that PWFP remains NP-complete under the restriction that a path $s - t$ contains *at least one arc* from a forbidden pair. This allows us to conclude that ICV is coNP-complete for minimal dependencies instances as well. □

3.4.2 Ensuring Coordination

Given a partitioned abstract complex task, we can verify whether it is *instantiated plan-coordinated* or not. If it is, autonomous agents can construct instantiations for their sub-instances concurrently. Ultimately, we would like to *ensure* instantiated plan-coordination. When coordinating the agents for an abstract complex task, we effectively *decouple* it into independent subtasks. For each of the subtasks, an instantiated plan can be constructed in isolation. We will therefore use the term *instantiated plan-decoupling* for our coordination mechanism to ensure instantiated plan-coordination.

Several coordination mechanisms exist. One could e.g. remove tasks from the abstract complex task, prune conditions, or add precedences. The mechanism we propose *prunes conditions* and adds *precedences*. This way, we can ensure that each *consistent* partitioned abstract complex task can be decoupled.

Instantiated plan-decoupling will come at a price. By transforming an abstract complex task, some of the freedom agents have in choosing among conditions and precedences will be limited. Therefore, we will try to *minimise* the impact of instantiated plan-decoupling on the abstract complex task.

For the Plan-Decoupling Problem in Section 2.2.4, we minimised the number of intra-agent precedences added. However, an abstract complex task contains both dependencies and precedences. These relations interfere with each other. In the next example, we will present a simple case in which they interfere.

EXAMPLE 3.11. In Figure 3.15, four tasks are shown with precedences $t_1 \prec t_3, t_4 \prec t_2$ and dependencies $c_{e_2} \precsim c_{p_1}, c_{e_3} \precsim c_{p_4}$. Not both c_{p_1} and c_{p_4} can be in a feasible instantiated plan, as it would be cyclic. To coordinate this abstract complex task, we could prune either of these preconditions. However, agents could add precedences between the tasks. As in plan-decoupling, we should add a precedence $t_1 \prec t_2$ or $t_4 \prec t_3$ to prevent a cyclic joint instantiated plan. Which precedence to add is up to which preconditions are instantiated.

We define the Instantiated Plan-Decoupling Problem (IPD) to be the problem of determining whether a *consistent* partitioned abstract complex task can transformed into a *decoupled* partitioned abstract complex task. Our optimisation cri-

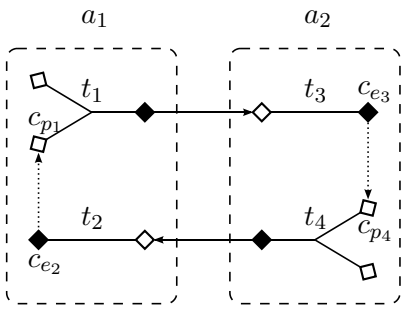


Figure 3.15: Abstract Complex Task

terion for decoupling a partitioned abstract complex task will encompass (i) the number of added precedences and (ii) the number of pruned conditions.

DEFINITION 3.23 (Instantiated Plan-Decoupling Problem (IPD))**.** *Given a* consistent *partitioned abstract complex task* $\mathcal{T}^a = (\{T_i\}_{i=1}^n, \prec, \precsim)$ *with* $T = T_1 \cup T_2 \cup \cdots \cup T_n$ *and an integer* K*, can* $\mathcal{T}^a$ *be* transformed *into an abstract complex task* $(\{T_i'\}_{i=1}^n, \prec', \precsim')$ *with:*

- $C \subset \mathcal{C}_p(T) \cup \mathcal{C}_e(T)$ *the set of all conditions pruned from* $\mathcal{T}^a$*,*
- $\Delta = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$ *a coordination set with* $\Delta_i \subset T_i \times T_i$ *and* $\prec' = \prec \cup \Delta$*,*

such that:

(i) *IPCV is a yes-instance for* $(\{T_i'\}_{i=1}^n, \prec', \precsim')$*,*

(ii) $|C| + |\Delta| \leq K$*?*

IPD is a generalisation of the PD problem, defined in Definition 2.8. If we provide IPD with no dependencies and concrete tasks $(\{T_i\}_{i=1}^n, \prec, \emptyset)$, IPD equals PD.

PD is Σ_2^p-complete. Hence, IPD is Σ_2^p-hard. We will tighten this bound to Σ_2^p-completeness. The additional representational power of abstract tasks and dependencies does *not* increase the complexity of decoupling.

PROPOSITION 3.8. *IPD is in* Σ_2^p*.*

Proof. For a consistent partitioned abstract complex task $(\mathbf{T}, \prec, \precsim)$ and $K > 0$, a certificate for IPD is a coordination set Δ of added precedences and a set of conditions C to be pruned. We have to verify whether:

(i) the result, $(\mathbf{T}', \prec', \precsim')$, of adding precedences in Δ to, and pruning the conditions in C from $(\mathbf{T}, \prec, \precsim)$ is a yes-instance for IPCV,

(ii) $|C| + |\Delta| \leq K$.

Checking whether $|C| + |\Delta| \leq K$ is trivial. Next, we have to construct the abstract complex task $(\mathbf{T}', \prec', \precsim')$. Adding precedences in Δ and pruning conditions in C can be performed in polynomial time. Verifying whether this instance is a yes-instance for IPCV is coNP-complete. Therefore, IPD is in Σ_2^p. □

Just like for verifying coordination, instantiated plan-decoupling for a consistent partitioned abstract complex task is not harder than plan-decoupling. To show that decoupling with respect to dependencies instances is not trivial, we will prove instantiated plan-decoupling for these instances to be Σ_2^p-complete as well. We call this problem the *instantiation-decoupling problem*.

DEFINITION 3.24 (Instantiation-Decoupling Problem (ID))**.** *Given a* consistent *partitioned dependencies instance* $(\{T_i\}_{i=1}^n, \precsim)$ *with* $T = T_1 \cup T_2 \cup \cdots \cup T_n$ *and an integer* K*, can* $(\{T_i\}_{i=1}^n, \precsim)$ *be* reduced *into an instance* $(\{T_i'\}_{i=1}^n, \precsim')$ *for which* $C \subset \mathcal{C}_p(T) \cup \mathcal{C}_e(T)$ *is the set of all conditions pruned from* $(\{T_i\}_{i=1}^n, \precsim)$*, such that:*

(i) *ICV is a yes-instance for* $(\{T_i'\}_{i=1}^n, \precsim')$*,*

(ii) $|C| \leq K$*?*

ID turns out to be Σ_2^p-complete. To prove Σ_2^p-hardness of ID, we will construct a reduction from the *partitioned path with forbidden pairs* problem [34, 35].

DEFINITION 3.25 (Partitioned Path with Forbidden Pairs Problem ($\exists\forall\neg$PWFP))**.** *Given a PWFP instance* $(G = (V, A), \mathcal{F}, s, t)$ *and a partitioning* $\{\mathcal{F}_1, \mathcal{F}_2\}$ *of* $\mathcal{F}$*, does a set* $\mathcal{X}_1 = \{a \mid \{a, a'\} \in \mathcal{F}_1\}$ *exist such that for every* $\mathcal{X}_2 = \{a \mid \{a, a'\} \in \mathcal{F}_2\}$*, there does* not *exist a path from* s *to* t *for the set of arcs* $A' = (A \setminus \{a, a' \mid \{a, a'\} \in \mathcal{F}\}) \cup \mathcal{X}_1 \cup \mathcal{X}_2$*?*

For the $\exists\forall\neg$PWFP problem, an arc is selected from every forbidden pair. The set of forbidden pairs is split into two subsets $\mathcal{F}_1, \mathcal{F}_2$. The problem is to determine if a selection of arcs from the forbidden pairs in p_1 can be made such that no path from s to t exists in the directed, acyclic graph G. Irrespective of the arc selected for each forbidden pair in $\mathcal{F}_2$, no path $s - t$ exists.

EXAMPLE 3.12. In Figure 3.16, a PWFP instance is shown with two forbidden pairs $\{a_1, a_2\}, \{a_3, a_4\}$. If we partition them into the sets $\mathcal{F}_1 = \{\{a_3, a_4\}\}$ and $\mathcal{F}_2 = \{\{a_1, a_2\}\}$ we have a yes-instance for the $\exists\forall\neg$PWFP problem. To see this, we let $\mathcal{X}_1 = \{a_3\}$. Clearly, no path from s to t exists whatever choice for $\mathcal{X}_2$.[18]

In Valk [35], $\exists\forall\neg$PWFP is proven to be Σ_p^2-complete. We will use this result to prove Proposition 3.9. The proof for this proposition is similar to the proof of Σ_2^p-hardness of the plan-coordination problem defined in the same work of Valk [35] and related work of Ter Mors [34].

PROPOSITION 3.9. *ID is* Σ_2^p*-hard.*

[18] For $\mathcal{F}_1 = \{\{a_1, a_2\}\}$ and $\mathcal{F}_2 = \{\{a_3, a_4\}\}$, the instance would be a yes-instance as well.

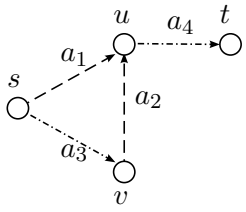


Figure 3.16: A PWFP instance.

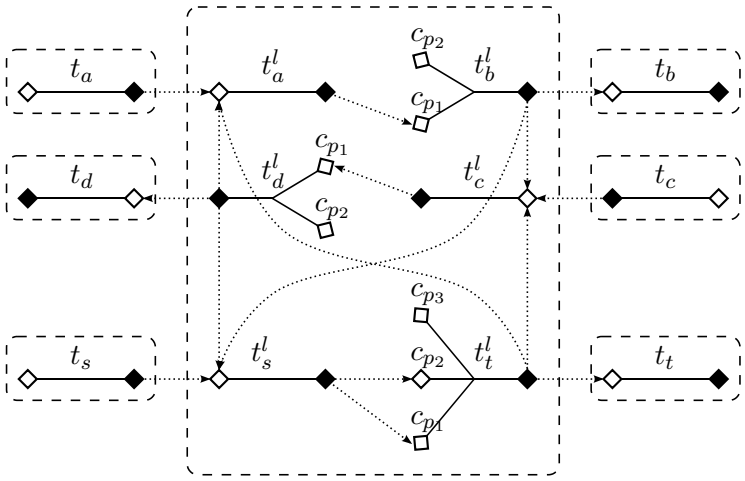


Figure 3.17: Transformation from $p \in \mathcal{F}_1$ in ∃∀¬PWFP to a task partition in ID.

Proof. We start by transforming an instance $(G, \mathcal{F}_1 \cup \mathcal{F}_2, s, t)$ of ∃∀¬PWFP into an instance $(\mathbf{T}, \precsim)$ for ID. This transformation is similar to the transformation used in proving ICV coNP-hard.

Again, for each of the n vertices $v_i \in V$ we define a task partition $T_i = \{t_i\}$, which contains a single concrete task. For each arc $(v_i, v_j) \in A$ *not involved* in a forbidden pair, we construct the dependency $c_e^i \precsim c_p^j$.

We *only* transform each of the m forbidden pairs $p_j = (\{(v_a, v_b), (v_c, v_d)\})$ in $\mathcal{F}_2$ into a task partition $T_{n+j} = \{t_a^j, t_b^j, t_c^j, t_d^j\}$, as shown in Figure 3.13 and discussed in the proof of the coNP-hardness result for ICV.

The ∃∀¬PWFP problem is to find a set $\mathcal{X}_1$ such that no path exists from s to t. On the other hand, for ID, we have to find a set of conditions C to prune such that *no cyclic* instantiation can be constructed for the given instance. The goal of this reduction is to link the pruned conditions to the arcs in $\mathcal{X}_1$. This, we will model by creating a more sophisticated task partition for each forbidden pair $p \in \mathcal{F}_1$, which is shown in Figure 3.17.

For each of the K forbidden pairs $p_l = \{(v_a, v_b), (v_c, v_d)\} \in \mathcal{F}_1$ we construct a task partition $T_{n+m+l} = \{t_a^l, t_b^l, t_c^l, t_d^l, t_s^l, t_t^l\}$. Tasks t_a^l, t_c^l, t_s^l are concrete, tasks t_b^l, t_d^l

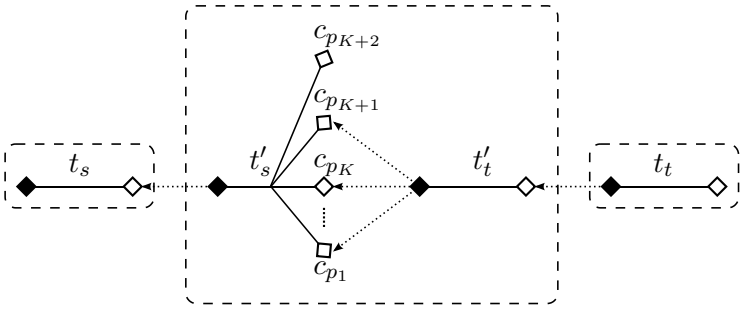


Figure 3.18: Task partition for modelling a path from t_t to t_s.

have a single effect and a disjunctive precondition of size two. Task t^l_t is has a single effect and a disjunctive precondition of size three. Tasks t^l_s and t^l_t are defined for vertices s and t respectively.

Note. Because we have K forbidden pairs in $\mathcal{F}_1$, we have K corresponding task partitions as well. This K is the measure $|C| \leq K$ of the ID problem.

We add the same condition dependencies to this task partition as for the other task partition: $c^{a_l}_e \precsim c^{b_l}_{p_1}, c^{b_l}_e \precsim c^{c_l}_p, c^{c_l}_e \precsim c^{d_l}_{p_1}, c^{d_l}_e \precsim c^{a_l}_p$. These ensure that not both arcs of a forbidden pair can be selected. Additionally, we add dependencies $c_e(t_a) \precsim c^{a_l}_p, c^{b_l}_e \precsim c_p(t_b), c_e(t_c) \precsim c^{c_l}_p, c^{d_l}_e \precsim c_p(t_d)$ to connect this task partition to the other tasks.

To have a coordinated instance correspond to the non-existence of a path from s to t, we add the condition dependencies $c_e(t_s) \precsim c^{s_l}_p, c^{s_l}_e \precsim c^{t_l}_{p_1}, c^{s_l}_e \precsim c^{t_l}_{p_2}, c^{t_l}_e \precsim c_p(t_t)$. If either $c^{s_l}_e \precsim c^{t_l}_{p_1}$ or $c^{s_l}_e \precsim c^{t_l}_{p_2}$ is instantiated, a path from t_s to t_t exists. If simultaneously $c^{a_l}_e \precsim c^{b_l}_{p_1}$ or $c^{c_l}_e \precsim c^{d_l}_{p_1}$ is selected, the instantiation for the partition is cyclic.[19]

To have a path from t_s to t_t correspond with an uncoordinated instance, we should have a path from t_t to t_s as well. Therefore, we create a task partition $T_{n+m+K+1} = \{t'_s, t'_t\}$. Task t'_t is a concrete task, while t'_s is an abstract task with $K+2$ preconditions. These tasks are connected to tasks t_s, t_t by $c_e(t_t) \precsim c^{t'}_p, c^{s'}_e \precsim c_p(t_s)$. Next, we add dependencies $\{c^{t'}_e \precsim c^{s'}_{p_1}, c^{t'}_e \precsim c^{s'}_{p_2}, \ldots, c^{t'}_e \precsim c^{s'}_{p_{K+1}}\}$. In Figure 3.18, this task partition is shown. Precondition $c^{s'}_{p_{K+2}}$ is independent of an effect.

We have to check for the transformation above if it results in a valid instance of ID. For ID, we require a *consistent* partitioned dependencies instance $(\mathbf{T}, \precsim)$ and an integer K. The integer K equals the number of forbidden pairs in $\mathcal{F}_1$. Consistency of the created instance is easily checked. The PWFP instance used for the $\exists\forall\neg$PWFP problem is acyclic. Arcs A are converted straight into dependencies or into one or more task partitions.

[19] Note the close relation between the set $\mathcal{X}_1$ and the set of conditions C to prune.

If all arcs would be transformed into dependencies, an acyclic instantiation exists in which precondition $c^{s'}_{p_{K+2}}$ is selected. In this case, the instance is therefore consistent. However, for some arcs, task partitions are constructed. For all abstract tasks, it is possible to select a precondition that does not depend on any effect. This implies that no cycle can pass through a task partition. Therefore, an instance for which forbidden pairs are transformed is consistent as well.

A yes-instance of ∃∀¬PWFP should correspond to a yes-instance of ID, under the transformation. A certificate for a yes-instance of ∃∀¬PWFP is the set $\mathcal{X}_1$. Each arc $a_i \in \mathcal{X}_1$ has a corresponding forbidden pair $p_i \in \mathcal{F}_1$, which has been transformed into a task partition T_{n+m+i}. So, when arc $a = (v_a, v_b) \in \mathcal{X}_1$ we will select dependency $c_e^{a_{n+m+i}} \precsim c_{p_1}^{b_{n+m+i}}$. Upon selecting this dependency, both $c_e^{a_{n+m+i}}$ and $c_{p_1}^{b_{n+m+i}}$ are selected. By propagation, precondition $c_{p_2}^{b_{n+m+i}}$ is therefore pruned; $c_{p_2}^{b_{n+m+i}} \in C$.

To see that the pruned conditions coordinate the instance, we have to show that no path $t_s - t_t$ exists. First, an almost trivial path could exist within each of the task partitions $\{T_{n+m+1}, T_{n+m+2}, \ldots, T_{n+m+K}\}$ for forbidden pairs $\mathcal{F}_1$. Instantiating $c_e^{s_l} \precsim c_{p_1}^{t_l}$ or $c_e^{s_l} \precsim c_{p_2}^{t_l}$ creates a path from t_s to t_t. However, *neither* of these dependencies can be instantiated because the dependency corresponding to the selected arc from $\mathcal{X}_1$ is selected. Selecting either of the dependencies that creates a path $t_s - t_t$ would infer a cycle local to the task partition.

In a similar way, the selected arcs $\mathcal{X}_2$ for the forbidden pairs $\mathcal{F}_2$ correspond to the selected dependencies in the task partition for the forbidden pair. Because none of the arcs in $\mathcal{X}_2$ can create a path from s to t, none of the corresponding dependencies can enable a path $t_s - t_t$ as well.

No path $s - t$ exists that contains only arcs that are not in a forbidden pair. Together with the fact that G is acyclic, we can conclude that the transformed instance for ID does not contain a path from t_s to t_t that passes through concrete tasks only.

So, by pruning all conditions in C, the dependencies instance $(T, \precsim)$ is coordinated. Because for each task partition $T \in \{T_{n+m+1}, T_{n+m+2}, \ldots, T_{n+m+K}\}$ exactly one dependency is selected and one precondition pruned, the size of C is exactly K. This K is *minimal*. If K would not be minimal, for at least one task partition $T_l \in \{T_{n+m+1}, T_{n+m+2}, \ldots, T_{n+m+K}\}$ no dependency is selected. The agent planning for $(T_l, \precsim_l)$ is free to select e.g. $c_e^{s_l} \precsim c_{p_1}^{t_l}$. So, the instance is therefore not coordinated. We conclude that the set C is a yes-certificate for ID.

A certificate for a yes-instance of ID is a set of conditions C to prune, with $|C| \leq K$. By pruning these conditions from the instance, all potential cyclic instantiations are prevented. We identify three types of task partitions that contain abstract tasks:[20]

[20] Conditions of concrete tasks cannot be pruned.

(i) task partitions $\{T_{n+m+1}, T_{n+m+2}, \ldots, T_{n+m+K}\}$, corresponding with forbidden pairs in $\mathcal{F}_1$,

(ii) task partitions $\{T_{n+1}, T_{n+2}, \ldots, T_{n+m}\}$, corresponding with forbidden pairs in $\mathcal{F}_2$,

(iii) task partition $\{t'_s, t'_t\}$, connecting t_t to t_s.

For the first set of task partitions, $\{T_{n+m+1}, T_{n+m+2}, \ldots, T_{n+m+K}\}$, an agent is free to select a dependency that connects t'_s to t'_t. However, this creates a direct path $s-t$. So, for coordination we should avoid the selection of such a dependency. With K task partitions, we cannot prune both $c^{t_l}_{p_1}$ and $c^{t_l}_{p_2}$ because it would require $2K$ conditions to be pruned. Therefore, either $c^{b_l}_{p_2}$ or $c^{d_l}_{p_2}$ should be pruned. As a result, dependency $c^{a_l}_e \precsim c^{b_l}_{p_1}$ or $c^{c_l}_e \precsim c^{d_l}_{p_1}$ will be selected respectively. This selection prevents the other dependencies within the partition to be selected. Otherwise, a cyclic instantiation is created for the partition.

For the task partitions $\{T_{n+1}, T_{n+2}, \ldots, T_{n+m}\}$, corresponding to $\mathcal{F}_2$, none of the task partitions can prevent the path that can be created for a forbidden pair in $\mathcal{F}_1$. For those partitions, at least K conditions have to be pruned. So, no conditions can be pruned from the task partitions $\{T_{n+1}, T_{n+2}, \ldots, T_{n+m}\}$.

Finally, we could break all cyclic instantiations by removing all preconditions of task t'_s that depend on $c^{t'}_e$. However, t'_s has $K+1$ preconditions, so removing them would prune more than K conditions.

Given C, we infer the set of condition dependencies to select. For each dependency $c^{u_i}_e \precsim c^{v_i}_{p_1}$ that is selected, the corresponding arc $(v_u, v_v) \in p_i$ will be in $\mathcal{X}_1$; $a_i = (v_u, v_v) \in \mathcal{X}_1$. For each of the task partitions $\{T_{n+m+1}, T_{n+m+2}, \ldots, T_{n+m+K}\}$, one dependency is selected. So, for each forbidden pair in $\mathcal{F}_1$ exactly one arc is in $\mathcal{X}_1$. Since C coordinates the instance, no dependency selection of the task partitions $\{T_{n+1}, T_{n+2}, \ldots, T_{n+m}\}$ can create a path from t_s to t_t. Neither does a path exist from t_s to t_t through concrete tasks only, because the instance is coordinated. Therefore, for each set $\mathcal{X}_2$ no path exists from s to t. So, $\mathcal{X}_1$ is a yes-certificate for $\exists\forall\neg$PWFP. □

PROPOSITION 3.10. *ID is Σ^p_2-complete.*

The Σ^p_2-completeness of ID follows directly from Propositions 3.8 and 3.9.

Again, even for a minimal dependencies instance, ID is Σ^p_2-complete. The proof is similar to the proof for Proposition 3.7.

PROPOSITION 3.11. *ID is Σ^p_2-complete for a* minimal *dependencies instance.*

Proof. All tasks $t \in T$ have a single effect. Again, we have to show that all preconditions $c_p \in \mathcal{C}_p(T)$ are in a feasible joint dependencies instantiation. For the task partitions, we can easily verify that each precondition can be instantiated in a feasible dependencies instantiation for it. When selecting $c^{s'}_{p_{K+2}}$, the joint dependencies instantiation is feasible if all instantiations for partitions are feasible.

Would another precondition of t'_s than $c^{s'}_{p_{K+2}}$ have been selected than no cyclic joint instantiation can be constructed. The chosen subset of forbidden pairs $\mathcal{X}_1 \subseteq \mathcal{F}_1$ prevents a path $s - t$. Hence, we could select any precondition that connects t'_t to t'_s. □

Optimal instantiated plan-decoupling between agents turns out to be computationally intractable. For practical uses, heuristics might pose a feasible alternative to compute sub-optimal decouplings. Despite being interesting, we will not discuss heuristics.

3.5 Discussion

In this chapter, we enriched the task coordination framework with *abstract tasks* that have preconditions and effects, and *condition dependencies* that relate preconditions and effects of tasks. These enrichments allowed us to reason about a *dependencies instance* and an *instantiation*. Within the framework, tasks with disjunctive effects and preconditions can modelled.

By partitioning the abstract complex task, we reasoned about coordination. With instantiated plan-decoupling, we proposed a *pre-planning coordination mechanism* which ensures coordination between agents for a given consistent partitioned abstract complex task. Our mechanism is a generalisation of plan-decoupling, the coordination mechanism discussed in Chapter 2.

Despite our extensions, the complexity results for IPCV and PCV are identical. Similarly, we obtained the same result for IPD as has been obtained for PD. While our enriched framework provides more modelling capabilities, it does *not harden* the coordination verification problem and the decoupling problem.

Even better, the enrichments in the enriched task coordination framework are far from trivial. By neglecting the precedence relation, we obtained the coNP-completeness result for ICV and Σ^p_2-completeness for ID. Hence, verifying and ensuring coordination for *abstract tasks and dependencies* is just as hard as verifying and ensuring coordination for *concrete tasks and precedences*.

Chapter 4

Application and Experiments

So far, we formally analysed instantiated plan-decoupling. In our introduction, we identified various application fields of autonomous agents. We would therefore like to apply our approach in a more real-life setting. By heuristically ensuring instantiated plan-coordination, we will conduct experiments by which we analyse how plan quality and planner run-time are affected by our approach.

In practise, our approach is used for achieving coordination. An agent is likely to accept the subtask resulting from coordination if it is beneficial to its planning process. Nevertheless, an agent's *planning freedom* is limited by ensuring coordination. Instantiated plan-coordination alters a partitioned complex task. Agents are limited in the feasible plans they can construct for their complex subtask. They have to give up some of their autonomy to ensure coordination for their planning processes. Giving up autonomy comes at a *cost*.[1]

In Steenhuisen et al. [33], the deterioration in agents' plan quality is called the *price of autonomy*. We could measure the price of autonomy by the number of additional precedences and pruned conditions. However, for an agent it is much more relevant as to what extend its feasible plans worsen. On the other hand, it is also important to identify with what ease a solution to its planning problem can be found.

An agent could plan for its coordinated subtask or have a central authority plan for it. When planning by itself, the subtask should be constrained by additional coordination constraints. If an agent perceives these additional costs to be higher than the costs for getting a plan from the central authority, it would be rational for agents to let go of their autonomy. In this chapter, we would like to empirically show how much costs are involved with decoupling. We focus on *instantiated plan-decoupling* and the role of *instantiation-decoupling* on its own. In work of Valk [35], the costs of *plan-decoupling* has been empirically tested.

[1] Costs can be measured in various ways. For us, costs are the degree to which plan quality worsens and planning run-time increases by ensuring coordination, compared to having no autonomy at all.

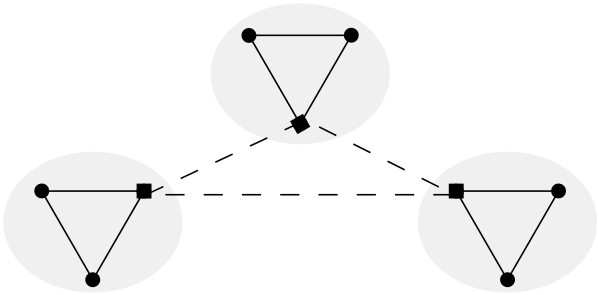


Figure 4.1: Infrastructure for a Logistics instance.

First, we will discuss the experimental design in Section 4.2. We continue with the experimental set-up in Section 4.2.4. Next, the empirical results for the experiments are in Section 4.3. Section 4.4 ends this chapter with a discussion.

4.1 Logistics Application Domain

Our goal is to test at what additional cost agents' autonomy comes. To compute these additional costs we should have a *reference*. Our reference will be a situation in which agents have *no* autonomy. In this situation, a central authority imposes a plan on them.

The application domain we will run our experiments for is the *Logistics* domain. Logistics is a domain that has been used in the International Planning Competitions (IPCs) of 1998 and 2000. It provides an intuitive structure and can be easily decoupled.

For Logistics problems, both a multi-agent system and a central authority are able to construct plans. We can *decouple* central Logistics instances into multi-agent planning instances. By decoupling central instances for a multi-agent planning approach, we can compare the performance of decoupled multi-agent planning to centralised planning. Decoupling allows for both centralised and decoupled planning to use the *same planners*.

4.1.1 Domain Specification

In Logistics, the goal is to transport a number of packages from one location to another. Locations are grouped into cities and only connected to each other within a city. Cities are connected to other cities by their airports. Airplanes flying between these airports transport packages. Within a city, trucks are available to transport packages from location to location.

EXAMPLE 4.1. In Figure 4.1, the infrastructure for a Logistics instance is shown. It has three cities, with three locations per city. A grey circle represents a city, dots

```
(:action LOAD-TRUCK
  :parameters   (?pkg - package ?truck - truck ?loc - place)
  :precondition (and (at ?truck ?loc) (at ?pkg ?loc))
  :effect       (and (not (at ?pkg ?loc)) (in ?pkg ?truck)))

(:action LOAD-AIRPLANE
  :parameters   (?pkg - package ?airplane - airplane ?loc - place)
  :precondition (and (at ?pkg ?loc) (at ?airplane ?loc))
  :effect       (and (not (at ?pkg ?loc)) (in ?pkg ?airplane)))

(:action UNLOAD-TRUCK
  :parameters   (?pkg - package ?truck - truck ?loc - place)
  :precondition (and (at ?truck ?loc) (in ?pkg ?truck))
  :effect       (and (not (in ?pkg ?truck)) (at ?pkg ?loc)))

(:action UNLOAD-AIRPLANE
  :parameters   (?pkg - package ?plane - airplane ?loc - place)
  :precondition (and (in ?pkg ?plane) (at ?plane ?loc))
  :effect       (and (not (in ?pkg ?plane)) (at ?pkg ?loc)))

(:action DRIVE-TRUCK
  :parameters   (?truck - truck ?from - place ?to - place ?city -
     city)
  :precondition (and (at ?truck ?from) (in-city ?from ?city) (in-
     city ?to ?city))
  :effect       (and (not (at ?truck ?from)) (at ?truck ?to)))

(:action FLY-AIRPLANE
  :parameters   (?plane - airplane ?from - airport ?to - airport)
  :precondition (at ?airplane ?from)
  :effect       (and (not (at ?plane ?from)) (at ?plane ?to)))
```

Listing 4.1: Operators for Logistics.

locations, and squares airports. The routes airplanes fly are dashed. Trucks drive the solid lines within a city. Packages reside at any of the locations.[2]

Before we define tasks, we first present PDDL-operators for Logistics in Listing 4.1.[3] Operators can be instantiated into actions. Agents planning for Logistics will plan with these actions, see Section 2.1.2. This gives insight in the way packages are transported. Packages can be loaded and unloaded from both trucks and airplanes. In between, they reside at a location. Trucks drive between locations within a city. Airplanes fly between the airports.

[2] An airport is a location as well.

[3] PDDL is the language used in the IPCs. It was initially developed by McDermott et al. [25].

```
(:init
  (in-city city0-l0 city0)
  (in-city city0-a0 city0)
  (in-city city0-a1 city0)
  (in-city city1-l0 city1)
  (in-city city1-a0 city1)
  (in-city city1-a1 city1)
  (at airplane0 city0-a0)
  (at truck0-0 city0-l0)
  (at truck1-0 city1-a0)
  (at package0 city0-a0)
)
(:goal
  (and
    (at package0 city1-l0)
  )
)
```

Listing 4.2: Initial and goal situation for a Logistics instance.

Operators are instantiated into actions during planning. A plan transforms an initial situation into a goal situation. In Listing 4.2, an example instance for Logistics is shown with two cities, two locations and one truck per city, one airplane, and one package. The goal is to have `package0` at location `city1-l0`.

Defining Tasks

To transport `package0` from `city0-a0` to `city1-l0`, for the problem instance of Listing 4.2, it should be transported by a plane to either airport `city1-a0` or `city1-a1` of `city1`. From one of these airports, it will be transported to `city1-l0` by truck. Hence, we can define two tasks for this package: one to transport it by plane and one to transport it by truck.

We define two types of tasks; tasks to transport a package by truck and tasks to transport by airplane. For trucks, we define the following *abstract* task:[4]

$$
\begin{aligned}
t^t &= (C_p^t, C_e^t), \text{ with for each } c_p^t \in C_p^t \text{ and } c_e^t \in C_e^t\text{:} \\
c_p^t &= (\gamma_{c_p}, \texttt{at(?package, ?location)}) \\
c_e^t &= (\gamma_{c_e}, \texttt{at(?package, ?location)}).
\end{aligned}
$$

Airplane tasks are identically modelled, with the exception that they are defined

[4]We used the PDDL notation to denote variables and will surround the arguments of atoms by parenthesis.

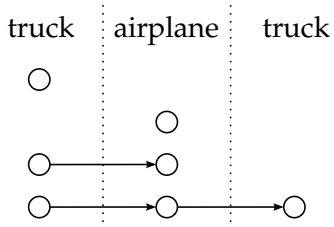


Figure 4.2: Orders for a Logistics instance.

for airports instead of locations:

$$t^a = (C_p^a, C_e^a), \text{ with for each } c_p^a \in C_p^a \text{ and } c_e^a \in C_e^a\text{:}$$
$$c_p^a = (\gamma_{c_p}, \texttt{at(?package, ?airport)})$$
$$c_e^a = (\gamma_{c_e}, \texttt{at(?package, ?airport)}).$$

To transport a package, one to three tasks are required.[5] These tasks are chained one after another. We call this task chain an *order*. Orders can be either *intra-city*, or *inter-city*. Intra-city orders are defined for packages that have their goal location in the same city as their origin. An intra-city order is modelled by a single task. We define intra-city orders for packages that have a goal location in another city than the city it originates from. These orders consist of one, two, or three tasks. In Figure 4.2, a couple of orders for Logistics are conceptualised.

Abstract Complex Task

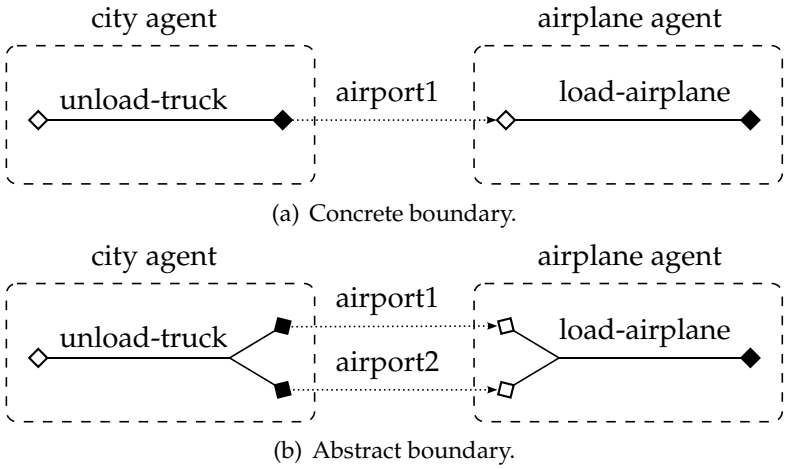
In Figure 4.2, we ordered tasks by a precedence relation. In fact, the tasks depend on each other by literals like `at(package0, city0-a0)`. Hence, we will model them as *condition dependencies*. We do not need the precedence relation to model Logistics problems.[6]

Given tasks and condition dependencies for Logistics, we can construct an *abstract complex task*. While it lacks precedence relations, it is a dependencies instance as well. In Example 4.2, we will discuss an example.

EXAMPLE 4.2. Suppose we have an inter-city order for a Logistics instance with two cities, one airport per city, and one additional location per city. Package `package0` has to be transported from location `city0-l0` to `city1-l0`. For this order, we define three concrete tasks:

$$t_1 = (\{(\gamma_1, \texttt{at(package0, city0-l0)})\}, \{(\gamma_2, \texttt{at(package0, city0-a0)})\})$$
$$t_2 = (\{(\gamma_3, \texttt{at(package0, city0-a0)})\}, \{(\gamma_4, \texttt{at(package0, city1-a0)})\})$$
$$t_3 = (\{(\gamma_5, \texttt{at(package0, city1-a0)})\}, \{(\gamma_6, \texttt{at(package0, city1-l0)})\}).$$

[5] If a package's initial location equals its goal location, we do not define a task for it.
[6] We *will* require precedences when decoupling, as we will see later on.



(a) Concrete boundary.

(b) Abstract boundary.

Figure 4.3: Boundaries between a city and airplane agent.

Additionally, we define the condition dependencies $c_e(t_1) \precsim c_p(t_2)$ and $c_e(t_2) \precsim c_p(t_3)$. This defines the abstract complex task $(\{t_1, t_2, t_3\}, \emptyset, \precsim)$ for this Logistics instance.

Defining Agents

One might expect us to define each truck and airplane to be an agent. However, we will define agents based on trucks and airplanes on geographic location. For each *city* we define single city agent. Additionally, we define a single *airplane* agent that reasons about all airplane tasks.

Multiple Airports per City

When dealing with concrete tasks, we have a *concrete* boundary between agents. In Figure 4.3(a), this situation is shown.

To test our instantiated plan-coordination approach, we should have disjunctive conditions at the boundary of agents. It should be unclear for the airplane agent at which airport packages must be unloaded by the trucks. We should have an *abstract boundary*. Therefore, we will need instances that have *more than one airport* per city. Figure 4.3(b) shows an example of an abstract boundary for Logistics.

All instances used in the planning competitions of 1998[7] and 2000[8] have a single airport per city. For the problem set of 2000, each city has only a single location and a single truck. Since we cannot use these instances to test our dependencies coordination mechanism, we will generate *custom* Logistics instances with several airports per city.

[7] `ftp://ftp.cs.yale.edu/pub/mcdermott/aipscomp-results.html`
[8] `http://www.cs.toronto.edu/aips2000/`

In the work of Valk [35], decoupling has been performed on standard Logistics instances. While his approach is based on plan-decoupling, it cannot handle tasks with disjunctive conditions. Hence, with instantiated plan-decoupling, we *enlarge* the set of Logistics instances we can ensure pre-planning coordination for.

4.2 Experimental Design and Set-up

In the previous section, we already identified the need for Logistics instances with multiple airports per city to test our coordination approach. In this section, we will further analyse the Logistics domain and discuss which instances will be used in our experiments.

4.2.1 Decoupling

To achieve instantiated plan-decoupling, we decouple with respect to both precedences and condition dependencies. In Chapter 3, we did not identify how to coordinate for both relations simultaneously. Here, we will *divide* instantiated plan-decoupling into a *plan-decoupling phase* and an *instantiation-decoupling phase*. We will show that for Logistics, the relative order of these phases is *irrelevant*.

For each package p_i to be transported, we can define an *order* o_i. An order is the sequence of tasks that is required for transporting p_i from its initial to its goal location: $o = \langle t_1, t_2, \ldots, t_n \rangle$. Each order o_i is *independent* of another order o_j. By this, we mean that none of the tasks in o_i depend on a task in o_j and vice versa. *Within* an order, for all preconditions of tasks $t_2, \ldots, t_n$, i.e. all tasks except t_1, depend on the effects of the preceding task. In Figure 4.3(b), as an example, the dependencies between unloading a package from a truck and loading it into an airplane are shown.

An order can be characterised as either an *intra-city* or *inter-city* order. Packages that have their initial and goal location in the same city have a corresponding intra-city order. Packages leaving the city have an inter-city order.

PROPOSITION 4.1. *Given a abstract complex task $(T, \emptyset, \precsim)$ for Logistics, each instantiated plan $(T', \emptyset, \precsim')$ for it transforms into the same plan $(T^c, \prec)$ by applying Algorithm 3.2.*

Proof. Assume the contrary: there exist two different instantiated plans $(T'^i, \emptyset, \precsim'^i)$ and $(T'^j, \emptyset, \precsim'^j)$ for the abstract complex task $(T, \emptyset, \precsim)$. Each of these instantiated plans yields a different plan by applying Algorithm 3.2. These plans are respectively $(T^{ci}, \prec^i)$ and $(T^{cj}, \prec^j)$.

The set T^{ci} and T^{cj} will contain the same tasks: $T^{ci} = T^{cj}$. This is because for each task $t'_i \in T'$, Algorithm 3.2 constructs a corresponding task $t^c_i \in T^c$.[9] So, the sets of precedences between tasks $\prec^i$ and $\prec^j$ should be unequal. Without loss of

[9] Note that an instantiation for an instance has a concrete task for *each* task in the instance. Task $t_i \in T$ is therefore instantiated into t'_i.

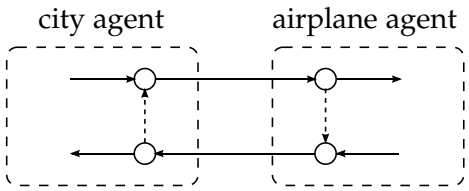


Figure 4.4: A potential cyclic plan for Logistics.

generality, we can assume that $\prec^i$ contains a precedence $t_k \prec t_l$ that is *not* in $\prec^j$. This implies that there exists a dependency between an effect of t_k and a precondition of task t_l: $c_e^k \precsim c_p^l$. In T^{cj}, a different precondition has to be selected for task t_l. However, we observed that each task, except the first, in an order depends on its predecessor *only* and has *no* preconditions that do not depend on an effect. Therefore, $t_l \in T^{cj}$ should have a dependency on $t_k \in T^{cj}$ as well and we have our contradiction. □

By Proposition 4.1, we can conclude that there is a direct relation between precedences and dependencies in Logistics. Given the dependencies, all precedences can be determined. The other way around, given the precedences between tasks, we can easily determine the dependencies between conditions. It allows us to *interchange* the phases of plan- and instantiation-decoupling. Precedences in a coordination set can be trivially converted into condition dependencies.

Plan-Decoupling

The primary goal of our experiments is to test the effect of instantiation-decoupling. Still, we have to ensure plan-coordination as well. Agents can come up with local plans that, when merged, result in a *cyclic* joint plan. In Figure 4.4, a potential cycle is shown for two agents.

For Logistics, we can plan-decouple heuristically using a *depth-partitioning algorithm* [36]. Here, we will shortly address how plan-decoupling can be achieved for Logistics.

In short, for Logistics, plan-decoupling can be achieved by partitioning the set of tasks assigned to a *city* agent into three disjoint subsets. Each order is from the local perspective of an agent either *incoming*, *outgoing*, or *local*. All tasks for packages that reside in the city and have to leave it are in the outgoing set. Tasks for packages with their goal location in the city and their initial location outside are in the incoming set. All remaining tasks that involve packages that reside within the city are in the local set.

Remark. To fully test the performance of coordination, all Logistics instances are generated *without* local orders. Each package will have to be transported from a location in one city, to a location in another city.

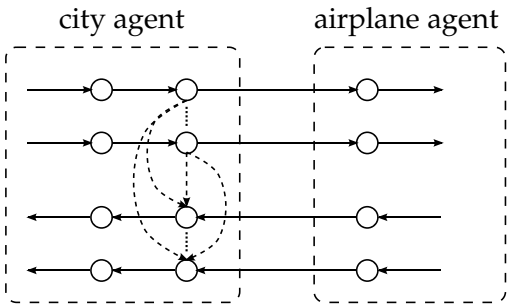


Figure 4.5: Additional (dashed) precedences for coordinating Logistics.

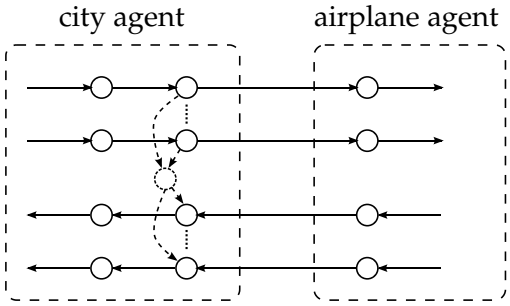


Figure 4.6: (Dashed) Dummy task for coordinating Logistics.

We will achieve plan-decoupling for Logistics instances in either of two ways:

- we add precedences from the outgoing orders to the incoming orders,
- we split each city c into two disjunctive cities c_i, c_o; all tasks of outgoing orders are defined for c_o, tasks of incoming orders are defined for c_i.

Add Precedences The first approach is identical to the one used by Steenhuisen et al. [33]. We require all outgoing orders to be completed before any incoming order is processed. When we require this for each agent, no cyclic joint instantiated plan can be constructed. In Figure 4.5, the additional precedences from outgoing orders to incoming orders are shown.

While many precedences might be added between tasks, which are hard to model in PDDL, we will use a more elegant solution in our experiments. We define an additional dummy task. This dummy task corresponds to a literal that needs to be achieved in PDDL. The use of the additional task is illustrated in Figure 4.6.

Splitting Cities Given a Logistics problem instance, we can define a new instance in which each city is split into two new cities: one with only outgoing orders

and one with only incoming orders.[10] In the newly obtained problem instance, all outgoing orders for the original city have been redefined for the 'outgoing' city. Incoming orders for the original city are redefined for the 'incoming' city.

The agent planning for the original city is split into two agents as well. Hence, we have one agent planning for incoming order, the other for outgoing orders. By splitting the orders over two cities, we *prevent* the construction of a cyclic joint instantiated plan.

Adding precedences to achieve plan-decoupling requires a planner to 'plan' for the additional precedences. By splitting cities, we modify the Logistics instances such that they are inherently plan-coordinated. No additional precedences are required to ensure plan-coordination.

Instantiation-Decoupling

To achieve instantiation-decoupling, we have to make sure that all feasible instantiations for the agents' instances can be merged into a *feasible* joint instantiation. Such an instantiation is feasible if it is both *correct* and *acyclic*. Because a Logistics instance is always acyclic, its instantiations will be acyclic as well. Consequently, for instantiation-decoupling we have to ensure that each instantiation yields a *correct* joint instantiation.

After planning, each agent has constructed a feasible instantiation for its tasks. When joining these instantiations, only the inter-agent condition dependencies can obstruct a feasible joint instantiation.

Each inter-agent dependency is related to the condition of having package p_i at airport ap_j. If a city has multiple airports, multiple dependencies between the dependent tasks are required. Package p_i could e.g. reside at ap_j, ap_k, or ap_l. By selecting one of these airports as the intermediate location for p_i, we have instantiation-coordinated these dependencies. In our experiments, we will achieve instantiation-decoupling by defining *intermediate* airport locations for packages.

PROPOSITION 4.2. *Given a partitioned dependencies instance* $(\mathbf{T}, \precsim)$ *for Logistics, by selecting a single airport for each* inter-city *order as intermediate airport results in a* minimum *number of conditions to be pruned.*

Proof. First, note that each order has to be instantiation-coordinated. For an inter-city order, it is required to coordinate the airports at which the package will reside. We coordinate by selecting a single inter-agent dependency $c^i_{e_k} \precsim c^j_{p_l}$. Let us call the tasks that are related to the dependency t_i and t_j. By propagation, selecting the dependency results in pruning all preconditions $C_p(t_j)$ except for $c^j_{p_l}$. One precondition remains, that has to be achieved, so all other effects $C_e(t_i)$ except $c^i_{e_k}$ have to be pruned as well.

[10] We will not use instances with local orders.

Finally, we note that orders are independent of each other. Consequently, selecting a dependency does not propagate further than the two tasks involved in the dependency. Therefore, it is essential to prune all these conditions. □

In the Logistics instances used in the experiments, the selection of intermediate airports is performed in two ways. In one configuration, we randomly selected *one airport per city* to be the intermediate airport for all outgoing and incoming orders. Essentially, it reduces the generated instances to instances with a single airport per city. In the second configuration, we randomly selected a *single airport per order per city* as intermediate airport. So, an inter-city order has two randomly assigned intermediate airports.

Generated Problem Instances

Based on the discussions on achieving plan- and instantiation-decoupling, we identify three axes over which instances could be generated. These are:

- *central* problem or *decoupled* problem,
- *split* cities or *original* cities,
- single airport *per city* or single airport *per order*.

While being different instances, we define several instances for a single Logistics problem. In Table 4.1, the characteristics of the Logistics problems used in the experiments are shown.

The identifier for a problem is in the first column of Table 4.1. Next, the number of cities in the problem is shown. The next two columns list the number of locations per city and the number of which are airports. Then, in column five, the number of trucks per city is shown. The number of airplanes is for the entire problem, not per city. Finally, the number of packages scales with the problem id x by: $\lfloor (4+x) \cdot (3+x)/12 \rfloor$.

4.2.2 Performance Measures

In the introduction of this chapter, we shed some light on how to determine the price of autonomy. We identified two measures: the *quality* of a plan and the *ease* with which such a plan can be constructed. Both of these terms are ambiguous. Therefore, we will discuss here what will be the exact measures for determining the performance of the planners.

Plan Quality

In classical planning, the quality of a plan is measured by the number of actions in that plan. During IPC-2000, e.g., quality was measured by the number of plan steps [1], the number of actions. A *shorter* plan is of *higher* quality. Later competitions featured more advanced features like temporal and numerical requirements,

Problem	Cities	Locations	Airports	Trucks	Airplanes	Packages
1	1	2	1	1	1	1
2	1	2	1	1	1	2
3	1	3	1	1	1	3
4	2	3	1	1	1	4
5	2	3	1	1	1	6
6	2	5	2	2	1	7
7	2	5	2	2	1	9
8	3	5	2	2	2	11
9	3	6	2	2	2	13
10	3	6	2	2	2	15
11	3	6	2	2	2	17
12	4	8	3	3	2	20
13	4	8	3	3	2	22
14	4	8	3	3	2	25
15	4	9	3	3	2	28
16	5	9	3	3	3	31
17	5	9	3	3	3	35
18	5	11	4	4	3	38
19	5	11	4	4	3	42
20	6	11	4	4	3	46
21	6	12	4	4	3	50
22	6	12	4	4	3	54
23	6	12	4	4	3	58
24	7	14	5	5	4	63
25	7	14	5	5	4	67
26	7	14	5	5	4	72
27	7	15	5	5	4	77
28	8	15	5	5	4	82
29	8	15	5	5	4	88
30	8	17	6	6	4	93

Table 4.1: Characteristics of the Logistics problems.

enabled by more recent versions of PDDL [14]. In our experiments, plan quality is measured in terms of the *number of actions* in a plan. This is the absolute number of actions in a plan, *not* the make-span of a plan.[11]

An IPC features several tracks in which planners can compete. We will focus here on the *satisficing* track. In this track, the goal is to find a near to optimal so-

[11]The make-span of a plan is the 'time' required for executing it. By executing actions concurrently, the make-span can be reduced.

lution. This allows us to use larger problem instances then when using *optimal* planners.

Run-time

Is it always better to have a shorter plan? If computing a slightly shorter plan takes a vast amount more time to compute, we might doubt this. Optimal planners e.g. use a lot of run-time to compute the best plan there is. Nevertheless, we choose to use sub-optimal planners for their larger applicability. Therefore, we will take *run-time* of planners to be our second measure. While there is clearly a trade-off between run-time and plan quality, we will and can not indicate how much a difference in quality is worth in terms of a difference in run-time.

4.2.3 Expected Results

Now we know what instances to test and what to measure, we can come up with some hypotheses. For each of them, we give a line of reasoning. We do not claim these hypotheses to be correct, but will later on evaluate them with respect to the achieved results.

HYPOTHESIS 4.1 (Plan Quality)**.** Given a Logistics problem, the total plan length for the decoupled instances will exceed the plan length for the central instance.

When planning for a central instance, a planner can come up with the *optimal* solution for this Logistics problem. After decoupling, the instances are instantiated plan-decoupled. This involves choosing intermediate airports and ordering the incoming and outgoing orders. Without knowledge of the planning problem, we cannot determine which airport to select for optimality. The decoupling heuristics are therefore likely to reduce the problem to an instance for which only sub-optimal plans exist.

On the other hand, by decoupling, the size of the subproblems is smaller than the problem itself. As a result, planners will have to search a smaller search space. Therefore, they should be able to find relatively good solutions for subproblems.

HYPOTHESIS 4.2 (Run-time)**.** Given a Logistics problem, the total run-time for the decoupled instances will be less than the run-time for the central instance.

Despite classical planning being harder, *plan-existence* for Logistics is in P [17]. *Plan-optimality*, in terms of plan-length, is NP-complete [17]. Domain-independent planners, however, cannot exploit domain-knowledge. They often search a state space. By decoupling, these state spaces are made considerably smaller. As a result, finding a solution should be easier and take less time.

HYPOTHESIS 4.3 (Airport per City and per Order)**.** Given a Logistics problem, the instantiation-decoupled instances with a random single airport per city will have a shorter plan length and require less run-time than the instances with a random single airport per order.

To start with the run-time aspect; if we remove airports from cities, the subproblems for city agents become smaller. Moreover, the largest reduction in subproblem size can be attributed to the airplane problem. Hence, we would expect run-time for instances with a single airport per city to be shorter than for instances with a single airport per order. While in principle it might be arbitrary to select a single airport for all orders, or an airport per order, the reduction in size of the city subproblems might raise plans of better quality.

4.2.4 Experimental Set-up

Before we can continue to with the empirical results, we first have to identify our experimental set-up. What domain-independent *planners* will be used and what restrictions do we impose on them during testing?

Planners Used

We selected five planners out of the many domain-independent planners for classical planning problems. These are: FF, LAMA, LPG, SGPlan and YAHSP. We selected these planners because they are either overall top-performers, very influential, or perform very well on Logistics instances. Next, we will shortly discuss each of these planners.

FF 2.3 FF [19] stands for 'Fast-Forward', which relates to the forward heuristic search it performs. FF has been one of the most influential planners for classical planning. It is based on the ground-breaking planner Graphplan [3]. In 2000 and 2002 it performed in its original form at the IPCs. Later editions featured planners based on ideas of FF or even FF itself. It won the 2000-competition and performed very well on Logistics.

LAMA LAMA [29] is the winner of the most recent planning competition of 2008 in the sequential satisficing track. It is a heuristic search planner, which combines the FF-heuristic with a heuristic based on landmarks. Given the fact that Logistics contains many landmarks, we included this planner here.[12]

LPG-td-1.0 LPG [15] competed in the 2002 and 2004 competition. It won the first and achieved a second place in 2004. LPG uses heuristics for guiding its local search for a plan. It provides good performance for Logistics instances.

To get the best possible plan, LPG can optimise for plan quality. On the other hand, it can optimise for speed as well. Either of these optimisation criteria can be given at the command line of LPG. In our results, we ran LPG for both quality as speed.

[12] A landmark is a fact that has to hold in each plan for a planning problem.

SGPlan-6 SGPlan-6 [22] competed in the sequential satisficing track of IPC-2008. Previous versions took part in the competitions of 2004 [5] and 2006. SGPlan won the 2006-competition and reached a second place at the 2004 edition. It partially decomposes a planning problem to solve each subproblem by another planner.

YAHSP 1.1 YAHSP [37] competed only in IPC-2004. It achieved the second price in the suboptimal track. YAHSP turns out to be a very fast planner for Logistics, as we will see next. This is why we included it in our experiments.

4.2.5 Computational Limits

Each of the planners was bound to two computational limits. We limited the maximum amount of memory to 2GB and the run-time to 30 minutes. These limits are in line with the computational limits of the most recent IPC in 2008. We ensured that each planner got a dedicated processor and 2GB of memory during its operation such that no other processes would interfere with it.

4.3 Planning Results

We ran each of the planners of the previous section on the generated Logistics instances. In this section, we will discuss the main results that can be obtained from these experiments. Both of the performance measures, plan quality and run-time, will be discussed.

First, we will discuss how plan quality and run-time of the planners are affected by decoupling. Next, we will study instantiation-decoupling in isolation by testing the instances with split cities. Finally, we will identify which heuristic for instantiation-decoupling gives better results. We compare randomly selecting a single airport for all orders within a city with randomly selecting airports per order.

4.3.1 Instantiated Plan-Decoupling

We ran each of the planners of Section 4.2.4 on both central and decoupled instances corresponding to the problems of Table 4.1. The cities of these instances are *not* split. In the decoupled instances, a random intermediate airport *per order* is selected to achieve instantiation-decoupling and a *dummy task* is added to achieve plan-decoupling.

Plan Quality

In Figure 4.7, *plan quality* is shown for the *decoupled* instances. All planners, except LPG, were able to solve *all instances*. Some planners perform better than others, but in general they perform alike. For instances corresponding to problems 27, 29, and 30, LPG was unable to solve one city subproblem as it ran out of memory.

Figure 4.8 shows the plan quality *relative* to the central instances. It shows the ratio between the number of actions required for the decoupled instances and the central instances: $\frac{\text{decoupled}}{\text{central}}$. Note that for the larger problem instances, many data points are omitted. These correspond to planners that were not able to construct a plan within the computational limits.

Only two of the tested planners, SGPlan and YAHSP, were able to solve all central instances. Both runs of LPG-td, for quality and speed, ran out of memory for instances 24 till 27. For the largest three problems, the number of facts in the instance was too large for them to handle. LAMA, top-performer in IPC-2008, ran out of memory for the central instances related to problems 18 and higher. For problems 14 and 15 it ran out of memory as well. FF could not plan for instance 24 and up because it required more than half an hour of CPU time.

In Figure 4.8, we see that plan quality is not affected much by instantiated plan-decoupling. Quality ranges between approximately 15% longer and 15% shorter plans. For larger instances, the margin narrows to around 10%. Therefore, autonomy of agents does not lead to a significant higher price to be paid in terms of plan quality. Moreover, it allows for larger instances to be solved as well.

Run-time

Plan quality is hardly affected by instantiated plan-decoupling. Now, we will see what the effect of instantiated plan-decoupling is for our second performance measure: *run-time*. In Figure 4.9, the run-time of the planners for the *decoupled* instances is shown.[13] As discussed earlier, YAHSP is an excellent performer on the Logistics domain. The largest decoupled instance took just over three seconds to plan for. The data points missing in the plot correspond to the data points that lack in Figure 4.7.

In Figure 4.10, we plot the ratio between the run-time required for the decoupled instances and the central instances. As we see, especially FF benefits from decoupling these instances. For larger problems, the speed-up is over a factor one hundred. For the other planners, the speed-up is more modest.

YAHSP benefits the least from decoupling. The type of heuristic search it performs might explain why. It uses a look-ahead strategy, combined with a best-first search [37]. In Logistics, this appears to be very beneficial. Mainly because Logistics problems only have reversible actions [40]. Hence, a planning process cannot run into dead-ends in the search space.

Remark. The run-time of LPG-td speed and SGPlan on the decoupled instances is very similar. In Chen et al. [5], LPG is explicitly mentioned as one of the base planners used in SGPlan. SGPlan pre-processes a planning instance and subsequently feeds it to one of its base planners. Curiously, it *is* able to plan for problem instances 27, 29, and 30. This is likely due to the fact that it uses a version of LPG that is compiled with a larger upper bound for the number of facts it can process.

[13] Note that the vertical scale is a *log* scale.

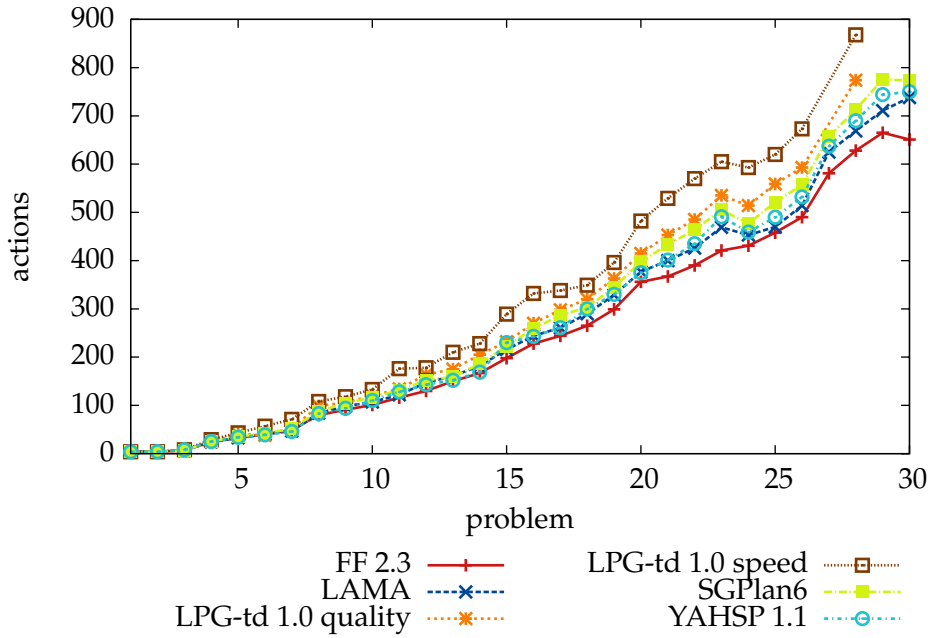


Figure 4.7: Plan quality - decoupled planning.

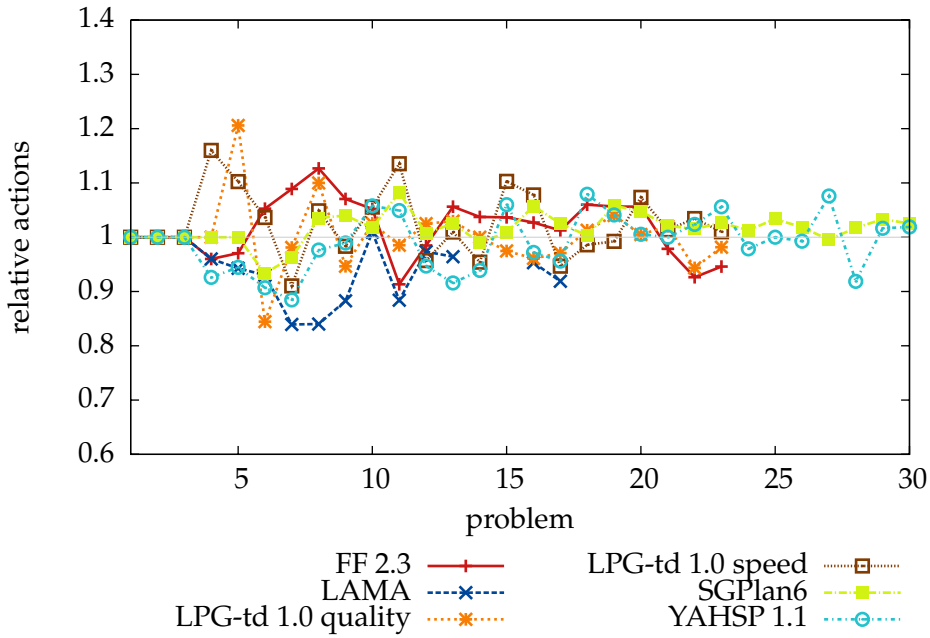


Figure 4.8: Relative plan quality (decoupled / central).

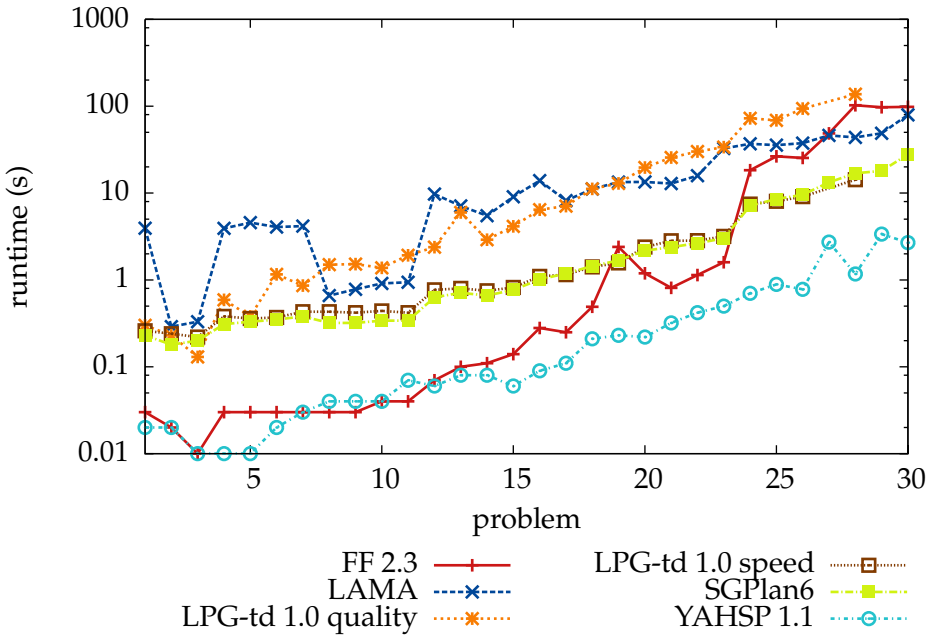


Figure 4.9: Run-time - decoupled planning.

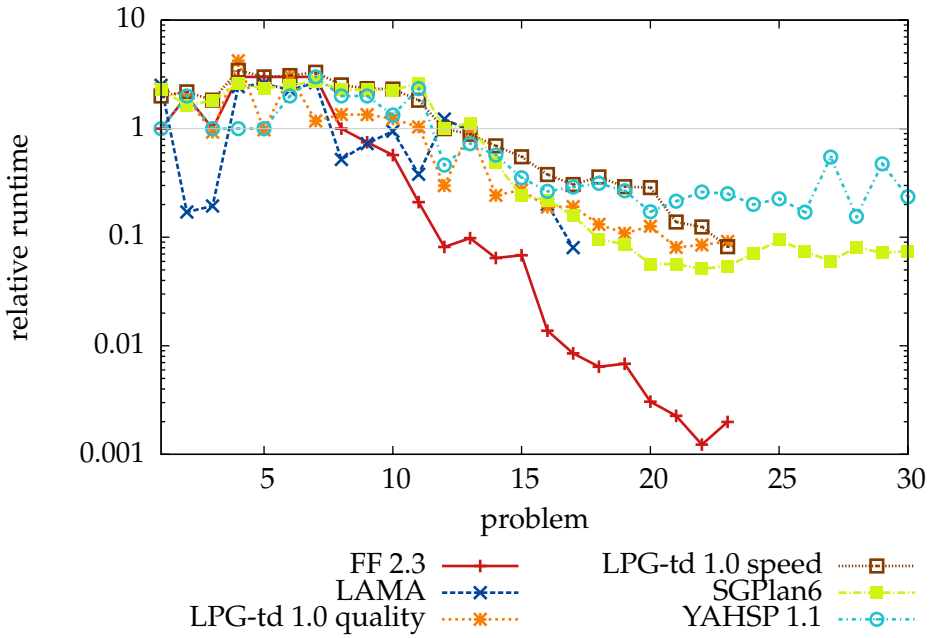


Figure 4.10: Relative run-time (decoupled / central).

4.3.2 Instantiation-Decoupling

Instantiated plan-decoupling does not impose a heavy burden on either plan quality or run-time. In terms of run-time, decoupling is even beneficial to the planning process. However, in the previously discussed experiments, we combined instantiation-decoupling and plan-decoupling. We would like to know what influence our extension, *instantiation-decoupling*, has on both plan quality and run-time.

We achieve instantiation-decoupling by selecting a random intermediate airport *per order*. Instead of using plan-decoupling to achieve plan-coordination, we *split* each city of a planning problem into a city with incoming and a city with outgoing orders. In this manner, we ensure the planning instances to be plan-coordinated. To allow for a fair comparison, both *central and decoupled* instances have split cities.

Plan Quality

By splitting each city into two, the size of the instances increases. We would therefore expect planners to have more difficulty with planning for the central instances.

In Figure 4.11, the ratio between *plan quality* of *instantiation-decoupled* split instances and of *central* split instances is shown. Most planners' plan quality improves by instantiation-decoupling. YAHSP benefits the most from instantiation-decoupling, especially compared to the relative plan quality of non-split instances, shown in Figure 4.8.

As expected, planners solved fewer instances. YAHSP was the only planner capable of solving problem 30. LPG ran faster into its maximum number of facts, because of the additional cities. All other planners reached their time or memory limit at a smaller problem instance.

FF is an exception on plan quality and the number of instances solved. It is the only planner that suffered from instantiation-decoupling. Moreover, compare Figure 4.11 to Figure 4.8. FF is the only planner capable of solving significantly more instances with split cities than without split cities. We have no idea as to why FF exhibits this behaviour.

Run-time

In Figure 4.12, the *run-time* ratio between *decoupled* split instances and *central* split instances is shown. Data points not in this figure correspond to the points not in Figure 4.11. Overall, the improvements in run-time are in line with the improvements shown in Figure 4.10 for the non-split instances.

Again, FF performs differently. Between problem instances 15 and 23, similar relative speed-ups are achieved. For larger instances however, FF's relative run-time decreases. Compared to its run-time for non-split instances, see Figure 4.10, it is less aided by decoupling. Nevertheless, for most instances, it achieves a greater speed-up than the other planners.

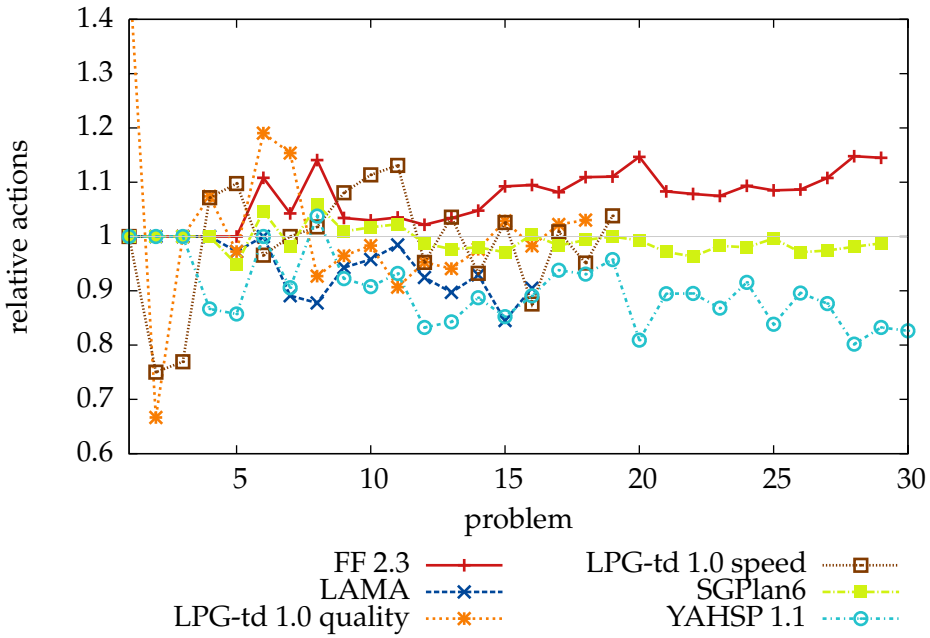


Figure 4.11: Relative plan quality for split instances (decoupled / central).

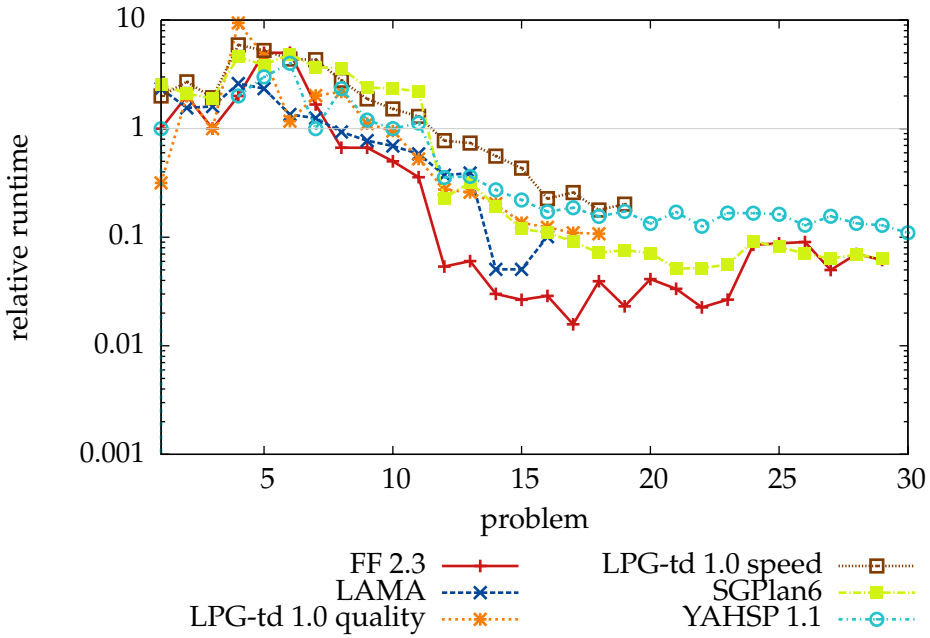


Figure 4.12: Relative run-time for split instances (decoupled / central).

Intermediate Airports for Orders

In Section 4.2.1, we identified two ways to perform instantiation-decoupling. So far, the results presented have been achieved by *randomly* selecting intermediate airports on a *per-order* basis. Here, we will compare this method to the selection of a single airport *per city*. We only vary the instantiation-decoupling method, but we ensure plan-coordination as well. Hence, the cities are *not split*.

In Figure 4.13, the *plan quality* ratio between an instantiation-decoupling *per city* and *per order* is shown: $\frac{\text{per city}}{\text{per order}}$. Almost all data points lie above the line $\frac{\text{per city}}{\text{per order}} = 1$. For most of the Logistics planning problems, ensuring instantiation-decoupling by randomly selecting an airport per-order leads to shorter plans than selecting a single airport per city.

LPG is the only planner that is unable to construct a plan for some instances, as we already saw in Figure 4.7. This explains why no data points for LPG speed and LPG quality appear for instances 27, 29, and 30.

In terms of *run-time*, it is better to coordinate by selecting a single airport per city. This follows from Figure 4.14. In this figure, we observe that for larger instances, it becomes more and more beneficial to do so. Only LAMA already benefits for small problems from selecting a single airport per city instead of one per order.

4.4 Discussion

Having discussed the results of our experiments, we will now evaluate Hypotheses 4.1, 4.2, and 4.3. After that, we will discuss some additional findings from our experimental work.

In Hypothesis 4.1, we suggested that the total plan length for decoupled instances would exceed the plan length for central instances. In general, we have *no* evidence that supports this hypothesis. On average, *plan quality* for decoupled instances is comparable to the quality of central instances. For instances with split cities, results vary more. Some planners benefit a little from instantiation-decoupling, while others perform worse after instantiation-decoupling.

Planning for larger instances did benefit from decoupling, in terms of *run-time*. While smaller problems have a run-time in the order of milliseconds. Hence, these measurements are not representative for the performance of instantiated plan-decoupling and instantiation-decoupling. For large planning instances, we a clear improvement in run-time compared to the planning for the central instances. Hence, we have evidence that supports Hypothesis 4.2.

Run-time decreases by decoupling, because decoupled instances are smaller in size than their central equivalents. They contain less objects and literals. Conse-

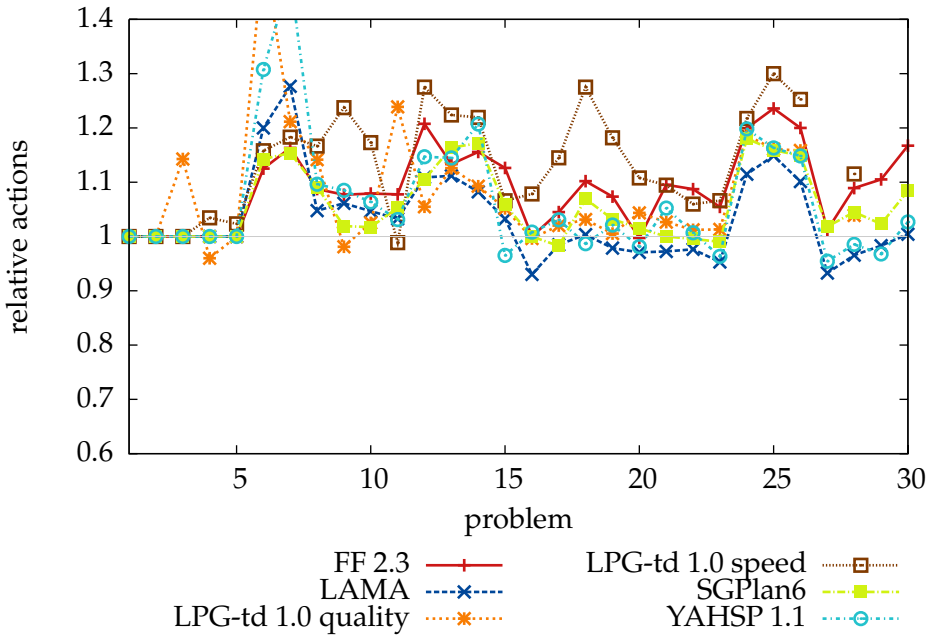


Figure 4.13: Relative plan quality, instantiated plan-decoupled (city / order).

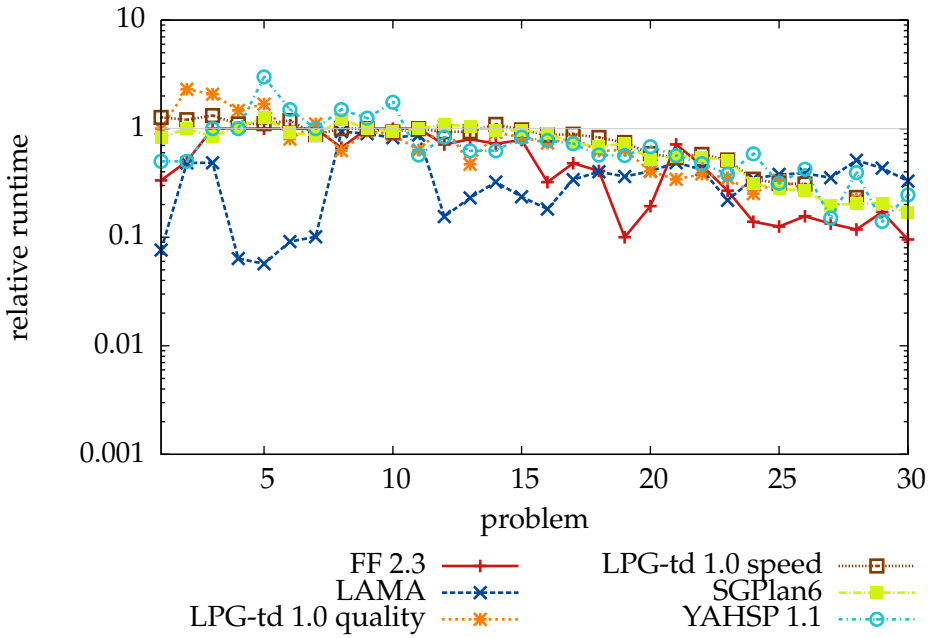


Figure 4.14: Relative run-time, instantiated plan-decoupled (city / order).

quently, the search space for planners becomes considerably smaller and it is easier for a planner to find a plan.

Finally, we have Hypothesis 4.3. It is two-fold; we suggested that achieving instantiation-decoupling by randomly selecting a single airport per city results in a better plan quality and run-time compared to randomly selecting an airport per order. We conclude that plan quality benefits from selecting airports per order. Run-time, on the other hand, benefits from selecting airports per city. Hence, we have no conclusive evidence for Hypothesis 4.3.

Plan quality seems to suffer from carrying all packages through a single airport per city. The explanation for this is simple. Some packages initially reside at an airport. After coordination, it is likely that these packages have to be transported to another airport. By coordinating on a per-order basis, these packages already are at their intermediate airport. We might explain the behaviour of *run-time* as well. By selecting a single airport per city, the decoupled airplane subproblem becomes less complex. Per city, only a single airport has to be considered.

We conclude that the *price of autonomy* agents have to incur for their autonomy is minimal. In terms of run-time, it is *beneficial* to decouple Logistics instances. While plan quality is hardly affected, especially run-time of the planners for Logistics improves by instantiated plan-decoupling. With the experiments discussed in this chapter, we have shown the applicability of instantiation-decoupling and the advantages it potentially has.

Moreover, more planning instances could be solved by using instantiated plan-decoupling. Subinstances for Logistics are smaller than the original instance and therefore more tractable to plan for. Hence, decoupling allowed us to plan for *larger instances* than a central planning process would.

One might wonder why plan quality is hardly affected. Adding precedences and pruning conditions could only worsen plan quality. However, we conducted our experiments with *satisficing* planners. These planners construct sub-optimal plans. By constructing a decoupling, planners were able to construct comparatively better plans for subproblems. This improvement covers to a large extend the burden decoupling puts on it.

Chapter 5

Conclusions and Future Work

The central research question for our work is how to ensure coordination of autonomous, self-interested planning agents for dependent abstract tasks.[1]. We presented instantiated plan-decoupling as an approach for ensuring instantiated plan-coordination. Instantiated plan-coordination is a composition of both plan-coordination and instantiation-coordination. By constructing a coordination set and pruning conditions from a dependencies instance, we ensure that all instantiated plans an agent comes up with, can be merged into a feasible joint plan. Optimal instantiated plan-decoupling for a given problem is in general computationally intractable. However, we used the application domain Logistics to show that instantiated plan-decoupling is more easily achieved for some types of abstract complex tasks. By ensuring instantiated plan-coordination prior to planning, planners required less run-time to construct a plan of comparable plan quality.

To start with, we will discuss our conclusions in Section 5.1. Based on our work, we will identify viable directions for future research in Section 5.2. Finally, in Section 5.3, we will broaden our scope and discuss the applicability of our instantiated plan-decoupling approach.

5.1 Conclusions

In retrospect, we used four levels of abstraction to reason about planning problems. At the highest level, we have the planning *problem* itself. In the task coordination framework, a finer grained notion of *tasks* was used. In the enriched task coordination framework, we defined *abstract tasks* with preconditions and effects. At the lowest level of abstraction, we identified *actions*.

Plan-coordination is defined at the level of tasks. However, for various problems, tasks are at a too high level of abstraction to reason about coordination. They prevent us from reasoning about preconditions and effects of these tasks. To accommodate this, we refined the notion of tasks to include these conditions. We positioned *abstract tasks* between the abstraction level of tasks and that of actions.

[1] See Section 1.2.

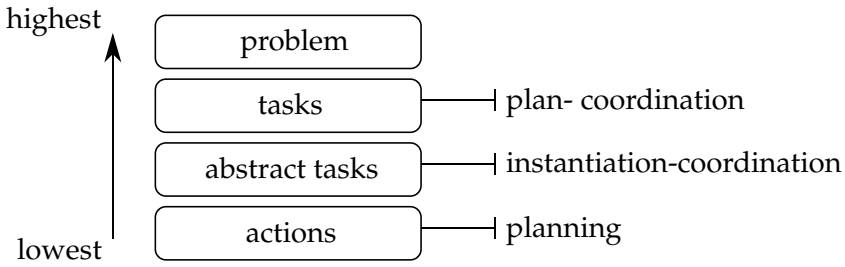


Figure 5.1: Levels of abstraction.

At this level, instantiation-coordination is defined. At the level of actions, plans are constructed. In Figure 5.1, the various abstraction levels are shown.

Instantiated plan-decoupling allows the coordination and planning phases to be separated. Decoupling is independent of the planning technique used at the level of actions. This ensures a broad applicability of our approach with respect to planning techniques.

Even better, decoupling could benefit these planners in some cases. We obtained significantly shorter planner run-times for decoupled Logistics instances, compared to the run-time required for non-decoupled instances. Despite this reduction in run-time, plan quality is hardly affected.

Given this observation, we note that the state-of-the-art planners we used might even benefit from decoupling planning problems. However, there is a snag in it. A domain-engineer, we, defined the tasks for Logistics. For decoupling to be a feasible extension to automated planners, tasks should be automatically detectable. This, we will identify as an interesting issue for future work in Section 5.2.

5.2 Future work

Our instantiated plan-decoupling approach is an enrichment of the plan-decoupling approach. We would like to see our approach as a stepping-stone towards an even more comprehensive pre-planning coordination approach. Hence, we will define, to our opinion, interesting issues that could be addresses in future work.

- By instantiated plan-decoupling, we allow agents to construct plans for their decoupled subproblems *concurrently*. On the other hand, we might construct an instantiated plan-decoupling approach that allows for a coordinated *sequential planning* process. In such a process, a set of agents constructs an instantiated plan. The implications of this plan for other agents are propagated to the next set of agents in the sequence. It would provide agents with more freedom to instantiate tasks. We wonder whether the instantiated plan-decoupling approach could be extended to allow for a sequential planning process.

- Tasks for the Logistics problem have been defined by us, the domain-engineer. In all previous work based on the task coordination framework, tasks were defined by hand. We initiated research to infer task definitions from classical planning problems. However, this research did not make it into this thesis.

  We defined tasks based on *landmarks* in classical planning problems.[2] A landmark is a fact that has to hold in every plan for the problem. In this sense, it is similar to a task, which has to be part of every plan as well.

5.3 Applicability

Our pre-planning decoupling approach might seem distant from any real-life applications. So far, our discussion was either formal or we applied our approach to an artificial planning problem. We do, however, feel that our approach has a broader applicability. It is not only applicable to the field of classical planning, as illustrated by the Logistics problem, but to other planning formalisms as well.

By means of our levels of abstraction, shown in Figure 5.1, we are able to abstract away from planning details. Whether tasks model actions in a classical planning problem or in an inter-organisational setting is irrelevant for our approach. We only require tasks to be defined for the application at hand. If only precedences and condition dependencies are required to model the problem, we can decouple it for use in a multi-agent setting with autonomous, self-interested agents.

When tasks, precedences, and dependencies are correctly defined, we proved that our instantiation plan-decoupling approach ensures agents to be coordinated with respect to their instantiated plans. Complications arise when other relations between tasks, like mutexes, are required. However, this leaves us with plenty of future work to be performed.

[2] A thorough discussion on landmarks can be found in Hoffmann et al. [20].

Bibliography

[1] F. Bacchus. The AIPS '00 planning competition. *AI Magazine*, 22(3):47–56, 2001.

[2] K.S. Barber and C.E. Martin. Autonomy as decision-making control. In *Intelligent Agents VII. Agent Theories Architectures and Languages, 7th International Workshop*, pages 343–345, 2000.

[3] A.L. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

[4] P. Buzing, A. ter Mors, J. Valk, and C. Witteveen. Task coordination for non-cooperative planning agents. In C. Ghidini, P. Giorgini, and W. van der Hoek, editors, *Proceedings of the Second European Workshop on Multi-Agent Systems (EUMAS 2004)*, pages 87–98, dec 2004.

[5] Y. Chen, C. Hsu, and B.W. Wah. SGPlan: Subgoal partitioning and resolution in planning. In *Proceedings of the Fourth International Planning Competition*, pages 30–33. International Conference on Automated Planning and Scheduling, 2004.

[6] J.S. Cox and E.H. Durfee. An efficient algorithm for multiagent plan coordination. In *Fourth International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 828–835, 2005.

[7] M.M. de Weerdt, A.W. ter Mors, and C. Witteveen. Multi-agent planning: An introduction to planning and coordination. In *Handouts of the European Agent Summer School*, pages 1–32, 2005.

[8] Y. Dimopoulos and P. Moraitis. Multi-agent coordination and cooperation through classical planning. In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 398–402, 2006.

[9] K. Erol, J. Hendler, and D.S. Nau. HTN planning: complexity and expressivity. In *Proceedings of the twelfth national conference on Artificial intelligence*, volume 2,

pages 1123–1128, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[10] R. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971.

[11] M.S. Fox, M. Barbuceanu, and R. Teigen. Agent-oriented supply-chain management. 12:165–188, April 2000.

[12] H.N. Gabow, S.N. Maheshwari, and L.J. Osterweil. On two problems in the generation of program test paths. *IEEE Transactions on Software Engineering*, SE-2(3):227–231, September 1976.

[13] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[14] A. Gerevini and D. Long. Plan constraints and preferences in PDDL3. Technical report, Dipartimento di Elettronica per lAutomazione, Università degli Studi di Brescia, via Branze 38, 25123 Brescia, Italy, August 2005.

[15] A. Gerevini and I. Serina. LPG: A planner based on local search for planning graphs with action costs. In M. Ghallab, J. Hertzberg, and P. Traverso, editors, *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems*, pages 13–22. AAAI, 2002.

[16] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory & Practice*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[17] M. Helmert. Complexity results for standard benchmark domains in planning. *Artificial Intelligence*, 143(2):219–262, 2003.

[18] M. Helmert. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173(5-6):503 – 535, 2009. Advances in Automated Plan Generation.

[19] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[20] J. Hoffmann, J. Porteous, and L. Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

[21] B. Horling, V. Lesser, R. Vincent, T. Wagner, A. Raja, S. Zhang, K. Decker, and A. Garvey. The TÆMS white paper, January 1999.

[22] C.W. Hsu and B. W. Wah. The SGPlan planning system in IPC-6. In *Short Papers of the Sixth International Planning Competition*, September 2008.

[23] V. Lesser, K. Decker, T. Wagner, N. Carver, A. Garvey, B. Horling, D. Neiman, R. Podorozhny, M. Nagendra Prasad, A. Raja, R. Vincent, , P. Xuan, and X.Q. Zhang. Evolution of the GPGP/TÆMS domain-independent coordination framework. *Autonomous Agents and Multi-Agent Systems*, 9:87–143, 2004.

[24] T.W. Malone and K. Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, 1994.

[25] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL - the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[26] B. Nebel and J. Koehler. Plan reuse versus plan generation: A theoretical and empirical analysis. *Artificial Intelligence*, 76(1-2):427–454, 1995.

[27] H.V.D. Parunak. Go to the ant: Engineering principles from natural multi-agent systems. *Annals of Operatins Research*, 75:69–101, 1997.

[28] J.S. Penberthy and D.S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *KR*, pages 103–114, 1992.

[29] S. Richter and M. Westphal. The LAMA planner, using landmark counting in heuristic search. Short Paper of the Sixth International Planning Competition, September 2008.

[30] Y. Shoham and M. Tennenholtz. On the synthesis of useful social laws for artificial agent societies (preliminary report). In *AAAI*, pages 276–281, 1992.

[31] J.R. Steenhuisen and C. Witteveen. Coordinating planning agents for moderately and tightly-coupled tasks. In Hamid R. Arabnia and P. L. Zhou, editors, *Proceedings of the International Conference on Foundations of Computer Science (FCS)*, pages 3–9. CSREA Press, jun 2007.

[32] J.R. Steenhuisen and C. Witteveen. Plan decoupling of agents with qualitatively constrained tasks. *Multiagent and Grid Systems*, 5(4), December 2009.

[33] J.R. Steenhuisen, C. Witteveen, and Y. Zhang. Plan-coordination mechanisms and the price of autonomy. In Fariba Sadri and Ken Satoh, editors, *Computational Logic in Multi-Agent Systems*, volume 5056 of *Lecture Notes in Artificial Intelligence*, pages 1–21. Springer-Verlag, 2008.

[34] A.W. ter Mors. Coordinating autonomous planning agents. Master's thesis, Delft University of Technology, Delft, The Netherlands, apr 2004.

[35] J.M. Valk. *Coordination among Autonomous Planners*. PhD thesis, Delft University of Technology, Delft, The Netherlands, 2005.

[36] J.M. Valk and C. Witteveen. Multi-agent coordination in planning. In M. Ishizuka and A. Sattar, editors, *PRICAI 2002: Trends in Artificial Intelligence: 7th Pacific Rim International Conference on Artificial Intelligence*, volume 2427 of *Lecture Notes in Artificial Intelligence*, pages 335–344. Springer, 2002.

[37] V. Vidal. A lookahead strategy for heuristic search planning. In S. Zilberstein, J. Koehler, and S. Koenig, editors, *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pages 150–160. AAAI Press, 2004.

[38] T. Wagner, J. Phelps, V. Guralnik, and R. VanRiper. An application view of COORDINATORS: Coordination managers for first responders. In *Proceedings of the Sixteenth Innovative Applications of Artificial Intelligence Conference*, 2004.

[39] D.S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.

[40] B.C. Williams and P.P. Nayak. A reactive planner for a model-based executive. In *Proceedins of the Fifteenth International Joint Conference on Artificial Intelligence*, pages 1178–1185, 1997.