



HAL
open science

Planning from operational models for deliberate acting in Robotics

Jérémy Turi

► **To cite this version:**

Jérémy Turi. Planning from operational models for deliberate acting in Robotics. Robotics [cs.RO]. INSA de Toulouse, 2024. English. NNT : . tel-04573729

HAL Id: tel-04573729

<https://laas.hal.science/tel-04573729>

Submitted on 13 May 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Doctorat de l'Université de Toulouse

préparé à l'INSA Toulouse

Planification à partir de modèles opérationnels pour l'action
délibérée en robotique

Thèse présentée et soutenue, le 26 février 2024 par
Jérémy TURI

École doctorale

SYSTEMES

Spécialité

Robotique et Informatique

Unité de recherche

LAAS - Laboratoire d'Analyse et d'Architecture des Systèmes

Thèse dirigée par

Daniel SIDOBRE et Arthur BIT-MONNOT

Composition du jury

M. Noury BOURAQADI, Président, IMT Nord Europe

M. Andrea ORLANDINI, Rapporteur, Istituto di Scienze e Tecnologia della Cognizione

M. Vicente MATELLÁN OLIVERA, Rapporteur, Universidad de León

Mme Karen GODARY-DEJEAN, Examinatrice, Université de Montpellier

M. Daniel SIDOBRE, Directeur de thèse, Université Toulouse III - Paul Sabatier

M. Arthur BIT-MONNOT, Co-directeur de thèse, INSA Toulouse

Membres invités

M. Félix Ingrand, CNRS Occitanie-Ouest

Résumé: Les récents développements technologiques dans le domaine de la robotique et de l'intelligence artificielle (IA) pourraient permettre l'utilisation de robots dans de nombreux domaines de notre vie. Les applications vont de l'industrie 4.0, qui vise à optimiser divers processus à l'aide de flottes d'agents autonomes, aux opérations de recherche et de sauvetage, sans oublier les robots d'assistance personnelle. À mesure que la complexité des plateformes robotiques augmente, les algorithmes de délibération doivent être améliorés, notamment pour gérer un nombre croissant d'agents, pour gérer des objectifs et des tâches complexes, et pour évoluer dans des environnements plus ouverts où les événements imprévus doivent être traités de manière autonome. Le niveau d'autonomie d'un agent dépend de cinq grandes fonctions de délibération : la planification, l'action délibérée, la surveillance, l'apprentissage et l'observation. Nous nous concentrons ici sur la fonction de planification, qui indique à l'agent ce qu'il doit faire pour accomplir ses missions, et sur la fonction d'action délibérée, qui adapte le comportement de l'agent au contexte d'exécution, ce qui le rend plus robuste face aux imprévus et aux aléas. Nous étudions en particulier l'interaction entre la planification et l'action délibérée. Bien qu'elles soient presque toujours utilisées ensemble, les approches dans la littérature ont tendance à les considérer séparément, ce qui limite leur interaction. Dans cette thèse, nous proposons une approche unifiée de la planification et de l'action délibérée, dans laquelle les deux systèmes sont en symbiose pour améliorer leurs performances respectives.

Nous présentons le système OMPAS (Operational Model Acting and Planning System), un moteur d'action qui exécute plusieurs tâches de haut niveau en parallèle en les raffinant en un ensemble de tâches et de commandes de plus bas niveau. OMPAS utilise un dialecte Lisp dédié (SOMPAS) pour définir le comportement de l'agent robotique. SOMPAS fournit des primitives pour gérer la concurrence et les ressources et, dû à son cœur restreint et à l'identification explicite de choix de délibération, permet la synthèse automatique des modèles de planification. Le moteur tire parti d'un planificateur temporel et hiérarchique qui utilise les modèles synthétisés pour anticiper et guider les décisions du système OMPAS. Le planificateur est utilisé de manière continue, c'est-à-dire qu'il planifie en même temps que l'exécution des tâches et s'adapte toujours à l'état actuel du système pour améliorer les décisions futures. Ces décisions devraient permettre d'éviter les blocages et d'optimiser de manière opportuniste l'achèvement de plusieurs tâches en parallèles.

Nous fournissons une évaluation de l'approche globale sur plusieurs domaines de la robotique. En particulier, OMPAS a été utilisé pour contrôler une flotte de robots dans une plateforme logistique simulée. Les résultats ont montré la capacité du système à gérer plusieurs tâches simultanées grâce à son système de gestion des ressources dédié. En outre, la planification continue améliore le temps total nécessaire à l'accomplissement de toutes les tâches d'une mission. La planification étant intégrée au cœur du système, aucun effort supplémentaire n'est requis de la part du programmeur du robot pour tirer parti de cette fonctionnalité.

Mots clés: IA, Robotique, Action délibérée, Planification en continu, Raffinement de tâche, Modèle opérationnel

Remerciements

Ce manuscrit est la consécration de plusieurs années de travaux que j'ai pu mener à terme grâce au support de nombreuses personnes que je tiens à remercier. Toutes ces personnes ont contribué, de près ou de loin, de manière directe ou indirecte, à l'accomplissement de ce doctorat.

Tout d'abord, merci à mes directeurs de thèse, Arthur Bit-Monnot et Daniel Sidobre pour m'avoir accompagné, conseillé, et guidé dans l'exécution de ces travaux. Merci à Arthur Bit-Monnot de m'avoir pris en stage, puis en thèse sur un domaine de compétence orthogonale à mes spécialités de base. Merci à Simon Lacroix, directeur de recherche au LAAS, de m'avoir accueilli dans l'équipe RIS pour mon stage de fin d'étude, puis par la suite en thèse. Merci à tous autres les membres permanents de l'équipe RIS, notamment Félix Ingrand, chargé de recherche, et Malik Ghallab, directeur de recherche émérite, pour m'avoir initié à cette discipline dont j'étais un complet néophyte aux débuts de cette thèse, et ce avec beaucoup de gentillesse et de bienveillance. Un second merci à Félix pour son aide durant la rédaction du manuscrit et pour la préparation de la soutenance de thèse. Enfin, je souhaiterais remercier les directeurs du LAAS, Liviu Nicu puis Mohammed Kaâniche, pour m'avoir accueilli dans leur laboratoire. Merci à l'école doctorale systèmes m'avoir fourni une bourse de thèse, et l'INSA de Toulouse de m'avoir accompagné durant ces quatre années de doctorat, après m'avoir formé à être ingénieur. Merci au département GEI de m'avoir permis d'effectuer des cours, une parenthèse qui fut très appréciable.

Merci à ceux qui m'ont précédé, Rafael, Amandine, Guillaume, Guilhem, Dario, Antoine, Gianluca, de m'avoir accueilli dans l'équipe avec beaucoup de bienveillance. Merci aux autres doctorants, qui sont amenés à soutenir dans les mois et années à venir, Philippe, Smaïl, Simon, Émile, Roland, Valentin, William, Anthony, Adrien, Harold, Virgile, Lou, Illinka, Dilon, Laure et Stéphy, pour les chaleureux échanges que nous avons eu tout au long de ces quatre années au LAAS. Un merci tout particulier à ceux (ils se reconnaîtront) qui m'ont accompagné durant mes tours de campus pour relâcher la pression, refaire le monde, ou pour tout simplement échanger sous les platanes du canal du Midi. Merci à mes amis, Maxime, Louis-Nicolas, Thomas, Loan, qui m'ont toujours soutenu, même bien avant la thèse, et qui ont continué à le faire tout au long de ce doctorat.

Merci à tous les membres de ma famille qui m'ont soutenu. Merci à ma sœur de m'avoir soutenu. Merci Mamina, merci grand-père pour m'avoir transmis votre expérience de vie et de m'avoir poussé à faire des études exigeantes. Merci à mon père de m'avoir inculqué le sens du travail, de la rigueur, de l'abnégation et du dépassement de soi. Merci à ma mère de m'avoir poussé à aller toujours plus loin, et à taper à la porte de la recherche pour voir ce qu'il s'y passait.

Enfin, merci à ma conjointe, Alicia, sans qui je n'aurais sûrement pas pu produire ce document aujourd'hui. Son support indéfectible a été un moteur qui m'a permis d'aller bien au-delà de ce que je pensais être capable de faire, et qui m'a prouvé, que je pouvais être (un peu) fier de moi.



THÈSE

En vue de l'obtention du
DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :
l'Institut National des Sciences Appliquées de Toulouse (INSA de Toulouse)

Présentée et soutenue le 26/02/2024 par :

Jérémy Turi

**Planning from operational models for deliberate
acting in robotics**

JURY

ANDREA ORLANDINI	Primo Ricercatore	Rapporteur
VICENTE MATELLÁN OLIVERA	Catedratico de Universidad	Rapporteur
DANIEL SIDOBRE	Maître de Conférences émérite	Directeur de Thèse
ARTHUR BIT-MONNOT	Maitre de Conférences	Co-Directeur de Thèse
NOURY BOURAQADI	Professeur	Examineur
KAREN GODARY-DEJEAN	Maîtresse de conférences	Examineur

École doctorale et spécialité :

EDSYS : Informatique 4200018

Unité de Recherche :

LAAS-CNRS

Directeur(s) de Thèse :

Daniel SIDOBRE et Arthur BIT-MONNOT

Rapporteurs :

Andrea ORLANDINI et Vicente MATELLÁN OLIVERA

Abstract:

Recent technological developments in the field of robotics and Artificial Intelligence could enable the use of robots in many areas of our lives. Applications range from Industry 4.0, which aims to optimize various processes using fleets of autonomous agents, to search and rescue operations, without forgetting personal assistance robots. As the complexity of robotic platforms increases, deliberation algorithms need to be improved, in particular to handle an increasing number of agents, to manage complex goals and tasks, and to evolve in more open environments where unforeseen events should be dealt with autonomously. The level of autonomy of an agent depends on five major deliberation functions: Planning, Deliberate Acting, Monitoring, Learning, and Observing. Here we focus on the planning function, which tells the agent what to do to accomplish its missions, and the deliberate acting function, which adapts the agent's behavior to the context of execution, making it more robust to contingencies and hazards. In particular, we study the interaction between Planning and Deliberate Acting. Although they are almost always used together, approaches in the literature tend to consider them separately, which limits their interaction. In this thesis, we propose a unified approach to Planning and Deliberate Acting, in which both systems are in symbiosis to improve each other's performance.

We present the Operational Model Acting and Planning System (OMPAS), a refinement based acting engine that executes multiple high-level tasks in parallel by refining them into a set of lower-level tasks and commands. OMPAS uses a custom Lisp dialect (SOMPAS) to define the behavior of the robotic agent. SOMPAS provides primitives for handling concurrency and resources, and, thanks to the restricted core language and the explicit identification of acting decisions, allows the automatic synthesis of planning models. The engine takes advantage of a temporal and hierarchical planner that uses the synthesized models to look ahead and guide the decisions of the acting system. The planner is used in a continuous fashion, i.e., it plans concurrently with the execution of tasks and always adapts to the current state of execution to improve the expected decisions. These informed decisions should avoid deadlocks and opportunistically optimize the completion of multiple parallel tasks.

We provide an evaluation of the overall approach on several robotics domains. In particular, OMPAS was used to control a fleet of robots in a simulated logistics platform. The results showed the ability of the system to handle several concurrent tasks thanks to its dedicated resource management system. In addition, continuous planning improves the total time to complete all tasks of a mission. Since planning is integrated into the core of the framework, no additional effort is required from the robot programmer to take advantage of this feature.

Keywords: AI, Robotics, Deliberate acting, Continuous Planning, Task refinement, Operational Model

Contents

List of Figures	i
List of Tables	ii
List of Algorithms	iii
1 Introduction	1
1.1 Motivation	1
1.2 State of the art	3
1.3 Contributions of the thesis	24
1.4 Motivating example: Gripper-Door	25
1.5 Outline	28
2 Architecture and Algorithms of the Operational Model Planning and Acting System (OMPAS)	29
2.1 Introduction	29
2.2 The Refinement Acting Engine (RAE)	30
2.3 The Operational Model Planning and Acting System (OMPAS)	44
2.4 Conclusion	57
3 Acting Language of OMPAS: Syntax, Semantic and Evaluation	59
3.1 Introduction	60
3.2 Acting languages: a review of the literature	60
3.3 The fundamentals of Scheme	68
3.4 Augmenting the Scheme core for control	76
3.5 Execution modules of the language	81
3.6 Configuration and control modules	85
3.7 Conclusion	94
4 Planning from Operational Models to Guide the Deliberation of OMPAS	95
4.1 Introduction	96
4.2 Background & related work	97
4.3 Automated generation of planning models from programs	108
4.4 Unique representation of the executed and anticipated processes	129
4.5 Guidance of the reactive deliberation of OMPAS using a planner	134
4.6 Conclusion	140

5 Evaluation of OMPAS: Simulated Domains and Integration with a Factory Simulator	143
5.1 Introduction	143
5.2 Setup of the evaluation	144
5.3 Gripper-Door: simulated domain	148
5.4 Gobot-Sim: integration with a factory simulator	155
5.5 Discussion	165
5.6 Conclusion	167
Conclusion	169
Bibliography	172
A Detailed translation techniques of SOMPAS programs into chronicles	185
A.1 Pre-processing of programs	185
A.2 Post-processing of the flow graph	188
A.3 Simplification of the Temporal Network of a generated chronicle	198
B Acting domains	201
B.1 Gripper domains	201
B.2 Gobot-Sim domain	206
C Long résumé en français	215
Acronyms	219

List of Figures

1.1	Schematic view of a deliberator and the interaction of the different deliberation functions with a centralized view of the models, data, and knowledge bases.	4
1.2	Schematic view of a three-layered architecture for a robotic system. . . .	10
1.3	Schematic representation of the initial state.	27
2.1	An architecture implementing Refinement Acting Engine (RAE).	31
2.2	Hierarchical representation of the methods of the <i>go2(?r)</i> task (2.2a) and the <i>move2room(?n)</i> (2.2b) task using the Hierarchical Task Network (HTN) formalism defined in Section 2.2.2.	34
2.3	Different traces of the deliberation of RAE in function of the method chosen to refine the task τ_1	39
2.4	Hierarchical representation of the <i>skills</i> of <i>place(?o, ?r)</i> in the RAE formalism.	41
2.5	Hierarchical decomposition of two tasks that are executed in parallel in vanilla RAE. Processes in red failed due to inapplicable commands or failure of a subtask in a method.	43
2.6	Architecture of the Operational Model Planning and Acting System. The Execution Manager, State Manager and Platform Manager inherit from the vanilla RAE architecture (see Figure 2.1). The Resource Manager, Acting Manager and PLanner Manager bring additional deliberation features to look ahead.	46
2.7	Example of an <i>Acting Tree</i> representing the execution of the task $t_1 = place(b_1, l_r)$ and $\tau_2 = place(b_2, l_r)$ of Example 2.3. The <i>Acting Tree</i> is only partially displayed. The methods of the tasks <i>place</i> use the new resource acquisition feature of Operational Model Planning and Acting System (OMPAS) to request the exclusive use of <i>Robby</i> during the execution of the skill. Processes in red have failed, triggering a retry of the task using a different method. We have an example of a retry for task τ_1	53
3.1	Illustration of the nested environments during the evaluation of a Scheme program. The <i>env</i> value represents the bindings at the moment the expression is evaluated. The standard bindings are supposed part of <i>env</i> and are not listed here to keep the example clear.	71
3.2	Example of interruption of an expression (<i>a</i>) consisting of a sequence of actions to be executed in less than 10 seconds. In the example of a real-time execution (<i>b</i>), the interruption <i>Int.</i> (in red) occurs during the action <i>move</i> , therefore <i>inspect r1 o3</i> is not executed.	79
3.3	Schematic representation of the modules loaded in the interpreter of the Execution Manager and their connections to the other managers of OMPAS.	81

3.4	Schematic representation of the configuration modules of the interpreter of REPL interpreter provided by OMPAS.	86
4.1	Examples of chronicles to represent actions of the Gripper-door domain (see Example 1.1): the command $pick(?o, ?r, ?g)$ (Figure 4.1a), and the methods $pick\&drop(?b, ?r, ?g, ?p)$ (Figure 4.1b) and $move\&drop(?b, ?r, ?g)$ (Figure 4.1c) of the task $place(?b, ?r)$	105
4.2	Schematic representation of the translation of a program defined in Scheme OMPAS (SOMPAS) into a chronicle.	108
4.3	<i>Flow graph</i> resulting from the translation of the program presented in Listing 4.1. The <i>flow graph</i> has been simplified: all nodes have been merged into a single one.	120
4.4	Chronicle resulting from the conversion of the <i>flow graph</i> of Figure 4.3. The chronicle has been postprocessed and simplified to make it more readable for the sake of this dissertation.	125
4.5	An example of an <i>acting tree</i> representing the execution of two tasks $t_1: place(b_1, l_r)$ and $t_2: place(b_2, l_r)$ (see Example 2.7). Both tasks are running in parallel, and both tasks have requested the exclusive use of <i>Robby</i> . OMPAS received the task t_1 at 0.4s (time relative to the start of the execution). The task t_1 was refined into $pick\&move(b_1, l_r, right, bedroom)$, which immediately requested the resource <i>Robby</i> . The resource was granted immediately (at relative time 0.4s) and the method requested the execution of the task $go2(bedroom)$. The task t_2 has been refined with the method $pick\&move(b_2, l_r, right, kitchen)$. The method is waiting for the resource <i>Robby</i> to continue its execution. All processes are currently running.	131
5.1	Mean of the total execution time T_E in Figure 5.1a of several configurations of OMPAS and the ratio of the deliberation time over the execution time RT_D in Figure 5.1b. The standard error for the two metrics is also represented.	154
5.2	Overview of a 6×6 job shop scenario in <i>Gobot-Sim</i> composed of one <i>input</i> machine (on the left) that feeds the environment with unprocessed packages, six <i>processing</i> machines that can do a predefined process, and one <i>output</i> machine (at the bottom right) that receives fully processed packages. Two robots can be used to dispatch packages on the machines. The recharge area (in yellow) is available at the bottom.	157
5.3	Continuous planning time for successive instances in <i>Satisfactory</i> and <i>Optimality</i> configurations	165
A.1	An example of successive operations on a forest $\{A, B, C, D, E\}$ using the Union-Find algorithms (see Algorithm A.1).	190
A.2	Lattices representing the lattice system of SOMPAS and its extension to handle specific types for planning	192

List of Tables

4.1	Definition and graphical representation of the Acting Processes (APs) in the <i>acting tree</i>	132
5.1	Table of configurations used in the OMPAS evaluation. The rows indicate the algorithm used by the SELECT function, while the columns indicate the configuration of the <i>continuous planning</i> function.	148
5.2	Results on the <i>Gripper-Door</i> domain averaged on three <i>easy</i> problems. .	151
5.3	Results on the <i>Gripper-Door</i> domain averaged on three <i>medium</i> problems.	151
5.4	Results on the <i>Gripper-Door</i> domain averaged on three <i>hard</i> problems. .	152
5.5	Results for the <i>Gobot-Sim_{JS}</i> and the <i>Gobot-Sim_{CS}</i> domains.	164

List of Algorithms

2.1	Main function of vanilla RAE.	35
2.2	Select function of Vanilla RAE.	36
2.3	Progress function of Vanilla RAE.	37
2.4	Retry function of Vanilla RAE.	37
2.5	Fluent monitoring process of OMPAS.	49
2.6	Main algorithm of OMPAS in which each new task τ is executed in a dedicated thread. The initial refinement of τ is part of the function call <i>Exec-Task</i> (τ)(see Algorithm 2.7).	50
2.7	Adaptation of RAE's procedure for executing a task τ . The procedure arbitrarily selects a method m that is applicable in the current state ξ and has not been tried before.	50
2.8	Plan-based method selection for a task τ	55
3.1	Overview of the recursive evaluation of expressions in Lisp	70
A.1	Union-find algorithm	191

Introduction

Contents

1.1	Motivation	1
1.2	State of the art	3
1.2.1	Deliberation in Robotics	3
1.2.2	Robotic architectures embedding deliberation	8
1.2.3	Blended deliberate Acting and Planning	13
1.2.4	Summary	23
1.3	Contributions of the thesis	24
1.4	Motivating example: Gripper-Door	25
1.5	Outline	28

1.1 Motivation

Recent technological developments in the field of *Robotics* and *Artificial Intelligence (AI)* could enable the use of robots in many areas of our lives. Applications range from *Future Industry*, which aims in particular to optimize various processes using fleets of autonomous agents, to search and rescue operations, without forgetting personal assistance robots. To integrate robots in such scenarios, the robots should be able to perform some parts of the missions autonomously, and this by limiting the need for a human to guide the robot.

Here we consider that a robot's perception, action, and reasoning capabilities define its degree of *complexity* and *autonomy*:

- **Perception** refers to the robot's ability to perceive and evaluate the environment in which it evolves.
- **Action** refers to the ability of the robot to evolve in this environment and to modify it by executing commands.
- **Deliberation** is the ability of a robot to organize and decide the actions it will take to accomplish its mission.

We consider robots through the combination of these capabilities, and a great deal of effort is required to design an autonomous agent. This long-term task is refined into

smaller engineering and research problems. In particular, in this dissertation we are interested in improving the deliberation capabilities of a fleet of robotic agents that should deal with multiple objectives while coping with unexpected events.

In the study by Ingrand and Ghallab (2017), the authors define deliberation in *Robotics* with six functions: *Planning*, *Acting*, *Observing*, *Goal Reasoning*, *Monitoring*, and *Learning*. Among these functions, we consider *Planning* and *Acting* as the core of the system's ability to achieve its goal and adapt to unexpected situations. Indeed, *Planning* gives the system the capacity to know "*what to do*" to accomplish a mission, while *Acting* finds "*how to do it*" to adapt to different contexts. This allows the system to autonomously decide what course of action to take given the execution context. However, while planning predicts the state the system is likely to end up in, most acting approaches tend to make decisions based on the current state of the system, without considering the long-term effects of deliberative decisions. To address this problem, research has explored ways to guide the choices made by the acting system with heuristics provided by a planner.

Many approaches already propose the integration of both acting and planning functions in the deliberation systems of robotic agents. However, their interaction is often limited to the execution of a high-level plan by an acting engine, where the acting engine is solely guided by the given plan. This integration is often limited by (i) the fact that both systems are often designed and developed separately, and (ii) because they use different models, which adds additional difficulties to their integration.

To avoid both problems, recent approaches such as the Refinement Acting Engine (RAE) (Patra, Traverso, et al. 2021) propose to design an acting system that natively supports the guidance provided by an automated planner. Such a system makes the choice to use a single model to perform both acting and planning, based on hierarchical operational models to model the capabilities of the robot.

In this thesis, we propose to continue the previous work by:

- Extending RAE with new ways to integrate planning techniques to guide the acting engine when controlling a fleet of robots performing multiple tasks in parallel.
- Relying on hierarchical operational models to both execute high-level tasks, and guide their execution with a hierarchical and temporal planner that provides useful heuristics about the methods to achieve those tasks, and how to optimally perform the different missions.

This chapter is organized as follows. First, we present how deliberation is viewed in *Robotics* and how it is integrated into a robotic architecture. The state of the art is then reviewed, focusing on the acting and planning capabilities of the deliberation system, and examining approaches that combine both functions. Based on this study, we present the contributions described in this thesis, along with a running example that is used as an illustration in the various chapters of this thesis.

1.2 State of the art

The goal of the following state of the art is to provide an overview of the deliberation methods used in *Robotics* and its larger role in a robotic architecture. The first part will be devoted to definitions proposed in the literature. Many reviews of the literature consider deliberation from different perspectives. Here we propose to examine the literature on how they integrate *Planning* and *Acting* in robotics deliberation architecture.

1.2.1 Deliberation in Robotics

Deliberation refers to the process by which an agent autonomously makes a decision. We also refer to deliberation as a cognitive capability. A deliberation system typically takes as input a set of information about the goals and the execution context:

- The mission to be accomplished,
- A representation of the world that includes both static knowledge and facts that are continuously updated,
- Models of the world and of the agent’s capabilities.

Based on this setup, deliberation has access to a set of algorithms that allows it to provide decisions as output.

There are many deliberation algorithms and systems in the literature. However, many of these deliberation algorithms can be classified into one of the six main *deliberation functions* that have been proposed by Ingrand and Ghallab (2017):

- **Planning:** searches for a sequence of actions that will lead a system to a desired state, either to achieve a goal or to perform a task.
- **Acting:** executes a set of tasks by refining them into the set of commands appropriate to the current context, while monitoring their execution and responding to changes in the state of the system.
- **Observing:** actively reasons about the state transition of the system by analyzing both the perceived state change and the internal processes currently being executed, such as plans and commands.
- **Monitoring:** analyzes the discrepancies between the estimated state and the observed state, interprets them in terms of the mission, and can make corrections to the system’s trajectory in the space in which the system evolves.
- **Goal Reasoning:** keeps track of the system’s high-level commitments and goals, and adapts the system’s goals as a function of the relevance of the current goals, either abandoning or modifying them.
- **Learning:** adapts and improves its behavior through experience. The models used are updated to make the system more responsive to future experience.

Figure 1.1 summarizes the interactions between the different functions present inside a deliberator. In this representation, the deliberation functions take advantage of shared

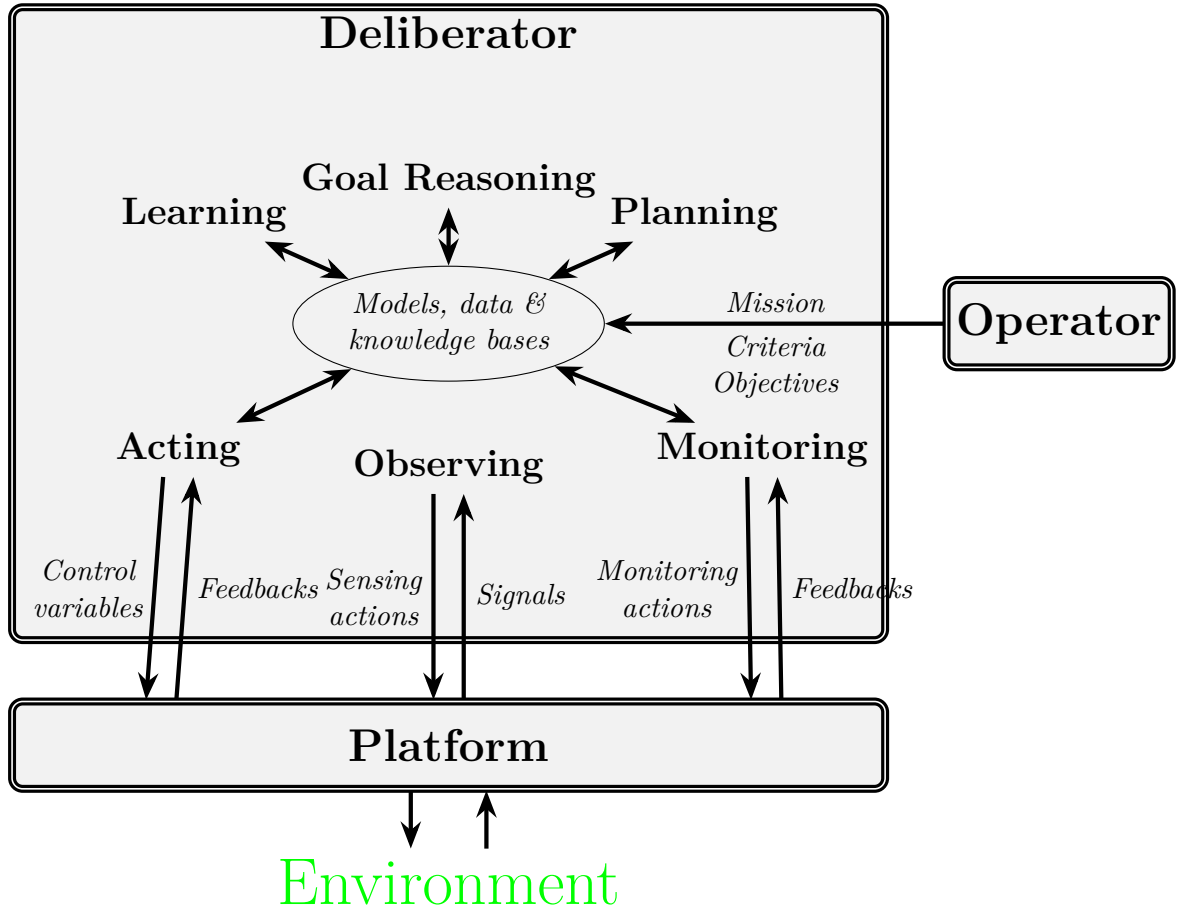


Figure 1.1: Schematic view of a deliberator and the interaction of the different deliberation functions with a centralized view of the models, data, and knowledge bases.

bases of models, data and knowledge. Only three functions are directly interacting with the rest of the agent: *Acting*, *Observing* and *Monitoring*.

For more details on each of the functions, the survey by Ingrand and Ghallab (2017) provides an overview of the literature for each of the functions mentioned above.

As mentioned before, this dissertation focuses on the approaches that combine *Acting* and *Planning*. Therefore, we extend their presentation in the following sections. This will provide the necessary definitions to present the core of the thesis.

1.2.1.1 Planning

This section provides a simple definition of planning. More details about approaches and systems in the planning literature are discussed in the Chapter 4. In its simplest form, *Planning* searches for a sequence of actions whose execution should move the system from an initial state to a final state in which its goals are met. In particular, we refer to *Task Planning*, which takes advantage of a representation of the agent capabilities as a

symbolic state transition system defined in a *descriptive model*.

Typically, a *descriptive model* is a Planning Domain (P_Δ), a tuple $(\mathcal{W}, \mathcal{A})$ where:

- \mathcal{W} describes the set of states that the system can be in. A state is typically represented as a set of state variables $sf(p_1, \dots, p_n) = v$. For example, $location(r1) = bedroom$ states that the location of $r1$ is *bedroom*.
- \mathcal{A} is the set of actions that can be performed to induce state transitions. An action is defined by a set of *preconditions* that define the state in which the action is applied. Preconditions are typically defined as a set of propositions over the state variables representing the world. The transitions due to the execution of the actions are defined as a set of *effects* that define the new values that the state variables take once the action has been executed.

Based on this definition of P_Δ , we define a planning problem (P_Π) as a tuple $(\mathcal{W}, \mathcal{A}, \mathcal{G})$, where \mathcal{G} represents the goal state to be achieved as a set of values that some state variables should take.

Here we define a *planner* as an algorithm that produces a plan to meet all goals of \mathcal{G} and that is valid with respect to the given model and the initial state of the system.

In its simplest form, planning uses a model of actions that describes the set of states in which the actions are applicable, and the set of updates that are applied to the state upon execution of the said action. This was first formalized as the STanford Research Institute Problem Solver (STRIPS) (Fikes 1971) planning. A more complete definition of planning can be found in the book *Automated Planning and Acting* (Ghallab, Nau, and Traverso 2016).

However, the STRIPS model is a simple representation of the real world that may not be sufficient for planning more complex problems. Several extensions have been proposed to improve the capabilities of the planning system to more accurately represent the possible evolutions of a system. Among them, we can note the followings:

- **Temporal Planning** is interested in the accurate representation of the timing and the order of the evolution of the state. It introduces an action model in which actions are no more considered as instantaneous, and that allows parallel execution. Among the temporal planners we can cite IxTeT (Ghallab and Laruelle 1994) and the Lifted Constraint Planner (LCP) (Bit-Monnot 2018).
- **Numerical Planning** extends the expressiveness of a planning problem to allow numerical expressions. Numerical planning takes advantage of specific techniques to find solutions that account for the use of a numerical resource, such as the fuel consumed by a truck as a function of the distance traveled. The Expressive Numeric Heuristic Planner (ENSHP) (Scala et al. 2016) is an example of numerical planner.
- **Hierarchical Planning** takes advantage of a hierarchical model of an agent as a tuple $(\mathcal{C}, \mathcal{T}, \mathcal{M})$, where the *Commands* (\mathcal{C}) represents the low-level capabilities of the agent, the *Tasks* (\mathcal{T}) represents the more complex and abstract capabilities

of the agent that should be refined using the *skills* defined in the set of *Methods* (\mathcal{M}). Each method represents a different way of performing a task defined in \mathcal{T} . A method traditionally consists of a sequence of lower-level commands and tasks, the execution of which fulfills the high-level goal of the task. In this context, the role of a hierarchical planner is to refine a high-level goal down to a sequence of commands that can be executed by the robot platform. The SHOP (Nau, Cao, et al. 1999) and FAPE (Bit-Monnot et al. 2020) systems are two examples of planners taking advantage of hierarchical models.

- **Nondeterministic Planning** supports probabilistic models of the agent in which the actions are no longer assumed to be deterministic. Although planning still searches for a valid plan to achieve the goal, non-deterministic planning is no longer interested in a sound solution but in the generation of policies, where each policy is a tuple (s, a) that maps a state s to an action a to perform. We can count on Stochastic Shortest Path (SSP) (Natarajan and Kolobov 2022) and RFF (Teichteil-Königsbuch, Kuter, and Infantes 2010) among the planning systems that rely on the generation of partial policies.

It should be noted that many systems combine several approaches, such as temporal and numerical planning. In deliberation, *Planning* is often used to generate behaviors in the form of a plan that is either executed directly on the platform, or given to an acting system that will execute the plan in a more robust way.

1.2.1.2 Acting

As stated by Ghallab, Nau, and Traverso (2016), automated deliberation can be separated into two systems: a planning system that generates a plan composed of a set of actions to be performed; an acting system in charge of monitoring the execution of plans and adapting the agent’s behavior to the current state of the system.

While *Symbolic Planning* deliberates at a high level of abstraction, acting is responsible for executing actions while coping with the real-time constraints of a robotic platform, e.g. recovering from failures, adapting to unexpected events, dealing with temporal constraints such as expected exogenous events and deadlines. Therefore, unlike planning algorithms that compute the plan offline, acting deliberation is online, i.e. each decision is a function of the actual perceived state of the system. Therefore, the models implied in the deliberation differ as they should define the executive behavior of the robotic agent, and should encode a robust behavior to adapt to contingent events.

While *Planning* has been properly formalized for decades since the early development of PLANEX1 using the STRIPS formalism (Fikes 1971), the notion of *Acting* appeared more recently with some attempts to formalize it. Before that, such deliberation function would be referred as *Supervision*. In a position paper on blended *Planning* and *Acting* (Nau, Ghallab, and Traverso 2015), the authors emphasized the need to formalize *Acting* to drive research in this area, just as the development of planning languages such as the Planning Domain Description Language (PDDL) did for *Planning* several decades ago.

In the book *Automated Planning and Acting* (Ghallab, Nau, and Traverso 2016), we find a definition for *Acting* that states its differences with a simple execution of a sequence of actions. As defined in the book, deliberate acting relies on *operational models*, which can be any model that takes advantage of generic algorithmic structures such as branches and loops to define a variety of behaviors. They are generally more expressive than *descriptive models* when it comes to error management and handling nondeterminism. They can be defined using general-purpose languages, taking advantage of rich programming tools that can be used to define agent behavior (including, for example, error handling, conditions, loops).

Whereas planning systems reason on *descriptive models* that explicit the dynamics of the world and the capabilities of the robotic agent, acting systems rely on a collection of *skills* defined in an Acting Domain (A_{Δ}) to achieve the tasks. The role of the acting engine is to execute the most appropriate *skills* to face the tasks.

To do that, an acting system has several deliberation abilities, which have been categorized in the survey by Ingrand and Ghallab (2017) as follows:

- The **Refinement of a task** gives the ability to the system to execute an abstract task by selecting a *skill* of A_{Δ} and executing it. The selected *skill* should achieve the high-level goal of the task and be executable in the current context. Skills are either provided by a model, or can be synthesized online to adapt to unhandled scenarios. By providing several alternatives to execute a task, it also gives the acting engine a choice about which course of action to take.
- The **Reaction to Events** allows the system to adjust its behavior. In some cases, the expected course of action should be adjusted to take into account the impact of the events on the execution of the current tasks. Failure to do so would be a cause of failure for another task being performed by the system.
- The **Parameterization** allows *skills* to be parameterized, making them more versatile. It is up to the acting engine to choose which set of parameters to use to parameterize a *skill*. For example, a *skill* might require one of the robot arm's to pick up an object on a table. Whether it is the *right* arm or the *left* is left to the discretion of the *acting engine*.
- The **Time Management** is critical in an acting system because it performs on-line deliberation. The acting system should be able to reason on explicit time to handle operational constraints such as deadlines, durations, rendezvous, time synchronization, etc.
- The **Nondeterminism** of action execution should be handled by the acting engine. In fact, the failure of a command does not necessarily invalidate the future decisions anticipated by e.g. a task planner. The acting engine can handle failures locally using recovery procedures. Furthermore, the acting engine should be robust to nondeterministic perception and unexpected events.
- **Plan Repair** is ultimately detected and triggered by the acting system, since it should be based on the divergence between the expected execution outcome and

the actual one. The acting system should therefore be able to detect when *Plan Repair* is necessary, i.e. when the acting system cannot fix the divergence itself, and pass on the important information needed by the *task planner* to provide a new plan.

- **Resource Management** is an additional feature that should be addressed by the acting engine. Especially when there are multiple concurrent goals that require limited resources. Therefore, the acting engine should have a reactive resource allocation strategy.
- **Formal Modeling** opens the way to ensure the behavior of an agent using Verification and Validation (V&V) tools. The acting engine can take advantage of such a model to check some properties of the system at runtime, such as detecting possible deadlocks.

Through these capabilities, *Acting* provides the reactive deliberation capabilities to the robotic system. In fact, the role of *Acting* has expanded over time with the evolution of robotic architectures.

1.2.2 Robotic architectures embedding deliberation

Deliberation is one of the main components in the architectures of autonomous robots. What is interesting is the evolution of deliberation functions embedded in robot architectures.

The presented approaches and systems have all been studied in the review “Robotic Systems Architectures and Programming” (Kortenkamp and Simmons 2008). What we propose is to focus on architectures that make deliberation the system’s top-level decider, meaning that down to the system’s lowest controllers, an action is taken because the deliberation system has decided to do so.

1.2.2.1 Sense-Plan-Act (SPA) architecture

One of the first architectures to embed deliberation was the Sense-Plan-Act (SPA) architecture. In SPA, the robot performs a mission by first sensing its environment to gather information that the planning system uses to find an appropriate sequence of actions to perform. This loops until the robot has completed its mission. This architecture was used on the Shakey robot at the Stanford University in the late 1960s (Nilsson 1969). However, such an approach is rather limited because the robot must wait until a complete valid plan is found before acting, even though the planning time may be long enough to require sensing again. The possibility of having an outdated plan is even more present when it comes to acting, since the environment is evolving (at least due to the robot’s actions) and should be taken into account by the robot.

1.2.2.2 Subsumption architecture

Previous approaches using SPAs architectures were slow because of this loop, which requires that each function should be completed before the next one is executed. To

address this problem, Brooks (1986) proposed the *Subsumption Architecture*, which decomposes the global loop of a SPA into a collection of state machines called *behaviors* that decomposes the system into several differentiated sets of sensors and actuators. Each *behavior* manages its own planning and execution. This makes the system more responsive to new information and easier to adapt to contingencies. Note that all *behaviors* run concurrently and can therefore request commands on the same actuators. To orchestrate the execution of concurrent behaviors, higher-level behaviors can be defined to override signals from lower-level behaviors, allowing multiple behaviors to be defined to control the same actuators, but activated in a context-dependent manner.

However, while this improves the responsiveness of a robot, it loses some of the ability to optimize its behavior as a function of higher-level goals that was provided by the global planning of the SPA architecture. This led to the development of *Layered Architectures*, the purpose of which is to take the best of both worlds to make a robot capable of optimizing its behavior to achieve a high-level goal, while still being reactive and able to adapt to unexpected events.

1.2.2.3 Layered architectures

Over the years, software developers of autonomous systems have needed an abstract architecture to facilitate the development of complex autonomous systems. Indeed, decomposing a complex system into simpler subsystems and formalizing their interaction would facilitate the development and improvement of the overall system by tackling smaller problems. A recurring decomposition that emerges from the various approaches is the layered system, where each layer corresponds to a different level of abstraction. In fact, over the years, researchers and software engineers have found that having different levels of autonomy would facilitate the development of specific reasoning tools and their combination; a single algorithm would either not scale or would be too difficult to develop and would lack versatility.

Many works proposed a 3-layer approach (Gat, Bonnasso, and Murphy 1998), in which we distinguish the following systems:

- The **Controller (or functional layer)**, which takes care of the low-level execution of motor commands and sensing of the robotic platform performed in the different modules.
- The **Sequencer (or executive layer)** that receives command execution requests, verifies their applicability, distributes them to the corresponding functional modules, and monitors their execution.
- The **Deliberator (or decision layer)**, which receives the more abstract form of objectives that the system should deal with by (i) finding an appropriate set of abstract actions to accomplish the mission, and (ii) refining those actions down to executable primitive commands that can be dispatched by the executive.

In such an architecture, the three layers interact either sequentially or in parallel to allow different levels of responsiveness at each layer. We will now present some examples of

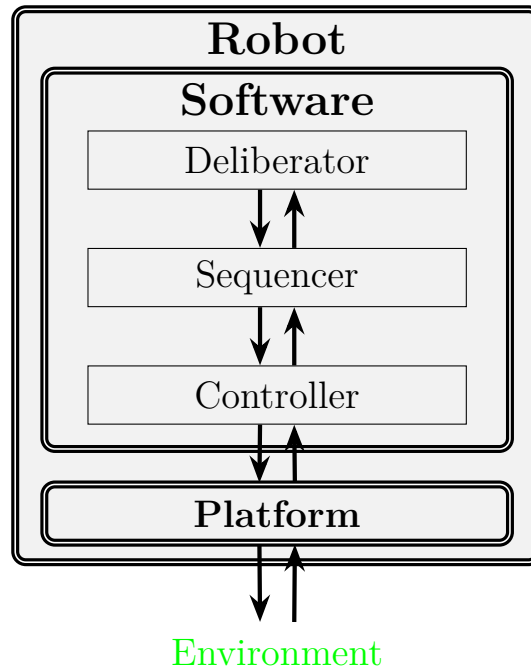


Figure 1.2: Schematic view of a three-layered architecture for a robotic system.

three-layered architecture.

Atlantis The Atlantis Architecture (Gat 1992) is a perfect example of three-layered architecture:

- The *Controller* is defined using the ALFA (Gat 1991) language. The peculiarity of this language is that *cognizant failures* should be handled explicitly in the programs. Indeed, a system can hardly guarantee error-free behavior, and should detect errors so that they can be handled by another system to recover from them.
- The *Sequencer* is using the Task Control Architecture (TCA) (Simmons 1991). TCA uses activities to represent rich capabilities by defining abstract operations that can be composed of lower-level activities, the lowest being the primitive activities that correspond to the executable actions of the controller.
- The *Deliberator* relies on a *task planner* based on the work of Miller (Miller 1985) that provides a plan with a bounded horizon that is executed by the *Sequencer*. The planner is called again when either the plan reaches its horizon, i.e. no more action is available, or the execution deviates from the plan, e.g. due to a command failure.

LAAS architecture The LAAS architecture (Alami et al. 1998) is another three-layered architecture. It has been deployed successfully on autonomous rovers (Ingrand,

Lacroix, et al. 2007). Unlike Atlantis, it proposes a more advanced decision layer consisting of both a *planner* and a *procedural engine*. The proposed architecture is as follows:

- The *functional layer* uses G^{en}oM (Foughali, Ingrand, and Mallet 2018), a functional module generator that would automatically generate appropriate controllers that ensure properties on the modules composing the functional layer. More recently, G^{en}oM templates could be used to automatically synthesize a corresponding Fiacre (Berthomieu, Zilio, and Vernadat 2020) formal model suitable for V&V (Dal Zilio et al. 2023). In particular, Fiacre models are used to ensure timing properties on critical systems.
- The *executive layer* was primarily using KHEOPS (Medeiros, Chatila, and Fleury 1996). Then the Requests and Resources Checker (R2C) system (Ingrand and Py 2002) was used to ensure the executability of a command before sending it to the functional layer.
- The *decision layer* of the LAAS architecture consists of two systems: a *task planner* that generates a plan consisting of a sequence of high-level actions, and a *supervisor* that executes the high-level plan by refining each action down to a sequence of commands that are passed to the executive layer. The task planner IxTeT (Ghallab and Laruelle 1994) supports temporal planning, allowing it to reason on partially ordered plans that are more robust to variations in the system. The *supervisor* is the Procedural Reasoning System (PRS) (Ingrand, Chatila, et al. 1996), an acting system that resolves open goals by executing procedures. PRS takes advantage of a hierarchical representation of agent capabilities that allows reasoning about high-level goals and the methods for achieving them by executing lower-level actions. Similarly to the Reactive Action Package (RAP) (R. J. Firby 1989), PRS achieves high-level goals by iteratively refining them into executable procedures composed of commands or tasks. The refinement is done at runtime, which avoids the need to generate the full command sequence before execution. The refinement strategy of PRS is to execute an arbitrary method in a trial-and-error fashion until the mission succeeds.

RAX architecture The development and deployment of the Remote Agent Experiment (RAX) (Muscettola, Nayak, et al. 1998) is considered by many to be an important milestone in the robotics community, particularly for those interested in deliberation. It was one of the first successful integrations of a complete deliberation system into an autonomous system with high performance and robustness requirements.

Remote Agent was one of the first to take advantage of *model-based programming*. The proposed methodology is to define the agent capabilities as a set of actions (or activities) that can be used in arbitrary ways. Based on these actions, the system is capable of *learning* and *searching* autonomously how to compose these actions to fulfill its objectives by, e.g. rely on an *automated planner* to adapt the behavior of the agent depending on the goals and the context.

Unlike other layered architectures, RAX is decomposed into four systems:

- The *decision layer* is composed of the temporal planner called Planner/Scheduler (PS) and the Mission Manager (MM). PS is capable of automatically generating plans that take into account the timing of access to limited resources and hard deadlines. MM triggers the replanning of PS based on information coming from the *executive layer*.
- The *executive layer* is the Smart Executive (EXEC) that schedules and manages the activities that make up the plans provided by PS. Similar to PRS, EXEC refines activities down to commands that can be executed on the robot platform. The activities and their refinements are defined using the Execution Support Language (ESL) (Gat 1997), another procedural language.
- The equivalent of the *functional layer* is the model-based Mode Identification and Reconfiguration (MIR) system, which is used to abstract the sensor values as abstract states. These abstract states can then be used at a higher level, in particular for fault detection. MIR does not fully define a functional layer, as it does not manage the low-level controllers of the system.

In this architecture, we can see that deliberation is present at each and every level. Indeed, MM provides goal reasoning that can be used by PS to give instructions in the form of plans to EXEC, which in turn provides reports to MM to adapt the behavior of the system. Even if the deliberation is decomposed into multiple systems, their entanglement is such that at best the system would have some degree of autonomy without the system, and at worst would not work at all. The EXEC and MIR systems also take advantage of the abstraction of the model of the system to facilitate deliberation.

CLARAty architecture The Coupled Layer Architecture for Robotic Autonomy (CLARAty) (Volpe et al. 2001) is another layered architecture that de facto merges the decision and executive layers into one, while keeping the functional layer separate. This stems from the fact that the procedural reasoning of the executive system and the planner should interact for most of the decisions made by the system. It also takes advantage of the effort to use a common representation for both planning and execution.

What the CLARAty architecture does is to add a third dimension to the specification of a robot architecture: in addition to the abstraction (vertical axis) and the system (horizontal axis), another horizontal axis orthogonal to the system should represent its granularity. In fact, most layers are themselves decomposed into subsystems, each operating at a finer granularity.

Such an architecture clearly demonstrated the need to decompose deliberation at different levels and, in particular, to add deliberative capabilities directly to control the functional layer, without the executive layer as an intermediary. Either way, since they often work in a bidirectional relationship, executive and decision can benefit from tight integration.

1.2.2.4 Teleo-reactive architectures

Interesting approaches inherited from RAX-PS (Muscuttola, Nayak, et al. 1998) and attempted to unify the different models used by the different deliberation functions. This led to the development of the Intelligent Distributed Execution Architecture (IDEA) (Muscuttola, Dorais, et al. 2002) and the Teleo-Reactive EXecutive (T-REX) (McGann et al. 2007; Py, Rajan, and McGann 2010), which merge the decision and execution layers into a hybrid system using a unified model for deliberation and execution.

In IDEA and T-REX, the system architecture is composed of reactors, where each reactor use a common design pattern. Each reactor is responsible for one function of the system, e.g., mission management, and has access to deliberation features such as lookahead. This way, deliberation is also brought at the functional layer, and is closer to the actuators and sensors of the system. Each reactor operates at a different level of abstraction, parameterized by the latency of the reactor, i.e. the maximum allowed time for deliberation, and a planning horizon that the reactor should consider. The autonomous system is thus defined as a collection of reactors interacting with each other using shared timelines representing the evolution of state variables over time and synchronized with a global clock. IDEA and T-REX rely on a Constraint Satisfaction Problem (CSP) approach to dispatch the temporal constraints generated by a planner.

Both systems have been integrated with Europa (Barreiro et al. 2012) up to the commands executed in the different reactors of the system, and the Advanced Planning and Scheduling Initiative (APSI) (Fratini and Cesta 2012) has also been integrated with T-REX. T-REX has been successfully integrated into several Autonomous Underwater Vehicles (AUVs) (Py, Rajan, and McGann 2010; Rajan and Py 2012; Rajan, Py, and Barreiro 2012). It should be noted that T-REX is planner independent by design. While the hybrid design may seem interesting, such system specification and debugging has been proven tedious on large models. Moreover, the programming of the *skills* is still required.

1.2.3 Blended deliberate Acting and Planning

In the previous section, we have seen that approaches can differ on how to integrate deliberation in a robotic architecture. In particular, the notion of *reactors* brought by IDEA (Muscuttola, Dorais, et al. 2002) and T-REX (McGann et al. 2007) were one of many approaches that blend *Planning* and *Acting* into a unified framework.

These approaches learned from the fact that robotic architectures almost always integrate a planning component and an execution component that are required to interact with each other. Most importantly, the interaction between the two components is bidirectional. On the one hand, it is obvious that the planner should give the plan to the execution. On the other hand, the execution should at least inform the planner about the success or failure of the plan. In any case, both functions should be implemented with these constraints in mind. Other approaches propose a more advanced integration of both functions, resulting in deliberation systems with enhanced autonomy. We

propose to study the systems that have such a unified approach of *Planning* and *Acting*.

In this review, we propose to study their deliberation strategy, the features provided by the combination of *Planning* and *Acting*, and how models of both deliberation functions interact with each other. We examine and attempt to classify the deliberation approaches that unify *Planning* and *Acting*. However, the unification of the two functions is broader than a simple addition of the features of each system, and should be considered as a new function, which we refer to as a *deliberation strategy*. A *deliberation strategy* might bring new deliberation features. But it could also improve the features of the two functions that were already present. Since the two functions are presented as a whole in these approaches, we examine how the *deliberation strategy* deals with the discrepancies between the *descriptive models* and the *operational models* used by the *planning* and *acting* systems, respectively.

1.2.3.1 Blended deliberation strategies

Plan-Exec: interleave Planning and Acting One of the most common deliberation strategy is *Plan-Exec*. In *Plan-Exec*, the *planner* is called first to generate a plan, which is then executed by a supervisor (either a simple executive or an acting engine).

PLANEX1 (Fikes 1971) is one of the first proposals for a mixed planning and execution system using a *Plan-Exec* strategy. It relies on the planner STRIPS to produce a plan that is executed by PLANEX1 in a *Plan-Exec* fashion. In PLANEX1 there is a direct mapping of the steps of the plan to motor commands of the robot platform.

The Reactive Action Package (RAP) (J. Firby 1987) system was one of the first systems to introduce the notion of reactive planning, in which the system must act primarily on the current state of the system rather than on expected states, as planners would do. At the time of its presentation, the authors pointed out the need to look-ahead, in addition to being reactive, in order to prevent undesirable behavior. In this sense, RAP has been used in a *Plan-Exec* fashion with the planners AP (Bonasso et al. 1996) and PRODIGY (Velooso and Rizzo 1998). In this approach, PRODIGY is used to automatically synthesize RAPs capabilities based on the operators used by the planner. The earlier deliberation architectures we have presented often rely on a *Plan-Exec* strategy, such as the LAAS architecture (Alami et al. 1998) and RAX (Muscuttola, Dorais, et al. 2002).

More recently, the Flexible Acting and Planning Environment (FAPE) (Bit-Monnot et al. 2020) planner proposed an approach to execute temporal plans in the form of chronicles using a dedicated acting system capable of directly using these chronicles to act. By incorporating traces directly into the chronicles, the system is able to check the validity of the plan at runtime and better cope with uncertainty. In particular, by directly mutating the chronicles used by the planner, FAPE can first try to repair with a timeout before replanning. This approach can improve the responsiveness of the system, as repair may be faster than replanning.

Similarly to FAPE, PLATINUM produces a plan that takes into account the temporal uncertainty inherent to the execution of a robotic system, especially in Human Robot Collaboration (HRC) contexts. Its internal representation allows a more flexible

adaptation of the system to contingencies.

At another level, the Robotic Operating System (ROS) (Quigley et al. 2009) has been used to implement a generic *Plan-Exec* deliberation strategy in a robotic system using the middleware to interface with any planner supporting the Planning Domain Description Language (PDDL) (McDermott et al. 1998). ROSPlan (Cashmore et al. 2015) and PlanSys2 (Martín et al. 2021) are two proposals for generic *Plan-Exec* frameworks to use planning in ROS and ROS2 environments respectively. They facilitate the invocation of a planner by decoupling planning from execution, using ROS as an intermediary between the different systems.

The Task Control Architecture (TCA) (Simmons 1992) proposed a *Plan-Exec* architecture in which *Planning* and *Acting* are concurrent processes. Indeed, the planner would anticipate the next step of the plan while executing the first. To do this, the new planning problem starts from the end state of the previous plan, assuming that execution will result in the expected state.

Rogue is another approach to interleaving *Planning* and *Acting* (Haigh, Veloso, and Bekey 1998). It can asynchronously receive new goals from multiple users, and is able to prioritize goals and focus on high-priority goals until they are achieved, recognize similar goals that can be achieved by common actions, and deal with emergencies by interrupting actions that can be restarted later appropriately, i.e. get back to a nominal state in which the action can be resumed. The interweaving of *Planning* and *Acting* is of particular interest here to gather useful information that would help a planner predict failures and thus avoid them (Haigh and Veloso 1998).

Plan-as-Advices: online planning to guide acting The second *deliberation strategy* is *Plan-as-Advices*, where the *planner* is used to guide the decisions that the *supervisor* should make at runtime. The fundamental difference with *Plan-Exec* is the role of the *planner*: On the one hand, the *planner* is the ultimate deliberative, on the other hand, it is used to inform the *supervisor* with its lookahead capabilities. The remarkable thing about how Atlantis works is how it interacts with the deliberator. When Atlantis was introduced, it was one of the first deliberation systems to use the *Plan-as-Advices* strategy. In fact, in Atlantis, the execution of actions is simulated to guide the acting engine (either RAP (R. J. Firby 1989) or ESL (Gat 1997) depending on the implementation).

In the approach developed by Beetz and McDermott (1994), the planner XFRM is used to sample the outcome of actions, and use this to parameterize the *skills* defined with the language RPL.

Propice-Plan (Despouys and Ingrand 2000) extends PRS with the guidance of the refinement process. Indeed, reactive method selection can lead to a suboptimal choice, which in the worst case can lead the system into deadlocks. Propice-Plan adds to PRS an *Anticipation Module (AM)* that simulates the execution of the different refinements to advise the acting system on the best method to choose, and a continuous planning module that adapts the high-level plan in function of the feedbacks from the acting system.

The PROcedure Planning and Execution Language (PROPEL) (Levinson 1995) proposes to use a unique model for both *Planning* and *Acting*. The agent's capabilities are defined as procedures in which the nondeterministic choices are explicitly defined and can be resolved using a planner. In addition, a procedure can be augmented with a simulation model representing the state transitions due to the execution of the procedure, which can be used by the planner to simulate the execution of the procedures.

1.2.3.2 Augmented deliberation capabilities: repair and replanning

One of the role of a deliberation system is to handle failures and unexpected events. One commonly used approach in deliberation is to repair a plan, or replan. In the case of replanning, it can be costly. Therefore, one might want to avoid replanning and resort to local repair to minimize the downtime of the system. Among the improvements brought to deliberation by blending *Planning* and *Acting*, the handling of repair and replanning is one of the main feature that benefits from it. In fact, many systems integrates both features to tackle this problematic.

One approach is to provide a plan that is flexible enough to handle a failure without requiring replanning. In fact, only a part of the plan may be invalidated and thus need not be re-synthesized. This is partly supported by PLANEX1 (Fikes 1971), which views the plan as a set of actions, the order of which is not explicit. Therefore, replanning is only triggered when no more actions of the plan are applicable.

In a more sophisticated approach, IxTeT-Exec (Lemai-Chenevier 2004) is a deliberation system that proposes to use a single model for both temporal planning and temporal dispatching. Temporal planning is handled by the CSP temporal planner IxTeT (Ghalab and Laruelle 1994) already present in the LAAS architecture, and complemented by the temporal dispatcher Exec, which interfaces between the planner and the supervisor PRS (Ingrand, Chatila, et al. 1996). The same structure used by IxTeT to search a temporal plan is used by Exec to trigger the execution of actions, but also to represent failed commands directly in the shared partial plan. Since the failure is represented directly in the subplan, IxTeT can dynamically search for a new plan. Of course, continuous planning and replanning is facilitated by this unification of the planning and execution model. However, in this approach, the role of Exec is only to dispatch the high-level actions down to PRS, which would have to refine them down to commands and dispatch them to the lower layers of the system.

The Integrated Planning, Execution and Monitoring (IPEM) (Ambros-Ingerson and Steel 1988) system is also a *plan-exec* framework that proposes an architecture specifically designed to handle execution failures and unexpected events. This is done with both local repair capabilities and the interleaving of planning and execution. Local repair is achieved at the control level using "IF-THEN" rules that, given an unwanted state defined in "IF", define a behavior to handle it in "THEN". In the context of missing information, the authors propose to interleave planning and execution to gather missing information that could be obtained by executing an action. Therefore, they use a partial plan representation that can be instantiated at runtime based on the observation of the

execution.

The cost of replanning can also be significantly reduced by synthesizing plans of shorter length. This is the approach taken by the CASPER/ASPEN (Chien, Knight, et al. 2000) system, which incrementally searches a plan on a small timescale to be more responsive to goal updates and unexpected events. The planning process is continuous, and is designed to take into account activities that have already begun execution when trying to optimize the scheduling of activities that are not yet committed. The Automated Scheduling and Planning ENVIRONMENT (ASPEN) is particularly suitable for replanning, since it is based on an iterative algorithm that allows anytime planning, either starting from its simplest definition as a set of goals, or replanning from a previous plan in which broken causal links should be replaced. However, the repair capabilities can only commit to one conflict at a time, which is limiting when multiple conflicts need to be dealt with, and leaves the system without a valid plan for an extended period of time. The Continuous Activity Scheduling, Planning, Execution, and Replanning (CASPER) proposes a balance between constant adaptation to changes in the system and reactivity of the system, the latter of which is strongly dependent on planning search time.

Cypress (D. Wilkins and K. Myers 1995) proposes the use of an intermediate language between a planner (SIPE-2 (D. Wilkins 1988)) and a supervisor (PRS (Ingrand, Chatila, et al. 1996)). It uses the ACT formalism (D. E. Wilkins and K. L. Myers 1997) to specify the model of the agent and as the interlanguage between the two deliberation systems. Since the planner and the supervisor use the same intermediate language, most deliberation features are easier to implement: events and reports can be specified with ACT.

The Continuous Planning and Execution Framework (CPEF) system (K. Myers 1999) extends Cypress's framework in several ways, one of which is to relieve the supervisor of the orchestration of activities and to use a higher-level plan manager that adapts plans with respect to the overall operation of the system.

1.2.3.3 Unified planning and acting models

When blending *Planning* and *Acting*, one might think about the models used by both functions. Indeed, traditionally the literature separates planning models from acting models. Planning systems often rely on *descriptive models*, which make explicit the dynamics of a system, and the *operational model*, which are programs executed on the robotic platform. The *descriptive model* is a formal model that should allow reasoning, and is used to systematically explore the state space of the system. The role of the *operational model* goes beyond the simple execution of a sequence of commands, and provide control structures that adapt the behavior based on the context and that are more robust to unexpected events. In the approaches studied here, some systems make the choice to use common models for planning and acting. This is particularly interesting because traditionally, planning and acting systems are developed separately, along with their models. However, some systems are de facto designed to integrate *Planning* and *Acting*, and choose to rely on a single model for both deliberation functions. Typically,

IxTeT-Exec takes advantage of its shared subplan to facilitate the interaction between the temporal planner and the executive.

CSP approaches To reason on time, many approaches relied on CSP models to guide an acting system using a temporal planner. In such configuration, a temporal planner provides a temporal plan that can be constrained at runtime by the acting engine. Among the systems using this approach, we can mention IxTeT-Exec (Lemai-Chenevier 2004), and the heirs of RAX-PS (Muscettola, Nayak, et al. 1998): IDEA (Muscettola, Dorais, et al. 2002) and T-REX (Py, Rajan, and McGann 2010). The acting engine provides the values of execution of the uncontrollable timepoints and the free variables that are not fully instantiated by the planner. Therefore, since the execution model remains in a CSP form, the running choices can be propagated at runtime to verify the consistency of the constrained plan with respect to the assumed bounds of the variables for a given plan; the opposite triggers plan repair. Since some timepoints are uncontrollable, e.g. the end time of an action, CSP approaches can model the plan as a Simple Temporal Network with Uncertainty (STNU) in which both controllable and contingent timepoints and constraints can be represented (Vidal and Ghallab 1996). In such a representation, a STNU is said to be *dynamically controllable* if there exists a set of dynamic execution strategy that assigns controllable timepoints such that all constraints are satisfied. This assumes that the values of contingent timepoints preceding the controllable steps are known.

While these systems aim to share common structures for *Planning* and *Acting*, the RAX spin-off Reactive Model-based Programming Language (RMPL) is one of the most action-oriented CSP-based approaches. It provides refinement, instantiation, time, non-determinism, and plan repair capabilities. RMPL unifies the representation of the planning, acting, and monitoring systems in two models: a *hierarchical constraint-based automata* representing the *system model* in which both nominal and fault state transitions are modeled; a *control model* defined with reactive programming constructs. The programs in RMPL are transformed into a Temporal Plan Network (TPN), an extension of Simple Temporal Network (STN) with symbolic constraints and decision nodes. A TPN can be used to synthesize a plan by finding a path in the network that satisfies both the temporal and symbolic constraints. Several works have been proposed both to extend the formalism of TPNs and to improve its use:

- reduce plan size to improve dynamic execution (Conrad, Shah, and Williams 2009),
- add error recovery, temporal flexibility, and conditional execution to TPNs based on a distribution of its assumed duration (Effinger, Williams, and A. Hofmann 2010),
- add probabilistic notions to TPN to represent weak and strong consistency for which algorithms can check their consistency (Santana and Williams 2014),
- as for STNUs, Temporal Plan Network with Uncertaintys (TPNUs) (Levine and Williams 2014) add the notion of uncertainty for contingent decisions.

Behavior Tree (BT) approaches Several works aim at merging *Planning* and *Acting*, using Behavior Trees (BTs) as acting models. BTs (Iovino et al. 2022) are widely used to define Artificial Intelligences (AIs) in video games, and have been increasingly used to design robot behaviors in the last decades. They are often used to define the capabilities used by the execution layer to refine sequences of abstract tasks generated by, for example, a task planner. A BT is defined as a directed tree composed of *root*, *child*, *parent*, and *leaf* nodes. The execution of a node is triggered by receiving a *tick*, which can be propagated to one or more children. The execution of a node returns either the state *success*, *running* or *failure*. A leaf node is called an execution node and can be either (i) an *Action*, which returns *running* on the first receipt of a *tick* and *success* once the command was successfully executed on the platform, or (ii) a *Condition*, which returns *success* if its proposition is true in the current context, *failure* otherwise. Other non-leaf nodes are used to define the control flow of the BT. The primary definition of BT distinguishes three non-leaf nodes:

- *Sequence*, which executes n children nodes in sequence, failing if at least one child fails, otherwise succeeding,
- *Parallel*, which checks all its children at once, and fails if at least one child fails, otherwise succeeds,
- *Fallback* which executes one node at a time, succeeding immediately if at least one child node succeeds and failing if all children fail.

Going back to the unification of *Planning* and *Acting*, the work on extended Behavior Trees (eBTs) (Rovida, Grossmann, and Krüger 2017) tries to unify scripted and planned procedures, where the extended formalism adds STRIPS-like effect annotations to the primitive eBT. The *skills* used by the actor are composed of primitive eBT and have a direct mapping to the actions produced by the planner. Once the actions of the plan have been expanded to form a complete eBT, post-processing is performed to optimize the overall execution time of the BT by organizing independent nodes in parallel and rearranging the sequence of actions produced by the planner.

HTN planning has been used in a hybrid approach to allocate abstract tasks among multiple nodes, and relies on BT to refine the abstract tasks into lower-level commands to act (Neufeld, Mostaghim, and Brand 2018). Since HTN and BT are both hierarchically structured, the interface between the planner and the actors is more accessible. Such a hybrid approach showed better performance than pure HTN deliberation, which requires a lot of replanning in a dynamic environment, where BT is by design reactive to handle local contingencies without requiring replanning at a higher level of abstraction. Similarly, a BT can be used to fully define the behavior of an agent, where a said BT is updated at runtime by continuous planning using the Hybrid Backward-Forward (HBF) algorithm (Colledanchise, Almeida, and Ögren 2019). Belief BTs (BBTs) are another extension of BT (Safronov, Colledanchise, and Natale 2020) from which policies can be synthesized by a planner to act in partially observable environments.

The literature on BT offers many contributions, especially in defining robot behavior. They are often compared to Finite State Machines (FSMs), and are said to be easier

to design, more human-readable, which makes them more suitable for debugging and analysis. However, they lack the expressive power of explicitly programmed procedures.

Logic-based approaches Rather than manually detailing the specification of capabilities, some approaches have proposed to define an agent’s capabilities based on logical rules of the world, and then use inference mechanisms to generate the behaviors. In such approach, the logical rules and inference mechanisms can be used to mimic planning capabilities in the sense that they can synthesize new behaviors to adapt to the various situation. Here we focus on the situation calculus approach, whose best representative is the GOLEX system (Hähnel, Burgard, and Lakemeyer 1998). GOLEX is an execution system for the Golog planner (Levesque et al. 1997); both systems rely on knowledge defined in the situation calculus, but do not use the similarity of their model to integrate advanced interleaving of *Planning* and *Acting*. The Platas system (Claßen et al. 2012) extends the capabilities of Golog-based systems to integrate other planners that outperform Golog at generating sub-plans and that can benefit from a PDDL model that can be mapped directly to the situation calculus. ReadyLog (Ferrein and Lakemeyer 2008) extended the usability of Golog by allowing the integration of calling procedures into Golog plans, making it a language in which the interleaving of imperative code and situation calculus makes explicit nondeterministic decisions that should be handled by planners. It can be defined as an *acting language* in which deliberation functions can be used to adapt the behavior of the agent while retaining the advantages of procedural behavior. Following this approach, Golog++(Mataré et al. 2021) both extends the temporal semantic of Golog and proposes a formalized interaction between the abstract model used by Golog and a platform that uses a different formalism.

Automata and Petri nets approaches Among the formal models used to define acting models, Petri nets have been used in several works to define robotic behavior as they naturally encompass synchronization and sequencing of different sensorimotor processes, and in particular to represent multi-robot plans (Ziparo et al. 2011). Petri nets can be found at the functional level (F.-Y. Wang et al. 1991) to simulate and verify the components of a system.

ASPiC (Lesire and Pommereau 2018) proposes to define *skills* in the form of Petri nets that can be combined using operators to model sequence, branching, and concurrency. Following this line of work, new works proposed the RobotSkill (RS) language to define *skills* from which *descriptive models* can be automatically derived (Lesire, Doose, and Grand 2020). The *skills* of the autonomous agent are defined as a *SkillSet*. Extensions of the approach propose the use of verification tools on formal models that can be automatically generated from the *SkillSet* (Albore et al. 2023), which is particularly useful in critical robotic systems. The same toolbox can be used to analyze the behavior of the executive and functional layers to detect undesired behaviors in non-nominal states and provide useful insights for correcting the *SkillSet* to facilitate *skill* modeling.

A work using IxTeT as a planner binds each action of a plan to a corresponding user-defined Finite State Automata (FSA) that specifies the states in which an action can

be (Chatilla et al. 1992). The model still takes advantage of a hierarchical representation in which low-level actions are refined using procedures.

Expanding PDDL for execution Among the various acting systems that exist, the CLIPS rule-based production system (Wygant 1989) has been used to respond to triggered rules by executing *skills*. A rule is triggered by an event. In CLIPS, once a rule is triggered, it is added to an agenda. Each rule of the agenda is then addressed with an appropriate skill. Recent work presents the CLIPS Executive (CX) (Niemueller, T. Hofmann, and Lakemeyer 2019), which proposes to build on CLIPS to define an acting system for which PDDL planners can be used to guide the system. The execution models are derived from the same PDDL file, for which PDDL has been extended to allow execution annotations, allowing a unified model for both planning and acting. CLIPS has been successfully deployed on the RoboCup Logistics League (T. Hofmann et al. 2021), for which it handles the multi-agent environment with a resource allocation system that activates a goal only if all required resources are available, improving the agents’ goal commitment and preventing them from pursuing a goal that is not applicable in the current context. Upon a goal execution request, a PDDL planner is called to locally plan the goal and verify its applicability given the current world state. However, planning does not seem to take into account the execution of concurrent goals, especially those that share the same resources and thus cannot be reasoned on independently.

Planning with operational models Rather than extending the expressiveness of *descriptive models*, the approach taken by the Refinement Acting Engine (RAE) seeks to use directly the *operational models* to plan. RAE (Ghallab, Nau, and Traverso 2016) uses its hierarchical operational model for both *Acting* and *Planning*. The extensions brought to RAE aim to guide the deliberation of the acting system using a planner that can directly use the hierarchical operational model. Similar to Propice-Plan (Despouys and Ingrand 2000) and its *Anticipation Module*, the planner refines ahead to inform the acting engine on which method is preferable to refine a given task. Essentially, such a planner simulates multiple methods to select the one that maximizes some utility, such as the efficiency of the method. Two planners have been proposed:

- RAEPlan (Patra, Ghallab, et al. 2019) is an anytime planning system that uses Monte Carlo Tree Search (MCTS) algorithms to sample the results of low-level commands in each method. The result of the command sampling allows the methods to be ranked by two metrics: their cost (defined as the sum of the cost of the commands) and their efficiency (the inverse of the cost).
- UPOM (Patra, Mason, Kumar, et al. 2020), which extends RAEPlan to take into account the nondeterministic result of commands in the selection of a method. By design, UPOM benefits from learned heuristics to speed up the search in the MCTS:
 - Learn π maps a tuple (τ, ξ) of a task and a state to a method and is used when the acting system does not have time to find a method,

- Learn π_i returns the best instantiation of arbitrary parameters for a method in a given state ξ ,
- LearnH gives a heuristic for branching in UPOM.

While they propose asymptotically optimal planning approaches, both planners cannot guarantee the long-term validity of the choices they propose, since they rely on the sampling of operational models.

Other approaches based on the RAE algorithms have been proposed to combine *Acting* and *Planning*:

- The simplest, Run-Lookahead (Ghallab, Nau, and Traverso 2016), calls a lookahead planner whenever a new action should be performed, and executes the first action of the returned plan online. In this way, the acting system copes with changes in the environment at each step of execution. However, it seems that calling the planner every time an action should be executed is unnecessary, and could be done only when needed, and this is what Run-Lazy-Lookahead (Ghallab, Nau, and Traverso 2016) proposes, by looking ahead only when the plan is finished, or the current plan is no longer considered feasible by a plan validator.
- Recent work by Bansod, Nau, et al. 2021 proposes to extend Run-Lazy-Lookahead to a hierarchical representation of the agent’s behavior, using the power of the hierarchical task and goal planner IPyhop. Thus, the planner can control an acting engine based on a hierarchical representation of the agent’s capabilities. Thanks to its compliant interface, the user of IPyhop can easily replan from the middle (Bansod, Patra, et al. 2022), improving the responsiveness of the actor. This is made possible by the improvements of IPyhop over GTPyhop (Nau, Bansod, et al. 2021), and in particular the hierarchical structure of the returned plan as a tree simplifies replanning triggered by the actor, which needs only a pointer to the part of the hierarchical tree that failed, and gives a head start to find a new suitable plan. However, the entire tree that has been executed will potentially be replanned, and even unnecessary parts of the plan that were not affected by the previous failure. IPyhopper (Zaidins, Roberts, and Nau 2023) tackles this problem by repairing only the necessary parts of the plans, and has demonstrated faster results than IPyhop on similar domains.

On another aspect, RAE may face problems in scaling up when it comes to its online refinement of tasks. Dec-RAE (Li, Patra, and Nau 2021) is a formalization of a decentralized version of the RAE algorithms to distribute the deliberation across multiple agents, which should solve some problems related to deploying such algorithms on a large scale.

Relying on a single model facilitates the exchange of information between deliberation systems, and thus produces systems with more advanced features. In addition to the improved features of the deliberation system, it is convenient for the roboticist to define only a single model, rather than specifying dedicated models for each deliberation system. This not only simplifies the configuration of the robotic platform, but also

prevents semantic mismatches that could arise from representing the same capability in different models.

1.2.4 Summary

Deliberation is still an active area of research. In particular, *Acting* could benefit from a formalization of its deliberation capabilities and models. This is also highlighted by Nau, Ghallab, and Traverso (2015), which proposes to formalize *Acting* in the same way that it has been done to advance research in *Planning* with the PDDL language (McDermott et al. 1998). This has been partially proposed in the book *Automated Planning and Acting* (Ghallab, Nau, and Traverso 2016), which formalizes the model of an agent that is suitable for both *Planning* and *Acting*.

By using a similar representation for *Planning* and *Acting*, it should be easier to integrate the two functions into a larger deliberation system. Indeed, even though both systems are often studied separately in the literature, they almost always end up working together to propose a mixed deliberation function.

Indeed, an acting system should be coupled with an automated planner to provide lookahead capabilities to an otherwise reactive system. Such systems therefore require a combination of:

- Reactive capabilities for acting to adapt the behavior of individual robots or the entire fleet to new requirements and contingencies,
- Explicit deliberation capabilities, e.g., symbolic planning, aimed at optimizing the global behavior of the system, e.g., to minimize exploitation costs or meet deadlines.

One obstacle to the integration of planning algorithms into acting systems is the discrepancy in the model they use. Most approaches use a *descriptive model* for the planner and a *operational model* for the acting system. Specifying different models requires the robot programmer to master modeling both, and to avoid semantic mismatches between the two models. In addition, interaction between the two systems requires constant translation between their internal representations and a commonly used language to exchange information.

Different approaches have tackled this problem by relying on a common representation of plans and execution on which both reasoning systems can work. We can summarize them as follows:

- Use the planning model as the reference and automatically extend its expressiveness for execution,
- Rely on the action model to plan directly,
- Use an intermediate representation that can be easily translated into acting and planning models.

Among the proposed approaches, the second has recently been explored in RAE (Ghallab, Nau, and Traverso 2016), a generic acting algorithm that by design can be coupled with any planning engine to guide its deliberation. The planning algorithms would rely on *operational models*, using sampling methods to anticipate the long-term effects of executing capabilities (Patra, Ghallab, et al. 2019; Patra, Mason, Ghallab,

et al. 2021). Other approaches proposed a unique modeling language from which both *descriptive models* and *operational models* can be derived (Lesire, Doose, and Grand 2020), but lacked the versatility and computational power proposed in procedure-based approaches. In contrast to other planner-centered approaches, in which the planner is the deliberation driver, the acting engine is the top-level deliberation system. The first implementation of RAE relies on Python as a modeling language whose expressive power is difficult to match by planning languages.

However, as powerful as Python is, its formal analysis is challenging due to its large core language, which makes it difficult to use more advanced planning techniques, such as temporal and hierarchical planning techniques, which requires formal models in the form of, e.g., chronicles as in IxTeT (Ghallab and Laruelle 1994) and *Aries* (Godet and Bit-Monnot 2022). Chronicles have been shown to be suitable for *Planning* and *Acting* in IxTeT-Exec (Lemai-Chenevier 2004), which relies on a *Plan-Exec* deliberation strategy whose reactivity is therefore determined by the speed of the planner. Moreover, the *skill* models used by IxTeT-Exec rely on a CSP approach, which can be challenging for the roboticist.

1.3 Contributions of the thesis

In this thesis we present the Operational Model Planning and Acting System (OMPAS), a new acting engine for deliberation systems. OMPAS (Turi and Bit-Monnot 2022a) can be seen as an extension of RAE (Ghallab, Nau, and Traverso 2016), targeting multiple agents and concurrent activities. It includes the following features:

- A refinement based acting system that supports concurrency and resource allocation,
- An acting language from which planning models can be automatically extracted,
- A unified framework for *Acting* and *Planning* with time and resources,
- The guidance of the decisions of the acting engine through continuous planning.

OMPAS extends RAE by adding native support for concurrency in operational models, and managing the interleaving of parallel tasks thanks to a system that handles the acquisition of shared resources. The goals of this extension are to *(i)* provide native support for concurrency in the system, *(ii)* explicitly model resources, and *(iii)* clearly identify decision points in the operational model.

OMPAS uses a hierarchical representation of agent capabilities, here defined with SOMPAS, a dedicated acting language based on Lisp. The Lisp dialect is based on the Scheme (Moretti 1979) variant for defining operational models for RAE systems. The language supports asynchronous execution, resource management, and explicit decision points.

The identification of decisions should enable the guidance of the acting engine’s decisions, e.g. by automated planning techniques. Unlike previous implementations

of RAE, this custom language enables automated analysis of the operational model to extract planning domains from acting domains and use dedicated tools to guide the acting engine through decision points. The analysis is facilitated by the restricted core of the language, which is limited to a few primitives.

We can then rely on existing planners, in particular *Aries* (Godet and Bit-Monnot 2022), to select the best method to accomplish each task faced by the system. The planning system can guide OMPAS in a continuous fashion and guide the runtime deliberation in the acting engine. The *Aries* planner searches and optimizes a plan indefinitely for all current missions, taking into account state updates and new missions.

Some of the work has been presented at international conferences and workshops.

- The paper “Extending a Refinement Acting Engine for Fleet Management: Concurrency and Resources” was presented at the International Conference on Tools for Artificial Intelligence (ICTAI) 2022 (Turi, Bit-Monnot, and Ingrand 2023) and presented extensions to the RAE algorithm to better handle concurrency and resource management in multi-agent scenarios.
- The workshop papers “Guidance of a Refinement-Based Acting Engine with a Hierarchical Temporal Planner” and “Enhancing Operational Deliberation in a Refinement Acting Engine with Continuous Planning” were presented at the Integrated Acting, Planning and Execution (IntEx) workshops of the International Conference on Automated Planning and Scheduling (ICAPS) in 2022 and 2023. Both papers proposed new ways of interleaving *Planning* and *Acting* to improve deliberation in a RAE-like system.

1.4 Motivating example: Gripper-Door

To illustrate the approach of this thesis, we present the following running example. Our scenario is inspired by the *Gripper* domain proposed in the first editions of the International Planning Competition (IPC).

Within this domain, a robot named *Robby* is tasked with moving *balls* around different *rooms*. The robot has two grippers - the *left* and the *right* - which allow it to grab and move objects at will. We extend our initial definition by adding doors between rooms. Robby’s grippers can be used to open and close doors. We call this new domain *Gripper-Door*.

State formulation We represent the world of Gripper-Door as a collection of state variables, where each variable is defined by a parameterized state function $sf(p_1, \dots, p_n)$ and a corresponding value v . The following are the state functions of Gripper-Door:

- $at\text{-}robby() \rightarrow \text{room}$ returns Robby’s current location,
- $pos(?b: \text{ball}) \rightarrow \text{location}$ returns the location of an object, which can be either a room or Robby.

- $\text{carry}(?g: \text{ gripper}) \rightarrow \text{ball}$ returns the value of the gripper, which is either *empty*, indicating that Robby is not carrying anything, or the object that Robby is currently carrying.
- $\text{connects}(?r1: \text{ room}, ?d: \text{ door}, ?r2: \text{ room}) \rightarrow \text{boolean}$ indicates that door $?d$ connects rooms r_1 and r_2 .
- $\text{opened}(?d: \text{ door}) \rightarrow \text{boolean}$ indicates whether a door is open or not.

Robby's capacities Robby can perform various actions that are applicable in states where their preconditions are met. These actions result in a new state in which the effects have been applied. Note that Robby is limited to performing only one action at a time. Below is a list of actions that Robby can perform:

- $\text{move}(?from: \text{ room}, ?to: \text{ room}, ?d: \text{ door})$ to move from Robby's current position, indicated by $?from$, to $?to$ through door $?d$:
 - preconditions: $\text{at-robbly}() = ?from$
 $\text{connects}(?from, ?d, ?to)$,
 $\text{opened}(?d)$
 - effects: $\text{at-robbly}() = ?to$
- $\text{pick}(?r: \text{ room}, ?b: \text{ ball}, ?g: \text{ gripper})$ where $?r$ denotes the location where Robby should pick up $?b$ using the gripper $?g$:
 - preconditions: $\text{at-robbly}() = ?r$,
 $\text{pos}(?b) = ?r$,
 $\text{carry}(?g) = \text{empty}$
 - effects: $\text{pos}(?b) := \text{robbly}$,
 $\text{carry}(?g) := ?b$
- $\text{drop}(?r: \text{ room}, ?o: \text{ object}, ?g: \text{ gripper})$ where $?r$ is the location where Robby should drop $?b$ with the gripper $?g$.
 - preconditions: $\text{at-robbly}() = ?r$,
 $\text{pos}(?o) = \text{robbly}$,
 $\text{carry}(?g) = ?o$
 - effects: $\text{pos}(?o) := ?r$,
 $\text{carry}(?g) := \text{empty}$
- $\text{open}(?d: \text{ door}, ?r: \text{ room}, ?g: \text{ gripper})$ opens $?d$ when Robby is in one of the connected rooms and $?g$ is empty.
 - preconditions: $\text{at-robbly}() = ?r$,
 $\text{connects}(?r, ?d, ?r_2)$,
 $\text{carry}(?g) = \text{empty}$
 - effects: $\text{opened}(?d) := \text{true}$
- $\text{close}(?d: \text{ door}, ?r: \text{ room}, ?g: \text{ gripper})$ closes $?d$ when Robby is in one of the connected rooms and $?g$ is empty.

```

preconditions:  at-robby(=?r,
                 connects(?r, ?d, ?r2)
effects:       opened(?d) := false

```

In this context, a problem is defined by an initial state consisting of the initial values of the state variables, and a goal as a target value for a given state variable. The goals are expressed as a set of new positions for the balls.

Example 1.1: Gripper-Door in a tiny house

Consider the following problem, where Robby is navigating through a house consisting of a bedroom, a kitchen, and a living room (denoted l_r). Four balls, b_1, b_2, b_3, b_4 , are located in different rooms, and Robby's goal is to move them all to the bedroom and back to the living room.

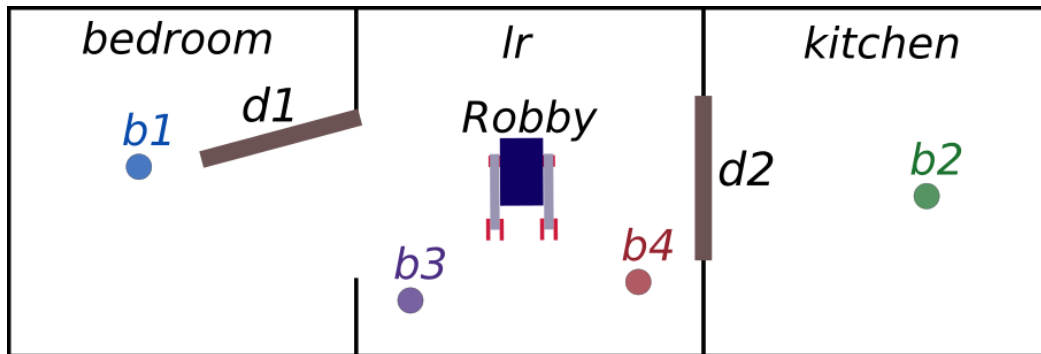


Figure 1.3: Schematic representation of the initial state.

The initial state is defined as follows:

```

connects( $l_r, d_1, bedroom$ ):= true
connects( $kitchen, d_2, l_r$ ):= true
opened( $d_1$ ):= true
opened( $d_2$ ):= false
at-robby():=  $l_r$ 
carry( $left$ ):= empty
carry( $right$ ):= empty
pos( $b_1$ ):= bedroom
pos( $b_2$ ):= kitchen
pos( $b_3$ ):=  $l_r$ 
pos( $b_4$ ):=  $l_r$ 

```

The goals are defined below:

pos(b_1)=	<i>bedroom</i>
pos(b_2)=	<i>bedroom</i>
pos(b_3)=	<i>bedroom</i>
pos(b_4)=	<i>bedroom</i>
at-robby()	l_r

1.5 Outline

The resulting contributions are presented in the following chapters.

- The **Chapter 2** introduces the architecture and algorithms of the acting engine OMPAS.
- The **Chapter 3** introduces the dedicated programming and acting language SOM-PAS, focusing on the syntax, semantics, and evaluation of programs defined with the language.
- The **Chapter 4** presents the guidance of OMPAS using planning capabilities. The techniques used to generate planning models and to integrate the planning system with the acting system are also developed in this chapter.
- The **Chapter 5** provides an evaluation of the deliberation capabilities of the system on the simulated *Gripper-Door* domain and the factory simulator *Gobot-Sim*.
- The **Conclusion** provides a synthesis and an analysis of the work presented, followed by suggestions for improvements and avenues for further exploration.

Architecture and Algorithms of the Operational Model Planning and Acting System (OMPAS)

Contents

2.1	Introduction	29
2.2	The Refinement Acting Engine (RAE)	30
2.2.1	State representation	30
2.2.2	Hierarchical operational model	30
2.2.3	Primitives of the operational models	32
2.2.4	Example of hierarchical operational model in the Gripper-Door domain	32
2.2.5	Deliberation of vanilla RAE	34
2.2.6	Limitations of vanilla RAE	39
2.3	The Operational Model Planning and Acting System (OMPAS)	44
2.3.1	Architecture of OMPAS	44
2.3.2	Platform Manager (PM)	45
2.3.3	State Manager (SM)	47
2.3.4	Execution Manager (EM)	48
2.3.5	Resource Manager (RM)	51
2.3.6	Acting Manager (AM)	53
2.3.7	PLanner Manager (PLM)	55
2.4	Conclusion	57

2.1 Introduction

In this chapter, we present the Operational Model Planning and Acting System (OMPAS), an acting system based on the deliberation algorithms and models of the Refinement Acting Engine (RAE) (Ghallab, Nau, and Traverso 2016). It extends RAE by reasoning on the interleaving of tasks thanks to explicit resource modeling in operational models, and lookahead techniques to schedule the access to these resources.

In this chapter, we introduce the deliberative capabilities of RAE, beginning with a description of the features, models, and algorithms proposed in the book *Automated Planning and Acting* (Ghallab, Nau, and Traverso 2016). We then examine the limitations of RAE algorithms and survey other RAE-based approaches to identify areas for improvement that will be targeted in OMPAS. Finally, this chapter presents the features, architecture, and deliberation capabilities of OMPAS.

2.2 The Refinement Acting Engine (RAE)

RAE is an automated acting system, developed by Ghallab, Nau, and Traverso (2016) that executes abstract tasks given by an operator through commands sent to a robotic platform. RAE then monitors the execution of the commands. To accomplish tasks, RAE uses a set of executable models called *skills*, which can be used to refine a task. The *skills* in RAE are organized hierarchically and are referred to as hierarchical operational models.

When attempting to perform a given task, RAE first identifies a relevant *skill* in the current state and then executes it. If a *skill* fails, a retry mechanism selects another *skill* to continue performing the task. A *skill* may fail if, for example, a condition is not met or a command fails. RAE has the ability to perform multiple tasks simultaneously.

Figure 2.1 illustrates the interaction of RAE with other systems in a robotic architecture. It receives tasks from an *Operator* and then sends execution commands to the *Robotic Platform*, which represents the lower-level stacks of the robotic architecture, such as the executive and control layers in a three-tier architecture. The Robotic Platform is responsible for updating the observed state that RAE uses to perform its deliberation. The Robotic Platform has the ability to send events such as "battery low". These events are handled by RAE in the same way as tasks, which involves refining them with *skills*.

Similar to the Procedural Reasoning System (PRS) (Ingrand, Chatila, et al. 1996), *skills* are implemented as executable procedures that are defined using an interpretable language. This language allows the use of traditional programming constructs, such as branching and looping, and includes primitives for accessing the RAE's deliberation features.

2.2.1 State representation

The reasoning functions of RAE rely on the system state. The state of RAE includes the observed state, which is read-only and returned by the *Robotic Platform*, along with an internal state that RAE and its executed procedures can update.

2.2.2 Hierarchical operational model

The *skills* used by RAE are defined in a hierarchical operational model on which the acting system can reason from. This model is formalized here as an Acting Domain (A_{Δ}). It is a tuple (W, C, T, M) , where:

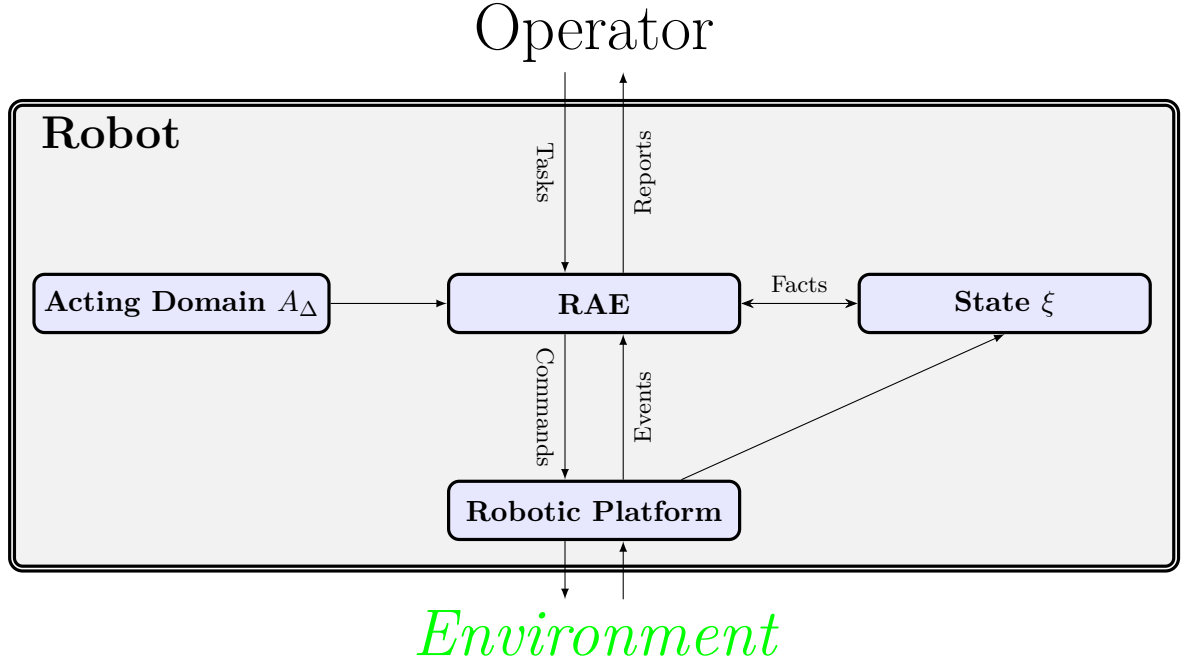
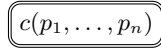


Figure 2.1: An architecture implementing RAE.

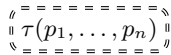
- W is the set of *state functions* representing the state of the world. Each state function is of the form $sf(p_1, \dots, p_n)$, where sf is the label of the state function, and p_i are the parameters.
- C is the set of *commands*. They are directly executable on the *Robotic Platform*. In the *Gripper-Door* domain of the Example 1.1, the commands are *pick*, *drop*, *move*, *open*, *close*.
- T is the set of *tasks*, the high-level capabilities of the system, e.g. *put balls in the bedroom*, *close all doors* in the *Gripper-Door* domain of the Example 1.1.
- M contains the methods that can refine a given task $\tau \in T$, where m represents a method that encapsulates a *skill* that can perform τ . A method is characterized by the following attributes:
 - The refined task τ ,
 - A list of *parameters* that contains the parameters of τ and may contain additional ones,
 - A set of *preconditions* that determine the applicability of the method in a given state,
 - The *body* of the method: a program that is executed to refine τ when the method is selected.

HTN representation of a hierarchical operational model We can represent a hierarchical operational model using the HTN formalism, where:

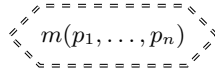
- a command is represented as a node with double simple rounded edges:



- a task is displayed as a node with double dashed rounded borders:



- a method is a node with double-dashed chamfered borders:



You can find in Figure 2.2 examples of HTNs for a proposition of operational model for the *Gripper-Door* domain.

2.2.3 Primitives of the operational models

The procedures used in RAE are not bound to any particular programming language. However, it must provide these operational primitives that can be used within the body of a method:

- $\text{READ}(sf, p_1, \dots, p_n)$ returns the value of the state variable $sf(p_1, \dots, p_n)$.
- $\text{WRITE}(sf, p_1, \dots, p_n, v)$ assigns the value v to the state variable $sf(p_1, \dots, p_n)$.
- $\text{EXEC}(c, p_1, \dots, p_n)$ requests the execution of the command $c(p_1, \dots, p_n)$ on the *Robotic Platform*.
- $\text{REFINE}(\tau, p_1, \dots, p_n)$ uses RAE to find a *skill* to refine the task $\tau(p_1, \dots, p_n)$.

2.2.4 Example of hierarchical operational model in the Gripper-Door domain

The Example 2.1 shows how hierarchical operational models are used to define complex behaviors composed of lower-level *skills* in the *Gripper-Door* domain.

Example 2.1

Take the Example 1.1. We define the task $go2(?r)$, the goal of which is that *Robby* lands at $?r$. We define the methods $noop(?r)$ and $recur(?r, ?a, ?n, ?d)$. The latter is used to recursively move *Robby* to a neighboring room until it reaches $?r$; $noop(?r)$ ends the recursion. The methods of $go2(?r)$ are defined as follows:

```

                noop(?r)
parameters:    ∅
preconditions: at-robby() = ?r
              body: nil
                recur(?r, ?a, ?n, ?d)
parameters:    ?a: room
              ?n: room
              ?d: door
preconditions: at-robby()= ?a
              ?a ≠ ?r
              connects(?a, ?d, ?n)
body:         Refine(move2room, ?a, ?n, ?d)
              Refine(go2, ?r)

```

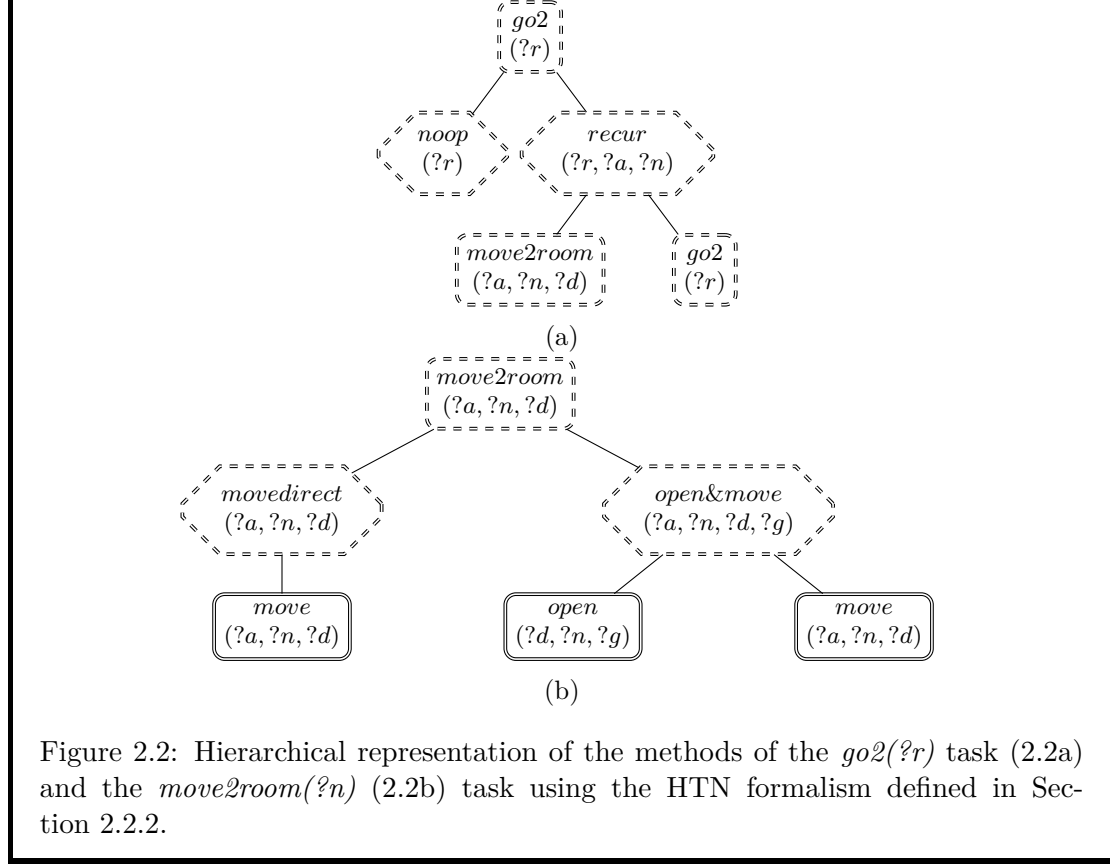
Similarly, the methods of the *move2room(?n)* task are defined as follows:

```

                movedirect(?a, ?n, ?d)
parameters:    ∅
preconditions: at-robby()= ?a
              ?a ≠ ?r,
              connects(?a, ?d, ?n)
              opened(?d)
body:         Exec(move, ?a, ?d)
                open&go(?a, ?n, ?d, ?g)
parameters:    ?g: gripper
preconditions: at-robby()= ?a
              ?a ≠ ?r
              connects(?a, ?d, ?n)
              ¬opened(?d)
body:         Exec(open, ?d, ?a, ?g)
              Exec(move, ?a, ?d)

```

The hierarchical representation of these capabilities is shown in Figure 2.2.



2.2.5 Deliberation of vanilla RAE

RAE deliberates on the model of the robotic agent defined in A_{Δ} . Its deliberation process is composed of several functions that have been presented in (Ghallab, Nau, and Traverso 2016; Patra, Mason, Ghallab, et al. 2021). We present them again, with a terminology more adapted to the formalism presented in this dissertation. Some functions have been simplified, but the deliberation remains the same. We study them to show how RAE handles the execution of high-level tasks, and to identify the limitations of these functions.

Terminology The functions of RAE use the following terminology:

- τ is an abstract task or event to be refined based on the agent model defined in A_{Δ} .
- ξ is a snapshot of the state of the system that can be obtained with the function GET-STATE. It aggregates both the perceived state given by the *Robotic Platform*, and the state internal of RAE. The state is represented as a collection of facts of the form $sf(p_1, \dots, p_n) = v$, where $sf(p_1, \dots, p_n)$ is a state variable, and v is its value. It can have predicates such as $opened(d_1) = true$, and supports more complex state variables such as $loc(r_1) = (2.1, 3.5)$, which represents the location of a robot r_1 in 2D coordinates.

Algorithm 2.1 Main function of vanilla RAE.

```

1: function MAIN
2:    $Agenda \leftarrow \{\}$ 
3:   loop
4:     for all new task or event  $\tau$  to be addressed do
5:        $\xi \leftarrow \text{GET-STATE}$ 
6:        $m \leftarrow \text{SELECT}(\xi, (\tau, \text{nil}, 1, \emptyset))$ 
7:       if  $m = \emptyset$  then  $\text{OUTPUT}((\tau), \text{"failed"})$ 
8:       else
9:          $\text{PUSH}(Agenda, \langle (\tau, m, 1, \emptyset) \rangle)$ 
10:      for all  $\sigma \in Agenda$  do
11:         $\xi \leftarrow \text{GET-STATE}$ 
12:         $\sigma \leftarrow \text{PROGRESS}(\sigma, \xi)$ 
13:        if  $\sigma = \emptyset$  then
14:           $\text{REMOVE}(Agenda, \sigma)$ 
15:           $\text{OUTPUT}(\tau, \text{"Success"})$ 
16:        else if  $\sigma = \text{retrial-failure}$  then
17:           $\text{REMOVE}(Agenda, \sigma)$ 
18:           $\text{OUTPUT}(\tau, \text{"Failure"})$ 

```

- $Output(msg)$ is the function used to report to the *Operator*.
- $Applicable(\tau, \xi)$ is the set of applicable methods to refine a given task τ . A method is applicable in ξ if its *preconditions* are true.
- We denote m as the *body* of a method that refines a given task τ . Here, a *body* is a sequence of instructions that can be a RAE primitive such as $Exec(c, p_1, \dots, p_n)$, or a generic programming construct for control flow. $m[i]$ is the i^{th} statement of a program.
- The *Agenda* is the list of all refinement stacks of all the tasks currently addressed by RAE. It is initialized to an empty list $\{\}$. A refinement stack $\sigma \in Agenda$ is a LIFO list of refinements $\rho = (\tau, m, i, \text{tried})$ where τ is an event, task, subtask, or a goal; m is an instance of a method that matches τ ; i is a pointer to the current step in the *body* of m initialized to *nil* (no step was executed); and *tried* is a set of instances of methods that failed to accomplish τ , initialized to \emptyset .

The deliberation algorithm of RAE relies on four principal functions: *Main* handles the initial reception of all tasks and events, and their monitoring, $Select(\xi, \rho)$ finds a suitable method given ρ and ξ , $Progress(\xi, \sigma)$ executes the operational model of a given refinement stack σ , and $Retry(\sigma)$ handles the failure of an operational model.

Main The MAIN function (Algorithm 2.1) continuously loops over two things:

1. It addresses any new tasks or events τ that the acting system should face by first finding an initial method m to refine τ using the $\text{SELECT}(\xi, (\tau, \text{NIL}, 1,))$ (line 6)

Algorithm 2.2 Select function of Vanilla RAE.

```

1: function SELECT( $\xi, \rho$ )
2:   ( $\tau, m, i, tried$ )  $\leftarrow \rho$ 
3:    $Candidates \leftarrow APPLICABLE(\xi, \tau) \setminus tried$ 
4:   if  $Candidates \neq \emptyset$  then
5:     return ARBITRARY( $Candidates$ )
6:   else
7:     return  $\emptyset$ 

```

function. If m is valid, a new refinement stack $\langle (\tau, m, O, \emptyset) \rangle$ is added to the *Agenda* (line 9).

2. It executes and monitors any refinement stack σ of the *Agenda* by calling the $PROGRESS(\xi, \sigma)$ function. All tasks are progressed in a round-robin fashion, i.e. one instruction per program at a time.

Select The $SELECT(\xi, \rho)$ function (Algorithm 2.2) finds a suitable method to refine τ . It first generates the set of applicable methods $Candidates$ by removing the already tried methods (line 3) from $Applicable(\xi, \tau)$. The $Applicable(\xi, \tau)$ set contains the methods defined in the Acting Domain (A_Δ) that can refine τ . If $Candidates$ is not empty, an arbitrary method is selected and returned (line 5), otherwise \emptyset is returned.

Progress The function $PROGRESS(\xi, \sigma)$ (Algorithm 2.3) executes and monitors the operational model m on top of σ based on ξ . It executes the current statement $m[i]$. It handles the $EXEC(COMMAND)$, $WRITE(ASSIGNMENT)$, and $REFINE(\tau)$ primitives. Depending on the primitive, we have the following behavior:

- $EXEC(COMMAND)$: RAE requests the execution of *command* on the platform (line 13) and monitors its status until it is completed (line 4), which can result either in a *Success* or a *Failure*.
- $WRITE(ASSIGNMENT)$: The internal state of the system is updated with the function $UPDATESTATE(ASSIGNMENT)$.
- $REFINE(\tau')$: τ' is refined with the method m' , which is obtained by calling $SELECT(\xi, \rho)$. If $m' = \emptyset$, τ' is considered a *Failure*. The failure is propagated to m and handled with the $RETRY(\xi, \sigma)$ function. If $m \neq \emptyset$, a new refinement $(\tau', m', 0, \emptyset)$ is pushed to σ and σ is returned.

Upon success of the execution of an instruction, the next step of the method that RAE should address is addressed. This statement is found using $NEXT(\xi, \sigma)$, which takes into account other programming constructs in m , such as branching. If there are no more statements in m , the method is considered a *Success* and the refinement is removed from σ .

Algorithm 2.3 Progress function of Vanilla RAE.

```

1: function PROGRESS( $\xi, \sigma$ )
2:   ( $\tau, m, i, tried$ )  $\leftarrow$  TOP( $\sigma$ )
3:    $instruction \leftarrow m[i]$ 
4:   if  $instruction$  is an already triggered action then
5:      $status \leftarrow$  EXECUTION-STATUS( $instruction$ )
6:     if  $status = running$  then return  $\sigma$ 
7:     else if  $status = failed$  then return RETRY( $\sigma$ )
8:     else if  $status = done$  then return NEXT( $\xi, \sigma$ ) return
9:   else if  $instruction = Write(assignment)$  then
10:    UPDATESTATE( $assignment$ )
11:    return NEXT( $\xi, \sigma$ )
12:   else if  $instruction = Exec(command)$  then
13:    EXECCOMMAND( $command$ )
14:    return  $\sigma$ 
15:   else if  $instruction = Refine(\tau')$  then
16:     $\xi \leftarrow$  GET-STATE
17:     $m' \leftarrow$  SELECT( $\xi, (\tau', nil, 0, \emptyset)$ )
18:    if  $m' = \emptyset$  then
19:      return RETRY( $\sigma$ )
20:    else
21:      return PUSH( $\sigma, (\tau', m', 1, \emptyset)$ )
22:
23: function NEXT( $\xi, \sigma$ )
24:   repeat
25:     ( $\tau, m, i, tried$ )  $\leftarrow$  POP( $\sigma$ )
26:     if  $\sigma = \langle \rangle$  then return  $\langle \rangle$ 
27:   until  $i < LAST(m)$ 
28:    $j \leftarrow$  NEXTSTEP( $m, \xi$ )
29:   PUSH( $\sigma, (\tau, m, j, tried)$ )
30:   return  $\sigma$ 

```

Algorithm 2.4 Retry function of Vanilla RAE.

```

1: function RETRY( $\sigma$ )
2:   ( $\tau, m, i, tried$ )  $\leftarrow$  POP( $\sigma$ )
3:   PUSH( $tried, m$ )
4:    $\xi \leftarrow$  GET-STATE
5:    $m' \leftarrow$  SELECT( $\xi, (\tau, nil, 1, tried)$ )
6:   if  $m \neq \emptyset$  then
7:     PUSH( $\sigma, (\tau, m', 0, tried)$ )
8:     return  $\sigma$ 
9:   else if  $\sigma \neq \emptyset$  then return RETRY( $\sigma$ )
10:  elsereturn retrial-failure

```

Retry The $\text{RETRY}(\sigma)$ function is introduced in Algorithm 2.4. $\text{RETRY}(\sigma)$ calls $\text{SELECT}(\xi, (\tau, \text{NIL}, 1, \text{tried}))$ again to find a new method for τ . The set *tried* of failed methods is extended by the last tried method m (line 3). If no more methods are applicable, τ is considered a *Failure*. The *Failure* is propagated to the top level, which will again resort to RETRY to handle the problem, and so on until either a new method is found, or the top level of the refinement stack is reached.

Example of execution of a task in vanilla RAE To illustrate the reasoning behind RAE, the Example 2.2 shows how the *skills* defined in Example 2.1 can be used by RAE to accomplish a task.

Example 2.2: Execution of a task in the gripper-door domain in vanilla RAE

Let us take the problem given in Example 1.1 and perform the task $\tau_1 = \text{go2}(\text{kitchen})$ using the *skills* defined in Example 2.1. The state ξ is defined in part as follows:

$$\begin{aligned} \text{at-robby}() &:= l_r \\ \text{connects}(\text{kitchen}, d_2, l_r) &:= \text{true} \\ \text{opened}(d_2) &:= \text{false} \\ \text{carry}(\text{left}) &:= \text{empty} \\ \text{carry}(\text{right}) &:= \text{empty} \end{aligned}$$

RAE asynchronously receives the task τ_1 , and calls $\text{SELECT}(\xi, (\tau_1, \text{NIL}, 1, \emptyset))$ to find a suitable method for τ_1 . $\text{Applicable}(\xi, \tau_1)$ is the set of instances of methods that might be executable based on the *skills* associated with τ_1 :

- $\text{noop}(\text{kitchen})$ is not applicable because $\text{kitchen} \neq l_r$,
- $m_1 : \text{recur}(\text{kitchen}, l_r, \text{bedroom}, d_1)$ is applicable,
- $m_2 : \text{recur}(\text{kitchen}, l_r, \text{kitchen}, d_2)$ is applicable.

$\text{Applicable}(\xi, \tau_1)$ chooses an arbitrary method from $\{m_1, m_2\}$:

- If it chooses m_2 , Robby goes to the kitchen by first opening d_2 and then moving. This possibility is shown in Figure 2.3a. The first attempt to open the door failed due to, e.g. due to a problem with the *left gripper*. RAE tries again with the *right gripper*, which succeeds. The failed method is shown in red on the HTN. Finally it calls $\text{go2}(\text{kitchen})$ again, the only applicable method being $\text{noop}(\text{kitchen})$.
- If it chooses m_1 , Robby moves to the bedroom, and recursively calls $\text{go2}(\text{kitchen})$ until it ends up in the kitchen. This scenario is shown in Figure 2.3b.

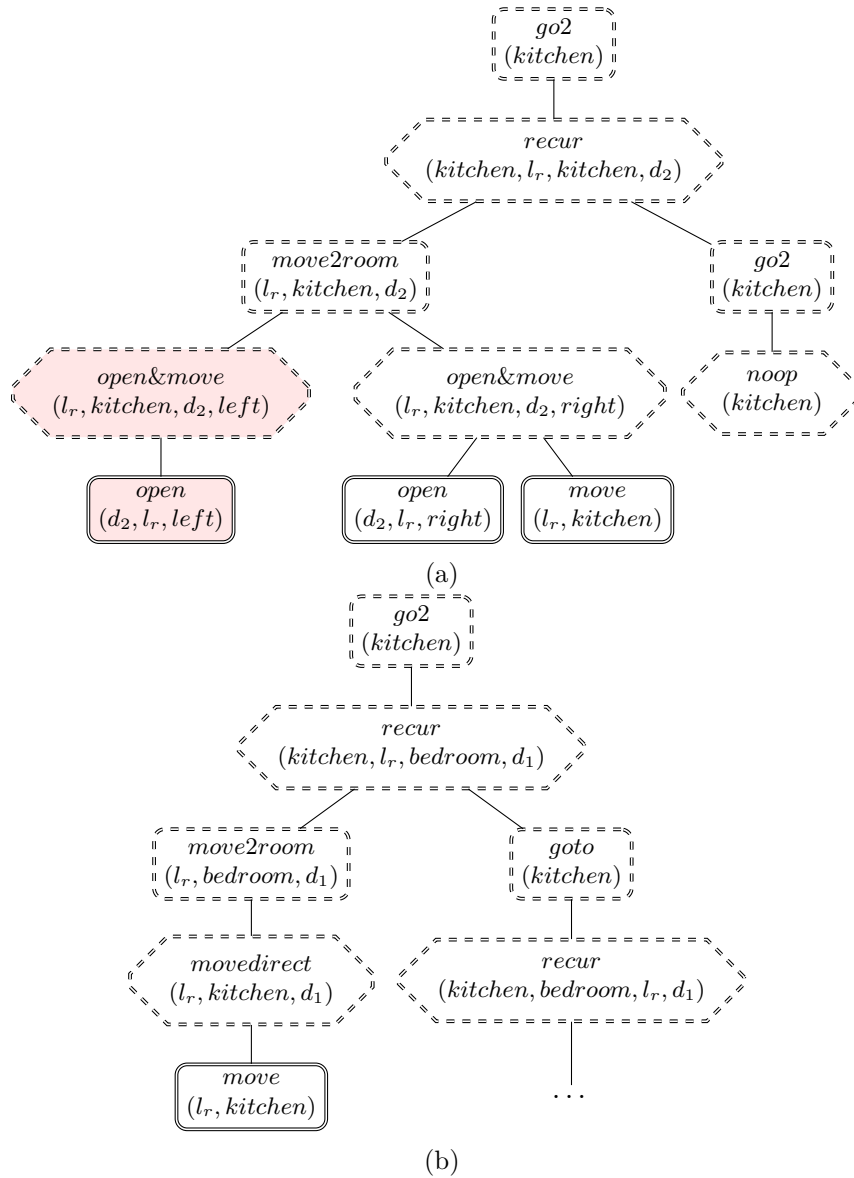


Figure 2.3: Different traces of the deliberation of RAE in function of the method chosen to refine the task τ_1 .

2.2.6 Limitations of vanilla RAE

Lack of lookahead capabilities In Example 2.2, it should be obvious that the task τ_1 is not guaranteed to finish, since there is a non-zero probability that *Robby* will go back and forth between l_r and *bedroom*, and this indefinitely. Without lookahead capabilities, RAE cannot guarantee the success of a task in bounded time, especially in these recursive cases. It is the responsibility of the programmer to handle this kind of scenario.

By design, the deliberations of RAE can be guided by any heuristic. In particular,

RAE could use planning techniques to make informed decisions based on their long-term effects.

To address these problems, the version of RAE proposed by Patra, Mason, Ghallab, et al. (2021) integrates UPOM, an anytime planner that guides the refinement of tasks into methods. For a given task τ , UPOM suggests refining it with the most efficient method based on current and future states.

Efficiency takes into account the total cost of the commands, and the probability of their failure. Therefore, it handles the non-deterministic results of commands, and should limit the number of retry.

Therefore, even if it does not guarantee the termination of the refinement, UPOM should guide RAE to limit the number of recursions in Example 2.2.

Limited interleaving of concurrent tasks One of the key features of RAE is its ability to handle multiple tasks simultaneously. Each task is associated with a particular refinement stack in the agenda, and each refinement stack proceeds one instruction at a time in a round-robin fashion. This situation is illustrated in Example 2.3, which we will examine. In this example, RAE must deal with two tasks received at the same time.

Example 2.3

Let us define a new task $place(?o, ?r)$, which places an object in the given room. Three methods can be defined to perform this task:

```

place_noop(?o, ?r)
parameters:   $\emptyset$ 
preconditions:  pos(?o) = ?r
body:  nil

move&drop(?o, ?r, ?g)
parameters:  ?g: gripper
preconditions:  carry(?g) = ?o
body:  Refine(go2, ?r)
       Exec(drop, ?o, ?r, ?g)

pick&drop(?o, ?r, ?g, ?p)
parameters:  ?g: gripper, ?p: location
preconditions:  pos(?b) = ?p,
               ?p  $\neq$  Robby
body:  Refine(go2, ?p)
       Exec(pick, ?o, ?p, ?g)
       Refine(go2, ?r)
       Exec(drop, ?o, ?r, ?g)

```

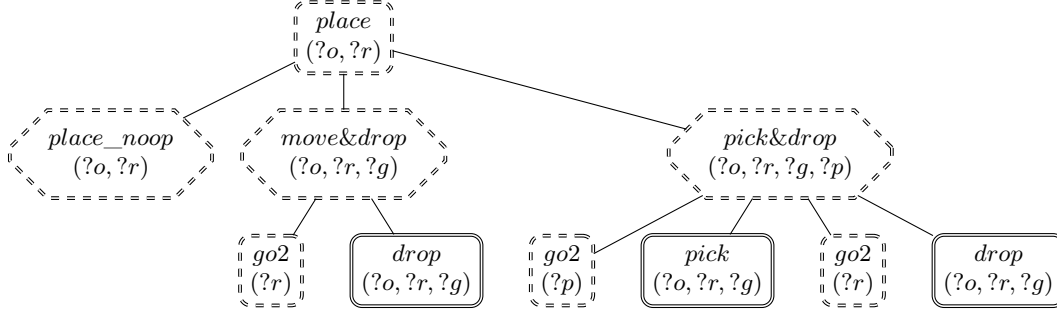


Figure 2.4: Hierarchical representation of the *skills* of $place(?o, ?r)$ in the RAE formalism.

Now, let us take the problem given in Example 1.1, where the initial state is defined as:

```

at-robbly() :=  $l_r$ 
carry(left) := empty
carry(right) := empty
pos( $b_1$ ) := bedroom
pos( $b_2$ ) := kitchen
connected( $l_r, d_1, bedroom$ ) := true
connected(kitchen,  $d_2, l_r$ ) := true
opened( $d_1$ ) := true
opened( $d_2$ ) := false
...

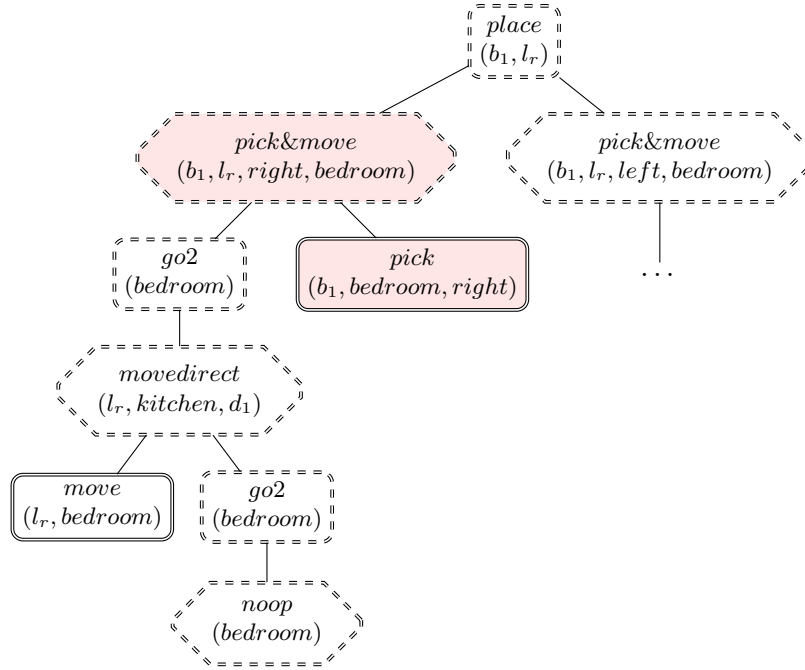
```

In this context, RAE should face two tasks at the same time: $\tau_1 : place(b_1, l_r)$ and $\tau_2 : place(b_2, l_r)$. We assume that the refinement of $\tau_3 : go2(bedroom)$ is informed by a planner, e.g. UPOM, and thus RAE makes the most efficient decisions. The hierarchical representation of the execution trace of tasks τ_1 and τ_2 is shown in Figure 2.5a and Figure 2.5b, respectively. The execution trace is as follows:

Executed command	High-level task
$move(l_r, bedroom)$	$place(b_1, l_r)$
$\times open(d_2, l_r, left)$	$place(b_2, l_r)$
$move(bedroom, l_r)$	$place(b_2, l_r)$
$\times pick(b_1, bedroom, right)$	$place(b_1, l_r)$
...	...

These traces show that several commands failed because their preconditions were not met at the time RAE requested their execution. This is due to the execution of τ_1 and τ_2 , which create threats to their respective execution. This means that if the *skill* currently running to refine one task requires the execution

of a command, it may pose a threat to the execution of the *skill* of another task, rendering the *skill* inapplicable.



(a)

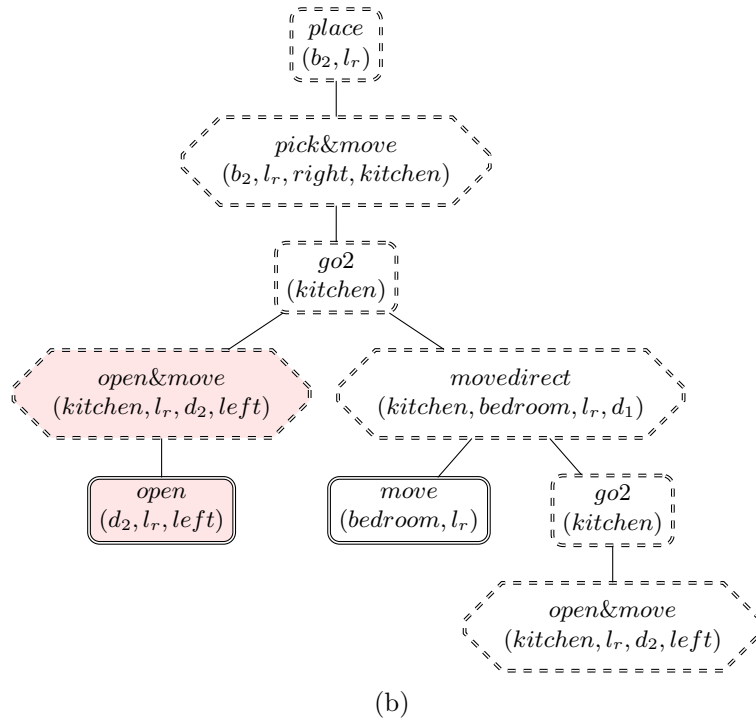


Figure 2.5: Hierarchical decomposition of two tasks that are executed in parallel in vanilla RAE. Processes in red failed due to inapplicable commands or failure of a subtask in a method.

The example we have just examined highlights a fundamental flaw in RAE’s design: it is ill-equipped to effectively handle multiple tasks competing for the same resources. This shortcoming leads to avoidable failures that could be mitigated by explicitly requesting exclusive access to shared resources, for example, by specifying the exclusive use of a robot like Robby for the task of fetching a ball.

Vanilla RAE, as originally defined, lacks an explicit resource concept. In vanilla RAE, resources can be managed in two primary ways: either by representing resource status with a state variable, such as *lock(Robby)*, indicating whether Robby is in use, or by implementing locking mechanisms directly in method bodies using programming language constructs.

In either case, the system should have the ability to reactively manage exclusive resource usage. However, the burden of implementing these utilities falls on the robot programmer, which is less than ideal. Ideally, the acting engine should provide native resource handling solutions, relieving the programmer of this responsibility.

Furthermore, both resource management approaches limit the system’s ability to reason about resource access and, consequently, task execution. In the first case, the state variable simply indicates that a resource is in use, without specifying which task is using it. In the second case, the use of programming language mutexes restricts the acting system’s control over the timing of resource access.

In addition, vanilla RAE lacks the ability to induce concurrency in the body of

methods. Concurrency is limited to the high-level tasks.

Ideally, the acting system should have the ability to orchestrate the concurrent progress of multiple tasks, making informed decisions such as prioritizing one task over another based on urgency or the potential to optimize overall system performance.

Therefore, our goal is to build on RAE and introduce a new acting system that not only inherits RAE's deliberation features for high-level task execution through a hierarchical operational model, but is also better adapted to handle concurrent tasks in parallel. This involves active deliberation about their coordinated execution, with two primary goals: (i) avoiding failures, deadlocks, and dead-ends that can result from simultaneous execution, and (ii) improving system performance by reasoning about task execution.

2.3 The Operational Model Planning and Acting System (OMPAS)

The Operational Model Planning and Acting System (OMPAS) is a novel acting system that uses hierarchical models to perform multiple high-level tasks simultaneously by executing commands on a robotic platform. In its role as RAE, the system accomplishes high-level tasks by decomposing them using the *skills* defined in the hierarchical operating model, and manages *skill* failures by resorting to the RETRY function to find another *skill* to perform the task it is refining. In addition, OMPAS manages the acquisition of resources required by *skills* during execution. The system explicitly signals the acquisition of a resource to manage concurrent access to limited resources. OMPAS supports procedures that can be executed concurrently within their bodies. OMPAS can also delay the instantiation of parameters until they are needed in the method body. Decision points are made explicit, allowing the acting system to use any heuristic to inform its decision, such as considering the long-term effects of the decisions, similar to RAE.

In this section, we present the architecture of OMPAS and its deliberation functions, along with their algorithms.

2.3.1 Architecture of OMPAS

The OMPAS architecture shown in Figure 2.6 is based on the RAE architecture (see Figure 2.1). Like RAE, OMPAS gets its assignments in the form of tasks from the *Operator* or events from the *Robotic Platform*.

The internal architecture of OMPAS consists of several *managers*, each responsible for one or more deliberation functions of OMPAS. These managers operate in parallel and can communicate with each other through asynchronous channels.

The **Platform Manager (PM)** is responsible for executing commands on the *Robotic Platform* and receiving perceived states and events from it. It monitors the command status for each request returned by the platform. The Platform Manager is the only manager that interacts with the lower-level stack of the robotic system, e.g.

the executive layer.

The **State Manager (SM)** manages both the perceived state provided by the Platform Manager and the internal state that OMPAS can reason about. It provides secure read and write access to the state and provides reasoning functions, such as the monitoring of boolean expressions whose values depend on the state.

The **Execution Manager (EM)** is responsible for executing and monitoring tasks and events that OMPAS must respond to. Each task corresponds to a procedure that the Execution Manager executes and monitors. Tasks are received directly from the operator, while events are assigned by the Platform Manager. The role of the Execution Manager can be compared to the main loop of RAE (see Algorithm 2.1).

The **Resource Manager (RM)** provides secure access to shared resources among multiple procedures. It handles multiple resource access requests and can reposition the access queue for a particular resource to increase system efficiency. The Resource Manager can make conscious decisions about which requests to approve in order to maximize a metric, such as minimizing the total time required to complete all ongoing tasks.

The **Acting Manager (AM)** is responsible for refining tasks into methods and instantiating free variables within method bodies. It also keeps track of all deliberative decisions made by OMPAS. This deliberation trace takes the form of an *Acting Tree*, which represents the hierarchical execution of the processes. The *Acting Tree* is used to track the refinement of tasks and replaces the *Agenda* used in vanilla RAE.

The purpose of the **PLanner Manager (PLM)** is to interface with an external planner that informs OMPAS of choices to make during procedure execution. The planner advises OMPAS on preferred methods for task refinement, instantiation of method parameters, and sharing of resources among tasks that request them. The Acting Manager is used as a middleman between the reactive execution of the Execution Manager and the predictive execution of a planner. We only describe its interfaces with other managers here, more details on its internal workings are given in the Chapter 4.

2.3.2 Platform Manager (PM)

The Platform Manager (PM) abstracts the deliberation of OMPAS from the specifics of a *Robotic Platform*. We define a *Robotic Platform* as any robotic system that is capable of *perceiving* its environment, and *acting* in its environment by executing commands.

The Platform Manager uses a standard interface to communicate with the robot platform, achieved through the use of generic middleware or frameworks such as the ROS (Quigley et al. 2009).

Command execution requests can be sent from the Platform Manager to the *Robotic Platform*, and their execution status can be received and transmitted to other modules of OMPAS. The *Robotic Platform* transmits perceived states and events to OMPAS through the Platform Manager.

Execution of commands The *Robotic Platform* should accept requests to *execute* commands, and *cancel* them. An execution request is received from the Execution

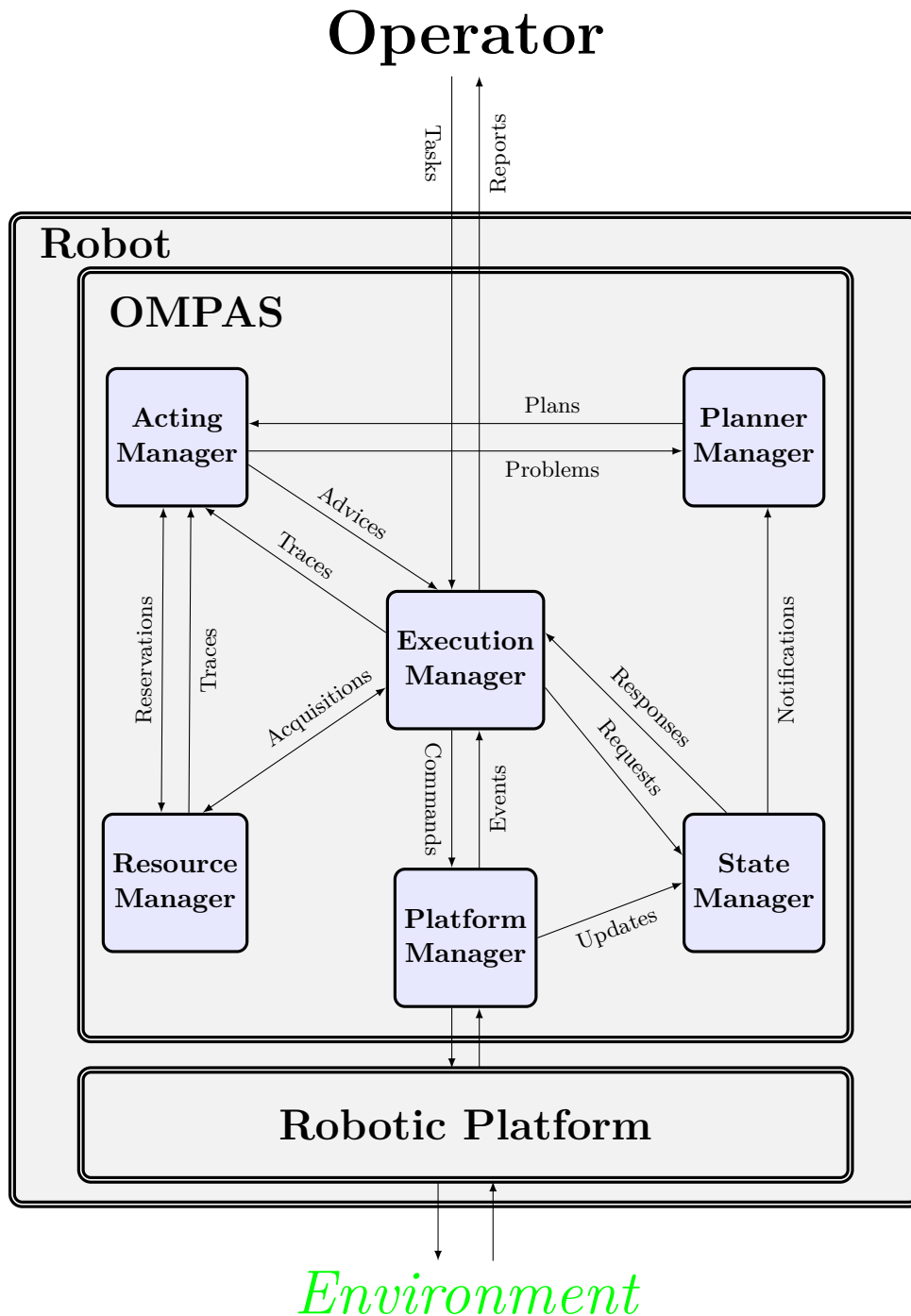


Figure 2.6: Architecture of the Operational Model Planning and Acting System. The Execution Manager, State Manager and Platform Manager inherit from the vanilla RAE architecture (see Figure 2.1). The Resource Manager, Acting Manager and Planner Manager bring additional deliberation features to look ahead.

Manager (EM) following the call to EXEC(COMMAND) in the body of a method. For a given command, EM waits for the command request to complete in order to resume the execution of a skill. Each execution request is associated with a unique *ID*, which can be used by the EM to monitor the execution of a particular command.

In OMPAS, the execution cycle of a command follows a specific pattern. First, the command enters a *pending* state until the *Robotic Platform* processes it. Next, as the command is executed, the platform may send progress information to OMPAS to estimate the time remaining. When the command is complete, the *Robotic Platform* sends a notification of either *Success* or *Failure*. Regardless of the outcome, the command is considered completed. Finally, the *skill* responsible for the command can resume its execution.

A cancel request can be sent at any time during the execution cycle of a command. It can be requested in the body of a method by calling the CANCEL(ID) primitive. The cancel request is propagated to the *Robotic Platform*, which returns a *Cancelled* state that can either be caught in the body of the method, or treated as a failure by OMPAS and therefore handled by the retry mechanism of the acting engine.

Update of the state The observation of the robot state of OMPAS is managed by the Platform Manager, which interacts with the other parts of the *Robotic Platform* to collect the information from the perception and knowledge management systems. The facts are transmitted by the Platform Manager to OMPAS through a unique interface. The State Manager is directly provided with the facts by the Platform Manager, which securely manages the update of the system status. The facts must be formatted so that the State Manager can support them.

2.3.3 State Manager (SM)

The State Manager (SM) aggregates all the facts that OMPAS can use to reason about a single state ξ . Certain facts come from the Platform Manager, while others reflect internal facts used by the methods executed in OMPAS.

In OMPAS, facts are defined by the date they were recorded by the State Manager and a key-value pair. The key in this pair is a state variable expressed in the format $sf(p_1, \dots, p_n)$, where sf is the name of the state function and p_i are the instantiated parameters. The state variable can take any value compatible with the procedural language used to define the body of the method.

OMPAS distinguishes between *static* and *dynamic* state functions:

- Static state functions have a value that cannot change, i.e. their value remains constant throughout execution. They can be used to represent static properties, such as the name of a robot.
- Dynamic state functions, on the other hand, represent state variables whose values can change during execution, and multiple facts representing the same state variable can be recorded during execution. These state variables can be used to represent the position of a robot that is to evolve at runtime.

The state ξ consists of two parts: the *internal* state, which contains all OMPAS-scoped facts and can be updated by methods, and the *perceived* state, which relies solely on updates from the Platform Manager. A fact can only be defined in a subset of the state, and facts in the perceived state are read-only.

Functions to manipulate the state The value of a fact is retrieved with the function `READ(sv)`, which returns the last value v of sv . The internal state can be changed with the following functions:

- `WRITE(sv, v)` adds the key-value pair (sv, v) to the internal state of OMPAS. If the key is already defined, the value will be overwritten.
- `ERASE(sv, v)` removes the key-value pair (sv, v) from the state, meaning that sv is no longer defined.

Fluent monitoring In OMPAS we define a fluent as an expression whose value depends on at least one state variable. Such a fluent can be used to wait for or monitor a certain state of the system, e.g. waiting for a robot to be at a certain location in order to continue the execution of a *skill*. To do this, the State Manager includes a feature to monitor the value of a boolean fluent, fluents that evaluate to boolean values until their value becomes true. To access this feature, the body of a method can use the primitive `WAIT-FOR(fluent)`, where *fluent* is a boolean fluent. The functionality of `WAIT-FOR` is illustrated in the algorithm 2.5. The function sends a new boolean fluent to monitor, and gets back a *FluentID* that it can use to wait until the State Manager notifies it that the fluent has become true.

The Execution Manager uses the `MONITORFLUENTS` function to monitor one or more fluents simultaneously. The Algorithm 2.5 describes the operation. When a new boolean fluent f is requested for monitoring, it is added to the list of fluents to be monitored. A unique *FluentID* is assigned to f , which is sent back to the subscriber executed in the Execution Manager. Each time the state is updated, all fluents are re-evaluated with the updated state. If a fluent becomes true, it is removed from the list of fluents and the subscriber is notified. All propositions are re-evaluated regardless of the facts added to the state. The evaluation time is a function of $\mathcal{O}(|fluents| \times |f|)$ and can be problematic when monitoring many propositions at once. An improved algorithm could use the pattern-matching capabilities of the RETE (Forgy 1989) algorithm to verify fluents only when there is a change in the value of one of the state variables of which they are composed.

2.3.4 Execution Manager (EM)

The Execution Manager (EM) is responsible for overseeing the execution of all concurrent tasks. Unlike vanilla RAE, concurrent tasks are not processed in a round-robin fashion. Each new task is executed in its own thread.

Algorithm 2.6 represents the adaptation of the main algorithm to deal with asynchronous task execution. The execution of the function `EXEC-TASK(τ)` (presented in

Algorithm 2.5 Fluent monitoring process of OMPAS.

```

1: function WAIT-FOR(fluent)
2:   FluentID  $\leftarrow$  SENDFLUENT(f)
3:   WaitEndMonitoring(FluentID)
4: function MONITORFLUENTS
5:   Fluents  $\leftarrow$  {}
6:   loop
7:     for all new fluent posted (f, subscriber) do
8:       FluentID  $\leftarrow$  INSERT(Fluents, f)
9:       SENDBACKID(subscriber, FluentID)
10:    if STATEUPDATED then
11:       $\xi$   $\leftarrow$  GET-STATE
12:      for all (FluentID, f)  $\in$  Fluents do
13:        r  $\leftarrow$  EVAL(f,  $\xi$ )
14:        if r=true then
15:          REMOVE(FluentID)
16:          NOTIFYENDMONITORING(FluentID)

```

Algorithm 2.7) is spawned in a new thread (line 5). EXEC-TASK(τ) handles both the refinement of the task τ by the method m and the execution of the body of the method. Each thread is associated with a *handle*, which can be used to monitor the execution of a task τ . If the thread is marked as terminated (line 9), the result of the evaluation of τ is printed.

With this design, the language used in the body of the method should ensure thread safety. In particular, the side effects of the primitive should prevent any race conditions and provide safe access to shared memory.

Unlike the MAIN function of RAE, concurrent task execution relies on the system's thread scheduler, allowing for easier utilization of multiple CPUs. However, this reduces control over thread order, resulting in different task execution rates.

In RAE systems, the execution of *tasks* and *commands* are handled in different ways. The former relies on the acting engine and the latter relies on the Platform Manager (PM).

Command execution When a command c is to be executed in a method m , a request is sent to the Platform Manager and the Execution Manager holds the execution of m until c is finished. The execution cycle of the command has already been defined in Section 2.3.2.

Task execution The execution of a task resorts to the primitive EXEC-TASK(τ) presented in the Algorithm 2.7. First, all methods $Applicable(\xi, \tau)$ in the current state ξ are created. Then the function SELECT-METHOD(τ, M) selects a method from $Candidates = Applicable(\xi, \tau) \setminus tried$. By default, it selects the first method

Algorithm 2.6 Main algorithm of OMPAS in which each new task τ is executed in a dedicated thread. The initial refinement of τ is part of the function call *Exec-Task*(τ)(see Algorithm 2.7).

```

1: function MAIN
2:   Handles  $\leftarrow$  {}
3:   loop
4:     for all new task or event  $\tau$  do
5:        $h \leftarrow$  SPAWN(Exec-Task( $\tau, \emptyset$ ))
6:     for all  $h \in$  Handles do
7:       if TERMINATED( $h$ ) then
8:          $r \leftarrow$  RESULT( $h$ )
9:         OUTPUT( $r$ )
10:        REMOVE(Handles,  $h$ )

```

Algorithm 2.7 Adaptation of RAE’s procedure for executing a task τ . The procedure arbitrarily selects a method m that is applicable in the current state ξ and has not been tried before.

```

procedure EXEC-TASK( $\tau, \textit{tried}$ )
   $M_{app} \leftarrow$  APPLICABLE( $\xi, \tau$ )
   $Candidates \leftarrow M_{app} \setminus \textit{tried}$ 
   $m \leftarrow$  SELECT-METHOD( $\tau, Candidates$ )
  if  $m = \emptyset$  then ▷ No untried applicable method left
    return failure
   $res \leftarrow$  EXEC-BODY( $m$ )
  if  $res = \textit{failure}$  then ▷ Retry, with  $m$  forbidden
    return EXEC-TASK( $\tau, \textit{tried} \cup \{m\}$ )
  else
    return  $res$ 

```

of *Candidates*, but other systems can guide the selection, e.g. with a learned heuristic or an anytime planner such as UPOM (Patra, Mason, Kumar, et al. 2020). The body of the returned method is executed with the EXEC-BODY(m) function. If the result of the execution is a *Failure*, the acting engine tries to execute τ again by selecting a method from those that are still applicable and have not yet been tried. The function EXEC-TASK(τ) is called until either a method is a *Success* or no more methods are applicable. In the latter case, τ is an error.

This task execution procedure is the same whether τ is a top-level task or a subtask of a method. While the overall system remains simple, most of the complexity of the procedure lies in the EXEC-BODY(m) function, which is responsible for interpreting the program of the user-defined body.

2.3.5 Resource Manager (RM)

To ensure safe execution of concurrent procedures, OMPAS relies on its Resource Manager (RM), which guarantees secure access to resources. The Resource Manager handles multiple resource access requests and maintains a dedicated wait list for each resource. Consider the problem presented in Example 2.3, where the robot *Robby* is required to perform two simultaneous tasks, τ_1 and τ_2 . To ensure safe operation, both tasks must request exclusive access to *Robby* to perform their respective activities.

While mutexes could handle this, we propose a resource system that allows for non-unary resources and richer waitlist management than First In First Out (FIFO). The Resource Manager has the freedom to define its own allocation strategy to improve overall system performance without necessarily guaranteeing correct access to a resource.

In complex robotic systems, efficient resource allocation is critical to ensure robust behavior and optimize operations. In situations where different tasks must be prioritized, the Resource Manager can choose to reserve a portion of a resource for a potentially high-priority task. For example, reserving twenty percent of a medical supply robot's load can allow it to respond to an urgent request without having to unload or complete other deliveries before committing to the urgent one. The allocation strategy is designed to be guided by any heuristic, such as a scheduling system. Thus, the Resource Manager cannot guarantee the order in which resources will be accessed, so it is uncertain when a resource access request will be granted. In other words, a request could potentially experience arbitrary delays if the Resource Manager decides to delay granting its request.

Even if the Resource Manager guarantees secure resource access, it relies on well-programmed methods. This is similar to the mutex in other programming languages, where explicit access within the method body is required. However, we ensure that proper programming of the method body will guarantee no race conditions in resource acquisition.

The proposed resource design has two goals: to allow exclusive use of resources and to allow the Resource Manager to define an allocation strategy that meets the robustness and efficiency requirements of a robotic system.

2.3.5.1 Definition of a resource

We define a resource r as an object with an initial capacity C_{init} . A resource r can be acquired at time t with an amount $c \leq C_t \leq C_{init}$, where C_t is the current capacity of r . Upon acquisition, the Resource Manager ensures that no race condition occurs that would lead to overallocation, and the current capacity is immediately reduced by the acquired amount c .

We distinguish between *unary* and *divisible* resources:

- A unary resource can be acquired by one task at a time. It has an initial capacity of one unit, and only one unit can be requested.
- A divisible resource has an initial capacity of C_{init} . Any amount c of the resource can be acquired as long as $c \in [0, C_{init}]$.

When a resource is released, its capacity is increased by the amount borrowed.

2.3.5.2 Acquisition of a resource

When a resource r with amount c is requested by a procedure, the Execution Manager sends a request to the Resource Manager. If the request is valid, i.e. $c \leq C_{init}(r)$, then the Execution Manager holds the execution of the procedure until the acquisition is granted. The Resource Manager grants the acquisition according to its allocation strategy. The allocation strategy can be different depending on the robot domain.

Default allocation strategy based on priorities If no allocation strategy is defined for the Resource Manager, requests are granted as follows. If r is available and the queue is empty, the request is granted directly. Otherwise, the request is queued and given a priority.

Priorities are in descending order. Whenever a new request is added to the queue, the queue is re-sorted to accommodate the new request. If two requests have the same priority, the earliest request date takes precedence. The default priority is 0. With this priority system, urgent operations take precedence over accessing a resource, such as recharging a robot with critical battery levels.

Example 2.4 provides a runtime example in which asynchronous acquisition requests are received by the Resource Manager. The allocation strategy is based on the priority of each request.

Example 2.4

Let us take a medical supplies delivery drone that has a load limit of 20kg. It first receives a request to deliver 10kg of medical supplies, the priority is low, 5. The associated request r_1 is granted immediately and the current capacity C_t is reduced to 10kg. A second request r_2 requires 8kg of the space with a medium priority of 10. Since the strategy is to keep twenty percent of the load for high priority tasks, r_2 is queued until more space becomes available. A third request, r_3 , requests 12kg with a medium priority of 11. Like r_2 , r_3 is queued, but before r_2 due to its higher priority. A fourth request r_4 arrives with a high priority of 15, requesting 3kg. The fourth request is granted immediately, and the capacity is now 7kg. The task that requested r_1 is now terminated, meaning that 10kg of the load is free to be used again; r_3 is granted and the capacity is now 5kg; r_2 is still waiting...

The proposed design utilizes resources to enable secure concurrent execution of robotic programs across domains. This framework also provides an opportunity to apply high-level allocation strategies to optimize the overall robotic system strategy by considering shared resource access by robotic programs.

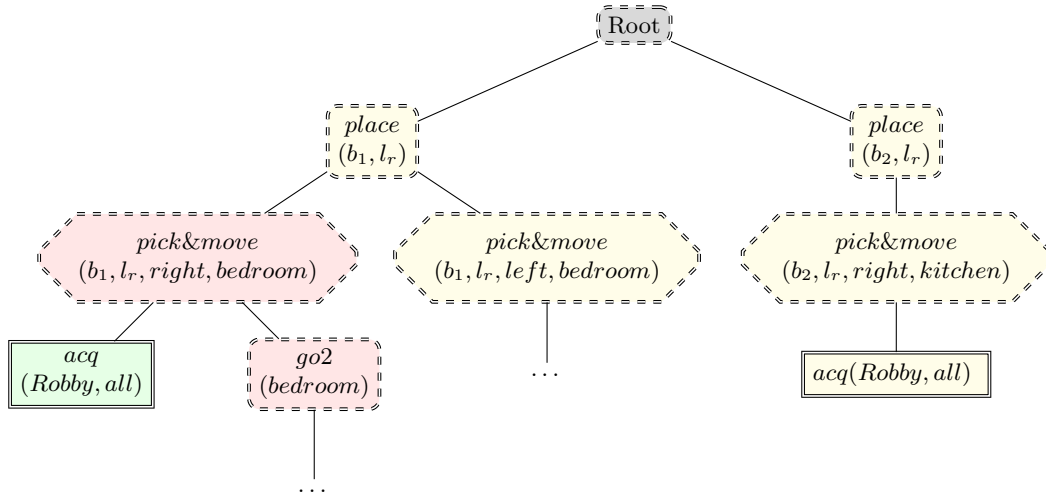


Figure 2.7: Example of an *Acting Tree* representing the execution of the task $t_1 = place(b_1, l_r)$ and $\tau_2 = place(b_2, l_r)$ of Example 2.3. The *Acting Tree* is only partially displayed. The methods of the tasks *place* use the new resource acquisition feature of OMPAS to request the exclusive use of *Robby* during the execution of the skill. Processes in red have failed, triggering a retry of the task using a different method. We have an example of a retry for task τ_1 .

2.3.6 Acting Manager (AM)

The Acting Manager (AM) is responsible for refining tasks and instantiating variables in the body of methods. The refinement and instantiation request is sent by the Execution Manager, which holds its execution until the Acting Manager responds. In order to carry out its deliberations, in particular to refine tasks, the Acting Manager should be aware of the choices made previously. In particular, when retrying a task, the Acting Manager should know what methods have been tried previously.

To facilitate access to the execution traces, the Acting Manager has an internal representation of the execution of programs on OMPAS. The execution of procedures is represented as a tree. The structure of the tree extends a HTN by adding the representation of the acquisition and instantiation of variables in the body of methods as sub-processes of a method. The tree is built using the information provided by the Execution Manager. We call it an *Acting Tree*, and more details are given in the Chapter 4. An example of an *Acting Tree* is shown in the Figure 2.7.

2.3.6.1 Selection of methods

At runtime, the selection of a method to refine a task τ is handled by the function `SELECT-METHOD(τ, M)` called in the function `EXEC-TASK(τ)` presented in Algorithm 2.7. The `SELECT-METHOD(τ, M)` uses the Acting Manager to select a suitable method, taking into account the methods applicable in the current state of the system and the methods previously tried. The candidates are obtained by computing

$Candidates = Applicable(\xi, \tau) \setminus tried$, the set of methods that can be used to refine the task τ . To choose a method among the candidates, several strategies are available in OMPAS. Like RAE, OMPAS can rely on an external program to provide a heuristic about which method to choose, e.g. by relying on a task planner. The design of OMPAS takes into account the possibility to plug in such a program. The following strategies are available in OMPAS.

Greedy selection The most basic way to refine a task is to select the first method from $Candidates$. This approach is deterministic.

Random selection Like vanilla RAE, OMPAS can refine τ by arbitrarily selecting m among the $Candidates$. This is a non-deterministic approach that may be more robust in some contexts.

Cost-based selection $Candidates$ can be sorted with a cost. The cost is calculated with the function $Cost(\xi, m, \lambda_{cost})$, which returns the cost of a given method. Once the cost of all candidates is computed, $Candidates$ is sorted in ascending order so that the method with the lowest cost is the first element of the list. Then the first element in the list is selected.

Planning-based selection In other versions of RAE, the acting system resorts to a planner to find the best method m to try to refine τ , taking into account the long-term effects of m .

We use the *Lazy – RefineAhead* approach, which refines the entire hierarchical network down to commands. It is similar to *Run – Refine – Ahead* (Bansod, Nau, et al. 2021). However, our approach relies on a planner in which time is explicit to represent concurrent execution in the body of the methods. It is *Lazy* because it uses the same hierarchical decomposition for the entire tree while the plan is valid.

In a previous work (Turi and Bit-Monnot 2022a), we introduced the $PlanSelectMethod(\tau, tried)$ (see Algorithm 2.8). The most important part of the algorithm is the call of the planner (line 7), which, based on the current state ξ , tries to find a plan that refines the task τ . This plan is then analyzed (line 8) to identify the method m used to refine τ . If this succeeds and the method is one of the allowed methods in $Candidates$, then the method is returned to the Execution Manager (EM) (line 10). Otherwise, the system may use some arbitrary heuristic to choose one of the allowed methods (line 12). The latter case might occur especially if the planner fails to find a plan within the allotted time. Note that if τ is a subtask of a method that was selected based on a plan π , then τ will also appear in that earlier plan. In this case, it is possible to check if the previous plan π is still valid, and if it is, avoid calling the planner to compute a new one (lines 5-6).

Algorithm 2.8 Plan-based method selection for a task τ .

```

1: function PLANSELECTMETHOD( $\tau$ , tried)
2:    $m \leftarrow \emptyset$ 
3:    $\xi \leftarrow \text{GET-STATE}$ 
4:    $Candidates \leftarrow \text{APPLICABLE}(\tau, \xi) \setminus \text{TRIED}(\tau)$ 
5:    $\pi \leftarrow \text{GET-PARENT-PLAN}(\tau)$ 
6:   if  $\pi = \emptyset$  or not IS-VALID( $\pi$ ) then
7:      $\pi \leftarrow \text{CALL-PLANNER}(\tau, \xi)$ 
8:    $m \leftarrow \text{GET-METHOD-FROM-PLAN}(\pi, \tau)$ 
9:   if  $m \neq \emptyset \wedge m \in Candidates$  then
10:    return  $m$ 
11:  else
12:    return ARBITRARY( $Candidates$ )

```

2.3.6.2 Arbitrary selection

OMPAS provides the ability to arbitrarily select a value from a set of options. The instantiation takes into account the state of the parameter at the moment of instantiation, not the state at the beginning of the execution of the method.

This feature is implemented in OMPAS by the function $\text{ARBITRARY}(set, \lambda)$, which selects an element $e \in set$, using λ as a heuristic for selecting e . The λ heuristic is optional. By default, the $\text{ARBITRARY}(set, \lambda)$ function returns the first element of the set.

The λ function is given by the robot programmer to improve the selection of the element given a domain. However, as for the other deliberation functions of OMPAS, the acting system is free to ignore λ and use its own heuristic to select e , e.g. by relying on a task planner that will be able to take into account the long-term effects of such a choice.

The Acting Manager has the responsibility for some key decision points of the acting system. The choice to keep these deliberation features from the Execution Manager is to decouple the execution from the deliberation, and to facilitate the integration of a planner that looks ahead of the execution to anticipate the decision points the Execution Manager will face. By keeping an internal representation of the execution trace, the Acting Manager can perform its deliberation as intended in refinement based acting engines, but also have a representation of the current state of the system that is more amenable to a task planner.

2.3.7 PLanner Manager (PLM)

The function of the PLanner Manager (PLM) is to interact with a planner to guide the deliberations of OMPAS.

The PLanner Manager creates a planning problem (P_{Π}) based on the *Agenda*, the system state (including both perceived and internal states), and the resource state (pro-

vided by the Resource Manager).

The task planner is used to anticipate the executable sequences that can be achieved from the current state of the system. Its primary function is to avoid deadlocks or dead ends. The planner should anticipate OMPAS's decisions to avoid such flows and inform about the necessary decision to be made. These informed decisions should result in safe execution flows, or even execution flows that improve overall system performance.

Once the task planner has found a solution, the PLanner Manager should analyze the plan and extract data that will guide the various reasoning processes of OMPAS.

The PLanner Manager is responsible for updating the planning problem at runtime based on new execution traces, facts, and changes in resource allocation.

Instantiation of the planner Each planner may have a different representation of the planning problem, and it is the role of the PLanner Manager to build it given the information accessible in OMPAS. To build a planning problem, the PLanner Manager has access to the execution traces of the system in the form of an *Acting Tree* given by the Acting Manager. The state is retrieved from the State Manager. Only the state variables that are involved in the planning model are retrieved, which reduces the size of the state encoding in the planning problem. The current state of the resources is also encoded in the planning problem so that the task planner can anticipate their acquisition.

Output of the planner The plan of the task planner is analyzed to determine the following information:

- For each task OMPAS encounters, the method that refines it,
- For each arbitrary variable, the element chosen to instantiate it,
- The order in which resources are acquired for different tasks.

Once the plan has been analyzed, the PLanner Manager informs the Acting Manager and the Resource Manager of the decisions that were made by the planner. This information is for the sole purpose of informing the managers. In other words, the managers are free to ignore it.

Update of the planner Because the planning process runs concurrently with execution, it's necessary to update the planning problem to reflect the current state of the system.

Therefore, managers must inform the PLanner Manager when they update the data structures shared with it. Thus, whenever the *Acting Tree*, the world state, or the resources evolve, the PLanner Manager should be kept informed.

The PLanner Manager analyzes any changes to determine if they will affect the planning process, possibly causing it to no longer produce a valid solution in light of the new system state. If such an impact on the planning problem is detected, the PLanner

Manager can restart the planner with an updated problem that takes into account the new system state.

It should be noted that certain planners have the ability to sustain their planning search process by consistently updating the planning problem, starting from a previous solution or even repairing it. The goal of PLanner Manager is to take advantage of the special features of a particular planner to improve the call to a planner, thus ensuring optimal guidance for OMPAS.

2.4 Conclusion

This chapter introduced OMPAS, a new deliberation system based on the deliberation capabilities of RAE. First, a presentation of vanilla RAE introduced the foundation on which OMPAS is built. We first presented the features of RAE, the formalism of hierarchical operational models, and the algorithms of the deliberation functions of RAE. Then we analyzed the deliberation limitation of vanilla RAE. Based on what other versions of RAE have proposed, we defined what new deliberation functions OMPAS should provide.

OMPAS targets the safe execution of multiple concurrent tasks in a multi-robot environment. It proposes an acting system that allows the parallel execution of high-level tasks, and supports concurrency in the *skills* executed by the acting engine to refine those high-level tasks. To ensure secure execution of concurrent procedures, OMPAS exposes a resource system that can be used in the *skills* to ensure exclusive access to shared resources.

The proposed architecture decomposes OMPAS into a set of modules called *managers*, where each manager is responsible for one or more deliberation features. Some deliberation features are inherited from RAE, such as executing and monitoring tasks, commands, and *skills*, and refining tasks into methods. Others are aimed at extending the deliberation features exposed by OMPAS to allow the instantiation of parameters in the body of methods. In addition, the acting manager is responsible for the strategy of allocating resources among the different *skills* by requesting them.

This decomposition of OMPAS into several modules should facilitate the integration of more advanced deliberation features directly into the acting engine. For example, one could extend the State Manager to infer new facts from the state, or to detect erroneous behavior by analyzing the state. We could also extend the Platform Manager to check the applicability of commands by using models of the commands. These models could also define invariants that should hold during the execution of the commands. In this thesis, we present the integration of a hierarchical temporal planner to guide several deliberation functions of OMPAS. This integration is presented in the Chapter 4.

Based on the architecture and deliberation functions of OMPAS, the next chapter presents an acting language dedicated to OMPAS.

Acting Language of OMPAS: Syntax, Semantic and Evaluation

Contents

3.1	Introduction	60
3.2	Acting languages: a review of the literature	60
3.2.1	Features of an acting language	61
3.2.2	Modeling languages	62
3.2.3	Interpretable languages	64
3.2.4	Summary	66
3.2.5	Proposition of an acting language for RAE	67
3.3	The fundamentals of Scheme	68
3.3.1	Scheme syntax and semantics	68
3.3.2	Scheme interpreter principles	69
3.3.3	Scheme evaluation environment	70
3.3.4	Special operators	71
3.3.5	Macro	73
3.3.6	Runtime errors	74
3.3.7	Standard modules of Scheme	75
3.4	Augmenting the Scheme core for control	76
3.4.1	Representing errors in robotic programs	77
3.4.2	Concurrency	77
3.4.3	Representing knowledge using maps	79
3.4.4	Interaction with the environment	80
3.5	Execution modules of the language	81
3.5.1	Platform module	81
3.5.2	State module	82
3.5.3	Acting module	83
3.5.4	Resource module	84
3.5.5	Useful control constructs	85
3.6	Configuration and control modules	85
3.6.1	Configuring OMPAS	86
3.6.2	Controlling OMPAS	93
3.7	Conclusion	94

3.1 Introduction

In the previous chapter we introduced the Operational Model Planning and Acting System (OMPAS), a new acting system that aims at the execution of multiple tasks using a collection of *skills* in the form of programs. The programs are defined as a hierarchical operational model that describes the *skills* of a robotic agent by the arbitrary composition of these programs. With such a model, the goal of the acting engine is to execute these programs and resolve the nondeterministic choices of these programs using the deliberation features of the acting engine.

The formalism of OMPAS allows any programming language to be used to define these *skills*, as long as it exposes primitives to take advantage of the capabilities of the acting engine. We call such a language an *acting language*.

In the literature we find several languages for similar purposes, often called *control languages*. These typically differ from an *acting language* in that they do not allow non-deterministic execution flows. Control languages allow defining the robust and reactive behavior of robotic agents used by the controller in a three-layer architecture. Over the years, several control languages have been developed, each offering specific features. While each language has its own advantages, most of them do not integrate all the acting features required by *OMPAS*, and in particular the explicit use of the resources required for the execution of *skills*. Therefore, we decided to develop SOMPAS, a new *acting language* based on a procedural language with a restricted core, and which implements acting primitives as modules.

The chapter is organized as follows. First, we review the existing modeling languages used to define the behavior of the robotic agent at the control level. Based on this review, we present a new procedural language that is used as a backend for our acting language. We then present the acting primitives available in SOMPAS, as well as the facilities provided by the language to define a hierarchical operational model that can be used by OMPAS to perform high-level tasks. The next chapter is dedicated to the use of the hierarchical operational model defined in SOMPAS to guide the deliberation of OMPAS with heuristics obtained from a hierarchical temporal planner.

3.2 Acting languages: a review of the literature

In many robotic systems, multiple models are used at different levels of the software architecture. In particular, the deliberation and control systems use different models to perform their tasks. While planning systems traditionally use *descriptive models*, control systems use what we call *operational models*. *Descriptive models* explicitly describe the possible evolution of the state of a system for each action that a robot can perform. *Operational models* define *how* those same actions can be performed by this same robot. While *descriptive models* usually take advantage of a deterministic model of the dynamics of the system, *operational models* do not have this luxury and should take into account unexpected events and possible failures, both of which are inherent in real robotic systems.

For several decades, languages have been developed to unify the modeling of planning domains. In this sense, PDDL and its derivatives have been widely used by the planning community to drive the development of planning systems.

In contrast, the field of control languages is more diverse. At first glance, each control system takes advantage of its own dedicated control language. This is limiting, as one would have to re-model the system when switching from one control system to another.

When choosing between different control systems, the robot programmer will certainly make his choice depending on the capabilities of the control system in question. However, the control language used is also of primary importance, as it is the main link between the robot programmer and the system.

In this section, we review several languages that have been used in various control systems. By reviewing these languages, we have identified the main features that a control language should implement. We also group these languages according to their nature. Finally, we present an *acting language* that meets the needs of OMPAS.

3.2.1 Features of an acting language

An *acting language* is essentially a control language extended with deliberation primitives. Control languages are used to define reactive and robust behavior for robotic systems. In this sense, we believe that a control language should implement the following features:

- Executing and monitoring commands on the robot platform. In fact, the first role of a program is to execute commands; the rest should define the context in which they are executed and control the choice between different sequences of commands.
- The control language should allow to adapt the behavior according to the known state of the world. Therefore, a control language should provide a primitive that allows reading the knowledge base of the system. More advanced features to update the knowledge base might also be useful.
- The language should implement basic programming constructs such as branches and loops to define the control flow of a program.
- The definition of event-based behavior is what makes a model robust and reactive to events. Possible events include those that are exogenous as well as those that are internal to the control system. One might want to monitor the state of the system, or define a program whose behavior changes as a function of the other programs running concurrently in the system.
- Concurrency should be allowed in the control system. Indeed, it is highly unlikely that a robotic system can define its behavior by relying solely on a sequence of actions. Moreover, it is possible to control a fleet of robots that perform tasks in parallel. Therefore, the control language should allow concurrent execution of programs. In the presence of concurrency, synchronization is essential, especially when order is required in the execution of multiple tasks.

- At the control level, an execution error can always occur. For cognizant errors, the roboticist should be able to define recovery behaviors directly in the programs.
- The control language may provide facilities to define the composition of programs in a hierarchical fashion, where high-level *skills* are defined by the composition of lower-level behaviors.

These modeling features are those we have identified as necessary to define models that are robust to hazards and can adapt to events. However, these features are primarily used in reactive models, and do not allow models that can take advantage of deliberation systems to better adapt their behavior. Therefore, in addition an *acting language* must incorporate explicit decision points as primitives in the language:

- Choice of an alternative course of action,
- Choice of instantiation of parameters in the body of programs,
- Choice of resource allocation strategy.

Some languages already have some of these acting features in addition to the control primitives, such as the PROPEL (Levinson 1995), which makes the invocation of a planner explicit.

Among the control languages that exist in the literature, we have identified two families: model-based languages, which define a model that is then compiled as an automaton, or interpretable languages. Both types of languages are interesting, each of them embracing different paradigms that could benefit from inspiring each other. Here we try to define a new interpretable acting language, but to be inspired by the programming paradigm of languages used to define automatons, both in the features they propose and in the constructs they expose. Let us first identify the modeling languages used to define capabilities in robotic applications, before moving on to the interpretable languages that have been developed over the years for execution, monitoring, and now acting systems.

3.2.2 Modeling languages

Finite State Machine (FSM) languages FSMs are a convenient way to define *skills*. The deterministic behavior is based on the states that the system can be in and the actions that should be performed in each state.

To define a *skill* as a state machine for robotic applications, one of the easiest ways is to rely on SMACH (Bohren and Cousins 2010). State machines use the ROS executive to implement them, which is convenient for prototyping robotic applications.

Many synchronous languages have been used to formalize the model of a robotic agent. In a synchronous language, all events are considered to occur at the same time, and their propagation is considered to be instantaneous. Synchronous languages are a convenient way to program an automaton. Among the synchronous languages that have been used in robotics, ESTEREL (Boussinot and De Simone 1991) has been used to design a controller for a mobile robot (Sowmya, Tsz-Wang So, and Hung Tang 2002).

ESTEREL relies on strong mathematical semantics, which makes it a formal language particularly suited for automated validation of the models. APL (Coste-Manière, Espiau, and Rutten 1992) and then Maestro (Coste-Maniere and Turro 1997) are two languages that propose to define *skills* based on the response to signals. Both languages are then compiled in ESTEREL.

Hierarchical State Machine (HSM) languages FSMs have been extended to a hierarchical formalism in which states can themselves be composed of substates. Proteus (Tellier et al. 2020) uses this formalism to define the *skills* of an agent in a hierarchical way in a language similar to C++, which is itself compiled in C++.

Petri nets languages While ESTEREL compiles a controller into a FSM, other languages propose to model a system, which is then compiled into a Petri net. Several systems use Petri nets to both define complex *skills* and have a formal model to verify formal properties for those *skills*, such as liveness, deadlocks, and deadends markings. Among the systems that take advantage of Petri nets, we can mention ProCoSa (Barbier et al. 2006) and ASPiC (Lesire and Pommereau 2018), which have been successfully deployed on several robotic architectures. However, using such systems still leaves the burden of defining Petri nets, either textually or with graphical tools. This is where RS (Lesire, Doose, and Grand 2020) is particularly interesting. The language proposes to define *skills* with a specific language that is then compiled into either a Petri net model for verification and validation, or as a SMACH (Bohren and Cousins 2010) model. Furthermore, in RS *skills* are defined in a *SkillSet* that explicitly represents the resources of a system and the state transitions of those resources.

Graphical languages To define *skills*, one could rely on graphical programming tools. Several languages extend the UML formalism to define *skills* whose model is verifiable. Among them, we can cite P (Desai et al. 2013), which is a synchronous language taking advantage of the semantics of UML. LightRocks (Thomas et al. 2013) is another attempt to extend UML to define *skills*.

Event-based languages The previous languages we have presented have chosen to rely on a textual language to model *skills*, which are then compiled as an automaton. With such a language, we have a state-oriented programming paradigm in which the behavior depends on the state of the system. However, the more complex a system is, the more states should be considered in modeling the system, which can be unmanageable for large robotic systems. Another way is to have an event-based programming language that defines the behavior of a system based on signals or events that are abstracted from the state in which they occurred. RMPL (Williams et al. 2003) is a language that allows Hierarchical Constraint Automata (HCA) to be defined naturally, exposing rich constructs to represent parallelism and synchronization between actions and conditions. Unlike other languages for defining automata, the programming style is closer to an imperative style. Another interesting synchronous language is the P_{Lan} EXecution

Interchange Language (Verma et al. 2006), which defines a *skill* in a formalism that can be compiled into a FSM. A *skill* is modeled with several nodes that are inherent to each *skill* and represent the behavior of the *skill* as a function of its state.

If we go back to the features of control languages that we enumerated earlier, most of them are implicit in such a model. For example, parallelism is de facto possible by implementing multiple state machines, and monitoring is the default mode of operation. Synchronization is less trivial: to implement it, two states should be simultaneously reachable from any state the system is in. It is easier in HSMs or StateCharts, which allow simultaneous sub-states in a macro state.

However, programming such a model requires some expertise in the proposed formalism. Moreover, error handling and recovery should be explicit, in the sense that the state reached by the system after an error should be explicitly handled by the automaton.

When it comes to the features expected for *skill* modeling, such as *skill* decomposition, nondeterministic decisions, and resource management, they are not specifically handled by formal languages. This is a fair missing feature, since these decision points are inherently nondeterministic and make it difficult to use a formal model to verify that the requirements are met by the *skill* model.

3.2.3 Interpretable languages

Where automatons are often compiled into C++ code, there are systems where *skills* are interpreted at runtime. In many cases, such a language would take the form of a Lisp dialect. The language would either be a standalone of Lisp or an extension of Lisp.

Lisp dialects One of the first languages for *skills* is RAP (R. J. Firby 1989), which represented the *skills* of a robot agent as Lisp procedures. It was one of the first to propose a refinement system in which a task could be accomplished by one or more methods, the choice of the method being made nondeterministically by the execution system. Parallelism was achieved by defining a "TASK-NET" for each method, which defined the subtask of a method. Order constraints could be defined for the TASK-NET in an HTN-like style.

Not long after, RPL (McDermott 1991) also proposed to build on a Lisp dialect to propose a language for *skills*. Unlike RAP, *skills* can be composed of other *skills*, but there is no notion of methods for abstract tasks. The interest of RPL lies in the powerful constructs proposed for defining reactive behaviors that explicitly take failures into account, such as defining multiple ways to handle a procedure that might fail (with try-one, try-all, try-in-order). Synchronization between processes is also explicit, where RAP relied on the TASK-NET to define the ordering constraints between processes.

CRAM Plan Language (CPL) (Beetz, Mösenlechner, and Tenorth 2010) is another language based on Common Lisp. Many of its constructs for expressing reactive behavior are similar to those in RPL. The peculiarity of CRAM is that it relies on a dedicated inference system that can be used directly on the language to express context-aware behavior. An interesting application of such an inference system is the ability to delay

the instantiation of a parameter until the moment it is needed, e.g. to execute a command. During execution, the parameter can be constrained to respect some necessary properties. When the parameter should be instantiated, the CRAM executive relies on a Prolog-based system to find a suitable value for the said parameter.

PROPEL (Levinson 1995) is another Lisp-based language. At the difference of RAP, RPL and CPL, it does not extend Common Lisp, but a subset of Lisp, which makes it lighter than other acting languages. Speaking of acting languages, it is one of the first to formalize the notion of decision points in procedures. There are two decision points in the language: "choose-value" and "choose-procedure". For both functions, the acting engine must arbitrarily choose an element from a set; a heuristic can be given to guide the choice. Since these choices are explicit, the system can rely on lookahead techniques to anticipate these choices. In addition, facilities are provided to define a simulation model of a procedure that can be used by the deliberation system to anticipate the effects of executing a particular procedure.

ESL (Gat 1997) is another language that extends Common Lisp. It has been successfully used to define the *skills* that run in the EXEC module of the RAX (Muscettola, Nayak, et al. 1998). It extends Common Lisp with constructs needed to define *skills*, and in particular to handle errors. In fact, it adopts the paradigm of cognizant errors, in which errors should be handled explicitly by the *skills*. Thus, for a given procedure that may fail, it is easy to define the behavior the system should have as a function of the type of failure that occurs.

Goal-oriented languages So far, we have seen languages that are either used to define automatons, or executable procedures that are evaluated at runtime. In the latter category, some distinguish themselves by describing the behavior of a system not by the actions it should do, but by the goals it should achieve. The language of the PRS system (Ingrand, Chatila, et al. 1996) takes advantage of *skills*, which are defined as a set of Knowledge Areas (KAs)¹. Each KA is responsible for achieving a specific goal. The body of a KA is an executable procedure defined with PRS primitives, in which subgoals can be defined to achieve the goal of the KA. With such a representation of *skills*, and in the same way as with RAP, the acting system has the choice of which *skill* to refine a given goal. Among the different iterations of PRS, we can mention Propice-Plan (Despouys and Ingrand 2000), whose language syntax differs slightly from that of PRS, but keeps the same operators.

Other languages are goal-oriented, such as Golog (Levesque et al. 1997), a *skill* language for both execution and planning that takes advantage of Prolog's inference mechanisms. Similar to CRAM, nondeterministic choices can use the Prolog engine to find suitable instantiations for some specific procedures in the body of the *skills*. However, these decisions seem to be handled locally, without considering the long-term effects of such decisions. Golog++ (Mataré et al. 2021) extends Golog to represent *skills* in a hierarchical way. However, it does not seem to imply a representation as a

¹The authors of PRS clarified that the terms plans, scripts, procedures, and KAs can be used interchangeably.

hierarchical operational model, where a task can be refined by different methods, and where one may be more appropriate than another depending on the context.

Multiagent oriented languages Following these goal-oriented modeling languages, AgentSpeak (Rao 1996) is defined as a language for Belief-Desire-Intention (BDI) systems. In AgentSpeak, a program is both goal-oriented and agent-oriented, as it aims to define the behavior of an agent based on its recollection of how the world is evolving and the actions of other agents of which it is aware. We can define an agent’s knowledge as beliefs, goals as desires, and plans or *skills* as intentions. With such a framework, the programmer defines the behavior of the agent to respond to new desires or changes in the agent’s beliefs. AgentSpeak has been used in the Jason (Bordini and Hübner 2006), and the JAHRVIS (Mayima, Clodic, and Alami 2022) acting systems, in which primitive actions are calls to JAVA functions.

Another recent multiagent control language is Resh (Carroll, Namjoshi, and Segall 2021). Resh provides the most advanced synchronization features found for defining robot capabilities. Resh takes advantage of a temporal engine to orchestrate parallel tasks. In Resh, parallelism and synchronicity are at the root of the language’s grammar. Many facilities are provided to define exactly how multiple tasks should be orchestrated, in a natural and very concise way.

Python With the increasing popularity of Python as a general-purpose programming language, several systems have chosen to rely on Python to define *skills*. The RAE system of Patra, Mason, Ghallab, et al. (2021) uses operational models defined as Python procedures. In this way, procedures can take advantage of the full extent of Python to define their behavior. However, Python embeds few constructs for controlling robots, and in particular lacks those expected in an acting language. To use it as an acting language, the robot programmer should implement his own module to bind an acting system to the Python interpreter. Recently, the CLAPLEEx system proposed Ala (Vapsi, Borrajo, and Veloso 2023), an *acting language* that can be compiled into Python code at runtime. This high-level language makes it easier to define robot behavior and reduces the burden of defining specific constructs in Python.

3.2.4 Summary

The various languages presented in this section have all the necessary features expressed to make them control languages. Their uniqueness lies in the way they implement these features. Some suggest focusing on specific features, such as ESL, which is particularly suitable for defining the behavior of an agent in which errors are explicitly enumerated and should be handled accordingly. We have seen that two categories of control languages can be distinguished: those used to define automatons, and those used to define procedures that are interpreted at runtime. The first category proposes to define reactive capabilities based on the states that the agent can be in, or the event that it can face. These languages are particularly suitable because they are often based on a mathematical formalism on which properties can be verified.

On the other hand, an interpretable language can be used to define capabilities as plans or policies that can adapt to the different situations that a robotic system may face. Procedures can use decision points that rely on the reasoning capabilities of the acting system to, for example, refine tasks into methods as proposed in RAP, Propel, PRS, and RAE. More advanced features even propose to manipulate these capabilities as first-class citizens that can be transformed as in RPL in combination with the planner XFRM (Beetz and McDermott 1994) or CRAM.

When it comes to modeling the agent using a library of *skills*, interpretable languages are the ones that propose the most flexible formalism. Some model-based languages use a hierarchical representation of *skills*, but never allow the choice of multiple *skills* to perform a given task.

3.2.5 Proposition of an acting language for RAE

The need to define a new interpretable *acting language* stems from the limitations of previous iterations of RAE, which relied on Python programs to define *skills*. Python has many advantages in that it is a widely used programming language, so anyone could start defining RAE *skills*. However, its powerful expressiveness is also its weakness, since the language relies on a grammar that is too rich to allow semantic analysis of *skills* defined with such a language. Furthermore, concurrency is not handled natively in the language. In fact, the Python interpreter relies on a global interpreter lock that prevents parallel execution in a single process. This limits the ability to define concurrent behavior.

Here are the desiderata for our acting language:

- An interpretable language with a restricted core.
- A language whose evaluation has no side effects except for well-defined operators, and whose evaluation is not context-sensitive.
- A language that natively supports concurrency.
- A language that has explicit decision points and whose evaluation is not deterministic, that is, not fixed by the semantics of the language.

These features will make the *acting language* more amenable to automated analysis, such as the automated generation of planning models from programs, as proposed in Chapter 4.

As we have seen, Lisp dialects have a long history of being used to define execution languages. Therefore, the best option seems to be to implement a subset of the Lisp dialect as proposed by Propel, keeping only the primitives necessary to define reactive behavior. Some of these more advanced constructs could be inspired by those proposed in CRAM (Beetz, Mösenlechner, and Tenorth 2010), ESL (Gat 1997), or even RMPL (Williams et al. 2003).

One particular feature that has been missing in the *acting languages* we have studied is the notion of resources. The allocation of resources is often supervised at a higher

level of reasoning, in most cases by a task planner. However, when concurrent *skills* are involved, these *skills* may request access to shared resources. Some languages treat them naively in RAP, ESL, and TCA (Simmons 1994): a process can wait for the availability of a resource and request its explicit lock. However, the semantics of the resource are restricted. We believe that the allocation of a resource should be a nondeterministic choice, since it affects not only the progress of the task that requires it, but also possibly other tasks that may request access to the same resources, with a potentially critical effect on the efficiency of execution.

Like RAP, PROPEL, and PRS, the language should facilitate the definition of the model by annotating the *skills* with metadata. At a minimum, the language must allow an abstract task to be associated with a set of methods that can refine it. However, the model should not be limited to defining the structure of a hierarchical operational model, e.g. defining the equivalent HTN of the operational model. The language should allow to model the whole acting domain of a robotic system, adding the definition of the low-level capabilities of the agent (primitive commands), a definition of the world (state functions), and other facilities that might help the acting system to do its job.

3.3 The fundamentals of Scheme

Since we wanted a language with a simple syntax, we focused on Lisp dialects. Lisp is an old language and stands for *List Processor*. It was developed by John McCarthy (McCarthy 1960), and has been widely used ever since, especially in AI applications. The language is based on the recursive evaluation of lists called S-Expressions (S-Exprs) and it belongs to the imperative and functional programming styles. In the 90s, a unification of the different Lisp dialects led to *Common Lisp* (Steele 1990), which is now the reference language from which many Lisp dialects inherit. Before this unification, a variant called *Scheme* (Moretti 1979) appeared, proposing a syntax slightly different from other dialects. In particular, it has fewer primitives and is mostly functional, which facilitates the analysis of the programs. The features of *Scheme* convinced us to use it as the basis of our own acting language. We deal with a subset of *Scheme*, which we present in this section.²

3.3.1 Scheme syntax and semantics

A programming language can be defined as a set of statements and expressions that can be organized to produce a computation. The *syntax* defines the rules to form correct statements or expressions, whereas the *semantics* gives meaning to those statements and expressions. Let us take the addition of one plus two, which in a mathematical language would be "1 + 2" (like in many programming language), and the semantics is the application of the addition operation to the two numbers, yielding the value 3. The

²It should be noted that the preliminary version of our Scheme dialect was heavily inspired by Peter Norvig's implementation of a Scheme interpreter in Python. He has proposed two tutorials at <https://norvig.com/lispy.html> and <https://norvig.com/lispy2.html> that combine Lisp training with an implementation tutorial.

evaluation of expressions is the determination of its value; in our example we would say that "1+2" evaluates to 3, which could be written as "1+2" \Rightarrow 3.

Unlike other programming languages, Scheme consists entirely of expressions, called S-Expressions (S-Exprs). There is no distinction between statements and expressions, which makes it an expression-oriented language. There are two types of expressions:

- *Atomic expression* such as *numbers* (e.g. 1), *symbols* (e.g. x), *boolean* (e.g. true). Some functions are associated with symbols such as "+" and "*".
- *List expression* which is defined by parentheses surrounding zero or more expressions, the first expression defining the meaning of the expression, the rest are the arguments. In the case where the first element of the expression is a *function* (e.g. "+"), the expression represents the application of the function to the rest of the list, which are referred as the arguments of the function call. If the first element is a *special operator*, its evaluation should be handled by the *executor*.

Comments are delimited by the semicolon character ";" and the return to line character.

3.3.2 Scheme interpreter principles

The execution of a Scheme program is based on the interpretation of the program, whose initial form is a sequence of characters. The interpreter consists of two parts:

- A *parser* which takes a program as input in the form of a sequence of characters, parses it according to the syntactic rules of the language, and translates the program into an internal representation. In most interpreters, the translation results in a tree structure, often called an abstract syntax tree, which reflects the nested structure of expression statements in the program.
- An *executor* that applies the semantic rules of the language to the internal structure produced by the *parser*, and thus carries out the computation of the program. In Scheme, the execution is equivalent to evaluating a single expression that composes the program.

3.3.2.1 Scheme Eval function

The principle of evaluation in the Scheme dialect can be summarized in the EVAL(*expr*, *env*) function presented in Algorithm 3.1. It takes as parameters an S-Expr *expr* and an evaluation environment *env*. If the *expr* is an *atom*, the value returned depends on its type and the *env*. If it is a list, the first element is considered to be the procedure *f* that should be applied to the rest of the list, called the *args*. The *args* are evaluated recursively before being passed to *f*. A procedure can be either (i) a *special operator*, whose specific semantics is handled line 6; (ii) part of the standard language library and associated with a symbol in *env*; (iii) a user-defined function called *lambda* that can be mapped in the current evaluation environment *env*. Unlike other procedure calls, *special operators* are specific functions that take the unevaluated arguments of an expression. Scheme's *special operators* are described in detail below in the Section 3.3.4.

Algorithm 3.1 Overview of the recursive evaluation of expressions in Lisp

```

1: function EVAL(expr, env)
2:   if expr is an atom then return VALUE(env, expr)
3:   else if expr is a list then
4:     f ← EVAL(expr[0], env)
5:     if f is a core operator then
6:       return F(expr[1..], env)
7:     else
8:       args ← []
9:       for i do in [1..|expr|]
10:        arg ← EVAL(expr[i], env)
11:        PUSH(args, arg)
12:       return APPLY(f, args)
13:
14: function VALUE(env, atom)
15:   if atom is a symbol and atom is defined in env then
16:     return GET(env, atom)
17:   else
18:     return atom

```

3.3.3 Scheme evaluation environment

The evaluation of an expression depends on a context defined by an *environment*. An *environment* is a dictionary that maps variables to a value. In Scheme, a variable is defined by a label, a symbol. The environment consists of a list of predefined bindings between symbols and standard Scheme procedures (e.g. "+" and "sqrt"). The *special operator* **define** can be used to map a symbol to any S-Expr in a given environment.

Scheme does not tolerate side effects other than the definition of a variable. Each expression is defined in a new environment env_n , which inherits the bindings of a parent environment env_p . No change in env_n will affect env_p . However, the special operator *begin* allows sequential evaluation of a list of S-Exprs in a common environment, so the definition of an environment local to the environment of *begin* can be used in the rest of the expressions. The result of the *begin* expression is the result of the evaluation of the last expression.

The example given in the figure 3.1 should clarify its function. First, the environment is defined only with bindings to standard Scheme functions (the "+" symbol is bound to a pointer to the interpreter's function addition). After evaluating line 2, the symbol "x" is bound to the value "1" in the environment of the "begin" expression that started line 1. In line 7, the value of x has been locally redefined to "2" after being redefined in line 5. However, because the scope of the second environment only extends to line 7, the value bound to "x" at line 8 is the value previously defined in the environment at line 2.

line	Program	Env
1	(begin	{}
2	(define x 1)	{}
3	(define y	{x=1}
4	(begin	{x=1}
5	(define x	{x=1}
6	(+ x 1))	{x=1}
7	x))	{x=2}
8	(+ x y))	{x=1, y=2}

Figure 3.1: Illustration of the nested environments during the evaluation of a Scheme program. The *env* value represents the bindings at the moment the expression is evaluated. The standard bindings are supposed part of *env* and are not listed here to keep the example clear.

3.3.4 Special operators

As mentioned earlier, Scheme supports a list of special operators with specific semantics. Their special treatment is deliberately omitted in Figure 3.1 to keep the function `EVAL` as simple as possible. The `define` and `begin` operators were introduced earlier to explain how nested environments work. Here we recall their use and introduce the rest of them.

(define sym val) `define` binds a symbol *sym* to an S-Expr *val* in the current environment *env* and returns the *nil* value. If *sym* was already bound to another value, the new binding is defined in the *env*, and holds until *env* is dropped. *Sym* can now be viewed as a variable that can be called in expressions that share *env*. In the following example, we define the variable `x` with a value of 10:

```
(define x 10) => nil
```

(begin e1 ... en) `begin` evaluates a list of *n* expressions (*e1*, ..., *en*) and returns the result of *en*. It is most often used to evaluate a sequence of expressions where the evaluation may depend on the result of previous expressions stored in local variables defined in a context shared by all expressions. Consider the following example, where "x" and "y" are bound to the values "10" and "20", respectively. Once bound, we compute their addition:

```
(begin
  (define x 10)
  (define y 20)
  (+ x y))
=> 30
```

(if cond l r) `if` evaluates an expression *cond* that returns a boolean expression *b*, and evaluates one of two expressions depending on *b*: if *b* = *true*, then the left expression

l is evaluated and the result of the expression is the result of l , otherwise the right expression r is the evaluated expression. The language supports *if* expressions where the right expression r is missing, and defaults to the value *nil*.

Here is an example of a comparison of two numbers that yields different results from two different calculations.

```
(if (> 3 10) (+ 3 3) (* 10 5)) => 50
```

(lambda args body) `lambda` creates a user-defined function that consists of evaluating the body of the expression in a context where the parameters defined in *args* are bound to values passed as arguments to the lambda. There are two ways to define the arguments of a lambda:

- *args* is a list (p_1, \dots, p_n) of n symbols that should be bounded to the n arguments during the evaluation. At runtime, if the number of arguments does not match the number of expected parameters, the evaluation of the lambda *fails*.
- *args* is a unique symbol *list* that is bound to a list of arbitrary length.

Here are three examples of lambdas:

1. We define the function `square` that takes a single parameter x and computes x^2 .

```
(begin
  (define square (lambda (x) (* x x))) => nil
; defines a new lambda in the environment bound to the
  symbol "square"
  (square 5) => 25
)
```

2. We define the function `rectangle_perimeter` that takes as parameter x and y that represent the length and width of a rectangle and computes its perimeter.

```
(begin
  (define rectangle_perimeter
    (lambda (x y) (+ (* 2 x) (* 2 y))))
=> nil
  (rectangle_perimeter 5)
=> "unexpected number of arguments for lambda"
  (rectangle_perimeter 10 5)
=> 30
)
```

3. We define the function `print_all` that print all the elements of a list of arbitrary length. Specific meaning of functions: `null?` returns *true* if the expression corresponds to *nil*, false otherwise, `car` returns the first element of a list (e.g. `(car (list 1 2))` \rightarrow 1), `cdr` returns a list without its first element (e.g. `(cdr (list (1 2)))` \rightarrow (2)).

```
(begin
  (define print_all
    (lambda args
      ;if the list of args is empty, return nil
      (if (null? args) nil
          (begin
            ;print the first element of the list
            (print (car args))
            ;recursive call of the function with the rest of the list
            (print_all (cdr args)))))))
```

(quote e) `quote` prevents the expression e from being evaluated recursively and returns it without applying any computation. The quote operator has a shortened form: $(\text{quote } e) \Leftrightarrow 'e$.

In the following example, the multiplication is not calculated and the expression is returned as is:

```
(quote (* 3 3)) => (* 3 3)
```

(eval e) `eval` explicitly evaluates the expression e . This can be useful to evaluate an expression resulting from a previous evaluation. Following the previous example, we can evaluate a quoted expression:

```
'(* 3 3) => (* 3 3)
(eval '(* 3 3))
=> 9
```

3.3.5 Macro

In Scheme, and in Lisp in general, macros can be defined to simplify the definition of a program. A macro is a special function that, unlike other functions, formats an expression before evaluating it. This is a powerful tool for defining complex programs in a compact way, and for thinking about programs as first-class objects that can be manipulated at runtime. A call to a macro requires its expansion before it is evaluated, meaning that the syntactic rules defined in the macro are applied to the expression before it is evaluated.

A macro is defined in the root environment of the evaluation. We use the operator `(defmacro label lambda)` to add a new macro to the environment. The macro's lambda takes a raw (unevaluated) expression as an argument and returns a new, formatted expression. Such lambdas can take advantage of the special operators `quasiquote` and `unquote` to simplify the definition of syntactic rules for formatting expressions.

As a simple example, we could define the operator `(cadr l)`, which returns the second element of a list l , as a macro:

```
(defmacro cadr (lambda (x) '(car (cdr ,x)))) => nil
```

Once defined in the environment, the macro can be called in the interpreter like any other function. Here is an example with a quoted list `'(3 4)`. The expression is first expanded using the lambda of the macro, then evaluated to the value `"4"`.

```
(cadr '(3 4))
=> (car (cdr '(3 4))) ; expansion of the macro
=> (car (cdr (3 4)))
=> (car (4))
=> 4
```

Quasiquote and Unquote As mentioned above, we rely on the special operators `quasiquote` and `unquote` to define macros.

Consider the following examples, in which a quasi-quoted expression is partially evaluated into an expression. The first phase expands the expression, the result of which is displayed in line 2. Then the resulting expression is the result of the expanded expression, where only the first sub-expression is evaluated, the rest is quoted.

```
(quasiquote (+ (unquote (* 3 6)) 10))
=> (cons '+ (cons (*3 6) '10)) ; expansion
=> (+ 18 10) ; resulting expression
```

The operators `quasiquote` and `unquote` have their corresponding annotations, `(quasiquote e) ⇔ 'e` and `(unquote e) ⇔ ,e`, respectively.

3.3.6 Runtime errors

So far, we have presented the evaluation of a Scheme program assuming perfectly defined expressions. However, as with many interpreted languages, programs that do not follow the syntax will provoke a *runtime error*. A *runtime error* occurs when the evaluation of an expression does not follow the syntax of the language. For example, the evaluation of `(+ 3 'x)` produces a *runtime error* because the addition of a symbol("x") and a number("3") is not allowed. In our implementation of Scheme, a *runtime error* is a structure composed of a verbose message that attempts to clarify the source of the error, and a *backtrace* that points to the expression in the program that caused the error, again to facilitate debugging. Most of the time, a *runtime error* is of the following type:

- *Wrong kind* errors are raised when a procedure expects a different kind of expression.

```
(* 3 t)
=> "Runtime error:
In *, t: got Symbol, expected Number"
(length 10)
=> "Runtime error:
In length, 10: got int, expected [List, Map]"
```

- *Wrong number of arguments* errors occur when the number of arguments is not in the expected range.

```
(cons 10 25 3)
=> "Runtime error:
In cons, "(10 25 3)": got 3 elements, expected 2"
```

3.3.7 Standard modules of Scheme

In addition to the special operators that have specific behavior, which is handled by the EVAL function, Scheme provides numerous computational functions for defining programs. In fact, it is a Turing-complete language, which means that any Turing machine can be defined with this language, thanks to the operators that make up the library of the Scheme dialect.

Depending on the Scheme dialect, the list of operators may differ. We group these functions into modules. A module is defined by

- A label, e.g. `math`, and a module description,
- A list of bindings, including native procedures, but also lambdas and macros that are loaded into the evaluation environment.

We use modules to make our Scheme interpreter modular, and to facilitate the definition of applications that require specific functions that can be easily loaded into the interpreter at compile time.

Therefore, we define all of Scheme's core operators in the *Core* module of OMPAS. We introduce some of the functions it provides, focusing on those that are specific to Lisp dialects. The documentation available at <https://plaans.github.io/ompas/> presents all the functions available in our Scheme interpreter. Here we present only a few of them.

List functions A peculiarity of Lisp dialects is to handle list as first-class objects. Various operators are available to manipulate list:

- `(list e1 ... en)` returns a list of `n` elements formed with the arguments of the expressions.

```
(list 1 '(2 3)) => (1 (2 3))
```

- `(car l)` returns the first element of a list:

```
(car '(1 2 3)) => 1
```

- `(cdr l)` returns a list without the first element.

```
(cdr '(1 2 3)) => (2 3)
(cdr (1)) => nil
```

- `(cons e1 e2)` returns a list based on the value of `e2`. If `e2` is a list, `e1` is prepended to `e2`. Otherwise, a list is created with `e1` and `e2`.

Mathematical functions Scheme's standard library includes the most basic math functions: `+`, `-`, `*`, `/`, `square`, `pow`, `sqrt`, etc.

Logical functions Logical functions can be calculated, such as `<`, `<=`, `>`, `>=`, `=`, `!`, `! =`. In the case of the operator `! =`, it can be handled natively by the interpreter or defined as a macro that converts an expression `(! = a b)` to the expression `(! (= a b))`.

Utility functions In addition to Scheme's core operators, we have added some useful functions. Here are just the ones we found particularly interesting, and whose semantics are not obvious at first glance.

- Predicate functions are used to check the type of an expression, e.g. `(int? e)` returns true if `e` is an int. There is an operator for each type of expression.
- `(rand-int-in-range i1 i2)` is a non-deterministic function that returns a random $n \in [i1, i2]$. Obviously, $i1 \leq i2$ should be true. There is an equivalent function for floats.
- The *IO* module is used to interact with the rest of the operating system, e.g. it contains the `print` function. The `load` operator reads a text file containing a SOMPAS program and evaluates it. The `write` operator edits a text file. As paths to files are used in those operators, the module provides facilities as `get-current-dir` that returns the working directory of the interpreter, and `set-current-dir` that will change the working directory if the new directory is valid, i.e. it exists in the operating system. We can also access to environment variables with `get-env-var` that takes as argument the label of the environment variable.
- A variety of macros and lambdas are provided to facilitate programming with complex functions that, when evaluated, call basic operators. These include the `let` and `let*` functions, which abstract the definition of bindings and are often used by Lisp programmers; `zip` and `unzip`, which are useful for pairing and unpairing lists of elements.
- We can stop the execution of a program for a certain time t with the function `(sleep t)`, where t is in seconds.

3.4 Augmenting the Scheme core for control

In its most basic form, Scheme is a procedural language that evaluates expressions recursively and sequentially. The core of Scheme is purely functional, which means that it respects the referential transparency expected of a functional language, achieved in particular by forbidding mutation. Furthermore, Scheme does not explicitly represent errors as a possible result of evaluation. Some additional types and functions may be needed to facilitate the definition of robotic programs, e.g. to specify error values. In addition, we specified in Section 3.2 that the core of *acting language* should support

concurrent evaluation of expressions. This is particularly useful for designing *skills* that require parallel execution of commands, such as taking photos while moving toward the target. In addition to concurrency, the Scheme dialect should allow interaction with the execution environment, and in particular interaction with the acting system.

In this section, we present the extensions that have been made to the Scheme dialect presented in the previous section. We refer to this extended dialect as Scheme OMPAS (SOMPAS), the dialect used to define procedures in OMPAS.

3.4.1 Representing errors in robotic programs

In OMPAS, a program can fail for several reasons. Either the evaluation of an expression caused a runtime error, or the primitives relying on OMPAS functions failed. In the latter case, we use a convention for representing error types that we call *Error*. An *Error* is an expression `(Err e)`, where *e* is an arbitrary expression that carries an explanation of the error. The expression *e* can then be used by programs to handle the error accordingly.

In SOMPAS we provide the following operators to manipulate errors:

- `(err e)` creates a *Error* with the explanation *e*.
- `(err? e)` returns true if the expression is an error, false otherwise.
- `(explanation e)` unwraps the explanation of the error. Causes a runtime error if *e* is not an error.
- `(check e)` takes a boolean parameter and returns a *error* if *e* is nil, which means false, otherwise true.

In addition to these operators, we add the construct `do`. Similar to `begin`, `do` evaluates a sequence of expressions and returns the last result, except that `do` stops evaluating if an error `err` is returned by one of its sub-expressions. When this happens, `err` is the result of the `do` expression. Structures with `do` and `check` are often used to define programs that check some facts before executing commands.

3.4.2 Concurrency

We have adapted *Scheme* to concurrency by adding new types and primitives to the language. With the following additions, it is possible to evaluate an expression in a new thread, wait for its result, or interrupt it. The first thing we define is the *handle*, a new type of *Atom* that represents the thread doing the asynchronous evaluation. A *handle* can be manipulated with the following functions:

- `(async e)` takes an expression as argument and creates a new thread³ in which the expression *e* is evaluated. The thread is marked ready for execution and the operator returns the thread's *handle*.

³In the current implementation, the interpreter creates a green thread that is handled by a dedicated scheduler.

- `(await h)` takes a *handle* `h` as an argument and returns the result of the evaluation performed in the thread associated with `h`. Until the result of the thread is marked as available, the interpreter waits for the termination of the concurrent thread.

Here is an example where two math computations are evaluated in parallel, with the last computation obligated to wait on the asynchronous evaluations.

```
(begin
  (define h1 (async (* 3 3)))
  (define h2 (async (* 4 5)))
  (+ (await h1) (await h2)))
=> 29
```

Interruptions In addition to providing operators to create threads, you may want to be able to interrupt them before they complete their concurrent evaluation. We define here that an interruption can only occur during the evaluation of an expression, and does not interrupt the call to a native function. When an interrupt signal is sent to a concurrent evaluation of a handle, the interrupt signal propagates recursively to all expressions currently being evaluated in the handle.

- `(interrupt h)` takes a *handle* as argument and sends an interrupt signal to the concurrent thread. The function then waits for the result of the interrupted evaluation. The default result of an interrupted evaluation is the expression `(Err interrupted)`
- `(uninterruptible e)` marks an expression `e` as uninterruptible. This means that if the interpreter has started the evaluation of `e`, it cannot be interrupted by an interrupt signal.
- `(race e1 e2)` takes as argument two expressions that are executed in parallel, the result of the expression is the result of the first expression that finished its evaluation, the second expression is interrupted. Therefore, the result of a race expression is non-deterministic.
- `(with-handler body handler)` evaluates `body` in a new thread and returns its result. If the execution is interrupted, then the lambda `handler` is evaluated (with the result of the interruption of `body` as parameter) and its result is returned as the result of the whole expression.

In Figure 3.2, we illustrate the evaluation of a robot program that executes some commands in a concurrent thread. A subset of the commands is marked as uninterruptible because their sequential execution should be completed to prevent the system from ending up in a broken state. The robot program assumes that the execution of the actions should not take more than 10 seconds, otherwise the sequence of commands should be interrupted. In the scenario presented here, the signal is received during the evaluation of the second command. Since the first three commands are in an uninterruptible expression, the execution of the command is not interrupted, and more importantly, the third command is still executed. Only the fourth command will not be executed.

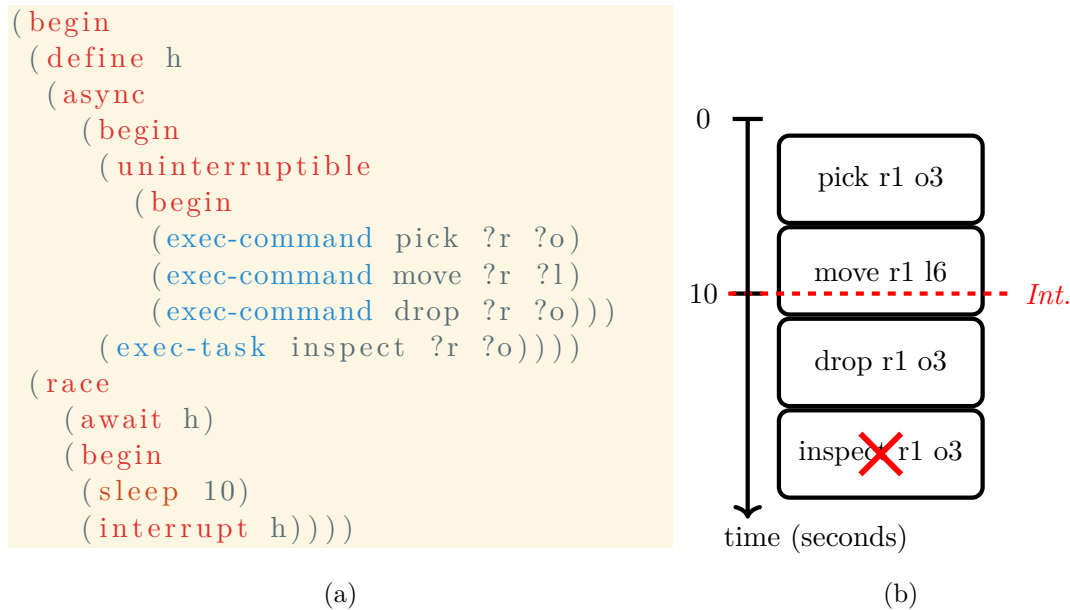


Figure 3.2: Example of interruption of an expression (a) consisting of a sequence of actions to be executed in less than 10 seconds. In the example of a real-time execution (b), the interruption *Int.* (in red) occurs during the action *move*, therefore *inspect r1 o3* is not executed.

3.4.3 Representing knowledge using maps

In Scheme, everything is either an *Atom* or a *List*. In SOMPAS, we add *Maps* as first-class citizens of our language. Maps are another kind of *Atom*, defined by a set of key-value pairs: the key and the value can be any expression. Maps are similar to association lists already defined in other Lisp dialects. As association lists, map tables are particularly suitable for representing knowledge in a robotic system. This way we can easily encode and manipulate a group of facts directly in robot programs. However, for reasons of efficiency, we decided to use native map tables to manipulate knowledge. Pragmatically, we represent the OMPAS state as a map, where the keys are state variables associated with the values they have.

We propose the following operators to manipulate *Maps*:

- `(map '(k1 v1) ... '(kn vn))` creates a new map, arguments are a list of key-value pairs.
- `(get map1 key)` returns the value corresponding to the key in map1.
- `(set map1 (key value))` returns a new map with the new entry.
- `(union map1 map2)` returns a new map resulting from the union of the two maps. Values of `map1` prevail in case of duplicated entries.

Here is an example of how to use a map. We define a map that represents the properties of a robot. Each time the map should be updated, we create a new object, as variables are immutable, we simply define a new binding.

```
(begin
  (define robot (map '(battery 10) '(wheels 4) (speed 5.6)))
  (define speed (get robot speed))
  (define robot (set robot (speed 7.4)))
  (define robot
    (union robot
      (map '(arms 2) (cameras 4))))
  robot
)
=> [ speed: 7.4
     battery: 10
     wheels: 4
     cameras: 4
     arms: 2 ]
```

3.4.4 Interaction with the environment

In Scheme, the evaluation environment *env* is a dictionary that maps symbols to expressions. Most functions are pure, meaning that their evaluation depends only on the arguments of the expression and does not produce side effects. However, some operators related to the acting features of the system may produce results based on a shared environment that can be mutated by the evaluation of an expression or from outside the interpreter by an external program.

Here we introduce the concept of the *Environment Context*. An *Environment Context* is a shared data structure that provides an interface to a memory space or functions external to the Scheme interpreter. Any data structure can be used as an *Environment Context* as long as it guarantees safe access and mutation of its internal data. Access to an *Environment Context* is possible through special functions that should be loaded into the interpreter. Functions that require access to an *Environment Context* have no referential transparency.

An *Environment Context* can only be defined during compilation of the interpreter. Generally, *Environment Context* are defined as part of specific Scheme modules that are loaded into the interpreter along with the functions that can be used to access them. Once defined in the top environment, each child environment has a reference to the *Environment Context*.

A simple example of *Environment Context* is a globally shared counter that can be used, for example, to count the number of times the *print* operator is called.

In our case, we use *Environment Contexts* to link the interpreter of SOMPAS with the managers of OMPAS, e.g. the State Manager.

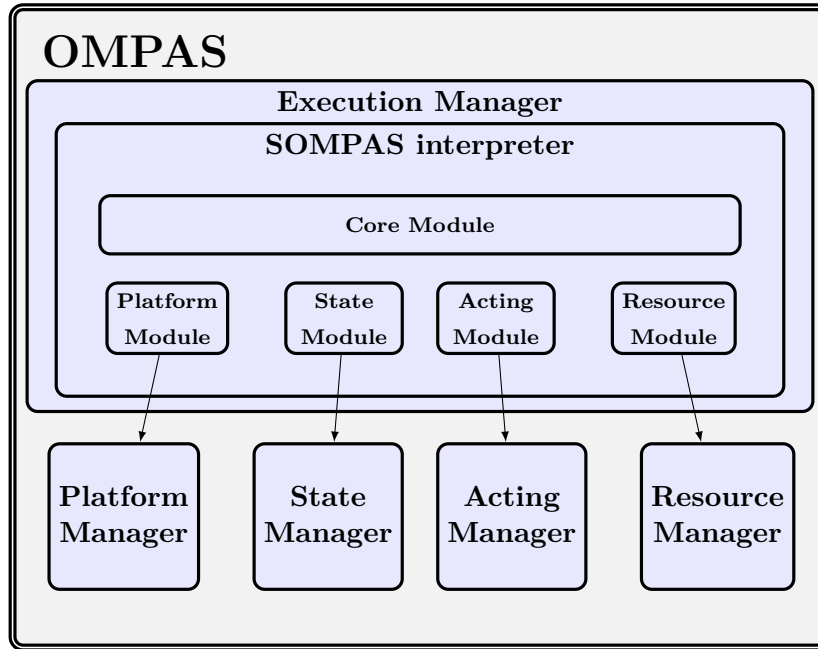


Figure 3.3: Schematic representation of the modules loaded in the interpreter of the Execution Manager and their connections to the other managers of OMPAS.

3.5 Execution modules of the language

So far we have presented the backbone of the language that we use to define acting models. Here we present how to add acting primitives as part of the language's library. To access the features of OMPAS, we define new modules that are loaded in the interpreter of the Execution Manager (EM). Each module is associated with its corresponding manager, as shown in figure 3.3. Each module provides functions to access the acting features of the manager to which it is bound.

3.5.1 Platform module

The primary function of an operational model is to define the contexts for executing commands on a robotic platform. As presented earlier in the chapter 2, the module responsible for executing commands is the Platform Manager. The *platform module* loaded in the SOMPAS interpreter provides the function `(exec-command c p1...pn)`, where $c(p1, \dots, pn)$ is a command to be executed on the robotic platform. This function requests the execution of the command on the Platform Manager (PM) and monitors the status of the command until the command is marked as finished (success or failure). The `exec-command` function can be interrupted by sending a cancel request to the platform. The execution of a command can be aborted by interrupting the call to the `exec-command` function. In this case, the evaluation of the function will stop when the Platform Manager (PM) returns the abort status of the command.

Here is an example of a program that starts the execution of a command in a new thread. The command asks the robot `?r` to move to position `(?x, ?y)`, where `?r`, `?x` and `?y` are parameters of the program. The program can cancel the movement of `?r` if the robot's battery reaches a critical level and asks it to move to the nearest recharging area.

```
(do
  (define h (async (exec-command 'move-to ?r ?x ?y)))
  (race (await h)
    (do
      (wait-for '(< (battery ,?r) 0.4))
      (interrupt h)
      (exec-command 'go_charge ?r))))
```

3.5.2 State module

The role of the *state module* is to provide an interface to the State Manager (SM) of OMPAS. As a reminder, the first responsibility of SM is to maintain a representation of the state in which the robot system is, and to provide facilities to monitor the value of boolean fluents that are function of the state. In addition, SM has a special representation of the objects in the world. Objects are grouped by type, where types are represented hierarchically. The default type of an object is *Object*, which can be specialized by defining specific types (e.g. *robot*). Therefore, the functions provided by the *state module* should allow the definition of behaviors that adapt to the state of the world as known by OMPAS:

- The function `(read-state 'sf p1...pn)` returns the last known value of a state variable `sf(p1,...,pn)`. The parameter `sf` should evaluate to a symbol, the rest of the parameters can be any kind of expression. In the following example, we define a program that moves a robot to the location of an object, the location being obtained at runtime by reading the state:

```
(exec-command 'move ?r (read-state 'loc ?o))
```

- The function `(wait-for fluent)` evaluates to `nil` when the expression *fluent* becomes true. Until then, the expression holds the evaluation of the program. The expression *fluent* is passed to SM, which checks its value every time the state is updated. If *fluent* does not evaluate to a boolean value, it causes a runtime error in the interpreter. The call to this function can be aborted. Here is an example of an expression that waits for a robot's battery to be below 0.4:

```
(wait-for (< (read-state battery r) 0.4))
```

- The function `(monitor fluent)` is the counterpart of `wait-for`. Actually, it is a macro that expands to `(wait-for (! fluent))`. So it waits for *fluent* to become false and returns `nil`. The following expression monitors that the communication medium is still running:

```
(monitor (= (read-state communication r) ok))
```

- The function `(instance o t)` returns `true` if the object `o` is of type `t`, `nil` otherwise. In the following example, we assume that `r1` is a robot, so the following expressions evaluate to `true`:

```
(instance r1 robot) => true
(instance r1 object) => true
```

- The function `(instances t)` returns a list of all objects of type `t`. We give an example where we have defined the type `robot` and the type `rover` (a subtype of `robot`). We have the following results:

```
(instances robot) => (r1 r2 r3)
(instances rover) => (r1 r3)
```

Here `r2` is just declared to be a `robot`.

3.5.3 Acting module

As a reminder, the Acting Manager (AM) is responsible for the refinement of tasks and the arbitrary instantiation of parameters in the body of methods. Therefore, the *acting module* of the SOMPAS interpreter should provide functions that allow access to these deliberation features.

- To execute a task, we define the function `(exec-task t p1 ... pn)`, where $t(p1, \dots, pn)$ is a task to be executed by refining it into a method and executing the method. The `exec-task` function handles the potential failure of a method and uses the retry mechanism of OMPAS to find another method to try. The `exec-task` function ends its evaluation when either a method has succeeded or no more methods are applicable. The function returns `nil` if the task is a success, an error otherwise. The error may be different depending on the cause of the failure. The `exec-task` function can be interrupted, resulting in the signal being passed to the body of the method currently being executed to refine the task.
- The Acting Manager (AM) of OMPAS is able to guide the arbitrary selection of an element from a set. The function call is expressed as `(arbitrary s h)`, where `s` is a list of arbitrary length and `h` is a function that takes a set as argument and returns a single element from it. The `arbitrary` function can return any element of the set. The `h` function can be thought of as a programmer-provided heuristic that OMPAS can freely use to select a value in `s`. In the absence of external guidance provided by OMPAS, the interpreter will default to returning the first element of the set.

Here are a couple of examples to illustrate how it works. Let us take a list of numbers and call `arbitrary` three times.

```

1: (arbitrary '(1 2 3)) => 1
2: (arbitrary '(1 2 3) cadr) => 2
3: (arbitrary '(1 2 3) cadr) => 1

```

The first call evaluates to the first element of the list. In this case, OMPAS has not been given any external guidance or user-defined heuristic. For the second call, the programmer suggests selecting the second element of the list. For the third call, the heuristic is bypassed by the external guidance provided to OMPAS.

3.5.4 Resource module

The Resource Manager (RM) is responsible for the resource allocation strategy among concurrent programs evaluated by the SOMPAS interpreter. The *resource module* provides functions to request access to these resources, but also to declare them at runtime:

- The function `(new-resource r c)` requests the creation of a new resource *r* of optional initial capacity *c* to RM: if *c* is defined, *r* is declared as *divisible*, otherwise *r* is declared as *unary*. If *r* is already declared, it causes a runtime error in the interpreter.
- The function `(acquire r q)` sends a request to RM to borrow quantity *q* of resource *r* and eventually evaluates to a *resource-handle* if the request was validated by RM and RM granted it. If *q* is greater than the maximum capacity of the resource, it provokes a runtime error. The evaluation of the program continues until the request is granted. The function call can be interrupted. A *resource-handle* *r_h* represents the acquisition of the quantity *q* of the resource *r*. The *resource-handle* is encapsulated in a *Handle* and has the following property: If all references of *r_h* are dropped, i.e. they are no longer defined in any environment of the interpreter, the borrowed resource is automatically released.
- The function `(release rh)` immediately releases the borrowed amount of a resource defined behind the *resource-handle* *r_h*, even if other references to *r_h* still exist.

This simple program should facilitate the understanding of how these operators can be used:

```

(begin
  (new-resource 'robby)
  (define rh (acquire 'robby))
  (exec-command 'move robby (read-state 'loc ?o))
  (release rh))

```

First, the resource *Robby* is declared as a *unary* resource. *Robby* is borrowed before it is moved to the location of *?o*. Once the command is executed, the resource is released.

3.5.5 Useful control constructs

The new primitives introduced in SOMPAS provide ways to define the behavior of an agent. Based on these primitives, we define programming constructs to express more complex behaviors in a readable and compact way. Some are inspired by languages like PRS (Ingrand, Chatila, et al. 1996) and CRAM (Beetz, Mösenlechner, and Tenorth 2010), others are specific to *SOMPAS*. All the following functions can be interrupted and stopped if one of their sub-expressions returns an error. Most of them can be defined in terms of the previously introduced operators:

- `(seq e1 ... en)` evaluates n expressions sequentially. It is similar to *do*, but its result is a list containing all the results of the n expressions in order. If one of the evaluations of $e_i, i \in \llbracket 1, n \rrbracket$ returns an error e , the evaluation of the rest of the expression is interrupted, and the result of the expression is e .
- `(par e1 ... en)` evaluates n expressions in parallel and waits for all of their results. The result of the expression is a list containing the result of the n expressions.
- `(repeat e n)` evaluates e n times and returns a list containing the result of all evaluations. However, if one of the evaluations returns an error, the loop is broken and the error is returned as the result of the expression.
- `(retry-once e)`: evaluates e , and if its result is an error, evaluates it again once and returns its result, otherwise returns the result of e .
- `(run-monitoring body e)` evaluates `body` while monitoring that `e` remains true, and returns the result of `body`. If `e` becomes false, `body` is interrupted.

With the definition of such an acting language, we can describe the behavior of a robotic agent in terms of operational models like the one shown in Figure 2.1. The language provides tools to explicitly let the acting engine make decisions during execution, while defining a large variety of *skills* with generic constructs. In the following section, we present how a programmer can configure an Acting Domain (A_Δ) loaded in OMPAS using functions provided by the *Configuration Module* of SOMPAS.

3.6 Configuration and control modules

The OMPAS system provides an interface to interact dynamically with the system through a command prompt. The core of this interaction is a REPL (which stands for Read-Print-Eval-Loop) that evaluates expressions passed through the command line and prints the result to the standard output, i.e. the result can be seen in real time.

The REPL function is quite simple: it loops on reading the string at the input of the system, e.g. the standard input of the program, then parses and evaluates this string, and finally prints the result of the evaluation to the standard output. Note that in the current implementation of OMPAS, the program supports parallel evaluation of multiple expressions coming from multiple inputs, e.g. the standard input or an external

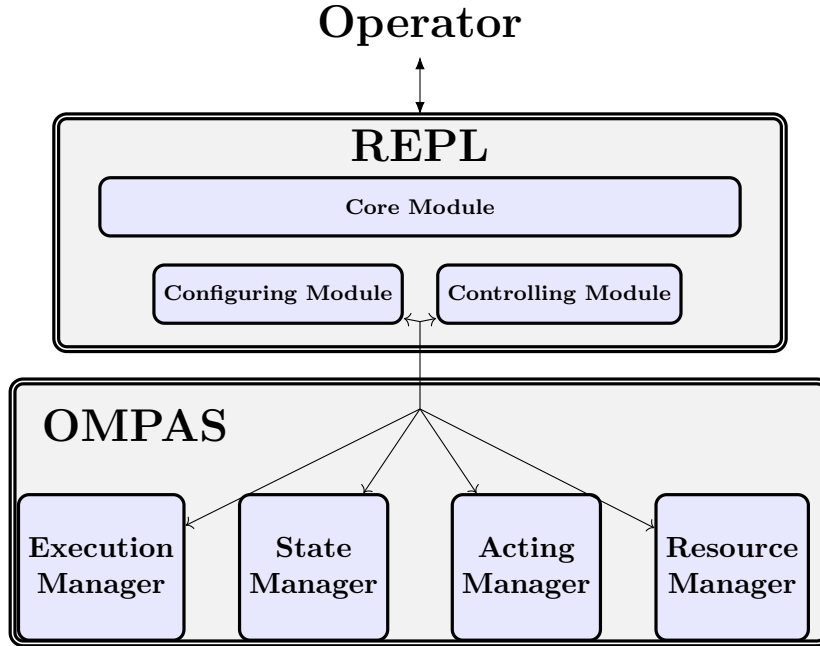


Figure 3.4: Schematic representation of the configuration modules of the interpreter of REPL interpreter provided by OMPAS.

program sending commands via TCP. The REPL can be used to execute any Scheme program, to define the domain of an agent by e.g. loading an external file with *load*, but also to control the acting engine and monitor its execution. All these features are defined in different modules that are loaded in the evaluation environment of the REPL interpreter. In Figure 3.4 we show how they interact with the managers of OMPAS. The *Configuration Module* provides functions to configure everything OMPAS needs to control a robot platform, including the robot agent’s Acting Domain (A_{Δ}). The *Controlling Module* provides functions to control and monitor the execution of OMPAS. Both modules can interact with the Execution Manager, State Manager, Acting Manager and Resource Manager of OMPAS. They either send new information to the managers that might mutate their internal data, or request reports on their internal data structure.

3.6.1 Configuring OMPAS

There are several ways to configure OMPAS. We illustrate them by defining part of the Acting Domain (A_{Δ}) of the *Gripper-Door* domain, as shown in Example 1.1. The operational model of the domain is the one shown in Example 2.1 and Example 2.3. The definition of the hierarchical operational model is done in the same way as shown in (Turi and Bit-Monnot 2022b).

3.6.1.1 Acting Domain (A_Δ)

The Acting Domain (A_Δ) of a robot agent is defined by a set of state functions, commands, tasks, and methods. The model of each object is stored in the Acting Manager. When OMPAS starts its main loop, lambdas are automatically generated and loaded into the execution environment of the Execution Manager to use the capabilities of the agent as functions in the body of methods. Here we describe how the components of A_Δ are declared and what their corresponding lambdas are in the execution environment.

State-functions The *state-function* abstract the acquisition of the value of a state variable that has been defined in the State Manager. The function `(def-state-function sf (:params (p1 t1)...(pn tn)) (:result t))` defines a new state function $sf(p_1, \dots, p_n)$. The parameters p_i are defined along their types t_i as a pair `(pi ti)`. The `(:result t)` defines t as the expected type of the value of the state variable. The parameters are optional. Here is the declaration of the *state-function* `pos(?o)`, which holds the value of the position of an object `?o`:

```
(def-state-function pos
  (:params (?o object))
  (:result location))
```

Each *state-function* declared in A_Δ will have a corresponding lambda loaded in the execution environment of EM. For the *state-function* `pos(?o)`, we have:

```
(define pos (lambda (?o)
  (read-state 'pos ?o)))
```

Similarly, functions can be defined to read the value of a static value in the State Manager. We declare them with the function `(def-function f (:params (p1 t1)) (:result r))`. The arguments are similar to those of `def-state-function`. However, the generation of the corresponding procedure is the following:

```
(define sf (lambda (p1 ... pn)
  (read-static-state p1 ... pn)))
```

`read-static-state` is equivalent to `read-state` but reads only the static part of the state. The difference between `read-static-state` and `read-state` is that since it reads only the static part of the state, we can consider that `read-static-state` has referential transparency in a given context of execution of OMPAS.

Commands The basic commands of a platform can be declared with the function `(def-command label (:params (p1 t1)...(pn tn)))` where `label` is the name of the command. The parameters are defined as a list of pairs `(pi ti)`, where `pi` is a label, and `ti` is a type. Here, we declare the command `move` as defined in *Gripper-Door*:

```
(def-command move (:params (?from room) (?to room) (?d door)))
```

This declaration results in the definition of the following lambda in the execution environment:

```
(define move (lambda (?from ?to ?d)
  (exec-command 'move ?from ?to ?d)))
```

Thus, the body of a method can request the execution of a *command* with an expression of the form `(move ?from ?to ?d)` (*?from*, *?to* and *?d* are assumed to be variables of a program) without explicitly calling the `exec-command` function.

Tasks We recall that in the RAE formalism, a task represents a capability that can be performed in several ways, depending on the context. For a given task, one or more methods should be defined, each method being adapted to different situations. In such a model, the role of the acting engine is to select the most appropriate method to perform the task.

In OMPAS, we declare tasks in a similar way as we declare commands. We use the function `(def-task label (:params (p1 t1)...(pn tn)))` where *label* is the name of the task. The parameters are again defined as a list of pairs `(pi ti)` where *pi* is the label of the parameter, and *ti* is its type. Parameters are optional. Let us define the task `go2(?r)` as defined in the Example 2.1.

```
(def-task go2 (:params (?r room)))
```

This declaration will result in the definition of the following lambda in the evaluation environment:

```
(define go2 (lambda (?r)
  (exec-task 'go2 ?r)))
```

Methods As explained earlier, a task should be defined with methods to perform it. At least one method should be defined for each task. Otherwise, a task will never be executable.

The function `(def-method label task params pre-conditions body)` defines a new method with the following parameters:

- *label* is the name of the method,
- *task* is an expression `(:task label)` where *label* is the name of the task that the method refines,
- *params* is an expression `(:params (p1 t1) ... (pn tn))` defining the parameters of the method that mirror those of the task it refines, and to which additional parameters can be appended,
- *pre-conditions* is an expression `(:pre-conditions e1 ... en)`, where *ei* should be a boolean expression that is evaluated to assess the applicability of the method depending on the state and its parameters.

- body is an expression (`:body exp`) where `exp` is the program to evaluate when the method is selected.

Let us declare the methods `noop(?r)` and `recur(?r, ?a, ?n, ?d)` of the task `go2(?r)`:

```
(def-method noop
  (:task go2)
  (:params (?r room))
  (:pre-conditions (= (at-robby) ?r))
  (:body nil))

(def-method recur
  (:task go2)
  (:params (?r robot) (?a room) (?n room) (?d door))
  (:pre-conditions
   (= (at-robby) ?a)
   (!= ?a ?r)
   (or (connects ?a ?d ?n) (connects ?n ?d ?a)))
  (:body
   (do
    (move2room ?a ?n ?d)
    (go2 ?r))))
```

This definition will result in several lambdas used by the engine to test the pre-conditions and generate the program of the method. We will demonstrate with the method `recur(?r, ?a, ?n, ?d)`.

- The pre-condition lambda tests the types of the parameters and the pre-condition expressions. It returns true if all expressions are true, an error (`Err check-error`) otherwise.

```
(define recur-pre-conditions (lambda (?r ?a ?n ?d)
  (do
   (check (instance ?r room))
   (check (instance ?a room))
   (check (instance ?n room))
   (check (instance ?d door))
   (check (= (at-robby) ?a))
   (check (!= ?a ?r))
   (check (or (connects ?a ?d ?n) (connects ?n ?d ?a)))
  )
  )))
```

- The body that first checks the pre-conditions of the method before evaluating the user-defined program.

```
(define recur (lambda (?r ?a ?n ?d)
```

```
(do
  (recur-pre-conditions ?r ?a ?n ?d)
  (move2room ?a ?n ?d)
  (go2 ?r)))
```

Events In addition to tasks, OMPAS can also handle events. We define an event as a program whose execution is triggered by a set of pre-conditions being true. An event can have a number of parameters. At runtime, the enumeration of all possible instantiations of the event is monitored. The instance of an event can be triggered *once* or *whenever* the pre-conditions are true. For example, given an event *check-battery(?r - robot)* and a list of *robots (r1,r2)*, the system will check the events *check-battery(r1)* and *check-battery(r2)*.

The function `(def-event label params pre-conditions body)` declares an event with the following arguments:

- `label` is the name of the event,
- `params` is an expression of the form `(:params (pi ti)...(pn tn))`, similar to the other parameter declarations,
- `pre-conditions` is an expression of the form `(:pre-conditions trigger e1 ... en)`, where `trigger` defines when the event can be triggered: either *once* or *whenever*; and the expressions *ei* are boolean expressions, similar to the pre-conditions of methods,
- `body` is an expression that represents the program that will be executed when the event is triggered.

As an example, we could define an event that triggers the closing of a door whenever it is opened. We assume that *close(?d)* is a task that acquires *Robby* and moves to a room from which it can close *?d*.

```
(def-event close_door
  (:params (?d door))
  (:pre-conditions whenever (opened ?d))
  (:body (close ?d)))
```

At runtime, events are checked by the State Manager (SM), which notifies the EM that new events have been triggered.

3.6.1.2 Types

Types are used to annotate the operational models used in OMPAS. The type system of OMPAS is hierarchical, which means that subtypes can be defined. The *root* types are *int*, *float*, *boolean*, *object*. The type annotation of an object of A_{Δ} supports compound types:

- `(list t)` type represents a list of objects of type t , e.g. the object *locations* represents a set of locations and can be of type `(list location)`.
- `(tuple t1 ... tn)` represents a list of n objects of possibly different types, e.g. the position of a robot in a 2D space can be represented by an object of type `(tuple float float)`.

New types can be defined in the State Manager of OMPAS using the function `(def-type t p)`, which creates a new type t with optional parent p . If p is already defined, it is added to the type hierarchy as a subtype of *object*. If p is not specified in the function call, t is defined as a subtype of *object*.

In addition to this operator, macros are available to define multiple types with a single expression using `(def-types e1 ... en)`, where e_i can be:

- A symbol t for a new type to be defined as a subtype of *object*.
- A list `(t1 ... tn p)` where t_i are new subtypes of p .

Note that the types are added in order, so the declaration of types can benefit from types declared just before in the same call to `def-types`. Now let us define the types of *Gripper-Door*:

```
(def-types
  (ball carriable)
  (door robot location))
```

ball is declared as a subtype of *carriable*, which in turn is defined as a subtype of *object*. The second line declares *door* and *robot* as subtypes of *location*, where *location* is declared as a subtype of *object*.

3.6.1.3 Objects

Objects can be declared in the State Manager with the function `(def-object o t)`, where o is the label of the object and t is the type of the object. Similar to the definition of types, several objects can be defined at once with `(def-objects e1 ... en)`, where e_i is an expression of the form `(o1 ... oi t)`, where o_i are labels of objects of type t . The functions `def-object` and `def-objects` are defined as macros, so there is no need to quote list when calling them. We can now define the static objects of the *Gripper-Door* domain.

```
(def-objects
  (right left gripper)
  (robby robot))
```

We define the *left* and *right* grippers of *Robby*, which itself is declared as a *robot*.

3.6.1.4 Facts

Facts can be defined in the internal state of the State Manager with the function `(def-facts f1 ... fn)`. The expressions `fi` are expressions of the form `(key value)`, where `key` and `value` can be arbitrary expressions. We use them to define the initial state of the *Gripper-Door* problem:

```
(def-facts
  ((opened d_1) true)
  ((opened d_2) nil)
  (at-robbby l_r)
  ((carry left) empty)
  ((carry right) empty)
  ((pos b_1) bedroom)
  ((pos b_2) kitchen)
  ((pos b_3) l_r)
  ((pos b_4) l_r))
```

Static values can be defined in the static part of the internal state of the State Manager with the function `(def-values f1 ... fn)`. It could typically be used to define the topology of the *Gripper-Door* problem:

```
(def-values
  ((connects l_r d_1) bedroom)
  ((connects kitchen d_2 kitchen) true))
```

3.6.1.5 Resources

Instead of declaring resources at runtime, they can be declared through the REPL with the function `(def-resource r c)`, where `r` is the resource label and `c` is an optional capacity that defaults to 1. Similar to `new-resource`, `def-resource` sends a declaration request to Resource Manager, which may result in a runtime error if the resource has already been declared. Multiple resources can be declared at once with `(def-resources e1 ... en)`, where `ei` can be:

- A single label defining a unary resource
- A list `(r c)` where `r` is the label of the resource and `c` is its initial capacity.

Let us define *Robby* and the grippers as unary resources

```
(def-resources robbby left right)
```

3.6.1.6 Bindings in the execution environment

Similar to `define` in a program, bindings can be defined so that they are available in the evaluation environment of the programs. Instead of `define` we use `(def-env label expr)`, where `label` is a symbol and `expr` is any expression, e.g. a lambda. This way we could define the lambda square:

```
(def-env square (lambda (x) (* x x)))
```

Or define a constant useful for some computations such as the value of π :

```
(def-env pi 3.141592653)
```

3.6.1.7 Configure the deliberation of OMPAS

The heuristic used by the SELECT function of OMPAS can be configured with the function `(set-select h)`, where h can take the value $\{greedy, random, score, \dots\}$, a list that will grow as new algorithms are added. `(get-select)` returns the configured heuristic, *greedy* by default.

3.6.2 Controlling OMPAS

The REPL interface can be used to send jobs to the acting engine. We can `(start)` the engine, or `(stop)` it.

The operator can send tasks to address using the function `(trigger-task t p1 ... pn)`, where t is the label of the task, (p_1, \dots, p_n) its instantiated parameters. The `trigger-task` command returns a *TaskID* t_{id} that can be used to wait on its termination with `(wait-task t_id)`, or cancel it `(cancel-task t_id)`. This *TaskID* is considered to be external to OMPAS and has nothing to do with its internal identification system. We can request the OMPAS ID of a task with `(get-task-id t_id)`.

Monitoring OMPAS To understand how OMPAS works, several functions format the internal state of OMPAS to make it human-readable. We distinguish runtime information from static information for a given instance of execution. The internal state of OMPAS can be monitored through the REPL with the following functions:

- The function `(get-state kind)` returns the whole state of the system, where the optional kind *(static,dynamic)* returns only part of the state,
- The current state of the resources can be monitored with `(get-resources)`, which outputs in a readable way, and for each resource, the current list of acquirers, the capacity of the resource, and the waiting list,
- For debugging purposes, the list of monitored fluents can be accessed via `(get-monitors)`. Each fluent is annotated with the time it was added to the monitoring system.

Once an Acting Domain has been defined with the functions previously presented, a user can verify its declaration with the function `(get-domain label)`, which returns the entire domain composed of the state functions, commands, tasks, methods and bindings of the domain, and a particular element of the domain if `label` corresponds to a previously defined element. The list of elements of each type is accessible with the functions `(get-state-functions)`, `(get-commands)`, `(get-tasks)`, `(get-methods)`.

Logs and loggers The internal functioning of OMPAS generates logs that can be viewed in special windows. The logs are separated into different topics, e.g. *log_platform* or *log_ompas*, which log messages from the platform and the different modules of OMPAS respectively. We can activate the logs with `(activate-logs l1 ... ln)` which takes as argument the labels of the logs to activate. They can be deactivated at runtime with `(deactivate-logs l1 ... ln)`. A log message is associated with a level of criticality defined in $\{trace, debug, info, warn, error\}$, where *error* is the highest level. We can define the minimum log level of messages to be stored in log files. For example, we can execute `(set-log-level debug)` to accept all messages with at least *debug* log level. As for other configurations, we can get the actual log level with `(get-log-level)`.

Other functions are available to control, configure and monitor OMPAS. The detailed list is available in the online documentation of OMPAS (<https://plaans.github.io/ompas/>).

3.7 Conclusion

In this chapter, we introduced a new programming language based on Scheme, a watered-down version of Lisp. The decisions to extend Scheme were based on the study of other languages used to model robotic behavior. This led to the addition of concurrency primitives to Scheme, and acting primitives to provide an interface to a robotic execution platform, and to access the deliberation functions proposed by OMPAS. This concurrency and acting extension to Scheme gave rise to SOMPAS, which is also used to model the behavior of an *acting language* in description files that can be loaded by OMPAS at runtime. The same language is used by the REPL of OMPAS to interact dynamically with the system and to use it as an interface to control a complete deliberation system.

Such a decision to define a new language dedicated to OMPAS is motivated by the desire to control the acting system using planning techniques, without resorting to dedicated planning models that mirror the operational models defined with SOMPAS. Our goal here is to provide a unified framework that both defines the procedures executed by the acting system and serves as a model for the planning engines. In the next chapter we will present how the programs defined with SOMPAS can be automatically analyzed to translate them into a formalism of *descriptive models* that can be used by a hierarchical temporal planner to anticipate the choices of the acting engine.

Planning from Operational Models to Guide the Deliberation of OMPAS

Contents

4.1	Introduction	96
4.2	Background & related work	97
4.2.1	Task Planning	97
4.2.2	Planning to guide a refinement based acting engine	101
4.2.3	Aries: hierarchical temporal planner	103
4.2.4	Unified model for Acting and Planning	107
4.3	Automated generation of planning models from programs	108
4.3.1	Encoding into an Intermediate Representation (IR)	109
4.3.2	Translation of a program into the Intermediate Representation	112
4.3.3	Encoding from the Intermediate Representation to chronicles	119
4.3.4	Additional SOMPAS' features to model planning problems	127
4.4	Unique representation of the executed and anticipated processes	129
4.4.1	Acting Process (AP)	129
4.4.2	Acting Variable (AV)	130
4.4.3	Acting tree	130
4.4.4	Linking the acting tree to the execution and planning models	133
4.5	Guidance of the reactive deliberation of OMPAS using a planner	134
4.5.1	Scope of the planning problem	134
4.5.2	Instantiation of the planner	135
4.5.3	Extraction of the decisions from the plan	138
4.5.4	Continuous update of the planner	139
4.6	Conclusion	140

4.1 Introduction

The primary function of an acting engine is to address abstract tasks through the execution of *skills*, which in turn can request the execution of commands on a robotic platform to achieve the goals associated with those abstract tasks. Here, we employ a refinement based acting engine that utilizes a hierarchical operational model. Within this model, each task can be further refined by one or more methods, where each method consists of a *skill*, which is essentially an executable program. Within this operational context, the acting engine must decide based on the prevailing context:

- Which methods should refine the tasks,
- How to instantiate parameters in the body of methods,
- the order in which resources should be allocated to the tasks that need them.

In addition, the acting engine has the ability to handle failed tasks by switching to an alternate method.

From a broader perspective, we want the acting engine to improve the overall performance of the system. To do this, the acting engine attempts to proactively mitigate the occurrence of failures, deadlocks, or impasses that may occur during the execution of *skills*. What is in the power of the acting engine is the decision it makes. It cannot avoid unexpected events, because they are unexpected by nature, but it can make decisions that reduce the likelihood of such an event occurring. In addition, the acting engine can make decisions to improve the efficiency of execution, such as decisions to reduce resource contention.

In its simplest form, the acting engine makes reactive decisions that depend only on the current state of the system. However, the acting system can make better decisions if it is guided by a heuristic that takes into account the long-term effects of such decisions.

Such a heuristic can be obtained through planning techniques. At its core, planning explores the possible states that the system can reach and tries to find a course in this state space that achieves goals or maximizes some utility measure. Here, the state transitions are induced by the decisions of the acting engine. Therefore, the planning engine should inform the acting engine which decision to make to reach a state in which the tasks it is executing are fulfilled.

From the beginning, the Operational Model Planning and Acting System (OMPAS) has been designed to benefit from planning technology. The PLanner Manager is the entry point of the acting engine to plug in a planning engine that will inform OMPAS of the decisions it should make based on what the planner has foreseen.

Therefore, in this chapter we propose to study the different planning techniques that exist and which one is suitable to guide the deliberation of OMPAS. This study will also justify the integration of a particular planner that takes advantage of a rich planning model that can be directly derived from the operational models used in OMPAS. The techniques used to extract such a planning model are also presented in this chapter. Finally, the integration of continuous guidance of the planner at runtime is explained. Continuous planning allows the planner to anticipate the possible execution flows induced by the decision points of the acting engine, while continuously updating the planning problem to match the current execution state of the system.

4.2 Background & related work

4.2.1 Task Planning

Task planning is a subset of the AI discipline that seeks to give agents autonomy by giving them the ability to know "*what to do*" to achieve a goal, typically defined as a desirable property that should hold in the final state. In its simplest definition, a planning algorithm is capable of generating a sequence of actions to be performed in order to reach a given state in which the goal is achieved, the actions being applicable by one or more agents available in the system. The planning system, also called *planner*, assumes that it has access to a model of the world and a model of the agent, representing its capabilities and the impact on the world of performing an action. The world model and the agent model are described in a so-called *descriptive model*, which defines the dynamics of the world from a high-level perspective.

One of the first formalizations of such a descriptive model is more than fifty years old, and was defined along the well-known Stanford Research Institute Problem Solver (STRIPS) (Fikes 1971) system. In STRIPS, the planning problem is described as a set of first-order formulas over the state of the system that should be true. The planner then uses the applicable actions of the agent to achieve its goal, here making the first-order formulas true. The peculiarity of the STRIPS model is that it describes actions by the state in which they are applicable, defined by first-order formulas over the state, and the changes to the state resulting from the execution of that action. STRIPS was coupled with the PLANEX1 (Fikes 1971) acting system. After the introduction of STRIPS, many other planning systems based their models on a similar state space representation.

As planning systems were developed around the world, it became difficult to compare them because they used proprietary models to define the descriptive model. To bring the different planning communities closer together, the planner independent language PDDL (McDermott et al. 1998) was developed. This language would facilitate both the definition of planning models and the sharing and exchange of planner technology. Of course, planners had to accept PDDL. In most cases, they transform the planning model into an internal representation that is more suitable for their algorithm.

Classical planning relies on many assumptions that are often unrealistic in a robotic context. The more abstract and ideal the model is, the further it is from its physical counterpart, and therefore the further the planner's output is from being executable. Therefore, the planning community seeks to enrich the possible models that a planner would support to answer the question "*what to do?*".

To support richer models, PDDL has been extended in several ways. In fact, the first version of PDDL remains simple, allowing only the definition of STRIPS-like models based on symbolic state transitions. Among these extensions, PDDL2.1 (Fox and Long 2003) supports the followings:

- Numerical extensions to represent numerical state variables, which can be used to define functions that describe continuous changes in state variables (e.g., increasing a robot's battery by a certain amount). Metrics can also be defined that the

planner should optimize in his plan (e.g., energy consumed).

- Temporal extensions to represent the duration of actions, conditions, and effects that can occur at the beginning, at the end, or throughout the action; continuous effects that depend on the duration of an action.
- Other extensions, such as representing conditional effects.

The following planning techniques often use PDDL2.1 extensions to define their models.

4.2.1.1 Temporal Planning

When it comes to integrating planning techniques into a robotic architecture, the planner should have a model that is close to the execution of the system. This means that time should be explicitly modeled to represent at least the duration of commands. Time modeling is also needed to represent concurrency and synchronization between processes, and to account for exogenous events. When dealing with another system, time should also be explicit, for the simplest need of synchronization with e.g. another robot agent.

The first temporal extension that has been proposed is to consider the duration of actions, but keep a state-based representation in which state transitions are instantaneous and do not consider continuously evolving effects, e.g., representing the depletion of a battery as a continuous process. This could be done by discretizing the state for each time step, but the size of the state space would be intractable. Another approach is to rely on timelines to represent state variables independently of the state. The state space representation can still be derived from this representation, but from the planner's point of view, it is easier to reason about intervals than about state transitions when it comes to temporal planning, especially when it involves parallel execution of actions.

While classical planning techniques typically search for a trajectory in state space, some temporal planning evolves in *plan-space*. *Plan-space* differs from classical planning, which reasons in terms of state transitions, where *plan-space* reasons in terms of partial plans. As in state space, the planner seeks a solution by adding or removing actions to the plan, but here the *plan-space* strategy reasons about *flaws*, where a flaw is an unsupported precondition or a conflict between actions (e.g. two actions that change the same state variable). The goal of the planner is therefore to iteratively resolve these flaws without creating *threats*, which are *flaws* created by adding an action that were previously resolved by another action. A plan is found when there are no more defects in the subplan. The search algorithm starts with an empty plan in which the goals are open defects that need to be resolved. Since the strategy starts from the goal, it is called a *backward* strategy, as opposed to *forward* strategies, which start from the initial state and search for a trajectory to the goal.

Typically, temporal planners rely on the following ingredients:

- Temporal primitives: unique timepoints or intervals (defined by a start and an end timepoints),
- State variables that can evolve, e.g. *position(robot1)*, and static values that are state-independent, e.g. *is_city(Paris)*,
- Temporal assertions about the persistence of a state variable or its evolution (discrete or continuous)

- Temporal constraints defined using the Allen algebra (Allen 1983) or by defining a STN (Dechter, Meiri, and Pearl 1991).
- Non-temporal constraints on the values and parameters of state variables.

Using these basic ingredients, the chronicles of the IxTeT planner (Ghallab and Laruelle 1994) represent actions as a set of variables V , a set of constraints over V , a set of conditions that are time-qualified constraints over V that should be true at a given time or during an interval, and a set of effects that define the temporal dynamics of the action with time-qualified assertions representing the continuous or discrete evolution of state variables that are a function of V . Similarly, the planners developed by the National Aeronautics and Space Administration (NASA) the Remote Agent Experiment Planner/Scheduler (RAX-PS) (from the DS1 mission) (Jónsson et al. 2000) and EUROPA (Barreiro et al. 2012), and the planner developed by the European Space Agency (ESA) the APSI (Fratini, Cesta, et al. 2011) are based on timelines and tokens. The PLATINUM framework (Umbrico et al. 2017) is yet another timeline-based approach that extends the ability of a temporal planner to account for temporal uncertainty. This allows the planner to plan in highly unpredictable environments such as HRC, where the human is uncontrollable but should still be taken into account during the planning process.

More recently, the FAPE (Bit-Monnot et al. 2020) is another planner that benefits from the rich expressiveness of chronicles to formulate a plan-space problem with rich search control heuristics based on analysis of the causal network generated by plan-space search. Also using a formulation like chronicles, the Lifted Constraint Planner (LCP) (Bit-Monnot 2018) encodes the planning problem as a CSP, and plan-space exploration is delegated to a standard CP solver. CPT is also a temporal planner that encodes the planning problem as a CSP. CPT takes advantage of the Partial Order Causal Link (POCL) planning algorithm, which it extends for temporal problems.

In addition to PDDL2.1, other modeling languages have been developed, mostly by NASA, such as the New Domain Definition Language (NDDL) for the EUROPA planner (Barreiro et al. 2012), the IxTeT language (Ghallab and Laruelle 1994), and the ASPEN Modelling Language (AML) used by ASPEN (Chien, Rabideau, et al. 2000). Actions in AML (here called activities) can be either primitive actions or abstract tasks (here called compounds). This allows a hierarchical representation of an agent model. AML presents an interesting semantic for resources: resources can be used by multiple users at the same time, and we distinguish *deplatable* resources that are consumed from *non-deplatable* resources that are only borrowed for the duration of the activity.

More recently, the Action Notation Modeling Language (ANML) (Smith, Frank, and Cushing 2008) proposed to support both hierarchical and generative planning in a unified framework. ANML is translatable to PDDL. It proposes a language that makes state transitions explicit in terms of precedent and new value for a state variable, rather than relying on a condition-effect pair, making the definition of an action more compact. In addition, actions can be defined with timing other than the classical start and end labels, e.g. to express a transitive effect that ends before the action ends. However, few planners support ANML: FAPE (Bit-Monnot et al. 2020), which focuses on the temporal and hierarchical features of ANML, and TAMER (Valentini, Micheli, and

Cimatti 2020), which supports numerical fluents.

4.2.1.2 Hierarchical Planning

Instead of relying on the classical representation of the agent’s capabilities with atomic actions, the hierarchical formalism allows the composition of capabilities using simple capabilities of the agent. Typically, such a model is formalized as a tuple (A, T, M) , where A is the set of primitive actions of the agent, e.g., moving, T is a set of abstract tasks representing more complex behaviors, e.g., moving an object to another room, and M is the set of *skills* that can be used to refine a given task $t \in T$. The goal of a hierarchical planner is therefore to find an appropriate decomposition to refine a task in a Hierarchical Task Network (HTN).

The Simple Hierarchical Ordered Planner 2 (SHOP2) planner (Nau, Au, et al. 2003) was awarded at the IPC for its outstanding performance thanks to its search algorithms that generates the planning steps in the same order as they will be executed, thus reducing order uncertainty during the planning phase. Based on the Simple Hierarchical Ordered Planner (SHOP) planning algorithm, Pyhop (Nau 2013) has been successfully adopted in many projects due to its use of Python to define both the planning engine and the planning domain in the form of Python programs. GTPyhop (Nau, Bansod, et al. 2021) extends Pyhop to support Hierarchical Goal Network (HGN) planning as in the Goal Decomposition Planner (GDP) (Shivashankar et al. 2012) where methods are defined to achieve a goal rather than refine a particular task. The Human-Aware Task Planner with Emulation of Human Decisions and Actions (HATP/EHDA) (Buisan et al. 2022) extends the Human Aware Task Planner (HATP) (Lallement, De Silva, and Alami 2014) planner to consider social rules in the HTN formalism, which is particularly useful in Human Robot Interaction (HRI) contexts. HATP/EHDA can emulate the behavior of a human agent and take it into account during the planning process.

The Planning and Acting in a Network Decomposition Architecture (PANDA) framework (Höller et al. 2020) provides a complete planning suite including *SAT* and *forward search* techniques. In addition, the PANDA framework embeds techniques for plan repair, plan and goal recognition, and plan validation.

The Hierarchical Domain Definition Language (HDDL) (Höller et al. 2020) is another extension of PDDL that supports a hierarchical description of an agent’s capabilities: in addition to the classical *primitive actions*, *abstract task* can be defined, for which one or more *methods* should describe how a high-level task can be achieved. HDDL is the official planning language for the hierarchical track of the IPC. HDDL has also been extended for temporal and numerical support in HDDL2.1 (Pellier et al. 2023).

Several hierarchical planners also support temporal models. Among them, SIADEX (L. A. Castillo et al. 2006) uses the forward search strategy of SHOP to which temporal reasoning is added to take into account a richer temporal formalism. The temporal constraints are encoded in a STN that is propagated during the search using a CP solver. In addition to supporting temporal planning, FAPE Bit-Monnot et al. 2020 also targets hierarchical decomposition problems. The hierarchical decomposition problem is expressed in the chronicles used to encode the planning problem, and each chronicle

has a dedicated temporal network compiled as a CSP. *Aries* (Godet and Bit-Monnot 2022; Bit-Monnot 2023b) is also a temporal planner that models the planning problem as a set of chronicles compiled into a CSP. *Aries* extends LCP (Bit-Monnot 2018) for hierarchical problems. The Hierarchical Temporal Event Planner (HTEP)(Cavrel, Pellier, and Fiorino 2023) is also a hierarchical temporal planner. It decomposes the temporal problem into a simpler problem by decomposing durative actions into a pair of two instantaneous actions, each representing the start and end of the durative action. The two actions are then constrained with a causal link representing the duration of the action. It then uses POCL as a search strategy to find a valid lifted solution, and uses a classical CP solver to instantiate all the remaining variables of the planning problem.

4.2.2 Planning to guide a refinement based acting engine

4.2.2.1 Guiding the refinement of tasks

Coming back to our problem, we want to use planning techniques to inform the acting engine what decisions to make. Previous approaches proposed to guide the refinement of tasks into methods. In Propice-Plan (Despouys and Ingrand 2000), programs are continuously simulated by an *Anticipation Module* that can inform PRS (Ingrand, Chatila, et al. 1996) on which *skill* to choose to refine a goal. The guidance uses an anytime algorithm, meaning that a solution is always available during the search process. However, the more time is given to the search phase, the better the solution should be.

Extensions to RAE have been proposed to provide the system with lookahead capabilities similar to Propice-Plan (Despouys and Ingrand 2000) to guide the choice of the method to achieve the goal, taking into account both the current state of the system and possible refinements of subtasks. *RAEPlan* (Patra, Ghallab, et al. 2019) is an anytime planning algorithm that has been integrated with RAE to guide method refinement at runtime. It uses a MCTS algorithm that simulates multiple executions. The MCTS is used to compute the estimated utility of all applicable methods by sampling the costs of the methods’ primitive actions. The cost of a method is the sum of the costs of all primitive tasks. Here, the utility is the *efficiency*, which is the inverse of the cost of a method. Then it chooses the method that maximizes the utility of the task, i.e. the one that has the highest efficiency.

RAE has also been integrated with *UPOM* (Patra, Mason, Kumar, et al. 2020), another planning algorithm based on a MCTS algorithm. Unlike *RAEPlan*, it takes into account the nondeterministic outcome of commands to guide the exploration of the search tree, and can use learning techniques to speed up the search and produce useful heuristics. Several techniques have been proposed along *UPOM*:

- *Learn π* maps a tuple (τ, ξ) of a task and a state to a method, and is used when the acting engine does not have enough time to execute *UPOM*,
- *Learn π_i* returns the best instantiation of arbitrary parameters for a method in a given state ξ , and *LearnH* gives a heuristic for branching in *UPOM*.

While they speed up the search and increase the quality of the chosen methods, they do not guarantee the long-term validity of the plan and the possibility of reaching the

high-level goal.

Along with the formalization of the RAE algorithms, several look-ahead algorithms have been sketched in (Ghallab et al., 2016). The simplest *Run-Lookahead* calls a lookahead planner whenever a new action should be executed, and only executes the first action of the returned plan. In this way, the acting system copes with changes in the environment at each step of execution. However, it seems that calling the planner every time an action should be executed is unnecessary, and could be done only when needed, and this is what *Run-Lazy-Lookahead* proposes, by looking ahead only when the plan is finished, or the current plan is no longer considered feasible by a plan simulator. Recent work by (Bansod, Nau, et al. 2021) proposes to extend *Run-Lazy-Lookahead* to a hierarchical representation of the agent’s behavior, using the power of a hierarchical task and goal planner IPyhop. The result is a new algorithm called *Run-Lazy-Refineahead*. This allows the planner to guide an acting engine based on a hierarchical representation of the agent’s capabilities. Thanks to its compliant interface, the user of IPyhop can easily replan from the middle (Bansod, Patra, et al. 2022), improving the responsiveness of the actor. This is made possible by the improvements of IPyhop over GTPyhop (Nau, Bansod, et al. 2021), and in particular the hierarchical structure of the returned plan, which simplifies replanning triggered by the actor, and which requires only a pointer to the part of the hierarchical tree that failed, and gives a head start on finding a new suitable plan.

However, the part of the tree that has not yet been executed will potentially be replanned, and even unnecessary parts of the plan that are not affected by the previous failure. IPyhopper (Zaidins, Roberts, and Nau 2023) addresses this problem by repairing only the necessary parts of the plans, and has demonstrated faster results than IPyhop on similar domains.

Previous work has focused on guiding the refinement of tasks or goals. Indeed, the choice of method is often the most important decision in a refinement based acting engine. However, as we saw in chapter 2, the acting engine must also be able to manage the progression of multiple tasks, i.e., when the progression of multiple tasks may conflict, the acting engine should be able to favor one task over another.

The previous approaches consider the problem of decomposing a single task and do not take into account the possible problem that may arise from the concurrent execution of tasks. In fact, the planning approach should take into account the concurrent tasks, and also has an explicit representation of time to decide how tasks are interleaved at runtime.

4.2.2.2 On guiding the progression of multiple tasks in a refinement based acting engine

In chapter 2, we have described a generic way to interface a planner with OMPAS, without specifying which planning system could be used. OMPAS uses a description of an agent’s capabilities as a set of hierarchical operational models. Therefore, it seems natural to use a planner that shares this hierarchical representation. In fact, preferred decisions are extracted from the plan synthesized by the planner. Therefore, analyzing

a plan with a structure similar to hierarchical operational models should facilitate the process.

In addition, the plan should provide guidance on how concurrent tasks should be interleaved and, in particular, at what point resources should be allocated to the tasks that request them. Therefore, the planner must explicitly reason about time.

Given the above requirements, the planner should support both hierarchical and temporal planning. Few planners integrate both features. Among them, we can count on SIADEx (L. Castillo et al. 2006), FAPE (Bit-Monnot et al. 2020), PlatinuM (Umbrico et al. 2017), and HTEP (Cavrel, Pellier, and Fiorino 2023).

Aries (Godet and Bit-Monnot 2022; Bit-Monnot 2023a) is a new planner currently under development. *Aries* targets planning problems that require temporal, hierarchical, and numerical reasoning. *Aries* uses a custom hybrid CP-SAT solver to enhance the performance of the planning engine (Bit-Monnot 2023a). Like IxTeT, the planning problems are encoded as chronicles, which are extended to support hierarchical models (Godet and Bit-Monnot 2022). By using chronicles as a model, it should be easier to represent operational models in this formalism. In fact, chronicles are more flexible in representing the internal dynamics of an action, which is needed to represent the temporal complexity of an operational model.

For all of the above reasons, *Aries* was chosen as the planner to guide the deliberations of OMPAS. It should be noted that the design of OMPAS allows for the integration of any planner, but given the scope of this thesis, we chose to focus our efforts on the tight integration of one particular planner. Furthermore, *Aries* is currently being developed by the same team as the author of this thesis. This has facilitated the integration of the planner; support has always been available to help with the integration of the planner; missing features have been quickly added to the planning engine. This has resulted in a level of integration that would have been difficult to achieve with any other planner.

4.2.3 *Aries*: hierarchical temporal planner

As mentioned before, *Aries* is a hierarchical temporal planner that returns a plan composed of a set of temporally qualified actions. These actions can result from the decomposition of high-level tasks, or they can be included in a generative way. *Aries* inherits from LCP (Bit-Monnot 2018), a temporal planner that encodes a planning problem (P_{Π}) as a set of chronicles. A chronicle is a rich formalism that allows more expressive temporal qualification of actions than the classical STRIPS (Fikes 1971) formalism. From the P_{Π} formulation, a set of rules transforms the problem into a corresponding CSP, which is solved by a dedicated solver. The extension proposed in *Aries* is the support for hierarchical planning (Godet and Bit-Monnot 2022) by extending the definition of a chronicle to support the definition of methods composed of subtasks, either primitive actions or abstract tasks.

4.2.3.1 Planning formalism: chronicles

A chronicle is a formalism used in planning to represent the temporal model of an action. Here, an action is either a low-level command or a method that refines a task. As with other planning formalisms, an action model is a predictive model that represents the set of states in which the action is applicable and the set of states resulting from its execution. The state is represented as a set of state variables of the form $sf(p_1, \dots, p_n) = v$, where sf is the label of a state function, (p_1, \dots, p_n) is the list of parameters, and v is its value. Here we assume that the parameters and values are of one of the following types *int*, *float*, *boolean*, *object*.

Unlike a classical STRIPS representation, conditions and effects can happen at any time between the start s and the end e of the action. The core of the chronicle is defined as a tuple (N, T, V, X, C, E, S) , where:

- *Name* (N) is the name of the action it models. It consists of a label and a list of variable parameters, e.g. $move(?from, ?to, ?door)$.
- *Task* (T) is the name of a task the action refines. It also consists of a label a list of variable parameters, e.g. $place(?object, ?room)$. If the chronicle represents a command, this attribute is the same as *Name*.
- *Variables* (V) is the set of variables of the chronicle. It contains the chronicle parameters (also present in N), the timepoints s , and e and possibly other variables.
- *Constraints* (X) is the set of constraints over V . For example, it may contain the structural constraint $s \leq e - 10$, which imposes the duration of the action.
- *Conditions* (C) is the set of temporal conditions of the form $[s_c, e_c] sf(p_1, \dots, p_n) = v$, where s_c and e_c define the interval during which the expression should be true, the said expression being defined as an equality constraint on the value v , and a state variable $sf(p_1, \dots, p_n)$. Conditions are more expressive than STRIPS pre-conditions, which are only relative to the start of an action. Conditions can be defined for any interval $[s_c, e_c]$ such that: $s \leq s_c \leq e_c \leq e$.
- *Effects* (E) is the set of temporal effects of the form $[s_e, t_e] sf(p_1, \dots, p_n) := v$, where $sf(p_1, \dots, p_n)$ is a state variable that transitions to the value v between s_e and t_e , meaning that the value is unknown during $]s_e, t_e[$. Like conditions, effects are more expressive than the STRIPS post-effect, which are only relative to e . Effects can be defined for any interval such that: $s \leq s_e \leq e_e \leq e$. In addition, the transition of a state variable can have a duration, e.g. to represent the time it takes to move between two locations over a temporal interval.
- *Subtasks* (S) is the set of subtasks of the form $[s_s, e_s] a(p_1, \dots, p_n)$, where a is the label of a subtask (either a command or a high-level task), followed by its parameters (p_1, \dots, p_n) ; $[s_s, e_s]$ denotes the interval of execution of the subtask.

Examples of chronicles are given in Figure 4.1 for the command $pick(?o, ?r, ?g)$ of the Gripper domain (see Example 1.1).

```

name: pick(?o,?r,?g)
task: pick(?o,?r,?g)
variables: s, e, ?o, ?r, ?g
constraints:  $s \leq e$ 
conditions: [s] pos(?o) = ?r
             [s] at-robbby() = ?r
             [s] carry(?g) = empty
effects: [s,e] carry(?g):= ?b
         [s,e] pos(?o):= Robby
subtasks:  $\emptyset$ 

```

(a)

```

name: pick&drop(?b,?r,?g,?p)
task: place(?b,?r)
variables: s, e, ?b, ?r, ?g, ?p
constraints:  $s \leq t_1 \leq t_2 \leq t_3 \leq t_4 \leq t_5 \leq t_6 \leq e$ 
conditions: [s] pos(?b) = ?p
             [s] at-robbby() != Robby
effects:  $\emptyset$ 
subtasks: [t1, t2] pick(?b,?p,?g)
          [t3, t4] go2(?r)
          [t5, t6] drop(?b,?r,?g)

```

(b)

```

name: move&drop(?b,?r,?g)
task: place(?b,?r)
variables: s, e, ?b, ?r, ?g
constraints:  $s \leq t_1 \leq t_2 \leq t_3 \leq t_4 \leq e$ 
conditions: [s] carry(?g) = ?b
effects:  $\emptyset$ 
subtasks: [t1, t2] go2(?r)
          [t3, t4] drop(?b,?r,?g)

```

(c)

Figure 4.1: Examples of chronicles to represent actions of the Gripper-door domain (see Example 1.1): the command $pick(?o,?r,?g)$ (Figure 4.1a), and the methods $pick\&drop(?b,?r,?g,?p)$ (Figure 4.1b) and $move\&drop(?b,?r,?g)$ (Figure 4.1c) of the task $place(?b,?r)$.

4.2.3.2 Planning problem: collection of chronicles

In *Aries*, the planning problem (P_{Π}) is defined as a tuple (C_0, C_I, C_{Δ}) , where:

- C_0 encodes both the initial state and the goals,
- C_I is the set of chronicle instances representing the actions that may exist in the plan, These are instantiations of the chronicle templates defined in C_T .
- C_T is the set of *chronicle templates* that can be used by the planner to add missing actions to the plan. Each possible action of the domain (e.g., *move(?from, ?to, ?door)*) is associated with a template.

Initial chronicle C_0 is the initial chronicle, representing both the initial state of the system and its goals. C_0 is contained in C_I . The interval of C_0 represents the planning interval $[0, \mathcal{H}]$, where \mathcal{H} is the planning horizon. If no limits are defined, $\mathcal{H} = +\infty$.

The past and current state is constructed with a set of instantaneous effects $\{[t_i] sv_i := v_i\}$ that represent the current state of the system. The timepoint t_i corresponds to the moment when the state variable was updated. In a classical planning problem, we consider $t_i = 0$. If events are known in advance, they can be encoded as timed effects in C_0 .

A goal can be represented in two ways:

- As a *goal condition* that is encoded by a set of n conditions on the state such that: $\{i \in [1; n] : [t_i] sv_i = v_i, t_i \leq \mathcal{H}\}$.
- As a *goal task* $[s_t, e_t]$ *task* where $e_t \leq \mathcal{H}$.

With such a definition of a goal, temporal constraints can be defined for it. For example, we can define the following for a goal task:

- a deadline t : *happens before* t ($e_t \leq t$),
- a start constraint: *start after* t ($t \leq s_t$),
- a maximum duration for *goal tasks*: *take less than* x *time units* ($e_t - s_t \leq x$).

All of these constraints are defined in the *Constraints* attribute of C_0 . Note that multiple goals can be defined in P_{Π} .

Chronicle templates The planning domain is defined as a set of chronicle templates C_T . A chronicle template can represent either a command or a method. The templates can be instantiated in multiple instances by the planner to populate the planning problem. Newly instantiated chronicles are added to the set of chronicle instances C_I .

Chronicle instances The set of chronicle instances C_I represents the set of parameterized actions that may be present in the final plan. Any $c \in C_I \setminus C_0$ is intended to refine a subtask of a chronicle of C_I . If a subtask is not refined by any chronicle present in the problem, the planner can use the templates present in C_T to instantiate a new chronicle that can refine the subtask.

4.2.4 Unified model for Acting and Planning

Most deliberation systems rely on different models for acting and planning because historically the algorithms of both domains were developed along with their models. On the one hand, deliberate acting reasons about the "how" and relies on operational models that express complex reactive behaviors that aim to be robust to hazards. In our system, *skills* are defined as programs executable in an interpreter, in particular we rely on the dedicated acting language SOMPAS. On the other hand, planning reasons on the "what" and takes advantage of descriptive models that can be defined using dedicated planning languages such as PDDL (McDermott et al. 1998) and ANML (Smith, Frank, and Cushing 2008).

However, relying on different models for *Planning* and *Acting* can lead to semantic mismatch. In fact, *descriptive models* abstract the inner workings of *skills*, especially in robotic systems with truly complex behaviors. The more abstract the *descriptive models* are from the *operational models*, the greater the chance that planning will mislead the acting system. At best, the planner will produce a suboptimal plan. At worst, it may induce incorrect behavior.

Some works have proposed using a single model for both *Acting* and *Planning*, such as IxTeT-Exec (Lemai-Chenevier 2004) or the T-REX (Py, Rajan, and McGann 2010), more systems are discussed in chapter 1. However, these approaches often rely on a formalism that can be difficult for the roboticist to understand. Instead of starting from the operational model, the approach taken by CX (Niemueller, T. Hofmann, and Lakemeyer 2018) is to build on the PDDL model and enrich the language to make it more suitable for execution. However, the modeling language has some limitations when it comes to representing branching, loops, and error handling mechanisms.

In other versions of RAE, the planning engines, such as *UPOM*, plan directly with the operational models defined in Python (Patra, Mason, Ghallab, et al. 2021). As a reminder, *UPOM* samples the simulated execution traces of operational models in a MCTS. In *UPOM*, the planning models are considered as black boxes: the planning engine does not reason about the semantics of actions, it analyzes the state resulting from the simulated execution of the said actions.

The algorithm proved to be efficient to guide the refinement of tasks in RAE, but it does not take into account the execution of concurrent tasks, which limits its ability to guide the interleaving of tasks in OMPAS.

In fact, to guide the interleaving of tasks, the planning algorithm should have access to a temporal model of the actions. However, previous implementations of RAE relied only on the operational models to perform both *Acting* and *Planning*, without having access to an explicit predictive temporal model of the actions. With such a model, and in particular by resorting to chronicles, we can use a temporal planner to anticipate the concurrent execution of several tasks, and in particular those that share limited resources.

Therefore, we need chronicles to represent the effect of the execution of programs. Moreover, we propose to derive the chronicles from a unique model provided by the roboticist to avoid semantic mismatch between the descriptive and operational models.

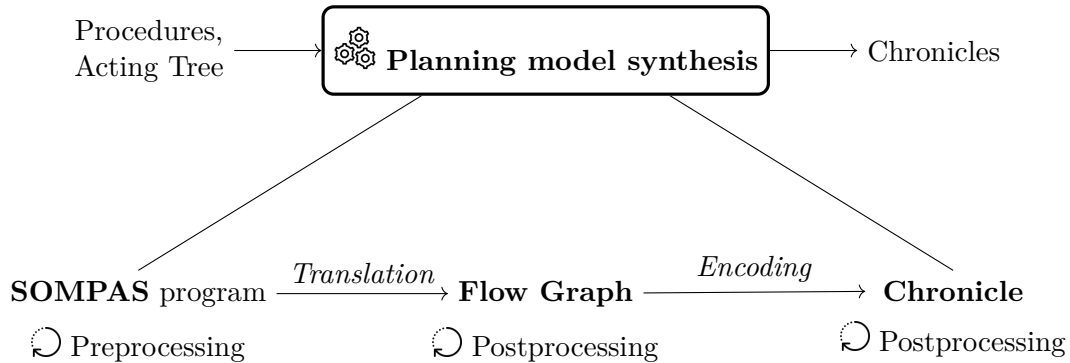


Figure 4.2: Schematic representation of the translation of a program defined in SOMPAS into a chronicle.

What we propose here is to automatically synthesize the chronicles from the operational models. By relying on SOMPAS to define the operational models, we use a procedural language that is simpler than Python, and whose well-defined semantics should facilitate the automated translation of the programs into chronicles. Once translated, the chronicles can be used to call a temporal planner, here *Aries*, to anticipate the execution of tasks, and guide the acting engine in the decisions it will make.

4.3 Automated generation of planning models from programs

The automated generation of chronicles from programs is, to our knowledge, a new way of blending *Acting* and *Planning*. The process consists in the syntactic and semantic analysis of the programs in order to generate a corresponding predictive model in the form of a chronicle.

The synthesis process is divided into two stages: first, the program is transformed into an Intermediate Representation (IR) that is more suitable for static analysis, then the chronicle is built from the lower-level formalism. An overview of the translation process is shown in Figure 4.2.

In our approach, we emphasize the interest of having an intermediate representation of the *skills* in a lower-level formalism. First, it removes syntactic sugar to obtain a restricted set of primitives. Second, it can be used to abstract the translation process from a specific formalism for the *skills* and the planning models. In fact, if either the *skill* formalism or the planning formalism is changed, only half of the process needs to be redefined. Finally, the lower-level formalism can facilitate the static analysis of the *skills* using a standard compilation pass.

To illustrate the translation of the programs, we will use the program defined in Listing 4.1. It is the program of the method *pick&drop(?b, ?r, ?g, ?p)* that refines the task *place(?b, ?r)*. The body contains both the check of the pre-conditions of the method and the body defined in the operational model of the method.

Listing 4.1: Program of the method $pick\&drop(?b, ?r, ?g, ?o)$ that refines the task $place(?b, ?r)$. It is the SOMPAS program of the operational model defined in Figure 2.2a.

```
(do
  (do
    (do
      (check (instance ?b ball))
      (check (instance ?r location))
      (check (instance ?g gripper))
      (check (instance ?p location)))
    (do
      (check (= (pos ?b) ?p))
      (check (!= ?p robbey))))
  (do
    (define r_h (acquire robbey))
    (go2 ?p)
    (pick ?b ?p ?g)
    (go2 ?r)
    (drop ?b ?r ?g)))
```

4.3.1 Encoding into an Intermediate Representation (IR)

During the early development of OMPAS, the intermediate representation of a program was defined in the Single Static Assignment (SSA) formalism (Turi and Bit-Monnot 2022b). It is a formalism in which each value is uniquely defined by a label, and a computation depends only on previously defined labels.

Going back to SOMPAS, the representation of the evaluation of an expression e in SSA is as follows: each step of the evaluation of an expression corresponds to a single line defined by a unique timepoint label t_i , a unique result label r_i , and the call to a *primitive*.

For example the translation of the expression $(+ x y)$ gives:

$$\begin{aligned} t_1 : r_1 &\leftarrow cst(+) \\ t_2 : r_2 &\leftarrow cst(x) \\ t_3 : r_3 &\leftarrow cst(y) \\ t_4 : r_4 &\leftarrow apply(r_1, r_2, r_3) \end{aligned}$$

where cst and $apply$ are primitives in the SSA form.

However, SSA does not aim to represent the concurrent execution of flows in programs. Therefore, we propose to extend the SSA formalism to a *flow graph* formalism to model branching and concurrency in programs.

A *flow graph* is a graphical representation of all possible execution flows of a program. This representation is particularly useful for checking the coverage of a program and for identifying paths that should be avoided, such as those that lead to failures, deadlocks,

or dead ends.

Previous work has focused on control flow graphs for Scheme (Shivers 1988), but since SOMPAS is simpler than generic Scheme dialects, we have derived our own formalism that is sufficient for synthesizing planning models.

4.3.1.1 Flow graph: definition

A *flow graph* is an oriented acyclic graph in which a node represents a set of primitive computations, e.g. a call to a primitive. As in SSA, each line is denoted by a unique timepoint label t_i , a unique result label r_i , and a computation can only be a function of previously defined labels. For SOMPAS programs, their flow graph representation is composed of nodes, each corresponding to the evaluation of an expression.

Node A node represents a sequence of n primitive calls, where each line is defined by a timepoint label referring to the end of the computation, a result label, and the call to a primitive. The generic representation of a node is as follows:

$$\boxed{\begin{array}{l} t_1 : r_1 \leftarrow \text{prim}_0 \\ \dots \\ t_n : r_n \leftarrow \text{prim}_n \end{array}}$$

Since in a node the computations are done sequentially (line by line), then $\forall i \in \llbracket 1, n \rrbracket, t_i \leq t_{i+1}$, and the result of the node is r_n .

In the *flow graph*, we use two specific nodes to represent both the entry point of the program, and its termination:

- The *root* node represents the beginning of a program. It is defined by the start timepoint s :

$$\boxed{s}$$

- The *result* node represents both the end of the evaluation of the expression, and its result. It is defined by the end timepoint e and its result r , both inherited from the last computation node of the flow graph:

$$\boxed{e : r}$$

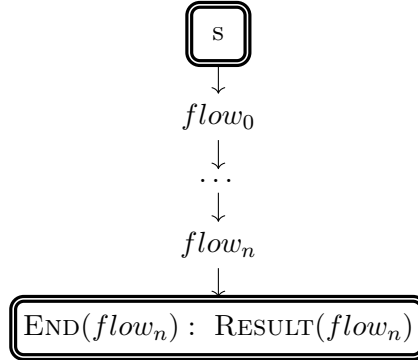
Flow We define a flow as a path between two nodes. A flow consists of a *root*, a *result* node, and in between a set of flows and nodes. By defining a flow with at least a *root* node and a *result* one, a complex program can be viewed as a result obtained during an interval. A flow f can be manipulated with the following operators:

- $\text{START}(f)$ returns the timepoint of the root node of a flow,
- $\text{END}(f)$ returns the timepoint of the result of the flow,
- $\text{RESULT}(f)$ returns the result of the last node of the flow.

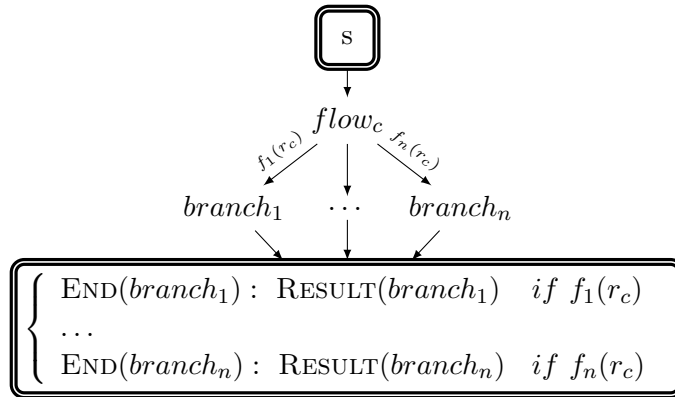
Flows can be composed in the following ways:

4.3. AUTOMATED GENERATION OF PLANNING MODELS FROM PROGRAMS 111

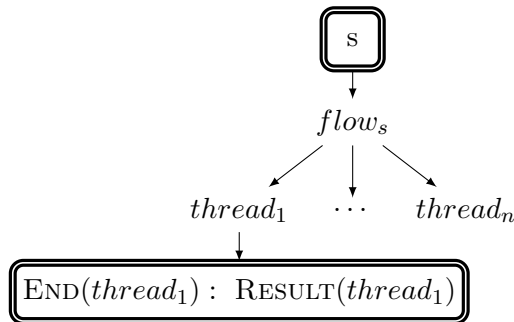
- A *sequential* flow is an ordered set of n flows such that $\forall i \in \llbracket 1, n \rrbracket, \text{END}(flow_i) \leq \text{START}(flow_{i+1})$.



- A *branching* flow represents the conditional execution of n flows, such that only one flow can be executed. It consists of a first *condition* flow $flow_c$, and n branches. A branch $branch_i, i \in \llbracket 1, n \rrbracket$ is executed if $f_i(r_c) = true$, where $r_c = \text{RESULT}(flow_c)$. For a given $r_c, \exists ! i \in \llbracket 1, n \rrbracket, f_i(r_c) = true$. The result of the flow is the result of the executed branch:

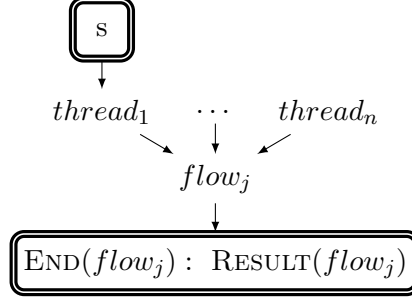


- A *concurrent* flow represents the parallel execution of n flows. All the flows $thread_i$ have a common predecessor $flow_s$ such that $\forall i \in \llbracket 1, n \rrbracket, \text{END}(flow_s) \leq \text{START}(thread_i)$:



$Thread_1$ is the main flow, and defines the result of the *concurrent* flow.

- A *joining* flow represents the synchronization of n flows. All the flows $thread_i$ have a common successor $flow_j$ such that $\forall i \in \llbracket 1, n \rrbracket, \text{START}(flow_j) \leq \text{END}(thread_i)$:



$Thread_1$ is considered the main flow, so it defines the start of the flow.

4.3.2 Translation of a program into the Intermediate Representation

Now that the formalism and accompanying definitions have been presented, we can dive into the heart of the matter. The process of translating a SOMPAS program into a chronicle was first outlined in (Turi and Bit-Monnot 2022b) and then adapted to the *flow graph* formalism in (Turi, Bit-Monnot, and Ingrand 2023).

4.3.2.1 Pre-processing of programs

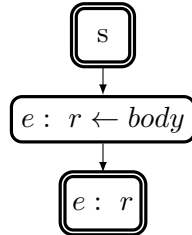
Before converting programs into a corresponding *flow graph*, the programs are preprocessed to facilitate their translation. During this preprocessing step, two transformations are applied to the programs:

- The calls to user-defined functions (lambda functions in SOMPAS) are expanded into new expressions from which the function call has been removed. The details of the transformation are given in Appendix A.1.1.
- The static expressions of the program are pre evaluated. A static expression is essentially a referentially transparent expression, that is an expression whose evaluation does not depend on any parameters outside the ones of the expression, and whose evaluation does not produce any side effects. This static evaluation can benefit from the distinction between *static* values and *dynamic* facts. Indeed, we can consider that the evaluation of $\text{READ-STATE}(sf, p_1, \dots, p_n)$ to be pure as long as all the arguments of READ-STATE are pure and that the state variable is static. This is particularly useful because much of the context is known at the time of the translation, such as the execution environment and the goal tasks. More details and examples of the pre-evaluation process are given in the Appendix A.1.2.

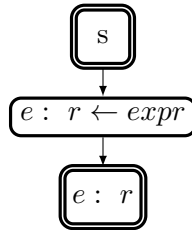
If we take the example given in Listing 4.1, the result of the preprocessing of the program is given in Listing A.1. For the rest of the process, that is the expanded version of the program that is used.

4.3.2.2 Iterative translation of a SOMPAS Program into a flow graph

Once the program has been preprocessed, we can translate the program into a corresponding flow graph. The process works as follows: the *flow graph* is derived from a program by iteratively translating any expression into a flow, and this until only primitive computations are present in the *flow graph*. As a reminder, every program in SOMPAS consists of a single expression, denoted here as `body`. Therefore, the translation starts from a *flow graph* with a single node consisting of the `body` expression:



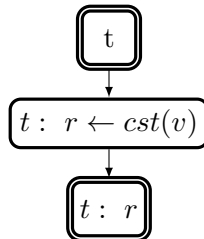
The *flow graph* representation of the expression `body` expression is obtained by expanding each node of the following form into a new flow, where `expr` is not a primitive:



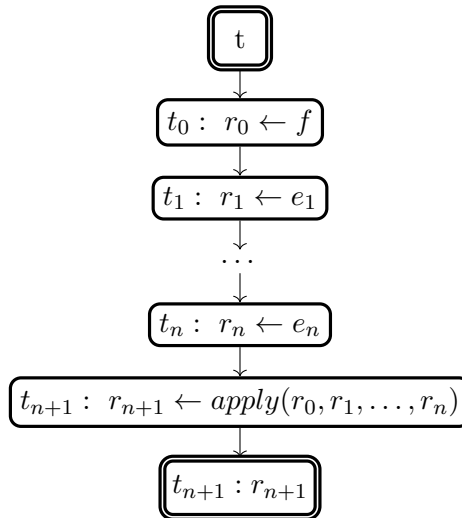
The expansion of `expr` is defined by the rules presented below, so that a unique rule applies to each `expr`. When applicable, the rule expands the single node into a new flow composed of simpler nodes. The rules proposed below are defined for any expression of SOMPAS.

Translation of an expression Given any `expr`, we define the following rules.

expr is an atom that evaluates to the value v . The corresponding flow is the following:



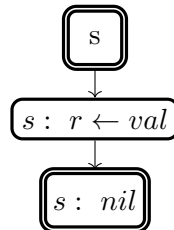
expr is a list (`f e1 ... en`) In SOMPAS this corresponds to the application of the function `f` to the `ei` parameters (where each `ei` might be an arbitrary expression). Following the definition of the `EVAL` function, it is expanded to:



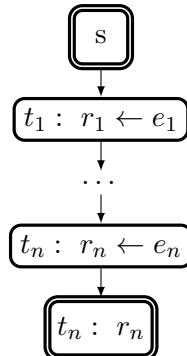
Note that after this expansion, other expansions will be triggered to, e.g., refine the computation e_i into a primitive expression or specialize the last line (function application) into a primitive expression depending on the nature of r_0 .

Translation of the special operators of SOMPAS

expr matches (define var val) This operator defines a name for the value val and returns nil . It is translated as follows, and all subsequent uses of var are replaced with the label r :

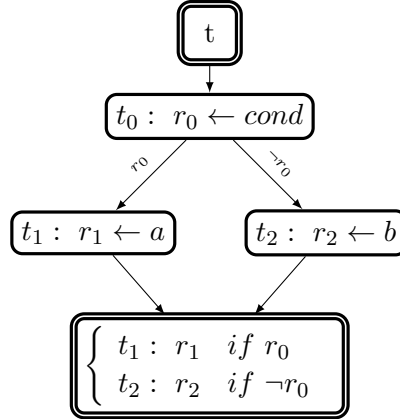


expr matches (begin e1 ... en) The **begin** operator evaluates a list of n expressions sequentially, returning the result of the last expression. It is translated as:

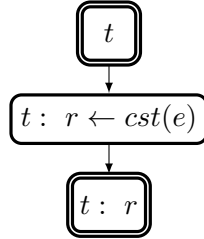


The result of the flow is r_n , the value of the last expression e_n . The end of the flow is defined as the end of the evaluation of e_n .

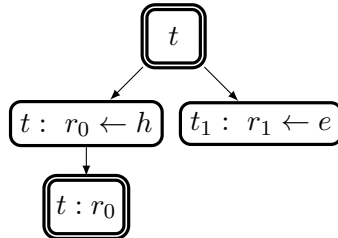
expr matches (if cond a b) The **if** operator first evaluates the `cond` expression, which returns the boolean r_0 . If r_0 is true, `a` is evaluated and is the result of the expression, otherwise `b` will be evaluated. This is translated as a *branching* flow that depends on r_0 : the result of `expr` is either the result of `a` or the result of `b`, depending on which branch is executed.



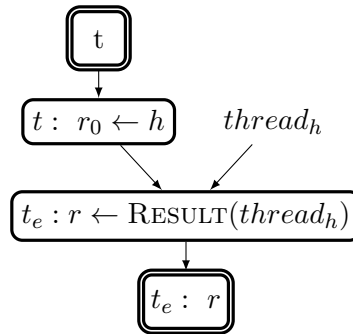
expr matches (quote e) The **quote** operator avoids evaluating the expression `e`. The expression `e` is therefore considered as an atom, and we obtain:



expr matches (async e) The **async** operator creates a new thread in which the expression `e` is evaluated, and immediately returns a handle that is used to refer to the concurrent evaluation, either to wait for its result or to interrupt it. Here, the handle represents the concurrent evaluation flow. The expression is translated as a *concurrent* flow:

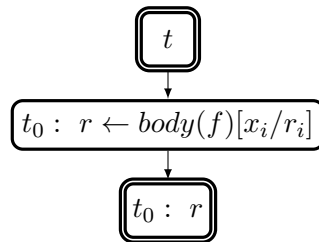


expr matches (await h) The **await** operator holds the evaluation of the current thread until the result of the handle `h` is available. Once available, it returns the result of `h`. The corresponding *flow graph* is a *join* flow, where $thread_h$ is the flow represented by `h`:

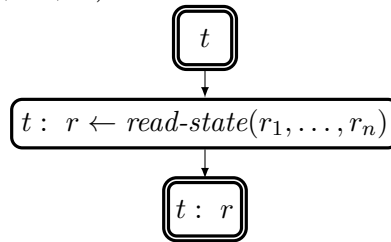


Translation of apply

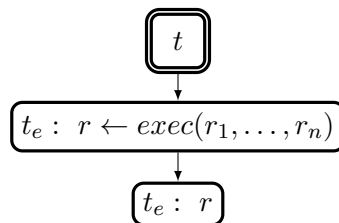
expr matches $apply(r_0, r_1, \dots, r_n)$ where r_0 is a **user defined function** f with parameters (x_1, \dots, x_n) . In this case we replace the expression with the body of the function f , where each parameter x_i has been replaced with the corresponding value of r_i :



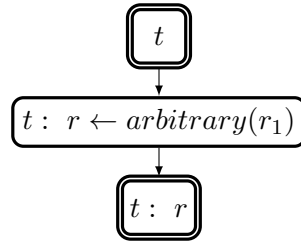
expr matches $apply(r_0, r_1, \dots, r_n)$ where r_0 is the *read-state* primitive:



expr matches $apply(r_0, r_1, \dots, r_n)$ where r_0 is either the **exec-task** or the **exec-command** operator:

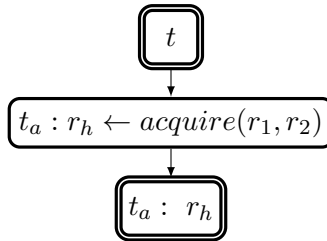


expr matches $apply(r_0, r_1)$ where r_0 is the *arbitrary* primitive:

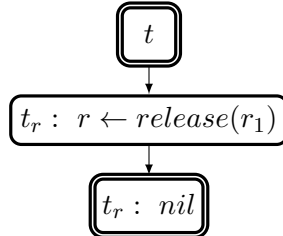


The label r_1 should represent a finite set of elements: either a list of atoms or a symbol type.

expr matches $apply(r_0, r_1, r_2)$ where r_0 is the *acquire* primitive. The *acquire* operator requests a resource r_1 with quantity r_2 , and holds the execution until the resource is granted. It can then return a *resource-handle* which can be used to release the resource. The corresponding *flow graph* is:



expr matches $apply(r_0, r_1)$ where r_0 is the *release* primitive. The *release* operator returns the borrowed amount of the resource. The corresponding *flow graph* is:



Termination If no more rules apply, then the program has been completely translated into a *flow graph*, meaning that all expressions are in primitive form. In our case, this means that each statement corresponds to one of the primitives: *cst*, *read-state*, *exec*, *acquire*, *release*, *arbitrary* or *apply* (with the restriction that the first parameter of *apply* must be a built-in function, since all user-defined functions have been expanded).

4.3.2.3 Post-processing the flow graph

Once the program is translated into a *flow graph*, a series of static analysis passes are applied to it. The goal is to simplify the structure of the *flow graph* by binding variables and, in the meantime, to identify invalid flows. We briefly present the post-processing steps here, more details are given in the Appendix A.2.

Binding labels Equivalent labels are bound using the Union-Find algorithm presented in Algorithm A.1. Two labels are said to be equivalent if they are bound to be equal. Binding two labels also means that the set of possible values they can take should intersect, i.e. that the set is constrained by both labels. We here represent the possible values of a label l as a *domain* $D(l)$. Thus, if two labels l_1 and l_2 are bound, the resulting domain of both labels is defined as the intersection of both domains $D(l_1) \cap D(l_2)$. More details about the domain representation can be found in the Appendix A.2.2.

Propagating constraints on the domain of labels In addition to matching the domains of bound labels, the domains of labels can be inferred from the expressions in which they are implied. For example, the result of a mathematical calculation must be a number. Therefore, the domain of labels bound to the translation of an expression should be bound based on the semantics of the operator. Both the label representing the result of the expression and the labels representing the parameters of the operator should be bound.

For some specific primitive expressions, we propose a set of constraints that would help the automated analysis of the translated programs. We describe these constraints in detail in the Appendix A.2.4.

In some cases, the constraints on the domain of a label can result in an empty domain, meaning that no valid value is allowed for the label. This results in a program that is invalid and would typically return a compilation error. In our case, since the program is interpreted, its evaluation would return a runtime error. If the program is declared invalid, the translation process will fail.

Specializing the program to get a predictive model Here we translate programs that are the bodies of methods that are executed in the acting engine. As a reminder, when selecting a method to refine a task, the acting engine expects the evaluation of the method to be successful. At runtime, the program may fail, and the acting engine is able to deal with this thanks to its retry mechanism as presented in Chapter 2. However, here we are translating the program in order to have an appropriate predictive model to use a planning algorithm to anticipate the execution of the program. Therefore, we want a representation of the program for which we make the assumption that it cannot fail. The failure of a method is characterized by an error return value (see Chapter 3 for more details on expression types in SOMPAS). So we enforce that the result r of the program cannot be an error. Therefore, we restrict the domain of r such that $D(r) \cap Err = \emptyset$.

Because of this special analysis of the program, aimed at the synthesis of a planning model, we relax the rule presented earlier, which states that an empty domain for a variable makes the program invalid. In fact, the additional constraint enforces some values on labels and can lead to empty domains. In this particular case, we state that a label with an empty domain invalidates the nodes and flows in which the label is present. Variables are then constrained to prevent those flows from being executed. In particular, the value of a condition in a *branching* flow can be constrained. In the case of *branching* flows, when specializing the *flow graph* to represent a predictive model,

we allow the removal of branches that cannot be taken by the program, resulting in a simplified model. The transformation is described in detail in Appendix A.2.5.

Example of translated program after postprocessing Based on these rules, we obtain the *flow graph* shown in Figure 4.3. The postprocessing steps presented earlier have been applied to the *flow graph*. In particular, the *flow graph* is now specialized to represent a predictive model that is free of erroneous execution paths.

The postprocessing removed all branching flows, of which there were ten. Among the branches that were removed, we have the translation of (check e) expressions in a do expression. As a reminder, the check operator returns an error if the expression it contains evaluates to false. Furthermore, if one of the sub-expression of a do expression evaluates to an error, the evaluation of the entire expression evaluates to the raised error. Since we assume that a program cannot fail, i.e. its result cannot be an error expression, this in fine constrains the result of check expressions, that should evaluate to true. In other words, we use the semantics of the check and do operators to encode preconditions.

Binding the labels also allowed us to reduce the number of primitive expressions by six times (from 156 nodes to 26). We can also see that most of the primitive expressions are bound to the timepoints s . This is due to the assumption that most computations are instantaneous.

4.3.3 Encoding from the Intermediate Representation to chronicles

Having this flow graph representation, it becomes quite natural to express them as predictive models adapted to planning and in particular in the formalism of *hierarchical chronicles* presented above.

Each chronicle is associated with the task accomplished by the method it represents. For a given method, a chronicle will contain a temporal variable for each of the t_i timepoints and a variable for each of the r_i intermediate results. In addition, a chronicle allows the definition of conditions, which we use to represent the *read-state* primitive, and subtasks, which we use to represent the *exec* primitive. The other primitives are translated as constraints, restricting the set of valid values for the variables.

For a given method, the corresponding chronicle is constructed by iterating through the *flow graph* and applying for a given flow the corresponding rule below.

4.3.3.1 Primitive expressions

Constant $t_i : r_i \leftarrow cst(v)$

The constraint $r_i = v$ is added to the chronicle.

Read state $t_i : r_i \leftarrow read-state(sv, p_1, \dots, p_n)$

The condition $[t_i, t_i] sv(p_1, \dots, p_n) = r_i$ is added to the chronicle.

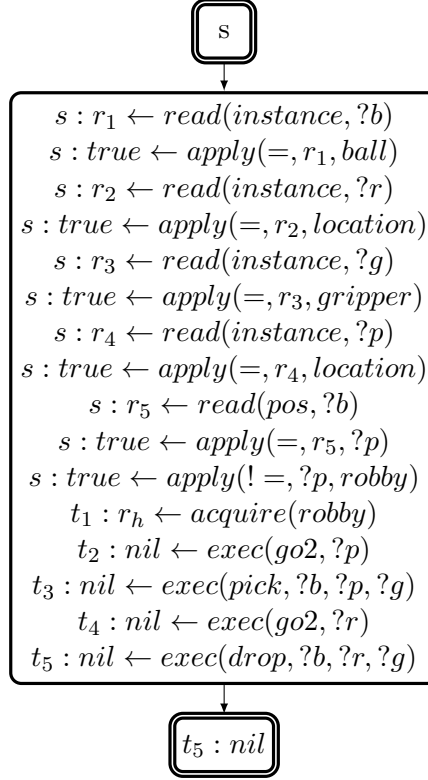


Figure 4.3: *Flow graph* resulting from the translation of the program presented in Listing 4.1. The *flow graph* has been simplified: all nodes have been merged into a single one.

Write state $t_i : r_i \leftarrow \text{write-state}(sv, p_1, \dots, p_n, v)$

An effect $[t'_i, t_i] sv(p_1, \dots, p_n) \leftarrow v$ is added to the chronicle, along with the constraint $t_{i-1} \leq t'_i \leq t_i$.

Subtask $t_i : r_i \leftarrow \text{exec}(\tau, p_1, \dots, p_n)$

The subtask $[t'_i, t_i] \tau(p_1, \dots, p_n)$ is added to the chronicle, along with the timepoint t'_i and the constraint $t_{i-1} \leq t'_i \leq t_i$. Note that in a hierarchical planner, all subtasks of a method must be successfully decomposed and do not provide any result value. The above translation is therefore only valid if the result r_i has been identified as necessarily successful by the error propagation.

Arbitrary $t : r \leftarrow \text{arbitrary}(\text{set})$

An *arbitrary* primitive is converted into a disjunctive constraint such that $\bigvee_{e \in \text{set}} r = e$.

Acquire $t : r_h \leftarrow \text{acquire}(?r, ?q)$

To represent the acquisition of the resource $?r$ with the quantity $?q$, we use the synthetic state variable $\text{quantity}(?r)$ and the function $\text{max-q}(?r)$.

- $quantity(?r : resource) \rightarrow int$ encodes the current quantity of the resource,
- $max-q(?r : resource) \rightarrow int$ encodes the maximum value that the resource can take (set to 1 for unary resources).

Therefore, the possible values for the state variable $quantity(?r)$ are bounded by 0 and $max-q(?r)$.

To encode the acquisition of a resource, we assume that the solver supports the numerical operators increment and decrement on numerical state variables. The acquisition of a resource is encoded as follows:

$$\begin{array}{ll}
 \text{constraints:} & 0 \leq ?q \leq max-q(?r) \\
 & 0 \leq q_1 \leq max-q(?r) \\
 & 0 \leq q_2 \leq max-q(?r) \\
 & t \leq t_r \\
 & t_r = max(Drops(r_h)) \\
 \text{conditions:} & c1 : [t] \text{ quantity}(?r) = q_1 \\
 & c2 : [t_r] \text{ quantity}(?r) = q_2 \\
 \text{effects:} & e1 : [t] \text{ quantity}(?r) -= ?q \\
 & e2 : [t_r] \text{ quantity}(?r) += ?q
 \end{array}$$

The timepoint t represents the moment of the acquisition. At this moment, the effect e_1 decreases the state variable $quantity(?r)$ of the borrowed amount $?q$, and the condition c_1 checks that the acquisition is valid, i.e. if the new value is within the allowed range.

Since an acquisition automatically implies a release, we propose to encode the release of the resource in the meantime using a timepoint t_r . At the time of release t_r we add an effect e_2 that increases $quantity(?r)$ of the previously borrowed quantity $?q$. The condition c_2 has the same function as the condition c_1 , i.e. it checks that the release is valid. As presented in Chapter 3, a resource can be released explicitly, or automatically when the last reference of the *resource-handle* is dropped. For the moment, we propose to constrain t_r in function of the moment of releases of the resource. Those moments are defined by the set $Drops(r_h)$, which is the set of all the timepoints at which a reference to r_h is dropped. We take the maximum, since it is the dropping of the last reference that triggers the automatic release.

Release $t_r : r \leftarrow release(r_h)$

The explicit release of a resource can be triggered with the *release* primitive and constrains the release date t_r . Previously, when encoding the acquisition of the resource, t_r was previously constrained based on the drop dates of the references to r_h . With an explicit release, we change the constraint to take into account the explicit release dates. A release of a resource can occur at various points in the program, especially if multiple threads have access to a reference of the *resource-handle*.

With all these rules in mind, we replace the previously defined constraint $t_r = max(Drops(?r))$ with the constraint $t_r = min(Releases)$, where *Releases* is the set of all release dates of r_h .

Function application $t_i : r_i \leftarrow \text{apply}(f, p_1, \dots, p_n)$

A new variable r_i is added to the chronicle, along with the constraint $r_i = f(p_1, \dots, p_n)$. Note that f is a built-in function by construction, so it is assumed that the solver is able to handle it, possibly with semantic bindings.

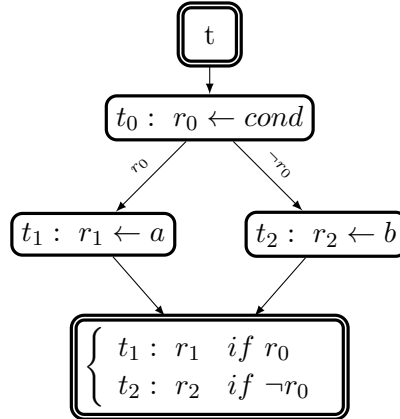
In the current version we support the following operands:

- f is the **+** operator: it is transformed into the constraint $r_i = \sum_{i=1}^n p_i$,
- f is the **-** operator: it is transformed into the constraint $r_i = p_1 - \sum_{i=2}^n p_i$,
- f is the **!** operator: it is transformed into the constraint $r_i = \neg p_1$,
- f is the **=** operator: it is transformed into the constraint $r_i = (p_1 = p_2)$.

Note that the **and** and **or** operators of SOMPAS are encoded as lambdas and have been expanded during the translation process.

4.3.3.2 Complex flows

Branching flow For a given flow of the form



we take advantage of the hierarchical formalism to transform it into a call to a synthetic task τ associated with two synthetic methods. The synthetic task τ takes as parameters the variable r_0 , a new variable r_{result} to represent the result of the task, and any variable that should be passed to the methods. The two synthetic methods are defined as follows:

- The first one represents the translation of the flows of the left branch. Its parameters are inherited from those of τ and may possibly have additional ones. It has two predefined constraints: $r_0 = true$ and $r_{result} = r_1$.
- The second method represents the translation of the flows of the right branch. Its parameters are inherited from the ones of τ and may possibly have additional ones. It also has two predefined constraints: $r_0 = false$ and $r_{result} = r_2$.

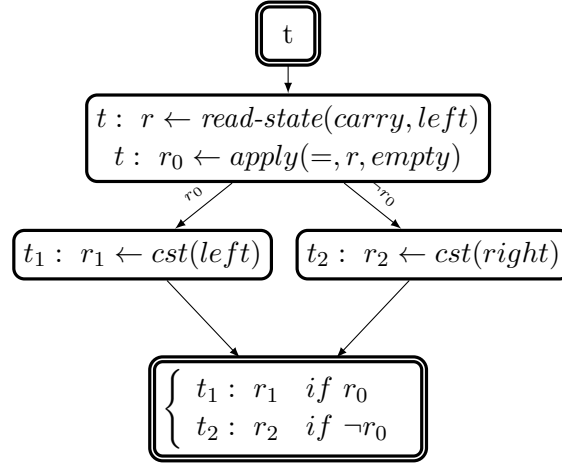
The Example 4.1 presents a simple example of the conversion of a *branching* flow into the chronicle formalism.

Example 4.1

Let us take the expression

```
(if (= (carry 'left) 'empty) 'left 'right)
```

This expression returns the gripper *left* if it does not carry anything, otherwise the *right* one. The corresponding (postprocessed) *flow graph* is:



The conversion of the *flow graph* results in the addition of the following elements to the main chronicle:

variables: $r, r_0, r_{result}, t, t_e$
 constraints: $r_0 = (r = \text{empty})$
 conditions: $[t] \text{carry}(\text{left}) = r$
 subtasks: $[t, t_e] \tau(r_0, r_{result})$

In addition two templates of methods are generated for the task τ :

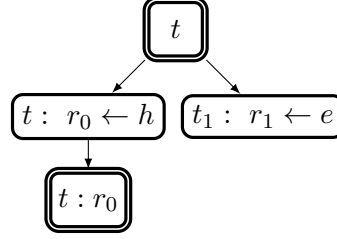
- The method m_l representing the left branch:

name: $m_l(r_0, r_{result})$
 task: $\tau(r_0, r_{result})$
 variables: r, r_0, r_{result}
 constraints: $r_0 = \text{true}$
 $r_{result} = \text{left}$

- The method m_r representing the right branch:

name: $m_r(r_0, r_{result})$
 task: $\tau(r_0, r_{result})$
 variables: r, r_0, r_{result}
 constraints: $r_0 = \text{false}$
 $r_{result} = \text{right}$

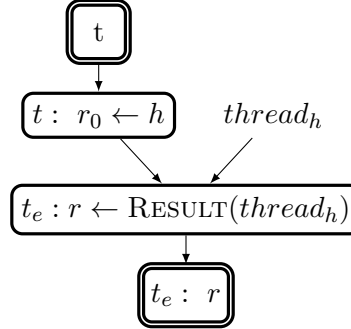
Concurrent flow For *concurrent* flows of the form



we constrain the start of the concurrent branch t' such that $t \leq t'$.

The semantics of the language say that if all references to h are released, then the asynchronous process of h is interrupted, which can be considered as a failure in the planning semantics. Therefore, we enforce that $t_1 \leq \max(Drops(h))$, where $Drops(h)$ is the set of all the moment the reference h is dropped.

Join flow For *join* flows of the form



we add the constraint $\text{END}(thread_h) \leq t$, and $r = \text{RESULT}(thread_h)$.

Success and ordering constraints We rely on the error analysis, to enforce that the plan is fault-free if it is deemed valid by an automated planner. In particular, if a condition is found to require a particular value to avoid a failing branch, we enforce this as an additional constraint on the corresponding variable. We also enforce the order in the various statements, i.e. $t_i \leq t_{i+1}$ for any two consecutive timepoints in the *flow graph*.

All these rules are used to build the chronicle of Figure 4.4, which is obtained from the *flow graph* of the Figure 4.3. Once the chronicle is constructed, the same postprocessing steps as described by Turi et al. (2022b) are used to simplify the structure of the resulting chronicle by removing unnecessary variables and reducing the number of constraints, conditions, effects and subtasks. Typically, the post-processing of a chronicle such as the one shown in Figure 4.3 can remove a quarter of the variables, more than half of the constraints and half of the conditions.

4.3.3.3 Postprocessing of chronicles

Once the program is converted from its intermediate form, here a *flow graph*, we obtain a corresponding chronicle suitable for planning. However, since the chronicle is constructed by iteratively transforming flows into a corresponding set of constraints and

```

name: pick&drop(?b, ?r, ?g, ?p)
task: place(?b, ?r)
variables: s, e, ?b, ?r, ?g, ?p, t1, t2, t3, t4, t5
constraints: ?p ≠ robbly
              $s \leq t_1 \leq t_2 \leq t_3 \leq t_4 \leq e$ 
              $0 \leq r_6 \leq 1$ 
              $0 \leq r_7 \leq 1$ 
conditions: [s] instance(?b) = ball
            [s] instance(?r) = location
            [s] instance(?g) = gripper
            [s] instance(?p) = location
            [s] pos(?b) = ?p
            [s] quantity(robbly) = r6
            [e] quantity(robbly) = r7
effects:    [s] quantity(robbly) -= 1,
            [e] quantity(robbly) += 1
substasks: [t1, t2] go2(?p)
            [t2, t3] pick(?b, ?p, ?g)
            [t3, t4] go2(?r)
            [t4, t5] drop(?b, ?r, ?g)

```

Figure 4.4: Chronicle resulting from the conversion of the *flow graph* of Figure 4.3. The chronicle has been postprocessed and simplified to make it more readable for the sake of this dissertation.

timed assertions, we can easily understand that artifacts come along with it. Some of these artifacts are useless with respect to the structure of the chronicle, and can therefore be removed by automated analysis of the resulting chronicle. We propose a series of postprocessings applied to the raw chronicle in order to obtain a refined one. The obtained chronicle is more readable, has fewer elements and therefore implies a less complex planning problem.

Simplify constraints We suggest simplifying constraints of the form $(true = c)$, where c is a constraint, and replacing it directly with c , which gives $(true = c) \implies c$.

Merge duplicated conditions The translation of a program into a chronicle can create duplicate conditions. Such conditions can be merged if they meet the following criteria: given $c_a : [s_a, e_a] sf_a(p_{1_a}, \dots, p_{n_a}) = v_a$ and $c_b : [s_b, e_b] sf_b(p_{1_b}, \dots, p_{n_b}) = v_b$, if $sf_a = sf_b \wedge s_a = s_b \wedge e_a = e_b \wedge p_{1_a} = p_{1_b} \wedge \dots \wedge p_{n_a} = p_{n_b}$, then c_b is removed and the labels v_a and v_b are bound in the same way as described in section 4.3.2.3.

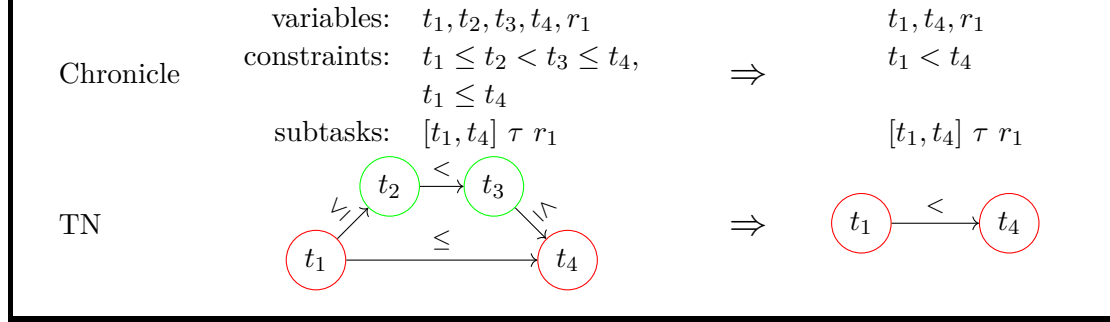
Remove useless variables During the translation of a program into a *flow graph* many temporal and result labels are created. The set of variables is analyzed to remove any synthetic variable l that is not in the set of *constraints*, *conditions*, *effects*, or *subtasks*.

Simplification of the temporal network of the chronicle During the conversion process, some temporal constraints involve timepoints that are no longer referenced in the rest of the chronicle. We call these timepoints *ghost timepoints* and propose an algorithm to check if such timepoints can be removed without changing the properties of the chronicle. For this purpose, we use Point Algebra (PA) (Gerevini 2005) to first check the path consistency of the Temporal Network (TN) resulting from the temporal constraints of the chronicle, and then to shrink the set of constraints by removing ghost timepoints. The details of the algorithm are given in Appendix A.3.

The Example 4.2 shows the simplification of the temporal network of a simple chronicle composed of ghost timepoints. The two ghost timepoints (in green) can be safely removed from the chronicle, and three constraints have also been removed.

Example 4.2

The chronicle shown below contains two ghost timepoints t_1 and t_2 , that can be removed from the chronicle. The path $p(t_1, t_4)$ is updated by calculating $p(t_1, t_4) \cap (p(t_1, t_2) \circ p(t_2, t_3) \circ p(t_3, t_4)) = \leq \cap (\leq \circ < \circ \leq) = <$.



4.3.4 Additional SOMPAS' features to model planning problems

The synthesis of chronicles based on the body of methods should allow the planning engine to anticipate the decisions of OMPAS. However, the model of the methods alone is not sufficient to take into account the effects of the execution of the commands on the robotic platform. Therefore, the planner should be provided with a model of the commands.

Here we propose to provide models for commands defined with the same procedural language used to define the body of methods, i.e. SOMPAS. In particular, we propose to use the same techniques described above to obtain a corresponding planning model of the commands. To do this, we add new operators and constructs to SOMPAS to provide models for commands. In addition, we have added the possibility to define the model of a task, so that the task is considered as a command by the planner, and should reduce the size of the planning problem, since the task should not be refined anymore.

4.3.4.1 Modeling commands

Here, we provide facilities to define the model of a command as a program, for which we can use to the same techniques to derive a planning model. The model of a command can be defined in a PDDL (Fox and Long 2003) fashion using the operator (`def-command-pddl-model` label params pre-conditions effects). For the *pick* command of the *Gripper-Door* domain, we have:

```
(def-command-pddl-model pick
  (:params (?b ball) (?r room) (?g gripper))
  (:pre-conditions
    (= (pos ?b) ?r)
    (= (at-roby) ?r)
    (= (carry ?g) empty))
  (:effects
    ('carry ?g ?b)
    ('pos(?b) roby)))
```

The fields of the operator are defined as follows:

- label is the name of the command (the command should be defined before),

- `params` is an expression of the form `(:params (p1 t1) ... (pn tn))` defined by parameter and type pairs,
- `pre-conditions` is an expression of the form `(:pre-conditions pc1 ... pcn)`, where each `pci` is a boolean expression,
- `effects` is an expression of the form `(:effects e1 ... en)`, where each `ei` is an expression of the form `(sf p1 ... pn v)` that defines the new value v of the state variable $sf(p_1, \dots, p_n)$.

The model can also be specified as a program to represent more complex models with the operator `(def-command-om-model label params body)`, where `label` and `params` are the same, and `body` is a program. The `pick` command can therefore be represented as:

```
(def-command-om-model pick
  (:params (?b ball) (? room) (?g gripper))
  (:body (do
    (check (= (pos ?b) ?r))
    (check (= (at-robbby) ?r))
    (check (= (carry ?gripper) empty))
    (effect 'carry ?g ?b)
    (effect 'pos ?obj robbby))))
```

Primitives to represent state transitions The *effect* primitive is used to describe the effects of a command on the observed world. The operator is similar to *assert*, but *effect* can only be used in models and cannot be invoked at runtime. In addition, *effect* is used to model changes to state variables that are only observed by the system, meaning that the system should not be able to deliberately change their values. The operator can be invoked as `(effect sf p1 ... pn v)`, where the first arguments (sf, p_1, \dots, p_n) define the state variable that takes the value v . Here is an example where the location of robbby is updated to the value *bedroom*:

```
(effect 'at-robbby 'bedroom)
```

Durative effects can also be modeled with `(durative-effect d sf p1 ... pn v)`, where the new argument d is the duration of the transition in seconds. Going back to our example, we could say that the transition from its previous value takes ten seconds.

```
(durative-effect 10 'at-robbby 'bedroom)
```

4.3.4.2 Modeling abstract tasks

In addition to modeling commands, models for tasks can be defined. The planner will rely on these models to refine a task, rather than choosing a method. In cases where an abstract model is sufficient to capture the essence of the task, or where the method of a task cannot be converted, it may facilitate the planning problem to define

a higher-level model for a task. Such a model can be defined with `def-task-pddl-model` and `def-task-om-model`, which are the counterparts of `def-command-pddl-model` and `def-command-om-model` for a task, respectively. Note that the sole purpose of the model is to abstract the hierarchical decomposition of a high-level task with a model that is used only by the planner. The acting engine will rely on the methods of the task to refine it during execution. Therefore, by relying on an abstract model of a task, the planner will not be able to guide the refinement of that task.

4.4 Unique representation of the executed and anticipated processes

In chapter 2 we introduced the *acting tree*, which is a representation of both the execution traces and the deliberation of OMPAS. It extends the HTN formalism to represent other deliberation functions of OMPAS, such as the acquisition of a resource and the arbitrary instantiation of a variable.

However, the purpose of the *acting tree* is broader than simply representing execution; it is also used to represent the anticipated processes that the system can perform. The *acting tree* is used to link the execution of programs in the Execution Manager, and the possible execution flows and informed decisions coming from the PLanner Manager.

Using the anticipated processes, the Acting Manager can guide some deliberation functions of OMPAS, such as the refinement of tasks, the instantiation of free parameters in the body of methods, and the allocation order of resources.

To represent both executed and anticipated processes, the *acting tree* uses a special formalism. Each call to a deliberation function is modeled as an AP whose value depends on whether it was instantiated by the execution or anticipated by the planner. Both the execution and the planner can update the *acting tree* by adding new acting processes. However, updates coming from the Execution Manager are considered static by the PLanner Manager. On the other hand, the Execution Manager is free to ignore the guidance offered by the planner if it does not match the context when the decision is actually made at runtime. Now we will present the formalism in detail.

4.4.1 Acting Process (AP)

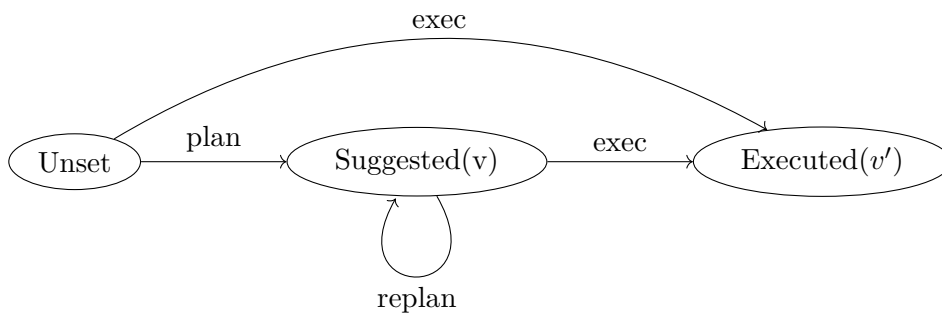
Each call to a deliberation function is modelled as an AP. An AP is defined by a unique ID, an execution interval and a set of Acting Variables (AVs) that are associated with it, whose values depend on whether they were instantiated by the PLanner Manager or the Execution Manager. We distinguish the following APs:

- *Root*: virtual task that represents the overall execution of OMPAS. All top-level tasks are children of the *Root* process,
- *Task*: represents the execution of a task τ ,
- *Method*: represents the execution of the program to refine a task τ ,

- *Command*: represents the execution of a command on the *Robotic Platform*,
- *Acquisition*: represents the acquisition cycle of a resource, which encapsulates the waiting, the use of the resource, and its release,
- *Arbitrary*: represents the instantiation of a free variable in the body of a method. We assume that an arbitrary process is instantaneous.

4.4.2 Acting Variable (AV)

An Acting Variable (AV) is a variable representing a choice that can have three states: *Unset*, *Observed(v)* or *Executed(v')*. Here is the state transition of an AV:



The AV starts in the state *Unset*. When the value v' is instantiated by the Execution Manager, then the AV is in the state *Executed(v')*. The AV is considered as a constant by the other managers of OMPAS. The planner can suggest the value v by setting the AV to the state *Suggested(v)*. After replanning, the value v can be updated and remains in the state *Suggested(v)*.

4.4.3 Acting tree

OMPAS simplifies the integration of a planner in a continuous way by providing a unique model on which both the execution and the planner can work. The tasks addressed by OMPAS are represented by their hierarchical structure in an acting tree. An *acting tree* is an AND-OR tree where the nodes are APs. An example of an *acting tree* is shown in Figure 4.5. In this *acting tree*, the generic representation of a node is as follows:

$$\boxed{ID: [s, e] \text{ process}}$$

As with APs, a node is uniquely defined by an *ID*, a start timepoint s , and an end timepoint e .

The *acting tree* consists of several types of nodes. We distinguish between *abstract nodes* and *leaf nodes*:

- The *abstract nodes* represent processes whose outcomes depend on subprocesses. This type of process includes *Root*, *Tasks*, and *Methods*. In Figure 4.5, they have dotted boundaries.

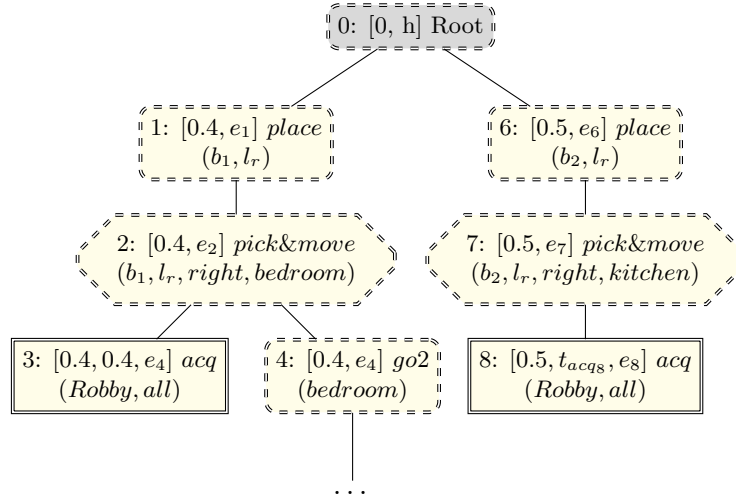


Figure 4.5: An example of an *acting tree* representing the execution of two tasks t_1 : $place(b_1, l_r)$ and t_2 : $place(b_2, l_r)$ (see Example 2.7). Both tasks are running in parallel, and both tasks have requested the exclusive use of *Robby*. OMPAS received the task t_1 at 0.4s (time relative to the start of the execution). The task t_1 was refined into $pick\&move(b_1, l_r, right, bedroom)$, which immediately requested the resource *Robby*. The resource was granted immediately (at relative time 0.4s) and the method requested the execution of the task $go2(bedroom)$. The task t_2 has been refined with the method $pick\&move(b_2, l_r, right, kitchen)$. The method is waiting for the resource *Robby* to continue its execution. All processes are currently running.

- The *leaf nodes* are processes whose termination depends on the call of a unique function. The *Command*, *Acquisition*, and *Arbitrary* nodes are *leaf nodes*. In Figure 4.5, they have simple borders.

The nodes corresponding to each AP are presented in Table 4.1. The table presents their definitions and the set of specific APs that define them.

A node also represents the status of an AP, which can be one of the following:

Running (yellow)	The process started, which means that the start timepoint and the parameters of the process are <i>observed</i> .	$ID: [s, e] process$
Executed (green)	The process has been successfully executed, all parameters are <i>observed</i> thanks to the traces of execution.	$ID: [s, e] process$
Failed (red)	The process failed, all parameters are <i>observed</i> , and the final timepoint represents the moment the process failed.	$ID: [s, e] process$
Planned (blue)	The process has been anticipated by the planner, and is added in the acting tree. Values of the parameters are <i>suggested</i> .	$ID: [s, e] process$

Name	Definition	Node
<i>Root</i>	Root AP whose children are all the top-level tasks.	$\boxed{\boxed{0:[0, horizon]Root}}$
<i>Task</i>	Task AP whose children are the methods that can refine τ : the tried ones, the currently executed, and the anticipated ones.	$\boxed{\boxed{Id:[s_t, e_t] \tau(p_1, \dots, p_n)}}$
<i>Method</i>	A method m that refines a given task τ . Its children are all the APs that are part of the body of the method, e.g. the call to execute a command.	$\boxed{\boxed{Id:[s_m, e_m] m(p_1, \dots, p_n)}}$
<i>Command</i>	A <i>leaf node</i> that represents the execution of a command c with parameters (p_1, \dots, p_n) during the interval $[s_c, e_c]$	$\boxed{\boxed{Id:[s_c, e_c] c(p_1, \dots, p_n)}}$
<i>Acquisition</i>	A <i>leaf node</i> that represents the acquisition cycle of a resource r with amount q . Three timepoints represent the acquisition: t_{req} is the moment the request has been sent, t_{acq} is when the resource has been acquired, and t_r is the instant of release.	$\boxed{\boxed{Id:[t_{req}, t_{acq}, t_r] acq(r, q)}}$
<i>Arbitrary</i>	Represents the call to the acting primitive <i>arbitrary</i> . It contains the <i>set</i> from which the value v has been chosen.	$\boxed{\boxed{Id:[t_a] arb(set) = v}}$

Table 4.1: Definition and graphical representation of the APs in the *acting tree*.

4.4.4 Linking the acting tree to the execution and planning models

As mentioned above, both the Execution Manager and the PLanner Manager can update the acting tree. However, a protocol is needed so that both managers can refer to the same processes in the tree.

4.4.4.1 Labeling the operational models

To refer to the same processes, the Execution Manager (EM) is the one that decides what will be the labels of each deliberation call in a given program. At runtime, the EM labels each program before evaluating it. The labeling process works as follows: each call to an acting primitive is replaced by a corresponding labelled call, which has exactly the same semantics as the acting primitive with an additional annotation on the ID of the call relative to the program. The labeling function takes any expression of the form (prime p1 ... pn), where prime is a call to an acting primitive, and transforms it into (l-prime id p1 ... pn), where id is unique for the primitive prime in the context of the program. Here is an example of a program (on the left) with several calls to acting primitives and their labelled counterparts (on the right):

<u>Raw program:</u>	<u>Labelled program:</u>
<pre>(begin (define ?r (arbitrary (instances robot))) (define h1 (acquire ?m)) (define h2 (acquire ?r)) (exec-task 'carry ?r ?p ?m) (release h2) (exec-task 'process ?m ?p))</pre>	<pre>(begin (define ?r (l-arbitrary 0 (instances robot))) (define h1 (l-acquire 0 ?m)) (define h2 (l-acquire 1 ?r)) (l-exec-task 0 'carry ?r ?p ?m) (release h2) (l-exec-task 1 'process ?m ?p))</pre>

We can see that the different calls to the `acquire` and `exec-task` acting primitives have been labeled differently.

In the current version of the system, the annotation system does not support the generation of expressions at runtime that contain new calls to acting primitives.

Each AP is associated with a unique *ID* when it is added to the acting tree. However, if case an AP has been anticipated by the planner, the EM should be able to refer to it without prior knowledge of its *ID*.

We propose the following protocol for referring to an AP by its position in the acting tree. For a given AP a_p , its reference is the path in the acting tree from the process *Root* to a_p . The path is defined by a set of labels, and refers to a unique child at each level of the tree. A label is defined by the type of child (Task, Command, Method, Arbitrary, Acquire) and a unique number, so that no two children of a node can have the same label. In Figure 4.5, we can refer to the acquire process of m_1 by its unique position in the tree with either its unique *ID* (3), or by the path from *Root* to the process (0/Task(0)/Refinement(0)/Acquire(0)).

4.4.4.2 Acting Process Binding (APB)

During the synthesis of the chronicles, the labeled calls to acting primitives of the programs are used to define an Acting Process Binding (APB). An APB binds a subset of the variables of the chronicle to an AP a_p of the acting tree. It is defined by the reference of a_p (either an *ID* or a path) and the variables of V corresponding to the AVs of a_p . We extend the definition of the chronicle with the field *Bindings* (B), which contains these APBs.

The APB are only defined for chronicles that represent processes that can have children. In this case they are *Method* processes. Therefore, for each type of child of a Method process (Task, Command, Acquisition, Arbitrary), we have a corresponding APB.

4.5 Guidance of the reactive deliberation of OMPAS using a planner

In OMPAS, calls to acting primitives can be triggered in the body of methods that are evaluated in the Execution Manager (EM). The Execution Manager (EM) may request guidance from the Acting Manager (AM), usually to instantiate the parameters of the corresponding Acting Process a_p . The guidance may use reactive algorithms, or take advantage of the processes anticipated by the planner. If a_p was anticipated by the planner, AM informs EM of the choices made by the planner. If the choices are still valid in the current state ξ , EM uses those choices to instantiate a_p . Otherwise, reactive algorithms are used to find a solution that is valid in ξ . In either case, the decisions made by EM are integrated into the *acting tree* as a set of *executed* AVs.

4.5.1 Scope of the planning problem

Local planning

A planning problem can be instantiated to solve locally a deliberation choice faced by OMPAS. Here, it is limited to refining one task. AM can refine a task τ by calling the function `PLANSELECTMETHOD(τ , tried)` (see Algorithm 2.8) to find a method m to refine τ . The PLanner Manager (PLM) encodes the corresponding problem to find a suitable solution. To do this, the PLM has access to the acting domain A_Δ to populate the planning problem (P_Π). In this case, the execution state of the other tasks is not taken into account. The State Manager provides the current state ξ , which defines the initial state in P_Π . The planner should return an answer in the allotted time¹, otherwise the Acting Manager (AM) will resort to another (reactive) heuristic to solve the refinement of the task. We assume that the facts of ξ hold during the allotted time given to the planner, so that the solution found by the planner is still valid despite the time that has elapsed for the search.

¹This value can be configured in OMPAS using a configuration file loaded via the REPL.

Global planning Local planning can provide good heuristics for avoiding local failures, and optimizing the time to complete a task. However, it has been shown in Example 2.3 that the anticipation of a given task τ should also take into account the interactions with the other tasks currently executed.

Here the planner is used in a continuous way to adapt to the evolution of the system and to guide the not yet executed APs. The planning problem (P_{Π}) is composed of the hierarchical decomposition of all running tasks, obtained with the *acting tree*, and is based on the current state ξ of the system. The analysis of the returned plan should give:

- the most promising method for each task not yet refined,
- instantiation for arbitrary variables,
- an access order for resources shared by concurrent tasks.

To do this, PLM constantly monitors updates to the *acting tree* and on the state of SM. When a new process is added to the *acting tree*, a new planning instance is created. The planning instance runs until either a plan is found, or until PLM determines that the updates to the *acting tree* and the recent observations received by the State Manager render the previous P_{Π} obsolete. In this case, PLM starts a new planning instance with an updated P_{Π} . With this configuration of PLM, the planner is constantly running a planning instance in the background, which will hopefully provide useful heuristics for the processes to run in the future.

In addition to anticipating the execution of processes, the planner is used to optimize the total time to execute all tasks in the *agenda*, i.e. all the top-level tasks defined in the *acting tree*.

4.5.2 Instantiation of the planner

In section 4.2.3.2 we introduced how a planning problem (P_{Π}) is instantiated in the *Aries* planner. In the following paragraphs, we will show how P_{Π} is constructed by the PLanner Manager (PLM) using the state given by the State Manager (SM), the current state of the resources given by the Resource Manager (RM), and the *acting tree* defined in the Acting Manager (AM). Let us see how the problem is instantiated at a given time t_{π} .

Instantiation of C_0 The initial chronicle C_0 is instantiated using the current state of the system returned by SM. Since the state returned by SM is composed of facts, the encoding of the planning problem is fairly straightforward. For each fact of the form (t, sv, v) , a new effect $[t] sv := v$ is added. Note that t is the last time sv was updated. To reduce the size of the coded problem, only the state variables referred to by the chronicles in C_I are added to C_0 .

Since typed objects can be defined in OMPAS, we add a synthetic state function that specifies the type t of an object o : $[0] instance(o) := t$. Since the fact is always true, we state that the state variable is true from the beginning.

The state of the resources is encoded using the two synthetic state functions *quantity(?r)* and *max-q(?r)*:

For each resource $?r$ defined in the Resource Manager, we add the effects:

- $[t]$ $quantity(?r) := q$, where $t = t_\pi$
- $[0]$ $max-q(?r) := C$

Goals are defined using the *acting tree*: it is the set of high-level tasks T_H , i.e. the children of *Root*, that are currently running. For each task $\tau_i \in T_H$, we define a *goal task* in C_0 , i.e. a subtask $[s_i, e_i] \tau(p_{i_1}, \dots, p_{i_n})$ is added to the *Subtasks* field of C_0 . Note that s_i and p_{i_j} are already instantiated because τ_i is running.

Example 4.3

Let us take the problem given in Example 2.3. We want to plan at $t_\pi = 0.7$. Based on the *acting tree* shown Figure 4.5, we have the following C_0 :

constraints:	$e_1 \leq \mathcal{H}, e_6 \leq \mathcal{H}$
effects:	$[0]$ $pos(b_1) := bedroom,$ $[0]$ $at-robby() := l_r,$...
	$[0.7]$ $quantity(Robby) := 0$
	$[0]$ $max-q(Robby) := 1$
subtasks:	$[0.4, e_1]$ $place(b_1, l_r)$ $[0.5, e_6]$ $place(b_2, l_r)$

Since *Robby* was acquired by the process acq_1 , $quantity(robby) := 0$. *Robby* is a unary resource, so $max-q(robby) := 1$.

Populating C_I For each program that is currently running, there is a corresponding chronicle representation stored in the Acting Manager. C_I is composed of all the chronicles representing the process currently marked as running in the acting tree, as well as the processes that have been previously anticipated by the planner, and that are still valid.

To construct C_I , the PLanner Manager traverses the *acting tree*, and for a given node it applies one of the following rules:

- The node is a *Task* τ : τ is defined as a subtask in the chronicle of the parent node. If the task is currently running, it means that a method m is refining it and the chronicle of the method C_m is added to C_I . If τ has not yet been refined, all the chronicles representing the possible refinements of τ will be added to C_I . If τ has been previously planned, these possible refinements have already been defined in the *acting tree*.
- The node is a *Command* c : c is defined as a subtask in the chronicle of the parent node, and the chronicle model of the command is added in C_I .

- The node is an *Acquire* or an *Arbitrary*: the process model is handled as part of the parent method's chronicle and does not require a separate chronicle.

Example 4.4 presents some chronicles corresponding to the processes present in the *acting tree* in Figure 4.5. In particular, we have the partial chronicles for the methods m_1 and m_2 , for which the bindings have been made explicit here.

Example 4.4

Let us take the *acting tree* defined in Figure 4.5. We represent the chronicles of the method m_1 and m_2 . This representation is only partial and aims at representing the link between chronicles and the *acting tree*. Here is the (partial) chronicle of m_1 :

```

name: pick&move( $b_1, l_r, right, bedroom$ )
task: place( $b_1, l_r$ )
variables: ...
constraints :  $e_4 \leq s_9, \dots$ 
conditions: ...
effects:  $\emptyset$ 
subtasks: [ $0.4, e_4$ ] go2( $bedroom$ )
           [ $s_9, e_9$ ] pick( $b_1, bedroom, right$ )
           [ $s_{10}, e_{10}$ ] go2( $l_r$ )
           [ $s_{11}, e_{11}$ ] drop( $b_1, l_r, right$ )
bindings: 3 : ( $0.4, 0.4, t_{r3}, Robby, 1$ )
           4 : (( $0.4, e_4$ ), ( $go2, bedroom$ ))
           2/Command(1) : (( $s_9, e_9$ ), ( $pick, b_1, bedroom, right$ ))
           2/Task(2) : (( $s_{10}, e_{10}$ ), ( $go2, l_r$ ))
           2/Command(2) : (( $s_{11}, e_{11}$ ), ( $drop, b_1, l_r, right$ ))

```

Here is the (partial) chronicle of m_2 :

name:	$pick\&move(b_2, l_r, right, kitchen)$
task:	$place(b_1, l_r)$
variables:	\dots
constraints:	$e_4 \leq s_9, \dots$
conditions:	\dots
effects:	\emptyset
subtasks:	$[s_{12}, e_{12}] go2(kitchen)$ $[s_{13}, e_{13}] pick(b_1, bedroom, right)$ $[s_{14}, e_{14}] go2(l_r)$ $[s_{15}, e_{15}] drop(b_1, l_r, right)$
bindings:	$8 : (0.5, t_{a_8}, t_{r_8}, robbly, 1)$ $6/Task(1) : ((s_{12}, e_{12}), (go2, bedroom))$ $6/Command(1) : ((s_{13}, e_{13}), (pick, b_1, bedroom, right))$ $6/Task(2) : ((s_{14}, e_{14}), (go2, l_r))$ $6/Command(2) : ((s_{15}, e_{15}), (drop, b_1, l_r, right))$

Using C_Δ to populate C_I For all subtasks $s \in S$ such that $S = \bigcup_{c \in C_0 \cup C_I} Subtasks(c)$, and s does not have a chronicle refining it in C_I (i.e. no method or command exists in the acting tree yet), the PLM adds a new chronicle instance that can refine s in C_I . If s is a task, it adds a chronicle in C_I for each method that can refine τ . Otherwise, s is necessarily a command, and the chronicle that models the command is added to C_I . These chronicles are created using the chronicle templates defined in C_Δ .

4.5.3 Extraction of the decisions from the plan

The plan returned by *Aries* consists of instantiated chronicles that should be analyzed to extract:

- the refinement of abstract tasks,
- the selection of arbitrary values,
- the order of acquisition of resources.

We rely on the fact that *Aries* encodes the chronicles as a CSP, where each possible plan decision is associated with a corresponding variable. With such an encoding, we extract the decisions from the plan by obtaining the value of the variables of the APBs proposed by *Aries*. In this way, only the use of these APBs is necessary to extract useful information to guide OMPAS. For each AV already instantiated by the Execution Manager (EM), the value found by the planner is used in the *acting tree*. The value it takes is $Planned(v)$, where v is the value of the variable, e.g. "left" for the value of the gripper parameter of a command. So the PLanner Manager returns the instantiated APBs to the Acting Manager. For processes that do not yet exist in the *acting tree*, e.g. a method for a task that has not yet been refined, the PLanner Manager adds an appropriate node to the *acting tree* and the AVs are instantiated using the APBs of the process.

Reservation of a resource In OMPAS, acquisitions can be anticipated by the planner. This results in an acquisition request called a *reservation*. A *reservation* is an acquisition request that is placed in the waiting queue of a resource. In response to an acquisition request coming from the Execution Manager (EM), the Acting Manager (AM) can pass the reservation ticket of a resource to EM. Until the reservation is passed to EM, the ticket remains in the waiting queue.

Reservations are extracted from the plan by analyzing all bindings of *Acquire* processes. These *Acquire* processes are used to sort the order of resource access requests: for all requests of a common resource r , the system sorts the reservations using t_{acq} in descending order, so that the earliest t_{acq} has the highest priority. The ordered reservation list is then sent to RM, which sends the reservation tickets back to AM. Example 4.5 should illustrate how this works.

Example 4.5

Consider a resource r that is scheduled to be acquired three times in the plan. Let us define the acquisitions a_1 that should happen at 10 seconds, a_2 at 13 seconds and a_3 at 12 seconds. The PLanner Manager sorts them from smallest to largest, we get the sorted list (a_1, a_3, a_2) . Given that the base priority of a reservation is 10, we get the corresponding reservations a_1 with priority (13), a_3 with priority (12), and a_2 with priority (11).

4.5.4 Continuous update of the planner

In addition to the initial creation of the planning problem (P_{Π}), the PLanner Manager (PLM) is responsible for instantiating new planning instances in function of the arrival of new runtime information. In particular, PLM monitors the instantiation of AVs, the update of the state, and the status of tasks. If new information renders the currently running planning problem (P_{Π}) obsolete, PLM interrupts the planner and then creates a new planning instance that reflects the latest updates received by PLM. If no planning instance is running, PLM immediately creates a new planning instance with the current state of the system. Here, we detail how P_{Π} is updated based on new information received by PLM.

Variable instantiation Executing an Acting Process in the Execution Manager results in the instantiation of Acting Variables in the *acting tree*. For an Acting Variable var whose value has become *Executed*(v), an additional constraint $var = v$ is added to the chronicle in which var is present.

Update of the state When the state is updated, the initial chronicle C_0 is modified to take into account the last values of all updated state variables. Since only the last value of a state variable is taken into account, this means that the previous effect is removed from C_0 .

Durative effects Some actions are defined with transitive effects. They represent a durative transition of a state variable between a previous value and a new value, without modeling all the transitory values of a state variable.

In reality, the state variable can take any number of values between the start of the effect and its end. Therefore, such a model may conflict with the continuous value update of a state variable as perceived by the *Robotic Platform*.

For example, consider the durative effect $e_1 : [s, e] \text{ pos}(b_1) := \text{bedroom}$, with $e = s + d$, where $d \in \mathbb{R}_+^*$, that is part of the model of a *drop* command. The effect $e_2 : [t_{obs}] \text{ pos}(b_1) := \text{robby}$ encodes the perceived state such that $s < t_{obs} < e$. As encoded in *Aries*, two effects on the same state variable cannot occur simultaneously, which means that if e_1 and e_2 are present in P_{II} , the planner cannot find a valid solution.

To handle these situations, we suggest the following rule: For $e_1 : [t] \text{ sv}_1 := v_1$, an effect of C_0 : if $\exists e_2 : [s, e] \text{ sv}_2 := v_2$, an effect present in a chronicle of C_I , such that $s \leq t \leq e \wedge \text{sv}_1 = \text{sv}_2$, we remove e_1 from C_0 . In this way, we preserve the predictive property of the model by preserving the effect that represents the final value of the state variable. The opposite would preserve the transient value of the state variable as perceived by the system, which would invalidate the planning model.

Handling subtasks When a new high-level task is received, the chronicle C_0 is updated to reflect the new task. The binding of the new task is also added to C_0 . If a subtask is terminated, this means that its *end* timepoint has been instantiated. In this case, we remove the subtask from its parent's chronicle, but keep the time constraints on its interval.

4.6 Conclusion

In this chapter we presented how a planning system can be used to guide the operational deliberation of OMPAS. The proposal consists in binding the execution of a program to a corresponding planning model by relying on an intermediate representation of the deliberation of the acting system.

As a planning formalism, we rely on chronicles, which have a rich temporal semantics and which have been extended to facilitate bindings with the *acting tree*. They are particularly suitable for representing problems where we expect the planner to guide the allocation of resources, the refinement of tasks, and the instantiation of free variables. The planning problem is constructed by scanning the *acting tree* for all running tasks and commands; the current state of the system is based on the information provided by the State Manager and the Resource Manager. Although the approach aims to be planner independent, we decided to focus our efforts on a strong integration with the planner *Aries*, which can take advantage of the hierarchical formalism and the temporal expressiveness of the chronicles. The resulting implementation benefits from the interface *Aries* provides to its solver, on which OMPAS has a hand to define dedicated search strategies.

However, the real novelty is the ability of the system to automatically generate the

planning models on the fly, based on the programs of the current tasks and methods. The process is based on the automated analysis of the programs, which are first transformed into a *flow graph*. This is a representation of a program in a form that is particularly suitable for static analysis, some inherent to compilers, others specialized for planning. The *flow graph* is then compiled into a chronicle, which is cleaned of artifacts before being handed over to the planner. With such a proposal, it is sufficient to define the capabilities of an agent in the form of a program and benefit from planning techniques to improve the performance of the acting engine, without specifying the corresponding planning models of the agent.

In the following chapter, the whole approach is evaluated and compared with other algorithms proposed in the literature. The evaluation will cover the system's ability to perform a high-level task as quickly as possible, its ability to adapt to exogenous events, and to take advantage of its lookahead capabilities at runtime.

Evaluation of OMPAS: Simulated Domains and Integration with a Factory Simulator

Contents

5.1	Introduction	143
5.2	Setup of the evaluation	144
5.2.1	Measured metrics	144
5.2.2	Compared systems: different configurations of OMPAS	145
5.3	Gripper-Door: simulated domain	148
5.3.1	Acting Domain	148
5.3.2	Case study	153
5.4	Gobot-Sim: integration with a factory simulator	155
5.4.1	Gobot-Sim	155
5.4.2	Versions of Gobot-Sim	159
5.4.3	Integration with OMPAS	160
5.4.4	Benchmark	161
5.4.5	Case Study	163
5.5	Discussion	165
5.6	Conclusion	167

5.1 Introduction

In this chapter, we propose to evaluate the approach developed in OMPAS. In particular, we want to assess its ability to handle multiple objectives using a fleet of agents. To this end, we present the integration of OMPAS on two domains:

- The *Gripper-Door* domain for which we provide a complete definition using the SOMPAS language,
- The *Gobot-Sim* domain, a factory simulator in which a fleet of robots should move packages between machines on which they will be processed.

Each domain is used to emphasize a particular deliberation feature provided by OMPAS. The first domain requires deliberation on the choice of the course of action, whereas the second one should benefit from the allocation strategy of the temporal planner to schedule the processing of packages. First, we present the setup used to evaluate each domain: the metrics used, the compared configuration of OMPAS, and the setup of the compared domains. Then, we successively present the two domains: their formalization, the models used by OMPAS, general results, and case studies of particular problem instances.

5.2 Setup of the evaluation

To assess the pertinence of the approach developed in OMPAS, we propose to evaluate the capabilities of the acting system. To do this, we propose to benchmark OMPAS on a set of different problem instances for each of the two domains. In each problem, a set of tasks \mathcal{T} defines the tasks that the acting engine should face. To face the tasks of \mathcal{T} , we define a maximum allotted time T_M . The value is defined for each one of the domains.

With such an evaluation, we want to ensure that the proposed approach improves the efficiency of the system when performing multiple tasks in parallel. In this section, we specify the setup used to evaluate the two proposed acting domains. We present the metrics as well as the different systems that have been benchmarked.

5.2.1 Measured metrics

To measure the performance of the system, we use a set of temporal metrics that should provide useful information about how the system performs on a given problem. The metrics used are as follows:

- T_E is the total execution time of the acting system, which is the total time to complete all the tasks in \mathcal{T} ,
- T_D is the total deliberation time, which is the sum of all deliberation times during execution,
- T_P is the total planning time, which is the sum of all planning times taken by the *continuous planner*,
- T_{WP} is the total time during which the deliberation waited for the *continuous planner* to make a decision. In OMPAS, when the Execution Manager handles the call of a deliberation process and if the *continuous planner* runs in the background, the deliberation waits for the response of the planner for a bounded time. If during the period the *continuous planner* was not able to provide a plan that would guide the deliberation process, the Execution Manager resorts to a reactive deliberation strategy. This period can be defined by the programmer using the `(set-deliberation-reactivity t)` operator, where `t` is a duration in seconds.

We also examine the total number of commands N_C required to complete all the tasks.

To evaluate the efficiency of the system, we use a metric that represents the throughput of the system. Here, the efficiency of the system is defined in terms of tasks per second. We represent this throughput by an *Efficiency Score* E_S . To calculate it, we introduce the *Coverage* Cov , which is the percentage of the problem's tasks that were successfully performed. We define it as follows:

$$Cov = \frac{|\{t \in \mathcal{T}, status(t) = success\}|}{|\mathcal{T}|}$$

We remind that \mathcal{T} is the set of tasks that should be performed.

Having defined the *Coverage*, we can now define how the *Efficiency Score* E_S is calculated:

$$E_S = \frac{Cov}{T_E}$$

We refer to E_S^* as the virtual best efficiency score for a given instance. Based on the E_S^* , we also compute the relative distance to the best score for a given problem instance. We define this metric as the distance to the best efficiency score (DE_S^*). We compute it as follows:

$$DE_S^* = \frac{E_S^* - E_S}{E_S^*}$$

To compare the results on the same instance of a problem, we use the normalized efficiency score \hat{E}_S normalized by the maximum time T_M allotted to solve the acting problem, which gives:

$$\hat{E}_S = E_S \times T_M$$

We also use the normalized distance to the best score $D\hat{E}_S^*$, which is defined as:

$$D\hat{E}_S^* = \frac{\hat{E}_S^* - \hat{E}_S}{\hat{E}_S^*}$$

For both the *Gripper-Door* and the *Gobot-Sim* domains the above metrics are evaluated.

5.2.2 Compared systems: different configurations of OMPAS

We propose here to evaluate the different configurations of OMPAS against each other. OMPAS can be configured in two ways:

- The algorithm used by the SELECT function of OMPAS to select an instantiated method to refine a task,
- The management of runtime deliberation with the *continuous planner*, which runs

in parallel with execution, Here we rely on *Aries* for *continuous planning*.

5.2.2.1 Select function configuration

As presented in the Chapter 2, OMPAS is equipped with several configurations for the SELECT function. As a reminder, SELECT (see Algorithm 2.2) takes as input an instantiated task τ , which it refines depending on the current state ξ . First, it generates the instantiated methods that are applicable in ξ . We call the set of applicable methods $Applicable(\tau, \xi)$. We then define $Candidates = Applicable(\tau, \xi) \setminus tried$, where *tried* is the set of methods already tried. Based on *Candidates*, we then select an element of the set. The selection depends on one of the following strategies:

- The *Greedy* strategy selects the first element of the set.
- The *Random* strategy selects a random element from the set.
- The *Cost* strategy selects the element with the lowest cost. If two items have the same cost, an arbitrary one is picked.
- The *UPOM* strategy relies on the anytime planner *UPOM* (Patra, Mason, Ghallab, et al. 2021), which has been reimplemented in OMPAS.
- The *Aries* strategy relies on the planner *Aries* used to resolve the hierarchical decomposition of the task locally and select the method chosen by the planner in its plan.

UPOM: anytime planner To compare our approach with the literature, we reimplement the algorithm of *UPOM* in OMPAS. The implementation of *UPOM* in OMPAS uses the same algorithm as described by Patra et al. (Patra, Mason, Ghallab, et al. 2021).

As a reminder, *UPOM* is an anytime planner that samples various execution flows. It is used to find a method m to refine a task τ so that m maximizes some utility. The utility can be defined in two ways:

- The *Efficiency*, which is the inverse of the cost of a method. The cost of a method is defined as the sum of the costs of its subtasks.
- The *Robustness*, which represents the probability of success of a method.

In our case, we compute the *Efficiency* of a method. The exploration of the execution flows is done in a Monte Carlo Tree Search (MCTS) in which at each node of the tree, a task is sampled by simulating the execution of the possible methods that can refine it. The result of the simulation gives the value of the utility of the method. The efficiency of a method depends on the utility of its subtasks. In particular, it depends on the efficiency of the instructions, which is based on the cost of the instructions. Here, the cost of the commands is sampled using the cost model provided by the programmer. If no cost model has been provided, the default cost for a command is set to one if the

command is applicable, zero otherwise. If a method cannot be sampled, for example because the method is a terminal node of the MCTS, then *UPOM* can also rely on the cost model of the method. This model can also be provided by the programmer. If it is not provided, we assume that an unsampled method has a cost of zero.

UPOM can be configured with the following parameters:

- d_{max} is the maximum depth of a branch to explore in the MCTS,
- n_{ro} is the number of rollouts of *UPOM* to refine a given task,
- C is the tradeoff between exploring less sampled method instances (high C) and exploiting more promising ones (low C).
- *timeout*, which is the maximum amount of time given to *UPOM* to sample the execution of the various methods. Here the timeout is defined by the *deliberation reactivity* of OMPAS,

We do not use *UPOM* with iterative exploration, i.e. *UPOM* always explores to the maximum depth limited by the parameter d_{max} . For the evaluation of OMPAS we use the same values for the parameters of *UPOM* for all configurations, i.e:

$$\begin{aligned}d_{max} &= 10 \\n_{ro} &= 10 \\C &= 2\end{aligned}$$

5.2.2.2 Continuous planning configuration

As mentioned above, OMPAS can take advantage of *continuous planning* to anticipate the flows of execution of OMPAS. Planning is thus a concurrent process with execution. *Continuous planning* has three different configurations:

- The *Deactivated* configuration does not use the *planner* in the background,
- The *Satisfactory* configuration, which calls the *planner* in the background each time a new planning process is created by the PLanner Manager, runs the *planner* to find a first plan that satisfies the planning problem. When the *planner* finds a plan, it is added to the *acting tree* of the Acting Manager, which can then be accessed by the deliberation processes to get the *planner*'s guidance. For more details on how to create the planning instance, see the Chapter 4. In this configuration, the *planner* has a minimum amount of time during which it cannot be interrupted. After this time has elapsed, the PLanner Manager is free to interrupt the planner at any time,
- The *Optimality* configuration, which has the same triggering and interrupting rules as the *Satisfactory* configuration, but in which the *planner* is configured to return a plan that optimizes the total makespan. In this configuration, the *planner* may return multiple plans, each of which improves the optimal metrics over the previous

	<i>Deactivated</i>	<i>Satisfactory</i>	<i>Optimality</i>
<i>Greedy</i>	$C(G)$	$C(G/Sat)$	$C(G/Opt)$
<i>Random</i>	$C(R)$	$C(R/Sat)$	$C(R/Opt)$
<i>Cost</i>	$C(C)$	$C(C/Sat)$	$C(R/Opt)$
<i>UPOM</i>	$C(G)$	$C(U/Sat)$	$C(U/Opt)$
<i>Aries</i>	$C(A)$	$C(A/Sat)$	$C(A/Opt)$

Table 5.1: Table of configurations used in the OMPAS evaluation. The rows indicate the algorithm used by the SELECT function, while the columns indicate the configuration of the *continuous planning* function.

one. In this case, the plan returned is the last plan returned by the *planner*. In some cases, the *planner* may exhaust the search space and return an optimal plan.

When configuring *continuous planning* we can set the reactivity of the planner, i.e. the maximum time during which a planning instance cannot be interrupted. This is a domain dependent setup¹, which is typically defined in the domain file loaded in OMPAS.

To easily refer to a particular configuration of OMPAS, we suggest using the table 5.1, which defines unique names for the configurations. Note that the table represents all the different configurations of OMPAS that we provide along this thesis. These configurations are referred to as $C(a/b)$, where a refers to the strategy of the SELECT function, while b refers to the configuration of the *continuous planning* feature. For example, $C(C/Opt)$ refers to the configuration that uses the *Cost* strategy and the *Optimality* configuration for *continuous planning*.

Now that we have presented how OMPAS is evaluated, we present in detail each of the domains, the results and a study-case that should highlight the functioning of the deliberation of OMPAS.

5.3 Gripper-Door: simulated domain

Throughout the presentation of this work, we have used the *Gripper-Door* domain to illustrate the workings of OMPAS. We continue to use it here to evaluate the deliberation capabilities of OMPAS.

5.3.1 Acting Domain

Based on the definition of the domain provided in Chapter 1, we provide the acting domain of the *Gripper-Door* domain defined in SOMPAS. The domain is defined in a single file consisting of the following program:

¹The SOMPAS function to set the reactivity of the planner is `(set-planner-reactivity t)` where t is a duration in seconds.

```
(begin
  (define gripper-door-path
    (concatenate (get-env-var "OMPAS_PATH") "
                  /domains/gripper_door"))
  (set-current-dir gripper-door-path)
  (load base.scm)
  (set-current-dir gripper-door-path)
  (load om.scm)
)
```

In this program the working directory depends on the main directory of OMPAS, called *OMPAS_PATH*, from which the files `base.scm` and `load.scm` are loaded.

The file `base.scm` contains all the basic definitions of the domain, namely the types, state functions and commands. Here we show part of the file where the type `door` is declared, along with the state functions `opened(?d)` and `connects(?r1, ?d, ?r2)`, and the command `move(?from, ?to, ?d)` with its model.

```
(def-types door)
; Additional state functions
(def-state-function opened
  (:params (?d door))
  (:result boolean))

(def-function connects
  (:params (?r1 room) (?d door) (?r2 room))
  (:result boolean))

; New commands
(def-command move (:params (?from room) (?to room) (?d door)))
(def-command-pddl-model move
  (:params (?from room) (?to room) (?d door))
  (:pre-conditions
   (= (at-robby) ?from)
   (connects ?from ?d ?to)
   (opened ?d))
  (:effects
   (durative 5 'at-robby ?to)))
```

The second file `om.scm` contains the operational model. In this file, we can find the declaration of the task `place(?o, ?r)` with its methods. Other tasks and methods are defined, which are used to refine the subtasks of `place(?o, ?r)`, such as task `go2(?r)`.

```
(def-task place (:params (?o carriable) (?r room)))

(def-method place_noop
  (:task place)
  (:params (?o carriable) (?r room))
  (:pre-conditions (= (pos ?o) ?r))
  (:body nil))

(def-method pick_and_drop
```



```

(:task place)
(:params (?o carriable) (?r room))
(:pre-conditions (!= (pos ?o) ?r) (!= (pos ?o) robbly))
(:body
  (do
    (define ?g (arbitrary (instances gripper)))
    (define res_g (acquire ?g))
    (define rh (acquire 'robbly))
    (define ?a (pos ?o))
    (go2 ?a)
    (pick ?o ?a ?g)
    (release rh)
    (define rh2 (acquire 'robbly))
    (go2 ?r)
    (drop ?o ?r ?g)
    (release rh2)
    (release res_g))))

```

A cost model is also provided by the operational model. Here we use an ad hoc state function *min-distance(?r1: room, ?r2: room)*, which gives the minimum number of moves to go from room *?r1* to room *?r2*. Here, we use precomputed values for *min-distance* that are loaded with the rest of the values of a given problem.

This heuristic can be used by the configuration that relies on the cost-based selection strategy. *UPOM* can use it to have an estimated cost of the task *go2(?r)*. In the following listing, we give the model of the method *m_move(?r, ?a, ?n, ?d)* that can refine the task *go2(?r)*.

```

(def-method m_move
  (:task go2)
  (:params (?r room) (?a room) (?n room) (?d door))
  (:cost (+ 1 (min-distance ?n ?r)))
  (:pre-conditions
    (= (at-robbly) ?a)
    (!= ?a ?r)
    (connects ?a ?d ?n))
  (:body
    (do
      (t_open ?a ?d)
      (move ?a ?n ?d)
      (go2 ?r))))

```

The entire operational model can be found in Appendix B.1.

5.3.1.1 Results

We evaluate the different configurations of OMPAS on the *Gripper-Door* domain on several problems of different complexity. The overall results are presented in the Table 5.2, the Table 5.3 and the Table 5.4.

For this evaluation, we compared the most basic configuration of OMPAS $C(R)$,

Config	T_E (s)	N_C	$Cov(\%)$	\hat{E}_S (T/s)	$D\hat{E}_S^*$ (%)	T_D (s)	T_{WP} (s)	T_P (s)
$C(R)$	21.7	4.3	100.0	21.0	0.0	0.0	0.0	0.0
$C(R/Sat)$	21.8	4.3	100.0	20.9	0.3	0.1	0.1	16.9
$C(R/Opt)$	21.8	4.3	100.0	20.9	0.3	0.1	0.1	16.7
$C(G)$	21.7	4.3	100.0	21.0	0.0	0.0	0.0	0.0
$C(G/Sat)$	21.8	4.3	100.0	20.9	0.3	0.1	0.1	17.0
$C(G/Opt)$	21.8	4.3	100.0	20.9	0.3	0.1	0.1	16.9
$C(C)$	21.7	4.3	100.0	21.0	0.0	0.0	0.0	0.0
$C(C/Sat)$	21.8	4.3	100.0	20.9	0.3	0.1	0.1	17.1
$C(C/Opt)$	21.7	4.3	100.0	20.9	0.3	0.1	0.1	16.9
$C(U)$	21.7	4.3	100.0	20.9	0.3	0.1	0.0	0.0
$C(U/Sat)$	21.8	4.3	100.0	20.9	0.3	0.1	0.1	16.9
$C(U/Opt)$	21.8	4.3	100.0	20.9	0.3	0.1	0.1	16.8
$C(A)$	21.8	4.3	100.0	20.9	0.5	0.1	0.0	0.0
$C(A/Sat)$	21.8	4.3	100.0	20.9	0.3	0.1	0.1	16.8
$C(A/Opt)$	21.8	4.3	100.0	20.9	0.3	0.1	0.1	16.8

Table 5.2: Results on the *Gripper-Door* domain averaged on three *easy* problems.

Config	T_E (s)	N_C	$Cov(\%)$	\hat{E}_S (T/s)	$D\hat{E}_S^*$ (%)	T_D (s)	T_{WP} (s)	T_P (s)
$C(R)$	81.3	16.2	100.0	5.8	48.6	0.1	0.0	0.0
$C(R/Sat)$	56.0	10.5	100.0	8.3	24.9	4.2	4.1	51.2
$C(R/Opt)$	54.1	10.1	100.0	8.6	23.1	3.5	3.4	49.4
$C(G)$	80.8	16.1	100.0	5.8	48.6	0.1	0.0	0.0
$C(G/Sat)$	50.4	9.4	100.0	9.2	17.7	2.9	2.8	45.7
$C(G/Opt)$	51.6	9.8	100.0	9.0	19.4	3.2	3.1	47.0
$C(C)$	43.7	8.7	100.0	10.6	5.2	0.1	0.0	0.0
$C(C/Sat)$	48.8	9.4	100.0	9.3	15.2	2.4	2.3	44.1
$C(C/Opt)$	47.4	9.1	100.0	9.7	12.7	2.3	2.3	42.7
$C(U)$	42.2	8.3	100.0	11.2	0.9	0.7	0.0	0.0
$C(U/Sat)$	46.9	9.0	100.0	10.0	11.2	2.8	2.5	42.1
$C(U/Opt)$	45.5	8.7	100.0	10.2	8.5	2.4	2.2	40.8
$C(A)$	47.6	9.4	100.0	9.6	12.6	0.8	0.0	0.0
$C(A/Sat)$	48.3	9.5	100.0	9.5	14.0	1.5	1.2	43.5
$C(A/Opt)$	47.1	9.3	100.0	9.8	12.0	1.4	1.2	42.5

Table 5.3: Results on the *Gripper-Door* domain averaged on three *medium* problems.

Config	T_E (s)	N_C	Cov (%)	\hat{E}_S (T/s)	$D\hat{E}_S^*$ (%)	T_D (s)	T_{WP} (s)	T_P (s)
$C(R)$	418.4	83.5	71.7	0.8	81.5	2.7	0.0	0.0
$C(R/Sat)$	324.7	50.0	81.7	1.2	71.7	78.2	76.3	319.2
$C(R/Opt)$	313.0	49.2	83.3	1.2	70.4	76.5	74.7	308.7
$C(G)$	403.0	80.4	75.0	0.8	79.7	2.6	0.0	0.0
$C(G/Sat)$	321.5	50.9	68.8	1.1	74.4	82.1	80.2	317.2
$C(G/Opt)$	332.6	51.8	81.7	1.1	73.4	82.1	80.2	328.1
$C(C)$	113.5	22.6	100.0	4.0	3.3	0.7	0.0	0.0
$C(C/Sat)$	137.2	23.0	100.0	3.3	20.2	25.3	24.5	132.4
$C(C/Opt)$	135.4	22.6	100.0	3.4	18.8	25.1	24.3	130.8
$C(U)$	127.1	22.7	100.0	3.6	14.2	16.4	0.0	0.0
$C(U/Sat)$	149.9	23.7	100.0	3.1	26.6	37.5	24.3	145.0
$C(U/Opt)$	158.8	22.9	98.3	2.8	32.1	37.9	24.7	152.5
$C(A)$	112.9	22.2	100.0	4.0	3.4	5.7	0.0	0.0
$C(A/Sat)$	114.4	22.4	100.0	4.0	4.6	8.9	4.0	109.3
$C(A/Opt)$	109.3	21.4	100.0	4.2	0.3	8.9	4.0	104.1

Table 5.4: Results on the *Gripper-Door* domain averaged on three *hard* problems.

which is comparable to vanilla RAE, with configurations that benefit from more advanced heuristics.

We compare these configurations on different problems at three arbitrary levels of complexity. In each problem, a unique robot is used to move balls to target rooms. The complexity of each problem varies with the parameters presented in the following table:

complexity	Tasks	Room	Max-distance
Easy	1	2	1
Medium	2	4	2
Difficult	4	8	3

For each level of complexity, the number of tasks to work on in parallel is different. Here it ranges from one for a *easy* problem to four for a *difficult* problem. The number of rooms defines the complexity of the navigation. To limit the displacement, we add a *max-distance* parameter that defines the maximum distance between two rooms, where the distance between two adjacent rooms is one. For a given door, its initial state is evenly distributed between open and closed.

For all available commands, we simulate them with an ad hoc duration of five seconds, which assumes that the robot is quite fast in performing actions. As for the parameters *deliberation reactivity* and *planner reactivity* of OMPAS, they are set to one second and two seconds respectively. This means that the system has at most one second to refine a task into a method, and the continuous planner has at most two seconds during which the planner cannot be interrupted. This also assumes that two seconds is a good interval, during which we assume the world does not change much. Again, these are ad hoc parameters, and no further study has been done to identify the best set of parameters.

For the three levels of complexity, we generated three different problems. For each one of the nine problems, each configuration was run ten times. We present the results for each one of the configurations and grouped by level of complexity. The results are averaged over the thirty runs (ten runs on three problems) for each level of complexity.

From the data given in the Table 5.2, we can see that on *easy* domains, all the configurations are mostly equivalent, with a maximum $D\hat{E}_S^*$ of 0.5%. On these problems, the advantage is given to the configurations that have a low deliberation time T_D . As the complexity of the problem increases, we can see that configurations that benefit from lookahead capabilities have a clear advantage over the other configurations.

For problems of *medium* complexity, the information of Table 5.3 shows that $C(U)$ has the advantage on other configurations. This configuration takes advantage of the anytime planner $UPOM$ to guide the refinement of tasks. Unlike other planner-based configurations, $UPOM$ can provide a solution at any time. This should explain its advantage over the configurations based on *Aries*, since there is no guarantee that the planner can find a solution in the limited time.

Finally, for the *difficult* problems, the best configuration is $C(A/Opt)$. This configuration has a slight advantage over other configurations. On this configuration, it is interesting to note that the *continuous planning* does not significantly affect the running time, as we have an average T_{WP} of four seconds.

Overall, we see that as the complexity of the problem increases, the configurations based on *Aries* and *continuous planning* improve the quality of the solutions, and the overhead cost of using such heuristics is amortized with respect to the total execution time T_E .

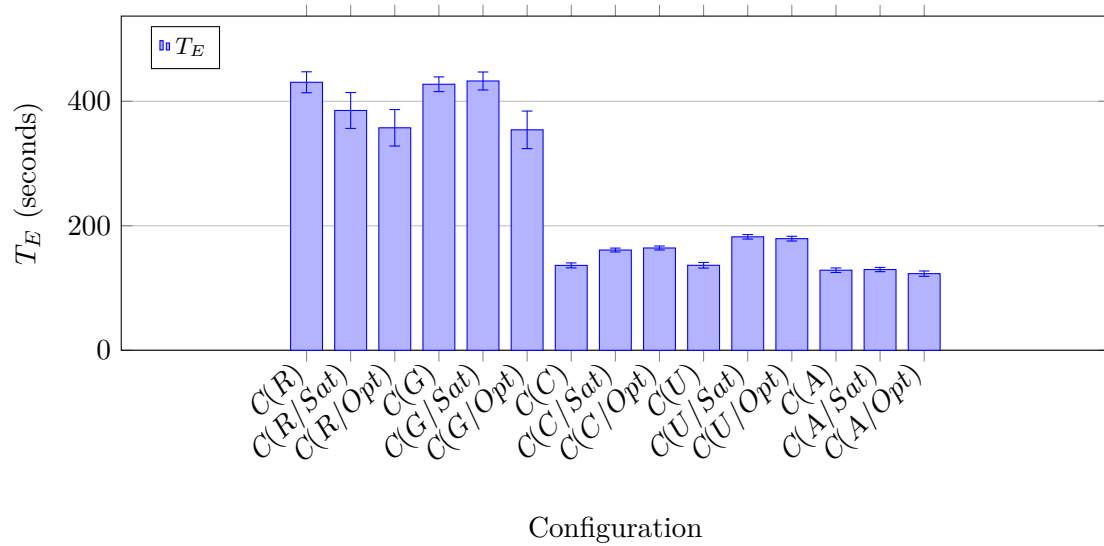
5.3.2 Case study

Now that we have presented global results on the performance of different configurations of OMPAS on the *Gripper-Door* domain, we will present more detailed results on a specific instance. We take an instance of *hard* complexity. In Figure 5.1 we show the execution time T_E with the different configurations of OMPAS. In Figure 5.1b we show the *Deliberation Time Ratio* (RT_D) for each of the configurations. The RT_D is defined as

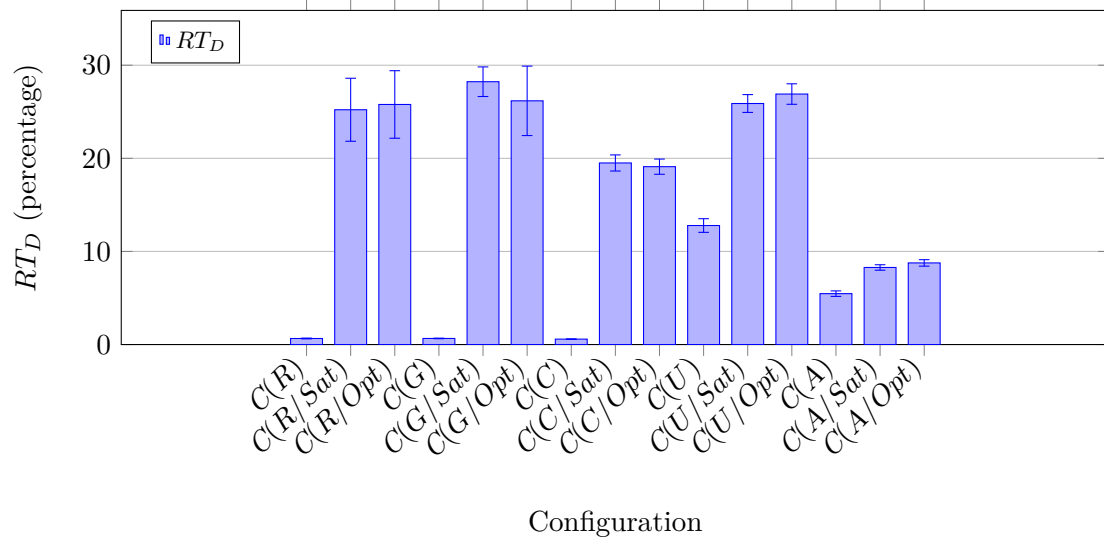
$$RT_D = \frac{T_D}{T_E} \times 100$$

We can see that the configurations using either the *Cost*, the $UPOM$ or the *Aries* strategies are almost three times better than the ones using the *Random* and the *Greedy* ones. Moreover, these strategies have a more consistent T_E as shown by the smaller standard error.

Moreover, we can see that *continuous planning* improves the performance of the less performant configurations, but increases the execution time for the ones that were already performing well. This could be explained by the effect of planning on execution time. In fact, as we can see in Figure 5.1b, deliberation represents more than 20% of the total execution time. It is interesting to note that *continuous planning* has a



(a)



(b)

Figure 5.1: Mean of the total execution time T_E in Figure 5.1a of several configurations of OMPAS and the ratio of the deliberation time over the execution time RT_D in Figure 5.1b. The standard error for the two metrics is also represented.

smaller impact on execution time when relying on *Aries* to refine tasks locally. This can be explained by the fact that *continuous planning* can benefit from the local plan, and it can speed up the search for the instance of *Aries* running *continuously* in the background.

Differences on the choices The differences in performance can be explained by the choices made by the *acting engine* at runtime. The configurations that use planning are informed by algorithms capable of looking ahead. The main improvement brought by the use of *Aries* is the limitation of the number of displacements of *Robby*. In fact, the planner naturally finds a solution that limits the recursions in the *go2(?room)* task. Moreover, the *continuous planner* is able to manage the access to the resources of the *gripper* globally, so that *Robby* might be able to move with both *gripper* transporting balls. This is also possible with other configurations, but it may happen arbitrarily. The planner makes sure that the choices made by OMPAS are on average "good" with respect to the overall performance of the system. Therefore, this is the combination of all the "good" choices made by the *acting engine* that results in a better *Efficiency Score*, even at a greater deliberation cost.

5.4 Gobot-Sim: integration with a factory simulator

As mentioned before, the extension of the operational language to represent hierarchical concurrent reasoning can be justified by logistics scenarios involving robotic platforms. In such a scenario, one of the sub-problems addressed is the processing of different objects at different locations, which is close to a Job Shop Scheduling Problem (JSSP) (Applegate and Cook 1991). The solution of such a problem is generally found thanks to scheduling. In robotic domains, we can generalize this problem to a temporal planning problem, where deliberation is used to generate the sequence of actions to fulfill a mission, while ordering them to optimize the JSSP subproblem and taking into account deadlines. We assume that the time to process an object is several orders of magnitude larger than the time between processes, which includes the time to transport an object between two locations, the time to move robots, and other durations that should be considered during execution but are negligible with respect to the overall process. To show the relevance of the presented approach, we present here the successful integration of OMPAS with *Gobot-Sim*, a new job shop benchmark for acting systems.

5.4.1 Gobot-Sim

Gobot-Sim is a factory simulator in which a fleet of robots should move packages between machines. The machines are used to process packages, each package having one or more processes applied to it. The goal is to process packages as fast as possible. *Gobot-Sim* has been implemented as a 2D video game that abstracts from the complexity of physics and collision between elements to focus on deliberation. This makes it simple and lightweight.

Gobot-Sim is implemented in the Godot engine. Godot Engine is an open source video game engine. Godot Engine can run headless, which means that no image is generated. This is especially useful if you want to run benchmarks on *Gobot-Sim*. *Gobot-Sim* uses version 3.5 of the game engine.

Gobot-Sim is simpler than the RoboCup Logistics League Simulation (Niemueller, Karpas, et al. 2016) because it is a 2D environment in which the manipulation of packages is simplified: the pick and place actions are instantaneous. *Gobot-sim* is similar to Craftbots (Nemiro et al. 2021) in that navigation is simplified. However, *Gobot-Sim* focuses on logistical problems similar to JSSP.

The simulator can receive keyboard and mouse input like a video game, but it can also receive commands from an external system via a TCP connection. The TCP connection is also used to send state updates to the external system. The format and information of the message exchange are described in detail in the documentation of the simulator. An implementation with OMPAS has been done using the TCP connection exposed by *Gobot-Sim*.

The simulator was developed in the same team as the author of this thesis. Some contributions have been made to improve the simulator and add specific features for the purpose of implementing it with OMPAS.

5.4.1.1 Environment of *Gobot-Sim*

As mentioned before, the *Gobot-Sim* environment is a factory that processes packages on different machines. A screenshot of the simulator is available in Figure 5.2. It shows a specific scenario in which two robots should move six packages between six machines, while monitoring their battery level. In *Gobot-Sim*, the factory is delimited by walls that are impenetrable. Inside those walls, we have machines, packages, robots, belts and recharge areas that we are going to detail.

Machine The factory consists of machines. Each machine can perform one or more processes, and each process is identified by an ID. A machine can process one package at a time. Graphically, processes are uniquely identified by a color. Processing machines receive packages on their input belt. The length of the input belt defines the entry buffer of the machine. The output belt of the machine is used to store packages pending a robot takes them to the next processing machine.

There are also specific machines in the environment. The input machines from which the packages arrive in the environment, and the output machines that take the fully processed packages and send them out of the factory.

Package A package is an object to which one or more processes are to be applied. Each process has a specific processing time. Two packages that should receive the same

¹The documentation on the communication format and information of the *Gobot-Sim* simulator is available at the following URL: <https://github.com/plaans/gobot-sim/blob/master/doc/communication.md>

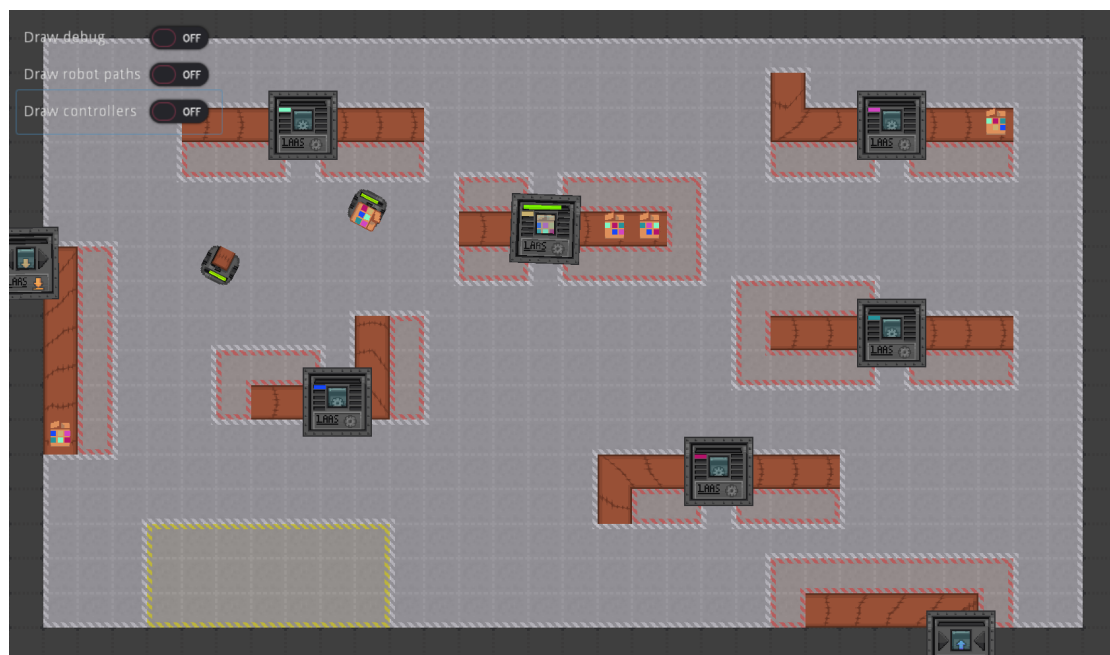


Figure 5.2: Overview of a 6×6 job shop scenario in *Gobot-Sim* composed of one *input* machine (on the left) that feeds the environment with unprocessed packages, six *processing* machines that can do a predefined process, and one *output* machine (at the bottom right) that receives fully processed packages. Two robots can be used to dispatch packages on the machines. The recharge area (in yellow) is available at the bottom.

process may have different processing times. The process to be applied to a package is ordered. Packages are moved from one machine to another by robots.

Robot A robot can move around the factory in a holonomic way, meaning that there are no restrictions on the direction in which it can move. It has a battery that discharges continuously. However, the rate of discharge is different when the robot is performing an action than when it is idle. Its battery can be recharged on recharge areas. The recharge rate can vary from robot to robot. Robots are the only manipulators in the factory. They can pick up one package at a time. They can carry the package (the additional package has no effect on the energy consumption, i.e. the consumption rate is not changed). Packages can only be dropped on machine belts.

To interact with packages on the belt, the robot should be in one of the interact areas around the belt.

5.4.1.2 Integration with Gobot-Sim

From the perspective of an external system, Gobot-Sim is viewed as a robotic platform. This means that Gobot-Sim sends perception information and receives commands to be executed on the platform.

State representation The perceived information is aggregated into a state representation of the system. The state consists of state variables. Each state variable represents a part of the state of the system. State variables can represent either static facts or dynamic facts. The state is composed of information about the robots, the machines, and the topology of the environment, e.g., locations of machines. The state is considered to be fully observable in the current version, but we see no limitation in restricting the observability of the state.

Every object in the environment (except the walls and the floor) is represented by a set of state variables and values in the state.

- Robots are defined by their position, their speed, and their battery level.
- Machines are represented by their progress rate in processing a given package.
- Packages are represented by their location and the remaining processes to be performed.

The complete definition of the state variables can be found in the documentation of the simulator¹.

5.4.1.3 Commands

Commands can be sent to Gobot-Sim to either robots or machines. The full description of the command arguments and format is given in the documentation¹.

Robot commands The robot can move and manipulate packages using the following commands:

- We can ask the robot to pick up a package with the command `pick(?r: robot)`. The command can only succeed if the robot is facing the belt on which the package is located, and the robot is in one of the interact areas of the belt. If there are several packages on a belt, the robot will pick the closest one. To pick a specific package, we can use `pick_package(?r: robot, ?p: package)`.
- A robot can use the `place(?r: robot)` command, which places the currently carried package on the belt the robot is facing. If the robot is not facing a belt, the command will be denied.
- To face a belt, a robot can rotate by a certain angle with `do_rotation(?r: robot, ?angle: float, ?speed: float)`, which will rotate robot `?r` by `?angle` in rad at `?speed` in rad.s^{-1} . We ask to rotate to an absolute angle with `rotate_to(?r: robot, ?angle: float, ?speed: float)`; the speed parameter is the same. In general, the robot only needs to rotate to face a line. This is exactly the role of the command `face_belt(?r: robot, ?b: belt)`, which will face a belt regardless of the robot's position.
- There are several commands to move the robot. The most basic one is `do_move(?r: robot, ?angle: float, ?speed: float, ?duration: float)` tells the robot `?r` to move in direction `?angle` at `?speed` for `?duration`. For very precise control of the robot's movement, this command might be useful. However, you might want to control the commands at a higher level. Therefore we have the command `navigate_to(?r: robot, ?x: float, ?y: float)`, which gives target coordinates to reach. Similarly, `navigate_to_cell(?r: robot, ?cx: int, ?cy: int)` moves to the cell defined by the two coordinates. At a higher level, we can move to a given area with `navigate_to_area(?r: robot, ?area: area)`.
- To recharge, we can even issue the command `go_charge(?r: robot)`, which will send the robot to the nearest recharge area. So we rely on the robot platform to find the closest area.

Machine command In the version of the simulator that we are using, processes on machines should be started explicitly with a command. The command is `process(?m: machine, ?p: package)`, where `?m` is the machine on which the package `?p` should be processed. Note that a prerequisite for the success of the command is that the package is present on the input belt of the machine, otherwise the command will be rejected.

5.4.2 Versions of Gobot-Sim

With the environment presented, we can now define different scenarios for handling packages. We consider two of them.

Gobot-Sim Jobshop The first is a job shop type problem where all packages are present at the start of the simulation. We call it *Gobot-Sim_{JS}*. In this configuration, each machine can perform only one process, and only one machine can perform a given process. The underlying problem we consider here is $n \times m$ JSSP, where n is a finite number of packages to be processed on a finite set of machines m . However, the model is more complex than a classic JSSP. In fact, a package has to wait at the entrance of a machine before it can be processed. At best, it is processed directly. Moreover, a package must be brought to a machine before it can be processed. Therefore, the travel time of the package should be taken into account.

Gobot-Sim Continuous-shop In the second, we consider a finite set of packages that can arrive in the environment at any time. Therefore, the autonomous agent should consider the new mission resulting from the arrival of the new package. We refer to this second configuration as *Gobot-Sim_{CS}*. In this case, the external agent has no knowledge of the incoming packages, which means that it cannot anticipate their impact on its agenda until they are introduced into the environment.

5.4.3 Integration with OMPAS

OMPAS can control the robotic platform by sending commands over a TCP connection. Updates on the status of the robotic platform and the command status are done at a fixed frequency of 60Hz.

Acting Domain Given the specification of the system, we define two types of events that should be handled by OMPAS at runtime.

- For each package $?p$ in the system, a new event *on_new_package(?p)* is triggered once and executes the task *t_process_package(?p)* in a new thread, whose goal is to process the package. One of its methods is *m_process_package(?p)*, which generates a list of tasks, each task representing the action to process the package $?p$ on a given machine, which includes getting the package, bringing it to the machine, and instructing the machine to process the package. The last task of the method *t_output_package(?p)* brings the package to the output machine.

```
(def-task t_process_package (:params (?p package)))
(def-method m_process_package
  (:task t_process_package)
  (:params (?p package))
  (:body
   (do
    (define tasks
     (mapf
      (lambda (process)
        '(t_process_on_machine ,?p
          (arbitrary ',(
            find_machines_for_process (car
              process))))
```

```

                                ,(cadr process)
                                ))
                                (package.all_processes ?p)))
                                (apply seq tasks)
                                (t_output_package ?p))))
(:params (?p package))
(:trigger (once))
(:body
  (do
    (wait-for '(instance (package.location ,?p)
                        belt))
    (t_process_package ?p))))

```

- For each robot $?r$ in the system, each time the battery level of $?r$ is under a critical level (here 40%), a new event $on_battery_low(?r)$ is triggered to charge the robot.

```

(:params (?r robot))
(:trigger (whenever (< (robot.battery ?r) 0.4)))
(:body (charge_robot ?r))

```

The full domain is given in Appendix B.2 along with an example of problem instance.

In such representation of the problem, we can distinguish two kinds of unary resources: robots and machines. Since the number of robots is smaller than the number of packages to be processed, it is obvious that one robot will contribute to the processing of several packages.

Therefore, the role of the acting engine is to assign robots to packages, and to schedule the passage of packages on the machine. In the current version of the system, the acquisition of a resource is sorted by a priority level, where the task that monitors the battery of a robot has the highest priority.

5.4.4 Benchmark

We benchmarked OMPAS on the two versions of *Gobot-Sim*, namely *Gobot-Sim_{JS}* and *Gobot-Sim_{CS}*. For each configuration, we compared a user-defined reactive allocation strategy against a *random* allocation strategy that benefits from *continuous planning*. The difference between the custom strategy and the random strategy lies in the operational models used in the two configurations. The reactive strategy, called *First Available (FA)*, takes advantage of a new operator of SOMPAS that attempts to acquire a list of resources, and returns the first one that is available. The operator is `(acquire-in-list l q)`, where `l` is a list of resources, and `q` is an optional amount to borrow. On success, it returns the tuple `(label rh)`, where `label` is the name of the borrowed resource and `rh` is its *resource-handle*. This results in the following operational model:

```

(def-task t_process_on_machine (:params (?p package) (?m
  machine) (?d int)))
(def-method m_process_on_machine
  (:task t_process_on_machine)

```

```
(:params (?p package) (?m machine) (?d int))
(:body
  (do
    (define h_m (acquire ?m))
    (define h_r (acquire-in-list (instances robot)))
    )
    (define ?r (first h_r))
    (t_carry_to_machine ?r ?p ?m)
    (release (second h_r))
    (t_process ?m ?p ?d)))
```

The random strategy is defined as follows:

```
(def-task t_process_on_machine (:params (?p package) (?m
  machine) (?d int)))
(def-method m_process_on_machine
  (:task t_process_on_machine)
  (:params (?p package) (?m machine) (?d int))
  (:pre-conditions true)
  (:score 0)
  (:body
    (do
      (define ?r (arbitrary (instances robot)
        rand-element))
      (define h1 (acquire ?m))
      (define h2 (acquire ?r))
      (t_carry_to_machine ?r ?p ?m)
      (release h2)
      (t_process ?m ?p ?d)
```

The only difference between the two models is how the resource is acquired in the body of the `m_process_on_machine` method. We use the second model because in the present work we do not support the translation of `acquire-in-list` into a corresponding planning representation.

Problem configuration The problems for each version of *Gobot-Sim* are generated as follows: In each configuration, we use a fleet of two robots. We have the following parameters for the problems generated for *Gobot-Sim_{JS}* version.

complexity	Packages	Processes	Process time
Easy	2	2	[2;8]
Medium	4	4	[2;15]
Difficult	6	6	[2;15]

For each problem we define the number of packages to process and the number of processes for each package. Then, the *process-time* is a range of values that a given

process will take on a machine. For each *easy* problem, it ranges from 2s to 8s. In the *Gobot-Sim_{JS}* version, each process can only be run on one machine. It should also be noted that all packages are in the system from the beginning of the problem instance.

For the *Gobot-Sim_{CS}* version, we use the following parameters, which are slightly different from the other version. Therefore, the parameters for creating a planning instance are as follows:

complexity	Packages	Processes	Process time	Machine/Process
Easy	2	[1;2]	[1;5]	3
Medium	4	[1;4]	[1;10]	2
Difficult	6	[1;6]	[1;20]	1

For each one of the complexities, we define a fixed number of packages that can have a range of processes to be done. As for the *Gobot-Sim_{JS}* version, the process time is a range of values. We also add some flexibility on the machines by allowing a process to be run on multiple machines. This gives the *acting engine* a choice of which machine to process a given package.

Results In the Table 5.5 we have the results on the *Gobot-Sim_{JS}* and *Gobot-Sim_{CS}* versions comparing the performances of OMPAS using the $C(R)$, the $C(R/Sat)$, and the $C(R/Opt)$ configurations. Here, the $C(R)$ configuration benefits from the custom reactive strategy. We do not benchmark the other configurations of OMPAS because there is no choice of methods to refine a task. For each task, there is a single task to choose. Hierarchical decomposition is only used to break down the model into multiple levels of abstraction.

The metrics used are those described at the beginning of this chapter. On all six problems, the configuration $C(R/Opt)$ outperforms all other strategies. Depending on the version and the complexity of the problem $D\hat{E}_S^*$ ranges between 9.3% and 15.9% for the reactive strategy. What is also interesting is that with a similar number of actions, the $C(R/Opt)$ outperforms the other configuration.

What is also interesting is that the planning time T_P increases drastically with the complexity of the problem, but the impact on the deliberation time stays minimal. In the worst case, the T_{WP} is 23.8% for the configuration $C(R/Opt)$ on the *difficult* problem of *Gobot-Sim_{JS}*, which represents approximately 5% of T_E .

5.4.5 Case Study

As for the *Gripper-Door* domain, we present a case study on a particular instance of the domain. Here we have chosen a problem of *hard* complexity of the *Gobot-Sim_{CS}* domain. We focus on the *continuous planning* process. In figure 5.3 we show for each of the planning instances the average time to compute a new solution. Unlike other systems that wait for the planner’s response to execute, here the planner searches for a solution in a parallel process. We can see that the planning time is non-linear and decreases as the execution reaches the end.

Table 5.5: Results for the *Gobot-Sim_{JS}* and the *Gobot-Sim_{CS}* domains.(a) Results for *Gobot-Sim_{JS}* averaged on three *easy* problems.

Config	T_E (s)	N_C	Cov (%)	\hat{E}_S (T/s)	$D\hat{E}_S^*$ (%)	T_D (s)	T_{WPP} (s)	T_P (s)
$C(R)$	79.2	40.0	98.9	8.3	12.2	0.0	0.0	0.0
$C(R/Sat)$	79.9	40.0	100.0	8.3	12.3	0.2	0.1	1.0
$C(R/Opt)$	70.8	40.0	100.0	8.8	4.3	0.2	0.1	1.2

(b) Results for *Gobot-Sim_{JS}* averaged on three *medium* problems.

Config	T_E (s)	N_C	Cov (%)	\hat{E}_S (T/s)	$D\hat{E}_S^*$ (%)	T_D (s)	T_{WPP} (s)	T_P (s)
$C(R)$	279.2	136.4	99.6	2.2	15.9	0.1	0.0	0.0
$C(R/Sat)$	289.6	136.4	100.0	2.1	18.7	2.4	2.2	24.2
$C(R/Opt)$	235.3	136.4	100.0	2.6	0.0	2.8	2.7	104.3

(c) Results for *Gobot-Sim_{JS}* averaged on three *difficult* problems.

Config	T_E (s)	N_C	Cov (%)	\hat{E}_S (T/s)	$D\hat{E}_S^*$ (%)	T_D (s)	T_{WPP} (s)	T_P (s)
$C(R)$	523.0	288.4	98.9	1.1	6.0	0.3	0.0	0.0
$C(R/Sat)$	551.6	277.9	89.4	1.0	19.3	26.2	25.8	242.2
$C(R/Opt)$	494.0	288.7	99.4	1.2	0.0	24.2	23.8	368.7

(d) Results for *Gobot-Sim_{CS}* averaged on three *easy* problems.

Config	T_E (s)	N_C	Cov (%)	\hat{E}_S (T/s)	$D\hat{E}_S^*$ (%)	T_D (s)	T_{WPP} (s)	T_P (s)
$C(R)$	53.9	26.0	98.9	11.3	13.5	0.0	0.0	0.0
$C(R/Sat)$	54.8	26.0	100.0	11.1	14.1	0.1	0.1	0.5
$C(R/Opt)$	48.2	26.0	100.0	12.7	2.9	0.1	0.1	0.5

(e) Results for *Gobot-Sim_{CS}* averaged on three *medium* problems.

Config	T_E (s)	N_C	Cov (%)	\hat{E}_S (T/s)	$D\hat{E}_S^*$ (%)	T_D (s)	T_{WPP} (s)	T_P (s)
$C(R)$	199.0	91.8	99.4	3.0	9.3	0.1	0.0	0.0
$C(R/Sat)$	203.1	91.9	99.5	3.0	10.3	0.8	0.7	9.5
$C(R/Opt)$	180.1	91.9	99.0	3.3	0.3	0.9	0.8	29.6

(f) Results for *Gobot-Sim_{CS}* averaged on three *difficult* problems.

Config	T_E (s)	N_C	Cov (%)	\hat{E}_S (T/s)	$D\hat{E}_S^*$ (%)	T_D (s)	T_{WPP} (s)	T_P (s)
$C(R)$	399.2	164.8	99.7	1.5	14.3	0.2	0.0	0.0
$C(R/Sat)$	384.9	164.9	99.7	1.6	11.7	3.0	2.8	49.8
$C(R/Opt)$	339.8	164.8	99.7	1.8	0.0	4.2	4.0	129.2

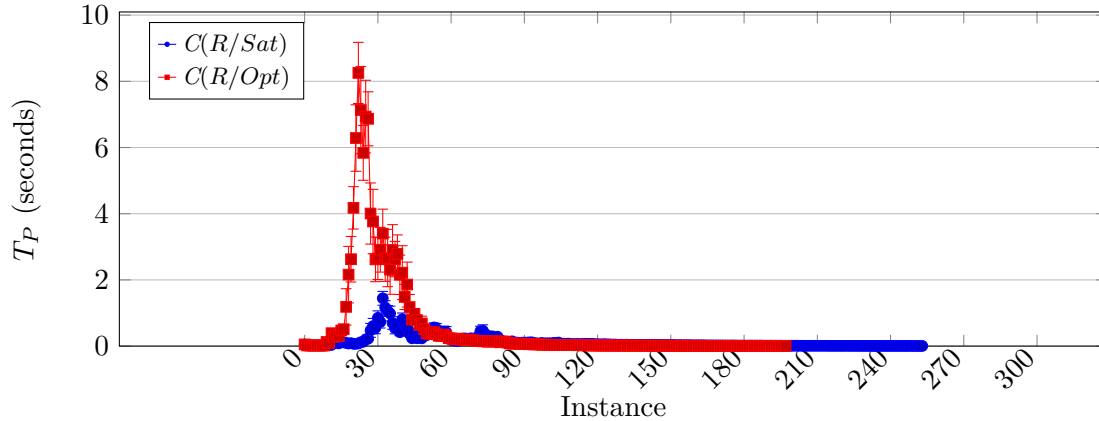


Figure 5.3: Continuous planning time for successive instances in *Satisfactory* and *Optimality* configurations

It is interesting to note that in the case of the *Gobot-Sim_{CS}* domain, the planning time increases during the first instance. This is explained by the arrival of new packages in the system, which increases the complexity of the planning problem with the number of packages. However, we still see that the required planning time decreases as the execution continues, with a minimum time in the last instances. We can also see that there is a large difference in planning time between the *Optimality* and *Satisfactory* configurations. This should explain why the performance of the *Optimality* version is worse on problem instances of *easy* and *medium* complexity.

5.5 Discussion

With the benchmark presented above, we wanted to evaluate the ability of OMPAS to handle multiple tasks and optimize a quality metric. We chose to optimize the total time, which is interesting because it introduces a trade-off between execution time and deliberation time. To do this, we set up two benchmarks, each targeting a specific feature of OMPAS. On the one hand, the *Gripper-Door* domain requires that the acting system makes informed decisions about task refinement, and task interleaving is of secondary importance to the efficiency of the system. On the other hand, the *Gobot-Sim* domain provides a context that requires scheduling capabilities from the acting engine. Moreover, the *Gobot-Sim* domain shows how OMPAS is able to adapt its execution to take into account tasks and events that may occur at any time. It is interesting to note that the improvement in performance brought by the planning techniques presented in this thesis is most noticeable on problems of the highest complexity. In the near future, we plan to benchmark OMPAS on extensions of the *Gripper-Door* domain in which:

- we use a fleet of robots instead of a single robot,
- the tasks to be performed are more complex than placing a ball in a room.

In the last extension, we propose to target "complex" tasks, such as building a toy from

parts scattered across all rooms. With this domain, we can introduce parallelism, and synchronization is required to first get all the parts and then build the toy. We expect that this domain will better highlight the approach presented in this dissertation.

Limitations of the evaluated domains While the domains showed that the most efficient approaches are those that use planning to guide the acting engine despite the increased deliberation cost, the evaluation is still limited.

First, the only dynamic feature considered is the response to new tasks, which was addressed in *Gobot-Sim_{CS}*. However, the ability of OMPAS to face failures is not emphasized in the evaluated domains. The *retry* mechanism of OMPAS could be tested by introducing non-determinism in the result of command execution. We could take inspiration from the previous evaluation of RAE systems in previous papers (Patra, Ghallab, et al. 2019; Patra, Mason, Kumar, et al. 2020; Patra, Mason, Ghallab, et al. 2021). For example, we could easily extend the *Gripper-Door* domain by relying on another simulator that supports stochastic models for commands. In this sense, OMPAS already provides facilities to define different models for planning and simulation. In Chapter 4 we introduced the `def-command-pddl-model` operator, which can be used to define a PDDL-like model that a planner can take advantage of. In addition to this *operator*, we have the `def-command-simulator-model` operator, which takes as input a model in the form of a SOMPAS program that only the internal simulator can access. The sole purpose of this *simulator* model would be to emulate the behavior of the *robotic platform*. This way we could have an evaluation close to the one proposed for the previous versions of RAE (Patra, Mason, Ghallab, et al. 2021), and this would make the comparison easier.

Second, we decided to define our own robotics domain based on a simple scenario where a robot should move objects. One could argue that we could have used the domains already provided in (Patra, Mason, Ghallab, et al. 2021). However, those domains were defined in Python. Therefore, to have the same domains, it would be necessary to translate those domains into SOMPAS at the cost of having slightly different models. In fact, since the operational models were defined in Python, they could use the full extent of the language, which they used to define the locking mechanism between tasks. In the near future, we plan to translate these domains from Python to SOMPAS and provide a fair comparison with the previous implementations of RAE. In particular, we could better compare the performance of our plan-based approach, which relies on the temporal and hierarchical planner *Aries*, with the nondeterministic planner *UPOM*.

On the integration with other simulators During the development of OMPAS the simulator *CraftBots* (Nemiro et al. 2021) was presented. In this 2D simulator, a fleet of robots should mine resources to craft objects in a limited time. Each task gives a reward, and the goal is to maximize the total reward in a limited time. This simulator introduces a different reward for each task, which adds an interesting problem of optimizing a metric different than the number of tasks or the makespan. In addition, there are constraints on the resources that can be mined: each resource has a different

weight, so a robot cannot move the same amount of each resource, and resources are only accessible during certain time windows. This adds interesting resource and time constraints to the domain.

However, few acting systems have been integrated with *CraftBots*. In fact, one of the few systems that have been integrated with *CraftBots* is the one that is part of the demonstration of the simulator. For this reason, we plan to extend *GobotSim* to integrate the interesting features of *CraftBots*:

- Processing a package gives a reward,
- The machines are only available during certain time windows.

With such an extension, in addition to efficiently processing packages, the goal would be to maximize the total reward while coping with contingencies.

Lack of comparisons with other systems Returning to the comparison with other RAE systems, OMPAS uses a dedicated implementation of *UPOM* that is specific to OMPAS. We compare *UPOM* with the hierarchical and temporal planner *Aries* in a setup that clearly favors *Aries*:

- The models used in OMPAS are deterministic, which allows the automatic translation into chronicles to be used by *Aries*, and the *UPOM* has no advantage in this setup.
- *UPOM* is not accompanied by the learned heuristics proposed in the previous works (Patra, Mason, Ghallab, et al. 2021), which showed to significantly improve the performances of *UPOM*.

To fully compare our approach with *UPOM*, the complete framework should be reimplemented in OMPAS and benchmarked on nondeterministic domains.

Finally, it would have been interesting to compare OMPAS with other acting frameworks such as CLIPS (T. Hofmann et al. 2021), which also targets metric optimization and resource management. To compare both approaches, we could integrate OMPAS into the Robocup Logistics League simulator (Niemueller, Karpas, et al. 2016). Even if the Robocup Logistics League simulator is close to *GobotSim* in terms of the task to be performed, we would be on common ground with CLIPS and would provide a fair comparison. In addition, OMPAS would gain an integration with ROS2 (Macenski et al. 2022), which would make it easier to share OMPAS with the rest of the community.

5.6 Conclusion

In conclusion, the provided evaluation proved the pertinence of the approach developed in this thesis, i.e. using hierarchical and temporal planning to guide the deliberation of a refinement based acting engine that aims to control a fleet of agents. In the near future, we plan to integrate OMPAS into other robotic domains in order to better compare it with other approaches, and to evaluate the other reactive features of OMPAS, such as the handling of failures. We are confident that OMPAS is well equipped to perform on other robotic domains.

Conclusion

In this thesis, we presented a new deliberation system that aims to achieve multiple goals using a fleet of robots. In particular, we presented a unified planning and acting approach based on a refinement acting engine that executes high-level tasks by refining them into executable commands. This resulted in the Operational Model Planning and Acting System (OMPAS) system, a new acting system that extends the capabilities of the Refinement Acting Engine (RAE) to better face the concurrent execution of tasks that require shared resources.

In particular, OMPAS supports concurrency in procedures and takes advantage of a dedicated resource management system to handle the interleaving of executed tasks. Similar to RAE, the deliberation of OMPAS can be guided by an automated planner. In addition to guiding the refinement of tasks into methods, the planner can be used to instantiate arbitrary parameters in the body of methods, but also to organize the order of access to shared resources among concurrent procedures. Thus, OMPAS extends RAE, RAP, PRS, and PROPEL to provide an acting system in which deliberation decisions are explicit and can be influenced by arbitrary heuristics to improve the performance of the system.

To interface with OMPAS, we have defined a new acting language. Based on a simplified version of the Lisp dialect Scheme, SOMPAS provides the facilities to define procedures that can take advantage of the deliberation features of OMPAS. In addition to the operators already defined in the core of the Scheme dialect, the language embeds special modules with control and acting primitives, in which the nondeterministic choices are explicit.

The language is also used as an interface between OMPAS and the user. Through files or a REPL, the programmer can configure the hierarchical operating model that OMPAS will use, but also interact with the system at runtime, such as starting the deliberation system, sending tasks or commands to be executed, or getting reports on the internal state of the system.

The definition of a dedicated acting language has facilitated the integration of a hierarchical and temporal planner to guide the deliberation of OMPAS. Since the language has a restricted core and a clearly identified semantic for the nondeterministic choices, it opens the way to an automated analysis of the programs defined with SOMPAS. In particular, we automatically synthesize the corresponding planning models of the methods defined in the hierarchical operational model of SOMPAS. The planning models take the form of chronicles, a rich planning formalism that allows for expressive temporal models. Chronicles are particularly suitable in our case, since we want to plan the nesting of procedures while optimizing the overall process.

We use these planning models to guide the deliberation of OMPAS at runtime. The planner continuously searches for plans that anticipate the course of action given the choices already made by OMPAS.

With such a generic approach, we were able to provide the interface to simulated

domains, in particular the *Gripper-Door* domain which has been presented in Chapter 1. We also interfaced OMPAS with Gobot-Sim, a factory simulator. In contrast to *Gripper-Door*, the complexity does not lie in the refinement of the tasks, but in the scheduling of the concurrently executed procedures on the system. We demonstrate this with these two experiments:

- OMPAS is well equipped to face multiple tasks that require the use of limited resources,
- OMPAS is able to improve its deliberation by using continuous planning to anticipate the course of action of the system with minimal downtime for the system.

Most importantly, the planning is transparent to the robot programmer and does not require an additional model. This makes OMPAS one of the most advanced proposals for blended planning and execution.

Limitations and future work The disadvantage of such an approach is that, at the moment, continuous planning is not scalable with respect to the number of objectives that OMPAS should face simultaneously. In order of complexity and work required, here are some of the possible improvements and avenues that could be explored to improve the planner’s performance:

- At this point, the planner restarts its search without any information about the previously found solutions. Therefore, we could improve the interface of the planner by providing the previous plan it found. If the planner could benefit from this information, we believe that the search time would be significantly reduced, as the planner would certainly converge to a valid solution faster.
- Another thing would be to relax the underlying temporal problem of the planning problem and solve only the classical part of the planning problem. This would certainly provide useful heuristics for refining the task and instantiating arbitrary variables. However, the ordering of resource access would not be anticipated.
- To reduce the complexity of solving a global planning problem, we could take inspiration from the work on Hierarchical Conformance Planning (HCP) (Kamperis, Y. Wang, and Castellani 2023), which proposes to automatically define several levels of abstraction, each of which is conformant with respect to the lower levels. The proposed approach also targets continuous planning and should provide guidance in a much faster way. However, where we gain search time, we lose globally scoped planning, which would take into account the interactions between the elements of the lower-level stack, here abstracted by the different levels. Therefore, we can expect a loss in quality of the plan, which could be quickly regained by the faster response of the planner.

By defining our system entirely, we also choose to rely on a dedicated execution language, with a homemade interpreter. No particular effort was made to optimize the evaluation of the programs, since the performance of the interpreter of SOMPAS was sufficient for the work presented in this thesis. However, for future iterations of the

system that require a fast evaluation of the programs, it would be necessary to compare its capacities, in particular with respect to interpreters of other Lisp dialects on the part of the language that they share.

Avenues to explore Apart from potential improvements to the performance of the proposed system, it could be extended in other directions. In particular, learning techniques at different tree levels could be integrated into OMPAS. Based on the traces provided by the *acting tree* of OMPAS, the system could learn the following heuristics:

- Having the trace of the execution of the commands, we could extract or improve the model of the commands. In fact, a limitation of our approach is that the model of the commands should still be provided by the user. One could argue that the planning system still needs to be provided with such a model, but having knowledge of the state in which the command started its execution and the state after the execution of the command, we can imagine that we could extract a model of a command composed of the preconditions of the command, its effects, and most importantly, its duration. We could rely on what exists in the literature such as the EXPO (Gil 1994) system, the OBSERVER (X. Wang 1995) system, or more recently the SAM algorithm (Juba, Le, and Stern 2021).
- With OMPAS we have proposed a system in which we can configure the heuristic that the system will use to guide part of its reasoning. For now, we can only define the function used to select a method to refine a task, but we could easily imagine extending this to the resource allocation strategy and so on. Even more interesting would be to provide OMPAS with meta-reasoning capabilities that would allow it to automatically choose its heuristic for a given domain. This would typically be feasible with reinforcement learning, since OMPAS is already capable of simulating the execution of operational models.

In addition to extending the autonomous capabilities of OMPAS, it would be interesting to adapt the system to generic frameworks such as ROS2. Currently, the OMPAS implementation has its own protocol for connecting to a platform using the gRPC framework. However, the majority of the robotics community uses ROS2 to bind robotics software together, and it would be a mistake to turn away from them.

Last but not least, OMPAS only exists because the author of this thesis decided to build a completely new system from scratch. This required a non-negligible amount of engineering to first implement a new interpreter for the concurrent Scheme dialect, and an acting system to benefit from it. On top of that, it is designed from the ground up to unify *Planning* and *Acting* in a single system. Therefore, it could be the perfect candidate to continue the line of work extended in this thesis, ergo to improve the interaction between *Planning* and *Acting* to increase the autonomous capabilities of robotic agents.

Bibliography

- Alami, Rachid, Raja Chatila, Sara Fleury, Malik Ghallab, and Félix Ingrand (1998). “an Architecture for Autonomy”. In: *The International Journal of Robotics Research* 17.4, pp. 315–337 (Cited on pages 10, 14).
- Albore, Alexandre, David Doose, Christophe Grand, Jérémie Guiochet, Charles Lesire, and Augustin Manecy (2023). “Skill-Based Design of Dependable Robotic Architectures”. In: *Robot. Auton. Syst.* 160 (Cited on page 20).
- Allen, James (1983). “Maintaining Knowledge About Temporal intervals”. In: *Commun. ACM* 26.11, pp. 832–843 (Cited on page 99).
- Ambros-Ingerson, José and Sam Steel (1988). “Integrating Planning, Execution and Monitoring”. In: *AAAI Conference* (Cited on page 16).
- Applegate, David and William Cook (1991). “A Computational Study of The Job-Shop Scheduling Problem”. In: *ORSA Journal on computing* 3.2, pp. 149–156 (Cited on page 155).
- Bansod, Yash, Dana Nau, Sunandita Patra, and Mak Roberts (2021). “Integrating Planning and Acting With a Re-Entrant HTN Planner”. In: *ICAPS Workshop on Hierarchical Planning (HPlan)* (Cited on pages 22, 54, 102).
- Bansod, Yash, Sunandita Patra, Dana Nau, and Mark Roberts (2022). “HTN Replanning from The Middle”. In: *International Florida Artificial Intelligence Research Society Conference (FLAIRS)*. Vol. 35 (Cited on pages 22, 102).
- Barbier, Magali, Jean-François Gabard, Dominique Vizcaino, and Olivier Bonnet-Torrès (2006). “ProCoSA: a Software Package for Autonomous System Supervision”. In: *National Workshop on Control Architectures of Robots*, pp. 37–47 (Cited on page 63).
- Barreiro, Javier, Matthew Boyce, Minh Do, Jeremy Frank, Michael Iatauro, Tatiana Kichkaylo, Paul Morris, James Ong, Emilio Remolina, Tristan Smith, and David Smith (2012). “EUROPA: a Platform for AI Planning, Scheduling, Constraint Programming, and Optimization”. In: *International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS)* (Cited on pages 13, 99).
- Beetz, Michael and Drew McDermott (1994). “Improving Robot Plans During Their Execution”. In: *Artificial Intelligence Planning and Scheduling (AIPS)* (Cited on pages 15, 67).
- Beetz, Michael, Lorenz Mösenlechner, and Moritz Tenorth (2010). “CRAM: A Cognitive Robot Abstract Machine for Everyday Manipulation in Human Environments”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, pp. 1012–1017 (Cited on pages 64, 67, 85).
- Berthomieu, Bernard, Silvano dal Zilio, and François Vernadat (2020). *A Fiacre V3.0 Primer* (Cited on page 11).
- Bit-Monnot, Arthur (2018). “A Constraint-Based Encoding for Domain-independent Temporal Planning”. In: *Principles and Practice of Constraint Programming*. Springer, pp. 30–46 (Cited on pages 5, 99, 101, 103).

- Bit-Monnot, Arthur (2023a). “Enhancing Hybrid CP-Sat Search for Disjunctive Scheduling”. In: *European Conference on Artificial Intelligence (ECAI)* (Cited on page 103).
- (2023b). “Experimenting with Lifted Plan-Space Planning as Scheduling: Aries in the 2023 IPC”. In: *2023 International Planning Competition at the 33rd International Conference on Automated Planning and Scheduling* (Cited on page 101).
- Bit-Monnot, Arthur, Malik Ghallab, Félix Ingrand, and David Smith (2020). “FAPE: A Constraint-based Planner for Generative and Hierarchical Temporal Planning”. In: *arXiv preprint arXiv:2010.13121* (Cited on pages 6, 14, 99, 100, 103).
- Bohren, Jonathan and Steve Cousins (2010). “The SMACH High-Level Executive”. In: *IEEE Robotics and Automation Magazine* 17.4, pp. 18–20 (Cited on pages 62, 63).
- Bonasso, Peter, James Firby, Erann Gat, David Kortenkamp, David Miller, and Marc Slack (1996). “Experiences With an Architecture for Intelligent, Reactive Agents”. In: *IJCAI Workshop on Intelligent Agents II Agent Theories, Architectures, and Languages (ATAL)*. Springer, pp. 187–202 (Cited on page 14).
- Bordini, Rafael and Jomi Hübner (2006). “BDI Agent Programming in AgentSpeak Using Jason”. In: *Computational Logic in Multi-Agent Systems: 6th International Workshop*. Springer, pp. 143–164 (Cited on page 66).
- Boussinot, Frédéric and Robert De Simone (1991). “The ESTEREL Language”. In: *Proc. IEEE* 79.9, pp. 1293–1304 (Cited on page 62).
- Brooks, Rodney (1986). “A Robust Layered Control System for a Mobile Robot”. In: *IEEE journal on robotics and automation* 2.1, pp. 14–23 (Cited on page 9).
- Buisan, Guilhem, Anthony Favier, Amandine Mayima, and Rachid Alami (2022). “HAT-P/EHDA: a Robot Task Planner anticipating and Eliciting Human Decisions and Actions”. In: *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2818–2824 (Cited on page 100).
- Carroll, Martin, Kedar Namjoshi, and Itai Segall (2021). “The Resh Programming Language for Multirobot Orchestration”. In: *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4026–4032 (Cited on page 66).
- Cashmore, Michael, Maria Fox, Derek Long, Daniele Magazzeni, Bram Ridder, Arnau Carrera, Narcis Palomeras, Natalia Hurtos, and Marc Carreras (2015). “ROSPlan: Planning in The Robot Operating System”. In: *International Conference on Automated Planning and Scheduling*. Vol. 25, pp. 333–341 (Cited on page 15).
- Castillo, Luis, Juan Fdez-Olivares, Óscar García-Pérez, and Francisco Palao (2006). “Temporal Enhancements of an HTN Planner”. In: *Current Topics in Artificial Intelligence: 11th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2005, Santiago de Compostela, Spain, November 16-18, 2005, Revised Selected Papers 11*. Springer, pp. 429–438 (Cited on page 103).
- Castillo, Luis A, Juan Fernández-Olivares, Óscar García-Pérez, and Francisco Palao (2006). “Efficiently Handling Temporal Knowledge in an HTN Planner.” In: *ICAPS*. Citeseer, pp. 63–72 (Cited on page 100).
- Cavrel, Nicolas, Damien Pellier, and Humbert Fiorino (2023). “On Guiding Search in HTN Temporal Planning with Non Temporal Heuristics”. In: *ICAPS Workshop on Hierarchical Planning (HPlan)*, pp. 28–34 (Cited on pages 101, 103).

- Chatilla, Raja, Rachid Alami, Bernard Degallaix, and Hervé Laruelle (1992). “Integrated Planning and Execution Control of Autonomous Robot Actions”. In: *IEEE International Conference on Robotics and Automation (ICRA)* (Cited on page 21).
- Chien, Steve, Russell Knight, Andre Stechert, Rob Sherwood, and Gregg Rabideau (2000). “Using Iterative Repair to Improve The Responsiveness of Planning and Scheduling”. In: (Cited on page 17).
- Chien, Steve, Gregg Rabideau, Russell Knight, Rob Sherwood, Barbara Engelhardt, Darren Mutz, Tara Estlin, Brandon Smith, forest Fisher, T. Barrett, G. Stebbins, and D. Tran (2000). “ASPEN-Automated Planning and Scheduling for Space Mission Operation”. In: *Space Ops* (Cited on page 99).
- Claßen, Jens, Gabriele Röger, Gerhard Lakemeyer, and Bernhard Nebel (2012). “Platas - Integrating Planning and The Action Language Golog”. In: *KI-Künstliche Intelligenz* 26.1, pp. 61–67 (Cited on page 20).
- Colledanchise, Michele, Diogo Almeida, and Petter Ögren (2019). “Towards Blended Reactive Planning and Acting Using Behavior Trees”. In: *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 8839–8845 (Cited on page 19).
- Conrad, Patrick, Julie Shah, and Brian Williams (2009). “Flexible Execution of Plans With Choice”. In: *International Conference on Automated Planning and Scheduling (ICAPS)*. Vol. 19, pp. 74–81 (Cited on page 18).
- Coste-Maniere, Eve and Nicolas Turro (Sept. 1997). “The MAESTRO Language and its Environment: Specification, Validation and Control of Robotic Missions”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 836–841 vol.2 (Cited on page 63).
- Coste-Manière, Eve, Bernard Espiau, and Eric Rutten (1992). “A Task-Level Robot Programming Language and Its Reactive Execution”. In: *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE Computer Society, pp. 2751–2752 (Cited on page 63).
- Dal Zilio, Silvano, Pierre-Emmanuel Hladik, Félix Ingrand, and Anthony Mallet (2023). “A Formal Toolchain for Offline and Run-Time Verification of Robotic Systems”. In: *Robot. Auton. Syst.* 159, p. 104301 (Cited on page 11).
- Dechter, Rina, Itay Meiri, and Judea Pearl (1991). “Temporal Constraint Networks”. In: *Artificial intelligence* 49.1-3, pp. 61–95 (Cited on page 99).
- Desai, Ankush, Vivek Gupta, Ethan Jackson, Shaz Qadeer, Sriram Rajamani, and Damien Zufferey (2013). “P: Safe Asynchronous Event-Driven Programming”. In: *ACM SIGPLAN Notices* 48.6, pp. 321–332 (Cited on page 63).
- Despouys, Olivier and Félix Ingrand (2000). “Propice-Plan: Toward A Unified Framework for Planning and Execution”. In: *Recent Advances in AI Planning*. Springer Berlin Heidelberg (Cited on pages 15, 21, 65, 101).
- Effinger, Robert, Brian Williams, and Andreas Hofmann (2010). “Dynamic Execution of Temporally and Spatially Flexible Reactive Programs”. In: *AAAI Conference on Artificial Intelligence Workshops* (Cited on page 18).
- Ferrein, Alexander and Gerhard Lakemeyer (2008). “Logic-Based Robot Control in Highly Dynamic Domains”. In: *Robot. Auton. Syst.* 56.11. Semantic Knowledge in Robotics, pp. 980–991. ISSN: 0921-8890 (Cited on page 20).

- Fikes, Richard (Aug. 1971). “Monitored Execution of Robot Plans Produced By Strips”. In: *IFIP Congress*. Ljubljana, Yugoslavia (Cited on pages 5, 6, 14, 16, 97, 103).
- Firby, James (1987). “An Investigation into Reactive Planning in Complex Domains”. In: *AAAI*. Vol. 87, pp. 202–206 (Cited on page 14).
- Firby, Robert James (1989). *Adaptive Execution in Complex Dynamic Worlds*. Yale University (Cited on pages 11, 15, 64).
- Forgy, Charles (1989). “Rete: a Fast Algorithm for the Many Pattern/many Object Pattern Match Problem”. In: *Readings in Artificial Intelligence and Databases*. Elsevier, pp. 547–559 (Cited on page 48).
- Foughali, Mohammed, Félix Ingrand, and Anthony Mallet (July 2018). “GENOM3 Templates: from Middleware Independence to Formal Models Synthesis”. In: *arXiv preprint* (Cited on page 11).
- Fox, Maria and Derek Long (2003). “PDDL2.1: an Extension to PDDL for Expressing Temporal Planning Domains”. In: *JAIR* 20, pp. 61–124 (Cited on pages 97, 127).
- Fratini, Simone and Amedeo Cesta (2012). “The APSI Framework: a Platform for Timeline Synthesis”. In: *Workshop on Planning and Scheduling with Timelines*, pp. 8–15 (Cited on page 13).
- Fratini, Simone, Amedeo Cesta, Riccardo De Benedictis, Andrea Orlandini, and Riccardo Rasconi (2011). “APSI-Based Deliberation in Goal Oriented Autonomous Controllers”. In: (Cited on page 99).
- Gat, Erann (1991). “Alfa: A Language for Programming Reactive Robotic Control Systems”. In: *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE Computer Society, pp. 1116–1117 (Cited on page 10).
- (1992). “Integrating Planning and Reacting in A Heterogeneous Asynchronous Architecture for Controlling Real-World Mobile Robots”. In: *National Conference on Artificial Intelligence* (Cited on page 10).
- (1997). “ESL: a Language for Supporting Robust Plan Execution in Embedded Autonomous Agents”. In: *IEEE Aerospace Conference*. Vol. 1. IEEE, pp. 319–324 (Cited on pages 12, 15, 65, 67).
- Gat, Erann, Peter Bonnasso, and Robin Murphy (1998). “On Three-Layer Architectures”. In: *Artificial intelligence and mobile robots* 195, p. 210 (Cited on page 9).
- Gerevini, Alfonso (2005). “Incremental Qualitative Temporal Reasoning: Algorithms for The Point Algebra and The ORD-Horn Class”. In: *Artif. Intell.* 166, pp. 37–80 (Cited on pages 126, 198).
- Ghallab, Malik and Hervé Laruelle (1994). “Representation and Control in IxTeT, a Temporal Planner”. In: *International Conference on AI Planning Systems (AIPS)*, pp. 61–67 (Cited on pages 5, 11, 16, 24, 99).
- Ghallab, Malik, Dana Nau, and Paolo Traverso (2016). *Automated Planning and Acting*. Cambridge University Press (Cited on pages 5–7, 21–24, 29, 30, 34, 102, 215).
- Gil, Yolanda (1994). “Learning by Experimentation: Incremental Refinement of Incomplete Planning Domains”. In: *Machine Learning Proceedings 1994*. Elsevier, pp. 87–95 (Cited on page 171).
- Godet, Roland and Arthur Bit-Monnot (2022). “Chronicles for Representing Hierarchical Planning Problems With Time”. In: (Cited on pages 24, 25, 101, 103).

- Hähnel, Dirk, Wolfram Burgard, and Gerhard Lakemeyer (1998). “Golex - Bridging the Gap Between Logic (Golog) and a Real Robot”. In: *KI Advances in Artificial Intelligence*. Springer, pp. 165–176 (Cited on page 20).
- Haigh, Karen and Manuela Veloso (1998). “Planning, Execution and Learning in A Robotic Agent”. In: *Artificial Intelligence Planning and Scheduling (AIPS)* (Cited on page 15).
- Haigh, Karen, Manuela Veloso, and George Bekey (Mar. 1998). “Interleaving Planning and Robot Execution for Asynchronous User Requests”. In: *Auton. Robot.* 5.1, pp. 79–95 (Cited on page 15).
- Hofmann, Till, Tarik Viehmann, Mostafa Gomaa, Daniel Habering, Tim Niemueller, and Gerhard Lakemeyer (2021). “Multi-Agent Goal Reasoning With The Clips Executive in The Robocup Logistics League.” In: *ICAART*, pp. 80–91 (Cited on pages 21, 167).
- Höller, Daniel, Gregor Behnke, Pascal Bercher, Susanne Biundo, Humbert Fiorino, Damien Pellier, and Ron Alford (2020). “HDDL: an Extension To PDDL for Expressing Hierarchical Planning Problems”. In: vol. 34. 06, pp. 9883–9891 (Cited on page 100).
- Ingrand, Félix, Raja Chatila, Rachid Alami, and Frédéric Robert (1996). “PRS: a High Level Supervision and Control Language for Autonomous Mobile Robots”. In: *IEEE International Conference on Robotics and Automation (ICRA)*. Vol. 1. IEEE, pp. 43–49 (Cited on pages 11, 16, 17, 30, 65, 85, 101).
- Ingrand, Félix and Malik Ghallab (June 2017). “Deliberation for Autonomous Robots: A Survey”. In: *Artif. Intell.* 247, pp. 10–44 (Cited on pages 2–4, 7).
- Ingrand, Félix, Simon Lacroix, Solange Lemai-Chenevier, and Frédéric Py (2007). “Decisional Autonomy of Planetary Rovers”. In: *J. Field Robotics* 24 (Cited on page 10).
- Ingrand, Félix and Frédéric Py (May 2002). “an Execution Control System for Autonomous Robots”. In: *IEEE International Conference on Robotics and Automation (ICRA)* (Cited on page 11).
- Iovino, Matteo, Edvards Scukins, Jonathan Styrud, Petter Ögren, and Christian Smith (2022). “A Survey of Behavior Trees in Robotics and Ai”. In: *Robot. Auton. Syst.* 154, p. 104096 (Cited on page 19).
- Jónsson, Ari, Paul Morris, Nicola Muscettola, Kanna Rajan, and Ben Smith (2000). “Planning in Interplanetary Space: Theory and Practice.” In: *Artificial Intelligence Planning and Scheduling (AIPS)*. Citeseer, pp. 177–186 (Cited on page 99).
- Juba, Brendan, Hai S Le, and Roni Stern (2021). “Safe Learning of Lifted Action Models”. In: *arXiv preprint arXiv:2107.04169* (Cited on page 171).
- Kamperis, Oliver Michael, Yongjing Wang, and Marco Castellani (2023). “Online Hierarchical Conformance Refinement Planning for Autonomous Robots”. In: *28th International Conference on Automation and Computing (ICAC)*, pp. 1–6 (Cited on page 170).
- Kortenkamp, David and Reid Simmons (2008). “Robotic Systems Architectures and Programming”. In: *Handbook of Robotics*. Ed. by B Siciliano and Oussama Khatib. Springer, pp. 187–206 (Cited on page 8).
- Lallement, Raphaël, Lavindra De Silva, and Rachid Alami (2014). “HATP: an HTN Planner for Robotics”. In: *arXiv preprint arXiv:1405.5345* (Cited on page 100).

- Lemai-Chenevier, Solange (2004). “IxTeT-Exec : Planification, Réparation de Plan et Contrôle d’Exécution avec Gestion du Temps et des Ressources”. PhD thesis (Cited on pages 16, 18, 24, 107).
- Lesire, Charles, David Doose, and Christophe Grand (2020). “Formalization of Robot Skills with Descriptive and Operational Models”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, pp. 7227–7232 (Cited on pages 20, 24, 63).
- Lesire, Charles and Franck Pommereau (2018). “ASPiC: an Acting System Based on Skill Petri Net Composition”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, pp. 6952–6958 (Cited on pages 20, 63).
- Levesque, Hector, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard Scherl (1997). “Golog: a Logic Programming Language for Dynamic Domains”. In: *The Journal of Logic Programming* 31.1-3, pp. 59–83 (Cited on pages 20, 65).
- Levine, Steven and Brian Williams (2014). “Concurrent Plan Recognition and Execution for Human-Robot Teams”. In: *International Conference on Automated Planning and Scheduling*. Vol. 24, pp. 490–498 (Cited on page 18).
- Levinson, Richard (1995). “A General Programming Language for Unified Planning and Control”. In: *Artif. Intell.* 76.1-2, pp. 319–375 (Cited on pages 16, 62, 65).
- Li, Ruoxi, Sunandita Patra, and Dana Nau (2021). “Decentralized Refinement Planning and Acting”. In: *International Conference on Automated Planning and Scheduling (ICAPS)* 31, pp. 225–233 (Cited on page 22).
- Macenski, Steven, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall (2022). “Robot Operating System 2: Design, Architecture, and Uses in The Wild”. In: *Science Robotics* 7.66, eabm6074 (Cited on page 167).
- Martín, Francisco, Jonatan Ginés Clavero, Vicente Matellán, and Francisco Rodríguez (2021). “PlanSys2: A Planning System Framework for ROS2”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. IEEE, pp. 9742–9749 (Cited on page 15).
- Mataré, Victor, Tarik Viehmann, Till Hofmann, Gerhard Lakemeyer, Alexander Ferrein, and Stefan Schiffer (2021). “Portable High-level Agent Programming With Golog++.” in: *International Conference on Agents and AI (ICAART)*. SCITEPRESS - Science and Technology Publications, pp. 218–227 (Cited on pages 20, 65).
- Mayima, Amandine, Aurelie Clodic, and Rachid Alami (Aug. 2022). “JAHRVIS, a Supervision System for Human-Robot Collaboration”. In: *IEEE RO-MAN*. IEEE (Cited on page 66).
- McCarthy, John (1960). “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”. In: *Commun. ACM* 3.4, pp. 184–195 (Cited on page 68).
- McDermott, Drew (1991). *A Reactive Plan Language*. Tech. rep. Research Report YALEU/DCS/RR-864, Yale University (Cited on page 64).
- McDermott, Drew, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins (1998). “PDDL: The Planning Domain Definition Language”. In: *Technical Report* (Cited on pages 15, 23, 97, 107).

- McGann, Conor, Frédéric Py, Kanna Rajan, Hans Thomas, Richard Henthorn, and Rob McEwen (2007). “T-REX: a Model-Based Architecture for AUV Control”. In: *Workshop on Planning and Plan Execution for Real-World Systems*. Vol. 2007 (Cited on page 13).
- Medeiros, Adelardo Adelino Dantas de, Raja Chatila, and Sara Fleury (1996). “Specification and Validation of A Control Architecture for Autonomous Mobile Robots”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Vol. 1, 162–169 vol.1 (Cited on page 11).
- Miller, David Paul (1985). *Planning by Search through Simulations*. Yale University (Cited on page 10).
- Moretti, Gino (1979). “The Lambda-Scheme”. In: *Computers and Fluids*. Elsevier (Cited on pages 24, 68, 217).
- Muscettola, Nicola, Gregory Dorais, Chuck Fry, Richard Levinson, Christian Plaunt, and Daniel Clancy (2002). “IDEA: Planning at the Core of Autonomous Reactive Agents”. In: (Cited on pages 13, 14, 18).
- Muscettola, Nicola, Pandurang Nayak, Barney Pell, and Brian Williams (1998). “Remote Agent: To Boldly Go Where No AI System Has Gone Before”. In: *Artif. Intell.* 103.1-2, pp. 5–47 (Cited on pages 11, 13, 18, 65).
- Myers, Karen (1999). “CPEF: Continuous Planning and Execution Framework”. In: *AI Magazine* 20.4, pp. 63–69 (Cited on page 17).
- Natarajan, Mausam and Andrey Kolobov (2022). *Planning With Markov Decision Processes: an Ai Perspective*. Springer Nature (Cited on page 6).
- Nau, Dana (2013). “Game Applications of HTN Planning with State Variables”. In: *ICAPS Workshop on Planning in Games* (Cited on page 100).
- Nau, Dana, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, William Murdock, Dan Wu, and Fusun Yaman (2003). “SHOP2: an HTN Planning System”. In: *JAIR* 20, pp. 379–404 (Cited on page 100).
- Nau, Dana, Yash Bansod, Sunandita Patra, Mark Roberts, and Ruoxi Li (2021). “GT-Pyhop: a Hierarchical Goal+ Task Planner Implemented in Python”. In: p. 21 (Cited on pages 22, 100, 102).
- Nau, Dana, Yue Cao, Amnon Lotem, and Hector Munoz-Avila (1999). “Shop: Simple Hierarchical Ordered Planner”. In: *International Joint Conference on AI (IJCAI)*, pp. 968–973 (Cited on page 6).
- Nau, Dana, Malik Ghallab, and Paolo Traverso (2015). “Blended Planning and Acting: Preliminary Approach, Research Challenges”. In: *AAAI Conference on Artificial Intelligence*. Vol. 29. 1 (Cited on pages 6, 23).
- Nemiro, Liudvikas, Gerard Canal, Oscar Lima, Michael Cashmore, and Mark Roberts (2021). “Designing an Adaptable Benchmark and Competition Simulation for integrated Planning and Execution”. In: *Workshop on the International Planning Competition (WIPC)* (Cited on pages 156, 166).
- Neufeld, Xenija, Sanaz Mostaghim, and Sandy Brand (2018). “A Hybrid Approach To Planning and Execution in Dynamic Environments Through Hierarchical Task Networks and Behavior Trees”. In: *AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 14. 1, pp. 201–207 (Cited on page 19).

- Niemueller, Tim, Till Hofmann, and Gerhard Lakemeyer (2018). “CLIPS-Based Execution for PDDL Planners”. In: *ICAPS Workshop on Integrated Planning, Acting and Execution (IntEx)* (Cited on page 107).
- (2019). “Goal Reasoning in The CLIPS Executive for Integrated Planning and Execution”. In: *International Conference on Automated Planning and Scheduling (AIPS)*. Vol. 29, pp. 754–763 (Cited on page 21).
- Niemueller, Tim, Erez Karpas, Tiago Vaquero, and Eric Timmons (2016). “Planning Competition for Logistics Robots in Simulation”. In: (Cited on pages 156, 167).
- Nilsson, Nils (1969). “A mobile automaton: an application of AI techniques”. In: *IJCAI* (Cited on page 8).
- Patra, Sunandita, Malik Ghallab, Dana Nau, and Paolo Traverso (2019). “Acting and Planning Using Operational Models”. In: *AAAI Conference on Artificial Intelligence*. Vol. 33. 01, pp. 7691–7698 (Cited on pages 21, 23, 101, 166).
- Patra, Sunandita, James Mason, Malik Ghallab, Dana Nau, and Paolo Traverso (2021). “Deliberative Acting, Online Planning and Learning With Hierarchical Operational Models”. In: *Artif. Intell.* (Cited on pages 23, 34, 40, 66, 107, 146, 166, 167, 216, 217).
- Patra, Sunandita, James Mason, Amit Kumar, Malik Ghallab, Paolo Traverso, and Dana Nau (2020). “Integrating Acting, Planning, and Learning in Hierarchical Operational Models”. In: *International Conference on Automated Planning and Scheduling (ICAPS)*. Vol. 30, pp. 478–487 (Cited on pages 21, 50, 101, 166).
- Patra, Sunandita, Paolo Traverso, Malik Ghallab, and Dana Nau (2021). “Coordination and Control of Hierarchically Organized interacting Agents”. In: *International Florida Artificial Intelligence Research Society Conference (FLAIRS)* (Cited on page 2).
- Pellier, Damien, Alexandre Albore, Humbert Fiorino, and Rafael Bailon-Ruiz (2023). “HDDL 2.1: Towards Defining A formalism and A Semantics for Temporal Htn Planning”. In: *ICAPS Workshop on Hierarchical Planning (HPlan)*. This is a challenge paper., pp. 49–53 (Cited on page 100).
- Py, Frédéric, Kanna Rajan, and Conor McGann (2010). “A Systematic Agent Framework for Situated Autonomous Systems”. In: *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 583–590 (Cited on pages 13, 18, 107).
- Quigley, Morgan, Ken Conley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Eric Leibs, Jeremy Berger, Rob Wheeler, and Andrew Ng (2009). “ROS: an Open-Source Robot Operating System”. In: *IEEE ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan, p. 5 (Cited on pages 15, 45).
- Rajan, Kanna and Frédéric Py (2012). “T-REX: Partitioned Inference for AUV Mission Control”. In: *Further Advances in Unmanned Marine Vehicles*, pp. 171–199 (Cited on page 13).
- Rajan, Kanna, Frédéric Py, and Javier Barreiro (2012). “Towards Deliberative Control in Marine Robotics”. In: *Marine Robot Autonomy*. Springer, pp. 91–175 (Cited on page 13).

- Rao, Anand (1996). “AgentSpeak (L): BDI Agents Speak Out in a Logical Computable Language”. In: *European Workshop on Modelling Autonomous Agents in a Multi-Agent World*. Springer, pp. 42–55 (Cited on page 66).
- Rovida, Francesco, Bjarne Grossmann, and Volker Krüger (2017). “Extended Behavior Trees for Quick Definition of Flexible Robotic Tasks”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, pp. 6793–6800 (Cited on page 19).
- Safronov, Evgenii, Michele Colledanchise, and Lorenzo Natale (2020). “Task Planning With Belief Behavior Trees”. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, pp. 6870–6877 (Cited on page 19).
- Santana, Pedro and Brian Williams (May 2014). “Chance-Constrained Consistency for Probabilistic Temporal Plan Networks”. In: *ICAPS 24.1*, pp. 271–279 (Cited on page 18).
- Scala, Enrico, Patrik Haslum, Sylvie Thiébaux, and Miquel Ramirez (2016). “Interval-based relaxation for general numeric planning”. In: *ECAI 2016*. IOS Press, pp. 655–663 (Cited on page 5).
- Shivashankar, Vikas, Ugur Kuter, Dana Nau, and Ron Alford (2012). “A Hierarchical Goal-Based formalism and Algorithm for Single-Agent Planning”. In: *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 981–988 (Cited on page 100).
- Shivers, Olin (1988). “Control Flow Analysis in Scheme”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 164–174 (Cited on page 110).
- Simmons, Reid (1991). “Coordinating Planning, Perception, and Action for Mobile Robots”. In: *ACM SIGART Bulletin 2.4*, pp. 156–159 (Cited on page 10).
- (1992). “Concurrent Planning and Execution for Autonomous Robots”. In: *Control Systems, IEEE 12.1*, pp. 46–50 (Cited on page 15).
- (1994). “Structured Control for Autonomous Robots”. In: *IEEE Trans. Robot. Autom.* 10.1, pp. 34–43 (Cited on page 68).
- Smith, David, Jeremy Frank, and William Cushing (2008). “The ANML Language”. In: *ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*. Vol. 31 (Cited on pages 99, 107).
- Sowmya, Arcot, David Tsz-Wang So, and Wan Hung Tang (July 2002). “Design of A Mobile Robot Controller Using Esterel Tools”. In: *Electron. Notes Theor. Comput. Sci.* 65.5, pp. 3–10 (Cited on page 62).
- Steele, Guy (1990). *Common Lisp: The Language*. Elsevier (Cited on page 68).
- Teichteil-Königsbuch, Florent, Ugur Kuter, and Guillaume Infantes (2010). “incremental Plan Aggregation for Generating Policies in Mdps”. In: *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1231–1238 (Cited on page 6).
- Tellier, Daniel, Meyer Millman, Brian McClelland, Kate Beatrix Go, Alice Balayan, Michael Munje, Kyle Dewey, Nhut Ho, Klaus Havelund, and Michel Ingham (2020). “Towards The Hierarchical State Machine Oriented Proteus Systems Programming

- Language”. In: *AIAA 2020-4222 Session: Advances in Software for Space* (Cited on page 63).
- Thomas, Ulrike, Gerd Hirzinger, Bernhard Rumpe, Christoph Schulze, and Andreas Wortmann (2013). “A New Skill Based Robot Programming Language Using Uml/p Statecharts”. In: *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, pp. 461–466 (Cited on page 63).
- Turi, J  r  my and Arthur Bit-Monnot (2022a). “Extending a Refinement Acting Engine for Fleet Management: Concurrency and Resources”. In: (Cited on pages 24, 25, 54).
- (2022b). “Guidance of a Refinement-Based Acting Engine with a Hierarchical Temporal Planner”. In: *ICAPS Workshop on Integrated Planning, Acting, and Execution (IntEx)* (Cited on pages 25, 86, 109, 112, 124).
- Turi, J  r  my, Arthur Bit-Monnot, and F  lix Ingrand (July 2023). “Enhancing Operational Deliberation in a Refinement Acting Engine with Continuous Planning”. In: *ICAPS Workshop on Integrated Acting, Planning and Execution (IntEx)*. Prague, Czech Republic (Cited on pages 25, 112).
- Umbrico, Alessandro, Amedeo Cesta, Marta Cialdea Mayer, and Andrea Orlandini (2017). “PLATINUM: a New Framework for Planning and Acting”. In: *AI*IA Advances in Artificial Intelligence*. Springer, pp. 498–512 (Cited on pages 99, 103).
- Valentini, Alessandro, Andrea Micheli, and Alessandro Cimatti (2020). “Temporal planning with intermediate conditions and effects”. In: *AAAI/AI*. Vol. 34. 06, pp. 9975–9982 (Cited on page 99).
- Vapsi, Annita, Daniel Borrajo, and Manuela Veloso (2023). “CLAPLEX a Control Language and Architecture for Planning and Execution”. In: *ICAPS Workshop on Integrated Acting, Planning and Execution (IntEx)* (Cited on page 66).
- Veloso, Manuela and Paola Rizzo (1998). “Mapping Planning Actions and Partially-Ordered Plans into Execution Knowledge”. In: *Workshop on Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments*. Citeseer, pp. 94–97 (Cited on page 14).
- Verma, Vandi, Ari Jonsson, Corina Pasareanu, and Michael Iatauro (2006). “Universal-Executive and PLEXIL: Engine and Language for Robust Spacecraft Control and Operations”. In: *Space 2006*. American Institute of Aeronautics and Astronautics (Cited on page 64).
- Vidal, Thierry and Malik Ghallab (1996). “Dealing With Uncertain Durations in Temporal Constraint Networks Dedicated To Planning”. In: *European Conference on Artificial Intelligence (ECAI)*. PITMAN, pp. 48–54 (Cited on page 18).
- Volpe, Richard, Issa Nesnas, Tara Estlin, Darren Mutz, Richard Petras, and Hari Das (2001). “The CLARAty Architecture for Robotic Autonomy”. In: *IEEE Aerospace Conference Proceedings*. Vol. 1. IEEE, pp. 1–121 (Cited on page 12).
- Wang, Fei-Yue, Kanstantinos Kyriakopoulos, Athanasios Tsolkas, and George Saridis (1991). “A Petri-Net Coordination Model for an intelligent Mobile Robot”. In: *IEEE Trans. Syst., Man, Cybern.* 21.4, pp. 777–789 (Cited on page 20).
- Wang, Xuemei (1995). “Learning by Observation and Practice: An Incremental Approach for Planning Operator Acquisition”. In: *Machine Learning Proceedings 1995*. Elsevier, pp. 549–557 (Cited on page 171).

- Wilkins, David (1988). *Practical planning: extending the classical AI planning paradigm*. Elsevier (Cited on page 17).
- Wilkins, David and Karen Myers (1995). “A Common Knowledge Representation for Plan Generation and Reactive Execution”. In: *J. Logic Comput.* 5.6, pp. 731–761 (Cited on page 17).
- Wilkins, David E and Karen L Myers (1997). “The Act Formalism”. In: *SRI International Artificial Intelligence Center* (Cited on page 17).
- Williams, Brian, Michael Ingham, Seung Chung, and Paul Elliott (2003). “Model-Based Programming of intelligent Embedded Systems and Robotic Space Explorers”. In: *Proc. IEEE* 91.1, pp. 212–237 (Cited on pages 63, 67).
- Wygant, Robert (1989). “CLIPS—a Powerful Development and Delivery Expert System Tool”. In: *Computers and Industrial Engineering* 17.1-4, pp. 546–549 (Cited on page 21).
- Zaidins, Paul, Mark Roberts, and Dana Nau (2023). “Implicit Dependency Detection for Htn Plan Repair”. In: *ICAPS Workshop on Hierarchical Planning (HPlan)*, pp. 10–18 (Cited on pages 22, 102).
- Ziparo, Vittorio Amos, Luca Iocchi, Daniele Nardi, Pier Palamara, and Hugo Costelha (2011). “Petri Net Plans : a formal Model for Representation and Execution of Multi-Robot Plans”. In: *Autonomous Agents and Multi-Agent Systems* 23.3, pp. 344–383 (Cited on page 20).

Detailed translation techniques of SOMPAS programs into chronicles

A.1 Pre-processing of programs

A.1.1 Expanding user-defined functions

During the preprocessing of the programs, we replace all calls to lambdas with new expressions in which the abstract calls are removed. Such an expression is expanded as a `begin` expression, where the parameter bindings are replaced by `define` expressions. Since there are translation rules for `begin` and `define` expressions, this should make the process easier. The expansion works like this: for each expression of the form

```
((lambda (p1 ... pn) body) v1 ... vn)
```

, the expression is replaced by

```
(begin
  (define p1 1)
  ...
  (define pn vn)
  body)
```

A.1.2 Pre-evaluation of static expressions of programs

To simplify the programs, the translated programs are pre-evaluated, i.e. any pure computation p_c contained in them is replaced by the result of the evaluation of p_c . In SOMPAS an expression e is pure if:

- e is an atom,
- e is a list in which all its arguments are pure, and in particular the first element of the list is a function that has been marked as pure. A function is pure if the result it produces only depends on its arguments and not on any external parameters, and the function has no side effects. For example the `+` operator is pure, whereas `rand-int-in-range` is not.

To pre-evaluate a program, we use the function $\text{PREVAL}(x, p_{env})$, which is similar to $\text{EVAL}(x, env)$ presented shown in Algorithm 3.1. It uses additional annotations to distinguish between *pure* and *unpure* expressions. An example of a pre-evaluation is given in Example A.1.

Example A.1

Let us take the following program:

```
(begin
  (define x (* 3 4))
  (define y (+ 5 6))
  (exec-command 'move_to ?r (list x y)))
```

The bindings of x and y result from pure computations, and can be evaluated. The last expression is partially evaluated as *assert* is *unpure*, but its last argument has been evaluated, and quoted to avoid a new evaluation that would result in a runtime error. The resulting program is:

```
(begin
  nil
  nil
  (exec-command 'move_to ?r '(12 11)))
```

A.1.3 Result of the expansion of the body of the method pick&drop

Here is the result of expanding the program presented in Listing 4.1 using the processes described above. Note that the $(\text{check } e)$ expressions have been expanded to the $(\text{begin } (\text{define } __r__ e)(\text{if } __r__ \text{ true } (\text{err nil})))$ expressions.

Listing A.1: Expanded program of the method pick&drop

```
(begin
  (define ___r___
    (begin
      (define ___r___
        (begin
          (define ___r___
            (begin
              (define a (instance ?b (quote ball)))
              (if a nil (err nil))))
            (if (err? ___r___) ___r___
              (begin
                (define ___r___
                  (begin
                    (define a (instance ?r (quote location))))
```

```

        (if a nil (err nil))))
      (if (err? ___r___) ___r___
        (begin
          (define ___r___
            (begin
              (define a (instance ?g (quote gripper)))
              (if a nil (err nil))))
          (if (err? ___r___) ___r___
            (begin
              (define a (instance ?p (quote location)))
              (if a nil (err nil))))))))))
    (if (err? ___r___) ___r___
      (begin
        (define ___r___
          (begin
            (define a
              (= (begin
                  (define ?b ?b)
                  (read-state (quote pos) ?b))
                ?p))
              (if a nil (err nil))))
          (if (err? ___r___) ___r___
            (begin
              (define a (!= ?p (quote roby)))
              (if a nil (err nil))))))))
    (if (err? ___r___) ___r___
      (begin
        (define ___r___ (define r_h (ctx-acquire 0 (quote roby))))
        (if (err? ___r___) ___r___
          (begin
            (define ___r___
              (begin
                (define ?r ?p)
                (ctx-exec-task 0 (quote go2) ?r)))
            (if (err? ___r___) ___r___
              (begin
                (define ___r___
                  (begin
                    (define ?obj ?b)
                    (define ?location ?p)
                    (define ?gripper ?g)
                    (ctx-exec-command 1 (quote pick) ?obj
                      ?location ?gripper)))
                  (if (err? ___r___) ___r___
                    (if (err? ___r___) ___r___

```

```

(begin
  (define ___r___
    (begin
      (define ?r ?r)
      (ctx-exec-task 2 (quote go2) ?r)))
  (if (err? ___r___) ___r___
      (begin
        (define ?obj ?b)
        (define ?location ?r)
        (define ?gripper ?g)
        (ctx-exec-command 3 (quote drop) ?obj
          ?location ?gripper)))))))))

```

A.2 Post-processing of the flow graph

The post-processing of a *flow graph* aims at checking the validity of the program, i.e. if the evaluation of such a program is possible, and at specializing the program to represent a predictive model that can be used by a planner to anticipate the successful execution of it. In this section, we introduce the tools necessary to represent labels, and in particular to represent their domains, which is the set of possible values they can take. Then we present how to manipulate *flow graphs* based on the representation of labels we have here.

A.2.1 Representation of the labels

In the *flow graph*, labels represent the intermediate values in a program. As variables, different labels can refer to the same value. They can be unified in order to simplify the representation of a program. As an example, consider the following pseudocode:

$$\begin{aligned}
 x &:= 1; \\
 z &:= x; \\
 y &:= x + 2;
 \end{aligned}$$

The value "x" can be replaced by "1" at any point "x" is present, which gives:

$$\begin{aligned}
 z &:= 1; \\
 y &:= 3;
 \end{aligned}$$

In addition to removing the line that assigns a constant value to "x", "y" can be statically computed.

To perform such processing on a synthesized *flow graph*, we can rely on the Union-Find algorithm, which unifies trees in a forest. We use it to unify the labels in the *flow graph*. To do this, we rely on a *symbol table*, whose underlying data structure is a *forest*, in which all labels are declared and associated with a unique *ID*, a domain, and a set $Drops(l)$, which is the set of moments at which a reference to the label is dropped. In fact, in programs, references are defined in a given scope. In particular, in Scheme, values are defined only in the scope in which they are created.

The *ID* is used to uniquely refer to a tree in the forest. For a label l , we denote $\mathcal{D}(l)$ as its domain.

To use the Union-Find algorithm, we add the following attribute to a label

- The ID of the parent, by default the ID of the label,
- The rank of the label, representing the number of children the node has. The default rank is 1, since a node is at least its own child.

The different functions of Union-Find are presented in the algorithm A.1. The function $UNION(X,Y)$ connects two labels based on their rank: the label with the highest rank becomes the parent of the other. The function $FIND(x)$ returns the parent of a node, which is found by recursively checking the parent of a node until the root of a tree is reached. In the meantime, the tree is flattened so that the parent of x is now the root of the tree. This operation reduces the computation time for the next time $FIND(x)$ is called. We add the $UNIONORDERED(x, y)$ function which forces x to be used as the parent of the tree. $FLATBINDINGS()$ reduces the level of each tree to one. In Figure A.1 we present an example of the application of the Union-Find algorithms, starting from a forest composed of atomic trees.

A.2.1.1 Label Binding

The analysis of the *flow graph* detects primitive expressions of the form $(t_i : r_a = r_b)$, where that r_a and r_b are labels. For such an expression, we bind r_a and r_b and remove the primitive expression from the flow graph. To do this, we use the Union-Find algorithm to perform the binding. In particular, we use the function $UnionOrdered(r_a, r_b)$ (see Algorithm A.1), which binds r_b to r_a such that:

- r_a replaces r_b wherever it appears in the program,
- the allowed values of r_a are restricted to contain only allowed values of r_b (i.e. $D(r_a) \leftarrow MEET(D(r_a), D(r_b))$),
- the sets $Drops(r_a)$ and $Drops(r_b)$ are merged because they refer to the same value.

A.2.2 Domain of a label

As mentioned before, a label is associated with a domain which represents the allowed values that a label can take. The typing system of SOMPAS is defined as a lattice, an oriented graph in which each pair of elements has a unique supremum, defined as the lowest common ancestor, and a unique infimum, defined as the greatest common children. Figure A.2a is the lattice representing the types of the variables in SOMPAS:


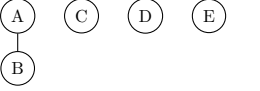
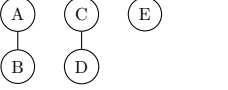
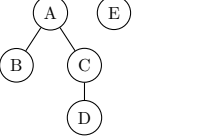
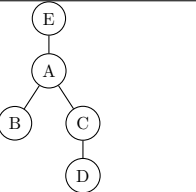
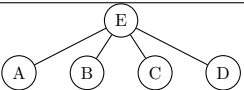
Operation		Resulting Forest
Init	→	
Union(A,B)	→	
Union(C,D)	→	
Union(A, D)	→	
UnionOrdered(E,B)	→	
FlatBindings()	→	

Figure A.1: An example of successive operations on a forest $\{A, B, C, D, E\}$ using the Union-Find algorithms (see Algorithm A.1).

Algorithm A.1 Union-find algorithm

```

function UNION(x,y)
   $p_x \leftarrow \text{FIND}(x)$ 
   $p_y \leftarrow \text{FIND}(y)$ 
  if  $p_x \neq p_y$  then
    if  $p_x.\text{rank} < p_y.\text{rank}$  then
       $p_x.\text{parent} \leftarrow p_y.\text{parent}$ 
    else
       $p_y.\text{parent} \leftarrow p_x.\text{parent}$ 
      if  $p_x.\text{rank} = p_y.\text{rank}$  then
         $x.\text{rank} \leftarrow x.\text{rank} + 1$ 

function FIND(x)
   $\text{parent} \leftarrow x.\text{parent}$ 
  if  $x \neq \text{parent}$  then
     $\text{parent} \leftarrow \text{FIND}(\text{parent})$ 
     $x.\text{parent} \leftarrow \text{parent}$ 
     $\text{parent}.\text{rank} \leftarrow \text{parent}.\text{rank} + 1$ 
  return parent

function FLATBINDINGS(f)
  for all  $x \in f$  do
    FIND(x)

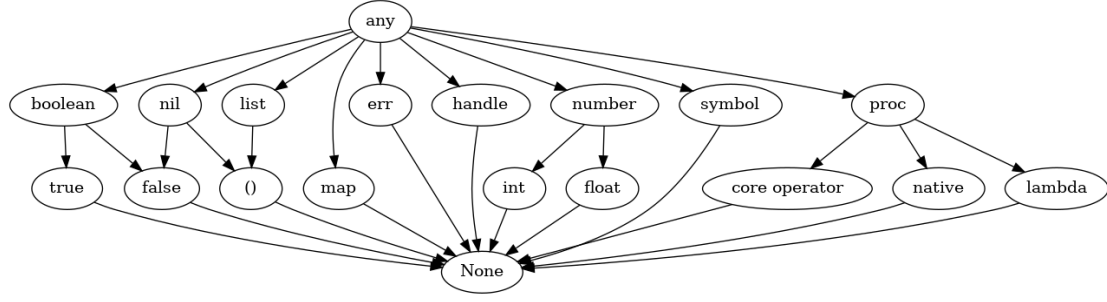
function UNIONORDERED(x,y)
   $p_x \leftarrow \text{FIND}(x)$ 
   $p_y \leftarrow \text{FIND}(y)$ 
  if  $p_x \neq p_y$  then
     $y.\text{parent} \leftarrow x.\text{parent}$ 
     $x.\text{rank} \leftarrow x.\text{rank} + 1$ 

```

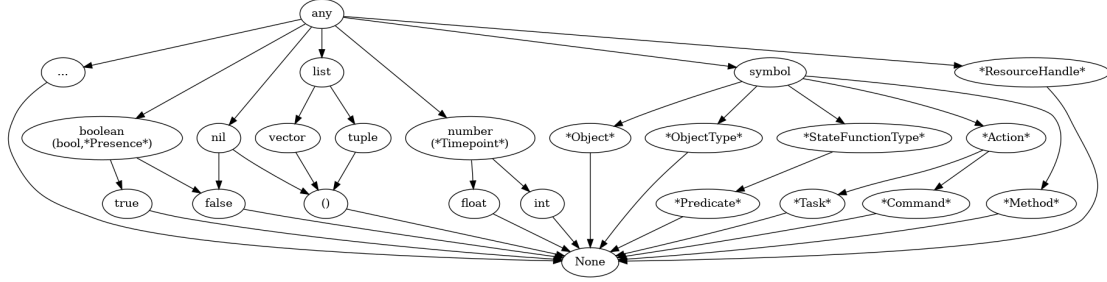
all expressions are of type *Any*, which is then specialized to, e.g. *Number*. The type \emptyset represents an empty domain, i.e. a label has no valid value. We use an extension of this lattice to represent the additional types inherent to our planning representation (see Figure A.2b).

The domain of a label is more complex than a single type. Since it represents all the possible values that a label can take, we need a definition of a domain that allows this. Therefore, we suggest that a domain can be defined with one of the following types:

- *Type*(t) where t is a type defined in the lattice,
- *Singleton*(t, v) where t the type is the type of singleton and v is its value. The value of v can be one of *Int*(i), *Float*(f), *Boolean*(b) or *Symbol*(s). Note that two singletons with the same value can be of different types, e.g. *Singleton*(*Symbol*, "robby") and *Singleton*(*Robot*, "robby").



(a) Lattice representing the typing system of SOMPAS.



(b) SOMPAS lattice extended with types required by planning

Figure A.2: Lattices representing the lattice system of SOMPAS and its extension to handle specific types for planning

- $UNIFY(d_1, \dots, d_n)$ is the union of multiple domains, e.g. $UNIFY(Type(True), Type(Nil))$
- $Composition(t, d)$ represents compound types such as $Composition(Handle, Type(Int))$, which represents the result of an asynchronous computation that returns an integer.
- $Subtraction(d_1, d_2)$ represents a domain as the subtraction of two domains: $d_1 \setminus d_2$.

A.2.3 Operations on the lattice of type

By using the lattice, we can perform some operations on the domains. This is particularly useful during the static analysis of a *flow graph*, since we want to check whether the unification of two labels l_1 and l_2 is allowed. In fact, a unification is permitted if and only if the resulting domain is not empty, i.e. it is different from \emptyset . To do this, we want to compute the intersection of the domains $\mathcal{D}(l_1)$ and $\mathcal{D}(l_2)$. In a generic lattice, we have two operators:

- $INFIMUM(n_1, n_2)$ returns the greatest child of two nodes.
- $SUPREMUM(n_1, n_2)$ returns the lowest parent of two nodes.

In the lattice we define the rule $D = Decomposition(t_1)$ which says that $D = \bigcup_i t_i$, and $D = t_1$. $Perfect(D)$ states that D is a perfect decomposition, such that $\forall (t_1, t_2) \in (D, D), t_1 \cap t_2 = \emptyset$. In addition, we define the following operators:

- The $\text{MEET}(d_1, d_2)$ operator that returns $d_1 \cap d_2$, It is used to check if a unification is allowed.
- The $\text{UNIFY}(d_1, d_2)$ operator that returns $d_1 \cup d_2$, It is used to define the result of a *branching* flow in which the domain of the result can be one of the domain of the results of the two branches.
- The $\text{SUBTRACT}(d_1, d_2)$ operator that returns $d_1 \setminus d_2$.

Operator Meet We define the following rules for the $\text{MEET}(d_1, d_2)$ operator in function of the pair (d_1, d_2) :

- $(\text{Type}(t_1), \text{Type}(t_2))$: $\text{Infimum}(t_1, t_2)$,
- $(\text{Type}(t_1), \text{Composed}(t_2, d_2))$: $\text{Composed}(\text{MEET}(t_1, t_2), d_2)$,
- $(\text{Type}(t_1), \text{Constant}(v, d))$: $\text{Constant}(v, \text{MEET}(t_1, d))$,
- $(\text{Composed}(t_1, d_1), \text{Composed}(t_2, d_2))$: $\text{Composed}(\text{MEET}(t_1, t_2), \text{MEET}(d_1, d_2))$,
- $(\text{Constant}(v_1, d_1), \text{Constant}(v_2, d_2))$:
 - If $v_1 = v_2$: $\text{Constant}(v_1, \text{MEET}(d_1, d_2))$,
 - else *Empty*
- $(d, \text{UNIFY}(d_1, \dots, d_n))$: $\text{UNIFY}(\text{MEET}(d, d_1), \dots, \text{MEET}(d, d_n))$,
- $(d, \text{Subtract}(d_1, d_2))$: $\text{UNIFY}(\text{MEET}(t_1, d_1), \text{MEET}(t_2, d_2))$.

Operator Unify We define the following rules for the $\text{UNIFY}(d_1, d_2)$ operator in function of the pair (d_1, d_2) :

- $(\text{Type}(t_1), \text{Type}(t_2))$: $\text{Supremum}(t_1, t_2)$,
- $(\text{Type}(t_1), \text{Composed}(t_2, d_2))$: $\text{Composed}(\text{UNIFY}(t_1, t_2), d_2)$,
- $(\text{Type}(t_1), \text{Constant}(v, d))$: $\text{Constant}(v, \text{UNIFY}(t_1, d))$,
- $(\text{Composed}(t_1, d_1), \text{Composed}(t_2, d_2))$: $\text{Composed}(\text{UNIFY}(t_1, t_2), \text{UNIFY}(d_1, d_2))$,
- $(\text{Constant}(v_1, d_1), \text{Constant}(v_2, d_2))$:
 - if $v_1 = v_2$: $\text{Constant}(v_1, \text{UNIFY}(d_1, d_2))$,
 - else: $\text{UNIFY}(d_1, d_2)$,
- $(d, \text{UNIFY}(d_1, \dots, d_n))$: $\text{UNIFY}(\text{UNIFY}(d, d_1), \dots, \text{UNIFY}(d, d_n))$,
- $(d, \text{Subtract}(d_1, d_2))$: $\text{SUBTRACT}(\text{UNIFY}(d, d_1), d_2)$.

Simplification of Sets Given $U = \text{UNIFY}(d_1, \dots, d_n)$, we simplify the result of U by:

- Removing all duplicates in U ,
- Removing all d such that: $\exists d' \in U, d \subset d'$,
- Merging all the decomposition of a domain d such that:
 $\exists U' \subset U, U' = \text{decomposition}(d) : U = (U \setminus U') \cup d$.

Operator Subtract We define the following rules for the $\text{SUBTRACT}(d_1, d_2)$ operator in function of the pair (d_1, d_2) :

- $(\text{Type}(t_1), \text{Type}(t_2))$: In function of $d = \text{MEET}(t_1, t_2)$
 - If $d \in D = \text{Decomposition}(t_1) \wedge \text{Perfect}(D)$: $D \setminus d$
 - else: $\text{Subtraction}(t_1, d)$,
- $(\text{Type}(t_1), \text{Composed}(t_2, d_2))$: $\text{Composed}(\text{SUBTRACT}(t_1, t_2), d_2)$,
- $(\text{Type}(t_1), \text{Constant}(v, d))$: $\text{Constant}(v, \text{SUBTRACT}(t_1, d))$,
- $(\text{Composed}(t_1, d_1), \text{Composed}(t_2, d_2))$: $\text{Composed}(\text{SUBTRACT}(t_1, t_2), \text{SUBTRACT}(d_1, d_2))$,
- $(\text{Constant}(v_1, d_1), \text{Constant}(v_2, d_2))$
 - If $v_1 = v_2$: $\text{Constant}(v_1, \text{SUBTRACT}(d_1, d_2))$
 - else: t_1 ,
- $(d, \text{UNIFY}(d_1, \dots, d_n))$: $\text{UNIFY}(\text{SUBTRACT}(d, d_1), \dots, \text{SUBTRACT}(d, d_n))$.

We illustrate the functioning of these operators in Example A.2.

Example A.2

Here is a list of computation on different domains. The annotation has been simplified such that $t = \text{Type}(t)$.

$$\text{MEET}(\text{List}, \text{Map}) = \emptyset$$

$$\text{MEET}(\text{Composed}(\text{Err}, \text{Int}), \text{Err}) = \text{Composed}(\text{Err}, \text{Int})$$

$$\text{MEET}(\text{Boolean}, \text{UNIFY}(\text{True}, \text{Symbol})) = \text{True}$$

$$\text{MEET}(\text{UNIFY}(\text{True}, \text{Number}), \text{UNIFY}(\text{Boolean}, \text{Int})) = \text{UNIFY}(\text{True}, \text{Int})$$

$$\text{Unify}(\text{UNIFY}(\text{False}, \text{Map}), \text{UNIFY}(\text{Boolean}, \text{List})) = \text{UNIFY}(\text{Boolean}, \text{Map}, \text{List})$$

$$\text{Unify}(\text{UNIFY}(\text{Symbol}, \text{Float}, \text{Int}, \text{Map}, \text{Proc}), \text{UNIFY}(\text{Boolean}, \text{List}, \text{Handle}, \text{Err}, \text{Map})) = \text{Any}$$

A.2.4 Constraints on the domain of the labels

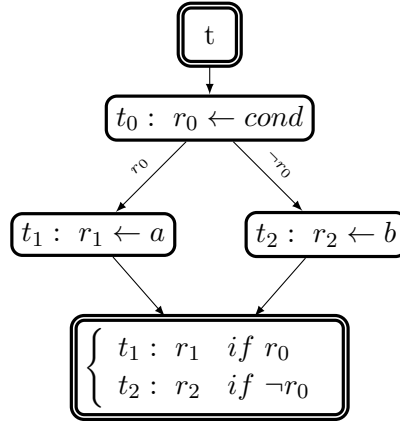
Some constraints can be defined on the domain of variables, since the domain of one label may depend on the domain of another. For example, we can define the domain of the result of a *branching flow* as a set composed of the domains of the result of each branch.

To represent such relationships between domains, we propose to define constraints on the domains of labels, where a constraint $C(d_1, \dots, d_n)$ restricts the possible value of $d \in \{d_1, \dots, d_n\}$.

We then use a constraint propagation system that, when updating $d \in \{d_1, \dots, d_n\}$, propagates $C(d_1, \dots, d_n)$ again. Constraints are automatically generated during the translation process.

We suggest to define the following rules based on the semantics of SOMPAS:

Constraint on domains in a if statement Let us take the expression (**if** cond a b) which corresponding *flow graph* is:



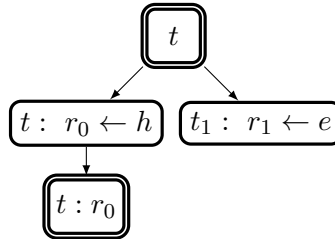
We define the following constraints:

$$D(r_0) \subseteq \text{Boolean}$$

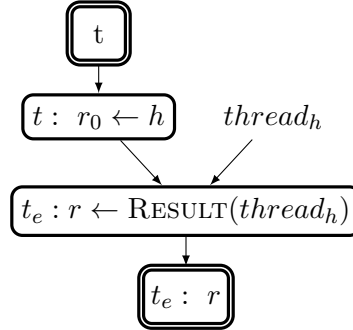
$$D(r) = D(r_1) \cup D(r_2)$$

where r is the result of the flow.

Constraints on variables implied in asynchronous computation Let us take the expression (**async** e), which corresponding *flow graph* is:

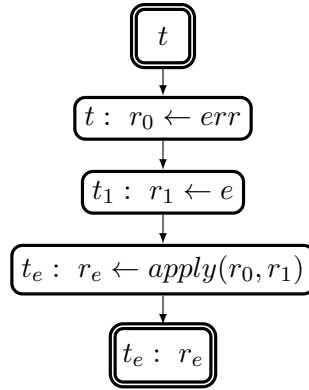


We define the constraint $D(r_0) = \text{Handle}(D(r_1))$. In the same way, we take the expression `(await e)`, which corresponding *flow graph* is:



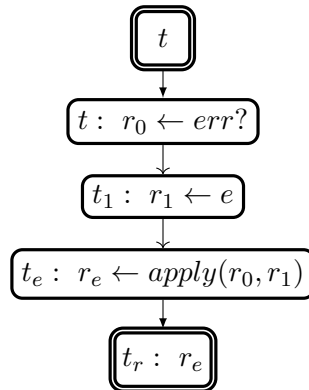
We define the constraint $D(r_0) = \text{Handle}(D(r))$.

Constraint on err Let us take the expression `(err e)` which corresponding *flow graph* is:



We define the constraint $D(r_e) = \text{Err}(D(r_1))$

Constraint on err? Let us take the expression `(err? e)` which corresponding *flow graph* is:



We define the constraints:

$$\begin{aligned}
 D(r_e) &\subseteq \textit{Boolean} \\
 D(r_e) = \textit{True} &\iff D(r_1) \subseteq \textit{Err} \\
 D(r_1) = \textit{False} &\iff D(r_1) \cap \textit{Err} = \emptyset
 \end{aligned}$$

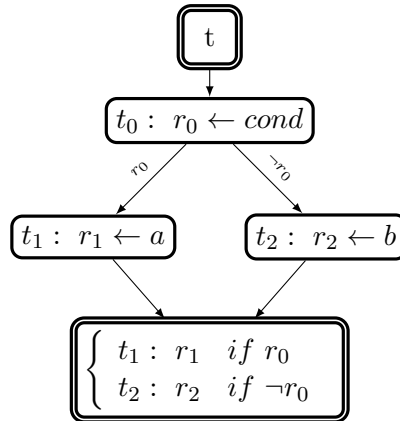
A constraint $C(d_1, \dots, d_n)$ is checked each time a d_i is updated, which may trigger new updates.

Note that more constraints could be defined, especially to infer the result type of more functions. Here we have focused on the constraints specifically needed to synthesize planning models.

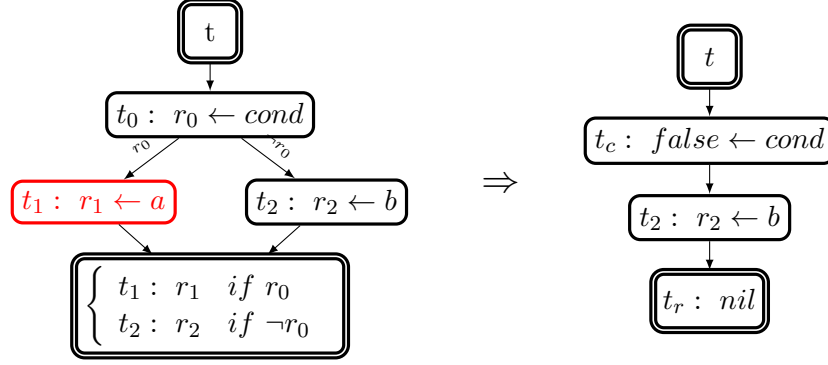
A.2.5 Removing erroneous paths in branching flows

Following the post-processing presented in Section 4.3.2.3, here we detail the process of removing erroneous flows in branching flows. This transformation of the *flow graph* is only acceptable in the context of synthesizing a corresponding predictive model of a program. We emphasize that it is not a strict representation of the program, since some parts have been removed.

With these assumptions in mind, we propose to remove invalid flows, i.e. flows that could lead to runtime errors in branching flows. Consider the following *branching* flow, which represents an **if** expression:



If the *left* branch contains a variable with an empty domain, the branch is invalidated: the flow becomes unreachable by declaring that $D(r_0) = \textit{False}$. The branch can be removed from the *flow graph* to simplify its structure.

*Detection of the invalid branches**Simplification*

The opposite is true if the *right* branch contains a variable with an empty domain.

If it is the main flow that contains a variable with an empty domain, then the whole flow is invalid.

A.3 Simplification of the Temporal Network of a generated chronicle

Here, we extend the presentation of simplifying the TN of a chronicle as presented in Section 4.3.3.3. We recall that in order to simplify the TN of the chronicle, we want to safely remove the *ghost timepoints* of the chronicle. We define *ghost timepoints* as temporal variables that exist in the TN of the chronicle, and that are not used in the other expressions of the chronicle (i.e. conditions, effects, subtasks).

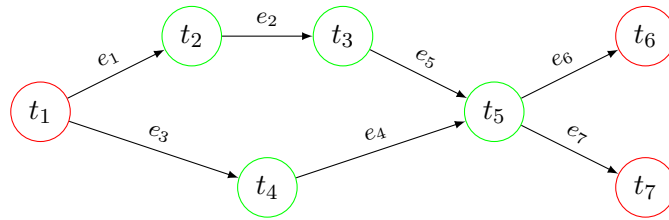
For this purpose we use PA (Gerevini 2005), to first check the path consistency of the TN resulting from the temporal constraints of the chronicle, and then to shrink the set of constraints by removing *ghost timepoints*. The simplification of a TN is allowed because the path consistency of TN has been checked beforehand.

We define $p : (\text{timepoint}, \text{timepoint}) \rightarrow \{<, >, =, \leq, \geq, \neq, \top, \perp\}$ as the PA relation between two timepoints. Any *ghost timepoint* t' can be certainly removed from the chronicle in the following cases:

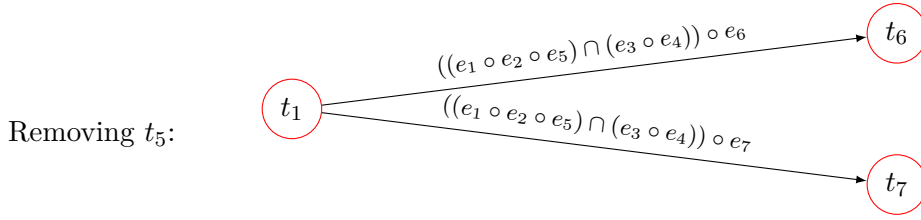
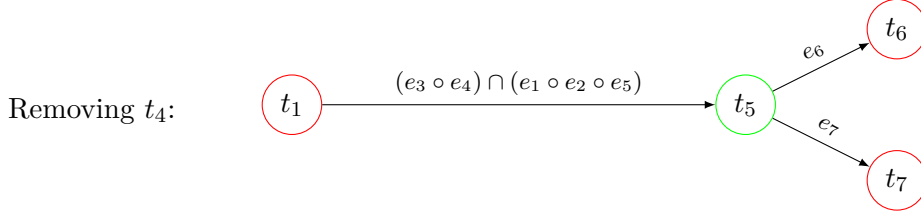
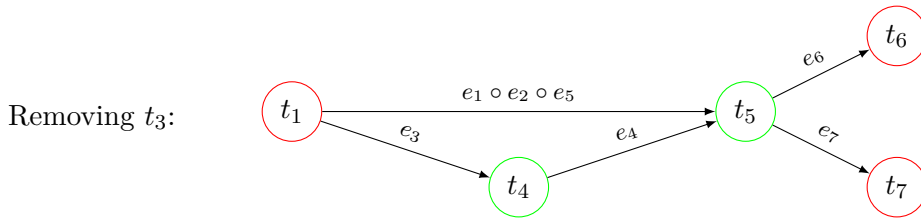
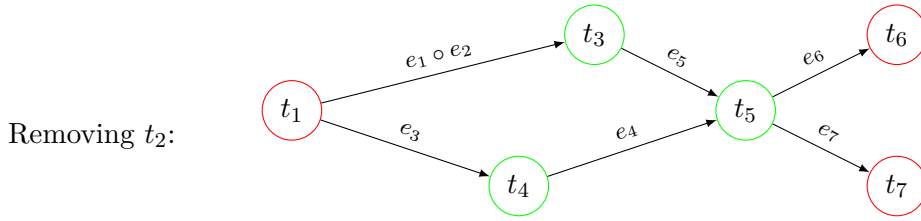
- t' has at most one relation. Any relation it is implied in can also be removed from the chronicle.
- t' has two relations with timepoints t_1 and t_2 , where $p_1 = p(t_1, t')$, $p_2 = p(t', t_2)$, and $p_3 = p(t_1, t_2)$ are the respective paths between t_1 and t' , t' and t_2 , and t_1 and t_2 in the network. We define $p'_3 = p_1 \circ p_2$ as the path between t_1 and t_2 through t' and note that p_3 can be replaced by $p''_3 = p_3 \cap p'_3$. The resulting path is constrained by both p_3 and p'_3 since the path consistency has been checked before. The relations p_1 and p_2 are then removed from the chronicle, and p_3 is overwritten by p''_3 .

Complete example of Temporal Network simplification In the following example we have a TN where the green timepoints are ghost timepoints, and the red ones cannot be removed from TN.

A.3. SIMPLIFICATION OF THE TEMPORAL NETWORK OF A GENERATED CHRONICLE199



Removing the ghost timepoints (in green) results in the following graph, where the paths $p(t_1, t_6)$ and $p(t_1, t_7)$ have been updated, by successively removing t_2, t_3, t_4 and t_5 :



Acting domains

B.1 Gripper domains

B.1.1 Gripper domain

Listing B.1: gripper_base.scm

```
(begin
  (def-types
    (room robot location)
    gripper
    (ball carryable))
  (def-objects
    (left right gripper)
    (empty carryable)
    (robby robot))

  ;state functions
  (def-state-function at-robby (:result room))
  (def-state-function pos (:params (?o carryable)) (:result location))
  (def-state-function carry (:params (?g gripper)) (:result carryable))
  (def-function capacity (:params (?r robot)) (:result object))

  ;actions
  (def-command move (:params (?from room) (?to room)))
  (def-command-pddl-model move
    (:params (?from room) (?to room))
    (:pre-conditions
      (= (at-robby) ?from)
      (!= ?from ?to))
    (:effects
      (durative 5 'at-robby ?to)))

  (def-command pick (:params (?o carryable) (?r room) (?g gripper)))
  (def-command-pddl-model pick
    (:params (?o carryable) (?r room) (?g gripper))
    (:pre-conditions
      (= (pos ?o) ?r)
      (= (at-robby) ?r)
      (= (carry ?g) empty))
    (:effects
      (durative 5 'carry ?g ?o)
      ('pos ?o 'robby)))

  (def-command drop (:params (?o carryable) (?r room) (?g gripper)))
```

```
(def-command-pddl-model drop
  (:params (?o carriable) (?r room) (?g gripper))
  (:pre-conditions
    (= (carry ?g) ?o)
    (= (at-robby) ?r))
  (:effects
    (durative 5 'carry ?g empty)
    ('pos ?o ?r )))
```

Listing B.2: gripper_om.scm

```
(begin
  (def-resources robby left right)
  (def-task go2 (:params (?r room)))
  (def-method go2_noop
    (:task go2)
    (:params (?r room))
    (:pre-conditions (= (at-robby) ?r))
    (:body nil))

  (def-method m_move
    (:task go2)
    (:params (?r room))
    (:pre-conditions (!= (at-robby) ?r))
    (:body (move (at-robby) ?r)))
  ;task with their methods
  (def-task place (:params (?o carriable) (?r room)))

  (def-method place_noop
    (:task place)
    (:params (?o carriable) (?r room))
    (:pre-conditions (= (pos ?o) ?r))
    (:body nil))

  (def-method pick_and_drop
    (:task place)
    (:params (?o carriable) (?r room))
    (:pre-conditions (!= (pos ?o) ?r) (!= (pos ?o) robby))
    (:body
      (do
        (define ?g (arbitrary (instances gripper)))
        (define res_g (acquire ?g))
        (define rh (acquire 'robby))
        (define ?a (pos ?o))
        (go2 ?a)
        (pick ?o ?a ?g)
        (release rh)
        (define rh2 (acquire 'robby))
        (go2 ?r)
        (drop ?o ?r ?g)
        (release rh2)
        (release res_g))))))
```

```

(def-method move_and_drop
  (:task place)
  (:params (?o carriable) (?r room) (?g gripper))
  (:pre-conditions (!= (pos ?o) ?r) (= (carry ?g) ?o))
  (:body
    (do
      (go2 ?r)
      (drop ?o ?r ?g))))
)

```

Listing B.3: gripper_problem.scm

```

(begin
  (def-objects
    (bedroom kitchen lr room)
    (b1 b2 b3 b4 ball))
  (def-facts
    (at-robby lr)
    ((pos b1) bedroom)
    ((pos b2) kitchen)
    ((pos b3) lr)
    ((pos b4) lr)
    ((carry left) empty)
    ((carry right) empty))

  (exec-task place b3 bedroom)
)

```

B.1.2 Gripper-door domain

Listing B.4: gripper_door_base.scm

```

(begin
  (define gripper-door-path
    (concatenate (get-env-var "OMPAS_PATH") "/domains/gripper_door"))
  (set-current-dir gripper-door-path)
  (load "../gripper/base.scm")

  (def-types door)
  ; Additional state functions
  (def-state-function opened
    (:params (?d door))
    (:result boolean))

  (def-function connects
    (:params (?r1 room) (?d door) (?r2 room))
    (:result boolean))
)

```

```

; New commands
(def-command move (:params (?from room) (?to room) (?d door)))
(def-command-pddl-model move
  (:params (?from room) (?to room) (?d door))
  (:pre-conditions
    (= (at-robbly) ?from)
    (connects ?from ?d ?to)
    (opened ?d))
  (:effects
    (durative 5 'at-robbly ?to)
  ))

(def-lambda is_door_of (lambda (?d ?r)
  (exists (instances room)
    (lambda (?r2)
      (connects ?r ?d ?r2)
    )))

(def-command open (:params (?d door) (?r room) (?g gripper)))
(def-command-om-model open
  (:params (?d door) (?r room) (?g gripper))
  (:body
    (do
      (check (= (at-robbly) ?r))
      (check (= (carry ?g) empty))
      (check (is_door_of ?d ?r))
      (durative-effect 5 'opened ?d true))))

(def-command close (:params (?d door) (?r room) (?g gripper)))
(def-command-om-model close
  (:params (?d door) (?r room) (?g gripper))
  (:body
    (do
      (check (= (at-robbly) ?r))
      (check (= (carry ?g) empty))
      (check (is_door_of ?d ?r))
      (durative-effect 5 'opened ?d false))))
)

```

Listing B.5: gripper_door_om.scm

```

(begin
  (define gripper-door-path
    (concatenate (get-env-var "OMPAS_PATH") "/domains/gripper_door"
    ))
  (set-current-dir gripper-door-path)
  (load "../gripper/om.scm")
  (remove-method m_move)

  (def-function min-distance
    (:params (?r1 room) (?r2 room))
    (:result int))
)

```

```

(def-method m_move
  (:task go2)
  (:params (?r room) (?a room) (?n room) (?d door))
  (:cost (+ 1 (min-distance ?n ?r)))
  (:pre-conditions
   (= (at-robby) ?a)
   (!= ?a ?r)
   (connects ?a ?d ?n))
  (:body
   (do
    (t_open ?a ?d)
    (move ?a ?n ?d)
    (go2 ?r))))

(def-task t_open (:params (?r room) (?d door)))
(def-method open_noop
  (:task t_open)
  (:cost 0)
  (:params (?r room) (?d door))
  (:pre-conditions (opened ?d))
  (:body nil))

(def-method open_direct
  (:task t_open)
  (:cost 1)
  (:params (?r room) (?d door) (?g gripper))
  (:pre-conditions (= (carry ?g) empty) (! (opened ?d)))
  (:body
   (open ?d ?r ?g)))

(def-method drop_and_open
  (:task t_open)
  (:params (?r room) (?d door))
  (:cost 3)
  (:pre-conditions
   (! (opened ?d))
   (forall
    (instances gripper)
    (lambda (?g) (!= (carry ?g) empty))))
  (:body
   (do
    (define ?g (arbitrary (instances gripper)))
    (define ?o (carry ?g))
    (drop ?o ?r ?g)
    (open ?d ?r ?g)
    (pick ?o ?r ?g))))
)

```

Listing B.6: gripper_door_problem.scm

```
(begin
```



```

(def-objects
  (b1 b2 ball)
  (r1 r2 room))

(def-initial-state
  ((at b1) r1)
  ((at b2) r1)
  (at-robby r2)
  ((carry left) no_ball)
  ((carry right) no_ball)
  ((connected r1 r2) yes)
  ((connected r2 r1) yes))

(trigger-task pick-and-drop b1 r2)
)

;r1 - r2

```

B.2 Gobot-Sim domain

Listing B.7: gobot_sim_base.scm

```

(begin
  ;types declaration
  (def-types machine package (robot parking_area belt location)
    interact_area)

  ;globals
  (def-function globals.robot_default_battery_capacity (:result float))
  (def-function globals.robot_battery_charge_rate (:result float))
  (def-function globals.robot_battery_drain_rate (:result float))
  (def-function globals.robot_battery_drain_rate_idle (:result float))
  (def-function globals.robot_standard_velocity (:result float))
  (def-function globals.robot_battery_charge_rate_percentage (:result
    float))
  (def-function globals.robot_battery_drain_rate_percentage (:result
    float))
  (def-function globals.robot_battery_drain_rate_idle_percentage (:result
    float))

  ;synthetic state functions
  (def-function travel-time (:params (?l1 location) (?l2 location)) (:
    result float))

  ;state function declaration
  ;robot state functions
  (def-state-function robot.coordinates (:params (?r robot)) (:result (
    tuple int int)))
  (def-function robot.instance (:params (?r robot)) (:result symbol))
  (def-state-function robot.battery (:params (?r robot)) (:result float))

```

```

(def-state-function robot.velocity (:params (?r robot)) (:result (tuple
  float float)))
(def-state-function robot.rotation_speed (:params (?r robot)) (:result
  float))
(def-state-function robot.in_station (:params (?r robot)) (:result
  boolean))
(def-state-function robot.in_interact_areas (:params (?r robot)) (:
  result (list interact_area)))
(def-function robot.default_velocity (:params (?r robot)) (:result int)
  )
(def-function robot.drain_rate (:params (?r robot)) (:result float))
(def-function robot.recharge_rate (:params (?r robot)) (:result float))
(def-state-function robot.location (:params (?r robot)) (:result
  location))

;machine state functions
(def-function machine.instance (:params (?m machine)) (:result symbol)
  )
(def-function machine.coordinates (:params (?m machine)) (:result (
  tuple int int)))
;(def-function machine.coordinates_tile (:params (?m machine)) (:result
  (tuple int int)))
(def-function machine.input_belt (:params (?m machine)) (:result belt))
(def-function machine.output_belt (:params (?m machine)) (:result belt)
  )
(def-function machine.processes_list (:params (?m machine)) (:result (
  list int)))
(def-function machine.type (:params (?m machine)) (:result symbol))
(def-state-function machine.progress_rate (:params (?m machine)) (:
  result float))
(def-state-function machine.package_processed (:params (?m machine)) (:
  result package))

;package state function
(def-function package.instance (:params (?p package)) (:result symbol))
(def-state-function package.location (:params (?p package)) (:result
  location))
(def-state-function package.processes_list (:params (?p package)) (:
  result (list (tuple int float))))
(def-function package.all_processes (:params (?p package)) (:result (
  list (tuple int float))))
(def-state-function package.closest_interact_area (:params (?p package)
  ) (:result interact_area))

;belt state function
(def-function belt.instance (:params (?b belt)) (:result symbol))
(def-function belt.belt_type (:params (?b belt)) (:result symbol))
(def-function belt.polygons (:params (?b belt)) (:result (list (tuple
  int int))))
(def-function belt.cells (:params (?b belt)) (:result (list (tuple int
  int))))
(def-function belt.interact_areas (:params (?b belt)) (:result (list
  interact_area)))

```

```

(def-state-function belt.packages_list (:params (?b belt)) (:result (
  list package)))

;parking_area state functions
(def-function parking_area.instance (:params (?pa parking_area)) (:
  result symbol))
(def-function parking_area.polygons (:params (?pa parking_area)) (:
  result (list (tuple int int))))
(def-function parking_area.cells (:params (?pa parking_area)) (:result
  (list (tuple int int))))

;interact_areas state functions
(def-function interact_area.instance (:params (?pa parking_area)) (:
  result symbol))
(def-function interact_area.polygons (:params (?pa parking_area)) (:
  result (list (tuple int int))))
(def-function interact_area.cells (:params (?pa parking_area)) (:result
  (list (tuple int int))))
(def-function interact_area.belt (:params (?pa parking_area)) (:result
  belt))

;command definition
(def-command process (:params (?m machine) (?p package)))
(def-command pick (:params (?r robot)))
(def-command pick_package (:params (?r robot) (?p package)))
(def-command place (:params (?r robot)))
(def-command do_move (:params (?r robot) (?a float) (?s float) (?d
  float)))
(def-command navigate_to (:params (?r robot) (?x float) (?y float)))
(def-command navigate_to_cell (:params (?r robot) (?cx int) (?cy int)))
(def-command navigate_to_area (:params (?r robot) (?area object)))
(def-command go_charge (:params (?r robot)))
(def-command-om-model go_charge
  (:params (?r robot))
  (:body (sleep 15)))

(def-command do_rotation (:params (?r robot) (?a float) (?w float)))
(def-command face_belt (:params (?r robot) (?b belt) (?w float)))
(def-command rotate_to (:params (?r robot) (?a float) (?w float)))

;Lambdas definition
(def-lambda go_random (lambda (?r ?l ?u)
  (let ((x (rand-int-in-range ?l ?u))
        (y (rand-int-in-range ?l ?u)))
    (navigate_to ?r x y))))

(def-lambda
  find_machines_for_process
  (lambda (?process)
    (begin
      (define __process__
        (lambda (?p seq)
          (if (null? seq)
              nil

```

```

        (if (contains (machine.processes_list (car
        seq)) ?p)
            (cons (car seq) (__process__ ?p (cdr
            seq)))
            (__process__ ?p (cdr seq))))))
    (define machines (instances machine))
    (define result (__process__ ?process machines))
    result)))

(def-lambda available_robots
  (lambda nil
    (begin
      (define __l_available_robots__
        (lambda (l)
          (if (null? l)
              nil
              (if (not (locked? (car l)))
                  (cons (car l) (__l_available_robots__ (cdr
                  l)))
                  (__l_available_robots__ (cdr l)))))))
      (__l_available_robots__ (instances robot))))))

(def-lambda find_output_machine
  (lambda nil
    (begin
      (define __lambda__
        (lambda (seq)
          (if (null? seq)
              nil
              (if (= (machine.type (car seq)) output_machine)
                  (car seq)
                  (__lambda__ (cdr seq))))))
      (__lambda__ (instances machine))))))

(def-lambda take_first
  (lambda (seq)
    (if (null? seq)
        nil
        (cons (caar seq) (take_first (cdr seq))))))
)

```

Listing B.8: gobot_sim_om.scm

```

(begin
  (def-task t_position_robot_to_belt (:params (?r robot) (?b belt)))
  (def-method m_position_robot_to_belt (:task t_position_robot_to_belt)
    (:params (?r robot) (?b belt))
    (:pre-conditions true)
    (:score 0)
    (:body (do

```

```

        (navigate_to_area ?r (car (belt.interact_areas ?b)))
        (face_belt ?r ?b 5)))

(def-task t_carry_to_machine (:params (?r robot) (?p package) (?m
machine)))
(def-task-om-model t_carry_to_machine
  (:params (?r robot) (?p package) (?m machine))
  (:body (sleep 15)))

(def-method m_carry_to_machine
  (:task t_carry_to_machine)
  (:params (?r robot) (?p package) (?m machine))
  (:pre-conditions true)
  (:score 0)
  (:body
    (do
      (t_take_package ?r ?p)
      (t_deliver_package ?r ?m))))

(def-task t_take_package (:params (?r robot) (?p package)))
(def-method m_take_package (:task t_take_package)
  (:params (?r robot) (?p package))
  (:pre-conditions true)
  (:score 0)
  (:body (do
    (t_position_robot_to_belt ?r (package.location ?p))
    (pick_package ?r ?p))))

(def-task t_deliver_package (:params (?r robot) (?m machine)))
(def-method m_deliver_package (:task t_deliver_package)
  (:params (?r robot) (?m machine))
  (:pre-conditions true)
  (:score 0)
  (:body
    (let ((?b (machine.input_belt ?m)))
      (do
        (t_position_robot_to_belt ?r ?b)
        (wait-for '((< (len (belt.packages_list ,?b)) (len (
          belt.cells ,?b))))
        (place ?r))))))

(def-task t_check_battery (:params (?r robot)))
(def-task-om-model t_check_battery
  (:params (?r robot))
  (:body nil))
(def-method m_check_battery
  (:task t_check_battery)
  (:params (?r robot))
  (:pre-conditions true)
  (:score 0)
  (:body
    (do
      (wait-for '((< (robot.battery ,?r) 0.4))
      (charge_robot ?r))

```

```

        (t_check_battery ?r))))

(def-task charge_robot (:params (?r robot)))
(def-task-om-model charge_robot (:params (?r robot)) (:body nil))
(def-method m_charge_robot
  (:task charge_robot)
  (:params (?r robot))
  (:body
   (do
    (define h (acquire ?r '(:priority 1000)))
    (go_charge ?r)
    (wait-for '(> (robot.battery ,?r) 0.9))
    (release h)
   )))

(def-task t_check_rob_bat)
(def-task-om-model t_check_rob_bat
  (:params )
  (:body nil))

(def-method m_check_initial_robots_batteries
  (:task t_check_rob_bat)
  (:params)
  (:pre-conditions true)
  (:score 0)
  (:body
   (do
    (define tasks (mapf (lambda (?r) '(t_check_battery ,?r)) (
      instances robot)))
    (define h (apply par tasks))
    (print "end check batteries")
   )))

(def-task t_process_packages)
(def-method m_process_initial_packages
  (:task t_process_packages)
  (:params)
  (:pre-conditions true)
  (:score 0)
  (:body
   (do

    (define list_packages (instances package))
    (define list-h (mapf (lambda (?p) (async (t_process_package
      ?p))) list_packages))
    (mapf await list-h)
   )))

(def-task t_process_package (:params (?p package)))
(def-method m_process_package
  (:task t_process_package)
  (:params (?p package))
  (:body

```

```

      (do
        (define tasks
          (mapf
            (lambda (process)
              '(t_process_on_machine ,?p
                (arbitrary ',(find_machines_for_process (car
                  process)))
                ,(cadr process)
              ))
            (package.all_processes ?p)))
          (apply seq tasks)
          (t_output_package ?p))))

    (def-task t_process (:params (?m machine) (?p package) (?d int)))
    (def-task-om-model t_process
      (:params (?m machine) (?p package) (?d int))
      (:body
        (sleep ?d)))

    (def-method m_process
      (:task t_process)
      (:params (?m machine) (?p package) (?d int))
      (:body
        (do
          (process ?m ?p)
          (wait-for '(!= (package.location ,?p) ,?m))))))

    (def-event on_new_package
      (:params (?p package))
      (:trigger (once))
      (:body
        (do
          (wait-for '(instance (package.location ,?p) belt))
          (t_process_package ?p))))

    (def-event on_battery_low
      (:params (?r robot))
      (:trigger (whenever (< (robot.battery ?r) 0.4)))
      (:body (charge_robot ?r)))

  )

```

Listing B.9: gobot_sim_random.scm

```

(begin
  (load (concatenate (get-env-var "OMPAS_PATH") "
    /ompas-gobot-sim/domain/om.scm"))

  (def-task t_process_on_machine (:params (?p package) (?m machine) (?d
    int)))
  (def-method m_process_on_machine
    (:task t_process_on_machine)

```

```

(:params (?p package) (?m machine) (?d int))
(:pre-conditions true)
(:score 0)
(:body
  (do
    (define ?r (arbitrary (instances robot) rand-element))
    (define h1 (acquire ?m))
    (define h2 (acquire ?r))
    (t_carry_to_machine ?r ?p ?m)
    (release h2)
    (t_process ?m ?p ?d)
  )))

(def-task t_output_package (:params (?p package)))
(def-method m_output_package
  (:task t_output_package)
  (:params (?p package))
  (:body
    (do
      (define ?r (arbitrary (instances robot)))
      (define h_r (acquire ?r))
      (define om (find_output_machine))
      (t_carry_to_machine ?r ?p om)
      (release h_r))))
)

```

Listing B.10: gobot_sim_fa.scm

```

(begin
  (load (concatenate (get-env-var "OMPAS_PATH") "
    /ompas-gobot-sim/domain/om.scm"))

  (def-task t_process_on_machine (:params (?p package) (?m machine) (?d
    int)))
  (def-method m_process_on_machine
    (:task t_process_on_machine)
    (:params (?p package) (?m machine) (?d int))
    (:body
      (do
        (define h_m (acquire ?m))
        (define h_r (acquire-in-list (instances robot)))
        (define ?r (first h_r))
        (t_carry_to_machine ?r ?p ?m)
        (release (second h_r))
        (t_process ?m ?p ?d))))

  (def-task t_output_package (:params (?p package)))
  (def-method m_output_package
    (:task t_output_package)
    (:params (?p package))
    (:body

```



```

      (do
        (define h_r (acquire-in-list (instances robot)))
        (define ?r (first h_r))
        (define om (find_output_machine))
        (t_carry_to_machine ?r ?p om)
        (release (second h_r))))
    )

```

Listing B.11: gobot_sim_falrptf.scm

```

(begin
  (load (concatenate (get-env-var "OMPAS_PATH") "
    /ompas-gobot-sim/domain/om.scm"))

  (def-lambda
    remaining-time (lambda (?p)
      (eval (cons '+ (cadr (unzip (package.processes_list ?p)))))))

  (def-task t_process_on_machine (:params (?p package) (?m machine) (?d
    int)))
  (def-method m_process_on_machine
    (:task t_process_on_machine)
    (:params (?p package) (?m machine) (?d int))
    (:pre-conditions true)
    (:score 0)
    (:body
      (do
        (define r-time (remaining-time ?p))
        (define h_m (acquire ?m '(:priority ,r-time)))
        (define h_r (acquire-in-list (instances robot) '(:priority
          ,r-time)))
        (define ?r (first h_r))
        (t_carry_to_machine ?r ?p ?m)
        (release (second h_r))
        (t_process ?m ?p ?d))))

  (def-task t_output_package (:params (?p package)))
  (def-method m_output_package
    (:task t_output_package)
    (:params (?p package))
    (:body
      (do
        (define h_r (acquire-in-list (instances robot)))
        (define ?r (first h_r))
        (define om (find_output_machine))
        (t_carry_to_machine ?r ?p om)
        (release (second h_r))))))
)

```

Long résumé en français

Chapitre 1: Introduction

Les récents développements technologiques dans le domaine de la robotique et de l'AI pourraient permettre l'utilisation de robots dans de nombreux domaines de notre vie. Les applications vont de l'industrie du future, qui vise à optimiser divers processus à l'aide de flottes d'agents autonomes, aux opérations de recherche et de sauvetage, sans oublier les robots d'assistance personnelle. L'une des limites à l'intégration des robots est leur capacité à effectuer des missions de manière autonome. Ici, nous considérons que l'autonomie d'un agent robotique repose sur trois piliers :

- sa capacité à percevoir le monde,
- sa capacité à agir sur le monde,
- sa capacité à délibérer pour effectuer des missions en fonction de ses capacités de raisonnement.

En particulier, nous nous intéressons au processus de délibération dans un système robotique. Ici nous nous concentrons sur sa capacité à planifier des tâches, i.e. à générer une séquence d'actions pour atteindre un but, et à agir en fonction du contexte d'exécution. Cette dernière capacité est ici nommée l'*Acting*. Dans sa plus simple expression, l'*Acting* ne prend en compte que le contexte courant d'exécution pour adapter son comportement, ce qui ne lui permet pas d'anticiper les effets à long terme de ses choix.

Une manière de pallier à ces limitations est d'interfacer le système d'*Acting* avec un planificateur pour guider sa délibération grâce à ses capacités d'anticipation. Or la littérature propose surtout des approches dans lesquelles les plans générés sont avant tout exécutés par le système d'*Acting*, et rarement pour guider la délibération à un plus bas niveau. Une des limites à une intégration plus poussée entre la planification et l'*Acting* est notamment l'utilisation de différents modèles pour planifier et agir, ce qui rend difficile l'interaction entre les deux systèmes.

Récemment, l'approche proposée par RAE est de définir un système d'*Acting* dans lequel le modèle opérationnel hiérarchique peut être directement utilisé pour planifier afin d'anticiper les futurs choix du système. Ici, nous proposons d'étendre cette approche en ajoutant de nouvelles capacités de délibération à RAE, et en automatisant la génération de modèles de planification dans un formalisme qui permet l'utilisation de techniques de planification plus poussées que ce qui est proposé pour les premières versions de RAE, mais qui nécessitent des modèles formels.

L'ensemble de ces ajouts s'est matérialisé dans OMPAS, un nouveau système d'*Acting* qui au-delà des extensions apportés à RAE propose un langage de programmation dédié qui permet notamment la concurrence dans les programmes exécuter, et dans lequel les choix de délibération sont explicites.

Ainsi, cette thèse est dédiée à la présentation de OMPAS dans les trois premiers chapitres, ainsi qu'à son évaluation sur différents domaines robotiques.

Chapitre 2: Le système OMPAS (Operational Model Planning and Acting System): Architecture et Algorithmes

Dans ce chapitre, nous présentons le Operational Model Planning and Acting System (OMPAS), un nouveau système d'*Acting* basé sur le système Refinement Acting Engine (RAE) (Ghallab, Nau, and Traverso 2016). RAE est un système d'*acting* basé sur le raffinement des tâches. Le système reçoit les missions sous forme de tâches à exécuter, qui sont raffinées en sélectionnant une méthode applicable dans le contexte courant d'exécution. Une méthode est un programme qui modélise une manière d'effectuer

une tâche. Une méthode peut être composée de tâches de plus bas niveau, ou de commandes qui seront directement exécutables sur la plateforme robotique. Le but de RAE est donc de raffiner successivement toutes les tâches jusqu'à arriver à une succession de commandes qui sont exécutables par la plateforme robotique.

Les commandes, tâches et méthodes sont définies dans ce qu'on appelle un modèle opérationnel hiérarchique, dans lequel les capacités de l'agent robotique sont décrites par un tuple (C, T, M) , où :

- C est l'ensemble des commandes primitives du système robotique, comme $move(?r_a, ?r_b)$,
- T est l'ensemble des tâches de haut niveau que peut effectuer l'agent robotique, comme $putTheTable()$,
- M est un ensemble de méthodes qui peuvent être utilisés par RAE pour raffiner une tâche de T . Une méthode ne peut raffiner qu'une seule tâche.

À partir d'un modèle opérationnel hiérarchique fourni par un programmeur, RAE est capable d'exécuter et superviser des tâches en parallèle.

Or, une des limites de RAE est sa capacité à gérer des tâches concurrentes, qui ont besoin d'accéder aux mêmes ressources, par exemple le bras du robot pour manipuler et transporter différents objets. Avec ses algorithmes de base, RAE ne peut pas garantir que l'exécution de plusieurs tâches concurrentes ne viendront pas interférer les unes avec les autres. Si l'on regarde les extensions proposées pour améliorer la délibération de RAE, le planificateur *UPOM* (Patra, Mason, Ghallab, et al. 2021) permet d'améliorer le raffinement de tâches en simulant l'exécution des méthodes et en sélectionnant la méthode avec la plus grande efficacité, mais il ne permet pas d'améliorer l'entrelacement de plusieurs tâches. En effet, *UPOM* ne prend en compte que le raffinement d'une unique tâche, sans prendre en compte le contexte d'exécution des autres tâches.

Au vu des limites de RAE, nous proposons OMPAS, un nouveau système d'*Acting* basé sur les fonctions de délibération de RAE. Notamment, nous améliorons la capacité de RAE à gérer la concurrence avec :

- un nouvel algorithme général qui crée un thread pour gérer l'exécution de chaque tâche de haut niveau indépendamment, à la différence de RAE qui gérait cela dans une boucle à la *round robin*,
- la gestion native de la concurrence dans le programme des méthodes,
- un système dédié à l'allocation de ressources entre plusieurs tâches concurrentes.

Dans OMPAS, la délibération est gérée par plusieurs *managers* :

- Le *Platform Manager* est l'interface entre OMPAS et la *plateforme robotique*. Notamment, il gère l'envoi de requêtes d'exécution de commandes et la supervision de leur exécution. C'est aussi lui qui mutualise la réception des informations de capteurs et autres venant des couches plus basses dans l'architecture robotique.
- Le *State Manager* est le système qui gère la base de connaissances sur l'état du monde connu comme tel par OMPAS. Il gère les accès asynchrones de lecture et d'écriture sur l'état du monde. Il propose aussi des fonctions avancées qui permettent des réactions d'OMPAS basées sur des événements qui sont fonctions de l'état du monde.
- Le *Execution Manager* est responsable de l'exécution des tâches et programmes dans OMPAS. En fonction des programmes robotiques, il va faire le lien avec les autres *managers* pour profiter de leurs fonctions de délibération.
- Le *Resource Manager* est responsable de la sûreté d'accès aux ressources concurrentes. En fonction de sa stratégie d'allocation, il allouera les ressources pour à la fois éviter des *deadlocks*, mais aussi pour améliorer les performances du système de manière opportuniste.
- Le *Acting Manager* est responsable du raffinement des tâches et de l'instanciation des paramètres des méthodes. Il tient à jour une base de connaissances représentant la trace d'exécution et de délibération d'OMPAS sous la forme d'un *Acting Tree*.
- Le *PLanner Manager* fait le lien entre OMPAS et un système de planification pour guider la délibération de OMPAS. Il se sert de l'état du système pour anticiper l'exécution des programmes dans OMPAS, et informer OMPAS de la meilleure méthode pour raffiner une tâche, des meilleurs paramètres et de l'ordre d'allocation des ressources.

Avec cette architecture, OMPAS est capable de gérer l'exécution de plusieurs tâches en parallèle avec des ressources limitées et est ainsi bien équipé pour être intégré dans des architectures multi-robots.

Chapitre 3: Le Langage d'Acting de OMPAS: Syntaxe, Sémantique et Évaluation

Les systèmes d'Acting basés sur le raffinement utilisent des modèles opérationnels qui sont essentiellement des programmes exécutables. Ainsi, un système donné n'est pas rattaché à un langage de programmation en particulier. Dans ce chapitre, nous proposons un nouveau langage d'Acting qui permette de profiter des capacités de délibération d'OMPAS. Ce langage sera utilisé pour définir le programme des méthodes. Le langage nommé Scheme OMPAS est un dialecte Lisp, basé sur sa variante Scheme (Moretti 1979). Ici, nous proposons un nouveau langage, dont le coeur a été simplifié au maximum pour permettre par la suite une analyse automatique des programmes pour en extraire des modèles formels. Le langage SOMPAS profite d'un interpréteur dédié, ce qui nous permet de faire les ajouts suivants dans le langage:

- Nous ajoutons dans Scheme des primitives pour exécuter des fonctions de manière concurrente,
- Nous ajoutons des primitives inhérentes au contrôle de robots, comme des requêtes pour exécuter des commandes ou tâches, et lire l'état du monde. De plus nous avons des primitives spécifiques pour déclarer de nouvelles ressources et pour les acquérir.

Nous utilisons ce même langage pour fournir une interface à un utilisateur extérieur via un *REPL*. Via ces interfaces l'utilisateur peut fournir le modèle opérationnel hiérarchique que pourra utiliser le système pour exécuter des tâches, ainsi que des fonctions pour superviser le fonctionnement interne de OMPAS.

Chapitre 4: Planification à partir de Modèles Opérationnels pour guider la Délibération de OMPAS

Dans ce chapitre, nous présentons une manière de guider la délibération de OMPAS en utilisant des techniques de planification. Notamment, nous utilisons le planificateur hiérarchique et temporel *Aries* pour anticiper le raffinement des tâches, l'instantiation des paramètres et l'allocation des ressources. Pour ce faire, nous proposons deux choses.

Premièrement, nous proposons de générer automatiquement les modèles de planification à partir des programmes des méthodes. Nous définissons une procédure de synthèse en deux étapes :

- Dans un premier temps, le programme est transformé en une représentation intermédiaire, ici un *flow graph*. Dans cette représentation, tout le sucre syntaxique est enlevé et le nombre de primitive est limité.
- Dans un second temps, le *flow graph* est converti en une chronique, un formalisme de planification dont pourra se servir *Aries* pour planifier.

Deuxièmement, nous proposons d'utiliser *Aries* à différents niveaux de la délibération. De la même manière que *UPOM* (Patra, Mason, Ghallab, et al. 2021), nous proposons de guider la sélection d'une méthode pour raffiner une tâche en utilisant de la planification dite locale. Néanmoins, nous rencontrons les mêmes limites qu'en utilisant *UPOM*, c'est à dire la planification ne prend pas en compte le contexte d'exécution des autres tâches.

Pour y remédier, nous proposons de planifier de manière continue l'exécution globale de OMPAS. Le planificateur cherche en continu une nouvelle solution en prenant en compte toutes les tâches couramment exécutées. Ainsi, le planificateur peut trouver des solutions en avance de phase qui seront ensuite stockées dans l'*Acting Tree*. Les solutions anticipées pourront ensuite être utilisées lorsque OMPAS en aura besoin. Ce même *Acting Tree* est aussi utilisé pour simplifier l'instanciation du problème de planification. En effet, vu qu'il représente l'exécution globale du système, il sert à définir l'état initial du système, auquel on ajoute les informations pourvues par le *State Manager* et le *Resource Manager* en ce qui concerne l'état du monde et l'état d'accès aux ressources.

De cette manière, la planification anticipe l'exécution d'OMPAS, notamment pour éviter des *dead-locks*, mais aussi pour optimiser le temps total pour gérer toutes les tâches.

Chapitre 5: Évaluation de OMPAS: Domaines Simulés et Intégration avec une Plateforme Robotique

Dans ce chapitre, nous fournissons une évaluation des capacités de délibération d'OMPAS sur deux domaines robotiques:

- Le premier est le domaine *Gripper-Door* qui est utilisé tout au long de cette thèse pour illustrer les travaux présentés,
- Le second est *Gobot-Sim*, un simulateur d'usine dans laquelle une flotte de robot déplace des paquets entre des machines, chaque machine permettant d'appliquer un *process* sur le paquet.

Sur ces deux domaines, le but est d'effectuer toutes les tâches d'un problème le plus rapidement possible. Pour cela, nous définissons le *Score d'efficacité (ES)* qui est défini de la manière suivante :

$$ES = \frac{SR \times T}{t}$$

Ici, SR est le pourcentage de succès des tâches définies dans le problème, par exemple 75% si trois tâches sur quatre ont été exécutées avec succès, T est le temps maximum alloué pour gérer le problème, et t est le temps pris par le système pour terminer le problème.

Dans l'un ou l'autre des domaines, on voit que l'apport de la planification permet d'avoir le meilleur ES . Notamment, les stratégies qui profitent de *Aries* pour raffiner localement une tâche, et de la planification continu qui cherche un plan qui optimise le temps total est dans 90% des cas la meilleure stratégie, et ce, malgré le coût de la planification. Dans le reste des cas, c'est que le problème est si simple qu'il n'a pas besoin de planification.

Cette évaluation nous permet de voir que la planification permet d'améliorer la qualité des choix faits par OMPAS. Néanmoins la campagne d'évaluation proposée ici ne permet par forcément de montrer toutes les capacités d'OMPAS, notamment dans sa gestion des erreurs. De plus, les domaines évalués ont été créés spécifiquement pour OMPAS, et il manque donc de comparatifs avec d'autres systèmes.

Conclusion

En conclusion, nous proposons ici un nouveau système d'*Acting* complet. La création de ce système nous a permis de repousser les limites de l'intégration de la planification dans un système d'exécution. Quelques limites doivent tout de même être soulevées comme la replanification qui ne prend pas en compte les solutions précédemment trouvées ou l'explosion de la complexité qui est fonction du nombre de tâches en parallèle qui doivent être planifiées en même temps. Au delà de ces limitations, nous pensons que OMPAS est le candidat parfait pour continuer la recherche concernant l'interaction entre les systèmes d'*Acting* et de planification, notamment en ajoutant des capacités d'apprentissage pour améliorer les heuristiques d'OMPAS.

Acronyms

- A_{Δ} Acting Domain. 7, 30, 34, 36, 85, 86, 87, 90, 93
- P_{Δ} Planning Domain. 5
- P_{Π} planning problem. 5, 55, 56, 103, 106, 134, 135, 139, 140
- AI** Artificial Intelligence. 1, 19, 68, 97, 215
- AM** Acting Manager. i, 45, 46, 53, 55, 56, 83, 86, 87, 129, 134, 135, 136, 138, 139, 147
- AML** ASPEN Modelling Language. 99
- ANML** Action Notation Modeling Language. 99, 107
- AP** Acting Process. iii, 129, 130, 131, 132, 133, 134, 135, 139
- APB** Acting Process Binding. 134, 138
- APSI** Advanced Planning and Scheduling Initiative. 13, 99
- ASPEN** Automated Scheduling and Planning ENvironment. 17
- AUV** Autonomous Underwater Vehicle. 13
- AV** Acting Variable. 129, 130, 134, 138, 139
- BDI** Belief-Desire-Intention. 66
- BT** Behavior Tree. 19
- CASPER** Continuous Activity Scheduling, Planning, Execution, and Replanning. 17
- CPEF** Continuous Planning and Execution Framework. 17
- CRAM** Cognitive Robot Abstract Machine. 85
- CSP** Constraint Satisfaction Problem. 13, 16, 18, 24, 99, 101, 103, 138
- CX** CLIPS Executive. 21
- eBT** extended Behavior Tree. 19
- EM** Execution Manager. i, 45, 46, 47, 48, 49, 52, 53, 54, 55, 81, 86, 87, 90, 129, 130, 133, 134, 138, 139, 144
- ENSHP** Expressive Numeric Heuristic Planner. 5
- ESA** European Space Agency. 99
- ESL** Execution Support Language. 12
- EXEC** Smart Executive. 12
- FAPE** Flexible Acting and Planning Environment. 6, 14, 99, 100, 103
- FSA** Finite State Automata. 20
- FSM** Finite State Machine. 19, 62, 63, 64
- GDP** Goal Decomposition Planner. 100
- HATP** Human Aware Task Planner. 100
- HATP/EHDA** Human-Aware Task Planner with Emulation of Human Decisions and Actions. 100
- HBF** Hybrid Backward-Forward. 19

- HCA** Hierarchical Constraint Automata. 63
- HCP** Hierarchical Conformance Planning. 170
- HDDL** Hierarchical Domain Definition Language. 100
- HGN** Hierarchical Goal Network. 100
- HRC** Human Robot Collaboration. 14, 99
- HRI** Human Robot Interaction. 100
- HTEP** Hierarchical Temporal Event Planner. 101, 103
- HTN** Hierarchical Task Network. i, 19, 32, 34, 38, 53, 68, 100, 129
- ICAPS** International Conference on Automated Planning and Scheduling. 25
- ICTAI** International Conference on Tools for Artificial Intelligence. 25
- IDEA** Intelligent Distributed Execution Architecture. 13, 18
- IntEx** Integrated Acting, Planning and Execution. 25
- IPC** International Planning Competition. 25, 100
- IPEM** Integrated Planning, Execution and Monitoring. 16
- IR** Intermediate Representation. 108
- JSSP** Job Shop Scheduling Problem. 155, 156, 160
- KA** Knowledge Area. 65
- LCP** Lifted Constraint Planner. 5, 99, 101, 103
- MCTS** Monte Carlo Tree Search. 21, 101, 107, 146, 147
- MIR** Mode Identification and Reconfiguration. 12
- MM** Mission Manager. 12
- NASA** National Aeronautics and Space Administration. 99
- NDDL** New Domain Definition Language. 99
- OMPAS** Operational Model Planning and Acting System. i, ii, iii, v, 24, 25, 28, 29, 30, 44, 45, 46, 47, 48, 49, 50, 51, 53, 54, 55, 56, 57, 60, 61, 75, 77, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 90, 91, 93, 94, 96, 102, 107, 109, 129, 130, 131, 134, 135, 139, 140, 143, 144, 145, 146, 147, 148, 149, 150, 152, 153, 154, 155, 160, 161, 163, 165, 166, 167, 169, 170, 171, 215, 216
- PA** Point Algebra. 126, 198
- PANDA** Planning and Acting in a Network Decomposition Architecture. 100
- PDDL** Planning Domain Description Language. 6, 15, 20, 21, 23, 97, 99, 100, 107, 127, 166
- PLEXIL** PPlan EXecution Interchange Language. 63
- PLM** PPlaner Manager. i, 45, 46, 55, 56, 57, 96, 129, 133, 134, 135, 136, 138, 139, 147
- PM** Platform Manager. i, 44, 45, 46, 47, 48, 49, 57, 81
- POCL** Partial Order Causal Link. 99, 101
- PROPEL** PROcedure Planning and Execution Language. 16, 62, 65, 68, 169
- PRS** Procedural Reasoning System. 11, 12, 15, 16, 17, 30, 65, 68, 85, 101, 169
- PS** Planner/Scheduler. 12

- R2C** Requests and Resources Checker. 11
- RAE** Refinement Acting Engine. i, v, 2, 21, 22, 23, 24, 25, 29, 30, 31, 32, 34, 35, 36, 37, 38, 39, 40, 41, 43, 44, 45, 46, 48, 49, 50, 54, 57, 66, 88, 101, 102, 107, 152, 166, 167, 169, 215
- RAP** Reactive Action Package. 11, 14, 65, 68, 169
- RAX** Remote Agent Experiment. 11, 12, 14, 18, 65
- RAX-PS** Remote Agent Experiment Planner/Scheduler. 99
- RM** Resource Manager. i, 45, 46, 51, 52, 56, 84, 86, 92, 135, 136, 139, 140
- RMPL** Reactive Model-based Programming Language. 18
- ROS** Robotic Operating System. 15, 45
- RPL** Reactive Plan Language. 15
- RS** RobotSkill. 20, 63
-
- SHOP** Simple Hierarchical Ordered Planner. 6, 100
- SHOP2** Simple Hierarchical Ordered Planner 2. 100
- SM** State Manager. i, 45, 46, 47, 48, 56, 57, 80, 82, 86, 87, 90, 91, 92, 134, 135, 140
- SOMPAS** Scheme OMPAS. ii, 24, 28, 60, 76, 77, 79, 80, 81, 83, 84, 85, 94, 107, 108, 109, 110, 112, 113, 118, 122, 127, 143, 148, 161, 166, 169, 170, 185, 189, 192, 195
- SPA** Sense-Plan-Act. 8, 9
- SSA** Single Static Assignment. 109, 110
- SSP** Stochastic Shortest Path. 6
- STN** Simple Temporal Network. 18, 99, 100
- STNU** Simple Temporal Network with Uncertainty. 18
- STRIPS** STanford Research Institute Problem Solver. 5, 6, 14, 19, 97, 104
-
- TN** Temporal Network. 126, 198
- TPN** Temporal Plan Network. 18
- TPNU** Temporal Plan Network with Uncertainty. 18
- T-REX** Teleo-Reactive EXecutive. 13, 18, 107
- TCA** Task Control Architecture. 10, 15
-
- V&V** Verification and Validation. 8, 11

Titre : Planification à partir de modèles opérationnels pour l'action délibérée en robotique

Mots clés : IA, Robotique, Action délibérée, Planification en continu, Raffinement de tâche, Modèle opérationnel

Résumé : Les récents développements technologiques dans le domaine de la robotique et de l'intelligence artificielle pourraient permettre l'utilisation de robots dans de nombreux domaines de notre vie. Les applications vont de l'industrie 4.0, qui vise à optimiser divers processus à l'aide de flottes d'agents autonomes, aux opérations de recherche et de sauvetage, sans oublier les robots d'assistance personnelle. À mesure que la complexité des plateformes robotiques augmente, les algorithmes de délibération doivent être améliorés, notamment pour gérer un nombre croissant d'agents, pour gérer des objectifs et des tâches complexes, et pour évoluer dans des environnements plus ouverts où les événements imprévus doivent être traités de manière autonome. Le niveau d'autonomie d'un agent dépend de cinq grandes fonctions de délibération : la planification, l'action délibérée, la surveillance, l'apprentissage et l'observation. Nous nous concentrons ici sur la fonction de planification, qui indique à l'agent ce qu'il doit faire pour accomplir ses missions, et sur la fonction d'action délibérée, qui adapte le comportement de l'agent au contexte d'exécution, ce qui le rend plus robuste face aux imprévus et aux aléas. Nous étudions en particulier l'interaction entre la planification et l'action délibérée. Bien qu'elles soient presque toujours utilisées ensemble, les approches dans la littérature ont tendance à les considérer séparément, ce qui limite leur interaction. Dans cette thèse, nous proposons une approche unifiée de la planification et de l'action, dans laquelle les deux systèmes sont en symbiose pour améliorer leurs performances respectives. Nous présentons le système OMPAS (Operational Model Acting and Planning System), un moteur d'action basé sur le raffinement qui exécute plusieurs tâches de haut niveau en parallèle en les raffinant en un ensemble de tâches et de commandes de niveau inférieur. OMPAS utilise un dialecte Lisp personnalisé (SOMPAS) pour définir le comportement de l'agent robotique. SOMPAS fournit des primitives pour gérer la concurrence et les ressources et, grâce au langage de base restreint et à l'identification explicite des décisions d'action, permet la synthèse automatique des modèles de planification. Le moteur tire parti d'un planificateur temporel et hiérarchique qui utilise les modèles synthétisés pour anticiper et guider les décisions du système agissant. Le planificateur est utilisé de manière continue, c'est-à-dire qu'il planifie en même temps que l'exécution des tâches et s'adapte toujours à l'état actuel de l'exécution pour améliorer les décisions attendues. Ces décisions éclairées devraient permettre d'éviter les blocages et d'optimiser de manière opportuniste l'achèvement de multiples tâches parallèles. Nous fournissons une évaluation de l'approche globale sur plusieurs domaines de la robotique. En particulier, OMPAS a été utilisé pour contrôler une flotte de robots dans une plateforme logistique simulée. Les résultats ont montré la capacité du système à gérer plusieurs tâches simultanées grâce à son système de gestion des ressources dédié. En outre, la planification continue améliore le temps total nécessaire à l'accomplissement de toutes les tâches d'une mission. La planification étant intégrée au cœur du système, aucun effort supplémentaire n'est requis de la part du programmeur du robot pour tirer parti de cette fonctionnalité.

Title: Planning from operational models for deliberate action in robotics

Key words: AI, Robotics, Deliberate acting, Continuous planning, Task refinement, Operational model

Abstract: Recent technological developments in the field of robotics and artificial intelligence could enable the use of robots in many areas of our lives. Applications range from Industry 4.0, which aims to optimize various processes using fleets of autonomous agents, to search and rescue operations, without forgetting personal assistance robots. As the complexity of robotic platforms increases, deliberation algorithms need to be improved, in particular to handle an increasing number of agents, to manage complex goals and tasks, and to evolve in more open environments where unforeseen events should be dealt with autonomously. The level of autonomy of an agent depends on five major deliberation functions: planning, deliberate acting, monitoring, learning, and observing. Here we focus on the planning function, which tells the agent what to do to accomplish its missions, and deliberate acting, which adapts the agent's behavior to the context of execution, making it more robust to contingencies and hazards. In particular, we study the interaction between planning and deliberate acting. Although they are almost always used together, approaches in the literature tend to consider them separately, which limits their interaction. In this thesis, we propose a unified approach to planning and acting, in which both systems are in symbiosis to improve each other's performance. We present the Operational Model Acting and Planning System (OMPAS), a refinement based acting engine that executes multiple high-level tasks in parallel by refining them into a set of lower-level tasks and commands. OMPAS uses a custom Lisp dialect (SOMPAS) to define the behavior of the robotic agent. SOMPAS provides primitives for handling concurrency and resources, and, thanks to the restricted core language and the explicit identification of acting decisions, allows the automatic synthesis of planning models. The engine takes advantage of a temporal and hierarchical planner that uses the synthesized models to look ahead and guide the decisions of the acting system. The planner is used in a continuous fashion, i.e., it plans concurrently with the execution of tasks and always adapts to the current state of execution to improve the expected decisions. These informed decisions should avoid deadlocks and opportunistically optimize the completion of multiple parallel tasks. We provide an evaluation of the overall approach on several robotics domains. In particular, OMPAS was used to control a fleet of robots in a simulated logistics platform. The results showed the ability of the system to handle several concurrent tasks thanks to its dedicated resource management system. In addition, continuous planning improves the total time to complete all tasks of a mission. Since planning is integrated into the core of the framework, no additional effort is required from the robot programmer to take advantage of this feature.