Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

MutRoSe: A Goal-Oriented Framework for Mission Specification and Decomposition of Multi-Robot Systems

Eric Bernd Gil

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Orientadora
Prof.a Dr.a Genaína Nunes Rodrigues

Brasília
2021

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

MutRoSe: A Goal-Oriented Framework for Mission Specification and Decomposition of Multi-Robot Systems

Eric Bernd Gil

Dissertação apresentada como requisito parcial para
conclusão do Mestrado em Informática

Brasília, 15 de Outubro de 2021

Dedicatória

Dedico este trabalho aos meus pais, à minha irmã, a todos os meus amigos, que estiveram comigo ao longo dos anos, e à minha namorada Bárbara, que me incentivou a seguir em frente e foi meu porto seguro durante todos os momentos não só do mestrado mas também da graduação. Essa conquista não seria possível sem eles, que foram meu apoio e incentivo.

Agradecimentos

Agradeço primeiramente à minha orientadora Genaína, por estar sempre presente e disponível para me ajudar no que fosse necessário e por me motivar a entrar pós-graduação. Agradeço também à CAPES pelo apoio financeiro durante boa parte do período do mestrado, o qual foi fundamental para o andamento do trabalho.

Abstract

Robots are increasingly being inserted in new environments in order to perform tasks previously executed by humans. In this sense, the need for a way of modeling and specifying multi-robot behavior arises, even more when collaboration of multiple robots is necessary. This modeling framework needs to account for the dynamic nature of environments robots are inserted in, robots capabilities, task dependencies and other important characteristics which are inherent of this type of system. With this in mind, the system designer must be able to specify the behaviors and interactions that must exist, taking into consideration these important characteristics. In the robotics domain, requirements can be expressed in terms of a robotic mission specification, which defines the tasks to be executed by the robots in order to achieve the desired goals. Alongside the mission specification, a process that takes it as input and generates the possible task instances to be executed, as well as existing constraints between them, is needed. This is the case because these task instances, and perhaps their valid combinations, are the ones used by an allocation process in order to generate the best task allocations possible. The present work proposes a goal-oriented modeling framework which makes use of goal models in tandem with hierarchical task network (HTN) planning in order to address the such challenges. The choice of a goal-oriented approach is due to the fact that (i) the robotic mission goals can be expressed as first-class entities, and (ii) the decomposition of goals into sub-goals and tasks is performed in a hierarchical way, as is the case of the hierarchical task decomposition and planning process performed through the use of HTNs. This approach allows end-users, with little or no experience in the robotics design area, to specify robotic missions in a high-level fashion. In this sense, the end-user can work together with the system designers since he/she is the one that is familiar with the domain in which robots will work. Also, the approach enables mission specifications to take into account the real-world variability, to be reusable and to avoid the need of having knowledge about the robots that will compose the system. Finally, this work proposes a way of automatically decomposing the mission, if the specification is sound.

Keywords: Multi-Robot, Specification, Hierarchical Planning, Modeling

Resumo

Título: MutRoSe: Um Arcabouço Orientado a Objetivos para Especificação e Decomposição de Missões para Sistemas Multi-Robôs

Introdução

Este trabalho se dá no contexto em que há um aumento na complexidade no projeto de sistemas multi-robôs (SMR), dado que cada vez mais os robôs vem sendo inseridos em novos ambientes onde os mesmos devem realizar tarefas que anteriormente eram realizados por humanos. Nesse contexto, as abordagens atuais se tornam difíceis de manter uma vez que consistem em código procedural [1] ou carecem de abstrações de alto nível que possam ser aprendidas por usuários sem experiência, ou com um baixo nível de experiência, sem grandes dificuldades. Nesse sentido, surgem dois grandes objetivos da comunidade, definidos no *H2020 Robotics Multi-Annual Roadmap (MAR)*[1], que são: (i) a integreação de robôs com operadores humanos de maneira intuitiva e (ii) a existência de definições de missões altamente abstraídas e algoritmos para interação com usuários não treinados. O foco do presente trabalho se dá no contexto de termos definições de missões altamente abstraídas, com a proposta do *framework* MutRoSe (*Multi-Robot systems mission Specification and Decomposition*) para especificação e decomposição de missões em alto nível para SMR.

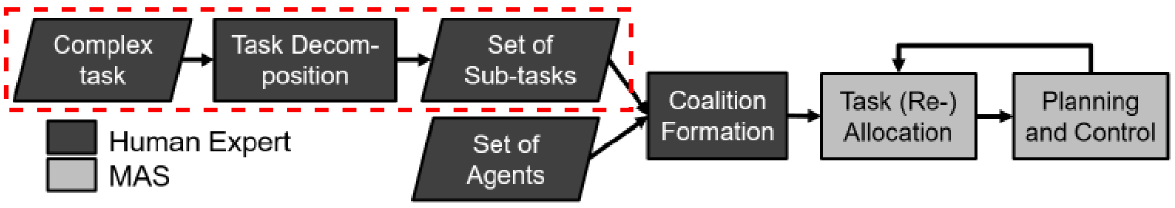


Figure 1: *Workflow* para o projeto de sistemas multi-robôs [2]

[1]https://www.eu-robotics.net/

Nesse contexto de projeto de SMR temos o *workflow* mostrado na Figura 1, onde dada uma missão (*Complex Task*) temos quatro principais blocos: (i) decomposição de tarefas (*Task Decomposition*), onde a missão é decomposta em um conjunto de sub-tarefas que devem ser executadas pelo sistema e que aqui chamamos de decomposição de missão, (ii) formação de coalisões (*Coalition Formation*), onde são definidos os times de robôs que vão executar as tarefas geradas no passo anterior, (iii) (re-)alocação de tarefas (*Task (Re-)Allocation*), onde as tarefas são alocadas para os times de robôs, e (iv) planejamento e controle, onde as tarefas são de fato executadas pelos robôs. Além disso, não existem implementações de sistemas completamente automatizados [2], *i.e.*, sistemas onde uma missão é dada para o SMR e o mesmo realiza os blocos subsequentes sem ajuda de humanos. Ademais, o maior nível de automação encontrado nos trabalhos existentes automatiza os três últimos blocos do *workflow*, que são a formação de coalisões, a (re-)alocação de tarefas e o planejamento e controle.

Dessa maneira, temos ainda que não existe uma abordagem de especificação de missões em alto nível focada em deliberação, o que dificulta a automação da etapa de decomposição de missões, dificultando também a automação por completo do *workflow*, e também dificulta a inserção do usuário final, que possui o conhecimento de domínio, no processo de projeto do SMR. Com isso, surge a primeira pergunta de pesquisa:

Pergunta de Pesquisa 1 (PQ1): Como especificar missões em alto nível para SMR que possam ser utilizadas para *reasoning* em abordagens deliberativas?

Em conjunto com uma abordagem de especificação de missões em alto nível, deve-se ter uma maneira de decompor a missão automaticamente, levando em conta as particularidades de SMRs. Com essa abordagem pretende-se gerar o conjunto de tarefas a serem executadas de maneira dinâmica levando em consideração o estado atual do mundo. Daí, surge a segunda pergunta de pesquisa:

Pergunta de Pesquisa 2 (PQ2): Como realizar a decomposição de missões de maneira automatizada levando em consideração restrições de ordem e execução?

Em suma, de maneira a responder essas perguntas de pesquisa, este trabalho propõe a criação de um *framework* que provê: (i) modelos em alto nível para especificações de missões para SMRs, os quais tornam mais fácil de inserir usuários finais no processo de projeto de SMRs, e (ii) uma maneira de automaticamente decompor essa especificação, gerando as possíveis instâncias de tarefas, as decomposições válidas da missão e as restrições entre as tarefas baseado no conhecimento do sistema a respeito do mundo.

Metodologia

Em uma visão geral do *framework* MutRoSe, ilustrada na Figura 2, podemos verificar que o mesmo consiste em duas etapas: (i) a especificação da missão, onde um time de pessoas, representado na figura pelo usuário final e um projetista SMR, especifica todos os modelos necessários, e (ii) a decomposição de missão, onde os modelos definidos na primeira etapa são utilizados em conjunto com o conhecimento do mundo para gerar as decomposições válidas da missão e as restrições da missão, ambas em termos das possíveis instâncias de tarefas.

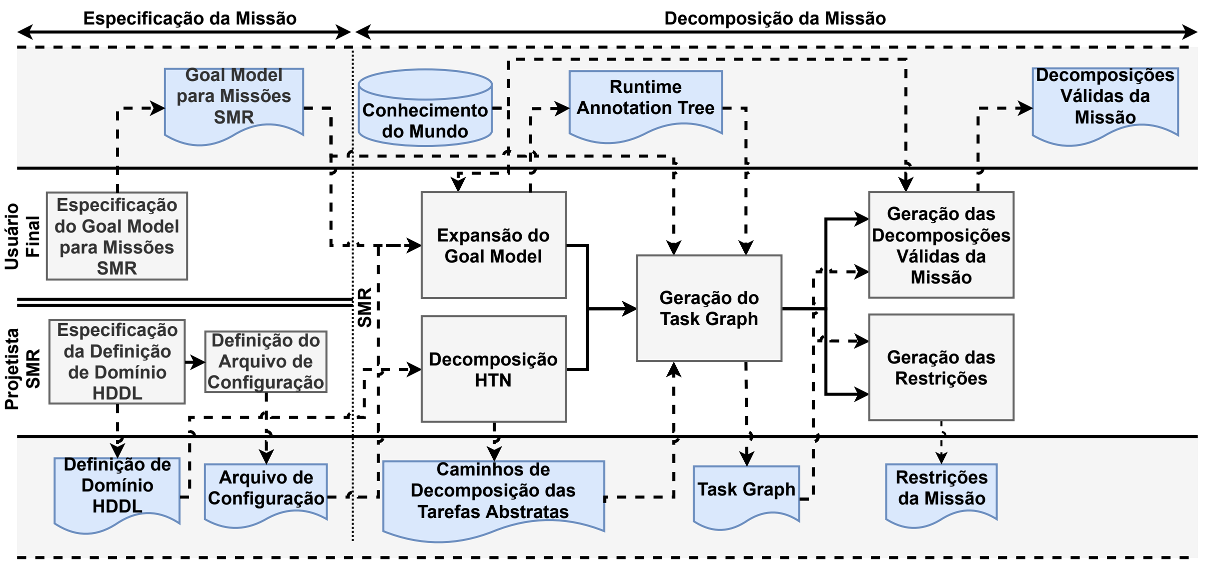

Especificação da Missão
Decomposição da Missão
Goal Model para Missões SMR
Conhecimento do Mundo
Runtime Annotation Tree
Decomposições Válidas da Missão
Usuário Final
Especificação do Goal Model para Missões SMR
Expansão do Goal Model
Geração do Task Graph
Geração das Decomposições Válidas da Missão
SMR
Projetista SMR
Especificação da Definição de Domínio HDDL
Definição do Arquivo de Configuração
Decomposição HTN
Geração das Restrições
Definição de Domínio HDDL
Arquivo de Configuração
Caminhos de Decomposição das Tarefas Abstratas
Task Graph
Restrições da Missão

Figure 2: Visão geral do *framework* MutRoSe

Os arquivos gerados na etapa de especificação da missão são três. O primeiro deles é o *goal model* para missões SMR, que representa uma visão global da missão com as definições das restrições entre as tarefas e possíveis parâmetros a serem instanciados utilizando o conhecimento do mundo. O segundo é a definição de domínio HDDL (Hierarchical Domain Definition Language) [3], onde são definidas as tarefas abstratas e suas possíveis decomposições em termos de métodos e ações. Por fim, o terceiro arquivo gerado é o arquivo de configuração, que faz ligações entre os dois modelos, *goal model* e definição de domínio HDDL, e também entre os predicados definidos no HDDL e o conhecimento do mundo.

Na etapa de decomposição da missão temos 5 grandes etapas. As duas primeiras delas, que podem ser realizadas em paralelo, são a expansão do *goal model*, onde o conhecimento do mundo é utilizado para realizar a instanciação de parâmetros, expandir o *goal model* de acordo e gerar a *Runtime Annotation Tree*, e a decomposição HTN (*Hierarchical Task*

Network) [4], onde as tarefas abstratas são decompostas e seus possíveis caminhos de decomposição são gerados. Em seguida, a terceira etapa é a geração do *Task Graph*, onde se utiliza os resultados das etapas anteriores e o *goal model* para gerar o *Task Graph*, que é uma estrutura intermediária que representa a missão expandida e com as decomposições das tarefas instanciadas. Por fim, as duas últimas etapas, que também podem ser realizadas em paralelo, são a geração das decomposições válidas da missão, onde o *Task Graph* é utilizado para gerar todas as decomposições válidas da missão dado o conhecimento do mundo, e a geração das restrições, onde as restrições entre as tarefas são geradas a partir do *Task Graph*.

Resultados

Foram realizados três experimentos para avaliação do trabalho, os quais foram: (i) modelagem de missões da comunidade de sistemas autoadaptativos, (ii) geração de iHTNs (*instantiated* HTNs) [5] e (iii) teste de desempenho. Na modelagem de missões da comunidade foram modelados exemplos de missões do repositório RoboMAX [6], onde os modelos foram validados com seus respectivos autores e foi gerada uma saída da decomposição com um exemplo de conhecimento do mundo. Já na geração de iHTNs buscou-se transformar a saída gerada pelo processo de decomposição em um modelo já conhecido da literatura, onde foi escolhida a iHTN, e com isso realizar uma breve simulação de um dos exemplos utilizados. Por fim, o teste de performance buscou avaliar a performance do processo de decomposição avaliando o efeito de diferentes *features* na decomposição, onde se aumentou o conhecimento do mundo de maneira a ter um tempo de decomposição cada vez maior.

Os resultados obtidos para os três experimentos foram positivos. No primeiro experimento os modelos foram validados pelos autores e as saídas geradas foram corretas, onde as ameaças à validade são a interpretação dos autores com relação às linguagens utilizadas e verificação manual da saída da decomposição. Com isso, buscou-se mitigar esses efeitos validando exemplos com autores que já tinham familiaridade com a notação do *goal model* e do HDDL. Já no segundo experimento as iHTNs foram corretamente geradas e a simulação para o exemplo que foi utilizado gerou o comportamento esperado, onde as ameaças à validade são a falta de evidência de que essa iHTN pode de fato ser executada e a verificação manual da corretude da iHTN. Dessa maneira, buscou-se mitigar esses efeitos realizando a simulação, que deu evidências de que de fato é possível executar as iHTNs geradas. Por fim, no terceiro experimento os tempos de decomposição foram aceitáveis, a menos de uma *feature* que deve-se ter cautela ao utilizar, apesar da abordagem atual ser totalmente exploratória, onde a única ameaça à validade é a expressividade do exemplo. Dessa forma, buscou-se mitigar esses efeitos utilizando exemplos reais conhecidos na comunidade que são as missões do RoboMAX.

Conclusões

Com o *framework* MutRoSe endereçamos problemas apontados na comunidade [7]. Esses problemas são: (i) a capacidade de lidar com a variabilidade do mundo real, que torna o conhecimento do mundo dinâmico e portanto devemos ter modelos parametrizados, (ii) a reusabilidade dos modelos, onde diferentes *goal models* podem utilizar a mesma definição HDDL, e (iii) a possibilidade de modelar a missão sem conhecimento dos robôs que vão compor o SMR, onde os robôs serão automaticamente alocados em um processo de alocação posterior a decomposição. Reforçamos que tal processo de alocação está fora do escopo desse trabalho. Além disso, um processo de decomposição da especificação foi proposto buscando evidenciar que de fato é possível automaticamente decompor a missão, onde ao combinarmos esse processo com processos de formação de coalisões, alocação de tarefas e execução de tarefas possibilita um processo totalmente automatizado, diferente das abordagens existentes atualmente.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The use of robots to perform tasks previously performed by humans is increasing with time. With this, the complexity of robotic systems is also increasing, due to the heterogeneity of robotic agents and the need of collaborative behavior in multi-robot systems (MRS). This increase in complexity makes the most common approaches used by MRS designers, which often consists in procedural code [1], too complex and hard to maintain. In addition, these approaches often lack high-level abstractions that can be more easily understood by non-expert users. Thus, the need for modeling frameworks arises, where the models need to account for the dynamic nature of environments robots are inserted in, robots capabilities, task dependencies, execution constraints and other aspects which are inherent of this type of system. Moreover, task decomposition processes that can make use of such models are also necessary in order for task allocation, control and execution to be performed seamlessly.

It has been pointed out in the literature that high-level modeling approaches can be utilized for behavioral modeling of robots [11]. According to Garcia et al [12], the H2020 Robotics Multi-Annual Roadmap (MAR)[1] establishes that tasks for deployed teams of robots will be configured and specified by users that may not be familiar with programming languages. In the same roadmap it is stated that for this to happen it is required that: (i) robots need to become intuitively integrated with human operators and (ii) we must have highly abstracted mission definitions and algorithms for interaction and operation with untrained users. Finally, it is claimed that precisely specifying missions and transforming them for automatic processing are amongst the main challenges in robotics software engineering. With this in mind, high-level approaches come up as a way to make

[1]https://eu-robotics.net

MRS mission specification more user-friendly while keeping the specification unambiguous.

According to Rizk et al [2], four main blocks have been identified in order to design MRS capable of accomplishing complex tasks or even a combination of them. In this work we call such an accomplishment as a *robotic mission*, or mission, for brevity. Those main MRS blocks are depicted as a workflow in Figure 1.1, where the blocks inside the red rectangle are the focus of this approach. The basic functioning consists in when given a specification of a complex task (i.e., a mission) one performs (i) task decomposition, where tasks are divided into subtasks until a finite set of elemental tasks (actions) is reached, followed by (ii) coalition formation, where given one's tasks and agents the robot coalitions (teams) are defined in order to work on subsets of the tasks. Then, (iii) task allocation is performed, where one's set of subtasks is allocated to the set of agents in the formed coalitions, and finally the agents perform (iv) task planning, control and execution, where the sequences of actions are then executed.

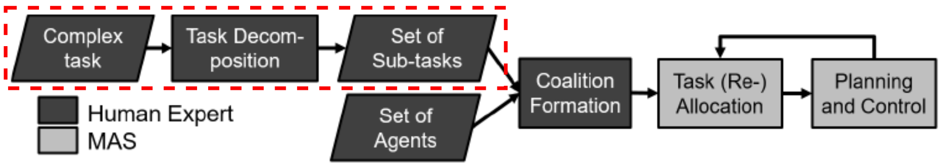


Figure 1.1: Workflow for MRS [2]

From the previously shown workflow, one can note that it requires significant work by a human expert. This is confirmed by the fact that there is no implementation of completely automated MRS, i.e., a system where a specification is given to it and it perform the subsequent blocks of the workflow in an automated fashion. In addition, the highest level of automation verified in existing approaches comprises only coalition formation, task allocation and task planning, control and execution (or control) [2]. This lack of works that automate the task decomposition step is possibly due to the lack of specification approaches that enable the system designer to specify the mission and its specificities in a clear and concise way. This specification could then be used by a task decomposition process to automatically generate the set of subtasks to be executed by the robots, which is finally given as input to further processes. This points out that mission specification plays a major role in the ability of the system to perform automated mission decomposition and execution, thus showing the importance of this work's proposal.

With the previous considerations in mind, it is worth mentioning that many approaches were proposed in order to use models with a higher level of abstraction for specifying MRS behavior, in order to avoid the complexity of procedural code [5,11–14,16]. These models are mostly used at runtime in order to control the execution of MRS systems given a resulting plan structure and cannot be used for reasoning in deliberative approaches. Moreover, all of them lack to capture some aspect of MRS modeling, which results in models that are too domain-specific or that are not sufficiently parameterized in order to account for the system's knowledge about itself and the environment, i.e., they are not general-purpose or do not completely deal with real-world variability. To the best of our knowledge, there is no approach that takes into account all the aspects needed for modeling and decomposing missions for multi-robot systems, where the main ones are: (i) consideration of the real world variability, which can be achieved by having parameterized models that can be instantiated using the current world knowledge, (ii) consideration of the several type of constraints [8], but mainly capability constraints and execution related constraints, (iii) the decoupling of the mission specification from the actual robots, where the end-user specifies the mission based on what needs to be performed and robots will be allocated accordingly during execution, and (iv) the focus on reusability, where models designed by end-users will make use of libraries of tasks and skills which can be defined by an expert for that specific domain or even for general use.

Some of the previously mentioned aspects are considered as future directions for research in [7] and aim to provide specifications that enable robotic applications to be robust in order to cope with the variability of real-world scenarios. When it comes to the task decomposition step, these aspects need to be taken into account in order to automatically decompose MRS missions without the need to generate new models or modify current ones and also to enable the user to specify constraints and restrictions which will create dependencies between tasks that need to be considered at a task allocation step for a more accurate calculation of the overall system utility. A task allocation process is out of the scope of this work but represents a crucial step towards more efficient MRS execution. In this sense, the result generated by a task (mission) decomposition step, which is part of this work, can provide the necessary information that an allocation process needs in order to find the best possible allocation which will improve the system's overall quality of service (QoS).

In addition to the lack of works that consider the previously described aspects, there exists indications that the models should be hierarchical since many previous works make use of this type of structure [5, 11, 12]. This is possibly due to the fact that hierarchical models are easy to systematically verify in an automated fashion, given that they can be easily represented in a tree-like data structure. Furthermore, this hierarchical structure must have a higher-level view of the mission than current planning structures, like HTNs, since there is the need to account for high-level constraints between abstract tasks and model parameters in order to enable automated reasoning of the number of task instances needed given the system's knowledge about the world. One hierarchical structure that attends these needs is the goal model, which additionally gives the designer means of eliciting mission requirements (goals/objectives) in a high-level fashion. With the previous considerations in mind, there are no works that actually propose a high-level behavior specification of multi-robot systems which takes into consideration what is needed to perform task decomposition and allocation in an automated fashion, as far as we know.

As previously said, this work proposes a modelling framework for mission specification and decomposition. The models that are proposed can capture what is needed for the system to reason about how can it perform the mission at a given instant, even though they consist of static structures created at design time. In the decomposition step, the previously designed models are used alongside the system's knowledge about the world with the aim of generating task instances, valid mission decompositions and constraints between tasks. This output can then be used by further processes, which are not part of the work, in order to achieve a higher level of automation than the ones found in current approaches. More specifically, it is expected that the output here generated can be used by an allocation process which can find the best allocation possible, given some predefined utility function.

1.2 Problem Definition

Given the lack of works that propose high-level approaches for mission specification that take into consideration the desired features previously described, this proposal investigates what is needed for this kind of approach to be feasible and its contributions to the community. The proposed models build upon the Contextual Runtime Goal Models (CRGM) formalism, as proposed in the GODA framework [17], in order to describe MRS missions and upon the HTN [4] formalism in order to specify tasks and their decompositions. Both of these formalisms were chosen in order to provide the means for describing MRS missions in a hierarchical fashion, given the benefits of hierarchical approaches previously discussed. In this sense, the main goal of this work is to provide a high-level goal-oriented way of specifying multi-robot missions, from where the first research question arises:

Research Question 1 (RQ1): How to specify high-level MRS missions that can be used for reasoning in deliberative approaches?

With a new way of specifying MRS missions in a high-level fashion it comes the opportunity of automating the task decomposition step of the MRS workflow. As such, we would like to further advance our contribution towards the ability to automatically decompose MRS missions, in order to enable the automation of the task decomposition process as part of the MRS workflow (*c.f. Figure 1.1*). In this work, such decomposition process is intended to be performed in an automated fashion, generating the task instances, alongside constraints between them, and (possibly) the valid ways of decomposing the mission in the end. Then, the outcome of our mission decomposition process is envisioned to be used by a task allocation process in order to decide the best possible way to execute the mission. Furthermore, this process needs to take into consideration the multiple kinds of dependencies that are inherent in this type of system. From this need, our second research question arises:

Research Question 2 (RQ2): Using a high-level multi-robot missions specification approach, how to perform automated mission decomposition taking into account multiple kinds of execution and ordering constraints?

In order to fill in those research gaps, the present work proposes the creation of a framework that provides: (i) high-level models for MRS mission specification, which make easier for end-users to be inserted in the MRS design process, and (ii) a way of automatically decomposing this specification by generating valid mission decompositions and constraints between tasks based on the current system's knowledge about the world.

1.3 Contributions

In order to answer the previously defined questions, this work has four main contributions. First, an extension of the CRGM language (which builds on i* [18]) is proposed in order to provide useful syntaxes for the multi-robot domain, which includes several new features in order to cope with real-world variability and to provide enough flexibility for the user, avoiding the need for he/she to know specific details about the system. Alongside this new language, this work makes use of HDDL [3], a language that provides means to define HTN planning domains, by using a subset of this language and adding some important concepts to it. Secondly, these two languages are used by the MutRoSe (Multi-Robot systems mission Specification and decomposition) framework in order to provide the users and designers of MRS a high-level and unambiguous way of performing mission specification for this kind of system. Through this MutRoSe framework, we aim to answer RQ1.

In addition, this work contributes with a mission decomposition process, in order to answer RQ2. This process takes into account the multiple kinds of dependencies that can exist between tasks and can provide sufficient information for a task allocation process to use for reasoning purposes. We also contribute with a precise specification of the transformation behind the decomposition process, in order to provide evidence on its correctness. With these three contributions in hand, a set of case studies are created, based on mission descriptions from the RoboMAX repository [6], which are validated by their respective authors in order to avoid misinterpretations. These case studies are used to demonstrate that one can in fact specify MRS missions in a high-level fashion (RQ1) while generating the correct output for a given input in order to corroborate with evidence on the correctness of the decomposition process (RQ2).

In a nutshell, the contributions of this work are as follows: (i) the definition of language extensions to be used for MRS missions specification, (ii) the creation of a framework (MutRoSe) to support the MRS missions specification and decomposition process, (iii) the definition of a mission decomposition process, which takes as input the mission specification and the system's knowledge about the world, that generates the task instances and valid mission decompositions and (iv) the specification of case studies using examples from the community in order to provide evidence on the work feasibility and correctness.

1.4 Document Overview

The remaining chapters are organized as follows. Chapter 2 provides the theoretical background needed in order to comprehend the main concepts used in this work. Chapter 3 gives an overview of the proposed work, presenting all the necessary details to understand the approach. Chapter 4 gives an in-depth analysis of the decomposition process, where we formalize all of the steps needed in order to perform the mission decomposition. Chapter 5 provides the processes performed in order to evaluate the approach and give indications of its correctness. Chapter 6 deals with the related work and the contributions of this work. Chapter 7 presents the work conclusions and discussions of possible future works.

Chapter 2

Theoretical Background

2.1 Multi-Robot Systems

A Multi-Robot System (MRS) is a specific type of Multi-Agent System (MAS) where the intelligent agents that compose the system are physical entities, namely robots [2]. Notably, a collaborative heterogeneous multi-robot system is a MRS where robots need to collaborate in order to perform tasks to accomplish a desired goal, where they may have different capabilities and tasks may require different sets of these capabilities [19,20]. According to Khamis et al [19], the popularity of MRS is due to the advantages this kind of system has when compared to single robot systems, which are:

- Resolving task complexity: some tasks may be quite complex for a single robot to do or they might even be impossible. This complexity may be also due to the distributed nature of the tasks and/or the diversity of the tasks in terms of different requirements.
- Increasing the performance: task completion time can be dramatically decreased if many robots cooperate to do the tasks in parallel
- Increasing reliability: increasing the system reliability through redundancy because having only one robot may work as a bottleneck for the whole system especially in critical times. This is not the case when multiple robots are present since if when performing tasks one robot fails, others could still do the job.
- Simplicity in design: having small and simple robots will be easier and cheaper to implement than having only single powerful robot with all the required capabilities.

These advantages made the research in this area increase over the years, attracting several researchers from academia and industry.

2.2 Mission Specification

Mission specification (often called behavior specification)is the process in which step-by-step instructions are generated to guide one or more robots to accomplish a set of tasks [21]. In particular, model-driven behavior specification [1] follows the Model-Driven Engineering (MDE) paradigm, where the system's designer builds models in order to specify the desired behaviors, goals, tasks, actions and possible environment conditions that enable their execution. The need of a model-driven approach comes from the fact that it is common practice for roboticists to specify a robot's behavior using conventional procedural code using complex arrangements of conditional statements [1], which can become too complex and hard to maintain as robotic systems are getting bigger and more complex with time. In the field of MAS it has been shown that high-level approaches can be utilized for the behavior specification of robots [1], which is another indication that research in MDE approaches for MRS behavior specification is indeed necessary.

2.3 Task Allocation

After the behavior specification step in MRS design one must have as a result the specification system goals, tasks and the conditions in which they must be executed. Then, a task decomposition process must be performed since tasks can have multiple ways of being accomplished, by means of different action sequences, depending on the conditions previously defined. This decomposition is the first step towards task allocation, which is the known as Multi-Robot Task Allocation (MRTA) problem. The MRTA problem is an optimization problem which involves decomposition, allocation and scheduling [8], where the main objective is to maximize the overall system utility when assigning tasks to robots taking into consideration the possible constraints. The types of constraints that can be consided are (i) capability constraints, related to robots capabilities, (ii) capacity constraints, related to physical capacities of robots like weight and speed, (iii) simultaneity constraints, where tasks must be executed at the same time, (iv) non-overlapping constraints, where tasks must not be executed at the same time, and (v) precedence constraints, where a task must be executed before another.

When solving the MRTA problem, several types of tasks can be present and this needs to be taken into consideration. In order to deal with these different tasks, a task types terminology was initially proposed by Zlot [22], where we have the following:

- Elemental Task: A task that is not decomposable. This type of task is also called primitive task or even action.

- Decomposable Simple Task: A task that can only be decomposed into elemental or in other decomposable simple subtasks. For this type of task, there must not be any multirobot-allocatable decomposition of this task.

- Simple Task: A task that is either an elemental task or a decomposable simple task.

- Compound Task: A task that can be decomposed into a set of simple or compound subtasks. There is a requirement that there is exactly one fixed full decomposition for it.

- Complex Task: A multiply decomposable task for which there exists *at least one* decomposition that is a set of multirobot-allocatable subtasks. Each subtask in the decomposition of this type of task can be either simple, compound or complex.

These concepts are illustrated in Figure 2.1. Also, the concept of full decomposability must be clear for a better understanding of this terminology. A task is fully decomposable if a set of simple subtasks can be derived within a finite number of decomposition steps, where this decomposition is called the full decomposition of this task.

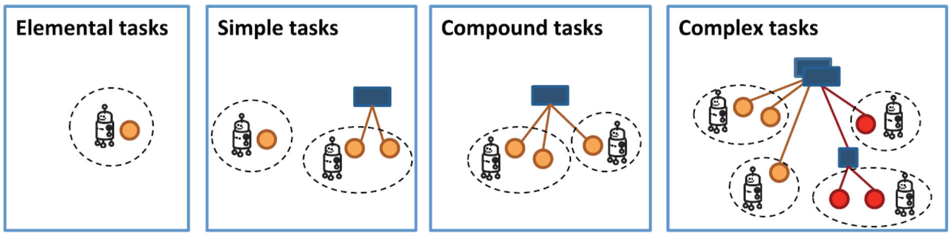


Figure 2.1: Illustration of the task terminology [8]

2.3.1 The iTax Taxonomy

In order to solve the MRTA problem we must take into consideration several aspects which are mostly related to the previously mentioned constraints. Another important aspect of this problem is the degree of interdependence of agent-task utilities, for which we have the iTax taxonomy [8]. This is a two-level taxonomy where in the first level we have a single dimension which actually defines the degree of interdependence of agent-task utilities and in the second level we have Gerkey and Matarić's taxonomy [23].

In Gerkey and Matarić's taxonomy, MRTA problems are classified in three distinct axes. In the first axes there is the distinction between Single-Task (ST) robots, which can only execute one task at a time, and Multi-Task (MT) robots, which can execute multiple tasks at a time. In the second axes there is the distinction between Single-Robot (SR) tasks, which can be executed by only one robot, and Multi-Robot (MR) tasks, which can (and sometimes need to) be executed by multiple robots. In the third and final axis we have the distinction between Instantaneous Assignment (IA), where the problem only involves instantaneous allocations, and Time-extended Assignment (TA), where the problem also involves planning for future allocations.

In the single dimension which is only present in iTax we have 4 categories, where we have increasing level of dependency in which higher categories involve lower ones, to describe task allocation problems: No Dependencies (ND), In-schedule Dependencies (ID), Cross-schedule Dependencies (XD) and Complex Dependencies (CD). The ND category is composed of task allocation problems with simple or compound tasks where the effective utility for a task does not depend on any other tasks or agents in the system, i.e., they have independent agent-task utilities. The ID category is composed of task allocation prolems with simple or compound tasks where the effective utility of an agent for a task depends on other tasks that were assigned to the agent, i.e., the agent-task utilities have intra-schedule dependencies where there may exist constraints between tasks on the agent's schedule. The XD category is composed of task allocation problems with simple or compound tasks where the effective utility of an agent for a task depends on its schedule and on the schedule of other agents, i.e., the agent-task utilities have inter-schedule dependencies in addition to the in-schedule dependencies. It is important to notice that dependencies in the XD category the dependencies are called "simple" dependencies, for which the optimal task decomposition can be determined before task allocation. Finally, the CD category is also composed of complex tasks where the effective utility of and agent for this type of task not only depends on the schedules of other agents but also on the chosen task decomposition, i.e., agent-task utilities for complex tasks have inter-schedule dependencies. In constrast to the XD category, in the CD category the task decomposition problem is coupled with the task allocation one where the optimal task decomposition cannot be determined before task allocation. These categories are illustrated in Figure 2.2, where the different types of dependencies are depicted.

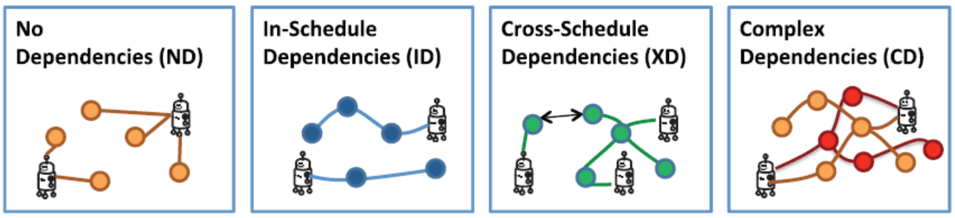

No Dependencies (ND)
In-Schedule Dependencies (ID)
Cross-Schedule Dependencies (XD)
Complex Dependencies (CD)

Figure 2.2: Illustration of the iTax categories [8]

2.4 Robots Capabilities

According to Kunze et al. [24] a capability is the ability to perform certain actions, where a robot has a capability due to its components. It is important to note that there is no unique definition of the term capability, since this is often interchanged with the term skill and sometimes these terms have different meanings. An example of the differentiation between these two terms is given by Olivares-Alarcos et al. [25], where the term skill only includes what the agent can do because of its physical qualities and, in addition, it implies that the achievement is positively quantified. Given the previous definition of capability, one must notice that capabilities are related to actions (or elemental tasks) and thus tasks that are decomposable may require different capabilities depending on the chosen decomposition. Also, a robot may lose or have reduced capabilities over time due to hardware or software malfunction or even broken hardware. The aforementioned aspects are reasons why capabilities do not automatically enable the robot to perform a task [25], alongside the current context inferred using the world knowledge. In this sense, the specification and allocation steps need to take capabilities in consideration, where the process in which it is determined if a robot is capable of performing some action or task is called capability matchmaking [24].

2.5 HDDL

Hierarchical Domain Definition Language (HDDL) [3] is an extension to the Planning Domain Definition Language (PDDL) for hierarchical planning and aims to cover the problems present in other hierarchical planning languages while being as close as possible from PDDL as it can, given that PDDL is a very mature and robust language in the non-hierarchical planning field. When a planning procedure is executed it uses the definitions in HDDL in order to decompose abstract tasks into a sequence of primitive tasks (called actions) by the use of methods. In HDDL there are two types of files to be defined:

- Domain files: Each of these files define a domain description which defines several concepts listed below.
  - types: The allowed types for variables
  - constants: The defined constants for the domain
  - predicates: The possible predicates to be used as preconditions and effects, which are boolean expressions
  - task: Abstract task with name and received parameters with their respective types

- method: Method with name, received parameters with respective types, preconditions and subtasks.
- action: Action (primitive task) with name, received parameters with their respective types, preconditions and effects.

- Problem files: Each of these filed define a problem to be solved using a specific domain and define the concepts listed below.
  - domain: The name of the considered domain
  - objects: Objects to be considered, which will replace variables
  - htn: Initial HTN, which contains the abstract tasks to be decomposed
  - init: Initialization containing predicate and functions initializations

2.6 HTN Planning

Hierarchical Task Network (HTN) Planning is used in this work in order to decompose tasks based on the definitions given by HDDL. Although planning is not the focus of this work, this is an important tool that is used in the decomposition process. In HTN Planning each state of the world is represented by a set of atoms and each action corresponds to a deterministic state transition, being similar to classical planning in this sense. In this kind of planning, the objective is to perform some set of tasks having as input a set of operators and a set of methods, which gives the rules and constraints for decomposing a task into subtasks (smaller tasks). The planning process decomposes non-primitive (abstract) tasks recursively into smaller subtasks until primitive tasks (actions) are reached [4], which can be directly performed using the planning operators. One of the building blocks of HTN planning is the task network, which definition is as follows [4].

Definition 1 Task Network A task network is a pair $w = (U, C)$, where U is a set of task nodes and C is a set of constraints. Each constraint $c \in C$ specifies a requirement that must be satisfied by every plan that is a solution to a planning problem.

Some kind of constraints are shown below. Given that π represents the a solution for w, which contains a set of actions that decompose it, we have:

- Precedence constraint, which is an expression of the form $u \prec v$ and states that the last action in the decomposition of a task u (defined as $last(u, \pi)$) must be executed before the first action in the decomposition of a task v (defined as $first(v, \pi)$).

- Before-constraint, which states a precondition that must be true before the first action in the decomposition of task u and is of the form $before(u, condition)$.
- After-constraint, which states the expected conditions that must be true after the last action in the decomposition of a task u and is of the form $after(u, condition)$.
- Between-constraint, which specifies a condition that must be true just after the last action in the decomposition of a task u, just before the first action in the decomposition of task v and in all states in between and is of the form $between(u, v, condition)$.

The other building block of HTN planning is the HTN method, which definition is as follows.

Definition 2 HTN Method An HTN method is a 4-tuple $m = (name(m), task(m), substasks(m), constr(m))$. $name(m)$ is an expression of the form $n(x_1, x_2, ..., xk)$ where n is a unique method symbol and $x_1, ..., x_k$ are all of the variable symbols that occur anywhere in m, $task(m)$ is a nonprimitive task (decomposed by m) and $(subtasks(m), constr(m))$ is a task network.

2.7 Goal Model

A goal model is a modeling structure which consists in the hierarchical decomposition of goals into sub-goals and tasks, using a tree structure. The main building blocks of a goal model are the goals, the tasks and the actors, even though we can have other structures like resources. Also, there are two types of decompositions which are the (i) OR decomposition, where the fulfillment of one child is sufficient for the fulfillment of the parent, and the (ii) AND decomposition, where all childs need to be fulfilled in order for the parent to be fulfilled.

In addition to the goal model description previously given there can also be contexts and runtime annotations, which compose a Contextual Runtime Goal Model (CRGM). In a CRGM the fulfilment of goals by the system takes into account the contexts, which are observations of the system and its environment, and the cooperation of instantiated goals and tasks at runtime, following the runtime rules which define the behavior by means of certain operators [17]. A simple example of a CRGM is shown in Figure 2.3 where the MRS system has to retrieve an object by the sequential execution of two tasks, which are Move To Location and Grasp Object, but it can only start moving if the battery level is higher than 50%. Here it is shown the context as a rectangle over the AND refinement, but in this work contexts are given as attributes of goals, as is the case of the pistar-GODA tool[1].

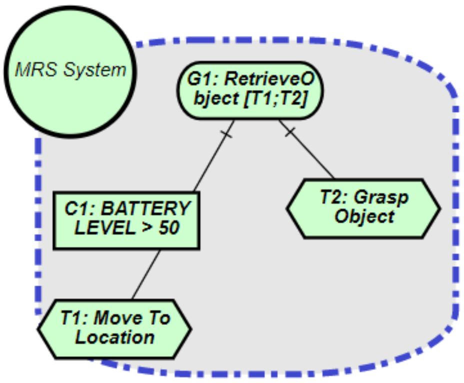


Figure 2.3: A simple example goal model example

2.7.1 Goal Types

According to Dastani et al. [26], a goal type is considered as a specific agent attitude towards goals. Several different goal types were proposed in goal-oriented approaches and were put together in [9] as shown in Figure 2.4. The three main types we are going to discuss here are: Achieve, Perform and Query.

	KAOS	Gaia	JACK	PRS	JAM	Jadex
achieve	x	x	x	x	x	x
maintain	x	x		x	x	x
cease	x					
avoid	x					
optimise	x					
test			x	x		
query					x	x
perform					x	x
preserve			x	x		

Figure 2.4: Different Goal Types [9]

A perform goal is a goal that, once activated, the agent generates plans (if possible) and acts accordingly, regardless of reaching the state denoted by the goal. This kind of goal has a redo flag associated with it, which when set to true makes the agent check for failure of the generated plans and re-executes the applicable ones in case it indeed has failed to reach the desired state.

[1]http://pistar-goda.herokuapp.com/

An achieve goal is a type of goal that has an achieve condition attached to it, which denotes the state the agent will plan and act in order to achieve it. If the achieve condition is not reached after plan execution, the agent will try to create new plans by applying the planning rules in a (possibly) different context and execute these new plans in order to reach the state denoted by the achieve condition. This differs from a perform goal with a redo flag, since the perform goal simply re-executes the (applicable) plans it created the first time and does not create new plans.

Last but not least, a query goal is used to retrieve information, which is a result for a query, and usually does not cause the agent to act, even though there are cases in which it will need to execute some plans in order to retrieve the desired information. This kind of goal is considered to succeed when the agent has all the information it is searching for.

2.8 Object Constraint Language

The Object Constraint Language (OCL) is a language that was initially created to overcome some limitations of UML [27]. This language rapidly grew in scope and it is now widely used in MDE approaches for expressing all kinds of (meta)model query, manipulation and specification requirements [10]. OCL is a typed language, in which all of its expressions evaluate to a type, declarative, where it does not contain assignments, and side-effect free, where it does not modify the state of the system.

In this work, we use two syntaxes of OCL which are (i) the types definitions, for variable declarations, and the (ii) collections operators *forAll* and *select*. A Collection in OCL is an abstract type which is implemented by any type that contains multiple elements of the same type. A diagram with the OCL type hierarchy is shown in 2.5, where we have four parameterized collection types which are Set, Bag, Sequence and OrderedSet and 4 basic types, in addition to the possible user-defined types. The fact that the collection types are parameterized gives us the possibility of having collections of any type we want, from the basic types to user-defined ones.

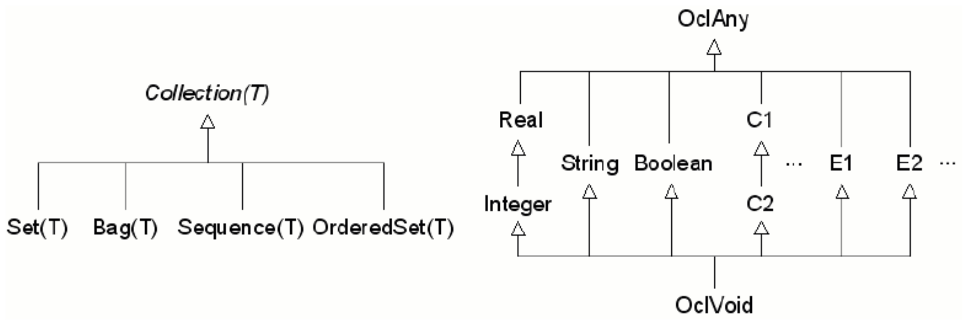


Figure 2.5: OCL type hierarchy diagram [10]

The basic syntax for the *forAll* operator is shown in Equation 2.1, where we have a variable c which is of a collection type, a variable x which is of the same type as the elements in c and a condition to be satisfied here called ϕ.

$$c \rightarrow \mathrm{forAll}(x : xt \mid \phi) \tag{2.1}$$

For the *select* operator we have the syntax defined in Equation 2.2. In this syntax the variable c is of a collection type, x is a variable that represents elements in c, where the type of the elements in c can be specified, if desired, by the xt type specification, and ϕ is a boolean expression which evaluates to true or false for each x in c, where elements where ϕ evaluates to true are selected.

$$c \rightarrow \mathrm{select}(x : xt \mid \phi) \tag{2.2}$$

Chapter 3

A Framework for High-Level MRS Mission Specification and Decomposition

As previously mentioned, this work proposes a framework for mission specification and decomposition for MRS. The process overview is shown in Figure 3.1, which shows the main steps to generate the JSON/XML task instances output. The main idea is that given a description we have (i) the mission specification step and (ii) the mission decomposition step. The mission specification step is where a team of people model all the necessary models and files in order to specify the mission for the MRS to execute. This step is represented in the image by the end-user and an expert designer which communicate with each other. This is the only manual process in the workflow, where the system designer generates the goal model for MRS missions and the HDDL Domain description. The goal model represents the global view of the mission with all of the constraints between abstract tasks and parameters for dynamic task instantiation. The HDDL, in its turn, is where abstract tasks, methods and actions, which are used to perform HTN decomposition, are defined. The mission decomposition step, in its turn, is an automated process performed by the system at runtime that takes all of the models and files from the mission specification step and performs the decomposition of the mission. This decomposition process generates the valid mission decompositions and possible constraints between tasks given the current state of the world stored in the world knowledge.

3.1 Goal Model for MRS Missions

In the goal model for MRS missions we have the basic structure of a CRGM, where we decompose goals into sub-goals or tasks, but with several features introduced in order

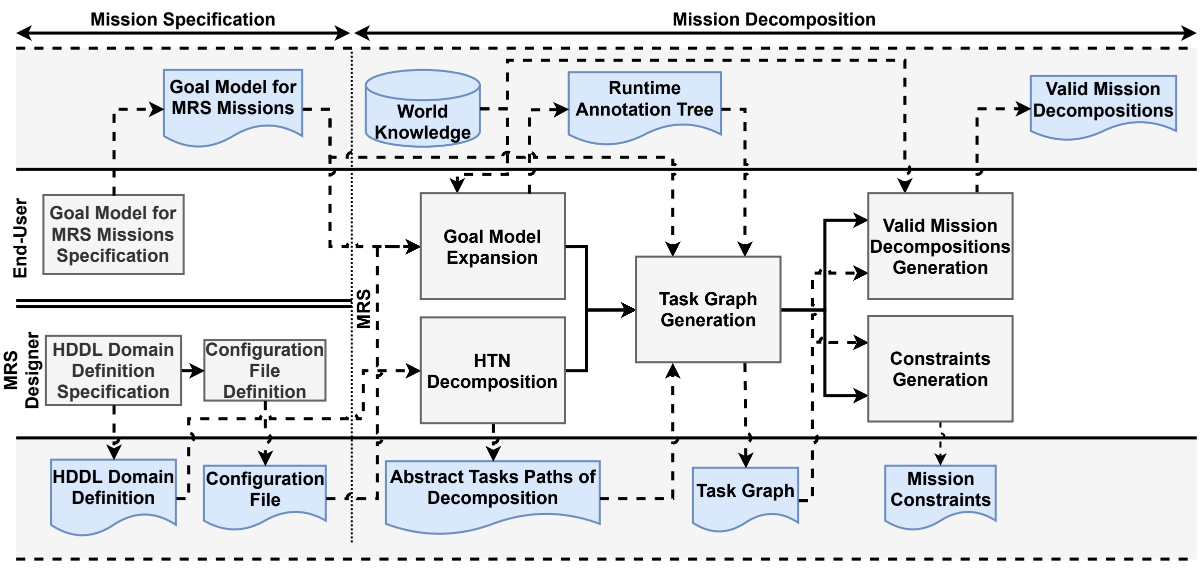


Figure 3.1: Overview of the work's proposed process

to account for the lack of knowledge about the system at design time. The introduced features are: (i) Controls/Monitors syntax, similar to that proposed in [28], in order to define variables and maintain a dataflow between goals and tasks, (ii) goal types, as proposed in [9], (iii) OCL [27] expressions for variables and conditions definitions, (iv) the fact that tasks must map to tasks in the HDDL domain definition and (v) the addition of the specification of the group and divisible attributes for goals. With this in mind, the goal model can be defined as in Definition 3.

Definition 3 The Goal Model can be represented by the tuple GM = (G,AT,E) where:

- G is the set of goal nodes (GN)
- AT is the set of task nodes (TN)
- E is the set of edges, which can be AND or OR edges
- L is the set of leaf nodes. For each leaf node l we have that $l \in AT$ or $l \in G$ iff $l.GT = Query$

In addition, goal model goal nodes and task nodes can defined as in definitions 4 and 5, respectively.

Definition 4 A Goal Model goal node can be represented by the tuple GN = (GT,CTRS,MTRS,AC,QP,CT,RT), where:

- GT is the GoalType property. This can assume one of three values: (i) Query, (ii) Achieve or (iii) Perform
- CTRS is the Controls property. This is a list of variables where each variable is a tuple $v = (v_n, v_t)$ where v_n is the variable name and v_t is the variable type
- MTRS is the Monitors property. As is the case of the Controls property, this is a list of variables where for each variable we can have $v_t = \emptyset$
- AC is the AchieveCondition property. This is only defined if $GN.GT = Achieve$
- QP is the QueriedProperty property. This is only defined if $GM.GT = Query$
- CT is the CreationCondition property. This represents a precondition for achieving the goal, which can be a boolean condition or a list of events
- RT is the goal node's runtime annotation

One important note with respect to goal nodes is that runtime annotations are expressions that are related to the goal node's children and can use one of three operators: (i) parallel (#), (ii) sequential (;) or (iii) fallback (FALLBACK). It is important to note that each goal can only contain runtime annotations with one type of operator

Definition 5 A Goal Model task node can be represented by the tuple TN = (L,P,RN) where:

- L is the Location property
- P is the Params property
- RN is the RobotNumber property

3.1.1 Controls/Monitors Syntax and OCL Variable Declaration

The idea of a dataflow inside the goal model was initially proposed in [28]. The idea is a link from a Goal/Task that produces some data (Controls), represented by some variable, and other Goals/Tasks that consumes this data (Monitors). In this work, this is represented as a property of a goal object, where we define the monitored variables list (Monitors) and the controlled variables list (Controls). With this syntax we aim at defining a dataflow in the goal model where we can relate goals. In this case a goal that controls some variable may need to be achieved before a goal that monitors this variable, depending if the data represented by the variable is outdated or not (or if it is available or not). Some mechanism need to be created in order to detect if data is outdated or not, but this kind of syntax can be very useful for reasoning at runtime. The variable declaration is made using OCL syntax of the form $variable\_name : variable\_type$, where the type is optional for monitored variables since they must already have been previously declared in the controlled variables list of some goal.

3.1.2 Goal Types and OCL Statements

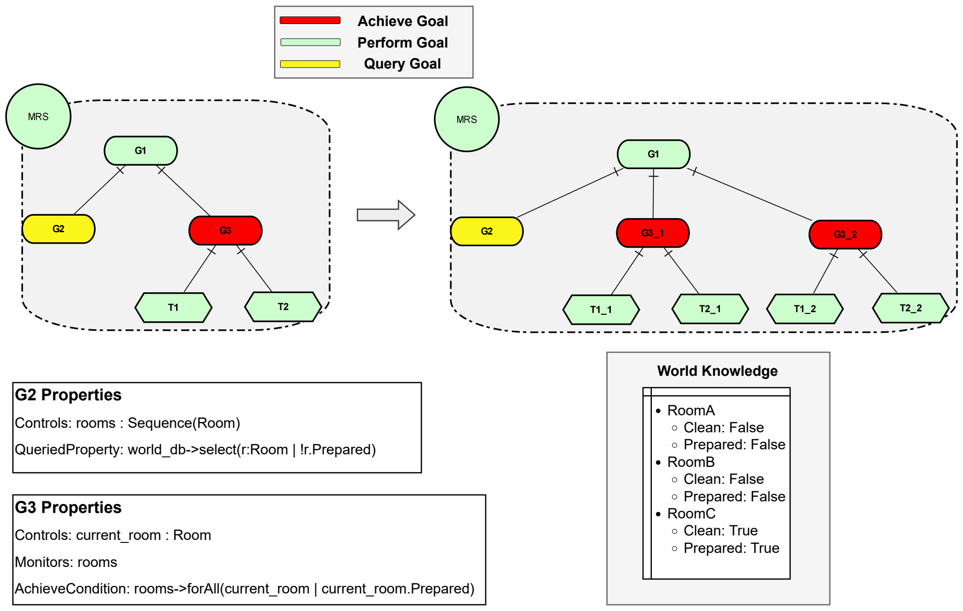
In the goal model for MRS mission specification goals can be of three different types, which are:

- AchieveGoal: When active, this goal type aims to achieve some specific condition, which in this work is represented by the AchieveCondtion. When it reaches its objective condition or it reaches the FailureCondition it terminates, either with success or failure, depending on the state the system is in. The condition to be achieved can be only a condition over some variable or an iteration over some collection variable where some condition needs to be achieved for every element inside it, where we use the forall OCL statement.

- PerformGoal: This is the default goal type in the goal model, where it only performs its children plans and doesn't expect to achieve any specific condition. Success in this case is achieved if its decomposition, by means of its children, is successfully achieved. In the literature, this kind of goal can also have a retry flag set to true, if desired.

- QueryGoal: This goal type represents a query to some knowledge base or the agents internal state. It has a QueriedProperty which is the name of the variable where the data is stored, which will be in its controlled variables list. The query in this type of goal is defined by a select OCL statement.

An example of the effect of an achieve goal type on the goal model parsing is shown in Figure 3.2. In this example, we can note that we use the Query goal G2 where we search for rooms which are not prepared in the world knowledge, which we call world_db, using a select OCL statement. The select statement in G2 instantiates the variable rooms, of type Sequence(Room), with values [RoomA, RoomB]. When the variable rooms is instantiated, the achieve goal G3 uses it in a forall statement using current_room as iteration variable where the objective is to have all of the elements in rooms prepared. Since rooms has two elements, the effect G3 has on the goal model parsing is to create two instances of itself, G3_1 and G3_2, where one must be executed in RoomA and the other on RoomB. With this two G3 instances we also have the creation of two instances for each child of G3, which are tasks T1 and T2.



G2 Properties
Controls: rooms : Sequence(Room)
QueriedProperty: world_db->select(r:Room | !r.Prepared)

G3 Properties
Controls: current_room : Room
Monitors: rooms
AchieveCondition: rooms->forAll(current_room | current_room.Prepared)

World Knowledge
- RoomA
  - Clean: False
  - Prepared: False
- RoomB
  - Clean: False
  - Prepared: False
- RoomC
  - Clean: True
  - Prepared: True

Figure 3.2: Example of the achieve goal effect on the goal model parsing

3.1.3 Tasks and Their Attributes

Tasks must have their names mapping to abstract tasks names in the HDDL domain definition. As defined in Definition 5, these objects can have three specific properties which are (i) Location, for which the value is given by a variable that needs to be defined in the parent goal monitored or controlled variables list, (ii) Params, for which the value is given by a list of variables that define additional parameters to the location one, and (ii) RobotNumber, which represents the number of robots to execute the task and can be expressed in two ways: (i) by an integer range in the format "[n1,n2]" where n1 is the minimum number of robots and n2 is the maximum one and (ii) by a single integer, which defines a fixed number of robots to execute the task. The RobotNumber attribute is based on the use of cardinalities proposed in [11].

3.1.4 Divisible and Group Goal Attributes

The concept of group and non-group goals is given in [29] and in this work we made a slight change to their definition. In this work we have that group goals are those that can be achieved by multiple robots or even by a single robot, if it is possible and desired. On the other hand, non-group goals are goals that must be achieved by a single robot, i.e., all of its children goals and tasks must be executed by a single robot. In addition to the definition of group and non-group goals, we further add the concept of divisible and non-divisible goals, which only apply to group goals. A divisible goal can be achieved by multiple different teams of robots, where these teams may be composed of multiple robots or by a single robot. In this sense, a divisible goal can have its children goals and tasks achieved by multiple teams of robots. Moreover, non-divisible goals are those which must be achieved by the same team of robots, i.e., its children goals and tasks will be achieved by a single team of robots. The default for every goal is to be a group divisible goal, if no value for the Group and Divisible parameters is given for the goal object.

It is important to note that we have an order of priority to execution constraints, where:

- Non-group goals have the higher priority
- Group and divisible goals have the lowest priority
- Group and divisible goals have no constraint, thus they have no priority

This leads to the fact that when a goal which is in a higher level in the goal model and states constraints of the higher level priorities, further defined lower level priorities constraints are not considered. In order to illustrate this property we use the goal model shown in Figure 3.3 as an example. First, suppose that goal G3 is a non-group goal and goal G4 is a group non-divisible goal. In this case we have that AT1, AT2 and AT3 are involved in non-group execution constraints with each other, since we have a non-group goal G3 which is a parent of all of these tasks. The constraint in G4 is not considered since a goal in a higher level than it establishes a higher priority constraint. Now, suppose G3 is a group non-divisible goal and G4 is a non-group goal. In this second case we would have that AT1 and AT2 are involved in a non-group execution constraint with each other, but each of them is involved in a group and non-divisible constraint with AT3.

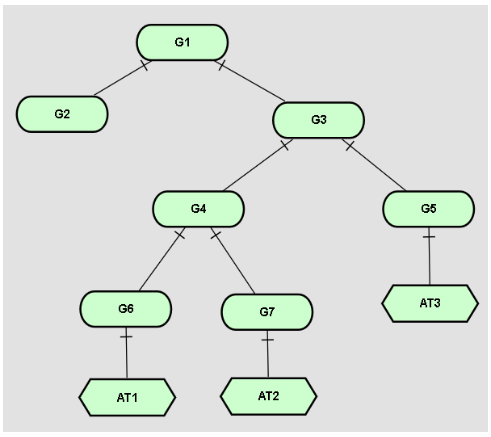


Figure 3.3: Example goal model to illustrate execution constraints properties

3.2 The Role of HDDL

The HDDL language [3] plays a major role in defining abstract tasks decomposition paths into actions. In this approach, modifications were made to the language in order to deal with robotic systems in specific. These modifications were made to the Panda Parser[1] code, which is the official parser for HDDL. In order for the language to have native constructs for dealing with robotic systems, some constructs were added to it. These new constructs are (i) the addition of two new native types, which are robot and robot-team, (ii) the addition of the capabilities domain attribute, where the list of capabilities is defined, and (iii) the required-capabilities attribute for action definitions, where the required capabilities for actions are defined. In order to formalize the language definition, we show the partial modified syntax with the excerpts where the capabilities and required-capabilities attributes were added:

$\langle \textit{domain} \rangle$ ::= (define (domain $\langle \textit{name} \rangle$)
    [$\langle \textit{require-def} \rangle$]
    [$\langle \textit{types-def} \rangle$]
    [$\langle \textit{constants-def} \rangle$]
    [$\langle \textit{predicates-def} \rangle$]
    [$\langle \textit{functions-def} \rangle$]
    [$\langle \textit{capabilities-def} \rangle$]
    $\langle \textit{comp-task-def} \rangle$*
    $\langle \textit{method-def} \rangle$*
    $\langle \textit{action-def} \rangle$*)

[1]https://github.com/panda-planner-dev/pandaPIparser

⟨*functions-def*⟩ ::= (:functions ⟨*atomic-formula-skeleton*⟩+)

⟨*capabilties-def*⟩ ::= (:capabilities ⟨*name*⟩*)

```
⟨action-def⟩ ::= (:action ⟨task-def⟩
            [:required-capabilities (⟨name⟩*)]
            [:precondition ⟨gd⟩]
            [:effects ⟨effect⟩])
```

3.3 Configuration File and World Knowledge

The configuration file is where we define all the necessary information of the world knowledge file, output file and necessary mappings for the decomposition process to be performed. The accepted formats for this file are XML or JSON, where it is detailed:

- The name, type and location of the file that represents the world knowledge, represented by the "world_db" tag. For now, the only format accepted is XML.
- The name and location of the file that represents the output file that is generated at the end of the process, represented by the "output" tag.
- The high-level location types which are user-defined, represented by the "location_types" tag. For now only one high-level location type is accepted. In this simple example, this type would be the Room type.
- The type mapping between the actual types, defined in OCL, and the types in HDDL, represented by the "type_mapping" tag. This is used in order for the designer to define the naming conventions in HDDL regardless of the OCL types.
- The variable mappings which are used in order to define the mapping between variables used in the tasks at the goal model and the variables in the task definition in HDDL, represented by the "var_mapping" tag. For now, this is used in order to know which HDDL variable represents the location variable defined in the goal model for each task. One important notice is that we identify the task by its id in the goal model.
- The semantic mappings that can exist between the knowledge and the structures in HDDL, defined by the "semantic_mapping" tag. The most common type of mapping

observed is between attributes and HDDL predicates, like for example an attribute "is_clean" of a Room type location in the world knowledge and an "clean" predicate in HDDL. For now, this mapping between attributes and predicates is the only valid one but other important kinds of mappings can be introduced if needed.

This configuration file is what enables the process to be customized to the user definitions. This is desired since types, predicates and other structures are domain specific and user-defined.

For the world knowledge we have that the only accepted format at the moment is an XML file. In this sense, it is basically a collection of records with attributes that can be mapped to predicates using the semantic mappings defined in the configuration file. With these mappings, we are able to define the initial state of the world to be used by the decomposition process.

3.4 Mission Decomposition

The mission decomposition process is where an automated decomposition of the mission is performed, based on the models created when performing mission specification. As can be seen in Figure 3.1 the decomposition process of the MutRoSe framework consists of five subprocesses: (i) Goal Model Expansion, where the goal model is expanded based on the world knowledge, (ii) HTN Decomposition, where the HDDL Domain Definition is used in order to generate the possible abstract tasks paths of decomposition, (iii) Task Graph Generation, where the intermediate structure that represents the decomposed mission, which is called the Task Graph, is generated, (iv) Valid Mission Decompositions Generation, where the valid mission decompositions given the current world state are generated, and (v) Constraints Generation, which generates the existing constraints between task decompositions.

3.4.1 HTN Decomposition

In this step the decomposition process of the abstract tasks is performed. This is done using the HTN theory and generating a structure similar to a Task Decomposition Graph (TDG) [30, 31] for each task. The main difference from the structure generated here from a TDG in fact is that we do not perform grounding of our variables, which is the process in which the variables are replaced by objects. In the generation of this non-ground structure we assume that variables are maintained throughout the process in the way they are declared, so that our methods do not add new variables and the decomposition process only deals with the variables declared at the abstract task parameters. In this process we have three major steps: (i) Parsing, where we use the a slightly modified version of the Panda parser in order to generate the necessary structures like tasks, methods and actions, (ii) Non-Ground TDG Generation, where we generate the non-ground TDG for each task based on the previously generated structures, and (iii) Task Decomposition, where we perform decomposition in the non-ground TDG of each task in order to generate the possible decomposition paths.

An example of this decomposition process is shown in Figure 3.4, where we have an abstract task AT1 to be decomposed, which requires three parameters `?v1 ?v2 ?v3`. At the first stage of the decomposition process we generate a TDG-like structure, based on a previous expansion of methods and tasks where methods preconditions are modelled as actions, which is shown in the left side of the figure. At the second stage, we evaluate preconditions and effects leaving everything as either a variable or a constant defined in HDDL, where the variables considered throughout the process are those defined in the abstract task's parameters. Alongside predicates evaluation, we also verify ordering constraints and generate all of the possible paths of decomposition for each method, which is shown at the right side of the figure. It is important to note that we only evaluate the predicates we can based on the world knowledge, assuming as true all of the predicates we cannot evaluate at the moment.

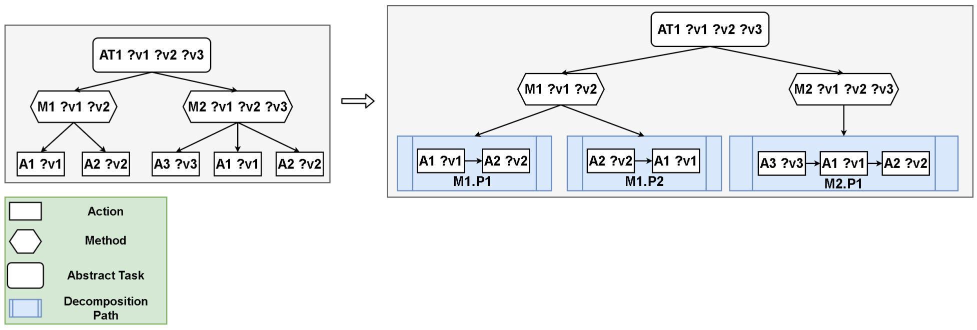


Figure 3.4: Illustration of the HTN decomposition process

In the previously illustrated example we initially had two methods that decompose the abstract task AT1: (i) M1 which requires variables `?v1 and ?v2`, and (ii) M2, which requires variables `?v1, ?v2 and ?v3`. We assume M1 has no preconditions and M2 has one precondition, which is transformed into task A3. In addition, method M1 has no ordering constraints and method M2 has the constraint that A1 must be executed before A2. With these constraints we have that M1, which decomposes AT1 into A1 and A2, has two paths of decomposition M1.P1 and M1.P2, and that M2, which decomposes AT1 into A1, A2 and A3, has only one path of decomposition M2.P1.

The goal model expansion process is the process where the goal model is expanded based on the world knowledge and the universal achieve type goals, which are the only nodes that can expand the goal model. The structure generated at the end of this process is the Runtime Annotation Tree, which represents the expanded goal model with goals, with the exception of means-end decomposed ones, represented by their runtime annotations and task instances IDs respectively changed based on the performed expansions. This process contains two steps, which are: (i) Abstract Task Instances Generation, where expansions on the goal model are performed in order to generate task instances with the correct IDs, and (ii) Runtime Annotation Tree Generation, in which the Runtime Annotation Tree, as defined in Definition 6, is generated based on the goal model expansion and the previously generated task instances. One important note about the Runtime Annotation Tree is that operator nodes can assume as value one of the tree runtime annotation operators previously described.

Definition 6 The Runtime Annotation Tree can be defined as the tuple RT = (OP,G,AT) where:

- OP is the set of operator nodes, which can assume as value one of the tree runtime annotation operators previously described in Section 3.1
- G is the set of goal nodes, which only contain the query goals and the means-end goals from the goal model
- AT is the set of tasks

An example of the goal model expansion process is shown in Figure 3.5. In this example we have a query goal that controls some variable *v*, instantiating it with two objects *[o1,o2]*, which is used by the universal achieve goal G3 in a forall OCL statement. In the abstract task instances process goal G3 expansion is performed and two instances of abstract task AT1 are generated, which are named AT1_1 and AT1_2. For goal G4, which is a simple Perform goal, we have no expansion and thus a single instance of AT2 is generated, named AT2_1. After task instances are generated, the runtime annotation tree generation process performs the goal model expansion once again and generates the Runtime Annotation Tree structure, which has all of its task nodes IDs set to the ones previously created. Note that the universal achieve goal G3 created an extra parallel operator in the tree, which is used to coordinate the generated instances of G3 when the goal model is expanded.

There is a convention in task IDs generation. For tasks that are not a child of any universal achieve goal or of a single achieve goal we have the task original goal model ID plus a "_" and the number of the generated instance, which starts in 1. For goals that

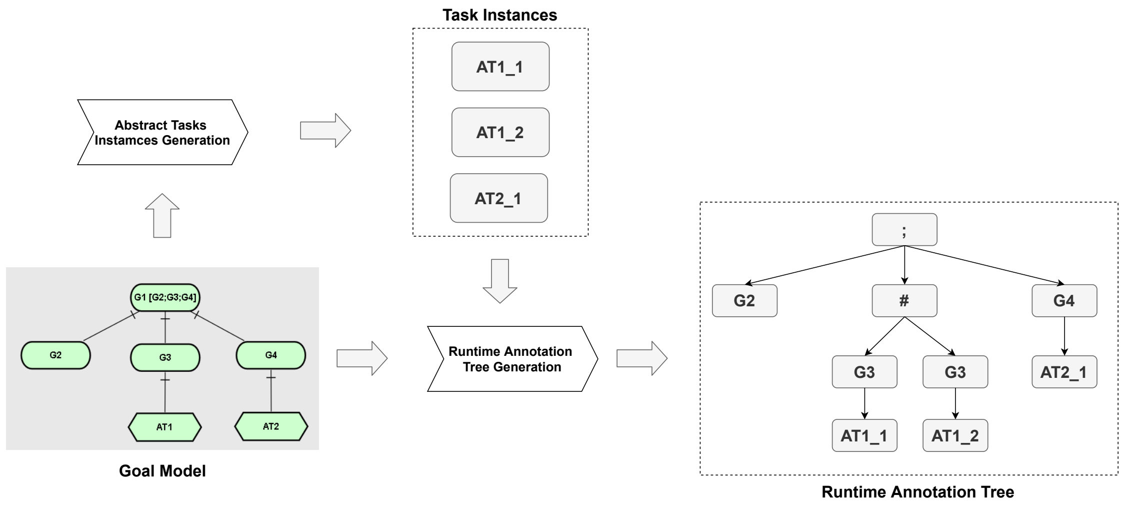


Figure 3.5: Illustration of the goal model expansion process

are a child of nested universal achieve goals we follow the previously described structure and add extra "_" plus the number of the generated instance for each universal achieve goal, where the rightmost number belongs to the lower level nested goal and the leftmost to the higher level one. For example, suppose we had one more universal achieve goal in the previous example that was a child of G3 and it generated 3 instances. In this case we would have six instances of AT1, namely: AT1_1_1, AT1_1_2, AT1_1_3, AT1_2_1, AT1_2_2 and AT1_2_3.

3.4.2 Task Graph Generation

The Task Graph generation process is the process where we generate an important intermediate structure named, as the process description says, the Task Graph. This graph is an intermediate structure that represents the mission decomposition, with the necessary expansions already made and all of the constraint links already generated. This process consists of four major steps: (i) Initial Task Graph Building, where we generate a non-minimal Task Graph, (ii) Final Context Dependency Edges Generation, where context dependency edges are modified, (iii) Execution Constraint Edges Generation, where the execution constraint edges are added to the Task Graph, and (iv) Task Graph Trimming, where the non-minimal Task Graph is trimmed in order to become minimal.

Definition 7 The Task Graph can be defined as the tuple TG = (OP,G,AT,E,D) where:

- OP is the set of operator nodes, which can be #, ;, FALLBACK

- G is the set of goal nodes, which can only be means-end goals
- AT is the set of abstract tasks
- E is the set of directed edges which can be one of four types. Thus: $\forall e \in E,\ e.type \in \{AND, OR, EC, CD\}$ where:
  - AND and OR refer to normal AND and OR decompositions from the goal model
  - EC represent execution constraint links, which have the group and divisible properties
  - CD represent context dependency links
- D is the set of abstract tasks decompositions

It is important to note that the set of nodes can be defined as N, where OP, G, AT and D are all subsets of N. In addition to this definition, we can define that a minimal Task Graph is one where there is not an operator node that has less than two children and that has no goal nodes. Mathematically, we have for a Task Graph TG:

$$Minimal(TG) : \forall n \in OP, (Children(n) \geq 2 \ or \ IsAchieve(n)) \ and \ \forall n \in N, n \notin G \quad (3.1)$$

where IsAchieve(n) returns true iff n is an operator node generated from a single child non-universal achieve goal or from a single child universal achieve goal that only generates one instance.

Context Dependency Before explaining how to generate the Task Graph, the concept of context dependency must be introduced since there special edges in the graph that represents it. A context dependency exists when a task has as an effect a condition that activates the context of some goal and their relation in the goal model is given by a parallel operator. In this case, all of the goal's child tasks are in a context dependency with the task that has the enabling effect and the constraint between them, which by the initial design would be a parallel one, is transformed into a sequential constraint. An example of this type of dependency is given in Figure 3.6, where we show goals and tasks with generic names and runtime annotations. In this example task T1 and goal G3 have a context dependency since T1 has as an effect "C1 = True" and G3 has as context "C1 == True" while they are involved in a parallel constraint because of G1's runtime annotation, which involves T1's parent G2. From the goal model, one would expect that T1 and G3's children, T3 and T4, would have parallel constraints between them, but they are actually involved in sequential constraints in which T1 must come before T3 and T4.

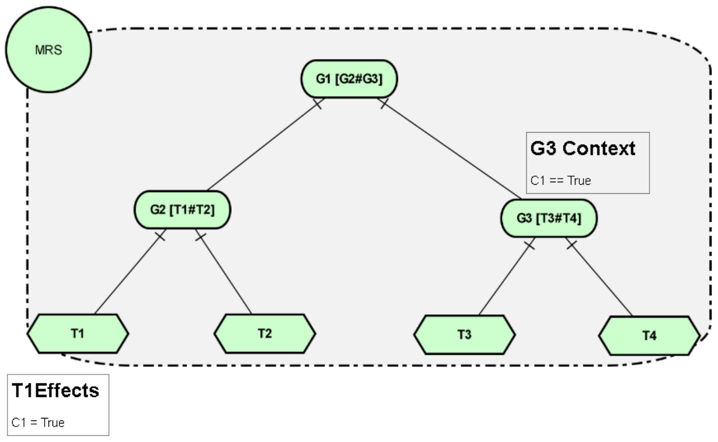


Figure 3.6: Example of a context dependency

Initial Task Graph Building As previously said, in this step the (possibly) non-minimal Task Graph is built. In order to do this, the Runtime Annotation Tree is traversed in a DFS fashion and the goal model is used to retrieve any additional information needed, since every node in the tree relates to a node in the goal model with the exception of forall generated nodes.

Final Context Dependencies Generation In this step we modify the context dependency type edges. This is done since the initial edges, generated in the first step, relate abstract task decomposition nodes with goal/operator type nodes. With this in mind, the goal here is for every context dependency edge we create a context dependency edge between its source node, which will be a decomposition node, and all children of the target node that are task nodes. In addition to this new edges, we have that all of the initially generated context dependency edges are erased since they aren't needed anymore.

Execution Constraint Edges Generation The execution constraint edges created in this step relate to the group and divisible attributes of goals from the goal model. In order to generate these edges the Task Graph is traversed in a DFS fashion, where we keep track of which tasks are involved in constraints and also of the priority of constraints in order to create the correct edges.

Task Graph Trimming In this final step´we take the Task Graph generated by previous steps and transform it in a minimal Task Graph, which follows the definition previously given. The graph is traversed node by node, where nodes that do not satisfy the minimal

Task Graph condition are erased and replaced by their child, with nodes IDs in the graph being modified accordingly.

3.4.3 Valid Mission Decompositions Generation

The valid mission decompositions generation process is the process where the Task Graph is traversed in order to define, as the process description says, the valid mission decompositions. A valid mission decomposition is simply a combination of task decompositions where for each abstract task we can have at most one of its decompositions, depending if OR decompositions are present in the specification. In this process there is only one step to be performed which is the Generation of Valid Mission Decompositions.

In order to verify which decompositions are valid we keep track of the world state for each mission decomposition, where the initial world state is given by the world knowledge. Each operator has its own logic for updating the world state, as will be explained in more detail in further sections. In addition to this, the presence of OR decompositions, which are treated as exclusive OR in this work, raises the need to verify all of the possible combinations of tasks that are children of OR decomposed nodes, with the world state being updated accordingly for each mission decomposition generated from this combinatorial process.

3.4.4 Constraints Generation

In the constraints generation process, all of the mission constraints between task decompositions are generated.

Definition 8 A constraint can be defined as the tuple CTR = (N1,N2,T) where:

- N1 is the first task/decomposition involved in the constraint
- N2 is the second task/decomposition involved in the constraint
- T is the type of the constraint, which can be Sequential (SEQ), Parallel (PAR), Fallback (FB) or Execution (EC)

It is important to note two things: (i) the order between N1 and N2 matters for sequential and fallback constraints, so there is the need to explicitly define which task is the first and which is the second, and (ii) Execution type constraints have two additional attributes which are the group and divisible attributes. Also, for each task in the constraint we know its ID in the Task Graph (TID).

This process consists of four major steps: (i) Initial Mission Constraints Generation, where all of the constraints between abstract tasks are generated, (ii) Transformation of Initial Mission Constraints, where we transform constraints between tasks in constraints between decompositions, (iii) Execution Constraints Generation, where we generate all of the execution constraints, and (iv) Trimming of Mission Constraints, where unnecessary constraints are trimmed in order to have the minimal set of constraints.

Initial Mission Constraints Generation In the initial mission constraints generation step we generate every possible constraint that can be inferred from the Task Graph, where sequential, fallback and parallel constraints between abstract tasks are generated.

Transformation of Initial Mission Constraints In the transformation of initial mission constraints we take the mission constraints generated in the previous step and transform them into constraints between decompositions instead of constraints between abstract tasks. If we have a sequential or fallback constraints, we simply generate constraints of the same type between all of the possible combinations of decompositions. If we find a parallel constraint, we generate sequential constraints between decompositions if and only if we have a context dependency involved and do nothing otherwise. At the end of this process we have only sequential and fallback constraints, since every other constraint that is not explicit is assumed to be parallel.

Generation of Execution Constraints In the generation of execution constraints we generate every possible execution constraint given the execution constraint edges of the Task Graph. These new constraints are added in the set of constraints generated in the previous steps.

Trimming of Mission Constraints The trimming of mission constraints process is where we trim sequential constraints in order to generate the minimal set of constraints, which we call C. This needs to be done because: (i) context dependency generated sequential constraints can make some of the sequential constraints generated in the initial mission constraints generation step unnecessary, (ii) parallel operators generated by universal achieve goals can introduce unnecessary constraints in C and (iii) certain specific combinations of operators can in the end generate more constraints in C than needed. With this in mind, C may not be the minimal set of constraints and we will need to trim for three cases: (a) unnecessary sequential constraint given a fallback constraint, (b) unnecessary sequential constraint given a sequential constraint and (c) unnecessary fallback constraint given a fallback constraint.

In order to better illustrate this, let's give some examples. For case (a) suppose we have three tasks t1, t2 and t3 and that we have three constraints c1 = t1 ; t2, c2 = t2 FB t3 and c3 = t1 ; t3. Since we know t1 is sequential with t2 by constraint c1 and that t2 is in a fallback relation with t3 by constraint c2, we can already infer that t1 is sequential with t3, since t3 always happens in case t2 fails, and thus constraint c3 is unnecessary. In this sense we can trim c3 and have the minimal set of constraints with c1 and c2 only. For case (b) suppose we have the same three tasks t1, t2 and t3 but now we have three sequential constraints c1 = t1 ; t2, c2 = t2 ; t3 and c3 = t1 ; t3. Since we know t1 is sequential with t2 by constraint c1 and that t2 is sequential with t3 by constraint c2, we can already infer that t1 is sequential with t3 and thus constraint c3 is unnecessary. In this sense we can trim c3 and have the minimal set of constraints with c1 and c2 only. For case (c) we can have the same example used for case (b), but constraints c1, c2 and c3 will be fallback instead of sequential. The reasoning behind this case is exactly the same, only changing the operator used.

3.5 A Proposal Illustration

In order to illustrate the process proposed in this work, we show a hospital environment example called *Room Preparation*. This problem consists of robots which collaborate in order to prepare hospital rooms when needed, in order to accommodate incoming patients. It is important to note that by preparing a hospital room we basically mean cleaning and moving furniture around in order to organize it in a predefined standard way. The basic idea of this mission is that (i) a single robot goes into a room and clean it (ii) leaving it afterwards in order to sanitize itself, since it may have been in contact with infectious diseases. When the first robot finishes cleaning the room (iii) a group of robots enters it in order to move furniture around to finish the room preparation process. An important thing to consider here is that these steps needs to happen for every room that is not prepared at the moment the MRS analyzes the mission.

3.5.1 Goal Model

The goal model for the *Room Preparation* mission is shown in Figure 3.7. In this model we have two Query goals which are G2, that fetches the rooms that are not prepared, and G8, that fetches the room where robots get sanitized, which is called "SanitizationRoom". It is worth mentioning that in goal G2 we have the controlled variable *rooms* of type *Sequence(Room)* and in goal G8 we have the controlled variable *sanitization_room* of type *Room*, where these variables are the ones which will hold the result of the select statements in each goal. We also have one achieve goal G3, that iterates over the rooms that are not prepared, based on the monitoring of the variable *rooms* from goal G2, and aims to achieve that the *is_prepared* attribute for each one of them is true. For this iteration to happen, G3 has a controlled variable *current_room* of type *Room* which will be used for the iteration over the values inside of the collection variable *rooms*. One last detail to note here is that goal G10 has as context that the room being considered, represented by the variable *current_room* from goal G3, is already clean, since this goal represents the moving of the furniture that needs to be made after the cleaning process. This context is represented by the *CreationCondition* goal property and its value is *assertion condition "current_room.is_clean"*, where assertion condition represents a boolean condition to be satisfied.

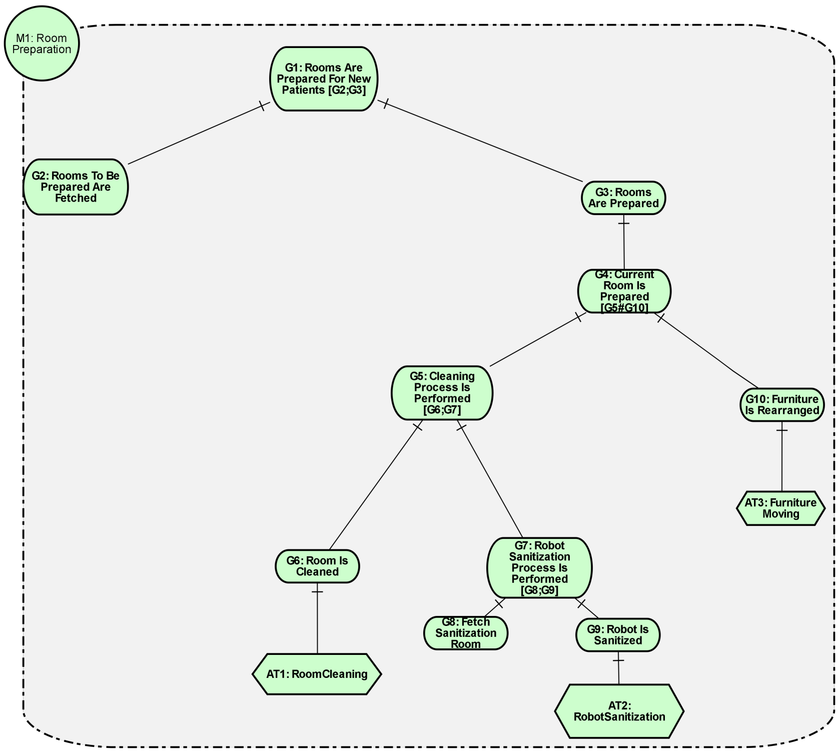

M1: Room Preparation
G1: Rooms Are Prepared For New Patients [G2;G3]
G2: Rooms To Be Prepared Are Fetched
G3: Rooms Are Prepared
G4: Current Room Is Prepared [G5#G10]
G5: Cleaning Process Is Performed [G6;G7]
G10: Furniture Is Rearranged
AT3: Furniture Moving
G6: Room Is Cleaned
G7: Robot Sanitization Process Is Performed [G8;G9]
G8: Fetch Sanitization Room
G9: Robot Is Sanitized
AT1: RoomCleaning
AT2: RobotSanitization

Figure 3.7: Goal model for the Room Preparation example

3.5.2 HDDL Domain File

The HDDL Domain file for the *Room Preparation* contains the necessary tasks, methods and actions alongside with the predicates, capabilities and type definitions. This file is shown in Listing 3.1. Starting with type definitions, in this example we define two robot types, which are MoveRobot and CleanerRobot, and the room type, which is the location type where tasks are performed. Also, we define four predicates: (i) clean, which indicates if a room is clean, (ii) sanitized, which indicates if a robot is sanitized, (iii) prepared, which represents if a room is prepared, and (iv) door-open, which indicates if the door of a room is open. Finally, we define four capabilities, which are moveobject, cleaning, sanitize and door-opening.

When it comes to tasks we define the RoomCleaning, RobotSanitization and FurnitureMoving tasks, each with their own decomposition methods. The RoomCleaning task is the only task with more than one method, which are: (i) the room-cleaning-door-open method, which decomposes the task into the clean-room action if the door-open predicate is true for a given room, and (ii) the room-cleaning-door-closed method, which decomposes the task into the open-door and clean-room actions, in this given order, if the door-open predicate is false for a given room. The RobotSanitization task has a single decomposition method called robot-sanitization which decomposes it into the sanitize-robot action. Finally, the FurnitureMoving task also has a single decomposition method called furniture-moving which decomposes it into the move-furniture action. All of these actions have preconditions and effects, as shown in Listing 3.1.

```
(define (domain hospital)
    (:types room - object
            MoveRobot CleanerRobot - robot)
    (:predicates
        (clean ?rm - room)
        (sanitized ?r - robot)
        (prepared ?rm - room)
        (door-open ?rm - room)
    )
    (:capabilities moveobject cleaning sanitize door-opening)

    (:task RoomCleaning :parameters (?r - CleanerRobot ?rm - room))
    (:method room-cleaning-door-open
        :parameters (?r - CleanerRobot ?rm - room)
        :task (RoomCleaning ?r ?rm)
        :precondition (and
            (door-open ?rm)
        )
        :ordered-subtasks (and
            (clean-room ?r ?rm)
        )
    )
    (:method room-cleaning-door-closed
        :parameters (?r - CleanerRobot ?rm - room)
        :task (RoomCleaning ?r ?rm)
        :precondition (and
            (not (door-open ?rm))
        )
        :subtasks (and
            (a1 (open-door ?r ?rm))
            (a2 (clean-room ?r ?rm))
        )
        :ordering (and
            (a1 < a2)
        )
    )

    (:action clean-room
        :parameters (?r - CleanerRobot ?rm - room)
        :required-capabilities (cleaning)
        :precondition (and
            (not (clean ?rm))
        )
        :effect (and
            (clean ?rm)
            (not (sanitized ?r))
        )
    )
    (:action open-door
        :parameters (?r - CleanerRobot ?rm - room)
        :required-capabilities (door-opening)
        :precondition (and
            (not (door-open ?rm))
        )
        :effect (and
            (door-open ?rm)
        )
    )

    (:task FurnitureMoving :parameters (?rt - robotteam ?rm - room))
    (:method furniture-moving
        :parameters (?rt - robotteam ?rm - room)
        :task (FurnitureMoving ?rt ?rm)
        :precondition ()
        :ordered-subtasks (and
            (move-furniture ?rt ?rm)
        )
    )

    (:action move-furniture
        :parameters (?rt - robotteam ?rm - room)
        :required-capabilities (moveobject)
        :precondition ()
        :effect (and
            (prepared ?rm)
        )
    )

    (:task RobotSanitization :parameters (?r - CleanerRobot ?srm - room))
    (:method robot-sanitization
        :parameters (?r - CleanerRobot ?srm - room)
        :task (RobotSanitization ?r ?srm)
        :precondition ()
        :ordered-subtasks (and
            (sanitize-robot ?r ?srm)
        )
    )

    (:action sanitize-robot
        :parameters (?r - CleanerRobot ?srm - room)
        :required-capabilities (sanitize)
        :precondition (and
            (not (sanitized ?r))
        )
        :effect (and
            (sanitized ?r)
        )
    )
)
```

Listing 3.1: HDDL Domain file for the Room Preparation example

The representation for the task decompositions generated when performing the HTN decomposition step is shown in Figure 3.8. The RoomCleaning task is the only task with more than two possible decompositions, depending if the room's door is open, in contrast to the RobotSanitization and FurnitureMoving tasks, which only have one possible decomposition each.

3.5.3 Configuration File and World Knowledge

The world knowledge XML file and the JSON configuration file are shown in Listings 3.2 and 3.3, respectively. From the configuration file we can verify that the only high-level accepted location type is the Room OCL type, we can also verify: (i) the type mappings

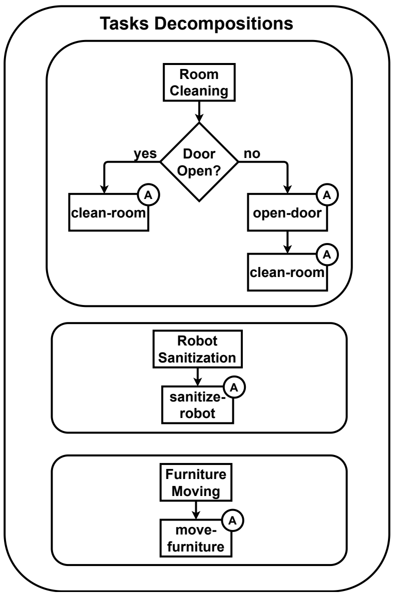


Figure 3.8: Task decompositions for the Room Preparation example

between OCL and HDDL, (ii) the variable mappings for tasks in the goal model and in the HDDL Domain and (iii) the semantic mappings between the HDDL Domain predicates and the attributes in the world knowledge.

```
<world_db>
    <Room>
        <name>RoomA</name>
        <is_clean>False</is_clean>
        <is_prepared>False</is_prepared>
        <door_open>False</door_open>
    </Room>
    <Room>
        <name>RoomB</name>
        <is_clean>False</is_clean>
        <is_prepared>False</is_prepared>
        <door_open>True</door_open>
    </Room>
    <Room>
        <name>RoomC</name>
        <is_clean>True</is_clean>
        <is_prepared>True</is_prepared>
        <door_open>True</door_open>
    </Room>
    <Room>
        <name>SanitizationRoom</name>
    </Room>
</world_db>
```

Listing 3.2: XML world knowledge file for the Room Preparation example

```
{
    "world_db": {
        "type": "file",
        "file_type": "xml",
        "path": "knowledge/world_db.xml",
        "xml_root": "world_db"
    },

    "output": {
        "output_type": "file",
        "file_path": "output/task_output.json",
        "file_type": "json"
    },
    "location_types": ["Room"],

    "type_mapping": [
        {
            "hddl_type": "room",
            "ocl_type": "Room"
        }
    ],

    "var_mapping": [
        {
            "task_id": "AT1",
            "map": [
                {
                    "gm_var": "current_room",
                    "hddl_var": "?rm"
                }
            ]
        },
        {
            "task_id": "AT2",
            "map": [
                {
                    "gm_var": "sanitization_room",
                    "hddl_var": "?srm"
                }
            ]
        },
        {
            "task_id": "AT3",
            "map": [
                {
                    "gm_var": "current_room",
                    "hddl_var": "?rm"
                }
            ]
        }
    ],

    "semantic_mapping": [
        {
            "type": "attribute",
            "name": "is_clean",
            "relates_to": "Room",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "clean",
                "arg_sorts": ["room"]
            }
        },
        {
            "type": "attribute",
            "name": "is_prepared",
            "relates_to": "Room",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "prepared",
                "arg_sorts": ["room"]
            }
        },
        {
            "type": "attribute",
            "name": "is_sanitized",
            "relates_to": "robot",
            "belongs_to": "robots_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "sanitized",
                "arg_sorts": ["robot"]
            }
        },
        {
            "type": "attribute",
            "name": "door_open",
                    "relates_to": "Room",
                    "belongs_to": "world_db",
                    "mapped_type": "predicate",
                    "map": {
                        "pred": "door-open",
                        "arg_sorts": ["room"]
                    }
                }
            ]
    }
```

Listing 3.3: JSON configuration file for the Room Preparation example

3.5.4 Mission Decomposition

Using all of the previously shown models and files we are able to generate our task instances with their respective locations, decompositions and required capabilities. The valid mission decompositions will be composed of these task instances which will be used alongside the generated constraints. Given what was shown in previous sections, the only valid mission decomposition is shown in Figure 3.9. Other task instances and constraints involving them can be found in the generated output, which is shown in Figure 3.10. For example, we have that for the *RoomA* location the door is open, given the world knowledge, and with this we can conclude that task *RoomCleaning* in *RoomA* will not be decomposed by method *room-cleaning-door-closed* and thus task with ID *AT1_1|2* is not present in the valid mission decomposition.

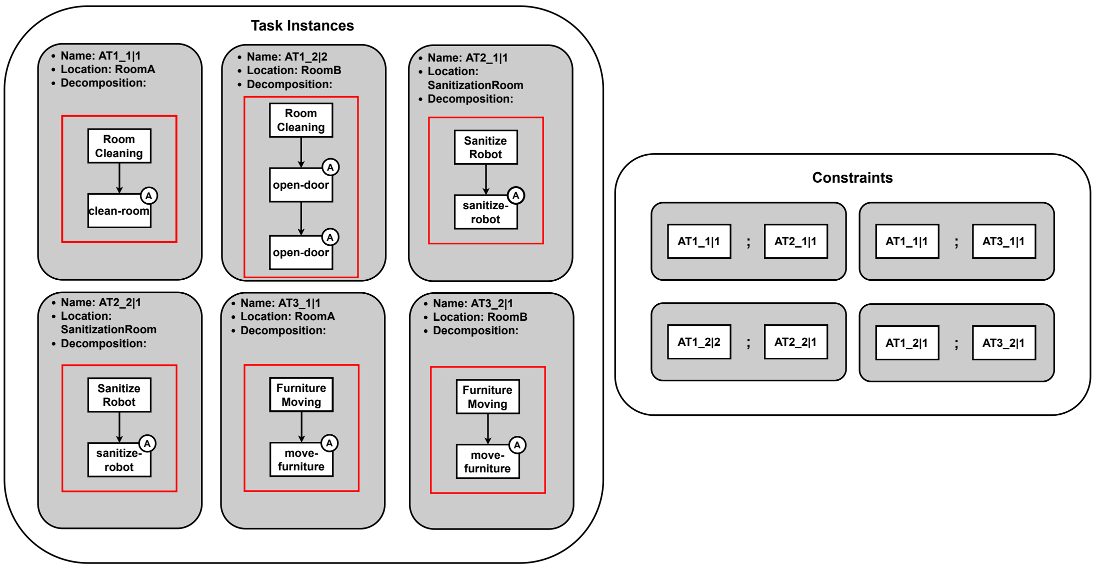
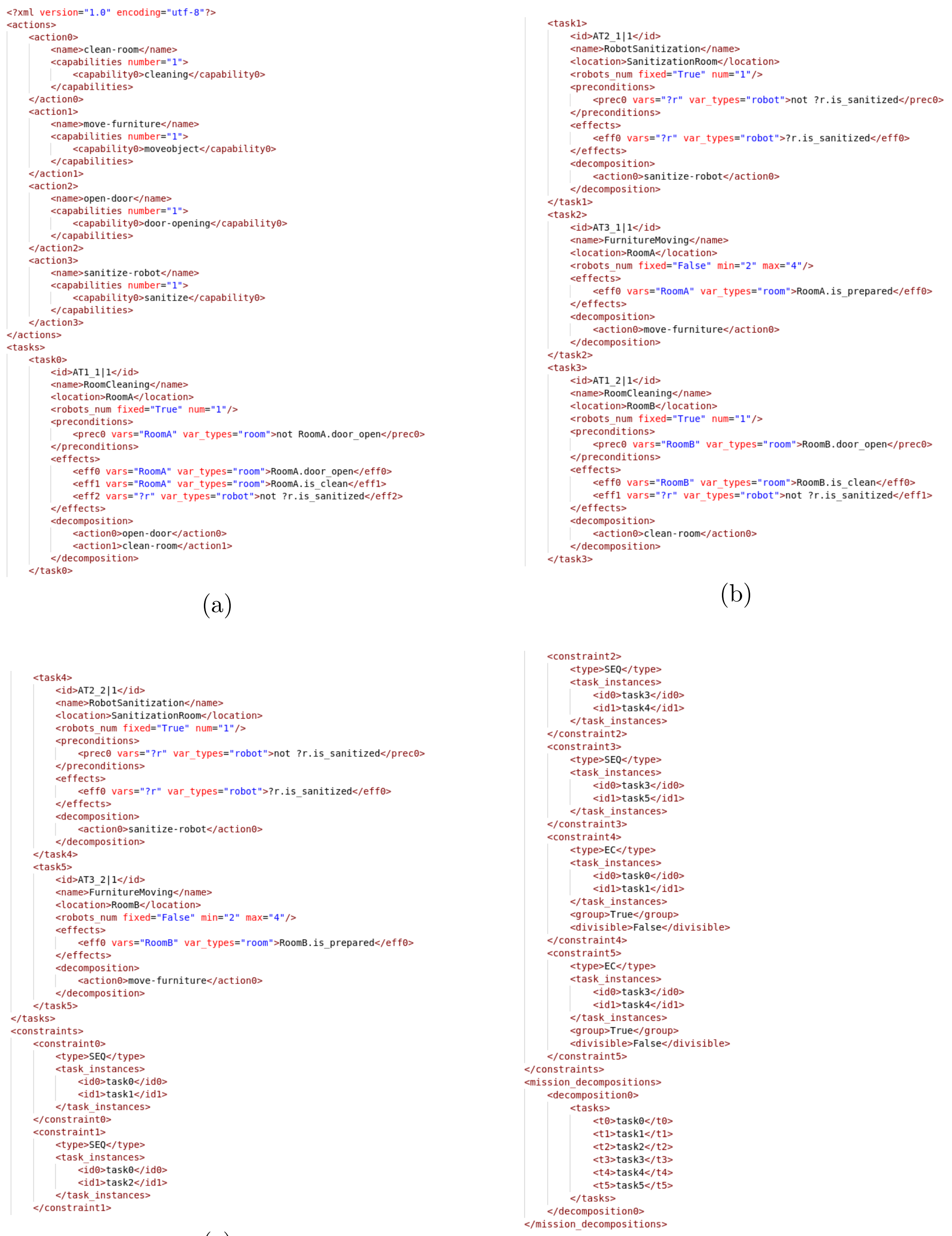


Figure 3.9: Valid mission decomposition for the Room Preparation example

As said in Section 3.4, the mission decomposition process generates a JSON/XML file with a lot of useful information for an allocation process to use and perform the best allocation possible. The output file for the *Room Preparation* example is shown in Figure 3.10 in an XML format.



```xml
<?xml version="1.0" encoding="utf-8"?>
<actions>
    <action0>
        <name>clean-room</name>
        <capabilities number="1">
            <capability0>cleaning</capability0>
        </capabilities>
    </action0>
    <action1>
        <name>move-furniture</name>
        <capabilities number="1">
            <capability0>moveobject</capability0>
        </capabilities>
    </action1>
    <action2>
        <name>open-door</name>
        <capabilities number="1">
            <capability0>door-opening</capability0>
        </capabilities>
    </action2>
    <action3>
        <name>sanitize-robot</name>
        <capabilities number="1">
            <capability0>sanitize</capability0>
        </capabilities>
    </action3>
</actions>
<tasks>
    <task0>
        <id>AT1_1|1</id>
        <name>RoomCleaning</name>
        <location>RoomA</location>
        <robots_num fixed="True" num="1"/>
        <preconditions>
            <prec0 vars="RoomA" var_types="room">not RoomA.door_open</prec0>
        </preconditions>
        <effects>
            <eff0 vars="RoomA" var_types="room">RoomA.door_open</eff0>
            <eff1 vars="RoomA" var_types="room">RoomA.is_clean</eff1>
            <eff2 vars="?r" var_types="robot">not ?r.is_sanitized</eff2>
        </effects>
        <decomposition>
            <action0>open-door</action0>
            <action1>clean-room</action1>
        </decomposition>
    </task0>
```

(a)

```xml
    <task1>
        <id>AT2_1|1</id>
        <name>RobotSanitization</name>
        <location>SanitizationRoom</location>
        <robots_num fixed="True" num="1"/>
        <preconditions>
            <prec0 vars="?r" var_types="robot">not ?r.is_sanitized</prec0>
        </preconditions>
        <effects>
            <eff0 vars="?r" var_types="robot">?r.is_sanitized</eff0>
        </effects>
        <decomposition>
            <action0>sanitize-robot</action0>
        </decomposition>
    </task1>
    <task2>
        <id>AT3_1|1</id>
        <name>FurnitureMoving</name>
        <location>RoomA</location>
        <robots_num fixed="False" min="2" max="4"/>
        <effects>
            <eff0 vars="RoomA" var_types="room">RoomA.is_prepared</eff0>
        </effects>
        <decomposition>
            <action0>move-furniture</action0>
        </decomposition>
    </task2>
    <task3>
        <id>AT1_2|1</id>
        <name>RoomCleaning</name>
        <location>RoomB</location>
        <robots_num fixed="True" num="1"/>
        <preconditions>
            <prec0 vars="RoomB" var_types="room">RoomB.door_open</prec0>
        </preconditions>
        <effects>
            <eff0 vars="RoomB" var_types="room">RoomB.is_clean</eff0>
            <eff1 vars="?r" var_types="robot">not ?r.is_sanitized</eff1>
        </effects>
        <decomposition>
            <action0>clean-room</action0>
        </decomposition>
    </task3>
```

(b)

```xml
    <task4>
        <id>AT2_2|1</id>
        <name>RobotSanitization</name>
        <location>SanitizationRoom</location>
        <robots_num fixed="True" num="1"/>
        <preconditions>
            <prec0 vars="?r" var_types="robot">not ?r.is_sanitized</prec0>
        </preconditions>
        <effects>
            <eff0 vars="?r" var_types="robot">?r.is_sanitized</eff0>
        </effects>
        <decomposition>
            <action0>sanitize-robot</action0>
        </decomposition>
    </task4>
    <task5>
        <id>AT3_2|1</id>
        <name>FurnitureMoving</name>
        <location>RoomB</location>
        <robots_num fixed="False" min="2" max="4"/>
        <effects>
            <eff0 vars="RoomB" var_types="room">RoomB.is_prepared</eff0>
        </effects>
        <decomposition>
            <action0>move-furniture</action0>
        </decomposition>
    </task5>
</tasks>
<constraints>
    <constraint0>
        <type>SEQ</type>
        <task_instances>
            <id0>task0</id0>
            <id1>task1</id1>
        </task_instances>
    </constraint0>
    <constraint1>
        <type>SEQ</type>
        <task_instances>
            <id0>task0</id0>
            <id1>task2</id1>
        </task_instances>
    </constraint1>
```

(c)

```xml
    <constraint2>
        <type>SEQ</type>
        <task_instances>
            <id0>task3</id0>
            <id1>task4</id1>
        </task_instances>
    </constraint2>
    <constraint3>
        <type>SEQ</type>
        <task_instances>
            <id0>task3</id0>
            <id1>task5</id1>
        </task_instances>
    </constraint3>
    <constraint4>
        <type>EC</type>
        <task_instances>
            <id0>task0</id0>
            <id1>task1</id1>
        </task_instances>
        <group>True</group>
        <divisible>False</divisible>
    </constraint4>
    <constraint5>
        <type>EC</type>
        <task_instances>
            <id0>task3</id0>
            <id1>task4</id1>
        </task_instances>
        <group>True</group>
        <divisible>False</divisible>
    </constraint5>
</constraints>
<mission_decompositions>
    <decomposition0>
        <tasks>
            <t0>task0</t0>
            <t1>task1</t1>
            <t2>task2</t2>
            <t3>task3</t3>
            <t4>task4</t4>
            <t5>task5</t5>
        </tasks>
    </decomposition0>
</mission_decompositions>
```

(d)

Figure 3.10: XML output file for the Room Preparation example

Chapter 4

An In-Depth Analysis of the Decomposition Process

In this section we aim to formalize the processes that were briefly explained in Chapter 3. We formalize the structures used in each step of the decomposition process in the order they were presented in the previous chapter. First, we start by defining the structures and steps in the HTN decomposition step, where we define the non-ground TDG and how do we generate the tasks paths of decomposition based on it. Afterwards, we go on to define the generation of abstract task instances and the Runtime Annotation Tree which are performed in the goal model expansion step. Then, we proceed to formally define the Task Graph, its properties and how it is generated in the Task Graph generation step. In addition, we define how this Task Graph is used alongside the world knowledge in order to generate the valid mission decompositions in the valid mission decompositions generation step. Finally, we also define how the Task Graph is used to generate the minimal set of constraints between tasks in the constraints generation step. With this, it is desired to provide an unambiguous description of the decomposition process, providing some induction proofs, mathematical definitions and pseudocodes.

4.1 HTN Decomposition

Before formalizing the HTN decomposition process we need to formalize what a non-ground Task Decomposition Graph (TDG) is. Inheriting from the definitions given by Bercher et al. [30] we have several (possibly modified) definitions to give:

- Abstract tasks consist of a single element $t(\overline{\tau})$ which consists in a parameterized name

- Primitive tasks (actions) consist of the 3-tuple $(t(\overline{\tau}), pre(\overline{\tau}), eff(\overline{\tau}))$ where $t(\overline{\tau})$ is a parameterized name, $pre(\overline{\tau})$ are preconditions and $eff(\overline{\tau})$ are effects
  - The latter two are conjunctions of literals and depend on the task's parameter variables $\overline{\tau}$
- A partial plan P consists of the 3-tuple $(PS, \prec, VC)$ where PS are the plan steps, ordering constraints $\prec$ and variable constraints VC
  - A plan step $l : t(\overline{\tau}) \in PS$ is a uniquely labeled task
  - The set $\prec$ is a strict partial order on PS
  - The set VC is defined over the variables $\overline{\tau}$ of the tasks of PS
- A method is a tuple $m = (t(\overline{\tau}), P_m, VC_m)$ that maps an abstract task $t(\overline{\tau})$ to its pre-defined partial plan P_m. VC_m denotes a set of variable constraints that relates the variables $\overline{\tau}$ of $t(\overline{\tau})$ with the variables of the partial plan P_m
- A hybrid planning domain is defined by $\mathcal{D} = (T_p, T_a, M)$ which contains the primitive and abstract tasks T_p and T_a, respectively, and a set of methods M
- A hybrid planning problem can be defined as $\mathcal{P} = (\mathcal{D}, P_{init}, C)$ which consists of a domain $\mathcal{D}$, an initial partial plan P_{init} and a set of constants C

With these definitions in hand, we define the non-ground TDG as in Definition 9. One thing to note about this definition is that the expression $v_m = NonGround_{VC_m \cup \{\overline{\tau}=\overline{c}\}}(P_m)$ means that for every variable in method $(t(\overline{\tau}), P_m, VC_m) \in M$, which is a decomposition method for task $v_t = t(\overline{c})$ and generates vertex v_m, we have:

- $\forall$ variable $v \in \overline{c}$ there exists a variable v' where $(v, v') \in VC_m$ or $v' = v$.
- There is some constant $c \in C$ where $(v, c) \in VC_m$ iff $\nexists\, v' \mid (v, v') \in VC_m$ or $v' = v$
- $\forall$ plan step $l : t(\overline{\tau}') \in P_m$ we have that $\overline{\tau}' \subseteq \overline{\tau}$.

With this definition we guarantee that for every method vertex $v_m \in V_m$ we only have variables (or constants) that relate to variables of the root task. Also, since every task vertex $v_t \in V_t \setminus TOP$ belongs to a partial plan of some method vertex, we guarantee that every variable in our task vertices also relate to variables in the root task. In short, every variable in the TDG relates to some variable in the root task, i.e., there is no addition of new variables in any vertex.

With all the definitions we establish the TDG and the decomposition process for each task present in the mission, which is represented by the goal model. Since we use a

TDG to decompose a single abstract task $\mathcal{T}$ we always have that the variables in the set of variables $\overline{\tau}$' of the TOP task are exactly the same as the ones defined in the set of variables $\overline{\tau}$ of $\mathcal{T}$ and that the method M which decomposes TOP decomposes it into $\mathcal{T}$. Thus, for each task in the goal model we create the non-ground planning problem $\mathcal{P} = (D, P_{init}, C)$ where we have that $P_{init} = (\{\mathcal{T}(\overline{\tau})\}, \emptyset, \emptyset)$ and:

- For $v_m = M = (TOP(\overline{\tau}'), P_m, VC_m)$ where $P_m = \{\mathcal{T}(\overline{\tau})\}$ and $VC_m = \emptyset$, since $\overline{\tau}' = \overline{\tau}$, we create a single task vertex where it holds:
  - $v_t = \mathcal{T}(\overline{\tau}) \in V_T$
  - $(v_m, v_t) \in E_{M \to T}$

If the same task is present twice in the goal model we have the same result in this process for both instances, since we only verify which task decompositions are valid given a world state in the valid mission decompositions generation step.

Definition 9 Let $\mathcal{P} = (\mathcal{D}, P_{init}, C)$ be a standard HTN planning problem with domain $\mathcal{D} = (T_p, T_a, M)$. Without loss of generality we assume that P_{init} contains just a single non-ground abstract task TOP for which there is exactly one method in M. The bipartite graph $\mathcal{G} = (V_T, V_M, E_{T\to M}, E_{M\to T})$, consisting of a set of task vertices V_T, method vertices V_M, and edges $E_{T\to M}$ and $E_{M\to T}$ is called the non-ground TDG of $\mathcal{P}$ if it holds:

1. **base case** (task vertex for the given task)
   $TOP \in V_T$, the TDG's root.

2. **method vertices** (derived from task vertices)
   Let $v_t \in V_T$ with $v_t = t(\overline{c})$ and $(t(\overline{\tau}), P_m, VC_m) \in M$. Then, for the method vertex $v_m = NonGround_{VC_m \cup \{\overline{\tau}=\overline{c}\}}(P_m)$ holds:
   - $v_t \in V_T$
   - $(v_t, v_m) \in E_{T\to M}$

3. **task vertices** (derived from method vertices)
   Let $v_m \in V_m$ with $v_m = (PS, \prec, VC)$. Then, for all plan steps $l : t(\overline{c}) \in PS$ with $v_t = t(\overline{c})$ holds:
   - $v_t \in V_T$
   - $(v_m, v_t) \in E_{M\to T}$

4. **tightness**
   G is minimal, such that 1. to 3. hold

4.1.1 Non-Groud TDG Generation

In this step we generate the non-ground TDG for each abstract task based on the structures generated in the parsing process. For each abstract task $\mathcal{T}(\overline{\tau})$ we generate the graph based on Definition 9 and the process for generating it is given further.

Abstract Task If we have an abstract task, as is the case of the abstract task $\mathcal{T}(\overline{\tau})$, we generate an abstract task vertex $v_t = \mathcal{T}(\overline{\tau})$, adding it to V_T and verify for each method $m = (t(\overline{c}), P_m, VC_m) \in M$ we continue the non-ground TDG generation process iff $t(\overline{c}) = \mathcal{T}(\overline{\tau})$. If the task vertex is being generated from a method vertex v_m, an AND edge is created and thus $E_{M \to T} = E_{M \to T} \cup (v_m, v_t)$.

Method If we have a method, we generate a method vertex v_m and add it to the set V_m. Since each method vertex will be generated from an abstract task vertex v_t we create an OR edge and thus $E_{T \to M} = E_{T \to M} \cup (v_t, v_m)$ For each plan step $l : t(\overline{\tau}') \in P_m$ we have two options:

- If $t(\overline{\tau}')$ is an abstract task we continue the non-ground TDG generation process for it iff there is no cycle between method m and task $t(\overline{\tau}')$. A cycle exists if the abstract task being considered was already defined in an abstract task vertex v_t that is a parent of v_m at some degree.
- If $t(\overline{\tau}')$ is a primitive task we add a task vertex $v_t = t(\overline{\tau}')$ and create an AND edge having $E_{M \to T} = E_{M \to T} \cup (v_m, v_t)$

4.1.2 Task Decomposition

In order to generate the paths of decomposition of an abstract task TDG's we traverse the TDG in a depth-first fashion. Remembering that edges in $E_{M \to T}$ are OR edges and $E_{T \to M}$ are AND edges we have three possible behaviors which will be further explained. First of all, let us start by defining what a decomposition path is using Definition 10.

Definition 10 A decomposition path can be defined as the tuple $DP = (D)$ where D is set a of primitive tasks t. For a given non-ground TDG represented by $\mathcal{G} = (V_T, V_M, E_{T \to M}, E_{M \to T})$ we have that $\forall t \in D : v_t = t \in V_T$ and $\nexists\, v_m \mid (v_t, v_m) \in E_{T \to M}$

We also need to define two other things: (i) the generated paths set $\mathcal{GP}$, which consists of a set of decomposition paths, and (ii) the world state set $\mathcal{WS}$, which consists of a set of non-ground predicates. Both are initially empty and are instantiated as the decomposition process is performed. Also, in this approach method preconditions are encoded as actions with no effects, preconditions equal to the given method's preconditions and which come before any other action in the given method's partial plan.

Method Vertices If we find a method vertex we verify its possible orderings given its partial plan P_m and the set of ordering constraints $\prec$ of this partial plan. These possible orderings are basically all of the partial plans P_m' that can be generated from P_m given the ordering constraints $\prec$, where the set of possible ordering is defined as $\mathcal{PM}$. Then, for each plan step $l : t(\overline{c}) \in P$' where P' $\in \mathcal{PM}$ we:

1. Rename the variables of each plan step $l : t(\overline{c})$ based in VC_m
2. Check if preconditions are met based on the world state $\mathcal{WS}$

    a If preconditions are met we continue the decomposition process for this plan step and retrieve its generated paths

    b If preconditions are not met, then ordering P' is not executable and we check the next ordering

When all of the partial plans in $\mathcal{PM}$ are checked we verify their generated paths and add each one of them to the set of generated paths $\mathcal{GP}$. Mathematically, naming the set of child generated paths as $\mathcal{CP}$, we have that $\forall C \in \mathcal{CP} : \mathcal{GP}$' $= \mathcal{GP} \cup P \ \forall P \in C$. This is the case because each child can generate multiple decomposition paths, so C represents the set of generated paths for each child of the method.

There are some things to note about the world state in this decomposition process. First, each ordering in $\mathcal{PM}$ has the same initial world state $\mathcal{WS}$. When checking the orderings we have for each task t in each ordering P' $\in \mathcal{PM}$ that we update the world state in two ways. The first one is by primitive task effects, which will be further explained. The second is in the precondition checking process where, if we call the set of preconditions of a child task t as $CPrec$, we have that $\forall prec \in CPrec : \mathcal{WS}$' $= \mathcal{WS} \cup p$ iff $p \notin \mathcal{WS}$.

Abstract Task Vertices If we find an abstract task vertex we go through all of its children, which are methods, rename the variables based on the variable constraints and call the decomposition process on each of them. When finished, we simply get the set of decomposition paths for each of them and add to $\mathcal{GP}$. The generated path for a primitive task simply consists of this task, which will be returned to the parent node that called the decomposition process on this vertex.

Primitive Task Vertices When we find a primitive tasks we rename the variables based on the variable constraints and apply the primitive task effects. If the effect predicate is already present in $\mathcal{WS}$, we check if the effect performs some change in it and, if it does, we apply the change. If the effect predicate is not present in $\mathcal{WS}$, we simply add it accordingly.

The Cyclic Decomposition Problem Cyclic decompositions are allowed in the decomposition process of the framework but there are some important details to consider first. Let us start with a simple example of a storage domain where object picking tasks are performed, shown in Listing 4.1.

```
(define (domain storage)
    (:types room - object)
    (:functions
        (objects ?rm - room)
    )
    (:capabilities pickobject)

    (:task PickObject :parameters (?r - robot ?rm - room))
    (:method object-pick-multiple
        :parameters (?r -robot ?rm - room)
        :task (PickObject ?r ?rm)
        :precondition (and
            (> (objects ?rm) 1)
        )
        :ordered-subtasks (and
            (pick-object ?r ?rm)
            (PickObject ?r ?rm)
        )
    )
    (:method object-pick-single
        :parameters (?r - robot ?rm - room)
        :task (PickObject ?r ?rm)
        :precondition (and
            (= (objects ?rm) 1)
        )
        :ordered-subtasks (and
            (pick-object ?r ?rm)
        )
    )
    (:action pick-object
        :parameters (?r - robot ?rm - room)
        :required-capabilities (pickobject)
        :precondition ()
        :effect (and
            (decrease (objects ?rm) 1)
        )
    )
)
```

Listing 4.1: HDDL storage domain example

One can verify that a decomposition of the task *PickObject* will depend on the number of objects to be picked in a storage room, represented by the (objects ?rm) function HDDL function. In the current state of the framework a specification must conform to certain conditions in order for the decomposition to be correctly performed. An expansion is detected if a method contains a greater than precondition and:

- Has a cycle with its parent task, as is the case of the method *object-pick-multiple* and the *PickObject* task
- All of the tasks that have effects on the function that created the need for an expansion. In the storage example, this is the *objects* function over the *?rm* variable of the *object-pick-multiple* method which maps to the *?rm* variable of the *PickObject* task. In this sense, We can verify that the only task that performs an operation over the *objects* on the storage example is the *pick-object* action, which is its direct children.
  - The *pick-object* which will be a child of the *object-pick-single* will not be present in the cycle, so this is not a problem
- Every effect related to the function that created the need for an expansion must be a decrease operator

The example shown in Listing 4.2 would configure a wrong specification of a cyclic decomposition, where the errors are highlighted in red.

```
(define (domain storage)
    (:types room - object)
    (:functions
        (objects ?rm - room)
    )
```

```
(:capabilities pickobject)

(:task RetrieveObject :parameters (?r - robot ?rm - room))
(:method retrieve-object
     :parameters (?r - robot ?rm - room)
     :task (RetrieveObject ?r ?rm)
     :precondition ()
     :ordered-subtasks (and
          (pick-object ?r ?rm)
     )
)

(:task PickObject :parameters (?r - robot ?rm - room))
(:method object-pick-multiple
     :parameters (?r -robot ?rm - room)
     :task (PickObject ?r ?rm)
     :precondition (and
          (> (objects ?rm) 1)
     )
     :ordered-subtasks (and
          (RetrieveObject ?r ?rm)
          (PickObject ?r ?rm)
     )
)
(:method object-pick-single
     :parameters (?r - robot ?rm - room)
     :task (PickObject ?r ?rm)
     :precondition (and
          (= (objects ?rm) 1)
     )
     :ordered-subtasks (and
          (pick-object ?r ?rm)
     )
)

(:action pick-object
     :parameters (?r - robot ?rm - room)
     :required-capabilities (pickobject)
     :precondition ()
     :effect (and
                    (assign (objects ?rm) 1)
                )
            )
)
```

Listing 4.2: HDDL storage domain example with specification errors

We deal with this problem by extending the decomposition path definition given in Definition 10, where this extended definition is given in Definition 11.

Definition 11 A decomposition path can be defined as the tuple $DP = (D, NE, ED, FP)$ where:

- D is set a of primitive tasks t. For a given non-ground TDG represented by $\mathcal{G} = (V_T, V_M, E_{T\rightarrow M}, E_{M\rightarrow T})$ we have that $\forall t \in D : v_t = t \in V_T$ and $\nexists\, v_m \mid (v_t, v_m) \in E_{T\rightarrow M}$
- NE is a boolean flag which states if the path needs expansion
- ED is a number (integer or float) which defines the decrease given in the world state for each cycle
- FP is the fragments to expand definition. Each fragment to expand $F \in FP$ consists in the tuple $F = (I, Pred)$ where:
  a I is a pair (i, f) which represent the first and last task index in the decomposition path that represent a cyclic fragment, respectively
  b Pred is the predicate that generates the cycle, like explained previously

In the example shown in Listing 4.1, the decomposition path that represents a cyclic decomposition would be:

D = [___method_precondition_object_pick_multiple, pick-object, ___method _precondition_object_pick_single, pick-object]
NE = True
ED = 1
FP = ((0,1),objects ?rm))

The tasks that start with "___method_precondition_" are the method preconditions encoded as actions. Assuming that we have a room *roomA* with 3 objects, which would evaluate to "objects roomA = 3", we would have the following path after expansion: ___method_precondition_object_pick_multiple → pick-object → ___method_precondit ion_object_pick_multiple → pick-object → ___method_precondition_object_pick_sin gle → pick-object

4.2 Goal Model Expansion

Before formalizing the goal model expansion steps, we need to provide a small formalization on the only structure that is able to expand the goal model: the forall expression used in universal Achieve goals. In order to do this, we perform an induction on the number of deep copies of the universal Achieve goal that contains the forall statement that are created.

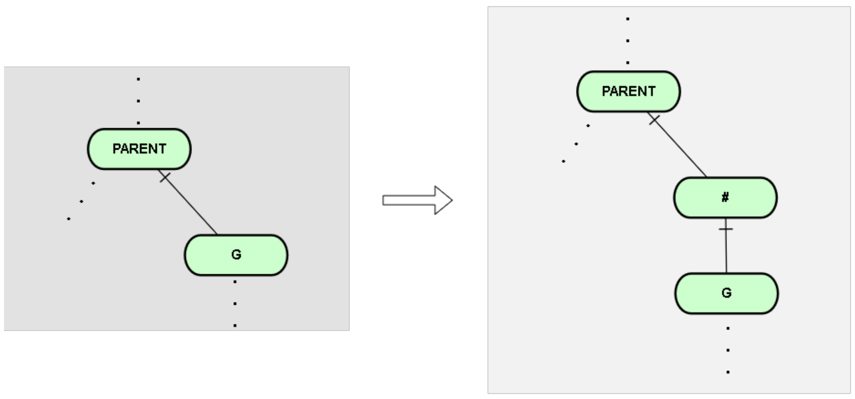


Figure 4.1: Generation of dummy parallel node for universal Achieve goal

First, assume that when an universal Achieve goal G with a forall statement is found a dummy parallel goal is created between the Achieve Goal and its parent represented by Parent(G) (as shown in Figure 4.1), since we know that forall statements depend on previously instantiated variables and thus this Achieve goal will never be the goal model root. Also, we define that Children(G) represents the number of children for a node in the goal model. Finally, we have that the creation of deep copies of goal G as children of the # operator can be specified in the code-like syntax of Algorithm 1:

Algorithm 1 Instantiation of deep copies of an universal Achieve goal

```
for $index = 0; index < c.size() - 1; index++$ do
    instantiateChild(#,G)
end for
```

Where *instantiateChild* is a function that takes the parent as first input and the node to be deep copied (alongside all of its children) as the second. This deep copy of the second node is appended as a child of the first node which gives us that for every execution of instantiateChild we have Children(#) = Children(#) + 1. With this in mind, we define the Lemma 4.2.1:

Lemma 4.2.1. *For every collection variable c with size n greater or equal to 1, we have that Children(#) = n for the parallel operator # generated by an universal Achieve goal G.*

Proof. Since this is an inductive proof, we need to provide the base case, which in this case is when n equals 1, and the inductive step.

- Base case (n=1): Assume Children(#) = 1 initially, as shown in Figure 4.1. If c has size n = 1, no execution of instantiateChild is performed and thus Children(#) = 1 = n.

- Inductive step (n=k): For Children(#) = 1 initially and c with size n = k we have k-1 executions of instantiateChild. Therefore, we have at the kth step Children(#) = 1 + k - 1 = k = n. For a variable c of size n = k+1 we would have one more execution of instantiateChild, so: Children(#) = k + 1, which equals n.

□

With this formalization in hand we can then proceed to formalize the two main steps, which are: (i) Abstract Task Instances Generation and (ii) Runtime Annotation Tree Generation.

4.2.1 Abstract Task Instances Generation

The formalization for this step will be on the number of instances generated for each task in the goal model. Since the only way to expand the goal model is by means of forall statements, we discard the trivial case where no forall statement is present and where each task has exactly one instance. We want to show that the number of instances generated for each task is related to the number and size of the forall statements present in the nodes that belong to the goal model path it belongs to.

Lemma 4.2.2. *The number of instances generated for a given task is $m = i_1 \times i_2 \times i_3 \times ... \times i_n$ where n is the number of forall statements in the path it belongs to and i_j is the number of instances generated by the jth forall statement.*

Proof. Since this is an inductive proof, we need to provide the base case, which in this case is when n equals 1, and the inductive step.

- Base case (n=1): Assume there is only one forall statement and this forall statement is present in a node that is a parent of task t at some level in the goal model tree. Suppose this forall statement generates i copies of its children and their children and so on, which is represented by the instantiateChild function. Since task t is a child of this node, we have i copies of t (which is a single instance) and thus $m = i \times 1 \implies m = i$

- Inductive step (n=k): Assume we have n = k forall statements, all of them present in nodes that are parents of task t at some level. Each forall statement generates i_j instances where $j \in [1, k]$. Since their order is not important when it comes to the number of generated instances, we have that the number of instances is $m = i_1 \times i_2 \times i_3 \times ... \times i_k$. For n = k+1 we add another forall statement in a node that is a parent of task t at some level. This new forall generates ik+1 copies of each instance that already exists. Since we have m instances, the new number of instances is $m' = i_{k+1} \times m \implies m' = i_1 \times i_2 \times i_3 \times ... \times i_k \times i_{k+1} = m' = i_1 \times i_2 \times i_3 \times ... \times i_{k+1}$.

$\square$

4.2.2 Runtime Annotation Tree Generation

There are some important considerations to be made before formalizing the Runtime Annotation Tree generation process:

- Nodes (mainly operator nodes) encapsulate all of the necessary information like: (i) their content, (ii) their children, (iii) their parent, (iv) their related goal model goal node from which they were generated, if applicable, (v) their execution related constraints, by means of the group and divisible attributes, and (vi) if they refer to a OR-decomposed node or not

- For every goal node we set the group and divisible attributes equal to their values in the GM

- When a goal is parsed, we generate the runtime annotation for it with its children as Goal/Task nodes to be further refined. Runtime annotations with multiple operators, which must be all of the same type, are flattened. Since we build the runtime annotation recursively using semantic and lexical analysis, only one operator can exist for each node in the goal model. With this in mind, assuming an operator *op* we perform the flattening as in Algorithm 2. One has to note that operators are always binary in the GM so they only have two children. An illustration of the flattening process is given in Figure 4.2, where OP1 and OP2 are of the same type.

- Query goals variable instantiation is taken into consideration during this process. They are used in order to define universal achieve goals expansions, i.e., to verify how many instances are generated from a forall statement

Algorithm 2 Flattening of runtime annotations

```
$finalChildren \leftarrow []$

for child in op.children do
    if child.type == OPERATOR then
        for grandchild in child.children do
            finalChildren.pushBack(grandchild)
        end for
    else
        finalChildren.pushBack(child)
    end if
end for
```

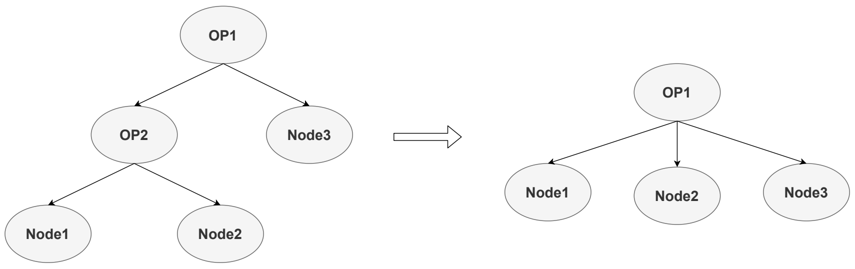


Figure 4.2: Illustration of the runtime annotation flattening process

- After the process is finished we use the AT instances generated in the AT instances generation process and replace task node contents, which previously referred to the same task ID, accordingly

We initialize the formalization by explaining the basic functioning of the process. In order to generate the Runtime Annotation Tree we perform a depth-first visit on the goal model where:

- If we find a leaf node, which can be a task or a query goal, we simply add it to the children list of their parent with no further operations on it. Query goals are added with the GOAL type and task nodes are added with the TASK type

- If we find a non-leaf node, which can be either achieve/perform goals with multiple children or means-end achieve/perform goals, we may have different possible cases. These cases will be further discussed, with the only note being that for each operator node generated from a goal node we link them by means of the related goal attribute

For non-leaf nodes we have three possible cases we can found: (i) Universal Achieve goals, (ii) Multiple children goal nodes and (iii) means-end (only child) goal nodes. These types are detailed next.

Universal Achieve Goals This case builds upon the forall formalization. For each universal Achieve goal found we have two cases:

A. If the Achieve goal generates multiple instances (2 or more), a parallel operator (#) node is created with multiple copies of this Achieve goal as its children, with the difference that they are transformed into non-universal Achieve goals. These Achieve goals will be further expanded following the rules for transformation of goal nodes. It is important to note that the generated parallel operator: (i) has no related goal, (ii) has no execution constraints (i.e. group is true and divisible is true) and (iii) is always an AND decomposition

B. If only one instance is created, the parallel operator node is omitted and the Achieve goal node is transformed into a non-universal Achieve goal node

We must note that the generated (or transformed) non-universal Achieve goal nodes (or node) are checked afterwards as either a multiple children or only child goal nodes.

Multiple Children Goal Nodes If a goal has multiple children we check it has a runtime annotation. This leads us with two cases:

A. If it does not have a runtime annotation, a parallel operator is assumed and a parallel operator node is created. Its children are instantiated based on the goal model, being either a goal or a task node with its content equal to its ID, in order to be further transformed

B. If it has a runtime annotation, an operator node with the corresponding runtime annotation operator as content is created. Its children are defined by means of the flattening process which was previously explained, but in the end they correspond to the goal model children of this node. The children in this case are also goal or task nodes with their contents set to their goal model IDs

Means-end Goal Nodes If a goal has only one child. its content is set to the ID of that goal in the goal model and it is transformed into a MEANSEND type node. This child is set to be a goal or task node, depending on which one it corresponds to in the goal model, and its content is set to the ID of that node in the goal model.

4.3 Task Graph Generation

The Task Graph is generated from the Runtime Annotation Tree, generated in the goal model expansion process. The Task Graph was previously defined in Definition 7, where it is important to note that in the same way each node in the Runtime Annotation Tree relates to one node in the goal model we have that in the Task Graph this property also holds, with the exception of decomposition nodes.

There is one important consideration to be made before formalizing the Task Graph generation process, which is the definition of a unique branch Task Graph. A Task Graph TG is said to be a unique branch Task Graph if it represents a goal model that in the end turns out to have only one path to follow. We can represent this in the following way:

$$UniqueBranch(TG) = true\ iff\ \forall n \in OP \cup G,\ OutEdges(n) = 1$$

where OutEdges(n) is the number of AND and OR edges where n is the source.

With these concepts in mind we can proceed to formalize the four main steps, which are: (i) Initial Task Graph Building, (ii) Final Context Dependency Edges Generation (iii) Execution Constraint Edges Generation and (iv) Task Graph Trimming.

4.3.1 Initial Task Graph Building

Using the Runtime Annotation Tree and the goal model we generate the initial Task Graph parsing the Runtime Annotation Tree in a DFS fashion. The rules for performing this step are further detailed.

Operator or Means-End Nodes When a Runtime Annotation Tree OPERATOR or MEANSEND type node is found we have two possibilities:

A. If we have an OPERATOR node that relates to a goal in the goal model or a MEANSEND node we know that we are not dealing with a forall generated node. In this case we:

   a. Check if the related goal has a context and verify if it is active or not using the initial world state

   b. Check if the related goal is of Achieve type and get its achieve condition

B. If we have an OPERATOR node that does not relate to any goal in the goal model we know that this is a forall generated node. In this case we simply get its achieve condition

After that we add the node to the Task Graph, filling the necessary properties based on what we have from the Runtime Annotation Tree and if it is a forall generated node or not. If this is not the first node added to the Task Graph we have a parent node from which we create an edge to the current node, which can be an AND or OR edge depending if the parent's runtime annotation relates to an OR-decomposed node or not. With the node added to the Task Graph we check if any context condition was checked and what was the result. If the context condition was not satisfied we perform checking against the previously inserted nodes to verify if any of them satisfies the context condition. This process is described in Algorithms 3 and 4, where we create a context dependency edge if we have a task that satisfies the context and is involved in a parallel decomposition with the current node.

Algorithm 3 Context condition checking algorithm

```
function ContextChecking(currentNode)
    $TG \leftarrow TreeLikeTaskGraph(currentTG)$
    $currentParent \leftarrow currentNode.parent$
    $visitedNodes \leftarrow new\ set()$
    $isParallel \leftarrow false$

    contextSatisfied ← RecursiveCtxChecking(originalNode, currentParent, TG, visitedNodes, isParallel)
    visitedNodes.insert(currentNode)

    return contextSatisfied
end function
```

Algorithm 4 Recursive context condition checking algorithm

```
function RecursiveCtxChecking(originalNode, currentNode, TG, visitedNodes, parallelParent)
    if currentNode == TG.root.parent then
        return false
    end if

    if currentNode.type != TASK then
        $isOr \leftarrow IsOrDecomposed(currentNode)$
        $isParallel \leftarrow IsParalell(currentNode)$
        $isFallback \leftarrow IsFallback(currentNode)$
        $ctxSatisfied \leftarrow false$

        if isOr || isFallback then
            if currentNode != TG.root then
                visitedNodes.insert(currentNode)
                return RecursiveCtxChecking(originalNode, currentNode.parent, TG,
                visitedNodes, false)
            else
                return false
            end if
        end if

        $parallelChecking \leftarrow parallelParent \mid\mid IsParallel$

        for child in currentNode.children do
            if !visitedNodes.find(child) then

                ctxSatisfied ← RecursiveCtxChecking(originalNode, child, TG, visit-
                edNodes, parallelChecking)
                visitedNodes.insert(child)

                if contextSatisfied then
                    break
                else
                    visitedNodes.insert(currentNode)

                    return RecursiveCtxChecking(originalNode, currentNode.parent,
                    TG, visitedNodes, false)
                end if
            end if
        end for

        return contextSatisfied
    else
        taskSatisfiesCtx ← false
        for decomposition in currentNode.decompositions do
            $contextSatisfied \leftarrow CheckContext(decomposition)$

            if ctxSatisfied then
                if parallelParent then
                    AddContextDependencyEdge(decomposition, originalNode)
                end if
                taskSatisfiesCtx ← true
            end if
        end for

        return taskSatisfiesCtx
    end if
  end function
```

One important note is that the *TreeLikeTaskGraph* function returns the Task Graph without CD and EC type edges, which is in fact a tree. Also, every Task Graph node's children vector is already ordered from left to right, which makes DFS traversing as easy as simply calling the process on children recursively. After checking the context we have two possible behaviors. If the context was satisfied we instantiate the necessary variables based on variable mappings produced by previous processes, call the building process on the children and return to its parent's building process, if any. If the context could not be satisfied we erase the node from the Task Graph and then return to the building process of its parent, if any.

Task Nodes When a Runtime Annotation Tree TASK node is found we simply add the node to the Task Graph, filling the necessary properties based on what we have from the Runtime Annotation Tree. Also, we create an edge from the parent node of this task to it which can be either an AND or OR edge, depending if the parent's runtime annotation relates to an OR-decomposed node or not.

Then we proceed to add the decomposition nodes to this task based on the possible task decomposition paths generated by the HTN decomposition process. Each decomposition node is added with an unique ID consisting of the task ID + |{D_ID} where {D_ID} is an integer that starts in 1 and is incremented for each decomposition we have. After that, we simply return to the task parent node and continue its building process.

4.3.2 Final Context Dependency Edges Generation

When the initial Task Graph is generated we proceed to generate the final context dependency links, which will consist of context dependency links where the source is a DECOMPOSITION node and the target is a TASK node. This is done as described in Algorithm 5.

As for the functions used we have that DFS-Visit generates a vector of the Task Graph TG nodes in a DFS fashion starting from a specific node, which in this case is the current edge's target. This DFS visit considers only AND and OR edges, since we have a tree using this approach. The AddEdge function simply adds in the Task Graph TG an edge

Algorithm 5 Final context dependency edges generation procedure

```
for $e \in E$ do
    if e.type == CD then
        for node in DFS-Visit(e.target,TG) do
            if node.type == TASK then
                AddEdge(e.source,node,CD,TG)
            end if
        end for

        RemoveEdge(e,TG)
    end if
end for
```

of a certain type, which is CD in this case, between two nodes, which are the current edge's source and the TASK node that is a child of the current edge's target in some degree in this case. Finally, RemoveEdge simply removes the an edge, which in this case is the current edge, from the Task Graph TG.

4.3.3 Execution Constraint Edges Generation

In order to generate the execution constraint edges we also go through the graph in a DFS fashion, using the DFS-Visit approach previously mentioned where only AND and OR edges are considered and the graph turns into a tree. We keep track of:

- The current active tasks
- The active tasks for each nodes that have execution constraints
- The nodes that have execution constraints and their attributes (group and divisible). These are kept in a stack
- The current node that is generating execution constraints, which is -1 when we don't have one
- The OR decomposed nodes

The basic idea is to keep track of the active execution constraints and for each task that is found create two edges in opposite directions with all of the tasks that are involved in this execution constraint. It is important to note that tasks that are OR-decomposed cannot have constraints between them and children of different instances of the same node in a forall expansion can only have constraints if they were defined in a parent of the forall node. The routine to create execution constraint edges is shown in Algorithm 6, where 0 is the ID of the root.

Algorithm 6 Execution constraint edges generation routine

```
$currentActiveTasks \leftarrow []$
$orNodes \leftarrow new\ set()$
$constraintNodes \leftarrow new\ stack()$
$nodeActiveTasks \leftarrow new\ map()$
$currentConstraintNode \leftarrow -1$

for node in DFS-Visit(0,TG) do
    if currentConstraintNode != -1 then
        if IsParentOf(node.parent,currentConstraintNode.parent) then
            if constraintNodes.size() == 0 then
                currentActiveTasks.clear()
                currentConstraintNode = -1
            else
                currentConstraintNode = constraintNodes.top()
                constraintNodes.pop()
            end if
        end if

        if (node.parent.isForAll && !IsParentOf(node.parent,currentConstraintNode))
         || orNodes.find(node.parent) then
            currentActiveTasks = nodeActiveTasks[node.parent]
        end if
    end if

    if IsOrDecomposed(node then
        orNodes.insert(node)
    end if

    if IsOrDecomposed(node) || node.isForAll then
        nodeActiveTasks[node] = currentActiveTasks
    end if

    if node.type == GOAL || node.type == OP then
        if !node.group || (node.group && !node.divisible then
            if currentConstraintNode == -1 then
                constraintNodes.push(node)
                currentConstraintNode = node
            else
                if currentConstraintNode.group then
                    currentConstraintNode.group = node.group
                    currentConstraintNode.divisible = node.divisible

                    constraintNodes.push(node)
                end if
            end if
        end if
    else if node.type == TASK then
        for task in currentActiveTasks do
            if !ExistsEdge(node,task,EC,TG) then
                AddBidirectionalExecConstraintEdges(node, task, EC, TG, currentCon-
                straintNode.group, currentConstraintNode.divisible)
            end if
        end for
    end if
end for
```

As for the functions we have that IsParentOf receives two nodes and checks if the firts is a parent of the second at some degree. In addition, IsOrDecomposed receives a single node and checks if it is and OR-decomposed node or not. The ExistsEdge function checks if there is an edge of a specific type between two nodes in a given Task Graph. Finally, AddBidirectionalExecConstraintEdges adds two execution constraint edges between two nodes in a given Task Graph in opposite directions, instantiated with the given group and divisible attribute values.

4.3.4 Task Graph Trimming

As previously established, the trimming process removes any nodes that do not satisfy the Minimal Task Graph condition, as defined in Equation 3.1. The routine to trim the Task Graph is, where TrimNode removes a node from a specific Task Graph, replaces it with its child and updates the ID's of all necessary nodes accordingly.

4.4 Valid Mission Decompositions Generation

For the valid mission decompositions generation process we need the Task Graph, as defined in Definition 7. Also, we need the set of world states S, which is defined in

Algorithm 7 Task Graph trimming routine

```
for node in TG do
    if node.type == GOAL then
        TrimNode(node,TG)
    else if node.type == OP then
        if OutEdges(node) == 1 && !IsAchieve(node) then
            TrimNode(node,TG)
        end if
    end if
end for
```

Definition 12.

Definition 12 The set of states S can be defined as the tuple S = (BS,FS,I) where:

- BS is the set of boolean states
- FS is the set of function states
- I is the initial state

It is important to note that $S = BS \cup FS$ and $I \subset S$.

We can also define the what a decomposition consists in, since it is a very important structure in this process. The definition of a decomposition is given in Definition 13. We must note that boolean preconditions and effects are expressed by means of boolean predicates. In this work we will express a boolean predicate as a tuple B = (pred,pos) where: (i) pred is a ground predicate (i.e. one where variables were replaced) and (ii) pos is a boolean flag which when set to true indicates that the predicate is positive and when set to false indicates that the predicate is negative. Function preconditions and effects, on the other hand, are expressed by means of function predicates. Function predicates are expressed as a tuple F = (pred,val,op) where: (i) pred is a ground predicate, (ii) val is a number value (float or integer) and (iii) op is the operation. For preconditions we have $op \in [=, !=, >]$ and for effects we have $op \in [assign\ (=), increase\ (+), decrease\ (-)]$. Boolean states are represented in the same way as boolean preconditions and effects. Function states, on the other hand, are represented as a tuple (pred,val) where pred is a ground predicate and val is a numeric value. In addition, we also define the set of valid mission decompositions as in Definition 14.

Definition 13 A decomposition d can be defined as the tuple D = (Pre,Eff,Fpre,Feff) where:

- Pre is the set of boolean preconditions
- Eff is the set of boolean effects
- Fpre is the set of function preconditions
- Feff is the set of boolean preconditions

In addition, remembering that D is the set of decompositions in a Task Graph we can define the following functions:

- Pre(d) are the sets of boolean preconditions for each decomposition $d \in D$
- Fpre(d) are the sets of function preconditions for each decomposition $d \in D$
- Eff(d) are the sets of boolean effects for each decomposition $d \in D$
- Feff(d) are the sets of function effects for each decomposition $d \in D$

Definition 14 The set of valid mission decompositions is named P. A valid mission decomposition $p \in P$ can be defined as the tuple p = (PD,PS) where:

- PD are the task decompositions involved in the valid mission decomposition
- PS is the resulting world state, where for each decomposition in PD we apply its effects in the initial state I taking in consideration the constraints between them

Some important concepts need to be given before the formalization of the valid mission decompositions generation process which are:

- The initial state of a specific node N (or n) is represented by I(N) (i(n)). The end state is represented by E(N) (e(n))
- When a state S satisfies some precondition Pre it is represented by $S \vdash Pre$. If it does not satisfy we have $SnvdashPre$

- Given a node N, for valid mission decompositions we have the following:

    a P'(N) is the set after checking the node and all of its children

    b P*(N) are the paths generated by the node, given some initial state i(N). This is only used with task nodes

    c P(N) is the set before checking the node and all of its children

- Valid mission decompositions operations are:

    a The parallel combination operator $\otimes$. This is the operator used for combining parallel decomposed paths. Assume two valid mission decomposition sets P1 and P2, which started with the same initial valid mission decomposition set P. Naming $P1 \otimes P2$ as P3 we have the following behavior:

$$\begin{aligned}
&\forall p1 \in P1 \text{ and } p2 \in P2, \text{we generate a path } p3 \in P3 \text{ where} \\
&\quad p3.PD = p1.PD \cup (p2.PD \setminus p1.PD) \\
&\quad p3.PS = p1.PS \cup p2.PS \text{ such that} \\
&\quad \forall s' \in p1.PS \nexists\, s'' \in p2.PS \mid \begin{cases} s''.pred = s'.pred \text{ and } s''.pos \neq s'.pos, \text{if } s' \in BS \\ s''.pred = s'.pred \text{ and } s''.val \neq s'.val, \text{if } s' \in FS \end{cases}
\end{aligned}$$

    b The sequential combination operator $\circ$. This is the operator used for combining sequentially decomposed paths. Assume two valid mission decomposition sets P1 and P2, where P1 is the initial valid mission decomposition set for the generation of P2. Naming $P1 \circ P2$ as P3 and given that the initial state for the generation of p2.PS is given by p1.PS we have the following behavior:

$$\begin{aligned}
&\forall p1 \in P1 \text{ and } p2 \in P2, \text{we generate a path } p3 \in P3 \text{ where} \\
&\quad p3.PD = p1.PD \cup (p2.PD \setminus p1.PD) \\
&\quad \forall s \in p3.PS \text{ we have : } \begin{cases} s = s' \;\forall\; s' \in p2.PS \\ s = s'' \;\forall\; s'' \in p1.PS \setminus p2.PS \end{cases}
\end{aligned}$$

For this process we go through the Task Graph in a DFS fashion and we have three possible cases: (i) decomposition nodes, (ii) task nodes and (iii) operator nodes. We formalize this process using a recursive evaluation on these three cases as follows.

4.4.1 Decomposition Nodes

The base case is given by decomposition nodes. For a decomposition $d \in D$ and an initial state i(d) we have the following end state e(d):

$$e(d) = i(d) \cup Eff(d) \cup Feff(d), \text{iff } i(d) \vdash Pre(d) \text{ and } i(d) \vdash Fpre(d)$$

where:

If $s' \in Eff(d)$ and $\exists\ s \in i(d) \mid s.pred = s'.pred$ and $s.pos \neq s'.pos\ then\ s' \ \in\ e(d)\ and\ s\ \notin\ e(d)$

If $s'\ \in Feff(d)\ and\ \exists\ s \in i(d) \mid s'.pred = s.pred$ then $s'' \in e(d)$ is given by:

$$\begin{cases} s'' = (s'.pred, s'.val), \text{ if } s'.op = \text{assign} \\ s'' = (s'.pred, s.val + s'.val), \text{ if } s'.op = increase \\ s'' = (s'.pred, s.val - s'.val), \text{ if } s'.op = decrease \end{cases}$$

4.4.2 Task Nodes

The next case is composed of tasks where we may have multiple possible decompositions. For every task $t \in AT$ we have two possible behaviors:

A If $P = \emptyset$ then for every decomposition $d \in D$ which is a child of t we create a path p where:

$$p.PD = d \text{ and } p.PS = e(d)$$

in which e(d) is the final state of d given the initial state i(d). If $i(d) \vdash Pre(d)$ and $i(d) \vdash Fpre(d)$ then $P\text{'} = P \cup p$, but if this condition is not satisfied path p is not created.

B If $P \neq \emptyset$ then for every decomposition $d \in D$ which is a child of t we have:

$$\forall p \in P,\ p.PD\text{'} = p.PD \cup d \text{ and } p.PS\text{'} = e(d) \text{ iff } p.PS \vdash Pre(d) \text{ and } p.PS \vdash Fpre(d)$$

where e(d) is the final state of d given the initial state $i(d) = p.PS$. If for some path $p \in P$ we have $p.PS \nvdash Pre(d)$ or $p.PS \nvdash Fpre(d)$ $\forall d \in D$ which is a child of t then $P\text{'} = P \setminus p$

4.4.3 Operator Nodes

The recursive step is given by operator nodes. The way valid mission decompositions are generated when an operator node is found depends on: (i) the node decomposition (AND or OR) and (ii) the runtime annotation operator it corresponds to. Here we will exemplify with task nodes as children but the same reasoning can be applied to other operator nodes as children. One important note to give is that the following exemplifications are valid for nodes with more than two children. If the node has a single child, which is the case of the achieve goals that satisfy the *IsAchieve* condition defined in the Minimal Task Graph definition (see Equation 3.1), we have that $P'(N) = P'(C)$, where N is the operator node and C is the only child. The reason we keep these single child achieve goals is to check if achieve conditions are satisfied by decompositions in $P'(N)$, where we trim decompositions that do not satisfy it.

4.4.4 AND Decomposed Operator

For every AND decomposed operator OP we must have for every child C of OP that $P\text{'}(C) \neq \emptyset$. If this is not the case, the mission does not have any valid mission decomposition. In Figure 4.3 it is shown an example of an AND decomposed operator node that will be used as a guide throughout the formalization process. Note that here we exemplify using two children but this can be easily generalized for any number of children.

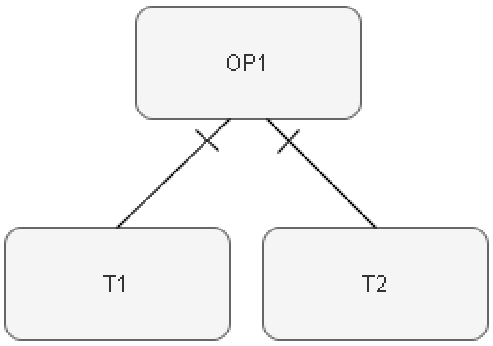


Figure 4.3: Example of an AND decomposed operator node

Sequential Operator For the paths generated by a sequential operator we have the following rules:

- If $P(OP1) = \emptyset$ then we have $P(T1) = \emptyset$ and $P(T2) = P\text{'}(T1)$ where P'(T1) are the paths generated by the decompositions of T1 given initial state I(OP1). In the end we have $P\text{'}(OP1) = P\text{'}(T2)$, where $P\text{'}(T2) = P\text{'} \circ P^{*}(T2)$.

- If $P(OP1) \neq \emptyset$ the we have $P(T1) = P(OP1)$ and $P(T2) = P\text{'}(T1)$ where $P\text{'}(T1) = P(OP1) \circ P^{*}(T1)$. In the end we have $P\text{'}(OP1) = P\text{'}(T2)$, where $P\text{'}(T2) = P\text{'}(T1) \circ P^{*}(T2)$

Paralell Operator For the paths generated by a parallel operator we have the following rules:

- If $P(OP1) = \emptyset$ then we have $P(T1) = \emptyset$ and $P(T2) = \emptyset$. In this case we have $P\text{'}(T1) = P^{*}(T1)$ and $P\text{'}(T2) = P^{*}(T2)$. In the end we have $P\text{'}(OP1) = P\text{'}(T1) \otimes P\text{'}(T2)$.
- If $P(OP1) \neq \emptyset$ then we have $P(T1) = P(OP1)$ and $P(T2) = P(OP1)$. In this case $P\text{'}(T1) = P(OP1) \circ P^{*}(T1)$ and $P\text{'}(T2) = P(OP1) \circ P^{*}(T2)$. In the end we have $P\text{'}(OP1) = P\text{'}(T1) \otimes P\text{'}(T2)$

Fallback Operator The paths generated by the fallback operator are the same as the ones for the parallel operator. This is the case because we consider that children of a fallback work in an atomic fashion, i.e., either they succeed and produce all of their effects or they fail and produce no effects. With this in mind, the initial world state for each child of a fallback operator is considered to be the same, as is the case for the parallel operator.

4.4.5 OR Decomposed Operator

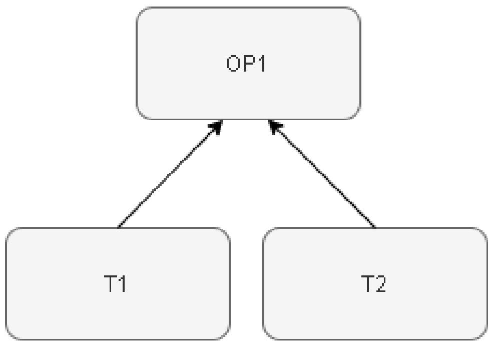


Figure 4.4: Example of an OR decomposed operator node

For every OR decomposed goal G we must have for some child C of G that $P\text{'}(C) \neq \emptyset$. If this is not the case, the mission does not have any valid mission decomposition. OR-decomposed goals can only have parallel runtime annotations, since they are in fact treated as exclusive (XOR). In Figure 4.4 it is shown an example of an OR decomposed operator node that will be used as a guide throughout the formalization process. It is important to note is that the OP1 operator must be a parallel operator (#). Finally, in the example the operator has only 2 children but it can be easily generalized, as is the case for the AND decomposed operator.

In this case, each decomposition for every task generates a new path. This means that we have $P(T1) = P(T2) = P(OP1)$ and $P'(OP1) = P'(T1) \cup P'(T2)$, where $P'(T1) = P(OP1) \circ P^*(T1)$ and $P'(T2) = P(OP1) \circ P^*(T2)$.

4.5 Constraints Generation

As is the case for the valid mission decompositions generation process, in the constraints generation process we need the Task Graph as defined in Definition 7. Using the Task Graph we generate an intermediate structure called the Constraint Tree, defined in Definition 15, from which we generate the initial mission constraints.

Definition 15 The Constraint Tree can be defined as the tuple CT = (N,E) where:

- N is the set of nodes, which can be of two types: (i) Task Nodes (TN), which are the leaf nodes of the tree, and (ii) Constraint Nodes (CN), which are the non-leaf nodes of the tree
- E is the set of edges

The constraint nodes set is represented as N.CN. A constraint node can be defined as the tuple CN = (OP,CT,CH) where:

- OP is the operator the node relates to, which can be Sequential, Parallel or Fallback
- CT is the set of constraints of the constraint node. These consist in the constraints generated by the node itself and all of the children generated constraints
- CH are the node's children. The children of a constraint node c are defined as Children(c) where:

$$Children(c) : \cup\ ch \in N \mid (c, ch) \in E$$

The task node set is represented as N.TN. A task node is defined as TN = (TID) where TID is the Task Graph ID of the task the node refers to.

Now that all of the necessary definitions were given we proceed to formalize the four main steps, which are: (i) Initial Mission Constraints Generation, (ii) Transformation of Initial Mission Constraints, (iii) Execution Constraints Generation and (iv) Trimming of Mission Constraints.

4.5.1 Initial Mission Constraints Generation

In order to generate the mission constraints we start by recursively building the Constraint Tree from the Task Graph. We transform the Task Graph in a tree by removing the Context Dependency and Execution Constraint edges. We go through this tree version of the Task Graph in a depth-first fashion and have three possibilities:

A If we find an operator we go instantiate a Constraint Node with the respective operator in the Constraint Tree. Then, we go through all of its children and instantiate the children nodes in the Constraint Tree respectively

B If we find a task we instantiate a Task Node with the respective task ID. Then, we go through all of the decomposition nodes

C If we find a decomposition node we generate nothing and simply return in order to proceed to the next node

In this way we generate the Constraint Tree, which we use to generate the mission constraints. The mission constraints are generated by traversing the Constraint Tree in a DFS fashion. In this process we have two possibilities, which are explained further.

Constraint Nodes If we find a constraint node we first generate the constraints for its children. Then we generate the constraints for that node based on their children nodes, where the way this is done is shown in further sections for each possible operator. After finishing generating the constraints for the children (CH), we add them to the set of constraints of this node (CT) for every child that is a constraint node (i.e., belongs to N.CN). Naming the constraint node as ct we can express this mathematically by:

$$\forall c \in ct.CH : ct.CT\text{'} = ct.CT \cup c.CT \text{ iff } c \in N.CN$$

Sequential Operator Constraint Node If the operator of the current constraint c is a sequential operator we generate constraints between pairs of its children from left to right as shown in Algorithm 8.

Algorithm 8 Sequential constraint node constraints generation routine

```
$j \leftarrow 1$
**for** $i = 0;\ i < c.children.size() - 1;\ i++$ **do**
    sequentialConstraintNodeConstraintGen(c.children[i],c.children[j])

    j++
**end for**
```

For the sequentialConstraintNodeConstraintGen function we have to take into account if the children are constraint or task nodes. This leads us with the following possibilities:

(i) If both nodes are tasks we simply generate a sequential constraint between them

(ii) If the first node is a constraint node and the second node is a task we name the first node as c1, the second node as t2 and the parent constraint node as c and have:

A If the first node is a sequential constraint node we have the generation of new constraints as follows:

$$\forall ct \in c1.CT, c.CT = c.CT \cup \{c'\} \text{ where } c' = CTR(ct.N2.TID, t2.TID, SEQ)$$

B If the first node is a fallback or a parallel node we have the generation of new constraints as follows:

$$\forall ct \in c1.CT,\ \ c.CT = c.CT \cup \{c', c''\} \text{ where } c' = CTR(ct.N1.TID, t2.TID, SEQ)\ and$$
$$c'' = CTR(ct.N2.TID, t2.TID, SEQ)$$

(iii) If the first node is a task and the second node is a constraint node we name the first node as t1, the second node as c2 and the parent constraint node as c and have:

A If the second node is a sequential or fallback constraint node we have the generation of new constraints as follows:

$$\forall ct \in c2.CT,\ \ c.CT = c.CT \cup \{c'\} \text{ where } c' = CTR(t1.TID, ct.N1.TID, SEQ)$$

B If the second node is a parallel constraint node we have the generation of new constraints as follows:

$$\forall ct \in c1.CT, c.CT = c.CT \cup \{c', c''\} \text{ where } c' = CTR(t1.TID, ct.N1.TID, SEQ) \text{ and}$$
$$c'' = CTR(t1.TID, ct.N2.TID, SEQ)$$

(iv) If both nodes are constraints we name the first node as c1, the second node as c2 and the parent constraint node as c and have:

A If the first node is a sequential constraint node we have:

a If the second node is a sequential or fallback node we have the generation of new constraints as follows:

$$\forall ct1 \in c1.CT,\ \forall ct2 \in c2.CT,\ c.CT = c.CT \cup \{c'\} \text{ where}$$
$$c' = CTR(ct1.N2.TID, ct2.N1.TID, SEQ)$$

b If the second node is a parallel node we have the generation of new constraints as follows:

$$\forall ct1 \in c1.CT,\ \forall ct2 \in c2.CT,\ c.CT = c.CT \cup \{c', c''\} \text{ where}$$
$$c' = CTR(ct1.N2.TID, ct2.N1.TID, SEQ) \text{ ànd}$$
$$c'' = CTR(ct1.N2.TID, ct2.N2.TID, SEQ)$$

B If the first node is a parallel or fallback node we have:

a If the second node is a sequential or fallback node we have the generation of new constraints as follows:

$$\forall ct1 \in c1.CT,\ \forall ct2 \in c2.CT,\ c.CT = c.CT \cup \{c', c''\} \text{ where}$$
$$c' = CTR(ct1.N1.TID, ct2.N1.TID, SEQ)\ and$$
$$c'' = CTR(ct1.N2.TID, ct2.N1.TID, SEQ)$$

b If the second node is a parallel node we have the generation of new constraints as follows:

$$\forall ct1 \in c1.CT,\ ct2 \in c2.CT,\ c.CT = c.CT \cup \{c', c'', c''', c''''\} \text{ where}$$
$$c' = CTR(ct1.N1.TID, ct2.N1.TID, SEQ),$$
$$c'' = CTR(ct1.N1.TID, ct2.N2.TID, SEQ),$$
$$c''' = CTR(ct1.N2.TID, ct2.N1.TID, SEQ), \text{ and}$$
$$c'''' = CTR(ct1.N2.TID, ct2.N2.TID, SEQ)$$

If the operator of the current constraint is a fallback operator we generate constraints between pairs of its children from left to right in the following fashion

Fallback Operator Constraint Node If the operator of the current constraint is a fallback operator we generate constraints between pairs of its children from left to right as shown in Algorithm 9.

Algorithm 9 Fallback constraint node constraints generation routine

```
$j \leftarrow 1$
for $i = 0;\ i < c.children.size() - 1;\ i++$ do
    fallbackConstraintNodeConstraintGen(c.children[i],c.children[j])

    j++
end for
```

For the fallbackConstraintNodeConstraintGen function we have to take into account if the children are constraint or task nodes for each pair of child nodes. This leads us with the following possibilities:

(i) If both nodes are tasks we simply generate a fallback constraint between them

(ii) If the first node is a constraint node and the second node is a task we name the first node as c1, the second node as t2 and the parent constraint node as c and have:

A If the first node is a sequential or parallel constraint node we have the generation of constraints as follows:

$$\forall ct \in c1.CT,\ \ c.CT = c.CT \cup \{c', c''\} \text{ where } c' = CTR(ct.N1.TID, t2.TID, FB) \ and$$
$$c'' = CTR(ct.N2.TID, t2.TID, FB)$$

B If the first node is a fallback constraint node we have the generation of constraints as follows:

$$\forall ct \in c1.CT, c.CT = c.CT \cup \{c'\} \text{ where } c' = CTR(ct.N2.TID, t2.TID, FB)$$

(iii) If the first node is a task and the second node is a constraint node we name the first node as t1, the second node as c2 and the parent constraint node as c and have:

A If the second node is a sequential or parallel constraint node we have the generation of constraints as follows:

$$\forall ct \in c1.CT, c.CT = c.CT \cup \{c', c''\} \text{ where } c' = CTR(t1.TID, ct.N1.TID, FB) \text{ and}$$
$$c'' = CTR(t1.TID, ct.N2.TID, FB)$$

B If the second node is a fallback constraint node we have the generation of constraints as follows:

$$\forall ct \in c2.CT,\ c.CT = c.CT \cup \{c'\} \text{ where } c' = CTR(t1.TID, ct.N1.TID, FB)$$

(iv) If both nodes are constraints we name the first node as c1, the second node as c2 and the parent constraint node as c and have:

A If the first node is a sequential or parallel constraint we have:

a If the second node is a sequential or parallel constraint we have the generation of constraints as follows:

$$\forall ct1 \in c1.CT,\ ct2 \in c2.CT,\ c.CT = c.CT \cup \{c', c'', c''', c''''\} \text{ where}$$
$$c' = CTR(ct1.N1.TID, ct2.N1.TID, FB),$$
$$c'' = CTR(ct1.N1.TID, ct2.N2.TID, FB),$$
$$c''' = CTR(ct1.N2.TID, ct2.N1.TID, FB), \text{ and}$$
$$c'''' = CTR(ct1.N2.TID, ct2.N2.TID, FB)$$

b If the second node is a fallback constraint we have the generation of constraints as follows:

$$\forall ct1 \in c1.CT,\ \forall ct2 \in c2.CT,\ c.CT = c.CT \cup \{c', c''\} \text{ where}$$
$$c' = CTR(ct1.N1.TID, ct2.N1.TID, FB)\ \textit{and}$$
$$c'' = CTR(ct1.N2.TID, ct2.N1.TID, FB)$$

B If the first node is a fallback constraint we have:

a If the second node is a sequential or parallel constraint we have the generation of constraints as follows:

$$\forall ct1 \in c1.CT,\ \forall ct2 \in c2.CT,\ c.CT = c.CT \cup \{c', c''\} \text{ where}$$
$$c' = CTR(ct1.N2.TID, ct2.N1.TID, FB) \text{ and}$$
$$c'' = CTR(ct1.N2.TID, ct2.N2.TID, FB)$$

b If the second node is a fallback constraint we have the generation of constraints as follows:

$$\forall ct1 \in c1.CT,\ \forall ct2 \in c2.CT,\ c.CT = c.CT \cup \{c'\} \text{ where}$$
$$c' = CTR(ct1.N2.TID, ct2.N1.TID, FB)$$

Parallel Operator Constraint Node If the operator of the current constraint is a sequential operator we generate constraints between all possible pairs of its children from left to right as shown in Algorithm 10.

Algorithm 10 Parallel constraint node constraints generation routine

```
for $i = 0;\ i < c.children.size() - 1;\ i++$ do
    for $j = i + 1;\ j < c.children.size();\ j++$ do
        parallelConstraintNodeConstraintGen(c.children[i],c.children[j])
    end for
end for
```

For the parallelConstraintNodeConstraintGen function we have to take into account if the children are constraint or task nodes for each pair of child nodes. This leads us with the following possibilities:

(i) If both nodes are tasks we simply generate a parallel constraint between them

(ii) If the first node is a constraint node and the second node is a task we name the first node as c1, the second node as t2 and the parent constraint node as c and have:

A For all types of constraints we have the generation of constraints as follows:

$$\forall ct \in c1.CT,\ c.CT = c.CT \cup \{c', c''\} \text{ where } c' = CTR(ct.N1.TID, t2.TID, PAR)\ and$$
$$c'' = CTR(ct.N2.TID, t2.TID, PAR)$$

(iii) If the first node is a task and the second node is a constraint node we name the first node as t1, the second node as c2 and the parent constraint node as c and have:

A For all types of constraints we have the generation of constraints as follows:

$$\forall ct \in c1.CT, c.CT = c.CT \cup \{c', c''\} \text{ where } c' = CTR(t1.TID, ct.N1.TID, PAR) \text{ and}$$
$$c'' = CTR(t1.TID, ct.N2.TID, PAR)$$

(iv) If both nodes are constraints we name the first node as c1, the second node as c2 and the parent constraint node as c and have:

A For all types of constraints we have the generation of constraints as follows:

$$\forall ct1 \in c1.CT,\ ct2 \in c2.CT,\ c.CT = c.CT \cup \{c', c'', c''', c''''\} \text{ where}$$
$$c' = CTR(ct1.N1.TID, ct2.N1.TID, PAR),$$
$$c'' = CTR(ct1.N1.TID, ct2.N2.TID, PAR),$$
$$c''' = CTR(ct1.N2.TID, ct2.N1.TID, PAR), \text{ and}$$
$$c'''' = CTR(ct1.N2.TID, ct2.N2.TID, PAR)$$

Task Nodes If we find a Task Node we simply return its related task ID to be used by its parent node in its constraint generation process.

4.5.2 Transformation of Initial Mission Constraints

The transformation of the initial mission constraints is the process where we transform task constraints into decomposition constraints. Let's call Decompositions(t) the set of Task Graph ID's for the decompositions of a given task, given its own ID in the Task Graph, and C the set of constraints. Let's also define as c.FB the set of fallback constraints, c.SEQ the set of sequential constraints and c.PAR the set of parallel constraints where $c.FB$, $c.SEQ$ and $c.PAR \subseteq C$. There are two possible behaviors, which will be further explained.

Sequential and Fallback Constraints For sequential and fallback constraints we have:

$$\forall c \in C.FB \cup C.SEQ,\ C' = (C \setminus c) \cup D \mid \forall d1 \in Decompositions(c.N1.TID),$$
$$d2 \in Decompositions(c.N2.TID)\ :\ D' = D \cup CTR(d1, d2, c.T)$$

The pseudocode representation for this process is given in Algorithm 11.

Parallel Constraints For parallel constraints we check for each decomposition of the first task if it is involved in a context dependency with the second task. If they in fact belong to a context dependency we create a sequential constraint between this decomposition and all of the decompositions of the second task. Let's define CD(d,t) as a boolean function that returns true if the decomposition d is involved in a context dependency with

Algorithm 11 Transformation of initial sequential/fallback mission constraints routine

```
$transformedConstraints \leftarrow []$

for $c$ in $c.FB \cup c.SEQ$ do
    D1 = getDecompositions(c.N1.TID)
    D2 = getDecompositions(c.N2.TID)

    for d1 in D1 do
        for d2 in D2 do
            tCtr = generateConstraint(d1,d2,c.T)

            transformedConstraints = transformedConstraints.pushBack(tCtr)
        end for
    end for
end for
```

the task t based on their Task Graph IDs.

$$\forall c \in C.PAR,\ C' = (C \setminus c) \cup D \mid \forall d1 \in \mathit{Decompositions}(c.N1.TID),$$
$$d2 \in \mathit{Decompositions}(c.N2.TID)\ :\ D\text{'} = D \cup CTR(d1, d2, SEQ)$$
$$\text{iff } CD(d1, c.N2.TID)$$

The pseudocode representation for this process is given in Algorithm 12.

Algorithm 12 Transformation of initial parallel mission constraints routine

```
$transformedConstraints \leftarrow []$

for c in C.PAR do
    D1 = getDecompositions(c.N1.TID)
    for d1 in D1 do
        if CD(d1,c.N2.TID) then
            D2 = getDecompositions(c.N2.TID)

            for d2 in D2 do
                tCtr = generateConstraint(d1,d2,SEQ)

                transformedConstraints = transformedConstraints.pushBack(tCtr)
            end for
        end if
    end for
end for
```

4.5.3 Execution Constraints Generation

In this step we get the set of constraints C after they are transformed and append execution constraints to it. Let's call the set of execution constraints as C.EC where $C.EC \subseteq C$. Also, let's call the set of task nodes in the Task Graph as AT. In order to generate execution constraints we check for every task $t \in AT$ if there is any execution constraint edge in its set of outer edges, represented by OuterEdges(t). If there is any execution constraint edge we check if we already created constraints between task t decompositions and the decompositions of the target task t' of the edge since execution constraint edges are bidirectional. If constraints were not created between t and t' we create execution constraints between every decomposition of t and every decomposition of t', where the group and divisible attributes are set according to what is defined by the execution constraint edge in the Task Graph. Denoting as Created(t,t') a boolean function that defines if execution constraints were created between t and t', we have the routine to create execution constraints shown in the pseudocode in Algorithm 13.

Algorithm 13 Routine for creation of execution constraints

```
$C.EC \leftarrow []$

for t in AT do
    for edge in OuterEdges(t) do
        if edge.type == EC then
            t' = edge.target
            if !Created(t,t') then
                for d1 in Decompositions(t) do
                    for d2 in Decompositions(t') do
                        execConstraint = generateConstraint(d1,d2,EC)

                        C.EC.pushBack(execConstraint)
                    end for
                end for
            end if
        end if
    end for
end for
```

4.5.4 Trimming of Mission Constraints

In this step we generate the minimal set of constraints, as explained in Section 3.4.4. The formalization for the case where a sequential constraint is trimmed because of a fallback constraint is shown below:

$$C' : \forall c \in C.SEQ, C' = C \setminus \{c\} \text{ iff } \exists\ c\text{'} \in c.SEQ,\ c\text{''} \in c.FB \mid c\text{'}.N1 = c.N1,$$
$$c\text{'}.N2 = c\text{''}.N1 \text{ and } c\text{''}.N2 = c.N2$$

For the case where a sequential constraint is trimmed because of a sequential constraint we have the formalization as follows:

$$C' : \forall c \in C.SEQ, C' = C \setminus \{c\} \text{ iff } \exists\ c\text{'}, c\text{''} \in c.SEQ \mid c\text{'}.N1 = c.N1,$$
$$c\text{'}.N2 = c\text{''}.N1 \text{ and } c\text{''}.N2 = c.N2$$

Finally, the formalization for the case where a fallback constraint is trimmed because of a fallback constraint is:

$$C' : \forall c \in C.FB, C' = C \setminus \{c\} \text{ iff } \exists\ c\text{'}, c\text{''} \in c.FB \mid c\text{'}.N1 = c.N1,$$
$$c\text{'}.N2 = c\text{''}.N1 \text{ and } c\text{''}.N2 = c.N2$$

Chapter 5

Evaluation

In this chapter we will present the evaluations perform on the work in order to provide evidence on its value and correctness. Three evaluation process were performed: (i) modeling of missions present in the community (and thus publicly available), where models for missions of the RoboMAX repository [6] were created and verified with their respective authors, and (ii) verification of the efficiency of the decomposition, in which some missions were tested for different world knowledges, where each one required different sizes of model expansions, and the time to perform the decomposition was measured.

5.1 Modeling of Missions From the Community

In this process we modelled five missions from the RoboMAX repository: (i) Vital Signs Monitoring, (ii) Seizure Recognition, (iii) Lab Samples Logistics, (iv) Deliver Goods/Equipment and (v) Food Logistics. The models, files and the results obtained from the decomposition for each one of these examples are shown in further sections. It is important to note that every mission description contains execution-related actions or time-related aspects that are not captured by the models in the framework and are assumed to be dealt with by execution engines or execution models. Also, the models for each mission were validated by the respective authors in order to avoid any misinterpretations and reduce the uncertainty in the generated outputs. All of the models for each mission can be found in a Github repository[1] and the models that are ommited in this section are shown in Appendix A.

[1]`https://github.com/ericbg27/RoboMax-Examples-Modelling`

5.1.1 Vital Signs Monitoring

The Vital Signs Monitoring (VSM) mission consists in a hospital domain where robots need to check patient's vital signs, if patients are available, where for each clinical condition different vital signs should be checked. The description of this mission is shown below, where red colored sentences are the ones that are not captured by the models.

Every two hours, all patient's vital signs should be checked. Therefore, the assigned robot should visit every occupied room. Before entering the room, the robot must check if the patient is available for vital signs checking, and if there is enough battery to perform all the tasks inside the room. Hospital rooms may have one or more beds. So the robot has to identify the correct patient either by asking or by scanning the patient's barcode. If the patient is not available, the robot should come back in 5 minutes. If there is not enough battery, the robot should recharge and the tasks need to be reassigned. Once inside the room, the robot should approach the patient, announce the procedure to be performed, and provide instructions to the patient. The failure of collecting any necessary vital signs should trigger a set of questions to assess the patient status for those signs that were failed to be collected. In case of non-response, an alert should be sent to the responsible nurse/doctor. Once the patient is checked, the robot should mark it as done and proceed to the next room. Patients subject to infectious diseases must be checked for heart rate, blood pressure, body temperature, and blood oxygenation. Patients subject to post-surgical conditions must be checked for heart rate and blood oxygenation. Patients with a history of diabetes must be checked for glucose levels in the blood. When a robot comes out of a room where a patient subject to infectious disease is being treated, it must be sterilized. All the patients in high risk must be checked in the first 15 minutes of the mission. If the robot is performing any duties with patient contact (touching the patient) it should disinfect its actuators afterwards. Additional robots can be assigned to help in case of impossibility of performing the mission in the required time. The responsible manager for the site must be alerted if no combination of robots can complete the mission.

The goal model that represents this mission is shown in Figure 5.1. The world knowledge used to test the decomposition is shown in Figure 5.2 alongside the generated task instances in Figure 5.3. One can verify that task instances were generated correctly since we have tasks related to the to patients in RoomA, given that this is the only room we have and both patients need to have their vital signs checked, and because we have:

- One instance of the EnterRoom, RobotSanitization and RechargeBattery tasks, since we only have a single room and a single way of decomposing these tasks

- Two instances of the Approach Patient, ProvideInstructions, AssessPatientStatus and SendAlert tasks, since we have two patients and only one way of decomposing both of these tasks
- Six instances of the CollectVitalSigns task, since we have two patients and three ways of decomposing this task

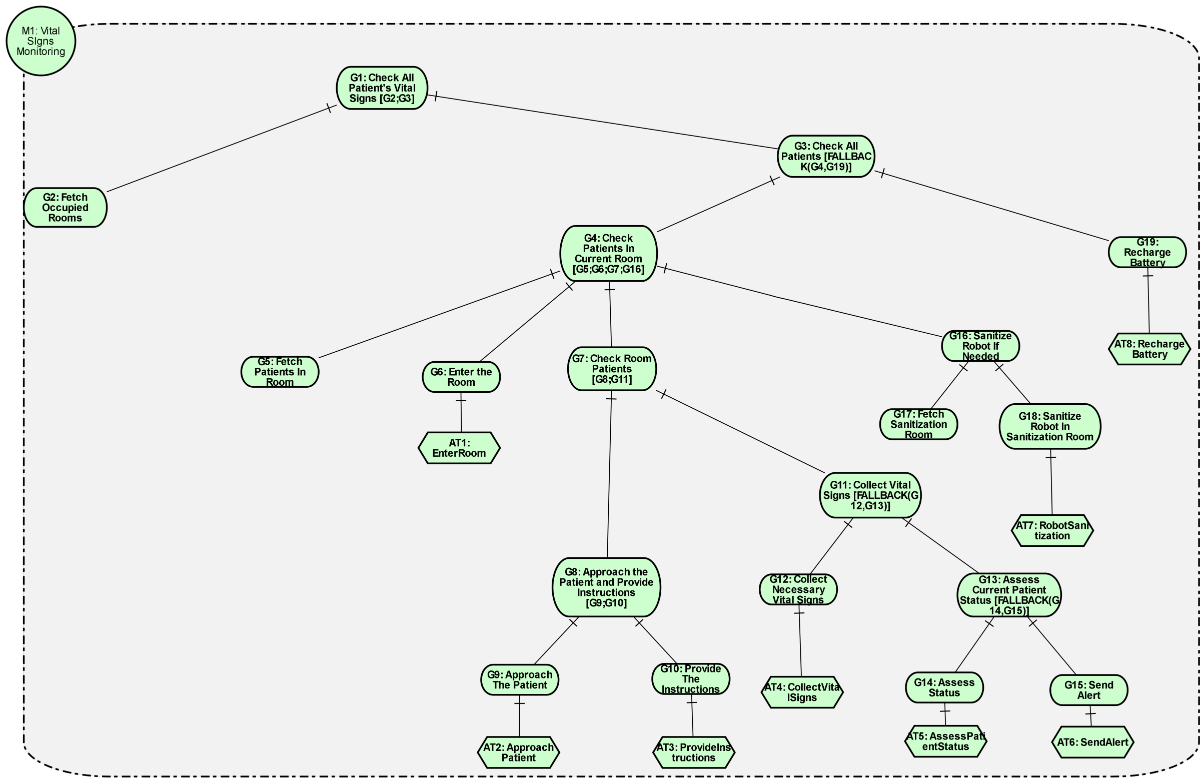
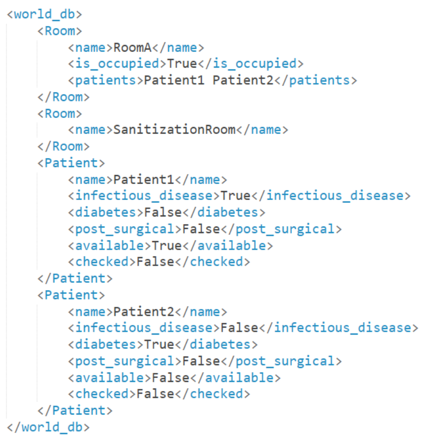

M1: Vital Signs Monitoring
G1: Check All Patient's Vital Signs [G2;G3]
G2: Fetch Occupied Rooms
G3: Check All Patients [FALLBACK(G4,G19)]
G4: Check Patients In Current Room [G5;G6;G7;G16]
G19: Recharge Battery
AT8: Recharge Battery
G5: Fetch Patients In Room
G6: Enter the Room
AT1: EnterRoom
G7: Check Room Patients [G8;G11]
G16: Sanitize Robot If Needed
G17: Fetch Sanitization Room
G18: Sanitize Robot In Sanitization Room
AT7: RobotSanitization
G11: Collect Vital Signs [FALLBACK(G12,G13)]
G8: Approach the Patient and Provide Instructions [G9;G10]
G12: Collect Necessary Vital Signs
G13: Assess Current Patient Status [FALLBACK(G14,G15)]
G9: Approach The Patient
G10: Provide The Instructions
AT4: CollectVitalSigns
G14: Assess Status
G15: Send Alert
AT2: Approach Patient
AT3: ProvideInstructions
AT5: AssessPatientStatus
AT6: SendAlert

Figure 5.1: Goal model for the Vital Signs Monitoring mission



```
<world_db>
    <Room>
        <name>RoomA</name>
        <is_occupied>True</is_occupied>
        <patients>Patient1 Patient2</patients>
    </Room>
    <Room>
        <name>SanitizationRoom</name>
    </Room>
    <Patient>
        <name>Patient1</name>
        <infectious_disease>True</infectious_disease>
        <diabetes>False</diabetes>
        <post_surgical>False</post_surgical>
        <available>True</available>
        <checked>False</checked>
    </Patient>
    <Patient>
        <name>Patient2</name>
        <infectious_disease>False</infectious_disease>
        <diabetes>True</diabetes>
        <post_surgical>False</post_surgical>
        <available>False</available>
        <checked>False</checked>
    </Patient>
</world_db>
```

Figure 5.2: Knowledge for the Vital Signs Monitoring mission

The valid mission decompositions are shown in Figure 5.4. We omit the generated constraints illustration given the amount of constraints in the mission, where one can verify by the output generated by the decomposition process that expected constraints were correctly generated. The terminal output of constraints obtained by running the decomposition binary is shown in Figure 5.5. It is important to note that all of the execution constraints are non-group constraints since goal G4 is a non-group goal. This leads us to have all of the possible combinations between task decompositions that are children of G4 taken two by two, except for decompositions of the same task instance.

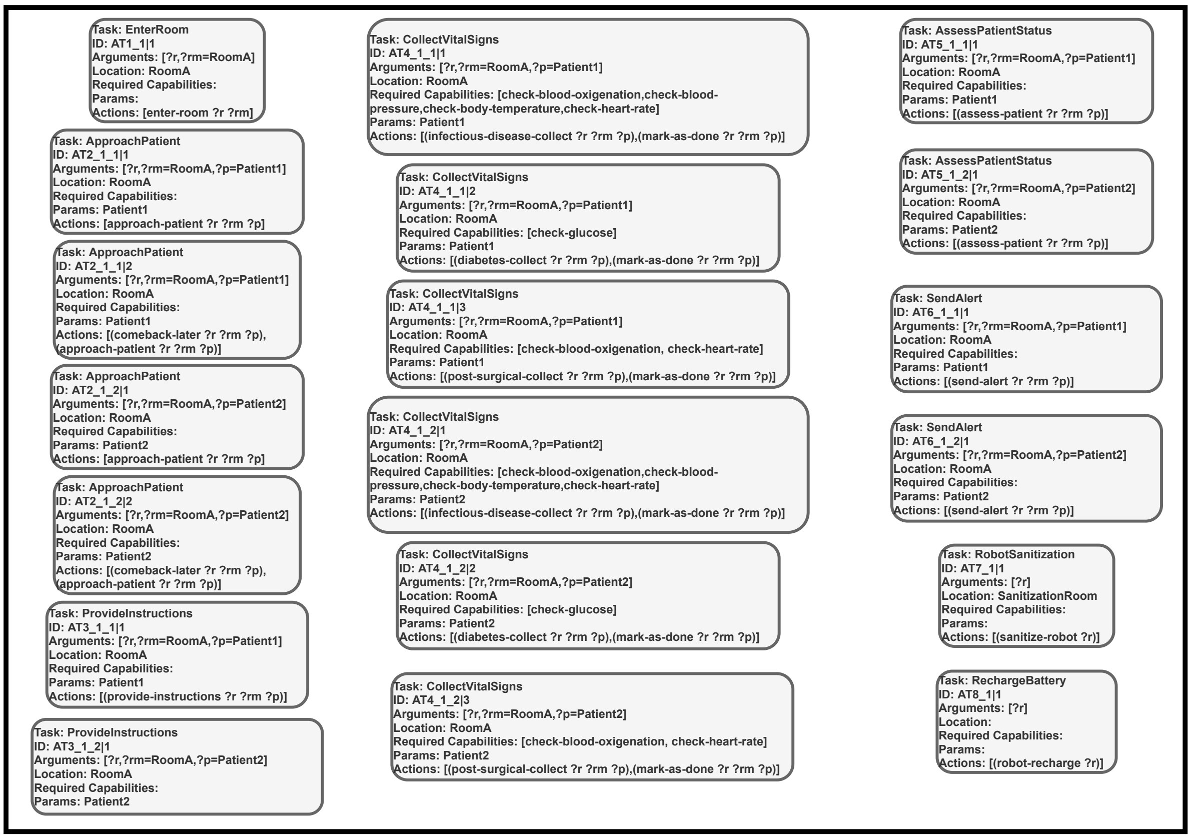

Task: EnterRoom
ID: AT1_1|1
Arguments: [?r,?rm=RoomA]
Location: RoomA
Required Capabilities:
Params:
Actions: [enter-room ?r ?rm]
Task: ApproachPatient
ID: AT2_1_1|1
Arguments: [?r,?rm=RoomA,?p=Patient1]
Location: RoomA
Required Capabilities:
Params: Patient1
Actions: [approach-patient ?r ?rm ?p]
Task: ApproachPatient
ID: AT2_1_1|2
Arguments: [?r,?rm=RoomA,?p=Patient1]
Location: RoomA
Required Capabilities:
Params: Patient1
Actions: [(comeback-later ?r ?rm ?p), (approach-patient ?r ?rm ?p)]
Task: ApproachPatient
ID: AT2_1_2|1
Arguments: [?r,?rm=RoomA,?p=Patient2]
Location: RoomA
Required Capabilities:
Params: Patient2
Actions: [approach-patient ?r ?rm ?p]
Task: ApproachPatient
ID: AT2_1_2|2
Arguments: [?r,?rm=RoomA,?p=Patient2]
Location: RoomA
Required Capabilities:
Params: Patient2
Actions: [(comeback-later ?r ?rm ?p), (approach-patient ?r ?rm ?p)]
Task: ProvideInstructions
ID: AT3_1_1|1
Arguments: [?r,?rm=RoomA,?p=Patient1]
Location: RoomA
Required Capabilities:
Params: Patient1
Actions: [(provide-instructions ?r ?rm ?p)]
Task: ProvideInstructions
ID: AT3_1_2|1
Arguments: [?r,?rm=RoomA,?p=Patient2]
Location: RoomA
Required Capabilities:
Params: Patient2
Task: CollectVitalSigns
ID: AT4_1_1|1
Arguments: [?r,?rm=RoomA,?p=Patient1]
Location: RoomA
Required Capabilities: [check-blood-oxigenation,check-blood-pressure,check-body-temperature,check-heart-rate]
Params: Patient1
Actions: [(infectious-disease-collect ?r ?rm ?p),(mark-as-done ?r ?rm ?p)]
Task: CollectVitalSigns
ID: AT4_1_1|2
Arguments: [?r,?rm=RoomA,?p=Patient1]
Location: RoomA
Required Capabilities: [check-glucose]
Params: Patient1
Actions: [(diabetes-collect ?r ?rm ?p),(mark-as-done ?r ?rm ?p)]
Task: CollectVitalSigns
ID: AT4_1_1|3
Arguments: [?r,?rm=RoomA,?p=Patient1]
Location: RoomA
Required Capabilities: [check-blood-oxigenation, check-heart-rate]
Params: Patient1
Actions: [(post-surgical-collect ?r ?rm ?p),(mark-as-done ?r ?rm ?p)]
Task: CollectVitalSigns
ID: AT4_1_2|1
Arguments: [?r,?rm=RoomA,?p=Patient2]
Location: RoomA
Required Capabilities: [check-blood-oxigenation,check-blood-pressure,check-body-temperature,check-heart-rate]
Params: Patient2
Actions: [(infectious-disease-collect ?r ?rm ?p),(mark-as-done ?r ?rm ?p)]
Task: CollectVitalSigns
ID: AT4_1_2|2
Arguments: [?r,?rm=RoomA,?p=Patient2]
Location: RoomA
Required Capabilities: [check-glucose]
Params: Patient2
Actions: [(diabetes-collect ?r ?rm ?p),(mark-as-done ?r ?rm ?p)]
Task: CollectVitalSigns
ID: AT4_1_2|3
Arguments: [?r,?rm=RoomA,?p=Patient2]
Location: RoomA
Required Capabilities: [check-blood-oxigenation, check-heart-rate]
Params: Patient2
Actions: [(post-surgical-collect ?r ?rm ?p),(mark-as-done ?r ?rm ?p)]
Task: AssessPatientStatus
ID: AT5_1_1|1
Arguments: [?r,?rm=RoomA,?p=Patient1]
Location: RoomA
Required Capabilities:
Params: Patient1
Actions: [(assess-patient ?r ?rm ?p)]
Task: AssessPatientStatus
ID: AT5_1_2|1
Arguments: [?r,?rm=RoomA,?p=Patient2]
Location: RoomA
Required Capabilities:
Params: Patient2
Actions: [(assess-patient ?r ?rm ?p)]
Task: SendAlert
ID: AT6_1_1|1
Arguments: [?r,?rm=RoomA,?p=Patient1]
Location: RoomA
Required Capabilities:
Params: Patient1
Actions: [(send-alert ?r ?rm ?p)]
Task: SendAlert
ID: AT6_1_2|1
Arguments: [?r,?rm=RoomA,?p=Patient2]
Location: RoomA
Required Capabilities:
Params: Patient2
Actions: [(send-alert ?r ?rm ?p)]
Task: RobotSanitization
ID: AT7_1|1
Arguments: [?r]
Location: SanitizationRoom
Required Capabilities:
Params:
Actions: [(sanitize-robot ?r)]
Task: RechargeBattery
ID: AT8_1|1
Arguments: [?r]
Location:
Required Capabilities:
Params:
Actions: [(robot-recharge ?r)]

Figure 5.3: Generated task instances for the Vital Signs Monitoring mission

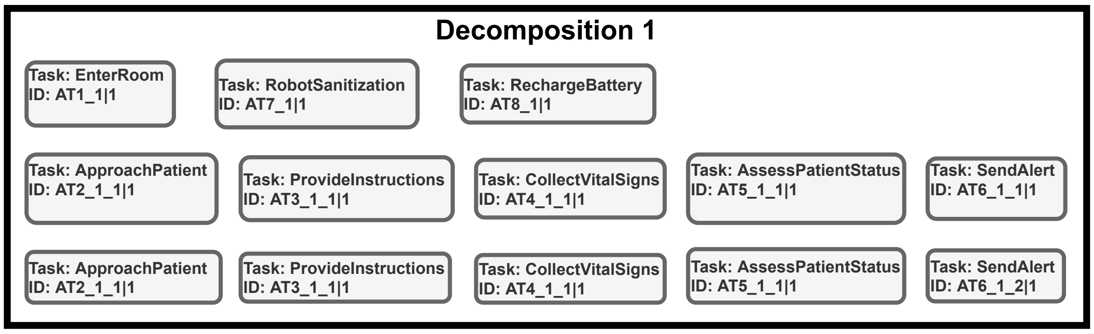
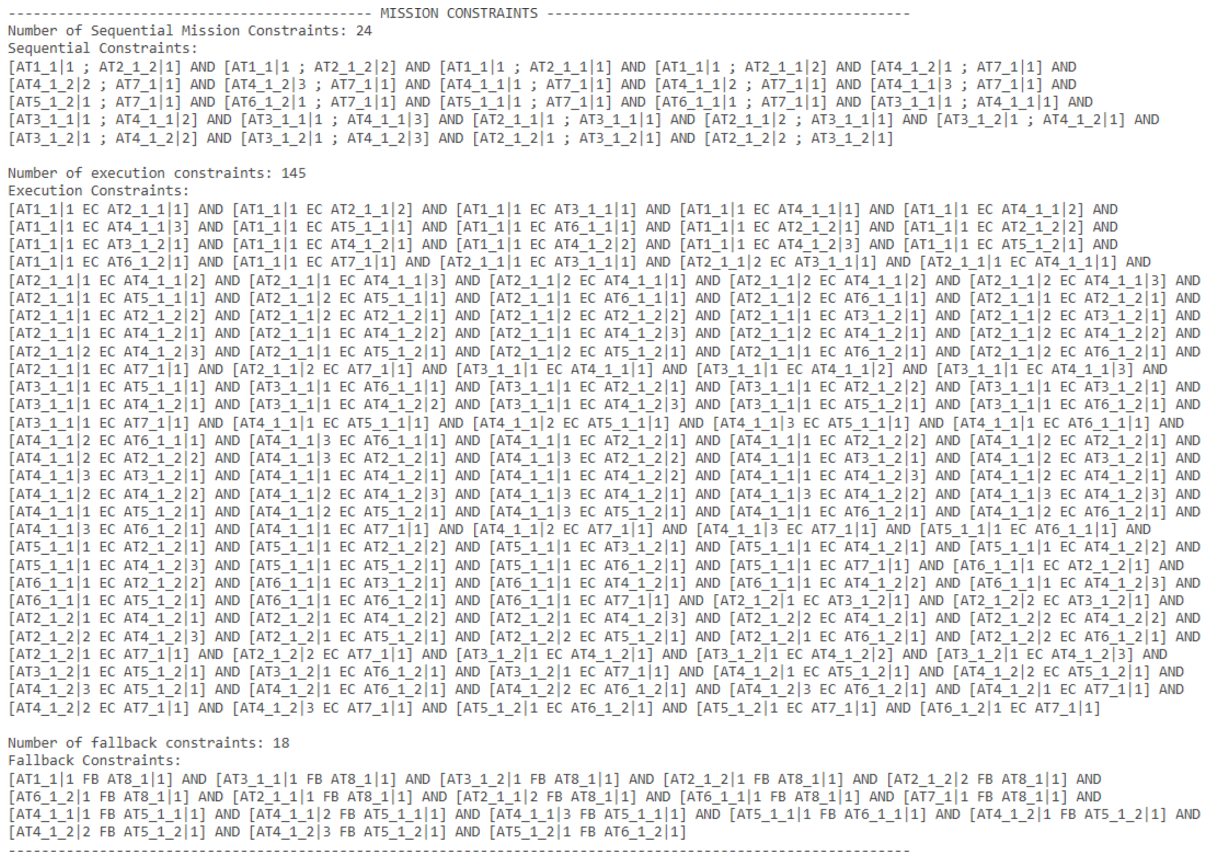


Figure 5.4: Mission decompositions for the Vital Signs Monitoring mission



```
--------------------------------------------- MISSION CONSTRAINTS --------------------------------------------
Number of Sequential Mission Constraints: 24
Sequential Constraints:
[AT1_1|1 ; AT2_1_2|1] AND [AT1_1|1 ; AT2_1_2|2] AND [AT1_1|1 ; AT2_1_1|1] AND [AT1_1|1 ; AT2_1_1|2] AND [AT4_1_2|1 ; AT7_1|1] AND
[AT4_1_2|2 ; AT7_1|1] AND [AT4_1_2|3 ; AT7_1|1] AND [AT4_1_1|1 ; AT7_1|1] AND [AT4_1_1|2 ; AT7_1|1] AND [AT4_1_1|3 ; AT7_1|1] AND
[AT5_1_2|1 ; AT7_1|1] AND [AT6_1_2|1 ; AT7_1|1] AND [AT5_1_1|1 ; AT7_1|1] AND [AT6_1_1|1 ; AT7_1|1] AND [AT3_1_1|1 ; AT4_1_1|1] AND
[AT3_1_1|1 ; AT4_1_1|2] AND [AT3_1_1|1 ; AT4_1_1|3] AND [AT2_1_1|1 ; AT3_1_1|1] AND [AT2_1_1|2 ; AT3_1_1|1] AND [AT3_1_2|1 ; AT4_1_2|1] AND
[AT3_1_2|1 ; AT4_1_2|2] AND [AT3_1_2|1 ; AT4_1_2|3] AND [AT2_1_2|1 ; AT3_1_2|1] AND [AT2_1_2|2 ; AT3_1_2|1]

Number of execution constraints: 145
Execution Constraints:
[AT1_1|1 EC AT2_1_1|1] AND [AT1_1|1 EC AT2_1_1|2] AND [AT1_1|1 EC AT3_1_1|1] AND [AT1_1|1 EC AT4_1_1|1] AND [AT1_1|1 EC AT4_1_1|2] AND
[AT1_1|1 EC AT4_1_1|3] AND [AT1_1|1 EC AT5_1_1|1] AND [AT1_1|1 EC AT6_1_1|1] AND [AT1_1|1 EC AT2_1_2|1] AND [AT1_1|1 EC AT2_1_2|2] AND
[AT1_1|1 EC AT3_1_2|1] AND [AT1_1|1 EC AT4_1_2|1] AND [AT1_1|1 EC AT4_1_2|2] AND [AT1_1|1 EC AT4_1_2|3] AND [AT1_1|1 EC AT5_1_2|1] AND
[AT1_1|1 EC AT6_1_2|1] AND [AT1_1|1 EC AT7_1|1] AND [AT2_1_1|1 EC AT3_1_1|1] AND [AT2_1_1|2 EC AT3_1_1|1] AND [AT2_1_1|1 EC AT4_1_1|1] AND
[AT2_1_1|1 EC AT4_1_1|2] AND [AT2_1_1|1 EC AT4_1_1|3] AND [AT2_1_1|2 EC AT4_1_1|1] AND [AT2_1_1|2 EC AT4_1_1|2] AND [AT2_1_1|2 EC AT4_1_1|3] AND
[AT2_1_1|1 EC AT5_1_1|1] AND [AT2_1_1|2 EC AT5_1_1|1] AND [AT2_1_1|1 EC AT6_1_1|1] AND [AT2_1_1|2 EC AT6_1_1|1] AND [AT2_1_1|1 EC AT2_1_2|1] AND
[AT2_1_1|1 EC AT2_1_2|2] AND [AT2_1_1|2 EC AT2_1_2|1] AND [AT2_1_1|2 EC AT2_1_2|2] AND [AT2_1_1|1 EC AT3_1_2|1] AND [AT2_1_1|2 EC AT3_1_2|1] AND
[AT2_1_1|1 EC AT4_1_2|1] AND [AT2_1_1|1 EC AT4_1_2|2] AND [AT2_1_1|1 EC AT4_1_2|3] AND [AT2_1_1|2 EC AT4_1_2|1] AND [AT2_1_1|2 EC AT4_1_2|2] AND
[AT2_1_1|2 EC AT4_1_2|3] AND [AT2_1_1|1 EC AT5_1_2|1] AND [AT2_1_1|2 EC AT5_1_2|1] AND [AT2_1_1|1 EC AT6_1_2|1] AND [AT2_1_1|2 EC AT6_1_2|1] AND
[AT2_1_1|1 EC AT7_1|1] AND [AT2_1_1|2 EC AT7_1|1] AND [AT3_1_1|1 EC AT4_1_1|1] AND [AT3_1_1|1 EC AT4_1_1|2] AND [AT3_1_1|1 EC AT4_1_1|3] AND
[AT3_1_1|1 EC AT5_1_1|1] AND [AT3_1_1|1 EC AT6_1_1|1] AND [AT3_1_1|1 EC AT2_1_2|1] AND [AT3_1_1|1 EC AT2_1_2|2] AND [AT3_1_1|1 EC AT3_1_2|1] AND
[AT3_1_1|1 EC AT4_1_2|1] AND [AT3_1_1|1 EC AT4_1_2|2] AND [AT3_1_1|1 EC AT4_1_2|3] AND [AT3_1_1|1 EC AT5_1_2|1] AND [AT3_1_1|1 EC AT6_1_2|1] AND
[AT3_1_1|1 EC AT7_1|1] AND [AT4_1_1|1 EC AT5_1_1|1] AND [AT4_1_1|2 EC AT5_1_1|1] AND [AT4_1_1|3 EC AT5_1_1|1] AND [AT4_1_1|1 EC AT6_1_1|1] AND
[AT4_1_1|2 EC AT6_1_1|1] AND [AT4_1_1|3 EC AT6_1_1|1] AND [AT4_1_1|1 EC AT2_1_2|1] AND [AT4_1_1|1 EC AT2_1_2|2] AND [AT4_1_1|2 EC AT2_1_2|1] AND
[AT4_1_1|2 EC AT2_1_2|2] AND [AT4_1_1|3 EC AT2_1_2|1] AND [AT4_1_1|3 EC AT2_1_2|2] AND [AT4_1_1|1 EC AT3_1_2|1] AND [AT4_1_1|2 EC AT3_1_2|1] AND
[AT4_1_1|3 EC AT3_1_2|1] AND [AT4_1_1|1 EC AT4_1_2|1] AND [AT4_1_1|1 EC AT4_1_2|2] AND [AT4_1_1|1 EC AT4_1_2|3] AND [AT4_1_1|2 EC AT4_1_2|1] AND
[AT4_1_1|2 EC AT4_1_2|2] AND [AT4_1_1|2 EC AT4_1_2|3] AND [AT4_1_1|3 EC AT4_1_2|1] AND [AT4_1_1|3 EC AT4_1_2|2] AND [AT4_1_1|3 EC AT4_1_2|3] AND
[AT4_1_1|1 EC AT5_1_2|1] AND [AT4_1_1|2 EC AT5_1_2|1] AND [AT4_1_1|3 EC AT5_1_2|1] AND [AT4_1_1|1 EC AT6_1_2|1] AND [AT4_1_1|2 EC AT6_1_2|1] AND
[AT4_1_1|3 EC AT6_1_2|1] AND [AT4_1_1|1 EC AT7_1|1] AND [AT4_1_1|2 EC AT7_1|1] AND [AT4_1_1|3 EC AT7_1|1] AND [AT5_1_1|1 EC AT6_1_1|1] AND
[AT5_1_1|1 EC AT2_1_2|1] AND [AT5_1_1|1 EC AT2_1_2|2] AND [AT5_1_1|1 EC AT3_1_2|1] AND [AT5_1_1|1 EC AT4_1_2|1] AND [AT5_1_1|1 EC AT4_1_2|2] AND
[AT5_1_1|1 EC AT4_1_2|3] AND [AT5_1_1|1 EC AT5_1_2|1] AND [AT5_1_1|1 EC AT6_1_2|1] AND [AT5_1_1|1 EC AT7_1|1] AND [AT6_1_1|1 EC AT2_1_2|1] AND
[AT6_1_1|1 EC AT2_1_2|2] AND [AT6_1_1|1 EC AT3_1_2|1] AND [AT6_1_1|1 EC AT4_1_2|1] AND [AT6_1_1|1 EC AT4_1_2|2] AND [AT6_1_1|1 EC AT4_1_2|3] AND
[AT6_1_1|1 EC AT5_1_2|1] AND [AT6_1_1|1 EC AT6_1_2|1] AND [AT6_1_1|1 EC AT7_1|1] AND [AT2_1_2|1 EC AT3_1_2|1] AND [AT2_1_2|2 EC AT3_1_2|1] AND
[AT2_1_2|1 EC AT4_1_2|1] AND [AT2_1_2|1 EC AT4_1_2|2] AND [AT2_1_2|1 EC AT4_1_2|3] AND [AT2_1_2|2 EC AT4_1_2|1] AND [AT2_1_2|2 EC AT4_1_2|2] AND
[AT2_1_2|2 EC AT4_1_2|3] AND [AT2_1_2|1 EC AT5_1_2|1] AND [AT2_1_2|2 EC AT5_1_2|1] AND [AT2_1_2|1 EC AT6_1_2|1] AND [AT2_1_2|2 EC AT6_1_2|1] AND
[AT2_1_2|1 EC AT7_1|1] AND [AT2_1_2|2 EC AT7_1|1] AND [AT3_1_2|1 EC AT4_1_2|1] AND [AT3_1_2|1 EC AT4_1_2|2] AND [AT3_1_2|1 EC AT4_1_2|3] AND
[AT3_1_2|1 EC AT5_1_2|1] AND [AT3_1_2|1 EC AT6_1_2|1] AND [AT3_1_2|1 EC AT7_1|1] AND [AT4_1_2|1 EC AT5_1_2|1] AND [AT4_1_2|2 EC AT5_1_2|1] AND
[AT4_1_2|3 EC AT5_1_2|1] AND [AT4_1_2|1 EC AT6_1_2|1] AND [AT4_1_2|2 EC AT6_1_2|1] AND [AT4_1_2|3 EC AT6_1_2|1] AND [AT4_1_2|1 EC AT7_1|1] AND
[AT4_1_2|2 EC AT7_1|1] AND [AT4_1_2|3 EC AT7_1|1] AND [AT5_1_2|1 EC AT6_1_2|1] AND [AT5_1_2|1 EC AT7_1|1] AND [AT6_1_2|1 EC AT7_1|1]

Number of fallback constraints: 18
Fallback Constraints:
[AT1_1|1 FB AT8_1|1] AND [AT3_1_1|1 FB AT8_1|1] AND [AT3_1_2|1 FB AT8_1|1] AND [AT2_1_2|1 FB AT8_1|1] AND [AT2_1_2|2 FB AT8_1|1] AND
[AT6_1_2|1 FB AT8_1|1] AND [AT2_1_1|1 FB AT8_1|1] AND [AT2_1_1|2 FB AT8_1|1] AND [AT6_1_1|1 FB AT8_1|1] AND [AT7_1|1 FB AT8_1|1] AND
[AT4_1_1|1 FB AT5_1_1|1] AND [AT4_1_1|2 FB AT5_1_1|1] AND [AT4_1_1|3 FB AT5_1_1|1] AND [AT5_1_1|1 FB AT6_1_1|1] AND [AT4_1_2|1 FB AT5_1_2|1] AND
[AT4_1_2|2 FB AT5_1_2|1] AND [AT4_1_2|3 FB AT5_1_2|1] AND [AT5_1_2|1 FB AT6_1_2|1]
--------------------------------------------------------------------------------------------------------------
```

Figure 5.5: Constraints terminal output for the Vital Signs Monitoring mission

5.1.2 Lab Samples Logistics

The Lab Samples Logistics (LSL) mission consists in a hospital domain where robots need to get samples collected by nurses and take them to a Pharmacy where they will be stored by a robotic arm. The description of the mission is shown below, where red colored sentences are the ones that are not captured by the models.

> Lab samples should be transported from patient floors to the laboratory. A nurse responsible for collecting a sample can request a delivery to the laboratory. Each sample before collection is identified with a barcode. The nurse inserts data about the sample in a track system. Samples are transported in secure locked drawers. In the laboratory, the sample can be picked by a robotic arm or laboratory personnel. Only authorized personnel or specific robots in specific locations should be able to open the locked drawers. The robotic arm picks up samples, scans the barcode in each sample, sorts, and loads the samples into the entry-module of the analysis machines. Information about where and when a given sample was picked should be logged securely. The time between sample collection and laboratory delivery should be tracked to be sure that it goes according to laboratory protocols.

The goal model that represents this mission is shown in Figure 5.6. The task instances generated from goal model alongside the HDDL Domain Definition and the world knowledge, depicted in Figure 5.7, are shown in Figure 5.8. One can verify that task instances were generated correctly since we have tasks only for the only delivery that we have, requested by Nurse1 at location Room3, and because we have one instance for each task since each task has only one way of being decomposed.

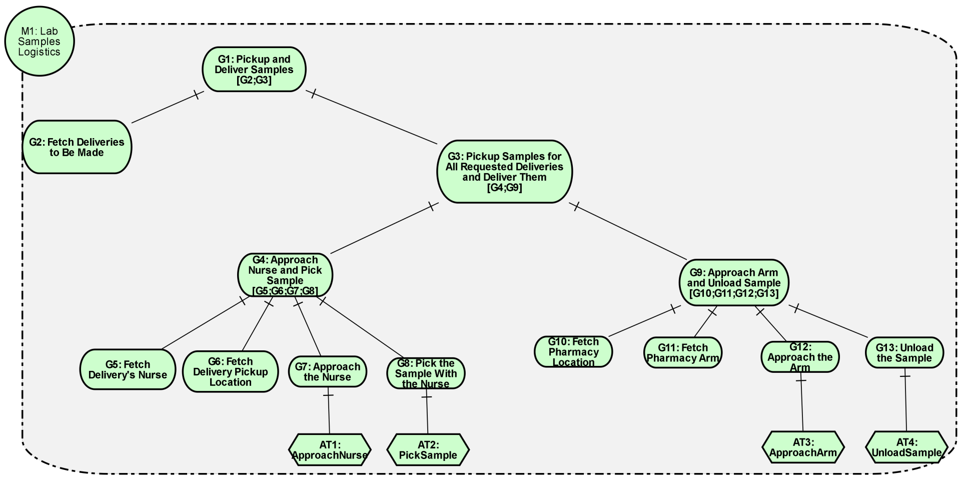

M1: Lab Samples Logistics
G1: Pickup and Deliver Samples [G2;G3]
G2: Fetch Deliveries to Be Made
G3: Pickup Samples for All Requested Deliveries and Deliver Them [G4;G9]
G4: Approach Nurse and Pick Sample [G5;G6;G7;G8]
G9: Approach Arm and Unload Sample [G10;G11;G12;G13]
G5: Fetch Delivery's Nurse
G6: Fetch Delivery Pickup Location
G7: Approach the Nurse
G8: Pick the Sample With the Nurse
G10: Fetch Pharmacy Location
G11: Fetch Pharmacy Arm
G12: Approach the Arm
G13: Unload the Sample
AT1: ApproachNurse
AT2: PickSample
AT3: ApproachArm
AT4: UnloadSample

Figure 5.6: Goal model for the Lab Samples Logistics mission

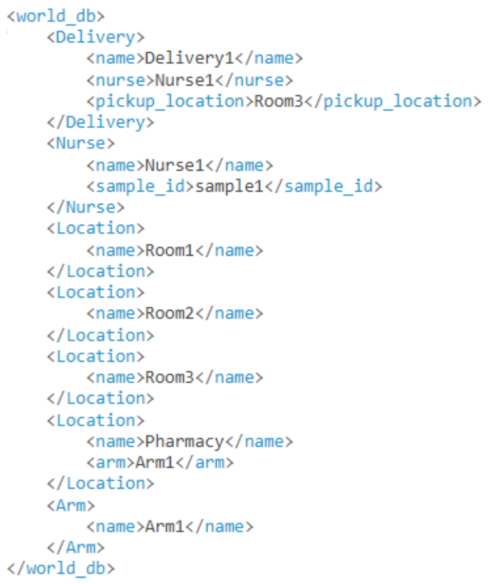
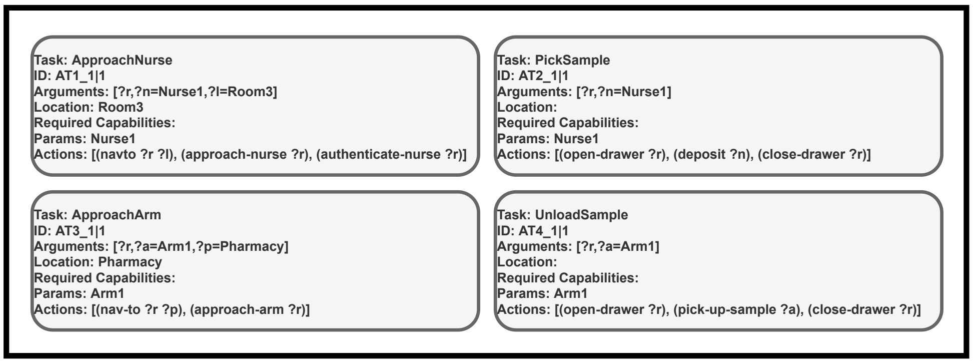

```
<world_db>
    <Delivery>
        <name>Delivery1</name>
        <nurse>Nurse1</nurse>
        <pickup_location>Room3</pickup_location>
    </Delivery>
    <Nurse>
        <name>Nurse1</name>
        <sample_id>sample1</sample_id>
    </Nurse>
    <Location>
        <name>Room1</name>
    </Location>
    <Location>
        <name>Room2</name>
    </Location>
    <Location>
        <name>Room3</name>
    </Location>
    <Location>
        <name>Pharmacy</name>
        <arm>Arm1</arm>
    </Location>
    <Arm>
        <name>Arm1</name>
    </Arm>
</world_db>
```

Figure 5.7: World Knowledge for the Lab Samples Logistics mission

The valid mission decompositions and the constraints are shown in Figure 5.9. Since we have no variability in this example, given that we have no OR decompositions in the goal model and each task in HDDL has exactly one method, we end up with only one valid mission decomposition which contains all of the generated task instances. We have 3


Task: ApproachNurse
ID: AT1_1|1
Arguments: [?r,?n=Nurse1,?l=Room3]
Location: Room3
Required Capabilities:
Params: Nurse1
Actions: [(navto ?r ?l), (approach-nurse ?r), (authenticate-nurse ?r)]

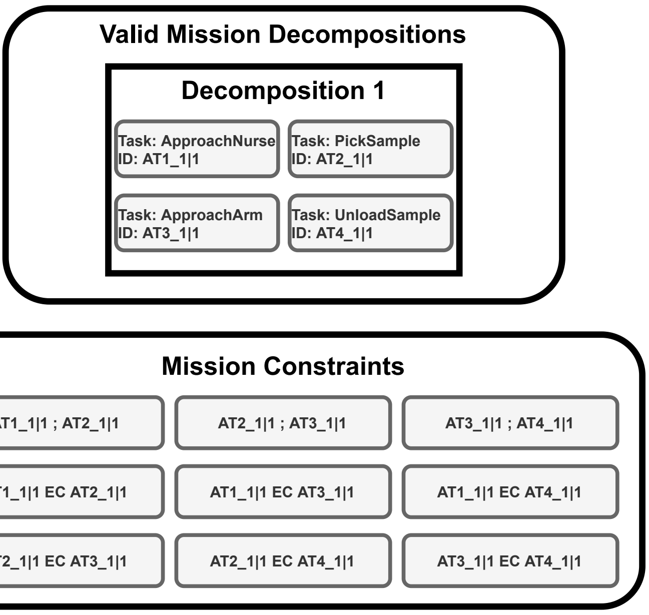
Task: PickSample
ID: AT2_1|1
Arguments: [?r,?n=Nurse1]
Location:
Required Capabilities:
Params: Nurse1
Actions: [(open-drawer ?r), (deposit ?n), (close-drawer ?r)]

Task: ApproachArm
ID: AT3_1|1
Arguments: [?r,?a=Arm1,?p=Pharmacy]
Location: Pharmacy
Required Capabilities:
Params: Arm1
Actions: [(nav-to ?r ?p), (approach-arm ?r)]

Task: UnloadSample
ID: AT4_1|1
Arguments: [?r,?a=Arm1]
Location:
Required Capabilities:
Params: Arm1
Actions: [(open-drawer ?r), (pick-up-sample ?a), (close-drawer ?r)]

Figure 5.8: Task instances for the Lab Samples Logistics mission

sequential constraints, generated following the order from left to right in the goal model, and 6 execution constraints. These execution constraints exist since G3 is a non-group goal, which leads us to end up with all of the combinations of task decompositions taken two by two in order to generate this kind of constraint.


Valid Mission Decompositions

Decomposition 1

Task: ApproachNurse
ID: AT1_1|1

Task: PickSample
ID: AT2_1|1

Task: ApproachArm
ID: AT3_1|1

Task: UnloadSample
ID: AT4_1|1

Mission Constraints

AT1_1|1 ; AT2_1|1	AT2_1|1 ; AT3_1|1	AT3_1|1 ; AT4_1|1
AT1_1|1 EC AT2_1|1	AT1_1|1 EC AT3_1|1	AT1_1|1 EC AT4_1|1
AT2_1|1 EC AT3_1|1	AT2_1|1 EC AT4_1|1	AT3_1|1 EC AT4_1|1

Figure 5.9: Mission decompositions and constraints for the Lab Samples Logistics mission

5.1.3 Deliver Goods/Equipment

The Deliver Goods/Equipment (DGE) mission consists in a hospital domain where robots need to retrieve objects from storages to requesting agents. The description of the mission is shown below, where red colored sentences are the ones that are not captured by the models.

When required, a robot must collect the required resources in the storage and deliver them to the requesting agent in a specified location. In the collection phase, the robot must go to the storage places where the resources can be found. The order to be followed is defined by the estimation of waiting time summed up with the path to be run by the robot. Once the robot reaches one storage and it is time to request the resource, the robot sends a message to the storage with the precise specification of the requested resources and waits until the resources have been retrieved. Once retrieved, the delivery phase begins. In this phase, the robot will make as many runs as necessary to take all the resources to the specified location. If the battery of the assigned robot runs low (10%) in the collection phase, the robot goes back to the recharging station and assigns the mission to another robot. If the battery of the assigned robot runs low (30%) in the delivery phase, the robot must return the resource to a checkpoint and assign the remaining task to another robot, which will know where the resource is positioned. In case of failure to return the resource to a checkpoint, an alert must be triggered and the report sent to the sector manager. When multiple items are required from different storages, multiple robots can be assigned to parallel collect-deliver tasks to reduce the time to finish the mission. Examples of goods / equipments are: (i) sterile medical equipment Logistics, which should be transported from sterile facilities to use destinations, and (ii) clean linens, which should periodically be moved from the laundry to storage rooms close to where it should be used.

The goal model that represents this mission is shown in Figure 5.10. The world knowledge used to test the decomposition is shown in Figure 5.11a alongside the generated task instances, which are shown in 5.11b. One can verify that the task instances were generated correctly since we have tasks related only to one delivery and the two objects that are requested in it, which are located in Storage's 1 and 2, respectively, and because we have:

- One instance of task GetObject for the SterileEquipment, at Storage1, and one for the CleanLinens, at Storage2
- One instance of the RechargeBattery task for each GetObject instance
- One instance of the DeliverObjects, ReturnObjectsToCheckpoint and AlertTrigger tasks, which are related to the only delivery that the system needs to perform

The valid mission decompositions, alongside generated constraints, are shown in Figure 5.12. Note that we have only one valid mission decomposition which contains all of the generated task instances since we don't have variability neither at goal model level or

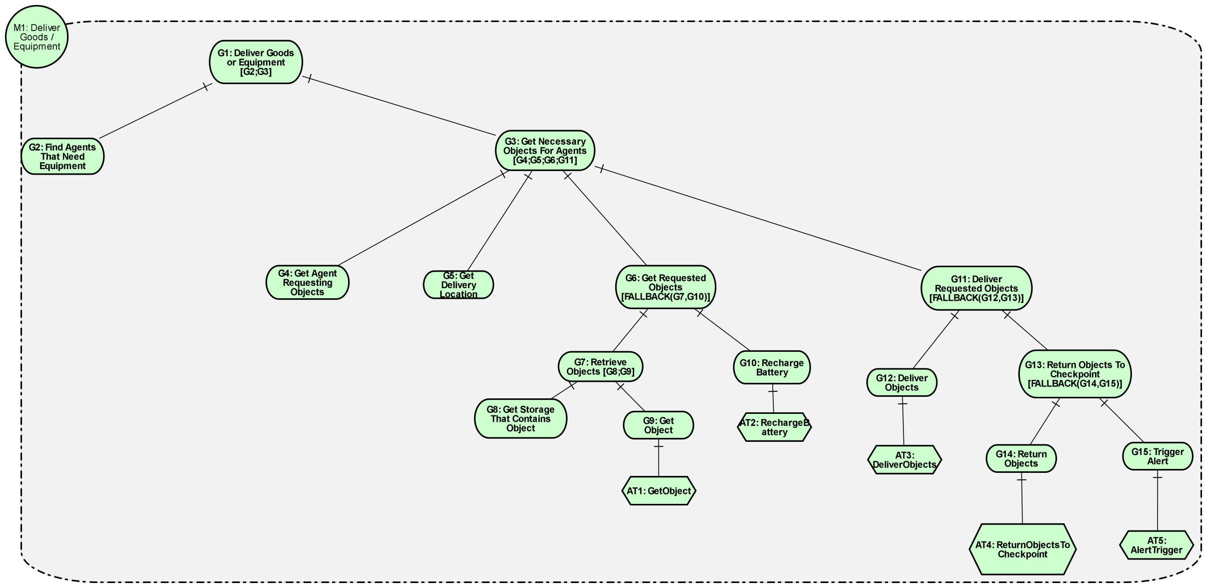
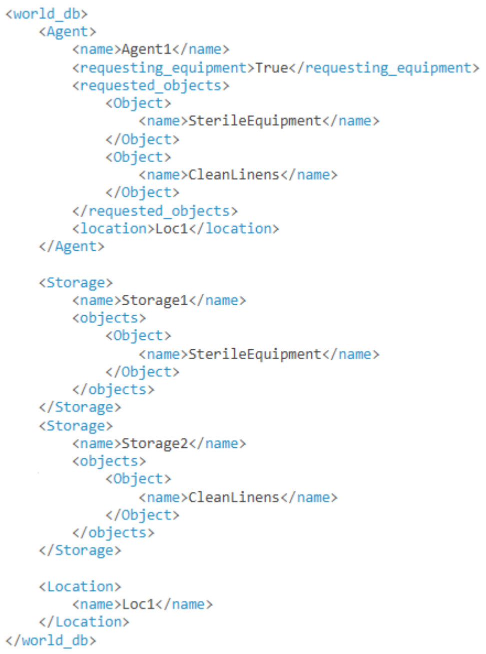

M1: Deliver Goods / Equipment
G1: Deliver Goods or Equipment [G2;G3]
G2: Find Agents That Need Equipment
G3: Get Necessary Objects For Agents [G4;G5;G6;G11]
G4: Get Agent Requesting Objects
G5: Get Delivery Location
G6: Get Requested Objects [FALLBACK(G7,G10)]
G11: Deliver Requested Objects [FALLBACK(G12,G13)]
G7: Retrieve Objects [G8;G9]
G10: Recharge Battery
G12: Deliver Objects
G13: Return Objects To Checkpoint [FALLBACK(G14,G15)]
G8: Get Storage That Contains Object
G9: Get Object
AT2: RechargeBattery
AT3: DeliverObjects
G14: Return Objects
G15: Trigger Alert
AT1: GetObject
AT4: ReturnObjectsTo Checkpoint
AT5: AlertTrigger

Figure 5.10: Goal model for the Deliver Goods/Equipment mission

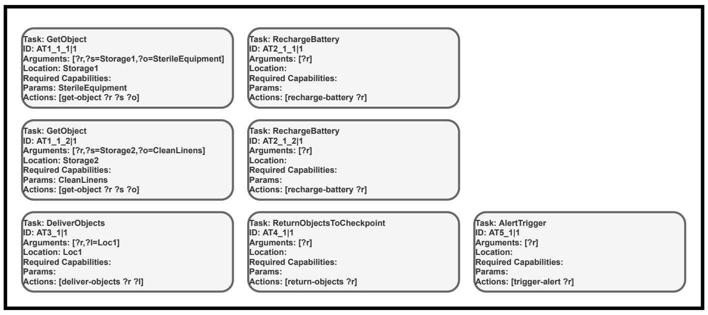


```
<world_db>
    <Agent>
        <name>Agent1</name>
        <requesting_equipment>True</requesting_equipment>
        <requested_objects>
            <Object>
                <name>SterileEquipment</name>
            </Object>
            <Object>
                <name>CleanLinens</name>
            </Object>
        </requested_objects>
        <location>Loc1</location>
    </Agent>

    <Storage>
        <name>Storage1</name>
        <objects>
            <Object>
                <name>SterileEquipment</name>
            </Object>
        </objects>
    </Storage>
    <Storage>
        <name>Storage2</name>
        <objects>
            <Object>
                <name>CleanLinens</name>
            </Object>
        </objects>
    </Storage>

    <Location>
        <name>Loc1</name>
    </Location>
</world_db>
```

(a)



Task: GetObject
ID: AT1_1_1|1
Arguments: [?r,?s=Storage1,?o=SterileEquipment]
Location: Storage1
Required Capabilities:
Params: SterileEquipment
Actions: [get-object ?r ?s ?o]

Task: RechargeBattery
ID: AT2_1_1|1
Arguments: [?r]
Location:
Required Capabilities:
Params:
Actions: [recharge-battery ?r]

Task: GetObject
ID: AT1_1_2|1
Arguments: [?r,?s=Storage2,?o=CleanLinens]
Location: Storage2
Required Capabilities:
Params: CleanLinens
Actions: [get-object ?r ?s ?o]

Task: RechargeBattery
ID: AT2_1_2|1
Arguments: [?r]
Location:
Required Capabilities:
Params:
Actions: [recharge-battery ?r]

Task: DeliverObjects
ID: AT3_1|1
Arguments: [?r,?l=Loc1]
Location: Loc1
Required Capabilities:
Params:
Actions: [deliver-objects ?r ?l]

Task: ReturnObjectsToCheckpoint
ID: AT4_1|1
Arguments: [?r]
Location:
Required Capabilities:
Params:
Actions: [return-objects ?r]

Task: AlertTrigger
ID: AT5_1|1
Arguments: [?r]
Location:
Required Capabilities:
Params:
Actions: [trigger-alert ?r]

(b)

Figure 5.11: Knowledge and generated task instances for the Deliver Goods/Equipment mission

HDDL level. Also, we generate four sequential constraints and four fallback constraints. These constraint are correctly generated since (i) for every agent requesting objects we have that getting the objects happens before delivering them (sequential constraint), (ii) for every object to be retrieved we have a fallback constraint between getting the object and recharging the battery, and (iii) for every agent we have fallback constraints between delivering the objects, returning them to a checkpoint and triggering an alert.

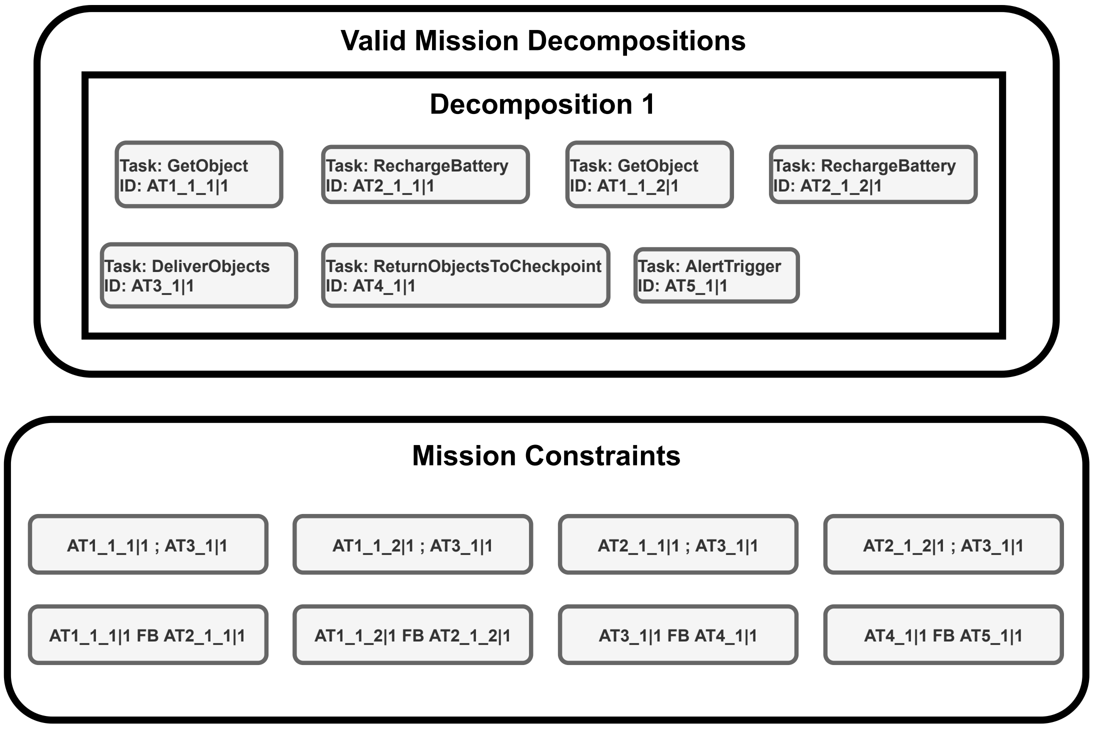

Valid Mission Decompositions
Decomposition 1
Task: GetObject ID: AT1_1_1|1
Task: RechargeBattery ID: AT2_1_1|1
Task: GetObject ID: AT1_1_2|1
Task: RechargeBattery ID: AT2_1_2|1
Task: DeliverObjects ID: AT3_1|1
Task: ReturnObjectsToCheckpoint ID: AT4_1|1
Task: AlertTrigger ID: AT5_1|1
Mission Constraints
AT1_1_1|1 ; AT3_1|1
AT1_1_2|1 ; AT3_1|1
AT2_1_1|1 ; AT3_1|1
AT2_1_2|1 ; AT3_1|1
AT1_1_1|1 FB AT2_1_1|1
AT1_1_2|1 FB AT2_1_2|1
AT3_1|1 FB AT4_1|1
AT4_1|1 FB AT5_1|1

Figure 5.12: Mission decompositions and constraints for the Deliver Goods/Equipment mission

5.1.4 Food Logistics

The Food Logistics (FL) example consists in two separate missions: (i) Delivery of Food and (ii) Pickup of Dirty Dishes. We assume there is some entity at runtime that will be able to coordinate the MRS to perform these missions periodically and we only need to be concerned about the behavior specific to each one of them. The description of this example is shown below, where red colored sentences are the ones that are not captured by the models. The knowledge used for both missions is the same and is shown in Figure 5.13.

The system should deliver food from the kitchen to an inpatient room. The delivery is made on an order-by-order basis in response to kitchen delivery requests. The food can be delivered into a room table by the robot (requires a special manipulation robot skill) or the food can be fetched from the robot's tray. Fetching from the robot tray requires cooperation with the inpatient, a companion, a nurse, or another robot. Some inpatients could be unable to fetch the food from the tray, and a companion visitor or nurse could not be available at the moment of the delivery. The information about if the inpatient, a companion visitor or nurse can be able to retrieve the food from the tray can be obtained based on the inpatient record according to information about the patient and the presence of a companion into the room. This information is subject to uncertainty. The robot can carry multiple meals, in the case of a person retrieving the meal the robot should indicate which meal should be retrieved by a given inpatient. It should track when and where each meal was retrieved. The robot should alert if a wrong meal is retrieved. Dirty dishes should be retrieved from the room. As with delivery, the dish retrieve can occur unassisted, with the cooperation of two robots or with the cooperation of robots and a human. Opening the room door can involve cooperation with another robot or a human. Someone in the room can call a robot to pick up the dishes. In that case, that person can signal if they can open the door when the robot comes to it.

5.1.4.1 Food Logistics Delivery

The goal model that represents the Delivery mission is shown in Figure 5.14. The task instances generated from this goal model alongside the HDDL Domain Definition and the world knowledge are shown in Figure 5.15. One can verify that task instances were generated correctly since we have tasks only for patients that need delivery of food that are Patients 1 and 3, which are located in Room's A and C, respectively, and because we have:

- One instance of task GetFood for RoomA and RoomC, since this task has only one way to be decomposed
- One instance of task DeliverToTable for RoomA and RoomC, since this task also has only one way to be decomposed
- Two instances of task DeliverToFetch for RoomA and RoomC, since this task can be decomposed in two ways given that the delivery with fetch can be performed with the cooperation of a robot and a human or with the cooperation of two robots

```xml
<world_db>
    <Delivery>
        <name>Delivery1</name>
        <patient>Patient1</patient>
        <delivery_location>RoomA</delivery_location>
    </Delivery>
    <Delivery>
        <name>Delivery2</name>
        <patient>Patient3</patient>
        <delivery_location>RoomC</delivery_location>
    </Delivery>

    <Pickup>
        <name>Pickup1</name>
        <patient>Patient1</patient>
        <pickup_location>RoomA</pickup_location>
    </Pickup>
    <Pickup>
        <name>Pickup2</name>
        <patient>Patient3</patient>
        <pickup_location>RoomC</pickup_location>
    </Pickup>

    <Location>
        <name>RoomA</name>
        <patient>Patient1</patient>
    </Location>
    <Location>
        <name>RoomB</name>
        <patient>Patient2</patient>
    </Location>
    <Location>
        <name>RoomC</name>
        <patient>Patient3</patient>
    </Location>
    <Location>
        <name>Kitchen</name>
    </Location>

    <Patient>
        <name>Patient1</name>
        <can_fetch>True</can_fetch>
        <can_open>False</can_open>
    </Patient>
    <Patient>
        <name>Patient2</name>
        <can_fetch>False</can_fetch>
        <can_open>False</can_open>
    </Patient>
    <Patient>
        <name>Patient3</name>
        <can_fetch>False</can_fetch>
        <can_open>True</can_open>
    </Patient>
</world_db>
```

Figure 5.13: Knowledge for the Food Logistics example

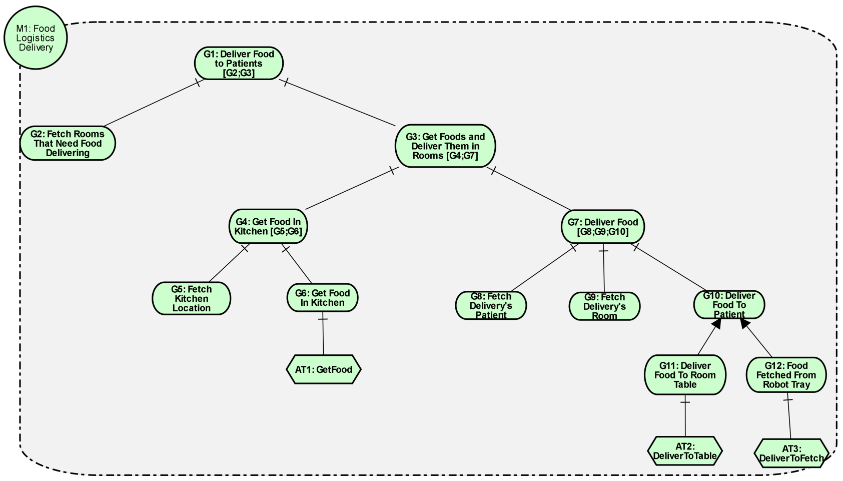
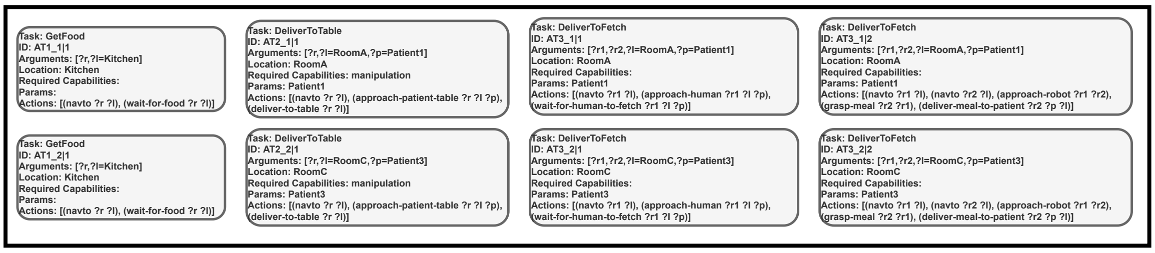

M1: Food Logistics Delivery
G1: Deliver Food to Patients [G2;G3]
G2: Fetch Rooms That Need Food Delivering
G3: Get Foods and Deliver Them in Rooms [G4;G7]
G4: Get Food In Kitchen [G5;G6]
G7: Deliver Food [G8;G9;G10]
G5: Fetch Kitchen Location
G6: Get Food In Kitchen
G8: Fetch Delivery's Patient
G9: Fetch Delivery's Room
G10: Deliver Food To Patient
AT1: GetFood
G11: Deliver Food To Room Table
G12: Food Fetched From Robot Tray
AT2: DeliverToTable
AT3: DeliverToFetch

Figure 5.14: Goal model for the Food Logistics Delivery mission



Task: GetFood
ID: AT1_1|1
Arguments: [?r,?l=Kitchen]
Location: Kitchen
Required Capabilities:
Params:
Actions: [(navto ?r ?l), (wait-for-food ?r ?l)]

Task: DeliverToTable
ID: AT2_1|1
Arguments: [?r,?l=RoomA,?p=Patient1]
Location: RoomA
Required Capabilities: manipulation
Params: Patient1
Actions: [(navto ?r ?l), (approach-patient-table ?r ?l ?p), (deliver-to-table ?r ?l)]

Task: DeliverToFetch
ID: AT3_1|1
Arguments: [?r1,?r2,?l=RoomA,?p=Patient1]
Location: RoomA
Required Capabilities:
Params: Patient1
Actions: [(navto ?r1 ?l), (approach-human ?r1 ?l ?p), (wait-for-human-to-fetch ?r1 ?l ?p)]

Task: DeliverToFetch
ID: AT3_1|2
Arguments: [?r1,?r2,?l=RoomA,?p=Patient1]
Location: RoomA
Required Capabilities:
Params: Patient1
Actions: [(navto ?r1 ?l), (navto ?r2 ?l), (approach-robot ?r1 ?r2), (grasp-meal ?r2 ?r1), (deliver-meal-to-patient ?r2 ?p ?l)]

Task: GetFood
ID: AT1_2|1
Arguments: [?r,?l=Kitchen]
Location: Kitchen
Required Capabilities:
Params:
Actions: [(navto ?r ?l), (wait-for-food ?r ?l)]

Task: DeliverToTable
ID: AT2_2|1
Arguments: [?r,?l=RoomC,?p=Patient3]
Location: RoomC
Required Capabilities: manipulation
Params: Patient3
Actions: [(navto ?r ?l), (approach-patient-table ?r ?l ?p), (deliver-to-table ?r ?l)]

Task: DeliverToFetch
ID: AT3_2|1
Arguments: [?r1,?r2,?l=RoomC,?p=Patient3]
Location: RoomC
Required Capabilities:
Params: Patient3
Actions: [(navto ?r1 ?l), (approach-human ?r1 ?l ?p), (wait-for-human-to-fetch ?r1 ?l ?p)]

Task: DeliverToFetch
ID: AT3_2|2
Arguments: [?r1,?r2,?l=RoomC,?p=Patient3]
Location: RoomC
Required Capabilities:
Params: Patient3
Actions: [(navto ?r1 ?l), (navto ?r2 ?l), (approach-robot ?r1 ?r2), (grasp-meal ?r2 ?r1), (deliver-meal-to-patient ?r2 ?p ?l)]

Figure 5.15: Task instances for the Food Logistics Delivery mission

The valid mission decompositions are shown in Figure 5.16, where task with ID AT3_2|1 is not present in any of them. This is the case because this task represents the DeliverToFetch task where a human fetches the food, which cannot happen with Patient3 at RoomC since we know by the world knowledge that this patient is not able to do so. The generated constraints for this example are shown in Figure 5.17, where we have 6 sequential constraints and 6 execution constraints, given that G3 is a group and non-divisible goal. We also have a non-group constraint in G4, which leads to instances of the GetFood task being non-group tasks. Notice that we generate sequential and execution constraints between the only decomposition for the GetFood task and all of the decompositions for the other two tasks, given that they relate to same patient (i.e., happen in the same room, since we have one patient for each room).

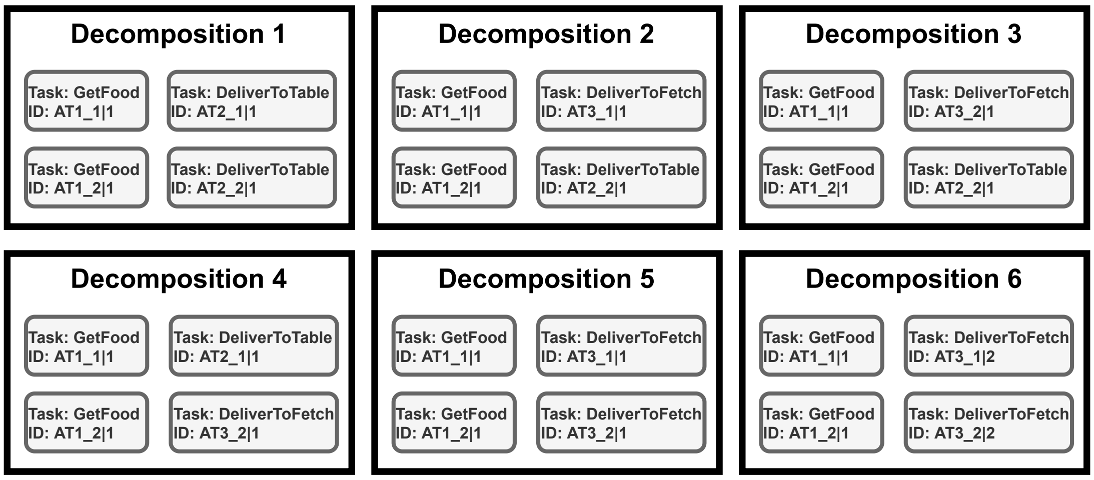

Decomposition 1
Task: GetFood ID: AT1_1|1
Task: DeliverToTable ID: AT2_1|1
Task: GetFood ID: AT1_2|1
Task: DeliverToTable ID: AT2_2|1
Decomposition 2
Task: GetFood ID: AT1_1|1
Task: DeliverToFetch ID: AT3_1|1
Task: GetFood ID: AT1_2|1
Task: DeliverToTable ID: AT2_2|1
Decomposition 3
Task: GetFood ID: AT1_1|1
Task: DeliverToFetch ID: AT3_2|1
Task: GetFood ID: AT1_2|1
Task: DeliverToTable ID: AT2_2|1
Decomposition 4
Task: GetFood ID: AT1_1|1
Task: DeliverToTable ID: AT2_1|1
Task: GetFood ID: AT1_2|1
Task: DeliverToFetch ID: AT3_2|1
Decomposition 5
Task: GetFood ID: AT1_1|1
Task: DeliverToFetch ID: AT3_1|1
Task: GetFood ID: AT1_2|1
Task: DeliverToFetch ID: AT3_2|1
Decomposition 6
Task: GetFood ID: AT1_1|1
Task: DeliverToFetch ID: AT3_1|2
Task: GetFood ID: AT1_2|1
Task: DeliverToFetch ID: AT3_2|2

Figure 5.16: Mission decompositions for the Food Logistics Delivery mission

5.1.4.2 Food Logistics Pickup

The goal model that represents the Pickup mission is shown in Figure 5.18. The task instances generated from this goal model alongside the HDDL Domain Definition and the world knowledge are shown in Figure 5.19. One can verify that task instances were generated correctly since we have tasks only for patients that need pickup of dishes that

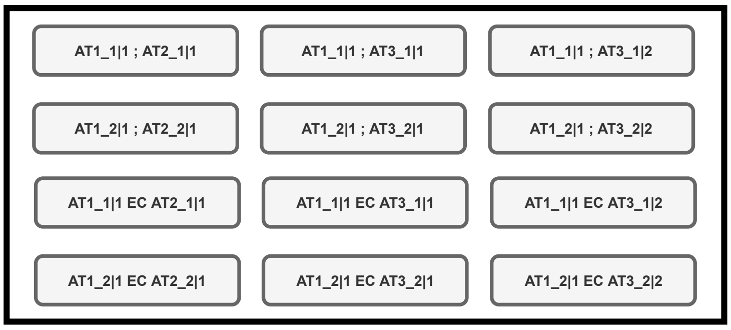


Figure 5.17: Generated constraints for the Food Logistics Delivery mission

are Patients 1 and 3, which are located in Room's A and C, respectively, and because we have:

- Four instances of task PickupDishes for RoomA and RoomC, since this task can be decomposed into:
  - PickupDishes with a human opening the door and the cooperation of the robot with a human to pick up the dishes
  - PickupDishes with a human opening the door and the cooperation of the robot with another robot to pick up the dishes
  - PickupDishes with the cooperation of the robot with another robot to open the door and with a human to pick up the dishes
  - PickupDishes with the cooperation of the robot with another robot to open the door and to pick up the dishes
- One instance of task RetriveDishes for RoomA and RoomC, since this task has only one way of being decomposed

The valid mission decompositions are shown in Figure 5.20, where tasks with ID's AT1_1|3 and AT1_1|4 are not present in any of them. This is the case since both of these task instances represent the PickupDishes task where a human opens the door, which is not possible in RoomA since we know from the world knowledge that Patient1 is not able to do so. The generated constraints for this example are shown in Figure 5.21, where we have 8 sequential constraints and 8 execution constraints, given that G3 is a group and non-divisible goal. Notice that we generate constraints of all of the decompositions for task PickupDishes and the decomposition for task RetrieveDishes, given that they relate to same patient (i.e., happen in the same room, since we have one patient for each room).

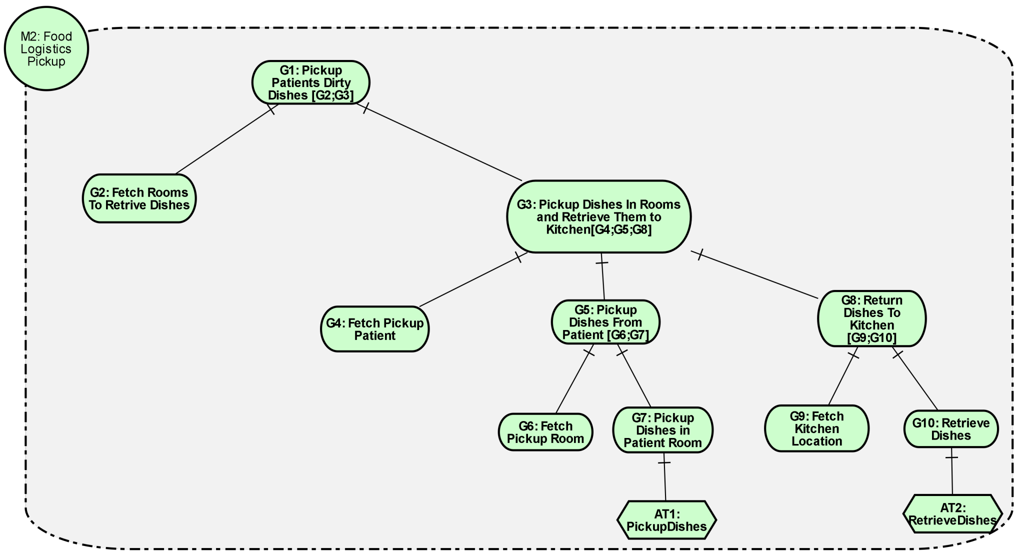
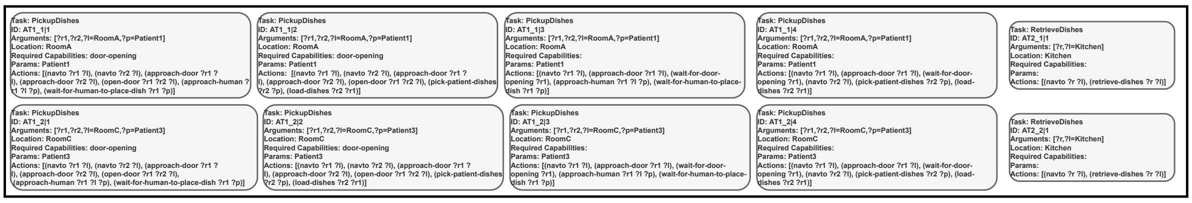

M2: Food Logistics Pickup
G1: Pickup Patients Dirty Dishes [G2;G3]
G2: Fetch Rooms To Retrive Dishes
G3: Pickup Dishes In Rooms and Retrieve Them to Kitchen[G4;G5;G8]
G4: Fetch Pickup Patient
G5: Pickup Dishes From Patient [G6;G7]
G8: Return Dishes To Kitchen [G9;G10]
G6: Fetch Pickup Room
G7: Pickup Dishes in Patient Room
G9: Fetch Kitchen Location
G10: Retrieve Dishes
AT1: PickupDishes
AT2: RetrieveDishes

Figure 5.18: Goal model for the Food Logistics Pickup mission


Task: PickupDishes
ID: AT1_1|1
Arguments: [?r1,?r2,?l=RoomA,?p=Patient1]
Location: RoomA
Required Capabilities: door-opening
Params: Patient1
Actions: [(navto ?r1 ?l), (navto ?r2 ?l), (approach-door ?r1 ?l), (approach-door ?r2 ?l), (open-door ?r1 ?r2 ?l), (approach-human ?r1 ?l ?p), (wait-for-human-to-place-dish ?r1 ?p)]

Task: PickupDishes
ID: AT1_1|2
Arguments: [?r1,?r2,?l=RoomA,?p=Patient1]
Location: RoomA
Required Capabilities: door-opening
Params: Patient1
Actions: [(navto ?r1 ?l), (navto ?r2 ?l), (approach-door ?r1 ?l), (approach-door ?r2 ?l), (open-door ?r1 ?r2 ?l), (pick-patient-dishes ?r2 ?p), (load-dishes ?r2 ?r1)]

Task: PickupDishes
ID: AT1_1|3
Arguments: [?r1,?r2,?l=RoomA,?p=Patient1]
Location: RoomA
Required Capabilities:
Params: Patient1
Actions: [(navto ?r1 ?l), (approach-door ?r1 ?l), (wait-for-door-opening ?r1), (approach-human ?r1 ?l ?p), (wait-for-human-to-place-dish ?r1 ?p)]

Task: PickupDishes
ID: AT1_1|4
Arguments: [?r1,?r2,?l=RoomA,?p=Patient1]
Location: RoomA
Required Capabilities:
Params: Patient1
Actions: [(navto ?r1 ?l), (approach-door ?r1 ?l), (wait-for-door-opening ?r1), (navto ?r2 ?l), (pick-patient-dishes ?r2 ?p), (load-dishes ?r2 ?r1)]

Task: RetrieveDishes
ID: AT2_1|1
Arguments: [?r,?l=Kitchen]
Location: Kitchen
Required Capabilities:
Params:
Actions: [(navto ?r ?l), (retrieve-dishes ?r ?l)]

Task: PickupDishes
ID: AT1_2|1
Arguments: [?r1,?r2,?l=RoomC,?p=Patient3]
Location: RoomC
Required Capabilities: door-opening
Params: Patient3
Actions: [(navto ?r1 ?l), (navto ?r2 ?l), (approach-door ?r1 ?l), (approach-door ?r2 ?l), (open-door ?r1 ?r2 ?l), (approach-human ?r1 ?l ?p), (wait-for-human-to-place-dish ?r1 ?p)]

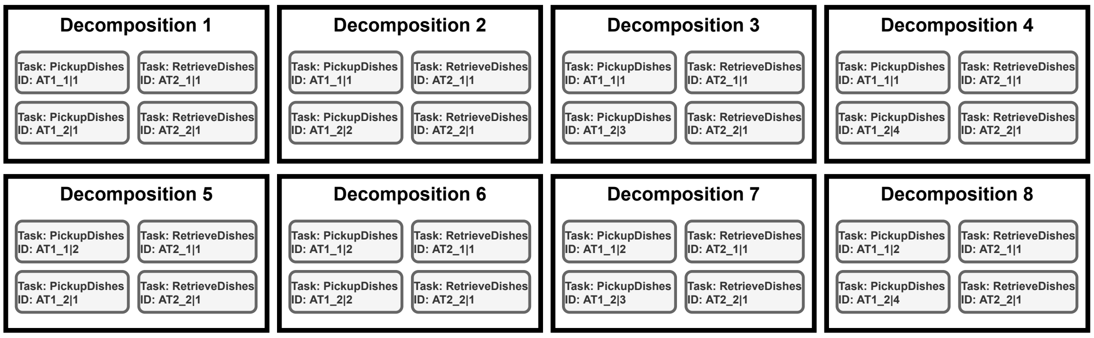
Task: PickupDishes
ID: AT1_2|2
Arguments: [?r1,?r2,?l=RoomC,?p=Patient3]
Location: RoomC
Required Capabilities: door-opening
Params: Patient3
Actions: [(navto ?r1 ?l), (navto ?r2 ?l), (approach-door ?r1 ?l), (approach-door ?r2 ?l), (open-door ?r1 ?r2 ?l), (pick-patient-dishes ?r2 ?p), (load-dishes ?r2 ?r1)]

Task: PickupDishes
ID: AT1_2|3
Arguments: [?r1,?r2,?l=RoomC,?p=Patient3]
Location: RoomC
Required Capabilities:
Params: Patient3
Actions: [(navto ?r1 ?l), (approach-door ?r1 ?l), (wait-for-door-opening ?r1), (approach-human ?r1 ?l ?p), (wait-for-human-to-place-dish ?r1 ?p)]

Task: PickupDishes
ID: AT1_2|4
Arguments: [?r1,?r2,?l=RoomC,?p=Patient3]
Location: RoomC
Required Capabilities:
Params: Patient3
Actions: [(navto ?r1 ?l), (approach-door ?r1 ?l), (wait-for-door-opening ?r1), (navto ?r2 ?l), (pick-patient-dishes ?r2 ?p), (load-dishes ?r2 ?r1)]

Task: RetrieveDishes
ID: AT2_2|1
Arguments: [?r,?l=Kitchen]
Location: Kitchen
Required Capabilities:
Params:
Actions: [(navto ?r ?l), (retrieve-dishes ?r ?l)]

Figure 5.19: Task instances for the Food Logistics Pickup mission


Decomposition 1
Task: PickupDishes ID: AT1_1|1 — Task: RetrieveDishes ID: AT2_1|1
Task: PickupDishes ID: AT1_2|1 — Task: RetrieveDishes ID: AT2_2|1

Decomposition 2
Task: PickupDishes ID: AT1_1|1 — Task: RetrieveDishes ID: AT2_1|1
Task: PickupDishes ID: AT1_2|2 — Task: RetrieveDishes ID: AT2_2|1

Decomposition 3
Task: PickupDishes ID: AT1_1|1 — Task: RetrieveDishes ID: AT2_1|1
Task: PickupDishes ID: AT1_2|3 — Task: RetrieveDishes ID: AT2_2|1

Decomposition 4
Task: PickupDishes ID: AT1_1|1 — Task: RetrieveDishes ID: AT2_1|1
Task: PickupDishes ID: AT1_2|4 — Task: RetrieveDishes ID: AT2_2|1

Decomposition 5
Task: PickupDishes ID: AT1_1|2 — Task: RetrieveDishes ID: AT2_1|1
Task: PickupDishes ID: AT1_2|1 — Task: RetrieveDishes ID: AT2_2|1

Decomposition 6
Task: PickupDishes ID: AT1_1|2 — Task: RetrieveDishes ID: AT2_1|1
Task: PickupDishes ID: AT1_2|2 — Task: RetrieveDishes ID: AT2_2|1

Decomposition 7
Task: PickupDishes ID: AT1_1|2 — Task: RetrieveDishes ID: AT2_1|1
Task: PickupDishes ID: AT1_2|3 — Task: RetrieveDishes ID: AT2_2|1

Decomposition 8
Task: PickupDishes ID: AT1_1|2 — Task: RetrieveDishes ID: AT2_1|1
Task: PickupDishes ID: AT1_2|4 — Task: RetrieveDishes ID: AT2_2|1

Figure 5.20: Mission decompositions for the Food Logistics Pickup mission

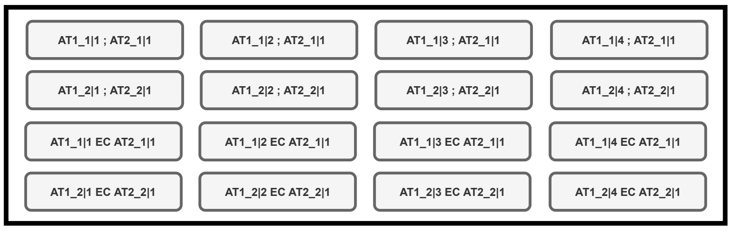

AT1_1|1 ; AT2_1|1	AT1_1|2 ; AT2_1|1	AT1_1|3 ; AT2_1|1	AT1_1|4 ; AT2_1|1
AT1_2|1 ; AT2_2|1	AT1_2|2 ; AT2_2|1	AT1_2|3 ; AT2_2|1	AT1_2|4 ; AT2_2|1
AT1_1|1 EC AT2_1|1	AT1_1|2 EC AT2_1|1	AT1_1|3 EC AT2_1|1	AT1_1|4 EC AT2_1|1
AT1_2|1 EC AT2_2|1	AT1_2|2 EC AT2_2|1	AT1_2|3 EC AT2_2|1	AT1_2|4 EC AT2_2|1

Figure 5.21: Generated constraints for the Food Logistics Pickup mission

5.1.5 Features Analyzed By The Examples

There are several features we want to analyze with these examples, which are as follows:

- Variability at goal model level: This just indicates the presence of OR decompositions in the goal model, which leads to the existence of multiple possible valid mission decompositions
- Variability at HDDL level: This happens when a single abstract task has multiple decomposition methods. These methods may or may not be exclusive, but each one generates a different decomposition for a task and thus leads to the existence of multiple possible valid mission decompositions
- Condition type contexts: This just indicates the presence of condition type contexts
- Trigger type contexts: This just indicates the presence of trigger type contexts
- Fallback operator: This just indicates if we have fallback operators, which generation special kinds of constraints
- Relationship type mapping query: This indicates the presence of a query related to entities which are mapped by a relationship mapping in the configuration file
- Ownership type mapping query: This indicates the presence of a query related to entities which are mapped by a ownership mapping in the configuration file
- HDDL as a library: This relates to the fact where we have more than one goal model (i.e., more than one mission) using the same HDDL Domain

	Work			
Feature	VSM	LSL	DGE	FL
Goal model Variability	✗	✗	✗	✓
HDDL Variability	✓	✗	✗	✓
Condition Contexts	✗	✗	✗	✓
Trigger Contexts	✓	✗	✓	✗
Fallback Operator	✓	✗	✓	✗
Relationship Mapping Query	✓	✗	✓	✓
Ownership Mapping Query	✗	✗	✓	✗
HDDL as a Library	✗	✗	✗	✓

Table 5.1: Features present in the modelled RoboMAX examples

Table 5.1 shows which of the features are evaluated by which example. Note that the Lab Samples Logistics (LSL) does not contain any of them, but it is an interesting example in the sense that its HDDL Domain Definition is more elaborated than most of the examples and thus it indicates that the HTN decomposition step is correctly performed. Also, the LSL example is a good example that the level of detail given in HDDL depends on a design choice, given that the lower-level abstractions and models are the ones which define the actions that can in fact be performed by the robots in the system. Finally, because of this higher level of detail in the HDDL Domain definition, this example is used to demonstrate the generation of iHTNs [5] from the decomposition process as shown in Section 5.2.

5.2 Generation of Examples iHTNs

In order to provide evidence on how this work's output yields sufficient information that can be used in order to perform transformation to other models, we show the results of an automated transformation process for previous modelled RoboMAX examples. In particular, one model of interest is the iHTN [5], which can be used for execution and even for allocation if robots are given as labels that can be replaced by an allocation process. In this sense, we perform the transformation process in three missions: (i) Lab Samples Logistics, (ii) Food Logistics Delivery and (iii) Food Logistics Pickup. These examples were chosen since they provide a more low-level HDDL Domain definition and thus they generate non-trivial iHTN models.

The process for the generation of iHTNs is simple. For each valid mission decomposition we verify the possible combinations of the tasks involved, by using the sequential and fallback constraints, and generate an iHTN for each one of them. Each iHTN has a task called "ROOT", which is assigned to all of the existing agents, and a method "ROOT_M", which decomposes this task in the tasks present in the valid mission decomposition combination being considered. The existing agents are: (i) robots, which are defined based on the tasks and execution constraints between them, and (ii) every task argument that is not equal to the location parameter. Another important thing to mention is that action locations are encoded in their names, where the action name is followed by a "_" and the location name for each location in which the action is performed.

One last note to be given is that, in order to avoid a high number of iHTNs being generated, the world knowledges for each example were reduced in order to have one instance of each task decomposition. This is done since the current approach can only generate sequential iHTNs and task instances generated by forall statements are inherently parallel, which would generate a high number of possible combinations for each valid mission decomposition and, consequently, a high number of iHTNs. All of the files related to this process can be found in a Github repository[2]. In this same repository, one can

[2]https://github.com/ericbg27/RoboMax-iHTN-examples

found the results obtained for the other missions modelled in Section 5.1 that were not illustrated in this section. Finally, the labels for the iHTN illustrations are given in Figure 5.22.

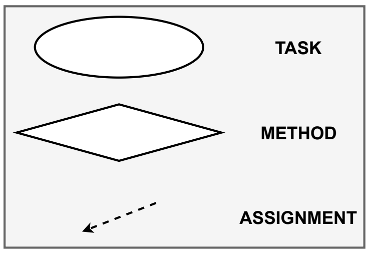


Figure 5.22: Labels for iHTN illustrations

5.2.1 Lab Samples Logistics

For the Lab Samples Logistics the world knowledge was not modified since we only have one sample delivery to be made, as shown in Figure 5.7 (see Section 5.1.2). This led us to generate one iHTN for the only valid mission decomposition possible, which is totally ordered and thus has only one possible ordering, which is illustrated in Figure 5.23. The JSON file generated by the automated iHTN generation process is given in Appendix B.

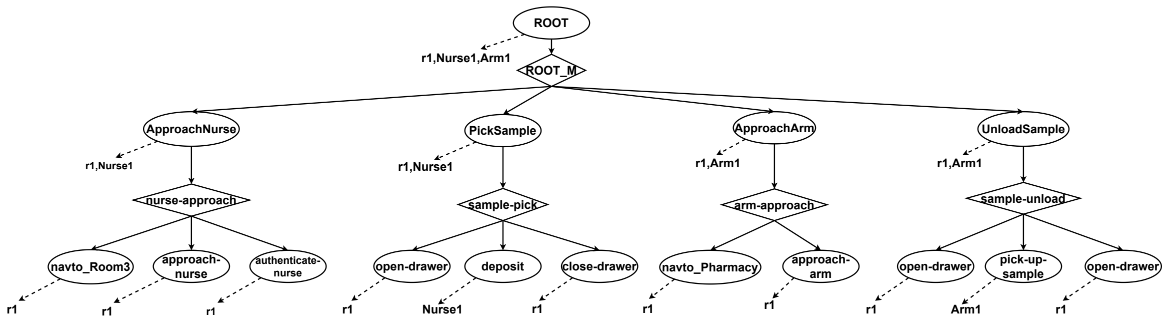


Figure 5.23: Illustration of the iHTN for the Lab Samples Logistics mission

5.2.2 Food Logistics Delivery

For the Food Logistics Delivery the world knowledge was modified in order to contain a single delivery, called *Delivery1*, to happen at location *RoomA* for patient *Patient1*. This led us to generate three different iHTNs since we can choose between executing the *GetFood* task with one decomposition of the *DeliverToTable* task or with one of each of the two decompositions of the *DeliverToFetch* task. The illustrations of these iHTNs are shown in Figures 5.24, 5.25 and 5.26. The JSON files generated by the automated iHTN generation process are given in Appendix B.

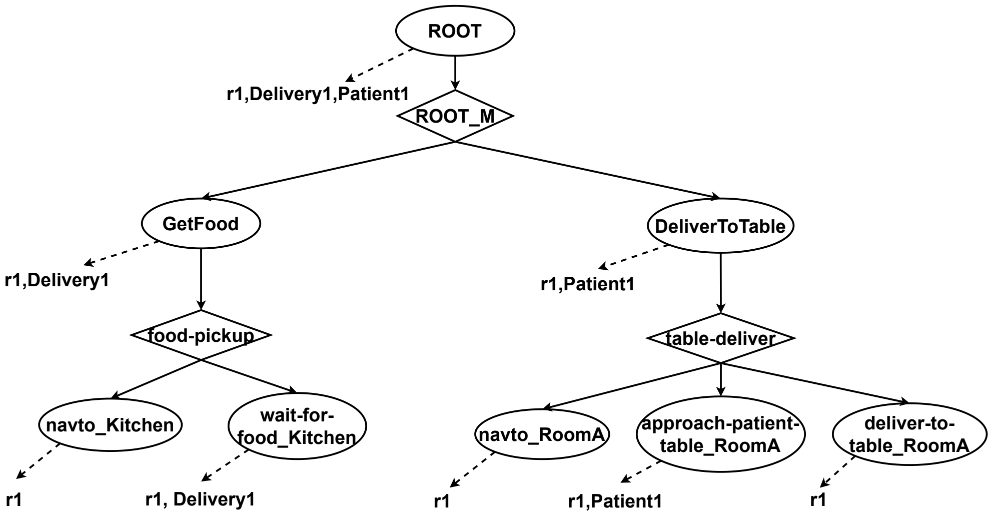


Figure 5.24: Illustration of the first iHTN for the Food Logistics Delivery mission

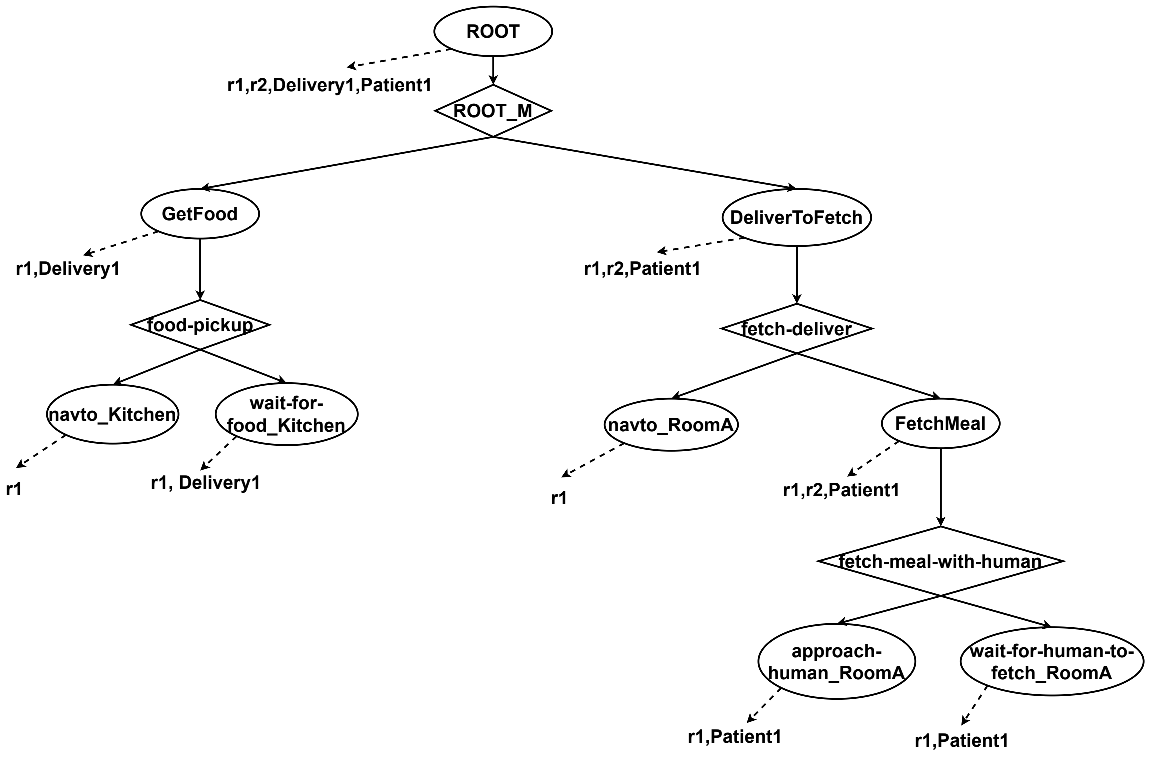


Figure 5.25: Illustration of the second iHTN for the Food Logistics Delivery mission

5.2.3 Food Logistics Pickup

As is the case for the Food Logistics Delivery mission, the world knowledge for the Food Logistics Pickup was also modified in order to contain a single pickup, called *Pickup1*, to happen at location *RoomA* for patient *Patient1*. This led us to generate two different iHTNs since we can choose between executing the PickupDishes task in two different ways, since Patient1 cannot open the door, where each of them is executed alongside the only

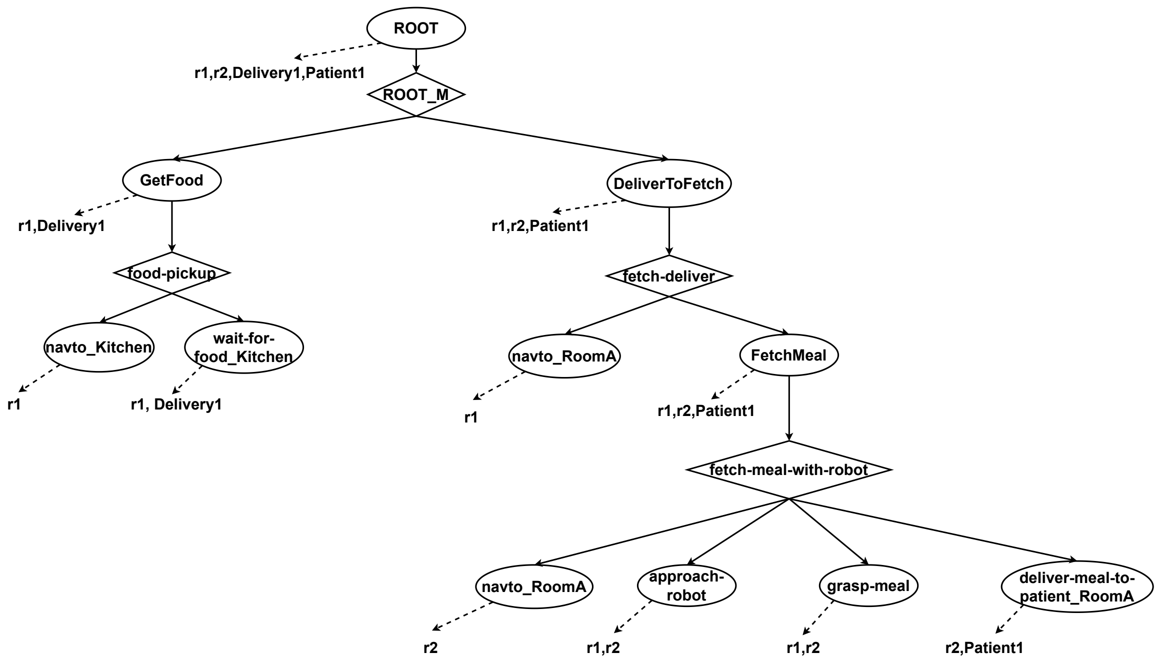


Figure 5.26: Illustration of the third iHTN for the Food Logistics Delivery mission

decomposition for the RetrieveDishes task. The illustration of these iHTNs are shown in Figures 5.27 and 5.28. The JSON files generated by the automated iHTN generation process are given in Appendix B.

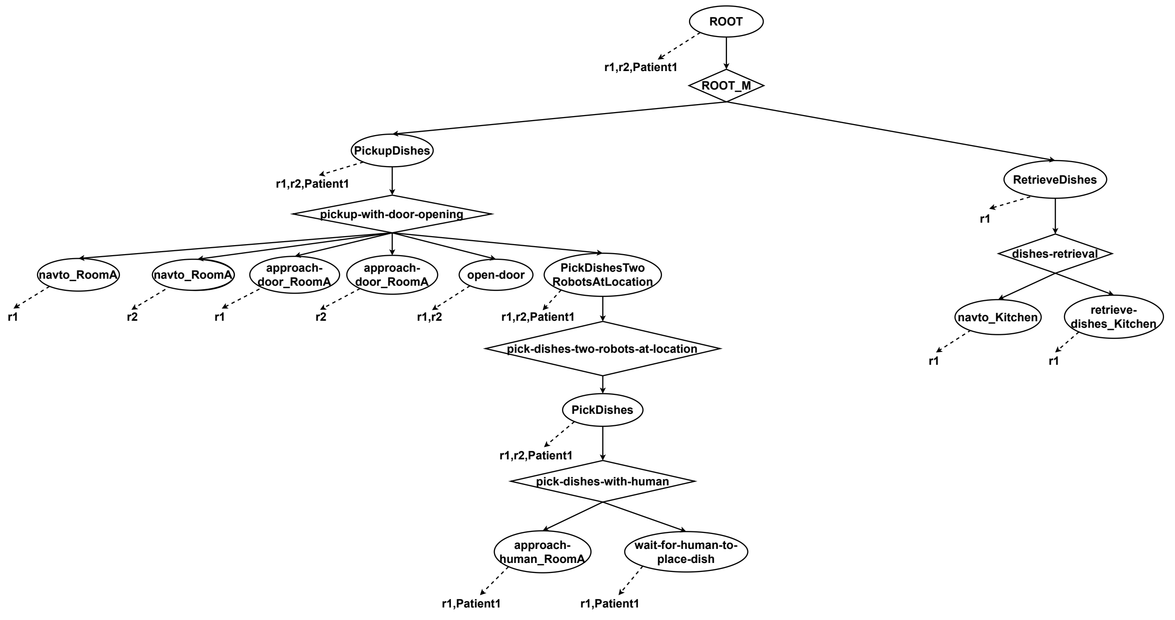


Figure 5.27: Illustration of the first iHTN for the Food Logistics Pickup mission

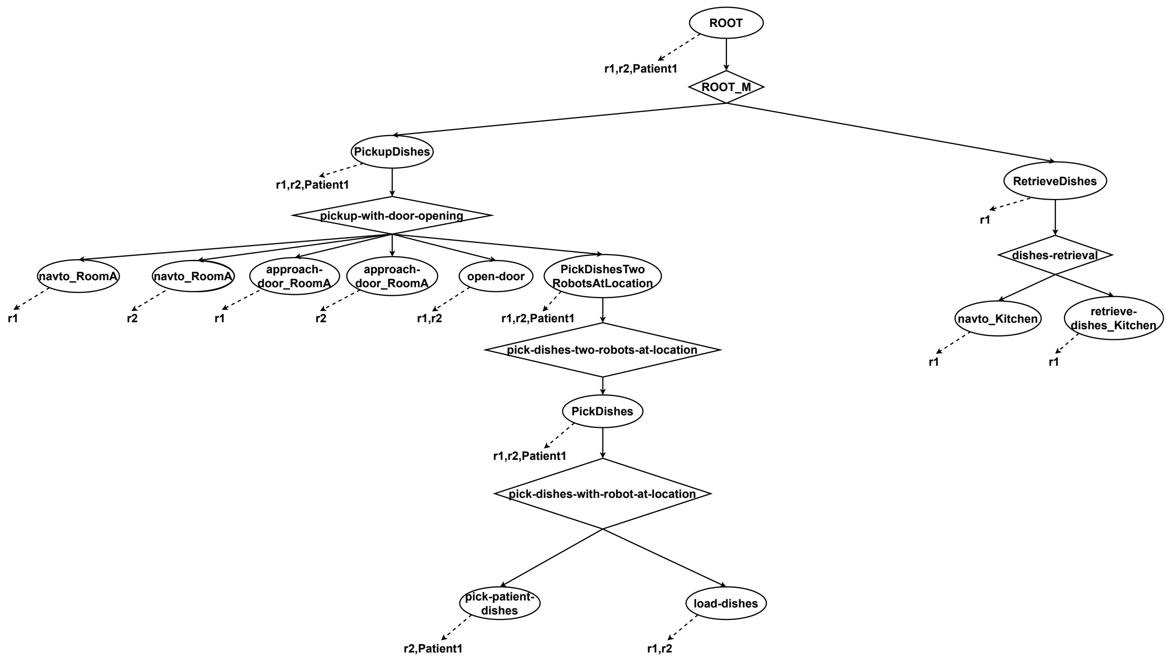


Figure 5.28: Illustration of the second iHTN for the Food Logistics Pickup mission

5.3 Verification of the Performance of the Decomposition

In this evaluation the goal is to show execution times of the decomposition process for different scenarios, in order to show evidence of feasibility of the approach. One must note that this is not the focus of the work since we use an exploratory approach that generates all of the task instances and valid mission decompositions. We perform this evaluation using tree different scenarios from Section 5.1: (i) the Vital Signs Monitoring mission, (ii) the Food Logistics Delivery mission and (iii) the Food Logistics Pickup mission. For these three examples we generate 13 different world knowledges which will be explained further. Every file related to this evaluation can be found in a Github repository[3].

5.3.1 Vital Signs Monitoring

The Vital Signs Monitoring (VSM) mission has no variability when it comes to the generation of valid mission decompositions because even though we have variability at HDDL level in the CollectVitalSigns task, only one decomposition is chosen for each patient that we have since the different decomposition methods refer to different patient conditions. This example consists in rooms which contain patients, where the goal model contains two forall statements. In this sense, for each world knowledge we generate a

[3]https://github.com/ericbg27/MutRoSe-Performance-Tests

number n of rooms where each room contains two patients. The values chosen for n are [2,3,5,10,20,50,100,200,1000,2000,3000,4000,5000] and it is important to note that every room is not occupied and every patient is available.

5.3.2 Food Logistics

The Food Logistics Delivery (FLD) mission has variability at goal model level, since it has an OR decomposition, and is expected to be the most problematic one when it comes to execution time. The Food Logistics Pickup (FLP) mission, on the other hand, has variability at HDDL level and this is considered throughout the valid mission generation process given that each patient that needs dishes to be picked up can open the door. These examples consist in delivery and pickup objects which contain the patients and their respective rooms, where the goal model for both missions contains one forall statement each. In this sense, for each world knowledge we generate a number n of deliveries/pickups, where each delivery/pickup relates to a single patient. The values chosen for n are [2,3,5,10,20,50,100,200,1000,2000,3000,4000,5000] and it is important to note that every patient can fetch the deliveries, for the Delivery mission, and every patient can open the door, for the Pickup mission.

5.3.3 Results

Table 5.2 shows the time results obtained for each execution given the number n. One important note to give is that results with "–" refer to executions that took more than 20 minutes to execute and results with "–" refer to executions that led to a memory overflow. These experiments were ran on a Dell Inspiron 7572 laptop with an Intel Core i7-8550U CPU @ 1.80 GHz processor and 16 GB of RAM, running Ubuntu 20.04 LTS subsystem for Windows in a Windows 10 operating system.

By the obtained results, we can verify that:

- Variability at HDDL level seems to have no big impact in the performance, since results for the Food Logistics Pickup were the better ones
- The number of forall statements in the goal model seems to have some impact on performance, but not a major one. This can be seen given that the growth of time in the Vital Signs Monitoring mission, which contains two forall statements, is greater than that for the Food Logistics Pickup mission, which has only one. This is true if and only if the observation with respect to variability at HDDL level is also true
- Variability at goal model level seems to have a huge impact on performance. This can be verified given the fact that the Food Logistics Delivery mission has: (i) a

	Mission		
n	VSM	FLD	FLP
2	52 ms	86 ms	24 ms
3	54 ms	133 ms	24 ms
5	53 ms	612 ms	27 ms
10	64 ms	394124 ms	30 ms
20	95 ms	–	42 ms
50	318 ms	–	146 ms
100	1106 ms	–	501 ms
200	4202 ms	–	1940 ms
1000	115732 ms	–	49086 ms
2000	536348 ms	–	208701 ms
3000	–	–	516774 ms
4000	–	–	965567 ms
5000	–	–	–

Table 5.2: Results for the performance evaluation. The n value corresponds to: (i) the number of rooms (VSM), (ii) the number of deliveries (FLD) and (iii) the number of pickups (FLP)

high increase rate of the time to decompose, verified when n goes from 5 to 10, and (ii) a high memory consumption, given that when n equals 20 we observed a memory overflow

One last note about this experiment is that the current implementation of the decomposition process does not take advantage of the parallel nature of some steps. This is the case since the steps of the decomposition approach are currently performed sequentially even though some of them can be performed in parallel, as can be observed in Figure **??** (ref. Section 3.4). In this sense, the measured times can differ from the best implementation possible, which takes advantage of those steps that are indeed parallel. Nevertheless, the current obtained values show us that the time complexity is exponential to some degree, which would not change if we changed the implementation.

5.4 Analysis of the Results

In this section we highlight some valuable discussions regarding the expectations and validity of our experimental outcomes.

In the first experiment, we analysed the ability of MutRoSe to express what is given in the missions natural language descriptions and to perform the accurate task decomposition process. We consider these aspects crucial for a deliberation process. To perform such experiment, we chose to model in MutRoSe four modelled mission specifications from the RoboMax artefact [6]. However, given those scenarios were expressed in natural language, their modeling in MutRoSe went through a manual process, which requires the ability to correctly interpret those scenarios and correctly express them in our MutRoSe framework. Therefore, in our first experiment, we validated for correctness of the modelled missions with their respective main RoboMax authors in order to mitigate issues from natural language interpretation. Also, the authors of all of the experiments had some experience with the goal modeling notation and/or HDDL, to isolate the issues that could arise from learning those languages. An attempt to perform an evaluation with another author which had no experience with these notations was carried out. However, due to time constraint the evaluation could not be further accomplished. Nevertheless, the gathered results obtained from this first experiment were positive since all the modelled examples were validated by their respective authors and the outputs obtained were the ones expected.

In the experiment regarding the evaluation of the iHTN generation, we aimed at providing evidence that we can transform our deliberation results into executable and valid models for robotic mission execution.The possible threats to validity in this evaluation were: (i) the fact that correctness of the iHTNs relied on the system designer and (ii) the lack of evidence that this iHTN can be in fact executed. To mitigate those threats, a simulation for the Lab Samples Logistics example was performed using MORSE simulator [32] and the expected behavior was correctly identified[4]. We reckon that more comprehensive experiments should be performed for the sake of generality of our conclusions.

Regarding the performance evaluation of the automated mission process in MutRoSe, we aimed at sheding some light over the computational feasibility of the process. It could be noticed that the performance of the process is quite sensitive to the design decisions, mostly regarding: (i) the number of forall statements, and (ii) the variability at the HDDL level and variability at goal model level.We noticed that the iterations in the forall statements together with the OR-decompositions in the goal model were the major reasons for a performance bottleneck in the mission decomposition process. Our educated guess suggest that, if variability at the goal model level cannot be avoided, one possible workaround for missions that require such feature is processing requests in batches. By doing this, one could avoid decomposing missions for a high number of requests at once. Some of the previous modelled examples, each one exploring different features, were used in order to perform the evaluation using real-world scenarios. One possible threat to validity in this evaluation is the expressiveness of the examples.Even though they are real-world scenarios, they may not explore all of the aspects (or their possible combinations) in a sufficient way to verify their effects in the decomposition.

Last, but not least, we should mention that an evaluation with users to assess the

[4]The simulation log can be seen in the Github repository https://github.com/ericbg27/RoboMax-iHTN-examples as the file Lab Sample Logistics/simulation_log.log

learning curve required from MutRoSe was attempted. Although we reached out dozens of candidates, feedbacks were scarce to report in this dissertation. For this reason, we plan to conduct a more thorough investigation on that respect as a future work.

Chapter 6

Related Work

In this section, we aim to give an overview of the related approaches that aim to model multi-robot (or multi-agent) missions. For now, only one approach that aims to model multi-robot team behavior in a high-level fashion was found and we aim to compare our approach to it. One important note is that we did not find any other approach that aims to model multi-robot team behavior in the same level of our approach, which takes into consideration the possible task dependencies, as specified in the iTax taxonomy, and constraints between them, while also taking into account the knowledge about the world, which introduces variability in the decomposition process.

6.1 ALICA

ALICA stands for A Language for Interactive Cooperative Agents. This language is proposed in [11] and it targets modeling of dynamic domains, where agents cannot always communicate beforehand and need to take decisions rapidly. This approach aims at specifying behavior based on a high-level goal where the system designer must model the whole team behavior (i.e., the mission). Also, this approach takes capabilities into consideration and implements an abstraction between plans and agents using roles and tasks. In addition, this language introduces the concept of utilities for task allocations, which is a major concern when trying to allocate tasks in an optimal way.

Besides being a language, the ALICA approach introduces a runtime (or execution) engine that runs in each agent of the system, which is shown in Figure 6.1. In this runtime engine ALICA plans are kept in the Plan Repository, which are interpreted by the ALICA Execution Engine at runtime. Also, conditions are evaluated and updates are performed based on the defined transitions and in the World Model, which consists on the agent's beliefs that can be obtained either by the agent itself or by the knowledge of other agents that is kept in the Shared World Model. In a more detailed view of the ALICA language,

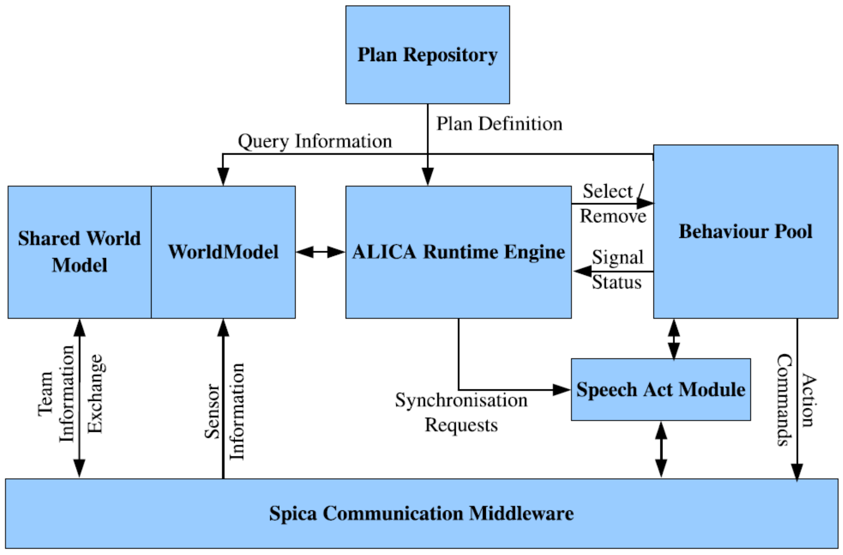


Figure 6.1: The ALICA execution engine (extracted from [11])

there are five main requirements it aims to achieve. It has to give to the developer a global perspective of the team behavior, since he/she should be able to model the team behavior as a whole and not single agent ones. Also, the language aims to have formal grounding such that the multi-agent behavior that results from the interpretation of the models can be proven. The language also needs to have abstraction in a way that it should abstract away from specific agents, or specific teams. Furthermore, the language needs to be extensible in order to accommodate for future developments. Finally, it must have reusability, where modelled strategies should be reusable and not tied to specific situations. These requirements show that this approach indeed follow a MDE perspective for multi-agent (or multi-robot to be more specific) behavior specification, in constrast to low-level approaches commonly used.

With what was previously explained, we can verify the main common points and the main differences between our approach and the ALICA one. The main common points are the global perspective of the team behavior (or mission), where the modeling focus on a common goal shared between agents, the formal grounding, which we inherit from HDDL that uses first-order logic to express preconditions and effects, the abstraction, in the sense of abstraction away from specific characteristics of robots, the definitions of plans in a hierarchical way and the search for all possible compositions of a mission that can be executed given the knowledge the system has about the world. In constrast to these common points, there are some main differences and gaps that this work aims to fill. The first difference is the possibility to have variable parameters in the model specification in order to account for multiple task instances, which makes sense given that this approach focus on highly dynamic environments like the robot soccer one but would not fit so well when it comes to an environment where we would need to execute tasks in different locations depending on the current context. Also, the utility calculation performed by the ALICA approach seems to fit for the environment that it focus on but does not seem to fit so well when the task dependencies and constraints defined by the iTax taxonomy need to be taken into consideration. It should be noted that this work does not deal with utility calculation, but our decomposition process takes into consideration these possible dependencies and constraints and generates task instances in a way that an allocation process can perform the utility calculation process in the right way. Finally, the ALICA approach seems to deal with tasks in the same level as the HDDL language while in our work we use the goal model to model the team behavior in an even higher level, treating goals as first class entities and specifying multiple tasks and the possible constraints between them.

6.2 PROMISE

PROMISE [12] is a DSL for high-level mission specification for MRS. This mission specification basically consists in the definition of global missions to be achieved by the MRS and its decomposition into local missions, which are robot-specific. Furthermore, this DSL is built upon the PsALM (Pattern bAsed Mission specifier) tool [33] catalog of LTL-based patterns, which are used as atomic tasks in PROMISE's framework. Finally, PROMISE introduces the concepts of (1) composition operators, which allow the definition of complex missions by combining other operators and managing events that may occur in the environment, and (2) delegate operators, which make a robot perform a specific task instantiated with parameters as locations or actions.

PROMISE provides a graphical and a textual framework, where an example of a mission specified in the graphical framework is shown in 6.2. The basic idea of this mission is that a robot, in this case called r1, is responsible for sequentially patrolling locations l1, l2 and l3. If r1 encounters an object in its way it requests for the help of another robot called r2. Robot r2 starts waiting in location l4 and, when help is requested, moves to l2, where it tries to load the object in its platform and communicates if an error occurs. In this specification we have the global mission decomposed by node (1) into the two local missions previously explained, where one is executed by robot r1 and the other by robot r2. Operators (2) and (4) are called event handler nodes where there is a default behavior, given by the path with no event, and paths which are activated depending on events that may happen. It is worth noticing that events are given as simple names and are represented by the gray ellipses. Other important operators which are present in the figure are: (i) the sequential operator represented by node (7), which sequentially execute

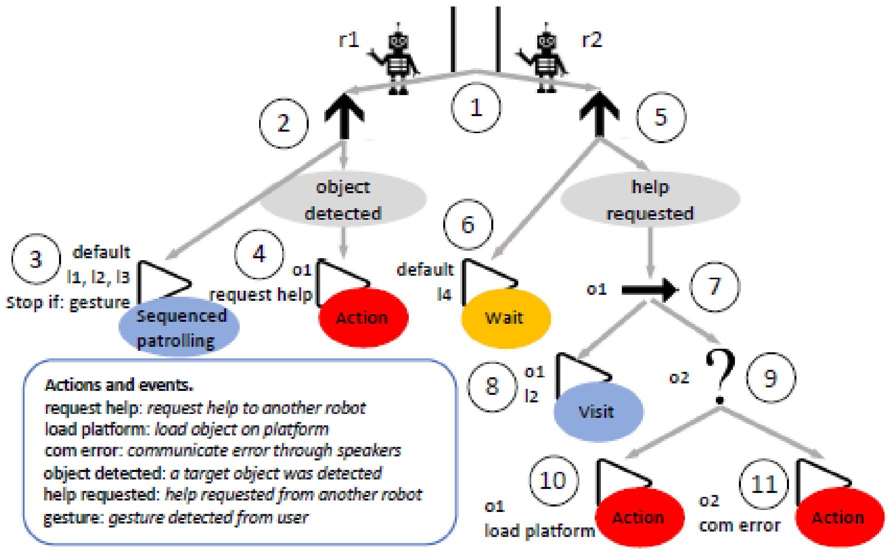

r1
r2
1
2
5
object detected
help requested
3
default
l1, l2, l3
Stop if: gesture
Sequenced patrolling
4
o1
request help
Action
6
default
l4
Wait
o1
7
8
o1
l2
Visit
o2
9
10
o1
load platform
Action
o2
11
com error
Action
Actions and events.
request help: request help to another robot
load platform: load object on platform
com error: communicate error through speakers
object detected: a target object was detected
help requested: help requested from another robot
gesture: gesture detected from user

Figure 6.2: An example of a mission specification in PROMISE (extracted from [12])

its children, (ii) the fallback operator, which is represented by node (9) and starts by executing its leftmost child and, in case it fails, go on to the next child until one succeeds, and (iii) the delegate operator, which is represented by nodes (3), (4), (6), (8), (10) and (11), and simply instantiates a task with given parameters.

It can be seen that the PROMISE DSL framework and this work both propose a way to perform specification of missions for MRS in a global perspective. The main difference between the two approaches is the level in which this specification is performed. In PROMISE, parameters and robots that execute certain tasks and actions must be known in advance, since they are given as input directly into the specification. In the proposed work, the goal model and HDDL domain specification leave everything as variables to be solved when the system has enough knowledge about the world and when an allocation process decides which robot (or team of robots) can perform the tasks in the best way, given some utility definition. This fact is what makes necessary for our approach to perform mission decomposition, since the result of this process can be different depending on the current state of the world and the system, whereas in PROMISE the decomposition is previously established and the specification model is used at execution time to control the system's execution, which makes sense since PROMISE is based on behavior trees.

Based on the previous comparison between this work and PROMISE we can see that these works can be used together in an approach for autonomous execution and control of MRS, if certain additional features, like events and multiple parameter tasks, are added to the proposed work. One can use what is proposed in this work in order to have a model that is parameterized. With this model and the system's knowledge, the MRS can perform task decomposition and allocation in order to have all of the task instances to be executed and the robots that will execute them. With everything defined, the system can perform a transformation of this output into a PROMISE specification and use this specification to control the MRS execution, taking advantage of the behavior tree structure of this DSL.

6.3 HiDDeN

HiDDeN [5] is a distributed deliberative architecture that manages the execution of a hierarchical plan. The core model of used in this approach to represent plans is the instantiated HTN (called iHTN), which is a structure built upon the HTN formalism. The iHTN definition is very similar to that of an HTN, where we have that an iHTN is a subset of an HTN where the planner has previously decided the methods and the variables to used. In this sense, we can say that an iHTN is an HTN where the non-determinism is already solved and the grounding process already happened. The HiDDeN approach itself makes use of a deterministic planner, where each abstract task has only one method.

Since the HiDDeN approach focuses on safety-critical environments where communication is restricted, as is the case on the military context, and on plan execution, control and failure recovery, it makes sense that the used structure is in fact deterministic. In fact, one of the constraints given is that the whole plan must be computed offline in order to be evaluated by human operators. In other environments these restrictions do not apply and replanning as an online mechanism is interesting, in order to increase the system overall utility or even to deal with unexpected events. The approach proposed in this work aims to give a higher level model of MRS missions and take into consideration several kinds of dependencies to be solved given the knowledge about the world. Furthermore, this work also proposes a task decomposition process that solves all of the dependencies that can be solved using the world knowledge and generates the task instances to be executed, leaving everything related to robots to be solved by a task allocation approach that aims to maximize the overall system utility.

After the previously mentioned allocation process, which is out of the scope of this work, where all of the non-determinism would be resolved, one could aim at generating an iHTN using its output and possibly the HDDL domain model and use it to perform control and execution of the mission. In this sense, this work can build upon the HiDDeN approach and a task allocation step in order to have a fully autonomous MRS that performs decomposition, allocation, scheduling, control and execution of missions based on a high-level mission specification made at design time and the system's knowledge.

6.4 Task Definition Language

The Task Definition Language (TDL) [13] is a DSL used to design and implement heterogeneous MRS. This language is divided in two parts: (i) the Robot Model, which is platform-specific and is used for modeling the robots and the simple actions that they perform, and (ii) the Task Model, which defines tasks and composite tasks without providing any information about what type of robots are going to perform the task, i.e., the task model is robot independent. The metamodel which defines both of these parts is shown in Figure 6.3. This is a language created following the a model-driven engineering (MDE) approach, which is a highly positive aspect of it. In its Task Model it allows the user to perform an easy composition of tasks, which are themselves compositions of actions, defined in the Robot Model. This Task Model is a high-level language since it abstracts away from specific robot characteristics. Also, the actions in this language have arguments, in order to have parameterized tasks that do not have fixed values by design. Finally, certain restrictions can be established using the sync and after keywords, which establish synchronization and precedence constraints, respectively.

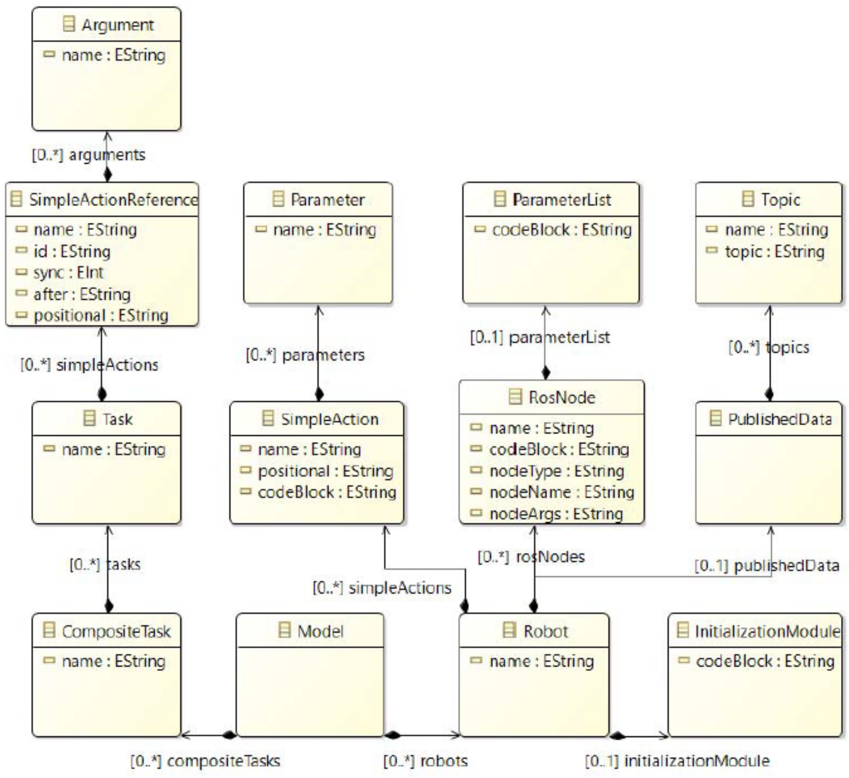


Figure 6.3: The TDL metamodel (extracted from [13])

An interesting characteristic of this language is its simplicity, but this is also a drawback. There is no way of formally defining actions preconditions and effects to be used at a planning phase and it seems to be no way of combining several composite tasks and defining restrictions between them. This is due to the fact that this is more focused in an execution level phase, where task instances are not necessarily created in an automated fashion but can even be given by an user, which is the case of the web interface that is proposed in the original work. In this work, we develop a mission specification that captures details of the world knowledge and also allows for the definition of restrictions between composite tasks, defining allowed ranges of robots that can execute them and building upon the capability concept in order to create constraints on execution. Unlike the TDL approach, this work is not focused on execution level but at planning and deliberation level, which requires the logical formalism in order to plan ahead of execution.

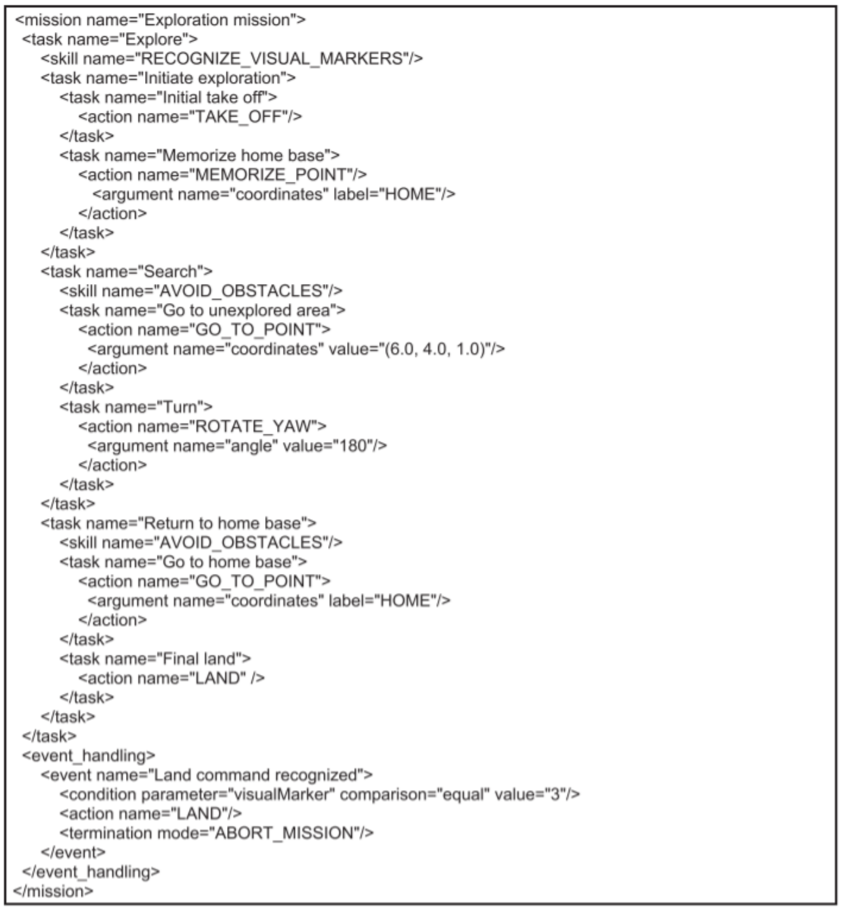
6.5 Mission Description Language

The Mission Description Language (MDL) [16] is an XML-based language used to describe the specifics of a mission to be performed by a team of autonomous vehicles. This language aims at filling the gap on formalizing mission specification for MRS missions using high-level concepts. MDL is based on descriptions present in other files, which are mainly (i) the Scenario Description Language (SDL), which describes the scenario the agents will operate in, (ii) the Team Description Language (TDL), which describe agent teams and their restrictions and (iii) the Disturbance Description Language (DDL), which describes the disturbances that exist in the mission environment.

This language is domain specific in the sense that it aims at specifying missions for vehicle robots. Also, it is a language that is more focused on execution, having constructs like formation, search pattern and division method, and not planning and deliberation, even though there is the concept of strategies, which define tactics and actions and conditions for their execution, and allows the definition of precedence constraints between mission phases. One more thing that can be seen is that in MDL we have the definition of vehicle categories, having 4 built-in ones and allowing for custom ones, and required sensors instead of using the capability concept, which is a better fit when defining a general-purpose language and accounting for evolution of robots capabilities. The previously described aspects of MDL is what differs the languages from the modeling framework proposed in this work, which is focused on being a general-purpose language focused on a parameterized specification of MRS missions to be used for deliberation and planning in dynamic environments.

6.6 Task-based Mission Specification Language

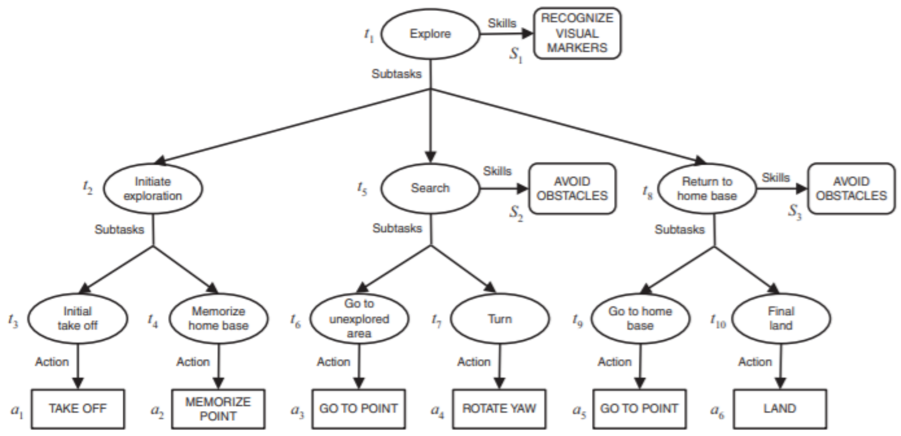
The Task-based Mission specification Language (TML) [14] is a specification language designed for the Aerostack framework [34], which is an open-source framework for aerial robotics. This language aims at providing a flexible representation for aerial robotics missions in the framework and also to use this representation for reactive planning in order to increase the system's adaptability. This language aims at providing a higher-level representation than code-based representations previously used and in order to do that TML uses XML syntax in order to be readable to humans and machines. An example of a TML mission specification is shown in Figure 6.4, where we can see the main structures of this language: (i) the mission, which is composed of tasks, (ii) the tasks, where each task requires a set of skills and can be composed of other tasks or of actions, (iii) the actions, where each action contains its arguments and their respective values, and (iv) the event handling section, which is composed of events with their respective conditions and the actions to be executed in the presence of the event.



```
<mission name="Exploration mission">
 <task name="Explore">
   <skill name="RECOGNIZE_VISUAL_MARKERS"/>
   <task name="Initiate exploration">
     <task name="Initial take off">
       <action name="TAKE_OFF"/>
     </task>
     <task name="Memorize home base">
       <action name="MEMORIZE_POINT"/>
         <argument name="coordinates" label="HOME"/>
       </action>
     </task>
   </task>
   <task name="Search">
     <skill name="AVOID_OBSTACLES"/>
     <task name="Go to unexplored area">
       <action name="GO_TO_POINT">
         <argument name="coordinates" value="(6.0, 4.0, 1.0)"/>
       </action>
     </task>
     <task name="Turn">
       <action name="ROTATE_YAW">
         <argument name="angle" value="180"/>
       </action>
     </task>
   </task>
   <task name="Return to home base">
     <skill name="AVOID_OBSTACLES"/>
     <task name="Go to home base">
       <action name="GO_TO_POINT">
         <argument name="coordinates" label="HOME"/>
       </action>
     </task>
     <task name="Final land">
       <action name="LAND" />
     </task>
   </task>
 </task>
 <event_handling>
   <event name="Land command recognized">
     <condition parameter="visualMarker" comparison="equal" value="3"/>
     <action name="LAND"/>
     <termination mode="ABORT_MISSION"/>
   </event>
 </event_handling>
</mission>
```

Figure 6.4: An example TML specification (extracted from [14])

From a TML specification a structure called the task tree is created, where the task tree that represents the specification in Figure 6.4 is shown in Figure 6.5. This structure is used for plan validation and execution, where two basic execution strategies are used by the TML intepreter. The first strategy is a task-based strategy, where the execution follows the sequence of tasks established in the task tree translating them into actions and the skills that need to be active. The second strategy is an event-driven strategy, where the presence of events is verified during action execution and reacts accordingly.



Notes: Ellipses represent tasks, rectangles represent actions and rounded rectangles represent skills

Figure 6.5: Example Task Tree obtained from a TML specification (extracted from [14])

The TML language is focused on reactive planning, rather than classical planning, since the authors claim that this type of approach is more simplified and thus is useful to cope with highly dynamic and unpredictable environments. With this in mind, it can be said that this language is completely focused on execution instead of planning and deliberation. Because of this reactive nature, constraints between tasks in this work are very simple, with the linear execution sequence being modified by the presence of events and conditions. In addition, all of the task instances to be executed must be defined in a TML mission specification, which could be avoided by adding constructs like iteration and the addition of variables, since it seems that only constants are used as arguments in the language. One more thing to note is that TML is a language focused on a very specific domain, even though the impression is that it requires a small effort for it to be transformed into a general-purpose language. The previously described aspects are the main differences between TML and the languages used in this work.

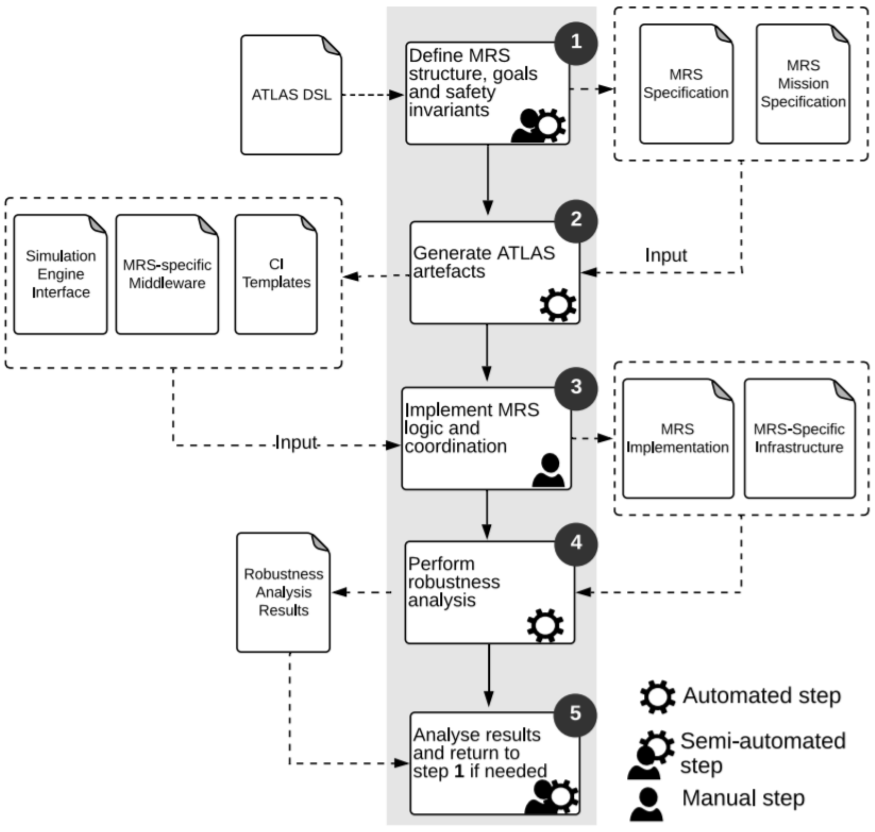

ATLAS DSL
Define MRS structure, goals and safety invariants
1
MRS Specification
MRS Mission Specification
Simulation Engine Interface
MRS-specific Middleware
CI Templates
Generate ATLAS artefacts
2
Input
Implement MRS logic and coordination
3
Input
MRS Implementation
MRS-Specific Infrastructure
Robustness Analysis Results
Perform robustness analysis
4
Analyse results and return to step 1 if needed
5
Automated step
Semi-automated step
Manual step

Figure 6.6: The ATLAS methodology (extracted from [15])

6.7 ATLAS

ATLAS [15] is a model-driven, tool-supported framework for the systematic engineering of MRS. The ATLAS methodology is shown in Figure 6.6, where we can see five main steps:

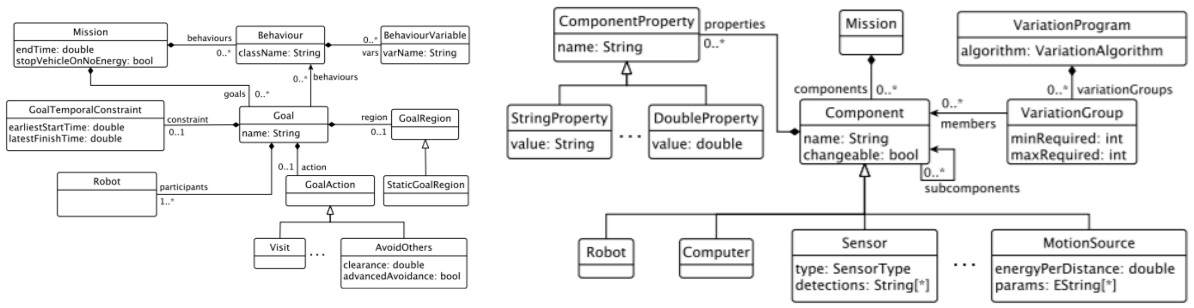
- Defining MRS structure, goals and safety invariants: This is where the mission is defined alongside necesary requirements. The main artifact in this stage is the mission definition, which is specified in the ATLAS DSL
- Generate ATLAS artifacts: This is where we have a model-drive code generation engine which generates three important artifacts. These artifacts consist in a middleware that enables the communication of the various system components, Collective Intelligence (CI) templates which will be used by the middleware to coordinate the robotic team, interface code that enables the middleware to communicate directly with the target robotic simulator and the necessary configuration files for the target robotic simulator

- Implement MRS logic and coordination: This is a step where engineers implement the logic and coordination of the MRS populating the CI templates previously generating with code

- Perform robustness analysis: This is where MRS simulation analysis is automatically executed, where logs are generated comprising messages exchanged and events that occurred

- Analyse results: This is where engineers analyze results in order to select the most effective MRS design and the most suitable CI algorithm

With respect to the ATLAS DSL, it is an XML-based language that it enables: (i) the specification of the MRS mission, including functional and non-functional requirements, and (ii) the characteristics of individual robots that compose the system, including capabilities, architecture and internal behaviors. There are two higher concepts in the ATLAS DSL which are the *Mission*, which includes the goals that need to be achieved and multiple properties related to them, and the *Component*, which represents the top level element of the hierarchical representation of the robotic system and can be either a *Robot* or a *Computer*. The mission and component metamodels are shown in Figures 6.7a and 6.7b, respectively.



(a)

(b)

Figure 6.7: ATLAS DSL mission and components metamodels (extracted from [15])

The ATLAS approach is focused in automatic simulation for effective engineering of MRS systems. This framework offers a code generation engine which generates artifacts that can be used afterwards by engineers once they decide the system configuration that satisfies their requirements. With this in mind, the ATLAS DSL specifies the mission in the form of goals and necessary actions to be performed, thus not performing any automated decomposition, and also models for the robots, since the system needs to be simulated in order to perform the evaluation process. In this sense, this work is complementary to the current work, where the output of given by the MutRoSe framework could be transformed into an ATLAS mission and then, given robot definitions, the ATLAS process could happen if desired. With this in mind, several features encountered in MutRoSe models are not present in ATLAS, being one example the parameters in models which are used to perform mission decomposition, given the fact that ATLAS focus is different than MutRoSe's and thus their idea of an MRS mission is quite different.

6.8 Related Works Comparison

In this section we perform a general comparison between this work and the related works previously mentioned. Table 6.1 summarizes this comparison, where we verify the features considered as the most necessary for a MRS mission specification framework. These features and the reasons they are necessary are as follows:

- Global Perspective: A model that gives a global overview of the mission is interesting since it allows users to understand the specification more clearly and easily. Instead of focusing only on the specification of tasks, end-users specifying the mission for a MRS will focus on specifying the mission to be achieved by the system as a whole, leaving local perspectives of tasks to be dealt with at lower levels of abstraction. In this sense, the model used for mission specification will use tasks that are available in some kind of library of tasks, as pointed in [7], instead of dealing directly with their specification. Another reason why this is interesting comes from the fact that end-users will often be users without robotic, ICT or mathematical expertise [7], but with expertise in the domain the robotic system will be executed. In this sense, a model that allows this domain expert to specify the mission as a whole, leaving lower-level details to be specified by a robotics expert, is desired.

- General-purpose models: As stated in [7], the robotics domain is divided into a large variety of sub-domains, including vertical ones (e.g., drivers, planning, navigation) and horizontal ones (e.g., defense, healthcare, logistics), with a vast amount of variability. In this sense, a language that can specify a mission using high-level abstractions is an interesting approach since it is suitable for a variety of different domains, leaving domain-specific concerns to lower-level models and/or specifications.

- Deliberative models: As shown in the MRS workflow presented in [2], there are several steps for the design of a cooperative MRS. A model that is focused on deliberation instead of execution can be used for reasoning about the real-world and the best way to allocate tasks to the available robots. This kind of model can be used as input to a mission decomposition process, which generates the set of sub-tasks to be executed that are allocated to robots based on a task allocation approach that tries to maximize some utility function. Models that are focused on execution should be generated at the end of this process, given pre-defined specifications of low-level behavior, in order for the MRS to in fact execute the previously modelled behavior. Therefore, deliberation focused models should be considered in order to allow the MRS to reason at runtime about the best way of executing the mission at hand, which is not possible in execution focused models that focus on dealing with the execution of tasks and possible failures the system may encounter with the robots already assigned to the necessary tasks.

- Capabilities as first-class entities: When having models that are used for deliberation in cooperative heterogeneous MRS, the notion of capabilities (often called skills) needs to be present. In this sense, one can define which capabilities are required to perform certain actions and the reasoning process can easily verify if a robot is capable or not of executing it by performing what is called capability matchmaking [24]. This is important since we may have various different types of robots that are able to perform different sets of behaviors in different fashion. Also, robots may also be able to acquire or lose capabilities throughout the system's lifetime and by having the notion of these capabilities we can perform efficient reasoning on the allocation of tasks.

- Parameterized models: A parameterized model is paramount when dealing with MRS in order to cope with the real-world variability. This variability is a concern that exists in the community [7] and needs to be dealt with. With the usage of parameters, the system can reason on the model at execution time (and in an autonomous fashion) in order to assess which tasks need to be performed and even how these tasks can be executed, i.e., which actions can be performed in order to accomplish a certain task given certain conditions that are verified using the knowledge the system has about the environment and itself.

- High-level constraints between tasks: Since we aim at having models that specify the mission as a whole, with all of the tasks that need to be (or may be) executed, we need to establish constraints between them in some way. Behavior trees, for example, allow for the specification of ordering constraints, establishing sequential, parallel and fallback operators that strictly define the order in which tasks need to be executed. Alongside these ordering constraints there are other types of constraints. In this work we define execution constraints, which specify if tasks need to be executed by the same robot (or team of robots) or if they can be executed by different ones. All of these constraints may impact the allocation process and often reduce the search space for task allocation algorithms.

- Fleet specification of a mission: This is related to the fact that the end-user does not need to allocate tasks to robots. As specified in [7], we have that: "the mission specification will represent the "needs" of the end-user and robots will be automatically assigned, and potentially re-assigned during the mission execution, according to the capabilities or robots and various quality parameters". We can go even further and establish that the end-user may even not be aware of the available robots and their exact number, since the mission specification model can be conceived before the MRS itself.

	Work							
Feature	ALICA [11]	PROMISE [12]	HiDDeN [5]	TDL [13]	MDL [16]	TML [14]	ATLAS [15]	MutRoSe
Global perspective	✓	✓	✓	✗	✓	✓	✓	✓
General-purpose models	✓	✓	✓	✓	✗	✗	✓	✓
Deliberative models	✓	✗	✓✗	✗	✗	✗	✗	✓
Capabilities	✗	✗	✗	✓✗	✓	✓	✓✗	✓
Parameterized models	✓✗	✓✗	✓✗	✓	✗	✗	✗	✓
High-level Constraints	✗	✓	✗	✗	✓✗	✓✗	✗	✓
Fleet Specification	✓	✓✗	✓✗	✓	✓✗	✓	✗	✓

Table 6.1: Related works comparison

Chapter 7

Conclusions and Future Work

MRS design is a problem with several challenges to be overcome. In the field of MRS/-MAS, many works were proposed for planning [2, 29], task allocation and execution [2]. Mission specification (or behavior specification), in particular high-level mission specification, on the other hand, did not receive the necessary attention, even though there are works that deal with this problem [1]. This kind of system has several constraints, task dependencies and particularities that make it difficult to find a design-time approach that can be used at runtime while taking these aspects into consideration. In this context, this work proposes the MutRoSe framework for mission specification and task decomposition that tackles this problem in order to enable users to specify robotic missions in a high-level fashion and for the MRS to reason over design time models in an automated fashion. The models in this framework are able to tackle important research topics that have been identified in the community [7], such as: (i) dealing with real-world variability, (ii) reusability and (iii) not having to specify the robots themselves in the specification. Furthermore, when combining the proposed task decomposition process with task allocation and task execution approaches, the MRS will be able to perform the steps of the MRS workflow [2] autonomously.

There are still some open gaps we envision addressing in MutRoSe. First, the proposed decomposition process can only deal with fully observable environments, which is a problem in environments where knowledge about the world can be outdated or even lacking information (e.g., open environments like the ones from military missions). Moreover, only one HDDL file is accepted for a given mission, whereas a more modular approach, with multiple HDDL files that contain similar tasks, can be the way to go to improve reusability even further. Additionally, each mission must be represented by a single goal model and there is no way of establishing a relation between missions, by for example defining priorities between them, which may be desirable in an environment where multiple missions must be executed. Finally, the current approach for decomposition is exploratory, which may introduce limitations or require workarounds to be used at runtime in some cases. In this sense, possible future works are:

- A probabilistic extension of the framework in order to tackle non-deterministic environments. This extension can also help with decomposition in partially observable environments
    - In addition, ways to define policies for the decomposition of the mission to work in partially observable can also be created
- Definition of ways to link multiple HDDL Domain files to a specific goal model, in order to improve reusability
- Definition of ways for establishing relations between missions in a same domain. This information may be important at runtime in order to define what needs to executed at the moment
- Creation of heuristics for the decomposition process, which will be able to reduce time-space complexity to perform decomposition of missions
- A model checking approach to verify the correctness of the mission and its decompositions. This would help to tackle challenges identified in [35]

References

[1] Paraschos, Alexandros, Nikolaos I Spanoudakis, and Michail G Lagoudakis: *Model-driven behavior specification for robotic teams.* In *AAMAS*, pages 171–178, 2012. vi, 1, 8, 118

[2] Rizk, Yara, Mariette Awad, and Edward W Tunstel: *Cooperative heterogeneous multi-robot systems: a survey.* ACM Computing Surveys (CSUR), 52(2):1–31, 2019. vi, vii, xiii, 2, 7, 115, 118

[3] Höller, Daniel, Gregor Behnke, Pascal Bercher, Susanne Biundo, Humbert Fiorino, Damien Pellier, and Ron Alford: *Hddl: An extension to pddl for expressing hierarchical planning problems.* In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9883–9891, 2020. viii, 5, 11, 23

[4] Ghallab, Malik, Dana Nau, and Paolo Traverso: *Automated Planning: theory and practice.* Elsevier, 2004. ix, 4, 12

[5] Lesire, Charles, Guillaume Infantes, Thibault Gateau, and Magali Barbier: *A distributed architecture for supervision of autonomous multi-robot missions.* Autonomous Robots, 40(7):1343–1362, 2016. ix, 2, 3, 95, 108, 117

[6] Askarpour, Mehrnoosh, Christos Tsigkanos, Claudio Menghi, Radu Calinescu, Patrizio Pelliccione, Sergio García, Ricardo Caldas, Tim J von Oertzen, Manuel Wimmer, Luca Berardinelli, *et al.*: *Robomax: Robotic mission adaptation exemplars.* In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 245–251. IEEE, 2021. ix, 5, 79, 101

[7] Dragule, Swaib, Sergio García Gonzalo, Thorsten Berger, and Patrizio Pelliccione: *Languages for specifying missions of robotic applications.* In *Software Engineering for Robotics*, pages 377–411. Springer, 2021. x, 3, 115, 116, 117, 118

[8] Korsah, G Ayorkor, Anthony Stentz, and M Bernardine Dias: *A comprehensive taxonomy for multi-robot task allocation.* The International Journal of Robotics Research, 32(12):1495–1512, 2013. xiii, 3, 8, 9, 10

[9] Braubach, Lars, Alexander Pokahr, Daniel Moldt, and Winfried Lamersdorf: *Goal representation for bdi agent systems.* In *International workshop on programming multi-agent systems*, pages 44–65. Springer, 2004. xiii, 14, 18

[10] Cabot, Jordi and Martin Gogolla: *Object constraint language (ocl): a definitive guide.* In *International school on formal methods for the design of computer, communication and software systems*, pages 58–90. Springer, 2012. xiii, 15

[11] Skubch, Hendrik, Michael Wagner, Roland Reichle, and Kurt Geihs: *A modelling language for cooperative plans in highly dynamic domains.* Mechatronics, 21(2):423–433, 2011. xiv, 1, 2, 3, 22, 104, 105, 117

[12] García, Sergio, Patrizio Pelliccione, Claudio Menghi, Thorsten Berger, and Tomas Bures: *Promise: high-level mission specification for multiple robots.* In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pages 5–8, 2020. xiv, 1, 2, 3, 106, 107, 117

[13] Losvik, Daneil Steen and Adrian Rutle: *A domain-specific language for the development of heterogeneous multi-robot systems.* In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 549–558. IEEE, 2019. xiv, 2, 109, 117

[14] Molina, Martin, Ramon A Suarez-Fernandez, Carlos Sampedro, Jose Luis Sanchez-Lopez, and Pascual Campoy: *Tml: a language to specify aerial robotic missions for the framework aerostack.* International Journal of Intelligent Computing and Cybernetics, 2017. xiv, 2, 111, 112, 117

[15] Harbin, James Robert, Simos Gerasimou, Nikolaos Matragkas, Athanasios Zolotas, and Radu Calinescu: *Model-driven simulation-based analysis for multi-robot systems.* In *MODELS 2021: ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems (MODELS).* York, 2021. xiv, 113, 114, 117

[16] Silva, Daniel Castro, Pedro Henriques Abreu, Luís Paulo Reis, and Eugénio Oliveira: *Development of a flexible language for mission description for multi-robot missions.* Information Sciences, 288:27–44, 2014. 2, 110, 117

[17] Mendonça, Danilo Filgueira, Genaína Nunes Rodrigues, Raian Ali, Vander Alves, and Luciano Baresi: *Goda: A goal-oriented requirements engineering framework for runtime dependability analysis.* Information and Software Technology, 80:245–264, 2016. 4, 13

[18] Yu, E: *Modelling strategic relationships for process reengineering. 1995.* University of Toronto, 1995. 5

[19] Khamis, Alaa, Ahmed Hussein, and Ahmed Elmogy: *Multi-robot task allocation: A review of the state-of-the-art.* Cooperative Robots and Sensor Networks 2015, pages 31–51, 2015. 7

[20] Gerasimou, Simos, Nicholas Matragkas, and Radu Calinescu: *Towards systematic engineering of collaborative heterogeneous robotic systems.* In *2019 IEEE/ACM 2nd International Workshop on Robotics Software Engineering (RoSE)*, pages 25–28. IEEE, 2019. 7

[21] Ulam, Patrick, Yoichiro Endo, Alan Wagner, and Ronald Arkin: *Integrated mission specification and task allocation for robot teams-design and implementation.* In *Proceedings 2007 IEEE International Conference on Robotics and Automation*, pages 4428–4435. IEEE, 2007. 8

[22] Zlot, Robert Michael: *An auction-based approach to complex task allocation for multirobot teams.* PhD thesis, Citeseer, 2006. 8

[23] Gerkey, Brian P and Maja J Matarić: *A formal analysis and taxonomy of task allocation in multi-robot systems.* The International journal of robotics research, 23(9):939–954, 2004. 9

[24] Kunze, Lars, Tobias Roehm, and Michael Beetz: *Towards semantic robot description languages.* In *2011 IEEE International Conference on Robotics and Automation*, pages 5589–5595. IEEE, 2011. 11, 116

[25] Olivares-Alarcos, Alberto, Daniel Beßler, Alaa Khamis, Paulo Goncalves, Maki K Habib, Julita Bermejo-Alonso, Marcos Barreto, Mohammed Diab, Jan Rosell, Joao Quintas, *et al.*: *A review and comparison of ontology-based approaches to robot autonomy.* The Knowledge Engineering Review, 34, 2019. 11

[26] Dastani, Mehdi, M Birna Van Riemsdijk, and John Jules Ch Meyer: *Goal types in agent programming.* In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1285–1287, 2006. 14

[27] OCL, OMG: *Object constraint language (ocl), version 2.4*, 2014. 15, 18

[28] Van Lamsweerde, Axel: *From system goals to software architecture.* In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pages 25–43. Springer, 2003. 18, 20

[29] Torreno, Alejandro, Eva Onaindia, Antonín Komenda, and Michal Štolba: *Cooperative multi-agent planning: a survey.* ACM Computing Surveys (CSUR), 50(6):1–32, 2017. 22, 118

[30] Bercher, Pascal, Gregor Behnke, Daniel Höller, and Susanne Biundo: *An admissible htn planning heuristic.* In *IJCAI*, pages 480–488, 2017. 25, 41

[31] Elkawkagy, Mohamed, Pascal Bercher, Bernd Schattenberg, and Susanne Biundo: *Improving hierarchical planning performance by the use of landmarks.* In *AAAI*, 2012. 25

[32] Echeverria, Gilberto, Nicolas Lassabe, Arnaud Degroote, and Séverin Lemaignan: *Modular open robots simulation engine: Morse.* In *2011 IEEE International Conference on Robotics and Automation*, pages 46–51. IEEE, 2011. 102

[33] Menghi, Claudio, Christos Tsigkanos, Patrizio Pelliccione, Carlo Ghezzi, and Thorsten Berger: *Specification patterns for robotic missions.* IEEE Transactions on Software Engineering, 2019. 106

[34] Sanchez-Lopez, Jose Luis, Ramón A Suárez Fernández, Hriday Bavle, Carlos Sampedro, Martin Molina, Jesus Pestana, and Pascual Campoy: *Aerostack: An architecture and open-source software framework for aerial robotics.* In *2016 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 332–341. IEEE, 2016. 111

[35] Luckcuck, Matt, Marie Farrell, Louise A Dennis, Clare Dixon, and Michael Fisher: *Formal specification and verification of autonomous robotic systems: A survey.* ACM Computing Surveys (CSUR), 52(5):1–41, 2019. 119

Appendix A

RoboMAX Missions Models

For every example in Section 5.1 were given the Goal Model, the world knowledge and an illustration of the output for each modelled RoboMAX mission. In this Appendix we give the HDDL Domain Definition and the configuration file for each one of them.

A.1 Vital Signs Monitoring

The HDDL Domain Definition for the Vital Signs Monitoring example is shown in Listing A.1 and the configuration file is shown in Listing A.2.

```
(define (domain hospital)
        (:types patient room - object)
        (:predicates
                (occupied ?rm - room)
                (infectiousdisease ?p - patient)
                (diabetes ?p - patient)
                (postsurgical ?p - patient)
                (available ?p - patient)
                (at ?p - patient ?rm - room)
                (checked ?p - patient)
        )
        (:capabilities check-heart-rate
            check-blood-pressure check-blood-oxigenation
            check-body-temperature check-glucose)

        (:task EnterRoom :parameters (?r - robot ?rm - room))
        (:method room-enter
                :parameters (?r - robot ?rm - room)
                :task (EnterRoom ?r ?rm)
                :precondition ()
                :ordered-subtasks (and
                        (enter-room ?r ?rm)
                )
        )
        (:action enter-room
                :parameters (?r - robot ?rm - room)
        )

        (:task ApproachPatient
            :parameters (?r - robot ?rm - room ?p - patient))
        (:method available-patient-approach
                :parameters (?r - robot ?rm - room ?p - patient)
                :task (ApproachPatient ?r ?rm ?p)
                :precondition (and
                        (available ?p)
                )
                :ordered-subtasks (and
                        (approach-patient ?r ?rm ?p)
                )
        )
        (:method unavailable-patient-approach
                :parameters (?r - robot ?rm - room ?p - patient)
                :task (ApproachPatient ?r ?rm ?p)
                :precondition (and
                        (not (available ?p))
                )
                :ordered-subtasks (and
                        (comeback-later ?r ?rm ?p)
                        (approach-patient ?r ?rm ?p)
                )
        )

        (:action approach-patient
                :parameters (?r - robot ?rm - room ?p - patient)
        )
        (:action comeback-later
                :parameters (?r - robot ?rm - room ?p - patient)
        )
```

```
(:task ProvideInstructions
    :parameters (?r - robot ?rm - room ?p - patient))
(:method instructions-providing
        :parameters (?r - robot ?rm - room ?p - patient)
        :task (ProvideInstructions ?r ?rm ?p)
        :ordered-subtasks (and
                (provide-instructions ?r ?rm ?p)
        )
)
(:action provide-instructions
        :parameters (?r - robot ?rm - room ?p - patient)
)

(:task CollectVitalSigns
    :parameters (?r - robot ?rm - room ?p - patient))
(:method infections-disease-checking
        :parameters (?r - robot ?rm - room ?p - patient)
        :task (CollectVitalSigns ?r ?rm ?p)
        :precondition (and
                (infectiousdisease ?p)
        )
        :ordered-subtasks(and
                (infectious-disease-collect ?r ?rm ?p)
                (mark-as-done ?r ?rm ?p)
        )
)
(:method diabetes-checking
        :parameters (?r - robot ?rm - room ?p - patient)
        :task (CollectVitalSigns ?r ?rm ?p)
        :precondition (and
                (diabetes ?p)
        )
        :ordered-subtasks(and
                (diabetes-collect ?r ?rm ?p)
                (mark-as-done ?r ?rm ?p)
        )
)
(:method post-surgical-checking
        :parameters (?r - robot ?rm - room ?p - patient)
        :task (CollectVitalSigns ?r ?rm ?p)
```

```
                :precondition (and
                        (postsurgical ?p)
                )
                :ordered-subtasks(and
                        (post-surgical-collect ?r ?rm ?p)
                        (mark-as-done ?r ?rm ?p)
                )
        )

        (:action infectious-disease-collect
                :parameters (?r - robot ?rm - room ?p - patient)
                :required-capabilities (check-heart-rate
                    check-blood-pressure check-blood-oxigenation
                    check-body-temperature)
        )
        (:action diabetes-collect
                :parameters (?r - robot ?rm - room ?p - patient)
                :required-capabilities (check-glucose)
        )
        (:action post-surgical-collect
                :parameters (?r - robot ?rm - room ?p - patient)
                :required-capabilities (check-heart-rate
                    check-blood-oxigenation)
        )
        (:action mark-as-done
                :parameters (?r - robot ?rm - room ?p - patient)
                :effect (and
                        (checked ?p)
                )
        )

        (:task AssessPatientStatus
            :parameters (?r - robot ?rm - room ?p - patient))
        (:method patient-assessing
                :parameters (?r - robot ?rm - room ?p - patient)
                :task (AssessPatientStatus ?r ?rm ?p)
                :ordered-subtasks (and
                        (assess-patient ?r ?rm ?p)
                )
        )
```

```
(:action assess-patient
        :parameters (?r - robot ?rm - room ?p - patient)
        :effect (and
                (checked ?p)
        )
)

(:task SendAlert
    :parameters (?r - robot ?rm - room ?p - patient))
(:method alert-sending
        :parameters (?r - robot ?rm - room ?p - patient)
        :task (SendAlert ?r ?rm ?p)
        :ordered-subtasks (and
                (send-alert ?r ?rm ?p)
        )
)
(:action send-alert
        :parameters (?r - robot ?rm - room ?p - patient)
)

(:task RobotSanitization :parameters (?r - robot ?srm - room))
(:method robot-sanitization
        :parameters (?r - robot ?srm - room)
        :task (RobotSanitization ?r ?srm)
        :ordered-subtasks (and
                (sanitize-robot ?r ?srm)
        )
)
(:action sanitize-robot
        :parameters (?r - robot ?srm - room)
)

(:task RechargeBattery :parameters (?r - robot))
(:method recharge-battery
        :parameters (?r - robot)
        :task (RechargeBattery ?r)
        :ordered-subtasks (and
                (robot-recharge ?r)
        )
)
```

```
        (:action robot-recharge
                :parameters (?r - robot)
        )
)
```

Listing A.1: HDDL Domain definition for the Vital Signs Monitoring mission

```
{
    "world_db": {
        "type": "file",
        "file_type": "xml",
        "path": "knowledge/world_db.xml",
        "xml_root": "world_db"
    },

    "output": {
        "output_type": "file",
        "file_path": "output/task_output.json",
        "file_type": "json"
    },

    "location_types": ["Room"],

    "type_mapping": [
        {
            "hddl_type": "room",
            "ocl_type": "Room"
        },
        {
            "hddl_type": "patient",
            "ocl_type": "Patient"
        }
    ],

    "var_mapping": [
        {
            "task_id": "AT1",
            "map": [
                {
                    "gm_var": "current_room",
                    "hddl_var": "?rm"
                }
            ]
        },
        {
            "task_id": "AT2",
            "map": [
                {
                    "gm_var": "current_room",
                    "hddl_var": "?rm"
                },
                {
                    "gm_var": "current_patient",
                    "hddl_var": "?p"
                }
            ]
        },
        {
            "task_id": "AT3",
            "map": [
                {
                    "gm_var": "current_room",
                    "hddl_var": "?rm"
                },
                {
                    "gm_var": "current_patient",
                    "hddl_var": "?p"
                }
            ]
        },
        {
            "task_id": "AT4",
            "map": [
                {
                    "gm_var": "current_room",
                    "hddl_var": "?rm"
                },
                {
                    "gm_var": "current_patient",
                    "hddl_var": "?p"
                }
            ]
        },
        {
            "task_id": "AT5",
            "map": [
                {
                    "gm_var": "current_room",
                    "hddl_var": "?rm"
                },
                {
                    "gm_var": "current_patient",
                    "hddl_var": "?p"
                }
            ]
        },
        {
            "task_id": "AT6",
            "map": [
                {
                    "gm_var": "current_room",
                    "hddl_var": "?rm"
                },
                {
                    "gm_var": "current_patient",
                    "hddl_var": "?p"
                }
            ]
        },
        {
            "task_id": "AT7",
            "map": [
                {
                    "gm_var": "sanitization_room",
                    "hddl_var": "?srm"
                }
            ]
        }
    ],

    "semantic_mapping": [
        {
            "type": "attribute",
            "name": "is_occupied",
            "relates_to": "Room",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "occupied",
                "arg_sorts": ["room"]
            }
        },
        {
            "type": "attribute",
            "name": "available",
            "relates_to": "Patient",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "available",
                "arg_sorts": ["patient"]
            }
        },
        {
            "type": "attribute",
            "name": "infectious_disease",
            "relates_to": "Patient",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "infectiousdisease",
                "arg_sorts": ["patient"]
            }
        },
        {
            "type": "attribute",
            "name": "diabetes",
            "relates_to": "Patient",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
```

```
            "pred": "diabetes",
            "arg_sorts": ["patient"]
        }
    },
    {
        "type": "attribute",
        "name": "post_surgical",
        "relates_to": "Patient",
        "belongs_to": "world_db",
        "mapped_type": "predicate",
        "map": {
            "pred": "postsurgical",
            "arg_sorts": ["patient"]
        }
    },
    {
        "type": "attribute",
        "name": "checked",
        "relates_to": "Patient",
        "belongs_to": "world_db",
        "mapped_type": "predicate",
        "map": {
            "pred": "checked",
            "arg_sorts": ["patient"]
        }
    },
    {
        "type": "relationship",
        "main_entity": "Room",
        "related_entity": "Patient",
        "relationship_type": "attribute",
        "attribute_name": "patients",
        "belongs_to": "world_db",
        "mapped_type": "predicate",
        "map": {
            "pred": "at",
            "arg_sorts": ["patient", "room"]
        }
    }
]
```

```
}
```

Listing A.2: JSON configuration file for the Vital Signs Monitoring mission

A.2 Lab Samples Logistics

The HDDL Domain Definition for the Lab Samples Logistics example is shown in Listing A.3 and the configuration file is shown in Listing A.4.

```
(define (domain hospital)
        (:types patient location delivery arm - object)
        (:predicates
                (deliverynurse ?n - nurse ?d - delivery)
                (deliverypickuplocation ?l - location ?d - delivery)
                (locationarm ?a - arm ?l - location)
        )

        (:task ApproachArm
            :parameters (?r - robot ?a - arm ?p - location))
        (:method arm-approach
                :parameters (?r - robot ?a - arm ?p - location)
                :task (ApproachArm ?r ?a ?p)
                :ordered-subtasks (and
                        (navto ?r ?p)
                        (approach-arm ?r)
                )
        )

        (:task ApproachNurse
            :parameters (?r - robot ?n - nurse ?l - location))
        (:method nurse-approach
                :parameters (?r - robot ?n - nurse ?l - location)
                :task (ApproachNurse ?r ?n ?l)
                :ordered-subtasks (and
                        (navto ?r ?l)
                        (approach-nurse ?r)
                        (authenticate-nurse ?r)
                )
        )
```

```
(:task PickSample :parameters (?r - robot ?n - nurse))
(:method sample-pick
        :parameters (?r - robot ?n - nurse)
        :task (PickSample ?r ?n)
        :ordered-subtasks (and
                (open-drawer ?r)
                (deposit ?n)
                (close-drawer ?r)
        )
)

(:task UnloadSample :parameters (?r - robot ?a - arm))
(:method sample-unload
        :parameters (?r - robot ?a - arm)
        :task (UnloadSample ?r ?a)
        :precondition ()
        :ordered-subtasks (and
                (open-drawer ?r)
                (pick-up-sample ?a)
                (close-drawer ?r)
        )
)


(:action navto
        :parameters (?r - robot ?loc - location)
)
(:action approach-nurse
        :parameters (?r - robot)
)
(:action authenticate-nurse
        :parameters (?r - robot)
)
(:action open-drawer
        :parameters (?r - robot)
)
(:action close-drawer
        :parameters (?r - robot)
)
(:action deposit
                :parameters (?n - nurse)
        )
        (:action approach-arm
                :parameters (?r - robot)
        )
        (:action pick-up-sample
                :parameters (?a - arm)
        )
)
```

Listing A.3: HDDL Domain definition for the Lab Samples Logistics mission

```
{
    "world_db": {
        "type": "file",
        "file_type": "xml",
        "path": "knowledge/world_db.xml",
        "xml_root": "world_db"
    },

    "output": {
        "output_type": "file",
        "file_path": "output/task_output.json",
        "file_type": "json"
    },

    "location_types": ["Location"],

    "type_mapping": [
        {
            "hddl_type": "delivery",
            "ocl_type": "Delivery"
        },
        {
            "hddl_type": "nurse",
            "ocl_type": "Nurse"
        },
        {
            "hddl_type": "location",
            "ocl_type": "Location"
        },
        {
            "hddl_type": "arm",
            "ocl_type": "Arm"
        }
    ],

    "var_mapping": [
        {
            "task_id": "AT1",
            "map": [
                {
                    "gm_var": "pickup_location",
                    "hddl_var": "?l"
                },
                {
                    "gm_var": "current_nurse",
                    "hddl_var": "?n"
                }
            ]
        },
        {
            "task_id": "AT2",
            "map": [
                {
                    "gm_var": "current_nurse",
                    "hddl_var": "?n"
                }
            ]
        },
        {
            "task_id": "AT3",
            "map": [
                {
                    "gm_var": "pharmacy_loc",
                    "hddl_var": "?p"
                },
                {
                    "gm_var": "pharmacy_arm",
                    "hddl_var": "?a"
                }
            ]
        },
        {
            "task_id": "AT4",
            "map": [
                {
                    "gm_var": "pharmacy_arm",
                    "hddl_var": "?a"
                }
            ]
        }
    ],



    "semantic_mapping": [
        {
            "type": "relationship",
            "main_entity": "Delivery",
            "related_entity": "Nurse",
            "relationship_type": "attribute",
            "attribute_name": "patient",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "deliverynurse",
                "arg_sorts": ["nurse", "delivery"]
            }
        },
        {
            "type": "relationship",
            "main_entity": "Delivery",
            "related_entity": "Location",
            "relationship_type": "attribute",
            "attribute_name": "pickup_location",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "deliverypickuplocation",
                "arg_sorts": ["location", "delivery"]
            }
        },
        {
            "type": "relationship",
            "main_entity": "Location",
            "related_entity": "Arm",
            "relationship_type": "attribute",
            "attribute_name": "arm",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "locationarm",
                "arg_sorts": ["arm", "location"]
            }
        }
    ]
}
```

Listing A.4: JSON configuration file for the Lab Samples Logistics mission

A.3 Deliver Goods/Equipment

The HDDL Domain Definition for the Deliver Goods/Equipment example is shown in Listing A.5 and the configuration file is shown in Listing A.6.

```
(define (domain hospital)
        (:types location storage obj agent - object)
        (:predicates
                (requestingequipment ?a - agent)
                (at ?o - obj ?s - storage)
                (at ?l - location ?a - agent)
                (requested ?o - obj ?a - agent)
        )
        (:capabilities )

        (:task GetObject
            :parameters (?r - robot ?s - storage ?o - obj))
        (:method object-get
                :parameters (?r - robot ?s - storage ?o - obj)
                :task (GetObject ?r ?s ?o)
            :precondition ()
            :ordered-subtasks (and
                    (get-object ?r ?s ?o)
            )
    )
    (:action get-object
            :parameters (?r - robot ?s - storage ?o - obj)
    )

    (:task RechargeBattery :parameters (?r - robot))
    (:method battery-recharge
            :parameters (?r - robot)
            :task (RechargeBattery ?r)
            :precondition ()
            :ordered-subtasks (and
                    (recharge-battery ?r)
            )
    )
    (:action recharge-battery
            :parameters (?r - robot)
    )

    (:task DeliverObjects :parameters (?r - robot ?l - location))
    (:method objects-delivery
            :parameters (?r - robot ?l - location)
            :task (DeliverObjects ?r ?l)
            :precondition ()
            :ordered-subtasks (and
                    (deliver-objects ?r ?l)
            )
    )
    (:action deliver-objects
            :parameters (?r - robot ?l - location)
    )

    (:task ReturnObjectsToCheckpoint :parameters (?r - robot))
    (:method object-returning
            :parameters (?r - robot)
            :task (ReturnObjectsToCheckpoint ?r)
            :precondition ()
```

```
                :ordered-subtasks (and
                        (return-objects ?r)
                )
        )
        (:action return-objects
                :parameters (?r - robot)
        )

        (:task AlertTrigger :parameters (?r - robot))
        (:method alert-trigger
                :parameters (?r - robot)
                :task (AlertTrigger ?r)
                :precondition ()
                :ordered-subtasks (and
                        (trigger-alert ?r)
                )
        )
        (:action trigger-alert
                :parameters (?r - robot)
        )
)
```

Listing A.5: HDDL Domain definition for the Deliver Goods/Equipment mission

```
{
    "world_db": {
        "type": "file",
        "file_type": "xml",
        "path": "knowledge/world_db.xml",
        "xml_root": "world_db"
    },

    "output": {
        "output_type": "file",
        "file_path": "output/task_output.json",
        "file_type": "json"
    },

    "location_types": ["Location", "Storage"],

    "type_mapping": [
        {
            "hddl_type": "agent",
            "ocl_type": "Agent"
        },
        {
            "hddl_type": "location",
            "ocl_type": "Location"
        },
        {
            "hddl_type": "storage",
            "ocl_type": "Storage"
        },
        {
            "hddl_type": "obj",
            "ocl_type": "Object"
        }
    ],

    "var_mapping": [
        {
            "task_id": "AT1",
            "map": [
                {
                    "gm_var": "object_storage",
                    "hddl_var": "?s"
                },
                {
                    "gm_var": "current_object",
                    "hddl_var": "?o"
                }
            ]
        },
        {
            "task_id": "AT3",
            "map": [
                {
                    "gm_var": "delivery_location",
                    "hddl_var": "?l"
                }
            ]
        }
    ],

    "semantic_mapping": [
        {
            "type": "attribute",
            "name": "requesting_equipment",
            "relates_to": "Agent",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "requestingequipment",
                "arg_sorts": ["agent"]
            }
        },
        {
            "type": "ownership",
            "owner": "Storage",
            "owned": "Object",
            "relationship_type": "attribute",
            "attribute_name": "objects",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "at",
                "arg_sorts": ["obj", "storage"]
            }
        },
        {
            "type": "ownership",
            "owner": "Agent",
            "owned": "Object",
            "relationship_type": "attribute",
            "attribute_name": "requested_objects",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "requested",
                "arg_sorts": ["obj", "agent"]
            }
        },
        {
            "type": "relationship",
            "main_entity": "Agent",
            "related_entity": "Location",
            "relationship_type": "attribute",
            "attribute_name": "location",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "at",
                "arg_sorts": ["location", "agent"]
            }
        }
    ]
}
```

Listing A.6: JSON configuration file for the Deliver Goods/Equipment mission

A.4 Food Logistics

The HDDL Domain Definition for the Food Logistics example is shown in Listing A.7. The configuration files are shown in the next sections.

```
(define (domain hospital)
        (:types
                delivery pickup patient location - object
        )
        (:predicates
                (patientcanfetch ?p - patient)
                (patientcanopen ?p - patient)
                (deliverypatient ?p - patient ?d - delivery)
                (deliverylocation ?l - location ?d - delivery)
                (pickuppatient ?p - patient ?pk - pickup)
                (pickuplocation ?l - location ?pk - pickup)
                (pickeddishes ?r - robot)
                (pickedmeal ?r - robot)
                (at ?r - robot ?l - location)
        )
        (:capabilities manipulation door-opening)
```

```
(:task GetFood
    :parameters (?r - robot ?l - location ?d - delivery))
(:method food-pickup
        :parameters (?r - robot ?l - location ?d -delivery)
        :task (GetFood ?r ?l ?d)
        :ordered-subtasks (and
                (navto ?r ?l)
                (wait-for-food ?r ?l ?d)
        )
)
(:action wait-for-food
        :parameters (?r - robot ?l - location ?d - delivery)
        :effect (and
                (pickedmeal ?r)
        )
)

(:task DeliverToTable
    :parameters (?r - robot ?l - location ?p - patient))
(:method table-deliver
        :parameters (?r - robot ?l - location ?p - patient)
        :task (DeliverToTable ?r ?l ?p)
        :precondition (and
                (pickedmeal ?r)
        )
        :ordered-subtasks (and
                (navto ?r ?l)
                (approach-patient-table ?r ?l ?p)
                (deliver-to-table ?r ?l)
        )
)
(:action approach-patient-table
        :parameters (?r - robot ?l - location ?p - patient)
)
(:action deliver-to-table
        :parameters (?r - robot ?l - location)
        :required-capabilities (manipulation)
)
```

```
(:task DeliverToFetch
    :parameters (?r1 ?r2 - robot ?l - location ?p - patient))
(:method fetch-deliver
        :parameters (?r1 ?r2 - robot ?l - location
            ?p - patient)
        :task (DeliverToFetch ?r1 ?r2 ?l ?p)
        :precondition (and
                (pickedmeal ?r1)
        )
        :ordered-subtasks (and
            (navto ?r1 ?l)
                (FetchMeal ?r1 ?r2 ?l ?p)
        )
)

(:task FetchMeal
    :parameters (?r1 ?r2 - robot ?l - location ?p - patient))
(:method fetch-meal-with-human
        :parameters (?r1 ?r2 - robot ?l - location
            ?p - patient)
        :task (FetchMeal ?r1 ?r2 ?l ?p)
        :precondition (and
                (patientcanfetch ?p)
        )
        :ordered-subtasks (and
                (approach-human ?r1 ?l ?p)
                (wait-for-human-to-fetch ?r1 ?l ?p)
        )
)
(:method fetch-meal-with-robot
        :parameters (?r1 ?r2 - robot ?l - location
            ?p - patient)
        :task (FetchMeal ?r1 ?r2 ?l ?p)
        :ordered-subtasks (and
            (navto ?r2 ?l)
                (approach-robot ?r1 ?r2)
                (grasp-meal ?r2 ?r1)
                (deliver-meal-to-patient ?r2 ?p ?l)
        )
)
```

```
(:task PickupDishes
    :parameters (?r1 ?r2 - robot ?l - location ?p - patient))
(:method pickup-with-door-opening
        :parameters (?r1 ?r2 - robot ?l - location
            ?p - patient)
        :task (PickupDishes ?r1 ?r2 ?l ?p)
        :precondition (and
                (not (pickeddishes ?r1))
        )
        :ordered-subtasks (and
                (navto ?r1 ?l)
                (navto ?r2 ?l)
                (approach-door ?r1 ?l)
                (approach-door ?r2 ?l)
                (open-door ?r1 ?r2 ?l)
                (PickDishesTwoRobotsAtLocation ?r1 ?r2 ?l ?p)
        )
)
(:method pickup-without-door-opening
        :parameters (?r1 ?r2 - robot ?l - location
            ?p - patient)
        :task (PickupDishes ?r1 ?r2 ?l ?p)
        :precondition (and
                (patientcanopen ?p)
                (not (pickeddishes ?r1))
        )
        :ordered-subtasks (and
                (navto ?r1 ?l)
                (approach-door ?r1)
                (wait-for-door-opening ?r1)
                (PickDishesOneRobotAtLocation ?r1 ?r2 ?l ?p)
        )
)

(:task PickDishesTwoRobotsAtLocation
    :parameters (?r1 ?r2 - robot ?l - location ?p - patient))
(:method pick-dishes-two-robots-at-location
        :parameters (?r1 ?r2 - robot ?l - location
            ?p - patient)
        :task (PickDishesTwoRobotsAtLocation ?r1 ?r2 ?l ?p)
        :precondition (and
                (at ?r1 ?l)
                (at ?r2 ?l)
        )
        :ordered-subtasks (and
                (PickDishes ?r1 ?r2 ?l ?p)
        )
)

(:task PickDishesOneRobotAtLocation
    :parameters (?r1 ?r2 - robot ?l - location ?p - patient))
(:method pick-dishes-one-robot-at-location
        :parameters (?r1 ?r2 - robot ?l - location
            ?p - patient)
        :task (PickDishesOneRobotAtLocation ?r1 ?r2 ?l ?p)
        :precondition (and
                (at ?r1 ?l)
                (not (at ?r2 ?l))
        )
        :ordered-subtasks (and
                (PickDishes ?r1 ?r2 ?l ?p)
        )
)

(:task PickDishes
    :parameters (?r1 ?r2 - robot ?l - location ?p - patient))
(:method pick-dishes-with-human
        :parameters (?r1 ?r2 - robot ?l - location
            ?p - patient)
        :task (PickDishes ?r1 ?r2 ?l ?p)
        :ordered-subtasks (and
                (approach-human ?r1 ?l ?p)
                (wait-for-human-to-place-dish ?r1 ?p)
        )
)
(:method pick-dishes-with-robot-at-location
        :parameters (?r1 ?r2 - robot ?l - location
            ?p - patient)
        :task (PickDishes ?r1 ?r2 ?l ?p)
        :precondition (and
                (at ?r2 ?l)
        )
        :ordered-subtasks (and
                (pick-patient-dishes ?r2 ?p)
                (load-dishes ?r2 ?r1)
        )
)
(:method pick-dishes-with-robot-not-at-location
        :parameters (?r1 ?r2 - robot ?l - location
            ?p - patient)
        :task (PickDishes ?r1 ?r2 ?l ?p)
        :precondition (and
                (not (at ?r2 ?l))
        )
        :ordered-subtasks (and
                (navto ?r2 ?l)
                (pick-patient-dishes ?r2 ?p)
                (load-dishes ?r2 ?r1)
        )
)

(:task RetrieveDishes :parameters (?r - robot ?l - location))
(:method dishes-retrieval
        :parameters (?r - robot ?l - location)
        :task (RetrieveDishes ?r ?l)
        :ordered-subtasks (and
                (navto ?r ?l)
                (retrieve-dishes ?r ?l)
        )
)

(:action approach-human
        :parameters (?r - robot ?l - location ?p - patient)
)
(:action approach-robot
        :parameters (?r1 ?r2 - robot)
)
(:action grasp-meal
        :parameters (?r1 ?r2 - robot)
        :effect (and
                (not (pickedmeal ?r2))
                (pickedmeal ?r1)
        )
)
(:action deliver-meal-to-patient
        :parameters (?r - robot ?p - patient ?l - location)
        :effect (and
                (not (pickedmeal ?r))
        )
)
(:action wait-for-human-to-fetch
        :parameters (?r - robot ?l - location ?p - patient)
        :effect (and
                (not (pickedmeal ?r))
        )
)

(:action wait-for-human-to-place-dish
        :parameters (?r - robot ?p - patient)
        :effect (and
                (pickeddishes ?r)
        )
)
(:action pick-patient-dishes
        :parameters (?r - robot ?p - patient)
        :effect (and
                (pickeddishes ?r)
        )
)
(:action load-dishes
        :parameters (?r1 ?r2 - robot)
        :effect (and
                (not (pickeddishes ?r1))
                (pickeddishes ?r2)
        )
)
(:action approach-door
        :parameters (?r1 - robot ?l - location)
)
```

```
        (:action open-door
                :parameters (?r1 ?r2 - robot ?l - location)
                :required-capabilities (door-opening)
        )
        (:action wait-for-door-opening
                :parameters (?r - robot)
        )
        (:action pickup-dishes-with-robot
                :parameters (?r1 ?r2 - robot ?l - location)
                :effect (and
                        (pickeddishes ?r1)
                )
        )
        (:action retrieve-dishes
                :parameters (?r - robot ?l - location)
        )

        (:action navto
                :parameters (?r - robot ?l - location)
        )
)
```

Listing A.7: HDDL Domain definition for the Food Logistics mission

A.4.1 Delivery

The configuration file for the Food Logistics Delivery mission is shown in Listing A.8.

```
{
    "world_db": {
        "type": "file",
        "file_type": "xml",
        "path": "knowledge/world_db.xml",
        "xml_root": "world_db"
    },

    "output": {
        "output_type": "file",
        "file_path": "output/task_outputDelivery.json",
        "file_type": "json"
    },
"location_types": ["Location"],

"type_mapping": [
    {
        "hddl_type": "location",
        "ocl_type": "Location"
    },
    {
        "hddl_type": "patient",
        "ocl_type": "Patient"
    },
    {
        "hddl_type": "delivery",
        "ocl_type": "Delivery"
    }
],

"var_mapping": [
    {
        "task_id": "AT1",
        "map": [
            {
                "gm_var": "kitchen_loc",
                "hddl_var": "?l"
            },
            {
                "gm_var": "current_delivery",
                "hddl_var": "?d"
            }
        ]
    },
    {
        "task_id": "AT2",
        "map": [
            {
                "gm_var": "delivery_room",
                "hddl_var": "?l"
            },
            {
                "gm_var": "current_patient",
                "hddl_var": "?p"
            }
        ]
    },
    {
        "task_id": "AT3",
        "map": [
            {
                "gm_var": "delivery_room",
                "hddl_var": "?l"
            },
            {
                "gm_var": "current_patient",
                "hddl_var": "?p"
            }
        ]
    }
],

"semantic_mapping": [
    {
        "type": "attribute",
        "name": "can_fetch",
        "relates_to": "Patient",
        "belongs_to": "world_db",
        "mapped_type": "predicate",
        "map": {
            "pred": "patientcanfetch",
            "arg_sorts": ["patient"]
        }
    },
    {
        "type": "attribute",
        "name": "picked_meal",
        "relates_to": "robot",
        "belongs_to": "robots_db",
        "mapped_type": "predicate",
        "map": {
            "pred": "pickedmeal",
                "arg_sorts": ["robot"]
            }
        },
        {
            "type": "relationship",
            "main_entity": "Delivery",
            "related_entity": "Patient",
            "relationship_type": "attribute",
            "attribute_name": "patient",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "deliverypatient",
                "arg_sorts": ["patient", "delivery"]
            }
        },
        {
            "type": "relationship",
            "main_entity": "Delivery",
            "related_entity": "Location",
            "relationship_type": "attribute",
            "attribute_name": "delivery_location",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "deliverylocation",
                "arg_sorts": ["location", "delivery"]
            }
        }
    ]
}
```

Listing A.8: JSON configuration file for the Food Logistics Delivery mission

A.4.2 Pickup

The configuration file for the Food Logistics Delivery mission is shown in Listing A.9.

```
{
    "world_db": {
        "type": "file",
        "file_type": "xml",
        "path": "knowledge/world_db.xml",
        "xml_root": "world_db"
    },

    "output": {
        "output_type": "file",
        "file_path": "output/task_outputPickup.json",
        "file_type": "json"
    },

    "location_types": ["Location"],

    "type_mapping": [
        {
            "hddl_type": "location",
            "ocl_type": "Location"
        },
        {
            "hddl_type": "patient",
            "ocl_type": "Patient"
        },
        {
            "hddl_type": "pickup",
            "ocl_type": "Pickup"
        }
    ],

    "var_mapping": [
        {
            "task_id": "AT1",
            "map": [
                {
                    "gm_var": "pickup_room",
                    "hddl_var": "?l"
                },
                {
                    "gm_var": "current_pickup_patient",
                    "hddl_var": "?p"
                }
            ]
        },
        {
            "task_id": "AT2",
            "map": [
                {
                    "gm_var": "kitchen_loc",
                    "hddl_var": "?l"
                }
            ]
        }
    ],

    "semantic_mapping": [
        {
            "type": "attribute",
            "name": "can_open",
            "relates_to": "Patient",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "patientcanopen",
                "arg_sorts": ["patient"]
            }
        },
        {
            "type": "attribute",
            "name": "picked_dishes",
            "relates_to": "robot",
            "belongs_to": "robots_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "pickeddishes",
                "arg_sorts": ["robot"]
            }
        },
        {
            "type": "relationship",
            "main_entity": "Pickup",
            "related_entity": "Patient",
            "relationship_type": "attribute",
            "attribute_name": "patient",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "pickuppatient",
                "arg_sorts": ["patient", "pickup"]
            }
        },
        {
            "type": "relationship",
            "main_entity": "Pickup",
            "related_entity": "Location",
            "relationship_type": "attribute",
            "attribute_name": "pickup_location",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "pickuplocation",
                "arg_sorts": ["location", "pickup"]
            }
        }
    ]
}
```

Listing A.9: JSON configuration file for the Food Logistics Pickup mission

Appendix B

Generation of iHTN Files

For every example in Section 5.2 we may have the modified world knowledge file and we have JSON files which consist in the iHTNs that represent the totally ordered valid mission decompositions for the example. In this appendix we give the necessary files to aid the explanation of the iHTN generation process for each of the used examples.

B.1 Lab Samples Logistics

The JSON output file for the only iHTN generated in the Lab Sample Logistics example is shown in Listing B.1. The world knowledge for this example was not modified and thus is not shown here.

```
{
    "0": {
        "name": "ROOT",
        "type": "task",
        "parent": "-1",
        "agents": [
            "r1",
            "Arm1",
            "Nurse1"
        ],
        "children": [
            "1"
        ]
    },
    "1": {
        "name": "ROOT_M",
        "type": "method",
        "parent": "0",
        "agents": [
            "r1",
            "Arm1",
            "Nurse1"
        ],
        "children": [
            "2",
            "7",
            "12",
            "16"
        ]
    },
    "2": {
        "name": "ApproachNurse",
        "type": "task",
        "parent": "1",
        "agents": [
            "r1",
            "Nurse1"
        ],
        "children": [
            "3"
        ]
    },
    "3": {
        "name": "nurse-approach",
        "type": "method",
        "parent": "2",
        "agents": [
            "r1",
            "Nurse1"
        ],
        "children": [
            "4",
            "5",
            "6"
        ]
    },
    "4": {
        "name": "navto",
        "type": "action",
        "parent": "3",
        "locations": [
            "Room3"
        ],
        "agents": [
            "r1"
        ],
        "children": ""
    },
    "5": {
        "name": "approach-nurse",
        "type": "action",
        "parent": "3",
        "locations": "",
        "agents": [
            "r1"
        ],
        "children": ""
    },
    "6": {
        "name": "authenticate-nurse",
        "type": "action",
        "parent": "3",
        "locations": "",
        "agents": [
            "r1"
        ],
        "children": ""
    },
    "7": {
        "name": "PickSample",
        "type": "task",
        "parent": "1",
        "agents": [
            "r1",
            "Nurse1"
        ],
        "children": [
                "8"
            ]
        },
        "8": {
            "name": "sample-pick",
            "type": "method",
            "parent": "7",
            "agents": [
                "r1",
                "Nurse1"
            ],
            "children": [
                "9",
                "10",
                "11"
            ]
        },
        "9": {
            "name": "open-drawer",
            "type": "action",
            "parent": "8",
            "locations": "",
            "agents": [
                "r1"
            ],
            "children": ""
        },
        "10": {
            "name": "deposit",
            "type": "action",
            "parent": "8",
            "locations": "",
            "agents": [
                "Nurse1"
            ],
            "children": ""
        },
        "11": {
            "name": "close-drawer",
            "type": "action",
            "parent": "8",
            "locations": "",
            "agents": [
                "r1"
            ],
            "children": ""
        },
        "12": {
            "name": "ApproachArm",
            "type": "task",
            "parent": "1",
            "agents": [
                "r1",
                "Arm1"
            ],
            "children": [
                "13"
            ]
        },
        "13": {
            "name": "arm-approach",
            "type": "method",
            "parent": "12",
            "agents": [
                "r1",
                "Arm1"
            ],
            "children": [
                "14",
                "15"
            ]
        },
        "14": {
            "name": "navto",
            "type": "action",
            "parent": "13",
            "locations": [
                "Pharmacy"
            ],
            "agents": [
                "r1"
            ],
            "children": ""
        },
        "15": {
            "name": "approach-arm",
            "type": "action",
            "parent": "13",
            "locations": "",
            "agents": [
                "r1"
            ],
            "children": ""
        },
        "16": {
            "name": "UnloadSample",
            "type": "task",
            "parent": "1",
            "agents": [
                "r1",
                "Arm1"
            ],
            "children": [
                "17"
            ]
        },
        "17": {
            "name": "sample-unload",
            "type": "method",
            "parent": "16",
            "agents": [
                "r1",
                "Arm1"
            ],
            "children": [
                "18",
                "19",
                "20"
            ]
        },
        "18": {
            "name": "open-drawer",
            "type": "action",
            "parent": "17",
            "locations": "",
            "agents": [
                "r1"
            ],
            "children": ""
        },
        "19": {
            "name": "pick-up-sample",
            "type": "action",
            "parent": "17",
            "locations": "",
            "agents": [
                "Arm1"
            ],
            "children": ""
        },
        "20": {
            "name": "close-drawer",
            "type": "action",
            "parent": "17",
            "locations": "",
            "agents": [
                "r1"
            ],
            "children": ""
        }
}
```

Listing B.1: JSON output for the only iHTN of the Lab Samples Logistics

B.2 Food Logistics

The world knowledge for both the Food Logistics Delivery and Pickup mission is the same and is shown in Listing B.2.

```
<world_db>
```

```
<Delivery>
        <name>Delivery1</name>
        <patient>Patient1</patient>
        <delivery_location>RoomA</delivery_location>
</Delivery>

<Pickup>
        <name>Pickup1</name>
        <patient>Patient1</patient>
        <pickup_location>RoomA</pickup_location>
</Pickup>

<Location>
        <name>RoomA</name>
        <patient>Patient1</patient>
</Location>
<Location>
        <name>RoomB</name>
        <patient>Patient2</patient>
</Location>
<Location>
        <name>RoomC</name>
        <patient>Patient3</patient>
</Location>
<Location>
        <name>Kitchen</name>
</Location>

<Patient>
        <name>Patient1</name>
        <can_fetch>True</can_fetch>
        <can_open>False</can_open>
</Patient>
<Patient>
        <name>Patient2</name>
        <can_fetch>False</can_fetch>
        <can_open>False</can_open>
</Patient>
<Patient>
        <name>Patient3</name>
```

```
                <can_fetch>False</can_fetch>
                <can_open>True</can_open>
        </Patient>
</world_db>
```

Listing B.2: Reduced world knowledge for the Food Logistics example

B.2.1 Food Logistics Delivery

The JSON iHTN files for the Food Logistics Delivery are shown in Listings B.3, B.4 and B.5.

```
{
    "0": {
        "name": "ROOT",
        "type": "task",
        "parent": "-1",
        "agents": [
            "r1",
            "Delivery1",
            "Patient1"
        ],
        "children": [
            "1"
        ]
    },
    "1": {
        "name": "ROOT_M",
        "type": "method",
        "parent": "0",
        "agents": [
            "r1",
            "Delivery1",
            "Patient1"
        ],
        "children": [
            "2",
            "6"
        ]
    },
    "2": {
        "name": "GetFood",
        "type": "task",
        "parent": "1",
        "agents": [
            "r1",
            "Delivery1"
        ],
        "children": [
            "3"
        ]
    },
    "3": {
        "name": "food-pickup",
        "type": "method",
        "parent": "2",
        "agents": [
            "r1",
            "Delivery1"
        ],
        "children": [
            "4",
            "5"
        ]
    },
    "4": {
        "name": "navto",
        "type": "action",
        "parent": "3",
        "locations": [
            "Kitchen"
        ],
        "agents": [
            "r1"
        ],
        "children": ""
    },
    "5": {
        "name": "wait-for-food",
        "type": "action",
        "parent": "3",
        "locations": [
            "Kitchen"
        ],
        "agents": [
            "r1",
            "Delivery1"
        ],
        "children": ""
    },
    "6": {
        "name": "DeliverToTable",
        "type": "task",
        "parent": "1",
        "agents": [
            "r1",
            "Patient1"
        ],
        "children": [
            "7"
        ]
    },
    "7": {
        "name": "table-deliver",
        "type": "method",
        "parent": "6",
        "agents": [
            "r1",
            "Patient1"
        ],
        "children": [
            "8",
            "9",
            "10"
        ]
    },
    "8": {
        "name": "navto",
        "type": "action",
        "parent": "7",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r1"
        ],
        "children": ""
    },
    "9": {
        "name": "approach-patient-table",
        "type": "action",
        "parent": "7",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r1",
            "Patient1"
        ],
        "children": ""
    },
    "10": {
        "name": "deliver-to-table",
        "type": "action",
        "parent": "7",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r1"
        ],
        "children": ""
    }
}
```

Listing B.3: JSON output for the first iHTN of the Food Logistics Delivery example

```
{
    "0": {
        "name": "ROOT",
        "type": "task",
        "parent": "-1",
        "agents": [
            "r2",
            "r1",
            "Delivery1",
            "Patient1"
        ],
        "children": [
            "1"
        ]
    },
    "1": {
        "name": "ROOT_M",
        "type": "method",
        "parent": "0",
        "agents": [
            "r2",
            "r1",
            "Delivery1",
            "Patient1"
        ],
        "children": [
            "2",
            "6"
        ]
    },
    "2": {
        "name": "GetFood",
        "type": "task",
        "parent": "1",
        "agents": [
            "r1",
            "Delivery1"
        ],
        "children": [
            "3"
        ]
    },
    "3": {
        "name": "food-pickup",
        "type": "method",
        "parent": "2",
        "agents": [
            "r1",
            "Delivery1"
        ],
        "children": [
            "4",
            "5"
        ]
    },
    "4": {
        "name": "navto",
        "type": "action",
        "parent": "3",
        "locations": [
            "Kitchen"
        ],
        "agents": [
            "r1"
        ],
        "children": ""
    },
    "5": {
        "name": "wait-for-food",
        "type": "action",
        "parent": "3",
        "locations": [
            "Kitchen"
        ],
        "agents": [
            "r1",
            "Delivery1"
        ],
        "children": ""
    },
    "6": {
        "name": "DeliverToFetch",
        "type": "task",
        "parent": "1",
        "agents": [
            "r2",
            "r1",
            "Patient1"
        ],
        "children": [
            "7"
        ]
    },
    "7": {
        "name": "fetch-deliver",
        "type": "method",
        "parent": "6",
        "agents": [
            "r2",
            "r1",
            "Patient1"
        ],
        "children": [
            "8",
            "9"
        ]
    },
    "8": {
        "name": "navto",
        "type": "action",
        "parent": "7",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r1"
        ],
        "children": ""
    },
    "9": {
        "name": "FetchMeal",
        "type": "task",
        "parent": "7",
        "agents": [
            "r2",
            "r1",
            "Patient1"
        ],
        "children": [
            "10"
        ]
    },
    "10": {
        "name": "fetch-meal-with-human",
        "type": "method",
        "parent": "9",
        "agents": [
            "r2",
            "r1",
            "Patient1"
        ],
        "children": [
            "11",
            "12"
        ]
    },
    "11": {
        "name": "approach-human",
        "type": "action",
        "parent": "10",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r1",
            "Patient1"
        ],
        "children": ""
    },
    "12": {
        "name": "wait-for-human-to-fetch",
        "type": "action",
        "parent": "10",
        "locations": [
            "RoomA"
            ],
            "agents": [
                "r1",
                "Patient1"
            ],
            "children": ""
        }
}
```

Listing B.4: JSON output for the second iHTN of the Food Logistics Delivery example

```
{
    "0": {
        "name": "ROOT",
        "type": "task",
        "parent": "-1",
        "agents": [
            "r2",
            "r1",
            "Delivery1",
            "Patient1"
        ],
        "children": [
            "1"
        ]
    },
    "1": {
        "name": "ROOT_M",
        "type": "method",
        "parent": "0",
        "agents": [
            "r2",
            "r1",
            "Delivery1",
            "Patient1"
        ],
        "children": [
            "2",
            "6"
        ]
    },
    "2": {
        "name": "GetFood",
        "type": "task",
        "parent": "1",
        "agents": [
            "r1",
            "Delivery1"
        ],
        "children": [
            "3"
        ]
    },
    "3": {
        "name": "food-pickup",
        "type": "method",
        "parent": "2",
        "agents": [
            "r1",
            "Delivery1"
        ],
        "children": [
            "4",
            "5"
        ]
    },
    "4": {
        "name": "navto",
        "type": "action",
        "parent": "3",
        "locations": [
            "Kitchen"
        ],
        "agents": [
            "r1"
        ],
        "children": ""
    },
    "5": {
        "name": "wait-for-food",
        "type": "action",
        "parent": "3",
        "locations": [
            "Kitchen"
        ],
        "agents": [
            "r1",
            "Delivery1"
        ],
        "children": ""
    },
    "6": {
        "name": "DeliverToFetch",
        "type": "task",
        "parent": "1",
        "agents": [
            "r2",
            "r1",
            "Patient1"
        ],
        "children": [
            "7"
        ]
    },
    "7": {
        "name": "fetch-deliver",
        "type": "method",
        "parent": "6",
        "agents": [
            "r2",
            "r1",
            "Patient1"
        ],
        "children": [
            "8",
            "9"
        ]
    },
    "8": {
        "name": "navto",
        "type": "action",
        "parent": "7",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r1"
        ],
        "children": ""
    },
    "9": {
        "name": "FetchMeal",
        "type": "task",
        "parent": "7",
        "agents": [
            "r2",
            "r1",
            "Patient1"
        ],
        "children": [
            "10"
        ]
    },
    "10": {
        "name": "fetch-meal-with-robot",
        "type": "method",
        "parent": "9",
        "agents": [
            "r2",
            "r1",
            "Patient1"
        ],
        "children": [
            "11",
            "12",
            "13",
            "14"
        ]
    },
    "11": {
        "name": "navto",
        "type": "action",
        "parent": "10",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r2"
        ],
        "children": ""
    },
    "12": {
        "name": "approach-robot",
        "type": "action",
        "parent": "10",
        "locations": "",
        "agents": [
            "r2",
            "r1"
        ],
        "children": ""
    },
    "13": {
        "name": "grasp-meal",
        "type": "action",
        "parent": "10",
        "locations": "",
        "agents": [
            "r2",
            "r1"
        ],
        "children": ""
    },
    "14": {
        "name": "deliver-meal-to-patient",
        "type": "action",
        "parent": "10",
        "locations": [
            "RoomA"
        ],
        "agents": [
                "r2",
                "Patient1"
            ],
            "children": ""
        }
}
```

Listing B.5: JSON output for the third iHTN of the Food Logistics Delivery example

B.2.2 Food Logistics Pickup

The JSON iHTN files for the Food Logistics Pickup are shown in Listings B.6 and B.7.

```
{
    "0": {
        "name": "ROOT",
        "type": "task",
        "parent": "-1",
        "agents": [
            "r1",
            "r2",
            "Patient1"
        ],
        "children": [
            "1"
        ]
    },
    "1": {
        "name": "ROOT_M",
        "type": "method",
        "parent": "0",
        "agents": [
            "r1",
            "r2",
            "Patient1"
        ],
        "children": [
            "2",
            "15"
        ]
    },
    "2": {
        "name": "PickupDishes",
        "type": "task",
        "parent": "1",
        "agents": [
            "r1",
            "r2",
            "Patient1"
        ],
        "children": [
            "3"
        ]
    },
    "3": {
        "name": "pickup-with-door-opening",
        "type": "method",
        "parent": "2",
        "agents": [
            "r1",
            "r2",
            "Patient1"
        ],
        "children": [
            "4",
            "5",
            "6",
            "7",
            "8",
            "9"
        ]
    },
    "4": {
        "name": "navto",
        "type": "action",
        "parent": "3",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r1"
        ],
        "children": ""
    },
    "5": {
        "name": "navto",
        "type": "action",
        "parent": "3",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r2"
        ],
        "children": ""
    },
    "6": {
        "name": "approach-door",
        "type": "action",
        "parent": "3",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r1"
        ],
        "children": ""
    },
    "7": {
        "name": "approach-door",
        "type": "action",
        "parent": "3",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r2"
        ],
        "children": ""
    },
    "8": {
        "name": "open-door",
        "type": "action",
        "parent": "3",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r1",
            "r2"
        ],
        "children": ""
    },
    "9": {
        "name": "PickDishesTwoRobotsAtLocation",
        "type": "task",
        "parent": "3",
        "agents": [
            "r1",
            "r2",
            "Patient1"
        ],
        "children": [
            "10"
        ]
    },
    "10": {
        "name": "pick-dishes-two-robots-at-location",
        "type": "method",
        "parent": "9",
        "agents": [
            "r1",
            "r2",
            "Patient1"
        ],
        "children": [
            "11"
        ]
    },
    "11": {
        "name": "PickDishes",
        "type": "task",
        "parent": "10",
        "agents": [
            "r1",
            "r2",
            "Patient1"
        ],
        "children": [
            "12"
        ]
    },
    "12": {
        "name": "pick-dishes-with-human",
        "type": "method",
        "parent": "11",
        "agents": [
            "r1",
            "r2",
            "Patient1"
        ],
        "children": [
            "13",
            "14"
        ]
    },
    "13": {
        "name": "approach-human",
        "type": "action",
        "parent": "12",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r1",
            "Patient1"
        ],
        "children": ""
    },
    "14": {
        "name": "wait-for-human-to-place-dish",
        "type": "action",
        "parent": "12",
        "locations": "",
        "agents": [
            "r1",
            "Patient1"
        ],
        "children": ""
    },
    "15": {
        "name": "RetrieveDishes",
        "type": "task",
        "parent": "1",
        "agents": [
            "r1"
        ],
        "children": [
            "16"
        ]
    },
    "16": {
        "name": "dishes-retrieval",
        "type": "method",
        "parent": "15",
        "agents": [
            "r1"
        ],
        "children": [
            "17",
            "18"
        ]
    },
    "17": {
        "name": "navto",
        "type": "action",
        "parent": "16",
        "locations": [
            "Kitchen"
        ],
        "agents": [
            "r1"
        ],
        "children": ""
    },
    "18": {
        "name": "retrieve-dishes",
        "type": "action",
        "parent": "16",
        "locations": [
            "Kitchen"
        ],
        "agents": [
            "r1"
        ],
        "children": ""
    }
}
```

Listing B.6: JSON output for the first iHTN of the Food Logistics Pickup example

```
{
    "0": {
        "name": "ROOT",
        "type": "task",
        "parent": "-1",
        "agents": [
            "r1",
            "r2",
            "Patient1"
        ],
        "children": [
            "1"
        ]
    },
    "1": {
        "name": "ROOT_M",
        "type": "method",
        "parent": "0",
        "agents": [
            "r1",
            "r2",
            "Patient1"
        ],
        "children": [
            "2",
            "15"
        ]
    },
    "2": {
        "name": "PickupDishes",
        "type": "task",
        "parent": "1",
        "agents": [
            "r1",
            "r2",
            "Patient1"
        ],
        "children": [
            "3"
        ]
    },
    "3": {
        "name": "pickup-with-door-opening",
        "type": "method",
        "parent": "2",
        "agents": [
            "r1",
            "r2",
            "Patient1"
        ],
        "children": [
            "4",
            "5",
            "6",
            "7",
            "8",
            "9"
        ]
    },
    "4": {
        "name": "navto",
        "type": "action",
        "parent": "3",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r1"
        ],
        "children": ""
    },
    "5": {
        "name": "navto",
        "type": "action",
        "parent": "3",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r2"
        ],
        "children": ""
    },
    "6": {
        "name": "approach-door",
        "type": "action",
        "parent": "3",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r1"
        ],
        "children": ""
    },
    "7": {
        "name": "approach-door",
        "type": "action",
        "parent": "3",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r2"
        ],
        "children": ""
    },
    "8": {
        "name": "open—door",
        "type": "action",
        "parent": "3",
        "locations": [
            "RoomA"
        ],
        "agents": [
            "r1",
            "r2"
        ],
        "children": ""
    },
    "9": {
        "name": "PickDishesTwoRobotsAtLocation",
        "type": "task",
        "parent": "3",
        "agents": [
            "r1",
            "r2",
            "Patient1"
        ],
        "children": [
            "10"
        ]
    },
    "10": {
        "name": "pick—dishes—two—robots—at—location",
        "type": "method",
        "parent": "9",
        "agents": [
            "r1",
            "r2",
            "Patient1"
        ],
        "children": [
            "11"
        ]
    },
    "11": {
        "name": "PickDishes",
        "type": "task",
        "parent": "10",
        "agents": [
            "r1",
            "r2",
            "Patient1"
        ],
        "children": [
            "12"
        ]
    },
    "12": {
        "name": "pick-dishes-with-robot-at-location",
        "type": "method",
        "parent": "11",
        "agents": [
            "r1",
            "r2",
            "Patient1"
        ],
        "children": [
            "13",
            "14"
        ]
    },
    "13": {
        "name": "pick-patient-dishes",
        "type": "action",
        "parent": "12",
        "locations": "",
        "agents": [
            "r2",
            "Patient1"
        ],
        "children": ""
    },
    "14": {
        "name": "load-dishes",
        "type": "action",
        "parent": "12",
        "locations": "",
        "agents": [
            "r1",
            "r2"
        ],
        "children": ""
    },
    "15": {
        "name": "RetrieveDishes",
        "type": "task",
        "parent": "1",
        "agents": [
            "r1"
        ],
        "children": [
            "16"
        ]
    },
    "16": {
        "name": "dishes-retrieval",
        "type": "method",
        "parent": "15",
        "agents": [
            "r1"
        ],
        "children": [
            "17",
            "18"
        ]
    },
    "17": {
        "name": "navto",
        "type": "action",
        "parent": "16",
        "locations": [
            "Kitchen"
        ],
        "agents": [
            "r1"
        ],
        "children": ""
    },
    "18": {
        "name": "retrieve-dishes",
        "type": "action",
        "parent": "16",
        "locations": [
            "Kitchen"
        ],
        "agents": [
            "r1"
        ],
        "children": ""
    }
}
```

Listing B.7: JSON output for the second iHTN of the Food Logistics Pickup example