Otto-Friedrich-Universität Bamberg

Professur für Angewandte Informatik, insbesondere Smart Environments

Masterarbeit

im Studiengang Angewandte Informatik
der Fakultät Wirtschaftsinformatik und Angewandte Informatik
der Otto-Friedrich-Universität Bamberg

Zum Thema:

Temporal HTN Planning with a Stream of Tasks: Anticipating Interleavability for Replanning

Vorgelegt von:

Felix Haase

Themensteller:
Prof. Dr. Diedrich Wolter

Abgabedatum:
11.04.2024

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Abbreviations

AI Artificial Intelligence
ANML Action Notation Modeling Language
CSP Constraint Satisfaction Problem
FAPE Flexible Acting and Planning Environment
IPC International Planning Competition
HTN Hierarchical Task Network
HDDL Hierarchical Domain Definition Language
ML Machine Learning
PDDL Propositional Domain Definition Language
PSP Plan-Space Planning
RL Reinforcement Learning
SSP State-Space Planning
RAE Refinement Acting Engine
STN Simple Temporal Network
TFD Temporal Fast Downward
QCN Qualitative Constraint Network

Abstract

HTN planning in a dynamic domain with a-priori unknown tasks and uncontrollable changes is a challenging problem. We focus in particular on the problem that occurs when the unknown tasks include deadlines as it might require interleaving a new task with an already existing plan. We introduce a hierarchical domain definition of the game "Overcooked!" as an example of a dynamic domain with a stream of tasks. In "Overcooked!" the task is to fulfill dish orders with strict deadlines by managing up to four cooks. We refine the actor model of the HTN planning system FAPE to support a stream of tasks. We aim to solve the problem of handling the stream of tasks by introducing two methods that optimize plans by anticipating the addition of new tasks. First, we present a new heuristic that evaluates the robustness of a plan in the case of action delays that may arise when interleaving the current plan with a new task. Second, we propose a method that inserts preparations for possible future tasks into the current plan by using the hierarchal domain definition. We find that our robustness heuristic only distributes some of the tasks between actors in the domain but does incentivize plans that are robust enough for the insertion of new tasks. Our preparation insertions achieve the interleaving of tasks and will succeed more likely in the case of tasks that have stricter deadlines. Both approaches cannot outperform the optimal solution that is acquired by optimizing the overall makespan.

1 Introduction

If you think of everyday life, planning is required in many different environments. Be it that you go on vacation and need to plan the trip and attractions to visit, in a warehouse where different parcels need to be moved to different trucks or in a restaurant where several meals must be prepared and served. While humans often naively solve those problems, it becomes very complex in large scenarios like warehouses. Additionally, with the increasing automation using robots, it is not possible to plan and manage this manually.

Artificial Intelligence (AI) planning techniques [Ghallab et al., 2004], which have already been introduced in the first wave of AI [Fikes and Nilsson, 1971], are a way to automate these problems. There are many different ways to represent and solve planning problems and only some of these representations allow the specification of a specific domain. For example, the representation of time is very crucial in many problems, but not all representations or planning systems support it.

Planning techniques can be divided broadly into classical planning, neoclassical planning and control strategies. Classical planning uses a set of actions that can be performed and finds a sequence of actions to achieve a goal state. Neoclassical planning introduces new ways of representing and solving planning problems, such as with a Constraint Satisfaction Problem (CSP). Control strategies such as Hierarchical Task Network (HTN) planning allow domain designers to define more strictly which actions are allowed and how they can be used. In HTN planning specifically, the problem is instead represented using high-level tasks – similar to recipes in a cookbook – that can be decomposed into subtasks or actions that can be performed directly. This reduces the search space a planning algorithm has to consider and may also guide the algorithm to more preferable plans.

In recent years, Machine Learning (ML) techniques, specifically Reinforcement Learning (RL), is used more often to solve some of these planning problems. However, while ML is a helpful tool to optimize planning in a specific domain, it may introduce several problems. First, it cannot be verified that the ML algorithm acts accordingly and consistently. Next, ML systems have to be trained in each environment with a significant amount of data and compute time which may not be available. And lastly, even if an ML system has been trained in the environment, that does not entail that it performs as well in new problem instances.

On the contrary, algorithmic approaches to planning are verifiable and can be used in many different domains. However, most planning algorithms assume a strict separation between planning and execution. This is especially relevant in dynamic domains where not all tasks are known initially or uncontrollable events may occur. Using online planning, a planner can dynamically receive domain updates and adjust the plan accordingly. Then the individual actions can then be executed in the domain using acting. Most variants of online planning consider plan repair or replanning in the case of a planning or action failure. However, it is very common to receive additional tasks that sometimes need to be executed in a given time window. While replanning approaches may help in the integration of these additional tasks, they will fail for especially strict deadlines, if the previous tasks were executed too lazily. To achieve better success rates, an online planning algorithm has to consider the possibility of new, additional tasks being inserted in the future.

1.1 The Game "Overcooked!"

We consider the game "Overcooked!" and its sequel "Overcooked!2" developed by Ghost Town Games and Ltd. and Team 17 as an example of a dynamic domain throughout this work. Both games are very similar, so we do not differentiate between them and call our domain *Overcooked*. The game models a restaurant kitchen with many simplifications that make the plan abstraction easier and more directly applicable. The core part of the game focuses on collaboration and can be played with up to four players. The players are given a small number of recipes in each level, for example, a lettuce salad with tomatoes or a burger with a patty, tomatoes and lettuce. The ingredients often need to be prepared (e.g. by chopping or frying them), then arranged on a plate and finally delivered to a conveyor belt that represents the clients. During playing, multiple challenges may occur: A fire may break out that has to be extinguished or players may be separated, preventing them from passing ingredients directly to one another. In the latter case, ingredients are passed by placing them on a counter or throwing them over obstacles between cooks for preparation and arranging. Additionally, the orders are not known in advance but are received during play and have a strict deadline. The goal of the game is to successfully finish as many orders as possible; when a deadline is missed it counts as a penalty. This game is a perfect playground for evaluating and improving planning approaches, as the game poses challenges similar to real life while offering many optional features that challenge the planning domain creation and planning algorithms. With an appropriate interaction layer, the game may also be used to test acting approaches without the need for a physical robot. As the game has the option for up to four players, the game can be used to evaluate planning for parallel actions when all cooks are supervised by a single planner, or in a multi-agent setting with possible human-robot interaction. To appropriately fulfill tasks, the agent(s) need to be able to receive tasks during execution. The Overcooked AI Benchmark [Carroll et al., 2019] uses a simplified version of the game and is used for the evaluation of human-AI collaboration.

1.2 Problem Statement and Overview

We consider the planning problem in a domain with temporal constraints and an a-priori unknown stream of tasks with deadlines. The goal of this work is to achieve higher success rates in the case of shorter deadlines in the stream of tasks by interleaving new tasks with currently executing tasks. We propose a robustness heuristic and the insertion of preparations to achieve this goal in the context of HTN planning. The results show that our robustness heuristic results in at most a small improvement, while the insertion of preparations can optimally result in a higher success rate.

We first introduce relevant background information on classical and HTN planning as well as the concepts of online planning and acting in Section 2. Then we go over the work related to our problem in Section 3. We introduce our approach with the robustness heuristic and preparation insertion together with the application domain and the actor model with a stream of tasks in Section 4. We then evaluate our approach in the domain *Overcooked* in Section 5 and discuss the evaluation and our approach in Section 6. Finally, we conclude this work in Section 7 with a summary and an outlook.

2 Background

This section introduces concepts and definitions relevant to this work. Section 2.1 introduces classical planning to illustrate the background of HTN planning. Then, Section 2.2 introduces HTN planning and temporal HTN planning as an extension of classical planning that is used in this work. Section 2.3 concludes this section by introducing the concepts of acting and online planning which are relevant for handling the dynamic domain.

2.1 Classical planning

Classical planning [Ghallab et al., 2004], first introduced by STRIPS [Fikes and Nilsson, 1971], is the widely adopted approach to planning upon which most extensions and other planning approaches build. A classical planning problem consists of a domain (for example *Overcooked*) and a specific planning problem, e.g., that a client needs to have a burger in the end. The domain specifies variables and state transition functions, called actions, while the problem gives an initial state and a goal state. Then, a planner has to find a sequence of actions, called a plan π, that can transform the initial state to the goal state. The following definitions are based on Ghallab et al. [2004, chap. 1,2] and Höller et al. [2019].

Definition 2.1 (Classical Planning Domain) *A classical planning domain is a tuple* $\Sigma = \langle \mathcal{L}, \mathcal{O}, \delta \rangle$.

- $\mathcal{L}$ *is a set of propositional environment variables.*
- $\mathcal{O}$ *is a set of operator symbols.*
- δ *is a triple* $(prec, add, del)$ *of functions* $f : \mathcal{O} \to 2^{\mathcal{L}}$ *that define the state transitions of actions as preconditions* $prec$ *and effects* add, del.

An action symbol $a \in \mathcal{O}$ *is* applicable *in a state* $s \subseteq L$ *iff* $prec(a) \subseteq s$. *When it is* applicable, *the resulting state is defined by the function* $\gamma : A \times 2^L \to 2^L$ *with* $\gamma(a, s) = (s \setminus del(a)) \cup add(a)$.
A plan $\pi = \{a_1, \ldots, a_n\}$ *is a sequence of actions.* π *is* executable *in a state* s_0 *iff all actions* $a_1, \ldots, a_n$ *are* applicable *in* $s_0, \ldots, s_{n-1}$, *where* $s_k = \gamma(a_k, s_{k-1})$. s_n *is then defined as the state produced by applying* π *in state* s, *denoted as* $\gamma(\pi, s)$.

An action is an expression of the form $a(r_1, \ldots, r_n)$ where a is the action symbol and $r_1, \ldots, r_n$ are terms (also called parameters). The parameters define the exact preconditions and effects of an action. Constant symbols D are symbols that describe objects in the domain such as a $Cook$ and are possible instantiations for terms r. Typing is a commonly used extension for planning that organizes constants in the set L. A type $D_t \subseteq D$ restricts variables in actions to be of a specific type such as $Cooks$. Types can be composed of other types by union or intersection (e.g. $Persons = Cooks \cup Clients$)

Definition 2.2 (Classical Planning Problem) *A classical planning problem is a tuple* $P = \langle \Sigma, s_0, g \rangle$.

- Σ *is a classical planning domain.*
- $s_0 \subseteq L$ *is the initial state.*
- $g \subseteq L$ *is the goal state.*

Given a problem P, the task in classical planning is to find a sequence of actions $\pi = \{a_1, \ldots, a_n\}$ that transform the initial state into the goal state. The goal state g only defines which propositional environment variables must be included in the final state, but other variables may be present as well.

2.2 HTN Planning

Hierarchical Task Network (HTN) planning is a planning paradigm that allows for the specification of complex tasks by the definition of subtasks. Therefore it is viewed as a control strategy for planning [Ghallab et al., 2004, chap. 11]. Generally, the goal in HTN planning is to decompose complex tasks into *primitive* executable actions. For instance, preparing a simple burger in *Overcooked* requires a bun, a burger patty and lettuce. Each of these ingredients may involve multiple preparation steps like chopping or frying, that again can be grouped into subtasks themselves. Eventually, we are left with an *executable* sequence of simple steps to create the requested burger. This is conceptually different from classical planning, as it defines tasks to execute instead of a goal state to achieve.

The basic notions of HTN planning are introduced in this section to serve as a basis for the definitions of temporal HTN planning in Section 2.2.1. The following definitions are based on Ghallab et al. [2004, chap. 11], Höller et al. [2019] and Bit-Monnot [2016].

Formally, a task is an expression of the form $\tau(r_1, \ldots, r_n)$ where $\tau \in \mathcal{O}$ is a task symbol and $r_1, \ldots, r_n$ are the parameters. There exist *primitive* tasks where τ is an action symbol and *nonprimitive* tasks.

Definition 2.3 (Task Network) *A task network is an acyclic digraph* $tn = (T, \prec, \alpha)$.

- T *is a set of tasks.*
- $\prec \subseteq T \times T$ *is a set of edges that define a strict partial ordering relation between the tasks*
- $\alpha : T \to \mathcal{O}$ *is a function mapping the tasks to operator symbols.*

A task network tn is called *primitive* if all of the task nodes refer to *primitive* tasks.

Definition 2.4 (HTN Planning Domain) *An HTN planning domain is a tuple* $\Sigma = \langle \mathcal{L}, \mathcal{O}, \mathcal{M}, \delta \rangle$. $\mathcal{L}, \mathcal{O}$ *are defined as in classical planning.* $\mathcal{M}$ *is a set of methods* $(\tau, tn), \tau \in \mathcal{O}$, *where* τ *is* nonprimitive, *that describe possible decompositions from one task into a task network.*

Definition 2.5 (Task Decomposition) *A method $m = (\tau, tn)$ decomposes a task network $tn_1 = (T_1, \prec_1, \alpha_1)$ into $tn_2 = (T_2, \prec_2, \alpha_2)$ denoted $tn_1 \xrightarrow{t,m}_D tn_2$, if there is a task node $t \in T_1$ with $\alpha_1(t) = \tau$. A copy $tn' = (T', \prec', \alpha')$ of tn is created that has no overlapping task with tn_1, i.e. $T_1 \cap T' = \emptyset$. Then tn_2 is defined as:*

$$\begin{aligned} tn_2 =& ((T_1 \backslash \{t\}) \cup T', \prec' \cup \prec_D, (\alpha_1 \backslash \{t \mapsto \tau\}) \cup \alpha') \\ \prec_D =& \{(t_1, t_2) | (t_1, t) \in \prec_1, t_2 \in T'\} \cup \\ & \{(t_1, t_2) | (t, t_2) \in \prec_1, t_1 \in T'\} \cup \\ & \{(t_1, t_2) | (t_1, t_2) \in \prec_1, t_1 \neq t \wedge t_2 \neq t\} \end{aligned}$$

Definition 2.6 (HTN Planning Problem) *An HTN Planning Problem is defined as a tuple $P = \langle \Sigma, s_0, tn_I \rangle$.*

- *$s_0 \subset L$ is the initial state.*
- *tn_I is the initial task network, consisting of the tasks that must be executed.*

A problem is called total order iff $\prec$ in tn_I and all task networks in methods $(\tau, tn) \in \Sigma$ defines a total ordering. Otherwise, it is called partial order.

The HTN planning problem is solved by decomposing the initial task network into an *executable* plan. Erol et al. [1994] find that solving total order problems has double EXPTIME and EXPSPACE-hard complexity [Papadimitriou, 2003], while partial order problems are undecidable.

Definition 2.7 (HTN Planning Solution) *If $tn = (T, \prec, \alpha)$ is* primitive*, then a plan $\pi = \{a_1, \ldots, a_n\}$ is a solution for P if all of the following conditions hold:*

- *The actions in π correspond to the tasks in T*
- *π is* executable *in s_0*
- *the total ordering $\{a_1, \ldots, a_n\}$ satisfies the ordering in $\prec$*

There are two conceptually different approaches to arriving at a plan π for an HTN planning problem. First, the State-Space Planning (SSP) approach as in Nau et al. [1999] is shown in algorithm 1. The search is executed as a forward search and is started with an initial state s_0. The first task in the task network is selected based on the ordering $\prec$ in the forward search. *primitive* tasks are applied directly and change the state, *nonprimitive* tasks are decomposed and change the task network. The algorithm is then applied recursively for the new state and task network.

Second, the Plan-Space Planning (PSP) approach as in Ghallab et al. [2004, chap. 5] is shown in algorithm 2. The search problem here is abstracted from states and the search

Algorithm 1: SHOP

Input: s; tn; Σ
Output: π
if $tn = \varnothing$ **then**
  return $\varnothing$
$\tau \leftarrow$ first task in tn;
$tn' \leftarrow$ remaining tasks in tn;
if τ *is* primitive *and there is an action for* τ applicable *in* s **then**
  nondeterministically choose an action a for τ;
  $\pi \leftarrow$ SHOP($\gamma(s,a)$,tn',Σ);
  if π = *FAIL* **then**
    return *FAIL*
  return *cons*(p,π)
else if τ *is not* primitive *and there is a decomposition of* τ applicable *in* s **then**
  nondeterministically choose a decomposition m of τ in s;
  $tn \xrightarrow{\tau,m}_D tn'$;
  return *SHOP*(s,tn',Σ)
else
  return *FAIL*

space consists of partial plans. In HTN planning, a task network that is not *necessarily primitive* is a partial plan. As such this approach is similar to constraint solving as it is a search over all possible solutions. The flaws shown in the algorithm may be unrefined tasks or causal threats[1]. Resolvers are modifications of a partial task network, that resolve the possible flaws. They are chosen nondeterministically to expand the task network into a set of search nodes. The representation of δ in PSP is often implemented using Constraint Satisfaction instead of direct manipulation of states [Georgievski and Aiello, 2015].

2.2.1 Temporal HTN Planning

Temporal HTN planning includes explicit information about time in the domain. This allows modeling task durations, concurrent execution, start points and deadlines for tasks. It is common to encode a planning problem as a CSP [Ghallab et al., 2004, chap. 8]. Stock et al. [2015a] introduced the notion of a MetaCSP, that solves several CSPs, in their case, one for variable bindings, and a Simple Temporal Network (STN) citepdechterTemporalConstraintNetworks1991 managing the temporal constraints on time points. This approach is also used in Flexible Acting and Planning Environment (FAPE) [Bit-Monnot, 2016]. The main components required to model the temporal HTN planning problem are introduced in the following paragraphs. The definitions are based on Ghallab et al. [2004, chap. 14] and Bit-Monnot [2016].

Time is represented quantitatively using time points. A time point is designated by a temporal variable (e.g. t_1). Constraints over temporal variables can be expressed using the simple arithmetic operators $=, \leq, \geq$ to specify absolute (e.g. $t_1 \geq 10$) or relative constraints (e.g. $t_1 + 5 \leq t_2$). The temporal variables are attached to events like the start

[1] During planning, causal links are added between actions to state that the cause of an action is the effect of another action, causal threats are then possible violations of these links.

Algorithm 2: PSP

Input: tn; Σ
Output: π
Flaws $\leftarrow$ flaws in tn;
if *Flaws* $= \varnothing$ **then**
  return π *as instantiation of* tn
$\varphi \leftarrow$ select a flaw in Flaws;
Resolvers $\leftarrow$ resolvers for φ;
if *Resolvers* $= \varnothing$ **then**
  return *FAIL*
nondeterministically choose $\rho \in$ *Resolvers*;
$tn' \leftarrow$ Transform(tn, ρ);
return *PSP*$(tn'; \Sigma)$

of an action or an instant at which a condition must hold.

A nontemporal variable $c \in D_t$ is a variable with a domain $dom(c) \subseteq D_t$. A numeric variable i has a domain $dom(i) \subset \mathbb{N}$ which is a finite subset of integers. A constraint over a set of variables $\{v_1, \ldots, v_n\}$ is a pair $(\langle v_1, \ldots, v_n \rangle, \gamma_R)$ where $\gamma_R \subseteq dom(v_1) \times \cdots \times dom(v_n)$ is the relation of the constraint, giving the allowed values for the tuple of variables. γ_R can be given as a table of allowed values or a function, e.g., $distance(d, d') = \Delta$ is met when Δ is the distance between d and d'. Numeric variables can also appear in temporal constraints. For instance $(distance(d, d') = \Delta) \wedge (t_s + \Delta \leq t_e)$ describes a constraint that defines that the delay between t_s and t_e is equal to the distance Δ between d and d'. This constraint may occur as the duration of an action $move(d, d')$, with time points t_s as the actions start and t_e as the actions end.

Definition 2.8 (State Variable) *An n-ary state variable* $sv(v_1, ..., v_n)$ *is a function* $x : time \times D_1^x \times \cdots \times D_n^x \rightarrow D_{n+1}^x$ *that maps time and a set of variables to a variable. Each* $D_i^x \subseteq D$ *is the union of one or more types. Time in a state variable is usually kept implicit and we say that sv has the value v at time t if* $sv(t, x_1, ..., x_n) = v$.

State variables are a convenient way to describe the change of a variable over time as a function of that variable over time. For instance $Ingredient.location : Time \times Ingredients \rightarrow Persons \cup Placement\ Areas \cup Tablewares$ specifies the location of any ingredient at any point in time, which may be in a person's hand, in an area or on a tableware. Complete knowledge about the value of each variable is not required for planning.

Definition 2.9 (Fluent) *A fluent is a pair of a state variable sv and a value v, denoted* $\langle sv = v \rangle$ *and is said to hold at time t if sv has the value v at time t.*

A fluent $Ingredient.location(lettuce1) = cook1$ may then hold at several points in time, for example between $cook1$ picking it up and $cook1$ placing it on a plate. Fluents are used to define and are extended by temporally qualified assertions.

Definition 2.10 (Temporally qualified assertions) *A temporally qualified assertion α is an assertion over a state variable sv in a temporal interval $[t_s, t_e]$. The temporal interval may also describe an instant t denoted as $[t]$.*
A persistence assertion, denoted $\alpha = \langle [t_s, t_e] sv = v \rangle$, requires sv to keep the same value v over the temporal interval t_s, t_e.
A change assertion, denoted $\alpha = \langle [t_s, t_e] sv : v_1 \mapsto v_2 \rangle$ describes that sv changes its value from v_1 at time t_s to v_2 at time t_e.
A temporally qualified assignment denoted $\alpha = \langle [t] sv := v \rangle$ describes that sv has the value v at time t.

Persistence assertions are typically used to express requirements such as goals or preconditions of actions. For instance $\langle [100, 200] loc(lettuce1) = counter1 \rangle$ requires the instance *lettuce*1 to be at the location *counter*1 from time 100 until time 200. The change assertion is typically used to combine pre and post-conditions of most planners into one and additionally to explicitly describe the value change as a process rather than an instantaneous effect. For instance, $\langle [100, 200] chopped(lettuce1) : false \mapsto true \rangle$ describes that the instance *lettuce*1 is getting chopped from time 100 until time 200. An assignment is a special case of a change assertion $\alpha = \langle [t-1, t] sv : any \mapsto v \rangle$ where *any* is a variable with a possibly infinite domain. For instance $\langle [0] chopped(lettuce1) := false \rangle$ states that the instance *lettuce*1 is not chopped at time 0.

State variables are a replacement for the propositional variables $\mathcal{L}$ and therefore also states. They make time explicit and move change states along individual variables instead of globally. They offer a more compact representation and higher expressivity than propositional variables. Additionally, it is now implicitly defined that an object can only be in one place at any time, which has to be done using external domain constraints in classical representations. Lastly, they do not have to be the result of an action but value changes can be predefined at any time interval.

Definition 2.11 (Timeline) *A timeline is a pair $(\mathcal{F}, \mathcal{C})$ where $\mathcal{F}$ is a set of temporally qualified assertions on a state variable and $\mathcal{C}$ is a set of constraints on objects and temporal variables appearing in $\mathcal{F}$.*

A timeline describes a partial evolution of the state variable over time. It is only partial because the timeline is supposed to be consistent in itself. Due to the inconsistent states while planning, several timelines for a single state variable may exist in a partial plan, but they must be combined at the end. A timeline is consistent if there exists a consistent instantiation of it, and it is necessarily consistent if all instantiations are consistent Bit-Monnot [2016].

Definition 2.12 (Chronicle) *A chronicle is a triple $\phi = (\pi, \mathcal{F}, \mathcal{C})$ where π is a partial plan composed of action instances and unrefined tasks and $(\mathcal{F}, \mathcal{C})$ is a set of timelines.*[2]

The chronicle in Bit-Monnot et al. [2020] is a temporal and hierarchical extension of partial plans in plan-space planning approaches [Ghallab et al., 2004, chap. 5] and extends the

[2]Technically it is a unification of timelines, but all background literature calls it a set of timelines.

notion of chronicles in [Ghallab et al., 2016, sec. 4.2.4] with the member π to keep track of tasks and actions. Since there is no explicit task network, the partial plan uses the same symbol π as in Definition 2.7 for a set of actions and unrefined tasks.

Tasks, methods and actions are defined differently in FAPE compared to the common HTN approach. Action templates combine the notion of an action and a method. Tasks $\mathcal{T}$ are no longer split between *primitive* and non-*primitive* tasks. Bit-Monnot [2016] argues that the distinction between *primitive* and *nonprimitive* tasks as well as actions and methods is only an artificial restriction and that it has no impact on the planning implementation.

Definition 2.13 (Action Template) *An action template is a tuple* $(head(A), task(A), dependent(A), subtasks(A), \mathcal{F}_A, \mathcal{C}_A)$ *where*

- $head(A) = a(r_1, \ldots, r_n)$ *is the action symbol* a *and the parameters* $r_1, \ldots, r_n$ *of* A*.*
- $task(A) \in \mathcal{T}$ *is the task* $\tau(r_1, \ldots, r_n)$ *that* A *achieves.*
- $dependent(A) \in \{\top, \bot\}$ *defines if* A *is task dependent, meaning if an action* a *fulfilling* A *can only be inserted if a* $task(A)$ *is achieved through* a*.*
- $subtasks(A)$ *is a set of temporally qualified subtasks written as* $\langle [t_s, t_e]\tau \rangle$*.*
- $(\mathcal{F}_A, \mathcal{C}_A)$ *is a consistent set of timelines.*

The start and end time points of the action template and the subtasks are all left implicit. An action template is high-level, if $subtasks(A) \neq \varnothing$, otherwise it is *primitive*. A method (c, tn) is part of a high-level action template through $subtasks(A)$ and $\mathcal{C}_A$. A task is *primitive*, if there is only one action template A that achieves it and that action template is *primitive*.

Definition 2.14 (Task Decomposition in FAPE) *A chronicle* $\phi_1 = (\pi_1, \mathcal{F}_1, \mathcal{C}_1)$ *can be refined into a chronicle* $\phi_2 = (\pi_2, \mathcal{F}_2, \mathcal{C}_2)$ *by decomposing an unrefined task* $\tau \in \pi_1$ *with a new action instance* a*. This transformation is denoted* $\phi_1 \xrightarrow{\tau, a}_D \phi_2$ *and does the following modifications to* ϕ_2

$$\begin{aligned} \pi_2 &\leftarrow (\pi_1 \backslash \{\tau\}) \cup \{a\} \cup subtasks(a) \\ \mathcal{F}_2 &\leftarrow \mathcal{F}_1 \cup \mathcal{F}_a \\ \mathcal{C}_2 &\leftarrow \mathcal{C}_1 \cup \mathcal{C}_a \cup \{task(a) = \tau\} \end{aligned}$$

This decomposition definition is much more straightforward compared to Definition 2.5, as it leaves the burden of constraint solving on the MetaCSP. Contrary to the traditional approach with methods, a high-level action instance a is added to the partial plan, since it may also define state variable changes and may thus be required for the execution. The additional constraint $\{task(a) = \tau\}$ ensures that a does refine τ correctly by unifying the parameters and time points of a and τ.

Definition 2.15 (Temporal HTN Planning Domain) *A temporal HTN planning domain is a tuple* $\Sigma = \langle \mathcal{SV}, \mathcal{O}, \mathcal{A} \rangle$. $\mathcal{O}$ *is defined equivalently to the standard HTN Planning Domain.* $\mathcal{SV}$ *is a set of state variables.* $\mathcal{A}$ *is a set of action templates.*

Definition 2.16 (Temporal HTN Planning Problem) *A temporal HTN planning problem is a pair* $P = \langle \Sigma, \phi_0 \rangle$. Σ *is a temporal HTN planning domain.* ϕ_0 *is the initial chronicle.*

While the definitions for temporal HTN planning in FAPE are quite different from HTN planning, fundamentally they are very similar. A chronicle ϕ is a temporal extension of a task network $\langle T, \prec, \alpha \rangle$. The tasks T are contained in π, and $\prec$ is contained in the constraints $\mathcal{C}$. α does not exist anymore, but the mapping from actions and tasks to their names is similar.

Definition 2.17 (Temporal HTN Planning Solution) $\phi = (\pi, \mathcal{F}, \mathcal{C})$ *is a solution for* P *if all of the following conditions hold:*

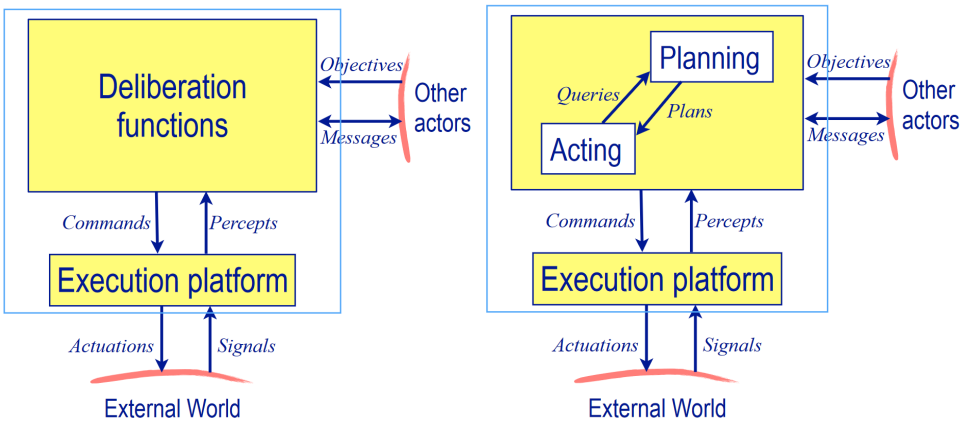
- ϕ *is reachable from* ϕ_0.
- π *does not include tasks.*
- *all assertions in* $\mathcal{F}$ *are causally supported.*
- $(\mathcal{F}, \mathcal{C})$ *is a set of necessarily consistent timelines.*

In temporal HTN planning, it is infeasible to use a SSP approach as actions might be required to be parallel and SSP cannot insert several actions at once. Therefore FAPE uses PSP as in Algorithm 2.

2.2.2 Task Insertion

HTN planning with task insertion [Geier and Bercher, 2011] is a formalism that lies between classical planning and HTN planning. The idea here is to allow the insertion of tasks into the task network during planning, without requiring them to be valid decompositions of compound tasks. This approach is helpful for example in domains where not all possible hierarchical tasks can be captured, and allows an agent to respond to unforeseen situations. Specifically with task insertion, HTN planning becomes decidable [Geier and Bercher, 2011] and removes the need for recursive methods [Bercher et al., 2019]. Additionally, it allows the usage of classical planning heuristics in combination with HTN planning heuristics.

FAPE supports task insertion, however there it is called and defined as action insertion. The initial chronicle can define persistence assertions as the goals a planner needs to achieve. An action is insertable if it is not marked *dependent* in the action template definition. There is then a separate plan modification action insertion to add the action to the chronicle ϕ that is very similar to Definition 2.14.



(a) With generic deliberation function (b) With only planning and acting

Figure 1: Conceptual view of an actor from Ghallab et al. [2016]

Definition 2.18 (Action Insertion) *A chronicle* $\phi_1 = (\pi_1, \mathcal{F}_1, \mathcal{C}_1)$ *can be refined into a chronicle* $\phi_2 = (\pi_2, \mathcal{F}_2, \mathcal{C}_2)$ *by inserting a free action instance* a*, i.e.* a *is not task-dependent. This transformation is denoted* $\phi_1 \xrightarrow{a}_I \phi_2$ *and does the following modifications to* ϕ_2

$$\pi_2 \leftarrow \pi_1 \cup \{a\} \cup subtasks(a)$$
$$\mathcal{F}_2 \leftarrow \mathcal{F}_1 \cup \mathcal{F}_a$$
$$\mathcal{C}_2 \leftarrow \mathcal{C}_1 \cup \mathcal{C}_a$$

The distinction between task insertion and action insertion is important since the insertion of a task allows the decomposition into multiple different actions. Additionally, it is easier to evaluate if it is possible to insert an action than a task since tasks do not have any constraints that can be used for validation.

2.3 Online Planning and Acting

The offline planning approach described in the previous sections does not yet describe how the plan will be executed in an environment like the game *Overcooked* and what happens if new tasks are received. In Ghallab et al. [2016] an actor is described as a system that uses deliberation functions and an execution platform to interact with the external environment and other actors. In particular, deliberation functions describe an interaction between acting, planning and other deliberation functions such as observing, monitoring, goal reasoning and learning [Ingrand and Ghallab, 2017] (see Figure 1). Directly relevant for this use case are only acting and planning.

Acting is then defined as the decision on how to perform actions given by a planner, considering the current state of the observed environment. It also involves reacting to unexpected states and triggering plan repairs or replanning in the online planning algorithm. Such an unexpected state is in our case the addition of new tasks during execution.

The actor architecture of FAPE (see Figure 2) implements this general model. The activity manager takes the role of both acting and state management and requests the planner for

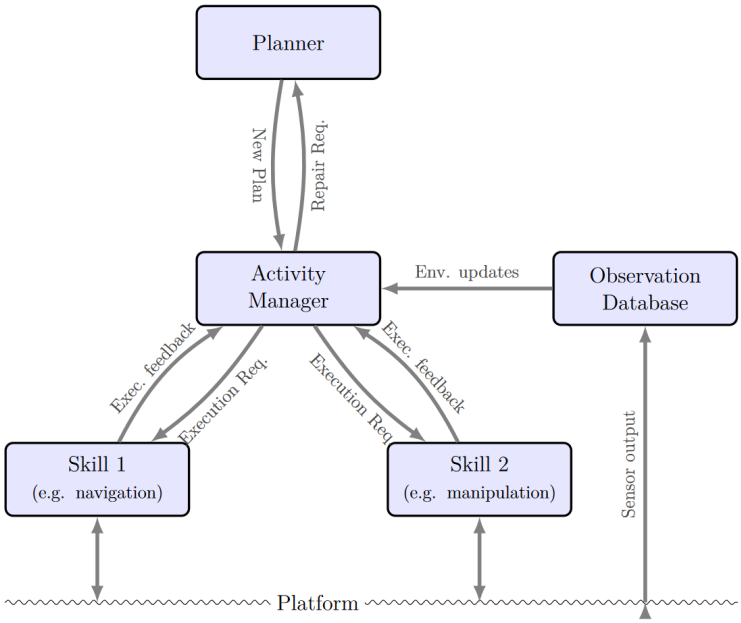


Figure 2: General architecture of FAPE for online planning and acting

updates. The skills are concrete implementations of action executions that are selected by the activity manager. The observation database tracks changes in the environment and informs the activity manager that needs to decide if plan repair is necessary. Additionally, FAPE includes a dispatchable STN, that allows marking time points as executed and incrementing the current time. When the current time is changed, all time points that are not executed have to take place after the current time. This allows to mark actions as executing or executed and the execution of some tasks is not possible anymore.

Online planning [Ghallab et al., 2016, sec. 2.6.2] is an algorithm that can update and refine a plan when a plan modification occurs. A plan modification in this case can be a failed action that has to be removed, a change in the environment or a new task. Online HTN planning [Dvorák et al., 2014] has to take the already executed actions and the task hierarchy into account. There are two possible procedures:

- Replanning: remove all pending actions and find an alternative plan.
- Plan repair: remove the smallest set of problematic actions and find alternative decompositions.

While replanning and plan repair will arrive at a successful plan if it exists, they have several different advantages and disadvantages. Replanning is a simple approach where a partial search of the initial problem is executed, while plan repair has to decide first which actions to remove and then execute a search. Plan repair is often implemented as an additional local search around conflicting actions [Bajada et al., 2014] while replanning can use the same search procedure. In the worst case, plan repair must remove all pending actions which is equal to replanning. When focusing on plan stability [Fox et al., 2006], plan repair will by design keep more of the same actions compared to replanning.

3 Related Work

The game "Overcooked!" has seen some attraction as an application domain in recent years. Classical planning using the Propositional Domain Definition Language (PDDL) has been considered [Yuxin Liu et al., 2020] as well as Human-Robot Interaction [Bishop et al., 2020, Rosero et al., 2021], Human-AI Interaction with RL [Carroll et al., 2019, Charakorn et al., 2020, Knott et al., 2021, Nalepka et al., 2021, Fontaine et al., 2021, Zhao et al., 2023, Sarkar et al., 2022, Ruhdorfer, 2023] and Bayesian Learning [Ribeiro et al., 2022]. The Overcooked AI benchmark introduced by Carroll et al. [2019] is however only concerned with cooperation and does not take more complex tasks into account. The domain only includes a single recipe that is supposed to be produced as often as possible. The RL agents can not correspond to individual tasks, but can only continually perform a single task. Additionally, the agents need to be trained for each possible task, making the agents infeasible to support many different tasks. The Bayesian learning approach is more flexible, as it can adjust online to the domain. It cannot however respond to individual tasks either.

The *Overcooked* planning problem is similar to dynamic pickup and delivery problems [Berbeglia et al., 2010] as it also includes picking up objects and transporting them. The International Planning Competition (IPC) [Taitler et al., 2024] includes many similar problems as classical and HTN domains such as the *Barman* or *Woodworking* problems. There are however no dynamic domains in the IPC. The Flatland challenge Mohanty et al. [2020] on the other hand, is a RL competition that considers the complex problem of large train networks that need to be controlled dynamically. The Airlift challenge [Bertsimas et al., 2019] is a competition at the 2024 ICAPS conference[3] that focuses on the problem of dynamic parcel delivery with planes, where some routes may be unavailable for some time.

Support for temporal HTN planning has been implemented in the systems CHIMP [Stock, 2017] and FAPE [Bit-Monnot et al., 2020]. CHIMP uses a custom domain definition language based on the SHOP2 language, while FAPE uses an extended version of Action Notation Modeling Language (ANML). There is a proposal for HDDL2.1 [Pellier et al., 2023] that adds support for the durative actions and time constraints from PDDL2.1 [Fox and Long, 2003] to the Hierarchical Domain Definition Language (HDDL) language [Höller et al., 2020], a hierarchical version of PDDL. The HDDL2.1 proposal is however not supported by any planner yet. Aries [Bit-Monnot and Godet, 2024] is a promising planning system that supports temporal HTN planning, however, at the time of writing the system was still in active development. The Unified-Planning system [Framba et al., 2024] by the AIPlan4EU H2020 Project[4] is an interesting project in this context, as it aims at implementing a common python API that can interact with any planner.

Acting and online planning have been considered in several ways. The Refinement Acting Engine (RAE) [Ghallab et al., 2016] introduces a general framework for acting using refinement methods. Refinement methods are similar to HTN methods in the definition, but the actor will only execute one refinement, so there is no plan created for the whole problem. It is possible to use online planning in RAE, however they only consider classical planners. Lastly, RAE focuses on a single-actor model, which is not directly considered

[3] See icaps24.icaps-conference.org/competitions/airlift

[4] See aiplan4eu-project.eu

here. FAPE [Bit-Monnot, 2016] implements acting and planning with temporal HTN domains, but cannot react to messages from users or other actors and can therefore also not add tasks during the execution. Additionally, it does not support online planning with fully hierarchical domains. Patra et al. [2020] integrates RAE with learning and planning, but because they use offline learning it can only be used in known domains. Bansod et al. [2021] present a reentrant HTN planner, building on the algorithms for classical planning provided in RAE. They do however not include temporal HTN planning. Turi and Bit-Monnot [2022] integrates temporal HTN planning and RAE, but they only present preliminary results as they use the unfinished planning system Aries [Bit-Monnot and Godet, 2024]. Boella and Damiano [2002] introduce an algorithm for replanning by keeping the plan mostly intact and removing actions until the partial plan subsumes more promising refinements. Then the planning process is restarted. Bansod et al. [2022] use a similar approach for replanning in HTN, but remove all pending actions from the hierarchy. Plan repair on the other hand tries to only remove a minimal part of a failing plan. Van Der Krogt and De Weerdt [2005] implement plan repair as an extension of planning, by considering the removal of actions from a plan as a valid plan modification during plan repair. A similar approach is used by Bajada et al. [2014], while using a plan quality metric as a target. Plan commitment [Babli et al., 2023] keeps the commitments made in a multi-agent setting intact and reduces the revisions necessary between agents. Task merging [Stock et al., 2015b] allows the combination and utilization of multiple actions for the same tasks and thus optimizing plans with new tasks during execution. To our knowledge, no approach has yet used heuristics or task insertion for future tasks, to improve success rates of execution in dynamic environments with new tasks.

4 Approach

We formulate the planning problem in the domain *Overcooked* as a temporal HTN planning problem in ANML in Section 4.1. We rely on acting to be able to respond to tasks that are received after the beginning of execution. The architecture for acting in FAPE Bit-Monnot [2016] is extended to handle this stream of tasks (see Section 4.2).

The particular problem to solve is that the received tasks have associated deadlines, which may prove impossible to achieve when simply solving tasks in sequence. The goal is to optimize the current plans, such that the new tasks can be inserted successfully. We consider two options for solving this optimization problem:

1. In Section 4.3 we introduce a heuristic for the robustness of a plan. This is expected to increase the ability to insert a new task at a later point in time.
2. In Section 4.4 we then introduce preparation insertion as an algorithm that uses the hierarchical domain definition to determine which tasks can be prepared given a set of possible tasks that may be received.

4.1 *Overcooked* Domain Definition

In this Section, we define the *Overcooked* domain in ANML Smith et al. [2008] that is used by FAPE. We only present an overview and the relevant decisions here. The complete domain is available in Appendix B.

The constants of the domain such as cooks or ingredients are defined as a type hierarchy. FAPE does not support multiple inheritance, so an ingredient cannot be choppable and fryable at the same time. We add relevant fluents to the types such as the location of an ingredient.

We define the durations of actions as constant functions so that they can be set for each problem and also be changed between different instances. Additionally, all of our actions are task-dependent (marked by the keyword `motivated`), unless stated otherwise.

We define seven low-level actions listed in Table 1. All low-level actions are prefixed with `a_` to differentiate them from high-level actions with the same name. The low-level actions include only the necessary arguments to specify the action, the variables necessary for the completion are included as constants in the action. For instance the action `a_drop` only specifies a person that drops something it carries. The new location of the carryable is defined as the placement area that is connected to the manipulation area where the cook is during this action (see Listing 1).

We define the action `a_move(Person,ManArea)` only with a target, to reduce the planning cost spent on it. The action behaves like a teleportation from the current area to the destination, but the duration can be set using `distance(ManArea,ManArea)`. If some duration is not defined, it is not possible to move between those locations. This duration can then be the cost of moving between these areas. We also consider two modifications of the action `a_move`. First, we can set the duration to a fixed amount, and second, we can make the action not task-dependent.

`a_move(Person,ManArea)`	`a_drop(Person,Carryable)`
`a_arrange(Person,Ingredient,Tableware)`	`a_pick_up(Person,Carryable)`
`a_chop(Cook,Choppable,Knife)`	`a_boil(Boilable,Pot)`
`a_fry(Fryable,Pan)`	

Table 1: Low-level actions in the domain *Overcooked*

```
constant PlArea pl;
constant ManArea man;
connected(pl, man);
[all] person.loc == man;
[all] carryable.loc == p :-> pl;
```

Listing 1: Constant variables in the action `a_drop`

The action `a_chop(Cook,Choppable,Knife)` requires that the cook does not carry anything while chopping and is marked as busy so they cannot chop multiple things at once. Similarly, tools are marked as processing during the actions `a_chop`,`a_boil` and `a_fry`. This is achieved using the change assertion `[all] sv == false :-> false`. A cook is not directly necessary for the actions `a_boil` and `a_fry`.

We define nine high-level totally-ordered actions listed in Table 2 that define all possible interactions required by an order. The two actions `m_get_to` and `m_fetch` require a specific cook or person, while the other actions abstract from a specific cook.

We use a common pattern in HTN problems, which is that all of the high-level actions have a first empty decomposition that only includes the actions post conditions. For example, the action `m_get_to` has as the first decomposition the persistence assertion that the person is at the target area (see Listing 2).

As we have previously defined that the actions `a_boil` and `a_fry` do not require a cook, we need to enforce that the processed ingredient is picked up before it becomes overcooked. This is done by adding a time constraint on the subtasks. For the action `m_fry`, this time constraint is `end(fetch) <= end(fry) + 30`, meaning that the ingredient must be picked up at most 30 time steps after the end of frying (see Listing 3). FAPE does not allow using functions here, so the time, until something is overcooked has to be fixed here. We use the value 30.

Lastly, we define the orders that have as subtasks the high-level actions that are needed to fulfill a specific order (see Table 3). The orders are all partially-ordered. While it is possible to define the orders in total order, that is not an interesting problem to solve under the condition that tasks are received dynamically. If all tasks are totally-ordered, it

`m_get_to(Person,ManArea)`	`m_fetch(Cook,Carryable)`
`m_transport_to(Carryable,PlArea)`	`m_chop(Choppable)`
`m_boil(Boilable)`	`m_fry(Fryable)`
`m_prepare_tableware(Tableware)`	`m_arrange(Ingredient,Tableware)`
`m_deliver(Tableware,Client)`	

Table 2: High-level actions in the domain *Overcooked*

```
:decomposition {
  person.loc == target;
};
```

Listing 2: Empty decomposition as default for action `m_get_to`

```
[all] contains {
  transport: m_transport_to(fryable,toolArea);
  fry: a_fry(fryable, pan);
  fetch: m_fetch(cook, fryable);
};

end(transport) <= start(fry);
end(fry) <= start(fetch);
end(fetch) <= end(fry) + 30;
```

Listing 3: Subtask constraints in the action `m_fry`

is not beneficial to use different cooks as the subtasks still need to be executed sequentially.

For instance, the order `order_lettuce_tomato_salad` must prepare one plate, one lettuce and one tomato must each be chopped and arranged on the plate, then the plate can be delivered to the client (see Listing 4). We enforce the constraints that before something can be arranged, the plate has to be prepared, lettuce and tomato have to be chopped before they are arranged, and finally, everything has to be arranged before the plate can be delivered.

4.2 Acting and Online Planning with a Stream of Tasks

The core element around which this thesis is built is that tasks may be received during the execution of a plan, so during the acting phase. This has been considered in RAE [Ghallab et al., 2016], but has not been implemented in the FAPE planner [Bit-Monnot, 2016]. We extend the theoretical actor model in FAPE (see Figure 2) with a task stream connecting the activity manager to other actors or users (see Figure 3). Technically, this interaction exists in FAPE already through the message $newgoal(g)$, but was not made explicit in the architecture.

The interactions between the components are implemented as in FAPE using message passing, with each component being a state machine. The observation database and the execution of skills are not considered in this work, as no interaction with a platform was introduced. For more details on that see Bit-Monnot [2016].

```
order_lettuce_salad(Client)
order_lettuce_tomato_salad(Client)
order_lettuce_tomato_cucumber_salad(Client)
order_lettuce_tomato_burger(Client)
```

Table 3: Orders in the domain *Overcooked* which are considered to be high-level actions.

```
action order_lettuce_tomato_salad(Client cl) {
  motivated;
  :decomposition {
    constant Lettuce l;
    constant Tomato to;
    constant Plate p;
    [all] contains {
      t_prep : m_prepare_tableware(p);
      t_chop_l : m_chop(l);
      t_arr_l : m_arrange(l, p);
      t_chop_to : m_chop(to);
      t_arr_to : m_arrange(to, p);
      t_del : m_deliver(p, cl);
    };
    end(t_prep) <= start(t_arr_l);
    end(t_prep) <= start(t_arr_to);
    end(t_chop_l) <= start(t_arr_l);
    end(t_chop_to) <= start(t_arr_to);
    end(t_arr_l) <= start(t_del);
    end(t_arr_to) <= start(t_del);
  };
};
```

Listing 4: The order `order_lettuce_tomato_salad`

4.2.1 Stream of Tasks

Each task that is managed by the activity manager is received through the stream of tasks. A task in this stream has an optional deadline $t_{deadline}$ and is received by the activity manager at a specific time $t_{receive}$. The task is a temporally qualified ground task $\langle [t_{receive}, t_{deadline}] \tau(r_1, \ldots, r_n) \rangle$. If no deadline is specified, it is $max(\mathbb{N})$. Specifically, the task is sent to the activity manager using the message $newtask(t)$.

4.2.2 Activity Manager

The activity manager is the core component of this model, managing all interactions between the other components. It is a state machine with the states IDLE, WAITING_FOR_PLAN and DISPATCHING. In the idle state, it is waiting for tasks to be submitted. This is also the initial state. It switches from IDLE to WAITING_FOR_PLAN when receiving a task. A planning request is sent to the planner component, and the activity manager waits until a plan has been found. Once a plan has been found, the state switches to DISPATCHING, where the plan is continuously dispatched to the available skills. When the plan has been fully executed, the state switches back to IDLE.

The messages that are available to the activity manager are as follows:

- Incoming:
  - $newtask(t)$: A new task that should be executed.
  - $newplan(\phi)$: A new plan is received from the planner.

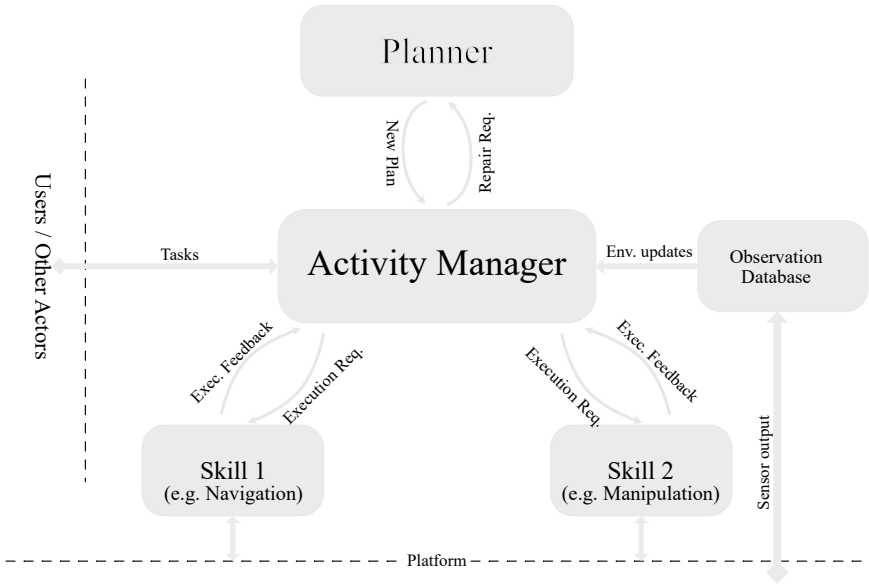


Figure 3: Architecture of FAPE with a Stream of Tasks

  - *noplan*: Planning failed.
  - *dispatchsuccess*(a) An action instance was successfully executed.

- Outgoing:
  - *planrequest*(ϕ): A request sent to the planner to search for a solution of ϕ.
  - *dispatch*(a): Dispatch an action instance a.

With these messages, the state machine of the activity manager is defined as follows. A visual representation of the state machine is shown in Figure 4.

- IDLE
  - receive *newtask*(t):
    1. integrate t into current plan ϕ
    2. send *planrequest*(ϕ)
    3. switch state to WAITING_FOR_PLAN
- WAITING_FOR_PLAN
  - receive *newplan*(ϕ):
    1. verify that ϕ corresponds to latest plan request sent
    2. switch state to DISPATCHING
  - receive *noplan*:
    1. shutdown
  - receive *newtask*(t):
    1. integrate t into current plan ϕ
    2. send *planrequest*(ϕ)

3. stay in state WAITING_FOR_PLAN

– receive *dispatchsuccess*(a):

1. mark a as executed
2. stay in state WAITING_FOR_PLAN

- DISPATCHING

1. for all actions a that can be executed
   (a) send *dispatch*(a)
   (b) mark a as executing
2. if all actions executed: switch to state IDLE

– receive *dispatchsuccess*(a):

1. mark a as executed
2. stay in state DISPATCHING

– receive *newtask*(t):

1. integrate t into current plan ϕ
2. send *planrequest*(ϕ)
3. switch state to WAITING_FOR_PLAN

The insertion of a new task $t = \langle [t_{receive}, t_{deadline}]\tau \rangle$ is handled by the plan modification Task Addition:

Definition 4.1 (Task Addition) *A chronicle $\phi_1 = (\pi_1, \mathcal{F}_1, \mathcal{C}_1)$ can be extended to a chronicle $\phi_2 = (\pi_2, \mathcal{F}_2, \mathcal{C}_2)$ by inserting a new temporally qualified task instance $t = \langle [t_{receive}, t_{deadline}]\tau \rangle$. This transformation is denoted $\phi_1 \xrightarrow{t}_I \phi_2$ and does the following modifications to ϕ_2*

$$\begin{aligned} \pi_2 &\leftarrow \pi_1 \cup \{\tau\} \\ \mathcal{F}_2 &\leftarrow \mathcal{F}_1 \cup \{t\} \\ \mathcal{C}_2 &\leftarrow \mathcal{C}_1 \end{aligned}$$

4.2.3 Planner Component

The planner component executes the planning algorithm, and on failure tries first repairing and then replanning. The planner uses the following messages:

- Incoming:
  – *planrequest*(ϕ): A request to search for a solution of ϕ.
- Outgoing:

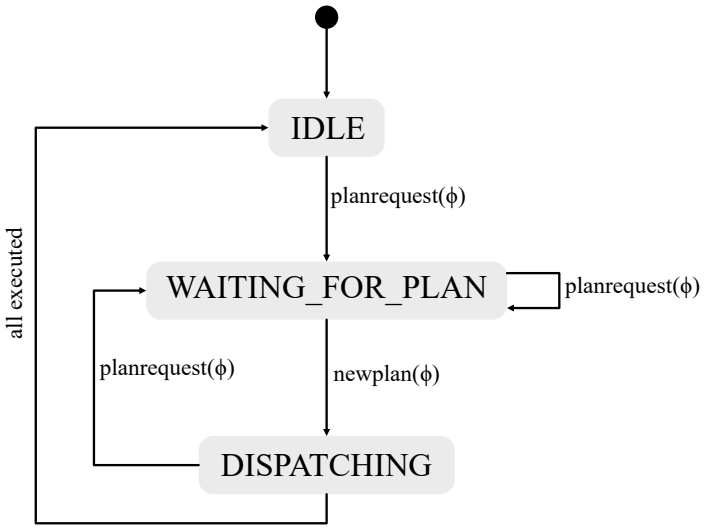


Figure 4: State machine diagram of the activity manager component

- *newplan*(ϕ): Send a plan that is a solution back to the activity manager.
- *noplan*: Indicate to the activity manager that no plan exists.

The planner has only two different states, IDLE and PLANNING. If it receives a new *planrequest*(ϕ), it initiates a search asynchronously and moves to the PLANNING state. In the PLANNING state it waits for the search to complete. If a search finishes successfully, the planner sends a *newplan*(ϕ) message. If the search fails, the planner issues a repair search and stays in the PLANNING state. If the repair search fails as well, the planner issues a replanning search and stays in the PLANNING state. If replanning fails as well, the planner sends a *noplan* message. If the planner receives a new *planrequest*(ϕ) in the PLANNING state, any running search is canceled and a new search for the new request is started asynchronously.

4.2.4 Skills Component

We do not implement skills in the context of this work but introduce the basics as a reference. A skill breaks down the abstract definition of an action into direct manipulations of the environment. As such, skills are inherently specific to the planning domain. In the *Overcooked* domain, this results in four basic manipulations, that can be executed in the game.

Movement: The action `a_move(Person p, ManArea to)` only describes a destination to go to and not the path to take. A shortest path algorithm should be used to find a path that the person can walk until it is at the destination. The path also needs to be translated into individual keypresses on W,A,S and D for the character's movement in the game.

Arranging, Picking Up and Dropping: All of these actions require the pressing of the space bar in the game.

Chopping: Chopping requires holding the CTRL key.

Boiling and Frying: These actions do not require any input. However, the cooking state should be monitored to avoid overcooking.

4.3 Robustness Heuristic

The robustness heuristic introduces a plan selection strategy that chooses plans where the addition of new tasks in the plan is more likely to succeed with a higher probability. Robustness is generally defined as follows.

Definition 4.2 (Robustness) *Robustness is the ability of a system to withstand stresses, pressures, perturbations, unpredictable changes, or variations in its operating environment without loss of functionality. A system that is designed to perform functionality in an expected environment is* robust *if it is able to maintain its functionality under a set of incidences. [Barber and Salido, 2015]*

As also mentioned in Barber and Salido [2015], the concrete "formalization depends on the system, on its expected functionality and on the specific set of incidences to be confronted". In considerations of robustness in planning [Lund et al., 2017], the incidences are perturbations in the start and end time points of actions. In a setting with static controllability, this leads to the most robust plans being ones that include as much margin between actions as possible.

In our case, controllability is mostly static, as we only replan on the arrival of new tasks. However, we allow delays for action time points, during planning or while executing. When adding new tasks during the execution, it is not beneficial to have large margins between the tasks. Therefore, the robustness with respect to adding new tasks is determined by how likely it is that the addition of a new task is possible.

Two cases must be considered: all decomposed actions of the added task will happen after the end of the current plan, or the decomposed actions be at least partially executed during the current plan. In the first case, the success of adding the new task is more likely when the current plan has a shorter makespan. In the latter case, the plan must be changed substantially. Some existing actions must be pushed back, while the deadline for those tasks must still be fulfilled. This works best when there are fewer dependencies between actions, and when the makespan is shorter.

We use perturbations directly as a metric for the robustness of a plan, similar to Wehner et al. [2023]. In our case, a perturbation is the delay of the start point of an action. While the end time point of an action may also be delayed, that is not possible due to the insertion of a new task, but due to a delay in the execution. Additionally, the end time point is automatically delayed when the start time point is delayed.

Definition 4.3 (Delay Perturbation) *A chronicle $\phi_1 = (\pi_1, \mathcal{F}_1, \mathcal{C}_1)$ can be changed into a chronicle $\phi_2 = (\pi_2, \mathcal{F}_2, \mathcal{C}_2)$ with a delay perturbation on an action $a \in \pi_1$ by an amount x denoted as $\phi_1 \xrightarrow{a,x}_P \phi_2$ by the addition of a temporal constraint as follows, where t^a_{start} is the starting time point of action a, t^p_{start} is the starting time point of the temporal HTN*

Planning Problem, and $mindelay(\phi_1, t^p_{start}, t^a_{start})$ *is the minimum delay between* t^p_{start} *and* t^a_{start} *in* ϕ_1*.*

$$\begin{aligned}\pi_2 &\leftarrow \pi_1\\ \mathcal{F}_2 &\leftarrow \mathcal{F}_1\\ \mathcal{C}_2 &\leftarrow \mathcal{C}_1 \cup \{t^p_{start} + (mindelay(\phi_1, t^p_{start}, t^a_{start}) + x) \leq t^a_{start}\}\end{aligned}$$

A new robustness heuristic is then introduced, that is used for node selection during planning. A heuristic in FAPE includes the two functions $g(\pi)$ and $h(\pi)$ as it uses the A* search algorithm for node selection. $g(\pi)$ describes the current cost and $h(\pi)$ describes the remaining cost of the plan. $h(\pi)$ can not be estimated for robustness and is therefore left at 0.

$$g(\phi) = \begin{cases} 0 & \text{if } inconsistent(\phi) \\ 1 + g(\phi \xrightarrow{a,x}_P \phi') & \text{otherwise} \end{cases} \tag{1}$$

$$h(\phi) = 0 \tag{2}$$

In Equation 1, a and x are chosen, such that the outcome is representative. As it is infeasible to calculate this representative outcome, a and x are chosen randomly in practice, with a being a randomly chosen action in π and x being a random variable using a half-normal distribution. The distribution variance is chosen as the minimum delay between the start of the action and the end of the plan. Additionally, x is at least 1, as else there would not be a perturbation. We introduce the notion of $\hat{g}(\phi, m)$ to capture the effect of m random delay perturbations.

$$a \sim \text{randomly choosen action} \in \pi \tag{3}$$

$$x \sim 1 + |\mathcal{N}(0, mindelay(\phi_1, t^a_{start}, t^p_{end}))| \tag{4}$$

$$\hat{g}(\phi, m) = \begin{cases} 0 & \text{if } inconsistent(\phi) \vee m = 0 \\ 1 + \hat{g}(\phi \xrightarrow{A,x}_P \phi', m-1) & \text{otherwise} \end{cases} \tag{5}$$

The arithmetic mean over a fixed amount of perturbations m and a fixed number of repetitions n is used as an approximation for $g(\phi)$. This heuristic can then be included during planning.

$$g(\phi) \approx \frac{1}{n} \sum_{i=1}^{n} \hat{g}(\phi, m) \tag{6}$$

4.4 Preparation Insertion

Preparing parts of a meal is a common strategy in restaurants to handle high loads. Due to the nature of the hierarchical plan, the essential subtasks for a task can be gathered.

Algorithm 3: PREPARATIONS: Generation of possible preparations

Input: $\phi' = \langle \pi, \mathcal{F}, \mathcal{C} \rangle$; $\mathcal{T}_p$
Output: T_p
$T_p \leftarrow \emptyset$;
foreach $\tau \in T_p$ **do**
  foreach *action template* A *for* τ **do**
    foreach $\tau_s \in subtasks(A)$ **do**
      if $parameters(\tau_s) \nsubseteq parameters(\tau)$ **then**
        foreach $t_p \in$ *temporally qualified ground instances of* $\tau_s, t_p \notin \pi$ **do**
          $T_d \leftarrow dependencies(A, t_p)$;
          if $T_d \neq \varnothing$ **then**
            if *all* T_d *executed* **then**
              $T_p \leftarrow T_p + \{t_p\}$
          else
            $T_p \leftarrow T_p + \{t_p\}$
return T_p

For instance, a salad requires that lettuce be chopped. Lettuce can be chopped when some cook has downtime in a plan. Then, when an order for a salad is received, the lettuce only needs to be put on a plate and served. However, the preparations should remain within the bounds of the current plan. Otherwise, an unlimited amount of preparations could be inserted, which would first block the activity manager from receiving any new tasks and second would correspond to a restaurant that prepares all of the available ingredients even when there are only a few orders.

The generation of potentially preparable tasks is shown in Algorithm 3. It receives a list of potential task symbols $\mathcal{T}_p$, in our case the potential orders that may be received through the stream of tasks. $\mathcal{T}_p$ may be known at the start but could be changed during the execution. For each $\tau_p \in \mathcal{T}_p$ the possible subtasks τ_s are gathered through all possible action templates. Subtasks that do not require any of the parameters of τ_p are independent of the individual order and thus potential subjects for preparation. We then generate all possible temporally qualified ground instances $\langle [t_{current}, mindelay(\phi, t^p_{start}, t^p_{end})] \tau_s(r_1, \ldots, r_n) \rangle$. A subtask may require that one or more previous tasks have been executed first. Those previous tasks have to be executed with the matching arguments before this task is subject to preparation. The parameters for the preparable tasks have to be ground and there cannot exist another task with the same parameters in π. Otherwise, a preparation could be added multiple times to a plan. It is however allowed that the same task is present multiple times in the result of the algorithm. Knowing how often a specific preparation is included can be helpful when choosing a preparation.

Example 4.1 *Suppose we have*

$$\mathcal{T}_p = \{\texttt{order\_lettuce\_tomato\_salad}\}$$
$$\pi = \{\texttt{m\_chop(lettuce1)}\}$$
and the typed ground instances of the problem
$$D_{Plate} = \{\texttt{plate1}\}, D_{Lettuce} = \{\texttt{lettuce1}\}, D_{Tomato} = \{\texttt{tomato1}\}$$

Algorithm 4: PrepPSP: Planning with preparation insertion

Input: $P = \langle \Sigma, \phi \rangle$; $\mathcal{T}_p$
Output: ϕ'
$\phi' \leftarrow PSP(P)$;
if $\phi' = FAIL$ **then**
  return *FAIL*
else
  $T_p \leftarrow PREPARATIONS(\phi', \mathcal{T}_p)$;
  nondeterministically choose $t_p \in T_p$;
  $\phi' \xrightarrow{t_p}_I \phi''$;
  $\phi'' \leftarrow PrepPSP(\langle \Sigma, \phi'' \rangle, \mathcal{T}_p)$;
  if $\phi'' = FAIL$ **then**
    return ϕ'
  else
    return ϕ''

The task `order_lettuce_tomato_salad` *has only one action template, corresponding to Listing 4. The ground instances that can be prepared are then*

$$\mathcal{T}_s = \{\texttt{m\_prepare\_tableware(plate1)}, \texttt{m\_chop(lettuce1)}, \texttt{m\_chop(tomato1)}, \texttt{m\_arrange(lettuce1)}, \texttt{m\_arrange(tomato1)}\}$$

Of these ground instances, `m_chop(lettuce1)` *cannot be prepared, as it is already included in* π *and* `m_arrange(tomato1)` *cannot be prepared, as the dependency* `m_chop(tomato1)` *is not in* π*. So the final list of preparable tasks is*

$$T_p = \{\texttt{m\_prepare\_tableware(plate1)}, \texttt{m\_chop(tomato1)}, \texttt{m\_arrange(lettuce1)}\}$$

To be able to use a preparation later, tasks need to have one possible decomposition that is fulfilled if all postconditions are met. Otherwise, the prepared instances could not be used in future submitted tasks due to the hierarchical decomposition.

Planning with preparations is done as depicted in Algorithm 4. When a plan has been found for a problem, possible preparations are generated. One preparation is nondeterministically chosen to be included in the current chronicle and added as a temporally qualified task. Planning is continued with the modified chronicle. This procedure is repeated recursively until no plan is found. Then the last valid plan is returned.

The preparation insertion is similar to action insertion. While action insertion is used to support goals or preconditions of tasks using generative planning instead of HTN planning, preparation insertion supports possible future tasks. The properties of preparation insertion vary significantly compared to task insertion.

We propose several concrete implementations of the non-deterministic *choose* method in algorithm 4. We call these implementations preparation selection strategies:

- A random selection can be used as an approximation. The random selection gives preparations that are present multiple times in T_p a higher likelihood.

- A greedy approach can be used to insert all possible preparations and then choose the best resulting plan.
- The time and resources required for a preparation task can be used to find the best combination of tasks to insert. The time and resource usage should be calculated as a static analysis before planning.
- Only the most common preparations are used. This prevents unnecessary further processing and preparations that are not necessary for most tasks are not prepared.
- A statistical or learning model of the environment can be used to predict the most likely tasks. Then the preparations can be chosen based on the likelihood of being necessary.
- The previous preparation insertions should be taken into account, to not over-provision one task and under-provision other tasks.
- Further metrics can be included based on the domain, like e.g. the storage life of ingredients after being prepared.

5 Experimental Evaluation

The acting system, the robustness heuristic and the task preparations were implemented directly inside the system FAPE [Bit-Monnot, 2016]. The statements that FAPE supports acting [Dvorák et al., 2014, Bit-Monnot, 2016, Bit-Monnot et al., 2020] do not hold. The acting implementation is not present in the current version of the system, and the original acting implementation was domain-dependent and only supported non-hierarchical problems. Additionally, the removal of actions from a chronicle is not implemented, which makes replanning and plan repair impossible. Lastly, when adding new tasks, they could not create actions during the current plan, as the timeline implementation does not allow splitting timelines to support a new timeline. Changes to a timeline can only be added before its first or after its last component. Fixing these issues proved to go beyond the scope of this thesis, however. The intuitive approach of regarding the planner as a black box and adapting the original problem to include already executed actions creates problems with the hierarchical representation and therefore makes planning impossible. Evaluation regarding the planning of tasks by interleaving with already planned tasks and replanning is thus done manually.

We evaluate our implementation in three aspects. First, we compare our hierarchical domain definition to the classical domain definition by Yuxin Liu et al. [2020] in Section 5.2. In Section 5.3 we evaluate our robustness heuristic in comparison to the default and makespan heuristic. Last, we manually evaluate the robustness heuristic and preparations in an acting environment in Section 5.4

The evaluation was done in an LXC container with 12 Intel(R) Xeon(R) Gold 6334 CPU 3.60GHz Cores, 32 GB RAM. Planning in FAPE is however a single-threaded process, so only one CPU core can be used. Each planning iteration was given 1000 seconds to complete.

We consider four different variants of our *Overcooked* domain definition:

- Fully hierarchical: All action templates are task-dependent and the duration of the action template `a_move` is fixed at 5.
- Fully hierarchical with durations: All action templates are task-dependent and the duration of the action template `a_move` is the shortest distance in the map calculated using the A* search algorithm.
- Partially hierarchical: The action template `a_move` is not task-dependent, so agents can move freely in the domain if necessary and the duration of the action template `a_move` is fixed at 5.
- Partially hierarchical with durations: The action template `a_move` is not task-dependent, so agents can move freely in the domain if necessary and the duration of the action template `a_move` is the shortest distance in the map calculated using the A* search algorithm.

Figure 5: The tutorial level of “Overcooked!2”

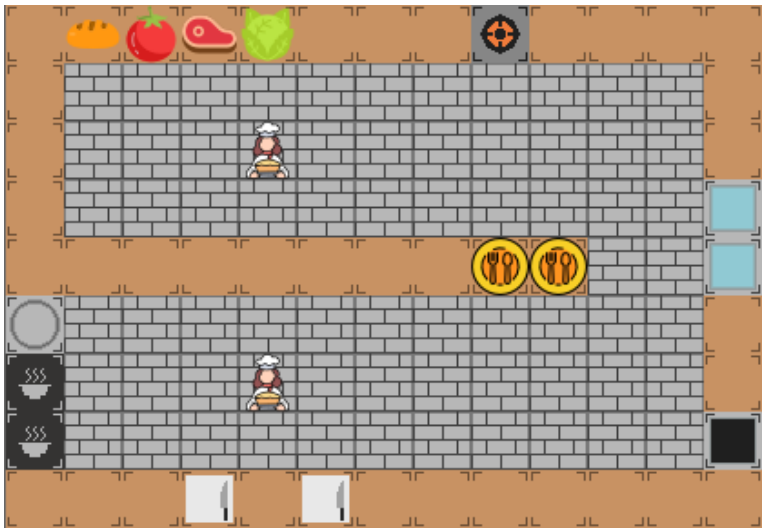


Figure 6: Complex map for making burgers from Yuxin Liu et al. [2020]

5.1 Environments

We consider two environments in this evaluation. The complete environment definition is included in Appendix C.

First, we consider the “tutorial level” of the “Overcooked!2” game shown in Figure 5. The environment provides the three ingredients lettuce, tomato and cucumber, the tableware plate and the tool knife. The orders that may be received are a simple salad, consisting of only lettuce, a lettuce tomato salad and lastly a salad with lettuce, tomato and cucumber. All of the ingredients have to be chopped using a knife before they are assembled on a plate. Each ingredient and tableware is present five times, there are four knives and two cooks available. The map has a narrow path to get to the locations of the knives. We define four possible arrange areas, two in the middle on each side and two on the bottom on each side. We chose this environment, as it contains all of the properties we consider and provides simple to complex tasks. Also, it is the first environment human players encounter when learning the game, so can serve as an example if the planning system is viable for this domain.

	success	search time	generated nodes	makespan
TFD	yes	$20s$	10546	125.1
Full hier.	yes	$31.0s \pm 0.9$	125 ± 0.0	232 ± 0.0
Full hier. + dur	yes	$37.9s \pm 1.6$	125 ± 0.0	368 ± 0.0
Part. hier.	yes	$563.0s \pm 45.1$	3754 ± 0.0	227 ± 0.0
Part. hier. + dur	yes	$628.0s \pm 16.9$	3744 ± 0.5	318 ± 0.0

Table 4: Results of TFD and our different domain variations for producing one burger

Second, we consider the "complex map" shown in Figure 6 taken from Yuxin Liu et al. [2020]. This environment is not from the official game but was introduced by them. It provides the ingredients beef, lettuce, tomato and burger bun, the tableware plate and the tools knife and pan. The orders received here are burgers with a burger bun, a steak, lettuce and tomato. The lettuce and tomato have to be chopped and the steak fried in the pan before they are assembled on a plate. Each ingredient and tableware is again present five times, there are two knives, two pans and two cooks available. The map has a long counter and cooks must walk around it to move between the ingredient locations and the tools, so the cooks are encouraged to pass ingredients over the counter. There are two arrange areas at the same place as seen in Figure 6. This environment provides a more complex recipe and an additional optimization goal with the transportation of ingredients over a counter.

5.2 HTN vs Classical Planning in *Overcooked*

In this section, we compare our HTN domain definition for *Overcooked* with the PDDL domain definition by Yuxin Liu et al. [2020]. It is not possible to directly compare the makespan between the domains, as not all of the action durations were provided. To plan using the HTN domain definition, we use FAPE [Bit-Monnot, 2016] and Yuxin Liu et al. [2020] use the Temporal Fast Downward (TFD) planner [Eyerich et al., 2009]. The environment is the "complex map" shown in Figure 6. The goal of this problem is to prepare either one or two burgers.

It is important to note that FAPE uses different heuristics by default during planning compared to TFD. FAPE uses the heuristic minimal spanning tree for partially hierarchical domains and depth-first search in fully hierarchical domains. While the selection of different heuristics is possible in FAPE, it is not considered in this comparison. Yuxin Liu et al. [2020] uses the shortest makespan heuristic. As such, FAPE only does satisficing while TFD tries to find the optimal plan.

Additionally, TFD uses SSP while FAPE uses PSP. This makes the generated nodes not directly comparable.

The results of planning the delivery of one burger are shown in Table 4. TFD is faster in the generation of the plan and visits significantly more nodes than FAPE. The search time for the partial hierarchy is also slower than the fully hierarchal version. The inclusion of the representative movement durations also increases the search time for both variants. The makespan is however shorter for the partial hierarchy compared to the full hierarchy, especially when using the representative movement durations.

	success	search time	generated nodes	makespan
TFD	no	1000s+	1000000+	-
Full hier.	no	1000s+	5360+	-
Full hier. + dur	no	1000s+	7430+	-
Part. hier.	no	1000s+	3800+	-
Part. hier. + dur	no	1000s+	3071+	-

Table 5: Results of TFD and our different domain variations for producing two burgers

The results for planning the delivery of two burgers are shown in Table 5. Both planners are unable to find a solution in this case. TFD does consider more search nodes than FAPE, as in the previous comparison.

5.3 Robustness Heuristic vs Default and Makespan Heuristics

In this section, we evaluate the robustness heuristic introduced in Section 4.3. Heuristics can be provided as a list in FAPE, to assign a priority. The additional heuristics are used to break ties between previous heuristics.

The default heuristics of FAPE are depth-first search ("dfs"), ordered decomposition ("ord-dec") and "soca" which is a simple comparison between the number of flaws and actions. We compare this default heuristic with four different combinations of adding either the shortest makespan or the robustness heuristic at index 2 or 3. The depth-first search should always be at index 1 for fully hierarchical problems, as the search would otherwise not finish in a reasonable timeframe. We choose $n = m = 10$ for the robustness heuristic as a tradeoff between accuracy and feasibility.

We compare the heuristic combinations in the following four different simple problems in the domain variant "Fully hierarchical with durations". This domain is chosen, as the previous evaluation showed that our partially hierarchical domain is not tractable and the durations are relevant for optimizations. The environment for the salads is the "tutorial level" (see Figure 5). For the burger it is the "complex map" (see Figure 6). The deadlines are necessary for the robustness heuristic, as it requires some point of failure. We choose the deadlines for each task such that all tasks can be completed by a single cook in sequence.

1. a lettuce salad, with a deadline of 150:
   ```
   [start, start+150] contains order_lettuce_salad(client1);
   ```
2. two lettuce salads, with a deadline of 300 each:
   ```
   [start, start+300] contains order_lettuce_salad(client1);
   [start, start+300] contains order_lettuce_salad(client2);
   ```
3. a tomato salad with a deadline of 200:
   ```
   [start, start+200] contains order_lettuce_tomato_salad(client1);
   ```
4. a burger with a deadline of 400:
   ```
   [start, start+400] contains order_lettuce_tomato_burger(client1);
   ```

	heuristic	success	plantime	generated nodes	makespan
salad	default	yes	$11.8s \pm 1.5$	46.0 ± 0.0	115.0 ± 0.0
	makespan@2	yes	$11.4s \pm 1.5$	53.0 ± 0.0	79.0 ± 0.0
	makespan@3	yes	$11.1s \pm 1.7$	53.0 ± 0.0	79.0 ± 0.0
	robustness@2	yes	$11.6s \pm 1.3$	48.6 ± 5.3	110.6 ± 25.5
	robustness@3	yes	$11.7s \pm 1.3$	45.8 ± 2.5	115 ± 21.2
two salads	default	no	$1000.0s+$	$39641.0\pm?$	–
	makespan@2	no	$1000.0s+$	38906.2 ± 1541.4	–
	makespan@3	no	$1000.0s+$	36023.0 ± 554.3	–
	robustness@2	no	$1000.0s+$	$16584.0\pm?$	–
	robustness@3	no	$1000.0s+$	$21336.0\pm?$	–
tomato salad	default	yes	$10.8s \pm 1.5$	75.0 ± 0.0	180.0 ± 0.0
	makespan@2	yes	$12.0s \pm 0.3$	119.0 ± 0.0	118.0 ± 0.0
	makespan@3	yes	$12.0s \pm 0.2$	115.0 ± 0.0	121.0 ± 0.0
	robustness@2	yes	$15.0s \pm 2.7$	171.4 ± 39.4	175.4 ± 2.2
	robustness@3	yes	$12.1s \pm 1.8$	79.2 ± 4.3	190.2 ± 3.9
burger	default	yes	$38.1s \pm 3.1$	125.0 ± 0.0	368.0 ± 0.0
	makespan@2	no	$1000.0s+$	5389.2 ± 448.1	–
	makespan@3	no	$1000.0s+$	5323.6 ± 234.7	–
	robustness@2	yes	$342.1s \pm 365.2$	2108.8 ± 2462.0	329.4 ± 21.29
	robustness@3[a]	40%	$28.6s \pm 0.8$	71 ± 7.0	333.0 ± 8.5

Table 6: Performance results for different heuristic combinations on different problems. The default heuristic is a priority combination with depth-first search ("dfs"), ordered decomposition ("ord-dec") and "soca" which is a simple comparison between the number of flaws and actions. The other heuristic combinations in the table are created by inserting them at the respective index (starting at 1) in the priority combination, e.g. makespan@2 would result in the list (dfs, makespan, ord-dec, soca).

[a]The values here are only for the case of success

The results are shown in table 6. Each heuristic variant was evaluated five times on each problem. The arithmetic mean value is shown first and the standard deviation second. When the standard deviation is ?, i.e. unknown, the calculation of the standard deviation returned nan.

Producing a single salad takes only around 10 seconds for all of the heuristics. The number of generated nodes is similar for all heuristics. The makespan heuristic combinations achieve a significant decrease in the makespan of the plan. The makespan of the robustness heuristics is not better than the default and has a high standard deviation.

None of the heuristics can provide a plan for two salads at the same time in under 1000 seconds. The default and makespan heuristics do however generate more nodes than the robustness heuristics.

All heuristics succeed at planning to prepare the tomato salad, the planning times are between 10 and 15 seconds, with the robustness@2 having the longest mean planning time at 15 seconds. The highest number of generated nodes has the robustness@2 heuristic, while robustness@3 and the default heuristic generate a similar number of nodes. The makespan values show a similar picture as when planning for the simple salad. The makespan heuris-

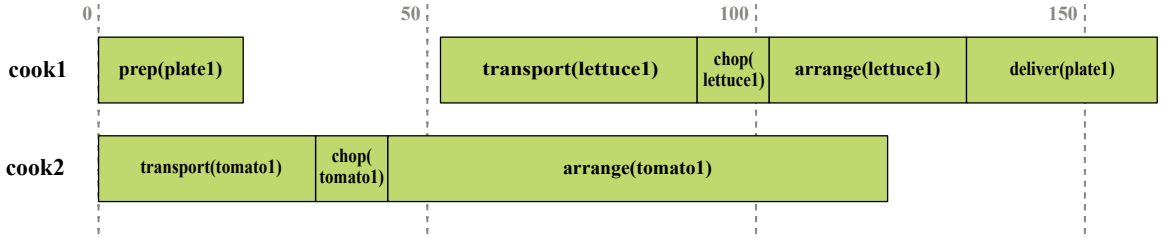


Figure 7: High-level actions divided by cook for the task `order_tomato_lettuce_salad` using the robustness@2 heuristic

tics have the shortest makespan value, and the makespan values of the robustness heuristic are not much different than the makespan value of the default heuristic.

The burger is only planned successfully by the default heuristic and the robustness@2 heuristic. The robustness@3 heuristic succeeds only two out of five times in the 1000 seconds and the makespan heuristics do not finish. The robustness@2 heuristic has however a very high standard deviation. The default heuristic generates the lowest number of nodes, while the mean for the robustness@2 heuristic is much higher, but also with a very high standard deviation. The highest number of nodes are generated by the makespan heuristics, but around seven times fewer nodes than in the two salads problem in the same time frame. The lowest makespan here is by the robustness@2 heuristic, but again with a high standard deviation.

We now consider one planning result of the robustness@2 for the problem of one tomato salad in detail. The total makespan of this solution is 161, which is the shortest result for this problem using the robustness@2 heuristic. The low-level actions that are performed are shown in the following code snippet. The overall timeline for high-level actions is shown for each cook in Figure 7.

This result shows that the tasks are split between the two cooks. One prepares the plate, chops and arranges the lettuce and delivers the salad, while the other one chops and arranges the tomato. Some of the high-level actions like the arranging of the tomato take very long. The low-level actions reveal that `cook2` remains at `manCounterMiddle2Top` between time 52 and 110, but only places the lettuce on the plate at time 110. Similarly, `cook1` stands waiting between time 22 and 52 before transporting the lettuce.

5.4 Acting with Robustness Heuristic or Preparation Insertion

In this section, we evaluate the impact of the robustness heuristic and the task preparations on acting. We use the environment "tutorial level" in the "Fully hierarchical with durations" domain with the following tasks:

- `[0,200] contains order_lettuce_tomato_salad(client1);`
- `[100,300] contains order_lettuce_tomato_salad(client2);`
- `[150,350] contains order_lettuce_tomato_salad(client3);`

Each task is received by the activity manager at the start of the temporal interval it is contained in. We chose the task `order_lettuce_tomato_salad`, as it is more complex

```
[0,2]     a_move(cook1, manPlateDispenser)
[0,10]    a_move(cook2, manTomatoDispenser)
[2,8]     a_pick_up(cook1, plate4)
[8,18]    a_move(cook1, manCounterMiddle2Top)
[10,16]   a_pick_up(cook2, tomato1)
[16,29]   a_move(cook2, manKnife3)
[18,22]   a_drop(cook1, plate4)
[29,33]   a_drop(cook2, tomato1)
[33,44]   a_chop(cook2, tomato1, knife3)
[45,51]   a_pick_up(cook2, tomato1)
[51,52]   a_move(cook2, manCounterMiddle2Top)
[52,65]   a_move(cook1, manLettuceDispenser)
[65,71]   a_pick_up(cook1, lettuce1)
[71,87]   a_move(cook1, manKnife4)
[87,91]   a_drop(cook1, lettuce1)
[91,102]  a_chop(cook1, lettuce1, knife4)
[103,109] a_pick_up(cook1, lettuce1)
[109,122] a_move(cook1, manCounterMiddle2Bottom)
[110,120] a_arrange(cook2, tomato1, plate4)
[122,122] a_arrange(cook1, lettuce1, plate4)
[132,142] a_move(cook1, manCounterMiddle2Top)
[142,148] a_pick_up(cook1, plate4)
[148,156] a_move(cook1, manDeliver)
[156,161] a_give(cook1, client1, plate4)
```

Listing 5: Low-Level actions for the task `order_tomato_lettuce_salad` using the robustness@2 heuristic

than the simple salad and provides enough opportunity for parallelization. The burger could give a more representative result, but simulating it by hand is not feasible. The temporal intervals were chosen such that a solution exists, but it is not possible to have each cook fulfill an order sequentially on their own.

We consider four different configurations:

- default: no additional changes to FAPE
- makespan: the makespan@2 heuristic is used to get the shortest plan for each received task.
- preparations: the optimal preparations are inserted into the default plan.
- robustness: the robustness heuristic is used for generating a plan for each task

Acting is simulated using an oracle approach: we manually add the optimal sequence of actions for the second and third tasks consecutively without replanning or repair. This measures the ability of the configurations to find a plan without replanning or repair. Since representative action durations are used, the durations for each action may be different. We therefore use the optimal durations that are calculated with the makespan@2 heuristic for the first task. These durations are:

- $dur(prep) = 20$

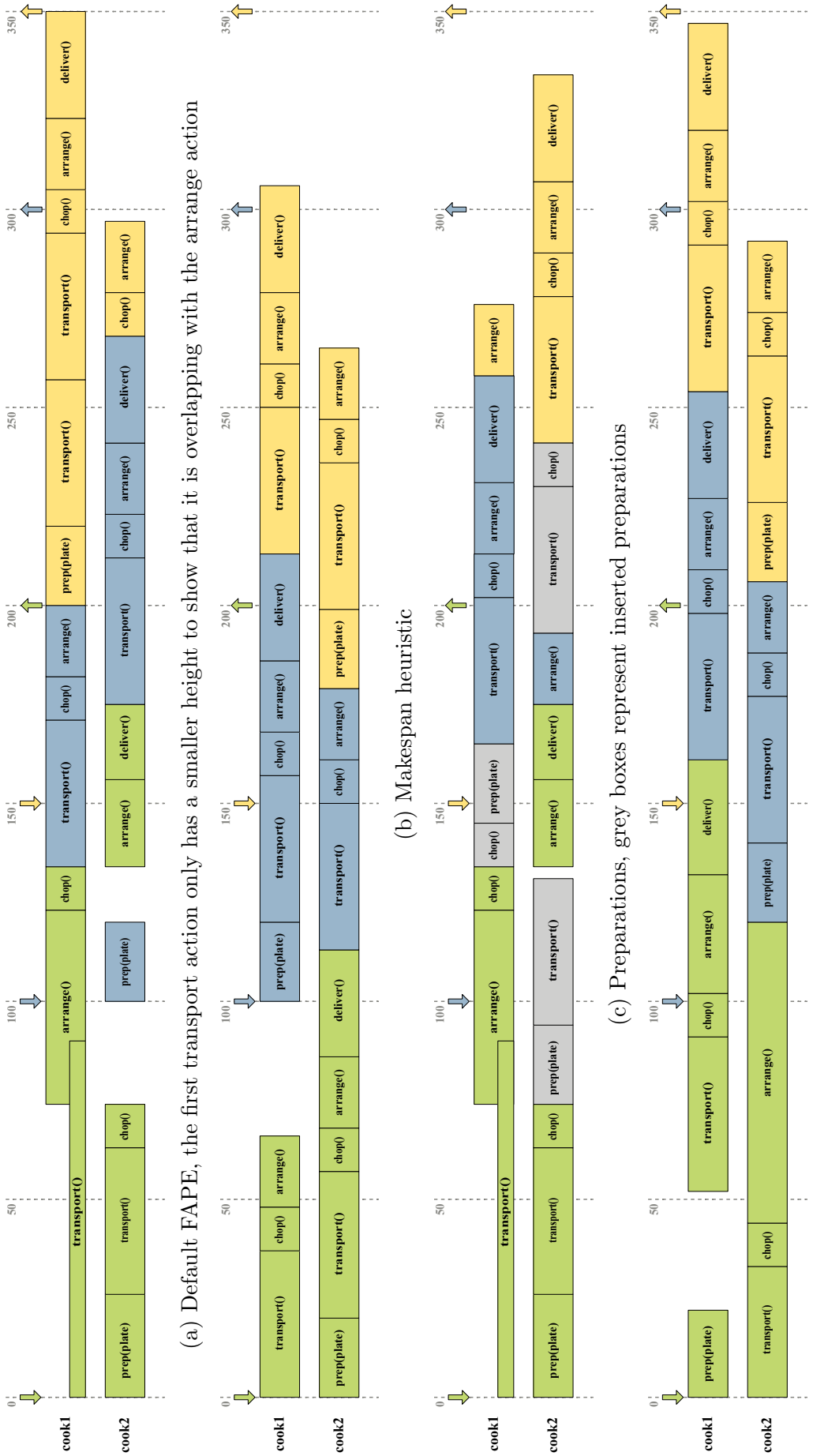
	success	margin			avg margin	total makespan
		task 1	task 2	task 3		
default	yes	25	32	0	19 ± 16.8	350
makespan	yes	87	87	44	72.7 ± 24.8	306
preparations	yes	25	42	16	27.7 ± 13.2	323
robustness	yes	39	46	3	29.3 ± 23.1	347

Table 7: Results of the manual acting evaluation

- $dur(transport) = 37$
- $dur(chop) = 11$
- $dur(arrange) = 18$
- $dur(deliver) = 27$

The results are shown in Table 7 and a visual representation is shown in Figure 8. The task `m_chop` was split into the two subtasks `transport` and `a_chop`, as they can be handled by different cooks.

All of the configurations succeed in this manual evaluation with optimal task addition. The makespan configuration represents the optimal configuration when not allowing preparations. It has the highest margin for each task and the highest average margin. It also has the shortest overall makespan. The preparations have the second shortest overall makespan. This configuration has lower margins for each task but a smaller standard deviation for the average margin. The robustness configuration has a higher margin than the preparations for the first and second tasks but is much lower for task 3. Therefore the average is a little higher, but the standard deviation is almost doubled. The total makespan is also only three time steps lower than failure. The default configuration has the lowest margins and reaches the deadline for task 3 just on time.



(a) Default FAPE, the first transport action only has a smaller height to show that it is overlapping with the arrange action

(b) Makespan heuristic

(c) Preparations, grey boxes represent inserted preparations

(d) Robustness heuristic

Figure 8: Timeline-like representations of the plan executed by acting. Each row shows the actions performed by one of the cooks. Each color represents a task, the downpointing arrows represent the start time of a task and the uppointing arrows the deadline.

6 Discussion

In this section, we will discuss the results of the evaluation and thus the validity of our approach. Additionally, we will discuss the replan and repair approach, as that could not be evaluated. Lastly, we go into some of the issues that we encountered with the system FAPE concerning acting.

In comparison with the classical domain by Yuxin Liu et al. [2020], our domain definition in HTN is significantly slower, especially when optimizing for the makespan. However, this comparison is not completely representative for several reasons. First, the action durations of the PDDL domain were not given. Second, the exact environment and goal they used are not available, so they might not have used as many ingredient instances as we have or only planned a burger with lettuce. Additionally, our domain definition does not allow chopping a steak into a burger patty before frying it, due to the limitation of single inheritance in FAPE.

The *Overcooked* domain we define in 4.1, is also not a complete representation of the game "Overcooked!". For instance, we do not define collisions between cooks in our domain and allow placing multiple ingredients in the same place. Next, we also do not implement the washing of dishes after clients have finished their meal, or fire breaking out when some ingredient gets overcooked. Another crucial mechanic in the game is throwing ingredients. As such, many more problems need to be solved in the *Overcooked* domain before a planning actor can play the game.

The robustness heuristic we introduced does not perform as well as we had proposed. While it succeeds at parallelizing tasks, it has arbitrary waiting times included. This problem likely comes from the fact that many options for variable bindings are kept in place for too long. During planning, the timelines are created incrementally. When threats occur between timelines, the timelines are separated. When the variable bindings are then resolved to be ground, these separations are kept in place and cannot be changed anymore. Through the robustness heuristic, some of the variable bindings are resolved sooner, but not soon enough. This is also the reason why cooks often just wait for several time steps for no apparent reason.

Additionally, the heuristic results in a high variance for the planning time and the makespan. This variance is most likely related to the low value of 10 we chose for n and m. A more thorough evaluation could reveal the best values for these parameters. However, we already see a decrease in the number of nodes generated by over 50% in the same planning time. This high variance and increase in heuristic evaluation makes this implementation of a robustness heuristic not feasible for real-world problems.

The informative value of the acting evaluation is very low, as it was done manually and only on a single example. In our example, we observe that the default approach can only succeed in adding the other tasks if it finds the optimal plan as there is no margin to the deadline of the third task. The approach with the robustness heuristic also has only a very small margin for the last task and is therefore also not likely to succeed. The preparations approach on the other hand has higher and more stable margins compared to the robustness heuristic. We can observe that at least four preparations can be inserted. Here we further added preparations after the second task has been inserted. Overall, both approaches are far from the optimal plan without preparations. We have used the

makespan heuristic for the optimal plan, as that can find the shortest plan possible.

As we have again used an oracle approach for the task preparations, it can be observed that the selection of preparations is very important. In this case, we have inserted two preparations of a plate, one preparation of a tomato and one preparation of a lettuce. This is exactly what is needed for future tasks, but depending on the preparation selection strategy, it could also be the case that up to four preparations of plates were inserted. As these plates are not required, too much time would be spent on preparing the plates so that the third or even the second task could not be finished anymore.

With plan repair or replanning, we may be able to resolve some of the issues that arise with initially nonoptimal plans. As the implementation of these components was out of scope for this work and they are not implemented in FAPE, we can not evaluate it. We could for instance not find a solution with the default approach for the addition of task 2, when the deadline is 260 instead of 300. With replanning, we could remove at least the arrange and deliver tasks to create more usable space for cook2. On the other hand, it could also be enough to delay those two tasks to have cook2 perform a transport for task 3 before delivering the salad for task 2. Additionally, plan repair and replanning may be required for preparations, if for example too many task preparations were added to the plan but not yet executed. These could be removed to find a solution for the new task first, and then again add some preparations.

The time required for planning is another problem in a dynamic environment with deadlines such as *Overcooked*. When the planner requires – as in our evaluations – 10 seconds for a simple task and over 30 seconds for a more complex task, this valuable time is now missing to complete it before the deadline. We can argue that FAPE is not an optimized system, and could be much faster if it was implemented in a low-level language like C or Rust, or used the best CSP solvers. However, the fact remains that the performance does not scale well as HTN planning is an undecidable problem. As such it may be beneficial to consider an approach similar to online planning, where the plan is refined when the task has to be executed in the coming time. FAPE does propose a similar approach, where the exact domain is unclear until a robot has used some kind of environment perception, which then modifies the environment or adds respective task decompositions.

We selected FAPE, as it promised to fulfill all requirements regarding the problem at hand, specifically temporal HTN planning and acting. As the acronym FAPE means "flexible acting and planning engine", this suggests that it does support acting natively. However, this is not true anymore, as they have removed most of the code regarding acting as it was domain-dependent. Therefore most of the acting architecture had to be rebuilt while keeping it domain-independent. Plan repair was not implemented and only replanning was done in FAPE. This replanning also only consisted of re-executing the planning process with updated goals. This procedure ignores the context of HTN, where some actions have been executed already, and cannot be changed later. As such, the removal of a task from a chronicle was not implemented. While they describe that problem as theoretically simple, it is complex in implementation, as variables cannot be removed from constraint satisfaction problems easily and references to the variables may consist in many different places. Additionally, constraints resolved in an STN will remain there and cannot be changed trivially.

Another issue of the current implementation of FAPE is, that the timelines were too rigid. When a new task is added to an existing chronicle, the timelines only allow the

addition of new changes after the first or before the last change. This operation may however be necessary such as in the case of Figure 8a, where the subtask `prep(plate2)` of the second task has to be added before the subtask `arrange(lettuce1)` of task 1. With these restrictions, it was therefore not possible to implement our goal of interleaving plans appropriately.

The issues regarding the timelines and replanning could be resolved by defining executed, executing and pending actions including their hierarchy in the problem, instead of directly changing the implementation. However, as we have already stated, adding this current plan as a problem in FAPE could not be achieved naively.

Bit-Monnot [2016] argues that qualitative time points in planning have the only benefit of creating instantaneous action effects. We disagree with this notion and propose that representing the temporal constraint problem using a Qualitative Constraint Network (QCN) [Ligozat, 2013] with the Interval Algebra [Allen, 1983] may have several benefits. First, the representation as a QCN allows a more intuitive definition of the constraints compared to time points. Next, a QCN leaves options for different concrete instantiations open, similar to the CSP for variable bindings. The search does not have to resolve these unknowns until necessary, therefore reducing the search space. Lastly, explicit durations and deadlines can still be used by enforcing additional constraints on the QCN. On a side note, there already exists an approach for evaluating the robustness of a QCN [Wehner et al., 2023] that could be used as a replacement for our implementation of the robustness heuristic.

7 Conclusion

In this work, we have presented a temporal HTN domain definition for *Overcooked*. This domain represents a challenging environment for planning, as it requires tasks that are partially ordered. We find that FAPE cannot plan two dish orders at once, similar to the observation for classical planning in the same domain [Yuxin Liu et al., 2020].

Next, we have introduced an actor model that is an extension of FAPEs actor model [Bit-Monnot, 2016]. This extension enables the explicit handling of a stream of tasks. We introduced two new strategies to anticipate the receiving of new tasks.

First, we defined a robustness heuristic as a plan selection strategy. This heuristic is supposed to incentivize plans that have a shorter makespan and make the addition and interleaving of new tasks more successful. We find that it has a small impact on the load distribution between cooks in the *Overcooked* domain. It is also able to find solutions when the makespan heuristic is not. We do however observe a high variance in the planning time for complex problems.

Second, we define the concept of preparation insertion in HTN planning. Due to the definition of subtasks in HTN planning, we can find subtasks that can be prepared. We find that the insertion of preparations likely results in more successful executions compared to not using them. We also find that the success of this preparation insertion relies very much on the preparation selection strategy

We identify several points for further research in this context:

- Our domain definition for *Overcooked* can be extended to include more of the concepts in the game such as collisions, fires breaking out or washing of used dishes. Eventually, we believe that this domain can serve as a relevant benchmark for the combination of planning and acting.
- Due to missing features in the FAPE planner, the implementation of the actor framework is not fully working and was therefore not evaluated. This has to be further investigated, for instance by using a different planner or adding the missing features, in particular timeline splitting and action removal or the definition of existing actions in the planning problem, to FAPE.
- While replanning in HTN planning has already been evaluated, there do not exist any studies yet that focus on plan repair in HTN planning. These two strategies should be further evaluated to find if the statements from classical planning apply to HTN planning.

Lastly, we have found that resources on acting are most of the time either theoretical or domain-dependent. We think that this may be connected to the fact that the IPC only focuses on different planning variants, while most of the other competitions in this context are domain specific competitions and challenges that require a complete system with domain-dependent implementations. We suggest that it may be time for a middle ground with acting benchmarks and competitions that incentivize domain-independent systems.

8 Acknowledgements

First of all, I want to thank Prof. Dr. Diedrich Wolter, my professor, for the opportunity to work on this topic for my thesis. He helped me with the brainstorming of possible approaches and referred me to different topics of interest.

Further thanks go to my fellow students, friends, and family for the ongoing support during the time of writing and coding. Special thanks to Tobias Schwartz and Sonja Ruschhaupt for proofreading and feedback on the thesis.

References

James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983. ISSN 0001-0782, 1557-7317. doi: 10.1145/182.358434. URL https://dl.acm.org/doi/10.1145/182.358434.

Mohannad Babli, Óscar Sapena, and Eva Onaindia. Plan commitment: Replanning versus plan repair. *Engineering Applications of Artificial Intelligence*, 123:106275, 2023. URL https://www.sciencedirect.com/science/article/pii/S0952197623004591.

Josef Bajada, Maria Fox, and Derek Long. Temporal Plan Quality Improvement and Repair using Local Search. In *STAIRS*, pages 41–50, 2014. URL https://books.google.com/books?hl=de&lr=&id=e93YBAAAQBAJ&oi=fnd&pg=PA41&dq=Temporal+Plan+Quality+Improvement+and+Repair+using+Local+Search&ots=PqS1AdfluE&sig=l2Q2mVk-kGvOL51kcsJuiUlm6Jg.

Yash Bansod, Dana Nau, Sunandita Patra, and Mak Roberts. Integrating Planning and Acting With a Re-Entrant HTN Planner. In *ICAPS Workshop on Hierarchical Planning (HPlan)*, 2021. URL http://www.cs.umd.edu/~nau/papers/bansod2021integrating.pdf.

Yash Bansod, Sunandita Patra, Dana Nau, and Mark Roberts. HTN Replanning from the Middle. *The International FLAIRS Conference Proceedings*, 35, May 2022. ISSN 2334-0762. doi: 10.32473/flairs.v35i.130732. URL https://journals.flvc.org/FLAIRS/article/view/130732.

Federico Barber and Miguel A. Salido. Robustness, stability, recoverability, and reliability in constraint satisfaction problems. *Knowledge and Information Systems*, 44(3):719–734, September 2015. ISSN 0219-3116. doi: 10.1007/s10115-014-0778-3. URL https://doi.org/10.1007/s10115-014-0778-3.

Gerardo Berbeglia, Jean-François Cordeau, and Gilbert Laporte. Dynamic pickup and delivery problems. *European Journal of Operational Research*, 202(1):8–15, April 2010. ISSN 03772217. doi: 10.1016/j.ejor.2009.04.024. URL https://linkinghub.elsevier.com/retrieve/pii/S0377221709002999.

Pascal Bercher, Ron Alford, and Daniel Höller. A Survey on Hierarchical Planning – One Abstract Idea, Many Concrete Realizations. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pages 6267–6275, Macao, China, August 2019. International Joint Conferences on Artificial Intelligence Organization. ISBN 978-0-9992411-4-1. doi: 10.24963/ijcai.2019/875. URL https://www.ijcai.org/proceedings/2019/875.

Dimitris Bertsimas, Allison Chang, Velibor V. Mišić, and Nishanth Mundru. The Airlift Planning Problem. *Transportation Science*, 53(3):773–795, May 2019. ISSN 0041-1655, 1526-5447. doi: 10.1287/trsc.2018.0847. URL http://pubsonline.informs.org/doi/10.1287/trsc.2018.0847.

Justin Bishop, Jaylen Burgess, Cooper Ramos, Jade B. Driggs, Tom Williams, Chad C. Tossell, Elizabeth Phillips, Tyler H. Shaw, and Ewart J. de Visser. CHAOPT: A Testbed for Evaluating Human-Autonomy Team Collaboration Using the Video

Game Overcooked!2. In *2020 Systems and Information Engineering Design Symposium (SIEDS)*, pages 1–6, April 2020. doi: 10.1109/SIEDS49339.2020.9106686. URL https://ieeexplore.ieee.org/abstract/document/9106686.

Arthur Bit-Monnot. *Temporal and Hierarchical Models for Planning and Acting in Robotics*. PhD thesis, Institut National Polytechnique de Toulouse-INPT, 2016. URL https://theses.hal.science/tel-04261612/.

Arthur Bit-Monnot and Roland Godet. Aries. LAAS-CNRS, January 2024. URL https://github.com/plaans/aries.

Arthur Bit-Monnot, Malik Ghallab, Félix Ingrand, and David E Smith. FAPE: A Constraint-based Planner for Generative and Hierarchical Temporal Planning, October 2020. URL https://arxiv.org/abs/2010.13121.

Guido Boella and Rossana Damiano. A Replanning Algorithm for a Reactive Agent Architecture. In G. Goos, J. Hartmanis, J. Van Leeuwen, and Donia Scott, editors, *Artificial Intelligence: Methodology, Systems, and Applications*, volume 2443, pages 183–192. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-44127-4 978-3-540-46148-7. doi: 10.1007/3-540-46148-5_19. URL http://link.springer.com/10.1007/3-540-46148-5_19.

Micah Carroll, Rohin Shah, Mark K Ho, Tom Griffiths, Sanjit Seshia, Pieter Abbeel, and Anca Dragan. On the Utility of Learning about Humans for Human-AI Coordination. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/hash/f5b1b89d98b7286673128a5fb112cb9a-Abstract.html.

Rujikorn Charakorn, Poramate Manoonpong, and Nat Dilokthanakul. Investigating Partner Diversification Methods in Cooperative Multi-agent Deep Reinforcement Learning. In Haiqin Yang, Kitsuchart Pasupa, Andrew Chi-Sing Leung, James T. Kwok, Jonathan H. Chan, and Irwin King, editors, *Neural Information Processing*, volume 1333, pages 395–402. Springer International Publishing, Cham, 2020. ISBN 978-3-030-63822-1 978-3-030-63823-8. doi: 10.1007/978-3-030-63823-8_46. URL https://link.springer.com/10.1007/978-3-030-63823-8_46.

Filip Dvorák, Roman Barták, Arthur Bit-Monnot, Félix Ingrand, and Malik Ghallab. Planning and Acting with Temporal and Hierarchical Decomposition Models. In *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*, pages 115–121, November 2014. doi: 10.1109/ICTAI.2014.27.

Kutluhan Erol, James Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. In *AAAI*, volume 94, pages 1123–1128, 1994. URL https://cdn.aaai.org/AAAI/1994/AAAI94-173.pdf.

Patrick Eyerich, Robert Mattmüller, and Gabriele Röger. Using the Context-enhanced Additive Heuristic for Temporal and Numeric Planning. *ICAPS*, 2009.

Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971. URL https://www.sciencedirect.com/science/article/pii/0004370271900105.

Matthew C. Fontaine, Ya-Chuan Hsu, Yulun Zhang, Bryon Tjanaka, and Stefanos Nikolaidis. On the Importance of Environments in Human-Robot Coordination, June 2021. URL http://arxiv.org/abs/2106.10853.

M. Fox and D. Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20:61–124, December 2003. ISSN 1076-9757. doi: 10.1613/jair.1129. URL https://jair.org/index.php/jair/article/view/10352.

Maria Fox, Alfonso Gerevini, Derek Long, and Ivan Serina. Plan Stability: Replanning versus Plan Repair. In *ICAPS*, volume 6, pages 212–221, 2006. URL https://cdn.aaai.org/ICAPS/2006/ICAPS06-022.pdf.

Luca Framba, Valentini Allessandro, Arthur Bit-Monnot, Alessandro Trapasso, Andrea Micheli, and Alberto Rovetta. Unified Planning. AIPlan4EU, February 2024. URL https://github.com/aiplan4eu/unified-planning.

Thomas Geier and Pascal Bercher. On the decidability of HTN planning with task insertion. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1955, 2011. URL http://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.090/Publikationen/2011/Geier11HybridDecidability.pdf.

Ilche Georgievski and Marco Aiello. HTN planning: Overview, comparison, and beyond. *Artificial Intelligence*, 222:124–156, May 2015. ISSN 0004-3702. doi: 10.1016/j.artint.2015.02.002. URL https://www.sciencedirect.com/science/article/pii/S0004370215000247.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Elsevier, May 2004. ISBN 978-0-08-049051-9.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 1 edition, July 2016. ISBN 978-1-107-03727-4 978-1-139-58392-3. doi: 10.1017/CBO9781139583923. URL https://www.cambridge.org/core/product/identifier/9781139583923/type/book.

Daniel Höller, Pascal Bercher, Gregor Behnke, and Susanne Biundo. On Guiding Search in HTN Planning with Classical Planning Heuristics. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pages 6171–6175, Macao, China, August 2019. International Joint Conferences on Artificial Intelligence Organization. ISBN 978-0-9992411-4-1. doi: 10.24963/ijcai.2019/857. URL https://www.ijcai.org/proceedings/2019/857.

Daniel Höller, Gregor Behnke, Pascal Bercher, Susanne Biundo, Humbert Fiorino, Damien Pellier, and Ron Alford. HDDL: An Extension to PDDL for Expressing Hierarchical Planning Problems. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34 (06):9883–9891, April 2020. ISSN 2374-3468. doi: 10.1609/aaai.v34i06.6542. URL https://ojs.aaai.org/index.php/AAAI/article/view/6542.

Félix Ingrand and Malik Ghallab. Deliberation for autonomous robots: A survey. *Artificial Intelligence*, 247:10–44, 2017. URL https://www.sciencedirect.com/science/article/pii/S0004370214001350.

Paul Knott, Micah Carroll, Sam Devlin, Kamil Ciosek, Katja Hofmann, A. D. Dragan, and Rohin Shah. Evaluating the Robustness of Collaborative Agents, January 2021. URL http://arxiv.org/abs/2101.05507.

Gérard Ligozat. *Qualitative Spatial and Temporal Reasoning*. John Wiley & Sons, 2013. URL https://books.google.com/books?hl=de&lr=&id=euFTs8EkK_cC&oi=fnd&pg=PA8&dq=Qualitative+Spatial+and+Temporal+Reasoning&ots=puL9nRnOvQ&sig=KtpYYanaoszYjtkND9TLx6RF8_o.

Kyle Lund, Sam Dietrich, Scott Chow, and James Boerkoel. Robust Execution of Probabilistic Temporal Plans. *Proceedings of the AAAI Conference on Artificial Intelligence*, 31(1), February 2017. ISSN 2374-3468. doi: 10.1609/aaai.v31i1.11019. URL https://ojs.aaai.org/index.php/AAAI/article/view/11019.

Sharada Mohanty, Erik Nygren, Florian Laurent, Manuel Schneider, Christian Scheller, Nilabha Bhattacharya, Jeremy Watson, Adrian Egli, Christian Eichenberger, Christian Baumberger, Gereon Vienken, Irene Sturm, Guillaume Sartoretti, and Giacomo Spigler. Flatland-RL : Multi-Agent Reinforcement Learning on Trains, December 2020. URL http://arxiv.org/abs/2012.05893.

Patrick Nalepka, Jordan Gregory-Dunsmore, James Simpson, Gaurav Patil, and Michael Richardson. Interaction Flexibility in Artificial Agents Teaming with Humans. July 2021.

Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence-Volume 2*, pages 968–973, 1999. URL https://dl.acm.org/doi/abs/10.5555/1624312.1624357.

Christos H. Papadimitriou. Computational complexity. In *Encyclopedia of Computer Science*, pages 260–265. 2003. URL https://dl.acm.org/doi/abs/10.5555/1074100.1074233.

Sunandita Patra, James Mason, Amit Kumar, Malik Ghallab, Paolo Traverso, and Dana Nau. Integrating acting, planning, and learning in hierarchical operational models. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 30, pages 478–487, 2020.

Damien Pellier, Alexandre Albore, Humbert Fiorino, and Rafael Bailon-Ruiz. HDDL 2.1: Towards Defining a Formalism and a Semantics for Temporal HTN Planning. In *Proceedings of the 6th ICAPS Workshop on Hierarchical Planning (HPlan 2023)*, pages 49–53, 2023. URL https://icaps23.icaps-conference.org/papers/hplan/HPlan2023_paper_4.pdf.

João G. Ribeiro, Cassandro Martinho, Alberto Sardinha, and Francisco S. Melo. Assisting Unknown Teammates in Unknown Tasks: Ad Hoc Teamwork under Partial Observability, January 2022. URL http://arxiv.org/abs/2201.03538.

Andres Rosero, Faustina Dinh, Ewart J. de Visser, Tyler Shaw, and Elizabeth Phillips. Two Many Cooks: Understanding Dynamic Human-Agent Team Communication and Perception Using Overcooked 2, October 2021. URL http://arxiv.org/abs/2110.03071.

Constantin Ruhdorfer. Into the minds of the chefs : Using Theory of Mind for robust collaboration with humans in Overcooked. Master's thesis, University of Stuttgart, 2023. URL http://elib.uni-stuttgart.de/handle/11682/13983.

Bidipta Sarkar, Aditi Talati, Andy Shih, and Dorsa Sadigh. Pantheonrl: A marl library for dynamic training interactions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 13221–13223, 2022. URL https://ojs.aaai.org/index.php/AAAI/article/view/21734.

David E Smith, Jeremy Frank, and William Cushing. The ANML Language. In *International Conference on Automated Planning and Scheduling*, 2008.

Sebastian Stock. Hierarchical Hybrid Planning for Mobile Robots. *KI - Künstliche Intelligenz*, 31(4):373–376, November 2017. ISSN 1610-1987. doi: 10.1007/s13218-017-0507-7. URL https://doi.org/10.1007/s13218-017-0507-7.

Sebastian Stock, Masoumeh Mansouri, Federico Pecora, and Joachim Hertzberg. Hierarchical Hybrid Planning in a Mobile Service Robot. In Steffen Hölldobler, Rafael Peñaloza, and Sebastian Rudolph, editors, *KI 2015: Advances in Artificial Intelligence*, volume 9324, pages 309–315. Springer International Publishing, Cham, 2015a. ISBN 978-3-319-24488-4 978-3-319-24489-1. doi: 10.1007/978-3-319-24489-1_28. URL http://link.springer.com/10.1007/978-3-319-24489-1_28.

Sebastian Stock, Masoumeh Mansouri, Federico Pecora, and Joachim Hertzberg. Online task merging with a hierarchical hybrid task planner for mobile service robots. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6459–6464, September 2015b. doi: 10.1109/IROS.2015.7354300.

Ayal Taitler, Ron Alford, Joan Espasa, Gregor Behnke, Daniel Fišer, Michael Gimelfarb, Florian Pommerening, Scott Sanner, Enrico Scala, Dominik Schreiber, Javier Segovia-Aguas, and Jendrik Seipp. The 2023 International Planning Competition. *AI Magazine*, page aaai.12169, April 2024. ISSN 0738-4602, 2371-9621. doi: 10.1002/aaai.12169. URL https://onlinelibrary.wiley.com/doi/10.1002/aaai.12169.

Jérémy Turi and Arthur Bit-Monnot. Guidance of a Refinement-based Acting Engine with a Hierarchical Temporal Planner. In *ICAPS Workshop on Integrated Planning, Acting, and Execution (IntEx)*, Singapore (virtual), Singapore, June 2022. URL https://hal.science/hal-03690039.

Roman Van Der Krogt and Mathijs De Weerdt. Plan Repair as an Extension of Planning. In *ICAPS*, volume 5, pages 161–170, 2005. URL https://cdn.aaai.org/ICAPS/2005/ICAPS05-017.pdf.

Jan Wehner, Michael Sioutis, and Diedrich Wolter. On robust vs fast solving of qualitative constraints. *Journal of Heuristics*, 29(4):461–485, December 2023. ISSN 1572-9397. doi: 10.1007/s10732-023-09517-8. URL https://doi.org/10.1007/s10732-023-09517-8.

Yuxin Liu, Qiguang Chen, Ke Jin, and Zhanhao Zhang. Planning for Overcooked Game with PDDL. *International Core Journal of Engineering*, 6(12), December 2020. ISSN 2414-1895. doi: 10.6919/ICJE.202012_6(12).0047.

Rui Zhao, Jinming Song, Yufeng Yuan, Haifeng Hu, Yang Gao, Yi Wu, Zhongqian Sun, and Wei Yang. Maximum entropy population-based training for zero-shot human-ai coordination. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 6145–6153, 2023. URL `https://ojs.aaai.org/index.php/AAAI/article/view/25758`.

Ich erkläre hiermit gemäß §17 Abs. 2 APO, dass ich die vorstehende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Bamberg, den 11.04.2024 Felix Haase

A Supplementary Material

If you have received this thesis in printed form, you will find a CD with supplementary material at the end. If you have received this thesis in a digital form, you can access the supplementary material in the git repositories github.com/sargreal/fape (changes to FAPE) and github.com/sargreal/ai-thesis-m (thesis, problem files and evaluation).

The contents of the CD are as follows:

- thesis:
  - problem:
    - anml: anml problem files.
    - hddl: initial implementation of the *Overcooked* domain in HDDL2.1
  - report: contains all latex files and figures used to write this thesis
  - evaluation: evaluation files in csv format and an jupyter notebook that was used for calculating mean and standard deviations
  - scripts: several scripts that were used to automate tasks such as planning, acting and visualization of solutions
  - solutions: several raw solutions and visualizations to the planning and acting problems
  - fape-planning-assembly-1.0.jar: the exact version of fape that was used for the evaluations
- fape: our adapted version of fape including `changes.diff` that is a git diff file describing all changes we made to the planner

B *Overcooked* Domain Definition

This domain definition is only a partial implementation of the game, focusing on the most important parts.

Listing 6: The *Overcooked* domain

```
/*** Types ***/

// Area definitions adapted from Race domain
type Area;
type ManArea < Area; // area from which a person manipulates
type OccupationArea < Area; // area where an ingrediant can be
    located
type StorageArea < OccupationArea; // area where objects are
   stored initially
type PlArea < OccupationArea;  // area to place objects
type ArrangeArea < PlArea; // area to arrange objects
type ToolArea < PlArea; // area to work with tools
type NavArea < Area; // other area where a person can be
```

```
type Person with {
  fluent (NavArea or ManArea) loc;
  // If the Person is carrying anything
  fluent boolean carrying;
  // If the Person is doing something
  fluent boolean busy;
};

type Cook < Person;
type Client < Person;

type Tool with {
  constant PlArea loc;
  fluent boolean processing;
};

type Knife < Tool;
type Pot < Tool;
type Pan < Tool;

type Carryable with {
  // Can either be at an placement area, carried by a Person
     or on a Tableware (Tablewares can be stacked)
  fluent (OccupationArea or Person or Tableware) loc;
};

type Tableware < Carryable;

type Plate < Tableware;

type Ingredient < Carryable with {
  fluent boolean arranged;
  fluent boolean prepared;
};

type Nori < Ingredient;
type BurgerBun < Ingredient;

type Choppable < Ingredient with {
  fluent boolean chopped;
};

type Shrimp < Choppable;
type Fish < Choppable;
type Lettuce < Choppable;
type Tomato < Choppable;
type Cucumber < Choppable;

type Boilable < Ingredient with {
  fluent boolean boiled;
};

type Rice < Boilable;

type Fryable < Ingredient with {
  fluent boolean fryed;
```

```
};

type Beef < Fryable;

/*** Constants ***/

constant boolean connected(OccupationArea a, ManArea b);
constant integer distance(ManArea a, ManArea b);

// Constants for durations of actions
constant integer arrangetime(Ingredient i);
constant integer droptime(Carryable c);
constant integer pickuptime(Carryable c);
constant integer givetime(Carryable c);
constant integer choptime(Choppable c);
constant integer boiltime(Boilable b);
constant integer frytime(Fryable f);

/*** Primitive actions ***/

action a_move(Person p, ManArea to) {
  // uncomment to make the action insertable
  motivated;
  constant ManArea from;
  // Set the duration to a constant value to simplify the
     model
  duration := distance(from, to);
  from != to;
  [all] {
    p.loc == from :-> to;
  };
};

action a_arrange(Person p, Ingredient i, Tableware t) {
  motivated;
  duration := arrangetime(i);

  constant ArrangeArea pl;
  constant ManArea man;
  connected(pl, man);

  [all] {
    t.loc == pl;
    p.loc == man;
    i.loc == p :-> t;
    p.carrying == true :-> false;
    i.arranged == false :-> true;
  };
};

action a_drop(Person p, Carryable ca) {
  motivated;
  duration := droptime(ca);

  constant PlArea pl;
  constant ManArea man;
```

```
  connected(pl, man);

  [all] {
    p.loc == man;
    ca.loc == p :-> pl;
    p.carrying == true :-> false;
  };
};

action a_pick_up(Person p, Carryable ca) {
  motivated;
  duration := pickuptime(ca);

  constant OccupationArea oc;
  constant ManArea man;
  connected(oc, man);

  [all] {
    p.loc == man;
    ca.loc == oc :-> p;
    p.carrying == false :-> true;
  };
};

action a_give(Person p1, Person p2, Carryable c) {
  motivated;
  duration := givetime(c);

  constant ManArea m;
  [all] {
    p1.loc == m;
    p2.loc == m;
    p1.carrying == true :-> false;
    p2.carrying == false :-> true;
    c.loc == p1 :-> p2;
  };
};

action a_chop(Cook co, Choppable ch, Knife k) {
  motivated;
  duration := choptime(ch);

  constant ManArea man;
  constant ToolArea ta;
  connected(ta, man);
  k.loc == ta;

  [all] {
    co.loc == man;
    ch.loc == ta;
    k.processing == false :-> false;
    co.busy == false :-> false;
    co.carrying == false :-> false;
    ch.chopped == false :-> true;
  };
};
```

```
action a_boil(Boilable b, Pot p) {
  motivated;
  duration := boiltime(b);

  constant ManArea man;
  connected(p.loc, man);

  [all] {
    b.loc == p.loc;
    p.processing == false :-> false;
    b.boiled == false :-> true;
  };
};

action a_fry(Fryable f, Pan p) {
  motivated;
  duration := frytime(f);

  constant ManArea man;
  connected(p.loc, man);

  [all] {
    f.loc == p.loc;
    p.processing == false :-> false;
    f.fryed == false :-> true;
  };
};

/*** Methods ***/

action m_get_to(Person p, ManArea to) {
  motivated;
  // Already there
  :decomposition {
    p.loc == to;
  };
  :decomposition {
    [all] contains ordered(
      a_move(p, to)
    );
  };
};

action m_fetch(Cook co, Carryable ca) {
  motivated;
  // Cook already has the ingredient
  :decomposition {
    ca.loc == co;
  };
  // Go to carryables location and get it
  :decomposition {
    constant OccupationArea pl;
    constant ManArea m;
    connected(pl, m);
    [start] ca.loc == pl;
```

```
    [all] contains ordered(
      m_get_to(co, m),
      a_pick_up(co, ca)
    );
  };
};

action m_transport_to(Carryable ca, PlArea pl) {
  motivated;
  // Already there
  :decomposition {
    ca.loc == pl;
  };
  // Fetch, get to target man area and then drop
  :decomposition {
    constant Cook co;
    constant ManArea m;
    connected(pl, m);
    [all] contains ordered(
      m_fetch(co, ca),
      m_get_to(co, m),
      a_drop(co, ca)
    );
  };
  // Let two cooks transport the carryable over a shared
     counter
  :decomposition {
    constant Cook co1;
    constant Cook co2;
    constant PlArea counter;
    constant ManArea mCook1;
    constant ManArea mCook2;
    constant ManArea mFinal;
    connected(pl, mFinal);
    connected(counter, mCook1);
    connected(counter, mCook2);
    mCook1 != mCook2;
    pl != counter;
    co1 != co2;
    [all] contains ordered(
      m_fetch(co1, ca),
      m_get_to(co1, mCook1),
      a_drop(co1, ca),
      m_get_to(co2, mCook2),
      a_pick_up(co2, ca),
      m_get_to(co2, pl),
      a_drop(co2, ca)
    );
  };
};

action m_chop(Choppable ch) {
  motivated;
  :decomposition {
    ch.chopped == true;
  };
```

```
  :decomposition {

    constant Knife k;
    constant Cook co;
    constant ToolArea p;
    k.loc == p;

    [all] contains ordered(
      m_transport_to(ch, p),
      a_chop(co, ch, k)
    );
  };
};

action m_boil(Boilable b) {
  motivated;
  :decomposition {
    b.boiled == true;
  };
  :decomposition {

    constant Pot p;
    constant Cook c;
    constant ToolArea ta;
    p.loc == ta;

    [all] contains {
      transp: m_transport_to(b,ta);
      boil: a_boil(b, p);
      fetch: m_fetch(c, b);
    };
    end(transp) <= start(boil);
    end(boil) <= start(fetch);
    // The cook has to fetch the ingredient before the
       boilable gets overcooked
    end(fetch) <= end(boil) + 30;
  };
};

action m_fry(Fryable f) {
  motivated;
  :decomposition {
    f.fryed == true;
  };
  :decomposition {
    constant Pan p;
    constant Cook c;
    constant ToolArea ta;
    p.loc == ta;

    [all] contains {
      transp: m_transport_to(f,ta);
      fry: a_fry(f, p);
      fetch: m_fetch(c, f);
    };
    end(transp) <= start(fry);
```

```
    end(fry) <= start(fetch);
    // The cook has to fetch the ingredient before the fryable
        gets overcooked
    end(fetch) <= end(fry) + 30;
  };
};

action m_prepare_tableware(Tableware t) {
  motivated;
  :decomposition {
    constant ArrangeArea a;
    t.loc == a;
  };
  :decomposition {
    constant StorageArea s;
    constant ArrangeArea a;
    [start] t.loc == s;
    [all] contains ordered(
      m_transport_to(t, a)
    );
  };
};

action m_arrange(Ingredient i, Tableware t) {
  motivated;
  // The ingredient is already on the tableware
  :decomposition {
    i.loc == t;
  };
  // The ingredient has to be added to the tableware
  :decomposition {
    constant ArrangeArea a;
    constant ManArea m;
    constant Cook c;
    connected(a, m);
    [all] t.loc == a;

    [all] contains ordered(
      m_fetch(c, i),
      m_get_to(c, m),
      a_arrange(c, i, t)
    );
  };
};

action m_deliver(Tableware t, Client cl) {
  motivated;
  // The plate is already delivered
  :decomposition {
    t.loc == cl;
  };
  // The plate has to be delivered to the client
  :decomposition {
    constant Cook co;
    constant Area a;
    [all] cl.loc == a;
```

```
    [all] contains ordered(
      m_fetch(co, t),
      m_get_to(co, a),
      a_give(co, cl, t)
    );
  };
};

/*** Orders ***/

action order_lettuce_salad(Client cl) {
  motivated;
  :decomposition {
    constant Lettuce l;
    constant Plate t;
    [all] contains ordered (
      unordered(
        m_prepare_tableware(t),
        ordered(
          m_chop(l),
          m_arrange(l, t)
        )
      ),
      m_deliver(t, cl)
    );
  };
};

action order_lettuce_tomato_salad(Client cl) {
  motivated;
  :decomposition {
    constant Lettuce l;
    constant Tomato to;
    constant Plate t;
    [all] contains {
      t_prep : m_prepare_tableware(t);
      t_chop_l : m_chop(l);
      t_arr_l : m_arrange(l, t);
      t_chop_to : m_chop(to);
      t_arr_to : m_arrange(to, t);
      t_del : m_deliver(t, cl);
    };
    end(t_prep) <= start(t_arr_l);
    end(t_prep) <= start(t_arr_to);
    end(t_chop_l) <= start(t_arr_l);
    end(t_chop_to) <= start(t_arr_to);
    end(t_arr_l) <= start(t_del);
    end(t_arr_to) <= start(t_del);
  };
};

action order_lettuce_tomato_cucumber_salad(Client cl) {
  motivated;
  :decomposition {
    constant Lettuce l;
    constant Tomato to;
```

```
    constant Cucumber cu;
    constant Plate t;
    [all] contains {
      t_prep : m_prepare_tableware(t);
      t_chop_l : m_chop(l);
      t_arr_l : m_arrange(l, t);
      t_chop_to : m_chop(to);
      t_arr_to : m_arrange(to, t);
      t_chop_cu : m_chop(cu);
      t_arr_cu : m_arrange(cu, t);
      t_del : m_deliver(t, cl);
    };
    end(t_prep) <= start(t_arr_l);
    end(t_prep) <= start(t_arr_cu);
    end(t_prep) <= start(t_arr_to);
    end(t_chop_l) <= start(t_arr_l);
    end(t_chop_to) <= start(t_arr_to);
    end(t_chop_cu) <= start(t_arr_cu);
    end(t_arr_l) <= start(t_del);
    end(t_arr_to) <= start(t_del);
    end(t_arr_cu) <= start(t_del);
  };
};

action order_lettuce_tomato_burger(Client cl) {
  motivated;
  :decomposition {
    constant BurgerBun bb;
    constant Lettuce l;
    constant Tomato to;
    constant Beef b;
    constant Plate t;
    [all] contains {
      t_prep : m_prepare_tableware(t);
      t_arr_bb : m_arrange(bb, t);
      t_chop_l : m_chop(l);
      t_arr_l : m_arrange(l, t);
      t_chop_to : m_chop(to);
      t_arr_to : m_arrange(to, t);
      t_fry_b : m_fry(b);
      t_arr_b : m_arrange(b, t);
      t_del : m_deliver(t, cl);
    };
    end(t_prep) <= start(t_arr_bb);
    end(t_prep) <= start(t_arr_l);
    end(t_prep) <= start(t_arr_to);
    end(t_prep) <= start(t_arr_b);
    end(t_arr_bb) <= start(t_arr_l);
    end(t_arr_bb) <= start(t_arr_to);
    end(t_arr_bb) <= start(t_arr_b);
    end(t_chop_l) <= start(t_arr_l);
    end(t_chop_to) <= start(t_arr_to);
    end(t_fry_b) <= start(t_arr_b);
    end(t_arr_l) <= start(t_del);
    end(t_arr_to) <= start(t_del);
    end(t_arr_b) <= start(t_del);
```

```
  };
};
```

C *Overcooked* Environment Definitions

C.1 Tutorial Level

Listing 7: The *Overcooked* environment "tutorial level"

```
/** Tutorial Environment **/

forall(Ingredient i) {arrangetime(i) := 10; };
forall(Carryable c) {droptime(c) := 4; };
forall(Carryable c) {pickuptime(c) := 6; };
forall(Carryable c) {givetime(c) := 5; };
forall(Choppable c) {choptime(c) := 11; };
forall(Boilable b) {boiltime(l) := 30; };
forall(Fryable f) {frytime(f) := 30; };

instance Cook cook1,cook2;
instance Client client1,client2,client3,client4,client5;

instance Knife knife1,knife2,knife3,knife4;
instance Lettuce lettuce1,lettuce2,lettuce3,lettuce4,lettuce5;
instance Tomato tomato1,tomato2,tomato3,tomato4,tomato5;
instance Cucumber cucumber1,cucumber2,cucumber3,cucumber4,
   cucumber5;
instance Plate plate1,plate2,plate3,plate4,plate5;


instance ManArea manTomatoDispenser,manLettuceDispenser,
   manCucumberDispenser,manDeliver,manCounterLeftBottom,
   manCounterRightBottom,manCounterMiddle1Bottom,
   manCounterMiddle1Top,manCounterMiddle2Bottom,
   manCounterMiddle2Top,manPlateDispenser,manKnife1,manKnife2,
   manKnife3,manKnife4;
instance ToolArea taKnife1,taKnife2,taKnife3,taKnife4;
instance ArrangeArea counterLeftBottom,counterRightBottom,
   counterMiddle1,counterMiddle2;
instance StorageArea lettuceDispenser,tomatoDispenser,
   cucumberDispenser,plateDispenser;

// Manipulation Areas <-> Placement Areas
connected(counterMiddle1,manCounterMiddle1Top) := true;
connected(counterMiddle1,manCounterMiddle1Bottom) := true;
connected(counterMiddle2,manCounterMiddle2Top) := true;
connected(counterMiddle2,manCounterMiddle2Bottom) := true;
connected(counterLeftBottom,manCounterLeftBottom) := true;
connected(counterRightBottom,manCounterRightBottom) := true;

connected(lettuceDispenser,manLettuceDispenser) := true;
connected(tomatoDispenser,manTomatoDispenser) := true;
connected(cucumberDispenser,manCucumberDispenser) := true;
```

```
connected(plateDispenser,manPlateDispenser) := true;

connected(taKnife1,manKnife1) := true;
connected(taKnife2,manKnife2) := true;
connected(taKnife3,manKnife3) := true;
connected(taKnife4,manKnife4) := true;

// Distances
distance(manLettuceDispenser,manTomatoDispenser) := 1;
distance(manLettuceDispenser,manPlateDispenser) := 3;
distance(manLettuceDispenser,manDeliver) := 5;
distance(manLettuceDispenser,manCucumberDispenser) := 9;
distance(manLettuceDispenser,manKnife1) := 14;
distance(manLettuceDispenser,manKnife2) := 16;
distance(manLettuceDispenser,manKnife3) := 14;
distance(manLettuceDispenser,manKnife4) := 16;
distance(manLettuceDispenser,manCounterLeftBottom) := 2;
distance(manLettuceDispenser,manCounterRightBottom) := 8;
distance(manLettuceDispenser,manCounterMiddle1Bottom) := 5;
distance(manLettuceDispenser,manCounterMiddle1Top) := 11;
distance(manLettuceDispenser,manCounterMiddle2Bottom) := 11;
distance(manLettuceDispenser,manCounterMiddle2Top) := 13;
distance(manTomatoDispenser,manLettuceDispenser) := 1;
distance(manTomatoDispenser,manPlateDispenser) := 2;
distance(manTomatoDispenser,manDeliver) := 4;
distance(manTomatoDispenser,manCucumberDispenser) := 8;
distance(manTomatoDispenser,manKnife1) := 13;
distance(manTomatoDispenser,manKnife2) := 15;
distance(manTomatoDispenser,manKnife3) := 13;
distance(manTomatoDispenser,manKnife4) := 15;
distance(manTomatoDispenser,manCounterLeftBottom) := 1;
distance(manTomatoDispenser,manCounterRightBottom) := 7;
distance(manTomatoDispenser,manCounterMiddle1Bottom) := 4;
distance(manTomatoDispenser,manCounterMiddle1Top) := 10;
distance(manTomatoDispenser,manCounterMiddle2Bottom) := 10;
distance(manTomatoDispenser,manCounterMiddle2Top) := 12;
distance(manPlateDispenser,manLettuceDispenser) := 3;
distance(manPlateDispenser,manTomatoDispenser) := 2;
distance(manPlateDispenser,manDeliver) := 2;
distance(manPlateDispenser,manCucumberDispenser) := 6;
distance(manPlateDispenser,manKnife1) := 11;
distance(manPlateDispenser,manKnife2) := 13;
distance(manPlateDispenser,manKnife3) := 11;
distance(manPlateDispenser,manKnife4) := 13;
distance(manPlateDispenser,manCounterLeftBottom) := 1;
distance(manPlateDispenser,manCounterRightBottom) := 5;
distance(manPlateDispenser,manCounterMiddle1Bottom) := 2;
distance(manPlateDispenser,manCounterMiddle1Top) := 8;
distance(manPlateDispenser,manCounterMiddle2Bottom) := 8;
distance(manPlateDispenser,manCounterMiddle2Top) := 10;
distance(manDeliver,manLettuceDispenser) := 5;
distance(manDeliver,manTomatoDispenser) := 4;
distance(manDeliver,manPlateDispenser) := 2;
distance(manDeliver,manCucumberDispenser) := 4;
distance(manDeliver,manKnife1) := 9;
distance(manDeliver,manKnife2) := 11;
```

```
distance(manDeliver,manKnife3) := 9;
distance(manDeliver,manKnife4) := 11;
distance(manDeliver,manCounterLeftBottom) := 3;
distance(manDeliver,manCounterRightBottom) := 3;
distance(manDeliver,manCounterMiddle1Bottom) := 4;
distance(manDeliver,manCounterMiddle1Top) := 6;
distance(manDeliver,manCounterMiddle2Bottom) := 6;
distance(manDeliver,manCounterMiddle2Top) := 8;
distance(manCucumberDispenser,manLettuceDispenser) := 9;
distance(manCucumberDispenser,manTomatoDispenser) := 8;
distance(manCucumberDispenser,manPlateDispenser) := 6;
distance(manCucumberDispenser,manDeliver) := 4;
distance(manCucumberDispenser,manKnife1) := 13;
distance(manCucumberDispenser,manKnife2) := 15;
distance(manCucumberDispenser,manKnife3) := 13;
distance(manCucumberDispenser,manKnife4) := 15;
distance(manCucumberDispenser,manCounterLeftBottom) := 7;
distance(manCucumberDispenser,manCounterRightBottom) := 1;
distance(manCucumberDispenser,manCounterMiddle1Bottom) := 8;
distance(manCucumberDispenser,manCounterMiddle1Top) := 10;
distance(manCucumberDispenser,manCounterMiddle2Bottom) := 2;
distance(manCucumberDispenser,manCounterMiddle2Top) := 12;
distance(manKnife1,manLettuceDispenser) := 14;
distance(manKnife1,manTomatoDispenser) := 13;
distance(manKnife1,manPlateDispenser) := 11;
distance(manKnife1,manDeliver) := 9;
distance(manKnife1,manCucumberDispenser) := 13;
distance(manKnife1,manKnife2) := 2;
distance(manKnife1,manKnife3) := 10;
distance(manKnife1,manKnife4) := 12;
distance(manKnife1,manCounterLeftBottom) := 12;
distance(manKnife1,manCounterRightBottom) := 12;
distance(manKnife1,manCounterMiddle1Bottom) := 9;
distance(manKnife1,manCounterMiddle1Top) := 3;
distance(manKnife1,manCounterMiddle2Bottom) := 11;
distance(manKnife1,manCounterMiddle2Top) := 9;
distance(manKnife2,manLettuceDispenser) := 16;
distance(manKnife2,manTomatoDispenser) := 15;
distance(manKnife2,manPlateDispenser) := 13;
distance(manKnife2,manDeliver) := 11;
distance(manKnife2,manCucumberDispenser) := 15;
distance(manKnife2,manKnife1) := 2;
distance(manKnife2,manKnife3) := 12;
distance(manKnife2,manKnife4) := 10;
distance(manKnife2,manCounterLeftBottom) := 14;
distance(manKnife2,manCounterRightBottom) := 14;
distance(manKnife2,manCounterMiddle1Bottom) := 11;
distance(manKnife2,manCounterMiddle1Top) := 5;
distance(manKnife2,manCounterMiddle2Bottom) := 13;
distance(manKnife2,manCounterMiddle2Top) := 11;
distance(manKnife3,manLettuceDispenser) := 14;
distance(manKnife3,manTomatoDispenser) := 13;
distance(manKnife3,manPlateDispenser) := 11;
distance(manKnife3,manDeliver) := 9;
distance(manKnife3,manCucumberDispenser) := 13;
distance(manKnife3,manKnife1) := 10;
```

```
distance(manKnife3,manKnife2) := 12;
distance(manKnife3,manKnife4) := 2;
distance(manKnife3,manCounterLeftBottom) := 12;
distance(manKnife3,manCounterRightBottom) := 12;
distance(manKnife3,manCounterMiddle1Bottom) := 9;
distance(manKnife3,manCounterMiddle1Top) := 7;
distance(manKnife3,manCounterMiddle2Bottom) := 11;
distance(manKnife3,manCounterMiddle2Top) := 1;
distance(manKnife4,manLettuceDispenser) := 16;
distance(manKnife4,manTomatoDispenser) := 15;
distance(manKnife4,manPlateDispenser) := 13;
distance(manKnife4,manDeliver) := 11;
distance(manKnife4,manCucumberDispenser) := 15;
distance(manKnife4,manKnife1) := 12;
distance(manKnife4,manKnife2) := 10;
distance(manKnife4,manKnife3) := 2;
distance(manKnife4,manCounterLeftBottom) := 14;
distance(manKnife4,manCounterRightBottom) := 14;
distance(manKnife4,manCounterMiddle1Bottom) := 11;
distance(manKnife4,manCounterMiddle1Top) := 9;
distance(manKnife4,manCounterMiddle2Bottom) := 13;
distance(manKnife4,manCounterMiddle2Top) := 3;
distance(manCounterLeftBottom,manLettuceDispenser) := 2;
distance(manCounterLeftBottom,manTomatoDispenser) := 1;
distance(manCounterLeftBottom,manPlateDispenser) := 1;
distance(manCounterLeftBottom,manDeliver) := 3;
distance(manCounterLeftBottom,manCucumberDispenser) := 7;
distance(manCounterLeftBottom,manKnife1) := 12;
distance(manCounterLeftBottom,manKnife2) := 14;
distance(manCounterLeftBottom,manKnife3) := 12;
distance(manCounterLeftBottom,manKnife4) := 14;
distance(manCounterLeftBottom,manCounterRightBottom) := 6;
distance(manCounterLeftBottom,manCounterMiddle1Bottom) := 3;
distance(manCounterLeftBottom,manCounterMiddle1Top) := 9;
distance(manCounterLeftBottom,manCounterMiddle2Bottom) := 9;
distance(manCounterLeftBottom,manCounterMiddle2Top) := 11;
distance(manCounterRightBottom,manLettuceDispenser) := 8;
distance(manCounterRightBottom,manTomatoDispenser) := 7;
distance(manCounterRightBottom,manPlateDispenser) := 5;
distance(manCounterRightBottom,manDeliver) := 3;
distance(manCounterRightBottom,manCucumberDispenser) := 1;
distance(manCounterRightBottom,manKnife1) := 12;
distance(manCounterRightBottom,manKnife2) := 14;
distance(manCounterRightBottom,manKnife3) := 12;
distance(manCounterRightBottom,manKnife4) := 14;
distance(manCounterRightBottom,manCounterLeftBottom) := 6;
distance(manCounterRightBottom,manCounterMiddle1Bottom) := 7;
distance(manCounterRightBottom,manCounterMiddle1Top) := 9;
distance(manCounterRightBottom,manCounterMiddle2Bottom) := 3;
distance(manCounterRightBottom,manCounterMiddle2Top) := 11;
distance(manCounterMiddle1Bottom,manLettuceDispenser) := 5;
distance(manCounterMiddle1Bottom,manTomatoDispenser) := 4;
distance(manCounterMiddle1Bottom,manPlateDispenser) := 2;
distance(manCounterMiddle1Bottom,manDeliver) := 4;
distance(manCounterMiddle1Bottom,manCucumberDispenser) := 8;
distance(manCounterMiddle1Bottom,manKnife1) := 9;
```

```
distance(manCounterMiddle1Bottom,manKnife2) := 11;
distance(manCounterMiddle1Bottom,manKnife3) := 9;
distance(manCounterMiddle1Bottom,manKnife4) := 11;
distance(manCounterMiddle1Bottom,manCounterLeftBottom) := 3;
distance(manCounterMiddle1Bottom,manCounterRightBottom) := 7;
distance(manCounterMiddle1Bottom,manCounterMiddle1Top) := 6;
distance(manCounterMiddle1Bottom,manCounterMiddle2Bottom) :=
   6;
distance(manCounterMiddle1Bottom,manCounterMiddle2Top) := 8;
distance(manCounterMiddle1Top,manLettuceDispenser) := 11;
distance(manCounterMiddle1Top,manTomatoDispenser) := 10;
distance(manCounterMiddle1Top,manPlateDispenser) := 8;
distance(manCounterMiddle1Top,manDeliver) := 6;
distance(manCounterMiddle1Top,manCucumberDispenser) := 10;
distance(manCounterMiddle1Top,manKnife1) := 3;
distance(manCounterMiddle1Top,manKnife2) := 5;
distance(manCounterMiddle1Top,manKnife3) := 7;
distance(manCounterMiddle1Top,manKnife4) := 9;
distance(manCounterMiddle1Top,manCounterLeftBottom) := 9;
distance(manCounterMiddle1Top,manCounterRightBottom) := 9;
distance(manCounterMiddle1Top,manCounterMiddle1Bottom) := 6;
distance(manCounterMiddle1Top,manCounterMiddle2Bottom) := 8;
distance(manCounterMiddle1Top,manCounterMiddle2Top) := 6;
distance(manCounterMiddle2Bottom,manLettuceDispenser) := 11;
distance(manCounterMiddle2Bottom,manTomatoDispenser) := 10;
distance(manCounterMiddle2Bottom,manPlateDispenser) := 8;
distance(manCounterMiddle2Bottom,manDeliver) := 6;
distance(manCounterMiddle2Bottom,manCucumberDispenser) := 2;
distance(manCounterMiddle2Bottom,manKnife1) := 11;
distance(manCounterMiddle2Bottom,manKnife2) := 13;
distance(manCounterMiddle2Bottom,manKnife3) := 11;
distance(manCounterMiddle2Bottom,manKnife4) := 13;
distance(manCounterMiddle2Bottom,manCounterLeftBottom) := 9;
distance(manCounterMiddle2Bottom,manCounterRightBottom) := 3;
distance(manCounterMiddle2Bottom,manCounterMiddle1Bottom) :=
   6;
distance(manCounterMiddle2Bottom,manCounterMiddle1Top) := 8;
distance(manCounterMiddle2Bottom,manCounterMiddle2Top) := 10;
distance(manCounterMiddle2Top,manLettuceDispenser) := 13;
distance(manCounterMiddle2Top,manTomatoDispenser) := 12;
distance(manCounterMiddle2Top,manPlateDispenser) := 10;
distance(manCounterMiddle2Top,manDeliver) := 8;
distance(manCounterMiddle2Top,manCucumberDispenser) := 12;
distance(manCounterMiddle2Top,manKnife1) := 9;
distance(manCounterMiddle2Top,manKnife2) := 11;
distance(manCounterMiddle2Top,manKnife3) := 1;
distance(manCounterMiddle2Top,manKnife4) := 3;
distance(manCounterMiddle2Top,manCounterLeftBottom) := 11;
distance(manCounterMiddle2Top,manCounterRightBottom) := 11;
distance(manCounterMiddle2Top,manCounterMiddle1Bottom) := 8;
distance(manCounterMiddle2Top,manCounterMiddle1Top) := 6;
distance(manCounterMiddle2Top,manCounterMiddle2Bottom) := 10;

// Tool locations
knife1.loc := taKnife1;
knife2.loc := taKnife2;
```

```
knife3.loc := taKnife3;
knife4.loc := taKnife4;

// State 0
[start] {
  cook1.loc := manCounterMiddle1Bottom;
  cook1.carrying := false;
  cook1.busy := false;
  cook2.loc := manCounterMiddle1Top;
  cook2.carrying := false;
  cook2.busy := false;
  lettuce1.loc := lettuceDispenser;
  lettuce2.loc := lettuceDispenser;
  lettuce3.loc := lettuceDispenser;
  lettuce4.loc := lettuceDispenser;
  lettuce5.loc := lettuceDispenser;
  lettuce1.chopped := false;
  lettuce2.chopped := false;
  lettuce3.chopped := false;
  lettuce4.chopped := false;
  lettuce5.chopped := false;
  lettuce1.arranged := false;
  lettuce2.arranged := false;
  lettuce3.arranged := false;
  lettuce4.arranged := false;
  lettuce5.arranged := false;
  tomato1.loc := tomatoDispenser;
  tomato2.loc := tomatoDispenser;
  tomato3.loc := tomatoDispenser;
  tomato4.loc := tomatoDispenser;
  tomato5.loc := tomatoDispenser;
  tomato1.chopped := false;
  tomato2.chopped := false;
  tomato3.chopped := false;
  tomato4.chopped := false;
  tomato5.chopped := false;
  tomato1.arranged := false;
  tomato2.arranged := false;
  tomato3.arranged := false;
  tomato4.arranged := false;
  tomato5.arranged := false;
  cucumber1.loc := cucumberDispenser;
  cucumber2.loc := cucumberDispenser;
  cucumber3.loc := cucumberDispenser;
  cucumber4.loc := cucumberDispenser;
  cucumber5.loc := cucumberDispenser;
  cucumber1.chopped := false;
  cucumber2.chopped := false;
  cucumber3.chopped := false;
  cucumber4.chopped := false;
  cucumber5.chopped := false;
  cucumber1.arranged := false;
  cucumber2.arranged := false;
  cucumber3.arranged := false;
  cucumber4.arranged := false;
  cucumber5.arranged := false;
```

```
  plate1.loc := plateDispenser;
  plate2.loc := plateDispenser;
  plate3.loc := plateDispenser;
  plate4.loc := plateDispenser;
  plate5.loc := plateDispenser;
  knife1.processing := false;
  knife2.processing := false;
  knife3.processing := false;
  knife4.processing := false;
  client1.carrying := false;
  client2.carrying := false;
  client3.carrying := false;
  client4.carrying := false;
  client5.carrying := false;
  client1.busy := false;
  client2.busy := false;
  client3.busy := false;
  client4.busy := false;
  client5.busy := false;
  client1.loc := manDeliver;
  client2.loc := manDeliver;
  client3.loc := manDeliver;
  client4.loc := manDeliver;
  client5.loc := manDeliver;
};
```

C.2 Complex Map

Listing 8: The *Overcooked* environment "complex map"

```
/** Map B or complex Burgers Environment **/

forall(Ingredient i) {arrangetime(i) := 10; };
forall(Carryable c) {droptime(c) := 4; };
forall(Carryable c) {pickuptime(c) := 6; };
forall(Carryable c) {givetime(c) := 5; };
forall(Choppable c) {choptime(c) := 11; };
forall(Boilable b) {boiltime(l) := 30; };
forall(Fryable f) {frytime(f) := 30; };

instance Cook cook1,cook2;
instance Client client1,client2,client3,client4,client5;

instance Knife knife1,knife2;
instance Pan pan1,pan2;
instance Lettuce lettuce1,lettuce2,lettuce3,lettuce4,lettuce5;
instance Tomato tomato1,tomato2,tomato3,tomato4,tomato5;
instance Beef beef1,beef2,beef3,beef4,beef5;
instance BurgerBun bun1,bun2,bun3,bun4,bun5;
instance Plate plate1,plate2,plate3,plate4,plate5;


instance ManArea manBurgerBunDispenser,manTomatoDispenser,
   manBeefDispenser,manLettuceDispenser,manDeliver,
   manCounterMiddle1Top,manCounterMiddle1Bottom,
```

```
    manCounterMiddle2Top,manCounterMiddle2Bottom,
    manPlateDispenser,manPan1,manPan2,manKnife1,manKnife2;
instance ToolArea taPan1,taPan2,taKnife1,taKnife2;
instance PlArea counterMiddle1;
instance ArrangeArea counterMiddle2;
instance StorageArea lettuceDispenser,tomatoDispenser,
    beefDispenser,plateDispenser,burgerBunDispenser;

// Manipulation Areas <-> Placement Areas
connected(counterMiddle1,manCounterMiddle1Bottom) := true;
connected(counterMiddle1,manCounterMiddle1Top) := true;
connected(counterMiddle2,manCounterMiddle2Bottom) := true;
connected(counterMiddle2,manCounterMiddle2Top) := true;

connected(lettuceDispenser,manLettuceDispenser) := true;
connected(tomatoDispenser,manTomatoDispenser) := true;
connected(beefDispenser,manBeefDispenser) := true;
connected(burgerBunDispenser,manBurgerBunDispenser) := true;
connected(plateDispenser,manPlateDispenser) := true;

connected(taPan1,manPan1) := true;
connected(taPan2,manPan2) := true;
connected(taKnife1,manKnife1) := true;
connected(taKnife2,manKnife2) := true;

// Distances
distance(manBurgerBunDispenser,manTomatoDispenser) := 1;
distance(manBurgerBunDispenser,manBeefDispenser) := 2;
distance(manBurgerBunDispenser,manLettuceDispenser) := 3;
distance(manBurgerBunDispenser,manDeliver) := 7;
distance(manBurgerBunDispenser,manCounterMiddle1Top) := 5;
distance(manBurgerBunDispenser,manCounterMiddle1Bottom) := 19;
distance(manBurgerBunDispenser,manCounterMiddle2Top) := 9;
distance(manBurgerBunDispenser,manCounterMiddle2Bottom) := 15;
distance(manBurgerBunDispenser,manPlateDispenser) := 22;
distance(manBurgerBunDispenser,manPan1) := 23;
distance(manBurgerBunDispenser,manPan2) := 24;
distance(manBurgerBunDispenser,manKnife1) := 22;
distance(manBurgerBunDispenser,manKnife2) := 20;
distance(manTomatoDispenser,manBurgerBunDispenser) := 1;
distance(manTomatoDispenser,manBeefDispenser) := 1;
distance(manTomatoDispenser,manLettuceDispenser) := 2;
distance(manTomatoDispenser,manDeliver) := 6;
distance(manTomatoDispenser,manCounterMiddle1Top) := 4;
distance(manTomatoDispenser,manCounterMiddle1Bottom) := 18;
distance(manTomatoDispenser,manCounterMiddle2Top) := 8;
distance(manTomatoDispenser,manCounterMiddle2Bottom) := 14;
distance(manTomatoDispenser,manPlateDispenser) := 21;
distance(manTomatoDispenser,manPan1) := 22;
distance(manTomatoDispenser,manPan2) := 23;
distance(manTomatoDispenser,manKnife1) := 21;
distance(manTomatoDispenser,manKnife2) := 19;
distance(manBeefDispenser,manBurgerBunDispenser) := 2;
distance(manBeefDispenser,manTomatoDispenser) := 1;
distance(manBeefDispenser,manLettuceDispenser) := 1;
distance(manBeefDispenser,manDeliver) := 5;
```

```
distance(manBeefDispenser,manCounterMiddle1Top) := 3;
distance(manBeefDispenser,manCounterMiddle1Bottom) := 17;
distance(manBeefDispenser,manCounterMiddle2Top) := 7;
distance(manBeefDispenser,manCounterMiddle2Bottom) := 13;
distance(manBeefDispenser,manPlateDispenser) := 20;
distance(manBeefDispenser,manPan1) := 21;
distance(manBeefDispenser,manPan2) := 22;
distance(manBeefDispenser,manKnife1) := 20;
distance(manBeefDispenser,manKnife2) := 18;
distance(manLettuceDispenser,manBurgerBunDispenser) := 3;
distance(manLettuceDispenser,manTomatoDispenser) := 2;
distance(manLettuceDispenser,manBeefDispenser) := 1;
distance(manLettuceDispenser,manDeliver) := 4;
distance(manLettuceDispenser,manCounterMiddle1Top) := 2;
distance(manLettuceDispenser,manCounterMiddle1Bottom) := 16;
distance(manLettuceDispenser,manCounterMiddle2Top) := 6;
distance(manLettuceDispenser,manCounterMiddle2Bottom) := 12;
distance(manLettuceDispenser,manPlateDispenser) := 19;
distance(manLettuceDispenser,manPan1) := 20;
distance(manLettuceDispenser,manPan2) := 21;
distance(manLettuceDispenser,manKnife1) := 19;
distance(manLettuceDispenser,manKnife2) := 17;
distance(manDeliver,manBurgerBunDispenser) := 7;
distance(manDeliver,manTomatoDispenser) := 6;
distance(manDeliver,manBeefDispenser) := 5;
distance(manDeliver,manLettuceDispenser) := 4;
distance(manDeliver,manCounterMiddle1Top) := 6;
distance(manDeliver,manCounterMiddle1Bottom) := 12;
distance(manDeliver,manCounterMiddle2Top) := 2;
distance(manDeliver,manCounterMiddle2Bottom) := 8;
distance(manDeliver,manPlateDispenser) := 15;
distance(manDeliver,manPan1) := 16;
distance(manDeliver,manPan2) := 17;
distance(manDeliver,manKnife1) := 15;
distance(manDeliver,manKnife2) := 13;
distance(manCounterMiddle1Top,manBurgerBunDispenser) := 5;
distance(manCounterMiddle1Top,manTomatoDispenser) := 4;
distance(manCounterMiddle1Top,manBeefDispenser) := 3;
distance(manCounterMiddle1Top,manLettuceDispenser) := 2;
distance(manCounterMiddle1Top,manDeliver) := 6;
distance(manCounterMiddle1Top,manCounterMiddle1Bottom) := 14;
distance(manCounterMiddle1Top,manCounterMiddle2Top) := 4;
distance(manCounterMiddle1Top,manCounterMiddle2Bottom) := 10;
distance(manCounterMiddle1Top,manPlateDispenser) := 17;
distance(manCounterMiddle1Top,manPan1) := 18;
distance(manCounterMiddle1Top,manPan2) := 19;
distance(manCounterMiddle1Top,manKnife1) := 17;
distance(manCounterMiddle1Top,manKnife2) := 15;
distance(manCounterMiddle1Bottom,manBurgerBunDispenser) := 19;
distance(manCounterMiddle1Bottom,manTomatoDispenser) := 18;
distance(manCounterMiddle1Bottom,manBeefDispenser) := 17;
distance(manCounterMiddle1Bottom,manLettuceDispenser) := 16;
distance(manCounterMiddle1Bottom,manDeliver) := 12;
distance(manCounterMiddle1Bottom,manCounterMiddle1Top) := 14;
distance(manCounterMiddle1Bottom,manCounterMiddle2Top) := 10;
```

```
distance(manCounterMiddle1Bottom,manCounterMiddle2Bottom) :=
   4;
distance(manCounterMiddle1Bottom,manPlateDispenser) := 3;
distance(manCounterMiddle1Bottom,manPan1) := 4;
distance(manCounterMiddle1Bottom,manPan2) := 5;
distance(manCounterMiddle1Bottom,manKnife1) := 3;
distance(manCounterMiddle1Bottom,manKnife2) := 3;
distance(manCounterMiddle2Top,manBurgerBunDispenser) := 9;
distance(manCounterMiddle2Top,manTomatoDispenser) := 8;
distance(manCounterMiddle2Top,manBeefDispenser) := 7;
distance(manCounterMiddle2Top,manLettuceDispenser) := 6;
distance(manCounterMiddle2Top,manDeliver) := 2;
distance(manCounterMiddle2Top,manCounterMiddle1Top) := 4;
distance(manCounterMiddle2Top,manCounterMiddle1Bottom) := 10;
distance(manCounterMiddle2Top,manCounterMiddle2Bottom) := 6;
distance(manCounterMiddle2Top,manPlateDispenser) := 13;
distance(manCounterMiddle2Top,manPan1) := 14;
distance(manCounterMiddle2Top,manPan2) := 15;
distance(manCounterMiddle2Top,manKnife1) := 13;
distance(manCounterMiddle2Top,manKnife2) := 11;
distance(manCounterMiddle2Bottom,manBurgerBunDispenser) := 15;
distance(manCounterMiddle2Bottom,manTomatoDispenser) := 14;
distance(manCounterMiddle2Bottom,manBeefDispenser) := 13;
distance(manCounterMiddle2Bottom,manLettuceDispenser) := 12;
distance(manCounterMiddle2Bottom,manDeliver) := 8;
distance(manCounterMiddle2Bottom,manCounterMiddle1Top) := 10;
distance(manCounterMiddle2Bottom,manCounterMiddle1Bottom) :=
   4;
distance(manCounterMiddle2Bottom,manCounterMiddle2Top) := 6;
distance(manCounterMiddle2Bottom,manPlateDispenser) := 7;
distance(manCounterMiddle2Bottom,manPan1) := 8;
distance(manCounterMiddle2Bottom,manPan2) := 9;
distance(manCounterMiddle2Bottom,manKnife1) := 7;
distance(manCounterMiddle2Bottom,manKnife2) := 5;
distance(manPlateDispenser,manBurgerBunDispenser) := 22;
distance(manPlateDispenser,manTomatoDispenser) := 21;
distance(manPlateDispenser,manBeefDispenser) := 20;
distance(manPlateDispenser,manLettuceDispenser) := 19;
distance(manPlateDispenser,manDeliver) := 15;
distance(manPlateDispenser,manCounterMiddle1Top) := 17;
distance(manPlateDispenser,manCounterMiddle1Bottom) := 3;
distance(manPlateDispenser,manCounterMiddle2Top) := 13;
distance(manPlateDispenser,manCounterMiddle2Bottom) := 7;
distance(manPlateDispenser,manPan1) := 1;
distance(manPlateDispenser,manPan2) := 2;
distance(manPlateDispenser,manKnife1) := 4;
distance(manPlateDispenser,manKnife2) := 6;
distance(manPan1,manBurgerBunDispenser) := 23;
distance(manPan1,manTomatoDispenser) := 22;
distance(manPan1,manBeefDispenser) := 21;
distance(manPan1,manLettuceDispenser) := 20;
distance(manPan1,manDeliver) := 16;
distance(manPan1,manCounterMiddle1Top) := 18;
distance(manPan1,manCounterMiddle1Bottom) := 4;
distance(manPan1,manCounterMiddle2Top) := 14;
distance(manPan1,manCounterMiddle2Bottom) := 8;
```

```
distance(manPan1,manPlateDispenser) := 1;
distance(manPan1,manPan2) := 1;
distance(manPan1,manKnife1) := 3;
distance(manPan1,manKnife2) := 5;
distance(manPan2,manBurgerBunDispenser) := 24;
distance(manPan2,manTomatoDispenser) := 23;
distance(manPan2,manBeefDispenser) := 22;
distance(manPan2,manLettuceDispenser) := 21;
distance(manPan2,manDeliver) := 17;
distance(manPan2,manCounterMiddle1Top) := 19;
distance(manPan2,manCounterMiddle1Bottom) := 5;
distance(manPan2,manCounterMiddle2Top) := 15;
distance(manPan2,manCounterMiddle2Bottom) := 9;
distance(manPan2,manPlateDispenser) := 2;
distance(manPan2,manPan1) := 1;
distance(manPan2,manKnife1) := 2;
distance(manPan2,manKnife2) := 4;
distance(manKnife1,manBurgerBunDispenser) := 22;
distance(manKnife1,manTomatoDispenser) := 21;
distance(manKnife1,manBeefDispenser) := 20;
distance(manKnife1,manLettuceDispenser) := 19;
distance(manKnife1,manDeliver) := 15;
distance(manKnife1,manCounterMiddle1Top) := 17;
distance(manKnife1,manCounterMiddle1Bottom) := 3;
distance(manKnife1,manCounterMiddle2Top) := 13;
distance(manKnife1,manCounterMiddle2Bottom) := 7;
distance(manKnife1,manPlateDispenser) := 4;
distance(manKnife1,manPan1) := 3;
distance(manKnife1,manPan2) := 2;
distance(manKnife1,manKnife2) := 2;
distance(manKnife2,manBurgerBunDispenser) := 20;
distance(manKnife2,manTomatoDispenser) := 19;
distance(manKnife2,manBeefDispenser) := 18;
distance(manKnife2,manLettuceDispenser) := 17;
distance(manKnife2,manDeliver) := 13;
distance(manKnife2,manCounterMiddle1Top) := 15;
distance(manKnife2,manCounterMiddle1Bottom) := 3;
distance(manKnife2,manCounterMiddle2Top) := 11;
distance(manKnife2,manCounterMiddle2Bottom) := 5;
distance(manKnife2,manPlateDispenser) := 6;
distance(manKnife2,manPan1) := 5;
distance(manKnife2,manPan2) := 4;
distance(manKnife2,manKnife1) := 2;

// Tool locations
knife1.loc := taKnife1;
knife2.loc := taKnife2;
pan1.loc := taPan1;
pan2.loc := taPan2;

// State 0
[start] {
  cook1.loc := manCounterMiddle1Top;
  cook1.carrying := false;
  cook1.busy := false;
  cook2.loc := manCounterMiddle1Bottom;
```

```
cook2.carrying := false;
cook2.busy := false;
lettuce1.loc := lettuceDispenser;
lettuce2.loc := lettuceDispenser;
lettuce3.loc := lettuceDispenser;
lettuce4.loc := lettuceDispenser;
lettuce5.loc := lettuceDispenser;
lettuce1.chopped := false;
lettuce2.chopped := false;
lettuce3.chopped := false;
lettuce4.chopped := false;
lettuce5.chopped := false;
lettuce1.arranged := false;
lettuce2.arranged := false;
lettuce3.arranged := false;
lettuce4.arranged := false;
lettuce5.arranged := false;
tomato1.loc := tomatoDispenser;
tomato2.loc := tomatoDispenser;
tomato3.loc := tomatoDispenser;
tomato4.loc := tomatoDispenser;
tomato5.loc := tomatoDispenser;
tomato1.chopped := false;
tomato2.chopped := false;
tomato3.chopped := false;
tomato4.chopped := false;
tomato5.chopped := false;
tomato1.arranged := false;
tomato2.arranged := false;
tomato3.arranged := false;
tomato4.arranged := false;
tomato5.arranged := false;
beef1.loc := beefDispenser;
beef2.loc := beefDispenser;
beef3.loc := beefDispenser;
beef4.loc := beefDispenser;
beef5.loc := beefDispenser;
beef1.fryed := false;
beef2.fryed := false;
beef3.fryed := false;
beef4.fryed := false;
beef5.fryed := false;
beef1.arranged := false;
beef2.arranged := false;
beef3.arranged := false;
beef4.arranged := false;
beef5.arranged := false;
bun1.loc := burgerBunDispenser;
bun2.loc := burgerBunDispenser;
bun3.loc := burgerBunDispenser;
bun4.loc := burgerBunDispenser;
bun5.loc := burgerBunDispenser;
bun1.arranged := false;
bun2.arranged := false;
bun3.arranged := false;
bun4.arranged := false;
```

```
  bun5.arranged := false;
  plate1.loc := plateDispenser;
  plate2.loc := plateDispenser;
  plate3.loc := plateDispenser;
  plate4.loc := plateDispenser;
  plate5.loc := plateDispenser;
  knife1.processing := false;
  knife2.processing := false;
  pan1.processing := false;
  pan2.processing := false;
  client1.carrying := false;
  client2.carrying := false;
  client3.carrying := false;
  client4.carrying := false;
  client5.carrying := false;
  client1.busy := false;
  client2.busy := false;
  client3.busy := false;
  client4.busy := false;
  client5.busy := false;
  client1.loc := manDeliver;
  client2.loc := manDeliver;
  client3.loc := manDeliver;
  client4.loc := manDeliver;
  client5.loc := manDeliver;
};
```