HOUSEKEEPING WITH
MULTIPLE AUTONOMOUS ROBOTS:
REPRESENTATION, REASONING, AND EXECUTION

by

Erdi Aker

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfillment of
the requirements for the degree of
Master of Science

Sabanci University
August 2013

HOUSEKEEPING WITH
MULTIPLE AUTONOMOUS ROBOTS:
REPRESENTATION, REASONING, AND EXECUTION

Approved by:

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

Assoc. Prof. Dr. Volkan Patoğlu
(Thesis Co-Advisor)   ....

Assoc. Prof Dr. Berrin Yanıkoğlu   ....

Asst. Prof. Dr. Hüsnü Yenigün   ....

Prof. Dr. Ali Rana Atılgan   ....

Date of approval: August 13, 2013

HOUSEKEEPING WITH
MULTIPLE AUTONOMOUS ROBOTS:
REPRESENTATION, REASONING, AND EXECUTION

Erdi Aker

Computer Science and Engineering, Master of Science, 2013

Thesis Supervisors: Esra Erdem, Volkan Patoğlu

Keywords: domestic service robot, answer set programming, commonsense knowledge

Abstract

We consider a housekeeping domain with static or movable objects, where the goal is for multiple autonomous robots to tidy a house collaboratively in a given amount of time. This domain is challenging in the following ways: commonsense knowledge (e.g., expected locations of objects in the house) is required for intelligent behavior of robots; geometric constraints are required to find feasible plans (e.g., to avoid collisions); in case of plan failure while execution (e.g., due to a collision with movable objects whose presence and location are not known in advance or due to heavy objects that cannot be lifted by a single robot), recovery is required depending on the cause of failure; and collaboration of robots is required to complete some tasks (e.g., carrying heavy objects). We introduce a formal planning, execution and monitoring framework to address the challenges of this domain, by embedding knowledge representation and automated reasoning in each level of decision-making (that consists of discrete task planning, continuous motion planning, and plan execution), in such a way as to tightly integrate these levels. At the high-level, we represent not only actions and change but also commonsense knowledge in a logic-based formalism. Geometric reasoning is lifted to the high-level by embedding motion planning in the domain description. Then a discrete plan is computed for each robot using an automated reasoner. At the mid-level, if a continuous trajectory cannot be computed by a motion planner because the discrete plan is not feasible at the continuous-level, then a different plan is computed by the automated reasoner subject to some (temporal) conditions represented as formulas. At the low-level, if the plan execution fails, then a new continuous trajectory is computed by a motion planner at the mid-level or a new discrete plan is computed using an automated reasoner at the high-level. We illustrate the applicability of this formal framework with a simulation of a housekeeping domain.

ÇOKLU OTONOM ROBOTLARLA EV İDARESİ: GÖSTERİM, AKIL YÜRÜTME, İCRA TAKİBİ

Erdi Aker

Bilgisayar Bilimi ve Mühendisliği, Yüksek Lisans, 2013

Tez Danışmanları: Esra Erdem, Volkan Patoğlu

Anahtar Kelimeler: ev içi hizmet robotu, çözüm kümesi programlama, sağduyusal bilgi

Özet

Sabit ve hareket edebilir eşyaların yer aldığı bir evi, belirli bir süre içerisinde, birden fazla robotun işbirliğiyle derli toplu hale getirmenin hedeflendiği bir ev idaresi ortamını ele alıyoruz. Söz konusu ortam şu zorlukları barındırmaktadır: robotların akıllıca davranabilmesi için sağduyusal bilgiye (örn. ev içindeki eşyaların bulunmaları gereken yerler) sahip olmaları gerekmektedir; uygulanabilir planların elde edilebilmesi için (örn. çarpışmalardan sakınmak amacıyla) geometrik kısıtlar kullanılmalıdır; plan icrası sırasında hataların gerçekleşmesi durumunda (örn. varlığı ya da yeri bilinmeyen bir eşya ile çarpışılması, ya da ağır bir eşyanın tek bir robot tarafından kaldırılamaması sonucu) sorunun turu göz önüne alınarak hata telafi edimelidir; bazı görevlerin yerine getirilebilmesi için (örn. ağır bir eşyanın taşınması) robotların işbirliğinde bulunması gerekmektedir. Bu zorlukların üstesinden gelmek amacıyla, karar verme sürecinin her seviyesine (kesikli görev planlama, sürekli hareket planlama ve plan icrası dahil olmak üzere) bu seviyeleri sıkıca bütünleştirecek bir şekilde bilgi gösterimi ve otomatik akıl yürütmenin gömüldüğü bir biçimsel planlama, icra ve denetleme sistemini öne sürüyoruz. Üst seviyede, eylemler ve değişimlerin yanı sıra, sağduyusal bilgiyi de mantık tabanlı biçimselcilikler yoluyla betimliyoruz. Geometrik akıl yürütmeyi, hareket planlamayı ortam gösterimine gömerek üst seviyeye çekiyoruz. Sonrasında otomatik akıl yürütücüler yardımıyla her bir robot için kesikli planlar hesaplıyoruz. Orta seviyede, kesikli planın sürekli seviyede uygulanabilir olmamasından dolayı hareket planlayıcının sürekli bir gezinge bulamaması durumunda, bazı (zamansal) koşulları otomatik akıl yürütücülere formüller vasıtasıyla sunarak farklı planlar hesaplıyoruz. Alt seviyede, plan icrasında hata olması durumda, orta seviyedeki hareket planlayıcıya yeni bir sürekli gezinge hesaplatıyor, ya da üst seviyedeki otomatik akıl yürütücüyü kullanarak yeni bir kesikli plan buluyoruz. Bu biçimsel sistemin uygulanabilirliğini simülasyon aracılığıyla gösteriyoruz.

ACKNOWLEDGEMENTS

I would like to thank my thesis advisors, Esra Erdem and Volkan Patoğlu, for their guidance throughout my research.

I also thank the members of my thesis jury, Berrin Yanıkoğlu, Hüsnü Yenigün, and Ali Rana Atılgan, for their comments and suggestions.

I would like to thank Ahmetcan Erdoğan for his participation in this work, his friendship, and for buying me lunch last week when I did not have enough credit in my SU card. Without him, this work would not have been as good, and I would be hungry.

During the period of this work, I have been a member of a research group, among many good people. I would like to thank Umut Öztok and Süha Mutluergil for their help, and making late nights in the office more bearable with their friendship and humor. I also thank Zeynep Doğmuş, Halit Erdoğan, Kadir Haspalamutgil, Giray Havur, Zeynep Sarıbatur, Peter Schüller, Fırat Tahaoğlu, and Tansel Uras, for their help, and being good company.

Inarguably, I am indebted to my parents Erman and Gülfer, and my little brother Gökhan, for their unconditional love and support. I am also grateful to all my friends, near me or far away, for being a part of my life. All this work would be meaningless without family and friends.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

Chapter 1

Introduction

Robots being a part of our daily lives inside our homes is not a dream of distant future. It is estimated that 2.5 million personal and domestic robots are sold in 2011 [73]. Still, we lack robots capable of fullfilling complex household tasks. While autonomous vacuum cleaners which specialize on floor coverage are becoming ubiquitous, a similar success has not been achieved for a robot that can accomplish a complicated task such as "tidying up a house". In this thesis, we focus on autonomous housekeeping robots whose task is to tidy up the house, from the perspective of cognitive robotics. In other words, our goal is to endow housekeeping robots with high-level cognitive capabilities so that they can operate in a smart and effective manner.

1.1   Challenges

Consider a house with several rooms. In each room, there is a number of objects. Some of them are stationary obstacles (e.g. sofa, table), while others are movable objects of different sorts (e.g. books, pillows). The movable objects can be misplaced, and we want autonomous robots to relocate any misplaced object to an appropriate location. Some of the challenges in such an environment can be listed as follows:

- **Representation and reasoning with commonsense knowledge** The task of tidying a house requires extensive knowledge about objects and their expected locations. For instance, a book found in the kitchen should be put on a bookshelf, or a dirty dish found on a table in the living room should be inserted into the dishwasher in the kitchen. This type of knowledge is trivial for human beings, and referred as "commonsense knowledge".

  By definition, every human is assumed to have commonsense knowledge. As a result, we do not explicitly mention any knowledge that falls into this category when we communicate with other human beings. For instance, parents ask their

children to tidy their rooms, and the children automatically deduce that their parents want the clothes lying on the floor to be relocated inside their wardrobes, without any need for further explanation. Unfortunately, robots do not have commonsense knowledge. To render them capable of housekeeping without any need of extensive human instruction or supervision, commonsense knowledge should be represented and made accessible to them.

- **Integration of high-level symbolic reasoning and low-level geometric reasoning** In order to tidy up a house, robots need to devise some plans. At the task level, these plans consist of a sequence of discrete actions. For instance, a high-level task plan may contain an action such as "going from bedroom to kitchen". Since a house-keeping robot operates in a physical environment, these high-level plans should also respect some low-level geometric constraints to be applicable. For instance, to be able to move from bedroom to kitchen, "a collision-free trajectory should exists". Therefore, an integration between high-level task planning and low-level geometric reasoning should be established for housekeeping robots to operate.

- **Planning with complex constraints** It goes without saying that robots need to behave as smart as possible to be considered successful in a task mostly performed by humans such as housekeeping. We should be able to request from a cleaning robot that it should finish its job by a certain deadline. Or a cleaning robot in need for help from another robot should arrange its plan to distract the other robot at little as possible. A human can handle these temporal constraints, therefore our housekeeping robots should also be able to plan accordingly.

- **Recovery from possible plan failures** When a plan execution fails, depending on the cause of the failure, a recovery should be made. If a robot collides with a movable object whose presence and location is not known earlier (e.g., a human may bring the movable object into the room while the robot is executing the plan), then the robot may ask the motion planner to find a different trajectory to reach the next state, and continue with the plan. If the robot cannot find a trajectory, then a new plan can be computed to tidy the room. Monitoring executions of plans by multiple cleaning robots, taking into decisions for recovery from failures, is challenging.

1.2 Our Approach

We address these challenges by utilizing the knowledge representation and reasoning formalisms, action language $\mathcal{C}+$ [41] and Answer Set Programming (ASP) [36, 62, 7], and relevant automated reasoners, such as SAT solver MINISAT [19] and ASP solvers

`iclingo` [34] and `dlvhex` [21] for high-level reasoning. These formalisms allow us to express some useful concepts in our representation such as concurrent actions and defaults. Also their respective reasoners provide effective mechanisms such as external predicates/functions to integrate external computational sources into the reasoning process. We also utilize probabilistic motion planning techniques for geometric reasoning such as Rapidly-exploring Random Trees (RRT) [59], and commonsense knowledge bases such as CONCEPTNET [64]. Our approach to the housekeeping problem can be summarized in three parts:

- **Representation** We represent the housekeeping domain using action language $\mathcal{C}+$. Using external predicates/functions, we embed geometric reasoning into our high-level representations. Also, we automatically extract knowledge about the expected locations of objects from commonsense knowledge base CONCEPTNET, and embed this knowledge into our representation using external predicates as well. Then, we reformulate the representation in $\mathcal{C}+$ in terms of Answer Set Programming. During this transformation, we apply some further simplifications while keeping the soundness of the formulation intact.

- **Reasoning** Once geometric reasoning is embedded using external predicates and functions, we compute hybrid plans using reasoners such as CCALC, `iclingo`, and `dlvhex`. To come up with plans that satisfy complex temporal goals, we provide queries with temporal formula to the reasoners.

- **Execution** In order to let the robots recover from possible plan failures, such as unknown objects blocking the path of a robot, disappearance of objects due to human intervention, or other failures caused by discrepancy between robots' knowledge of the world and the truth, we introduce an execution monitoring algorithm, and integrate it with our hybrid planning approach.

1.3 Contributions

Our contributions can be summarized as follows:

- We have represented the housekeeping domain in the action language $\mathcal{C}+$. We have investigated transformations from $\mathcal{C}+$ to other formalisms, and identified possible simplifications. Using these transformations and simplifications, we have obtained a description of the housekeeping domain in ASP. By this way, various automated reasoners can be used to find plans to tidy a house.

- We have introduced a method for housekeeping robots to extract relevant commonsense knowledge from an existing knowledge base and integrate this knowledge

3

with a logic-based domain description, using external predicates in the action description language $\mathcal{C}+$ and in ASP.

- We have embedded geometric reasoning in high-level domain description using external predicates. By this way, we have obtained feasible hybrid plans. Using different reasoners with varying capabilities, we have investigated different extents of this integration.

- We have introduced a novel execution monitoring algorithm for recovering from possible failures in housekeeping environment. Unlike the existing approaches in the literature, after identification of the failure, the algorithm does not simply replan, but tries to avoid a costly replan by utilizing other means of recovery whenever it is possible.

- We have obtained some experimental results regarding the performance of reasoners CCALC, iclingo, and dlvhex that can be used as a benchmark for similar robotic applications.

- We have implemented a housekeeping robot framework that utilizes the planning and monitoring approach we have described, using Robot Operating System (ROS) [78] tools and libraries. We have showed the applicability of our approach by doing dynamic simulations using Gazebo[1].

1.4   Thesis Outline

We continue with some preliminaries on logic-based domain representation, specifically action language $\mathcal{C}+$, and ASP in Chapter 2. In Chapter 3, we show how we apply the techniques mentioned in the previous chapter for the representation of housekeeping domain. We provide some details about reasoning on housekeeping domain, and show some benchmarks comparing different reasoners in Chapter 4. Chapter 5 describes our execution monitoring approach, and demostrates its applicability. Related work is briefly summarized in Chapter 6. We conclude in Chapter 7 by summarizing our results, and emphasizing possible future work.

---

[1]http://gazebosim.org/

Chapter 2

Representing Action Domains

For representing dynamic systems and planning, there exist many solutions in the literature. Some of them are graph or net based approaches, while others depend on logic-based formalisms. In this thesis, we mainly focus on logic-based formalisms, specifically the action description language $\mathcal{C}+$ [41] and Answer Set Programming [36, 62, 7].

2.1   Brief Summary of Existing Logic-Based Formalisms

An abundance of logical formalisms aim the representation of dynamic domains and reasoning about them. Here, some notable examples of these formalisms are briefly mentioned.

One of the first instances of a special purpose formalism for representing dynamic systems is situation calculus introduced by McCarthy [69], and further detailed by McCarthy et al. [71]. It is a first-order language (some variations may include second-order features) in which the dynamic world is modeled using action histories called *situations* [63]. A large body of studies have been made on this formalism, not only in theoretical sense but also from the practical point of view. One notable example of such application-oriented studies is GOLOG, a situation calculus based high-level robot programming language [60][61].

Fluent calculus is an extension to situation calculus that aim to solve the frame problem [71] [83]. It provides a special construct for the representations of *states* which helps with the state update axiomatization.

Event calculus is another formalism, introduced by Kowalski et al. [56]. The ontology of the formalism consists of *events*, *fluents*, and *time points*. There exist many dialects of the original formalism, and most of them make use of circumscription [70] for the purpose of dealing with frame problem. An action language called $\mathcal{E}$ is closely related to this formalism, and event calculus is used as the underlying logic in many implementations of this action language [54].

Temporal action logics (TAL) is a class of logics that originated from Sandewall et al.'s work [79]. It is a quite expressive framework that can handle non-deterministic, context-dependent, concurrent and durational actions [15]. It also has an award-winning reasoner called TALPLANNER [57].

Description logics (DL) is another class of logics that aims to represent application domain knowledge in a structured way, and to overcome the lack of formal semantics in other solutions suchs a frames and semantic networks [2]. Besides its original purpose, there also exist many studies that make use of this class of formalisms to solve planning problems [40].

Nonmonotonic causal logic is a formalism that represents actions and change in the environment in terms of *causal rules* [67]. It allows the representation of indirect effects of actions, implied action preconditions, concurrent interacting effects of actions, and spontaneous changes in the environment [87]. It is also the underlying logic of action language $\mathcal{C}+$ [41]. More detail will be provided regarding the syntax and semantics of causal logic and $\mathcal{C}+$ in Section 2.2 and 2.3.

Answer set programming (ASP) [36, 62, 7] is a logic programming paradigm based on stable model semantics [35]. Just like causal logic it is nonmonotonic, and this renders the formalism capable of representing defaults. Syntax and semantics of this language will be elaborated in Section 2.4.

In this thesis we use the nonmonotonic formalisms of $\mathcal{C}+$ and ASP for representing the dynamic domain of housekeeping.

2.2 Causal Logic

Causal logic [67] is the underlying formalism of the action language $\mathcal{C}+$ [41]. Here we provide a simple description of its syntax and semantics.

Syntax We start with a signature σ of symbols called *constants*. Every constant c has a nonempty set called *domain* denoted by $Dom(c)$ that identifies its possible values. An *atom* of σ is an expression of the form $c = v$ ("the value of c is v") where $c \in \sigma$ and $v \in Dom(c)$. A *formula* is a collection of atoms connected with logical connectives. A *theory* is a set of formulas. A *causal rule* is an expression of the form

$$\psi \Leftarrow \phi$$

where ψ and ϕ are formulas of the underlying signature σ. It reads as "if ϕ holds, then there is a cause for ψ to hold". A *causal theory* is a set of causal rules.

Semantics   An *interpretation* is a function mapping each constant c of σ to a value in $Dom(c)$. An interpretation I *satisfies* an atom $c = v$ if and only if $I(c) = v$. The satisfaction relation for an arbitrary formula is defined through the standard truth tables of logical connectives. A *model* of a theory is an interpretation that satisfies every formula in that theory. For a causal theory T and an interpretation I, T^I is defined as follows:

$$T^I = \{\psi \mid (\psi \Leftarrow \phi) \in T, I \models \phi\}.$$

An interpretation I is a model of a causal theory T if I is the only model of theory T^I. In other words, "if ϕ is true, then ψ is caused to be true".

For instance, let $\sigma = \{c\}$, $Dom(c) = \{v_1, v_2\}$, and $T = \{c = v_1 \Leftarrow c = v_1\}$. In this instance, the only possible interpretations are $I_1(c) = v_1$ and $I_2(c) = v_2$. Notice that I_1 is a model of T ($I_1 \models T$), since $T^{I_1} = \{c = v_1\}$ and I_1 is the only interpretation that satisfies T^{I_1}. On the other hand, I_2 is not a model ($I_2 \not\models T$) because $T^{I_2} = \{\}$ and both I_1 and I_2 satisfy T^{I_2}.

This example shows the nonmonotonicity of causal logic as well. Note that $c = v_1 \Leftarrow c = v_2$ is different from $c = v_2 \supset c = v_1$ ("$c = v_1$ if $c = v_2$") in classical logic, which has two models. Essentially, $c = v_1 \Leftarrow c = v_1$ expresses a default: "c normally has value v_1".

According to the semantics of causal theories: 1) every fact that obtains is caused, and 2) every fact that is caused obtains. The first commitment is called the "principle of universal causation" [86]. There are two cases, while describing a dynamic system, where this principle is "disabled": 1) when describing an initial state, 2) when describing actions. In other words, initial value of fluents and occurences of actions are considered "exogenous". We will see an example in the following.

When we represent a dynamic system, we are usually interested in reasoning about a sequence of states and actions over "time stamps" 0 to n. Then the constants in the signature are copied $n + 1$ times. For simplicity, we denote a copy of a constant c by putting time stamp i in front, like $i : c$.

Example (Bomb Disposal domain)   We can represent dynamic systems using causal theories. Let us represent the simple yet highly dangerous Bomb Disposal domain as an example.

In this domain, there is a bomb with two latches, "left" and "right". These latches can be facing either "up" or "down". If both of the latches are facing upwards, then the bomb is "defused".

First, we decide on the signature. Directions of the two latches and the bomb being defused or not defused constitute the state of world. The only action available is flipping

a latch. Therefore, the signature should consist of the following constants:

$$\sigma_{BD} = \{0 : up(left), 0 : up(right), 0 : defused, 0 : flip(left),$$
$$0 : flip(right), 1 : up(left), 1 : up(right), 1 : defused\}.$$

For simplicity, we only consider "histories" of length 1. Domain of every constant in the signature is defined as follows:

$$Dom(c) = \{True, False\} \quad (\forall c \in \sigma_{BD}).$$

Under this signature, the following causal theory is a representation of Bomb Disposal domain.

First, we state that latches can be facing up or down, and the bomb can be defused or not defused in the initial state (i.e. initial values of fluents are exogenous), as follows:

$$
\begin{aligned}
0 : up(left) &\Leftarrow 0 : up(left) \\
0 : \neg up(left) &\Leftarrow 0 : \neg up(left) \\
0 : up(right) &\Leftarrow 0 : up(right) \\
0 : \neg up(right) &\Leftarrow 0 : \neg up(right) \\
0 : defused &\Leftarrow 0 : defused \\
0 : \neg defused &\Leftarrow 0 : \neg defused
\end{aligned}
\tag{2.1}
$$

Similarly, actions can be executed or not executed at any time step, i.e they are exogenous:

$$
\begin{aligned}
0 : flip(left) &\Leftarrow 0 : flip(left) \\
0 : \neg flip(left) &\Leftarrow 0 : \neg flip(left) \\
0 : flip(right) &\Leftarrow 0 : flip(right) \\
0 : \neg flip(right) &\Leftarrow 0 : \neg flip(right)
\end{aligned}
\tag{2.2}
$$

Then, we represent the "commonsense law of inertia" to handle the "frame problem" by the following causal rules:

$$
\begin{aligned}
1 : up(left) &\Leftarrow 1 : up(left) \wedge 0 : up(left) \\
1 : \neg up(left) &\Leftarrow 1 : \neg up(left) \wedge 0 : \neg up(left) \\
1 : up(right) &\Leftarrow 1 : up(right) \wedge 0 : up(right) \\
1 : \neg up(right) &\Leftarrow 1 : \neg up(right) \wedge 0 : \neg up(right) \\
1 : defused &\Leftarrow 1 : defused \wedge 0 : defused \\
1 : \neg defused &\Leftarrow 1 : \neg defused \wedge 0 : \neg defused
\end{aligned}
\tag{2.3}
$$

According to these rules, the positions of the latches and the state of the bomb is preserved unless there is a cause for them to change. Direct effects of flipping switches are

represented as follows:

$$1 : up(left) \Leftarrow 0 : flip(left) \wedge 0 : \neg up(left)$$
$$1 : \neg up(left) \Leftarrow 0 : flip(left) \wedge 0 : up(left)$$
$$1 : up(right) \Leftarrow 0 : flip(right) \wedge 0 : \neg up(right)$$
$$1 : \neg up(right) \Leftarrow 0 : flip(left) \wedge 0 : up(right)$$

(2.4)

Finally, we can represent the condition in which the bomb is defused by the following causal rules:

$$0 : defused \Leftarrow 0 : up(left), 0 : up(right)$$
$$1 : defused \Leftarrow 1 : up(left), 1 : up(right)$$

(2.5)

Note that these causal rules express the "ramification" of flipping action.

2.3   Action Language $\mathcal{C}+$

Action language are formal models of parts of natural language that are used for describing dynamic systems [38]. There are various action languages, such as $\mathcal{A}$ [37], $\mathcal{B}$ [85], $\mathcal{C}$ [43], and $\mathcal{K}$ [20]. We use the action language $\mathcal{C}+$ [41] to describe the housekeeping domain.

Syntax   Similar to causal logic, we start with a *(multi-valued propositional) signature* that consists of a set σ of *constants* of two sorts, along with a nonempty finite set $Dom(c)$ of *value names*, disjoint from σ, assigned to each constant c. An *atom* of σ is an expression of the form $c = v$ ("the value of c is v") where $c \in \sigma$ and $v \in Dom(c)$. A *formula* of σ is a propositional combination of atoms. If c is a Boolean constant, we will use c (resp. $\neg c$) as shorthand for the atom $c = $ *True* (resp. $c = $ *False*).

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

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

$$\textbf{caused } F \textbf{ if } G \tag{2.6}$$

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

$$\textbf{caused } A \textbf{ if } G \tag{2.7}$$

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

$$\textbf{caused } F \textbf{ if } G \textbf{ after } H \qquad (2.8)$$

where F and G are as above, and H is a fluent formula. In (2.6), (2.7) and (2.8) the part **if** G can be dropped if G is $True$.

Semantics   The meaning of an action description can be represented by reducing it to causal theories described in Section 2.2. For a $\mathcal{C}+$ action description D and a natural number n, there exists a causal theory $T(D, n)$. To obtain such a causal theory, we transform every static causal law in D of the form (2.6) as

$$i : F \Leftarrow i : G \quad (0 \leq i \leq n) \qquad (2.9)$$

every action dynamic causal law of the form (2.7) as

$$i : A \Leftarrow i : G \quad (0 \leq i \leq n-1) \qquad (2.10)$$

and every fluent dynamic causal law of the form (2.8) as

$$(i+1) : F \Leftarrow (i+1) : G \wedge i : H \quad (0 \leq i \leq n-1). \qquad (2.11)$$

Furthermore, action description D constitutes a "transition system", and paths of length n in this transition system correspond to models of $T(D, n)$. This transition system can be thought of as a labeled directed graph whose nodes correspond to states of the world and edges to transitions between states. Every state is represented by a vertex labeled with a function from fluent constants to their values. Every transition is a triple $\langle s, A, s' \rangle$ that characterizes change from state s to state s' by execution of a set A of primitive actions. The transition system corresponding to Bomb Disposal domain in Figure 2.2 is provided in Figure 2.1.

Abbreviations   While describing action domains, we can use some abbreviations. For instance, we can describe the (conditional) direct effects of an action using expressions of the form

$$a \textbf{ causes } F \textbf{ if } G \qquad (2.12)$$

which abbreviates the fluent dynamic law

$$\textbf{caused } F \textbf{ if } \textit{True} \textbf{ after } a \wedge G$$
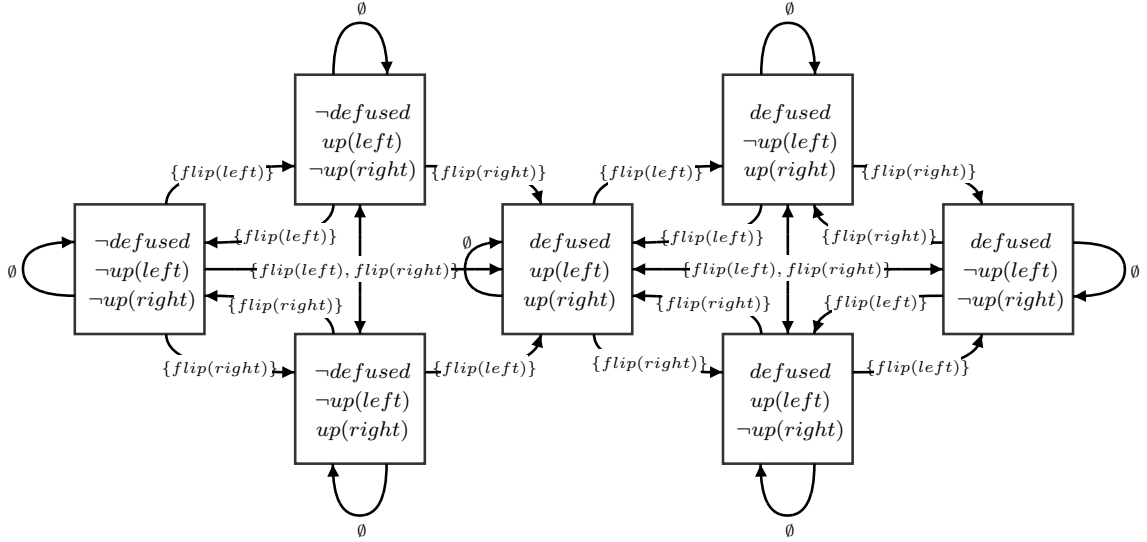
Figure 2.1: Transition diagram of Bomb Disposal domain

expressing that "executing action a at a state where G holds, causes F." We can formalize that F is a precondition of a by the expression

$$\textbf{nonexecutable } a \textbf{ if } \neg F \tag{2.13}$$

which stands for the fluent dynamic law

$$\textbf{caused } \textit{False} \textbf{ if } \textit{True} \textbf{ after } a \wedge \neg F.$$

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

$$\textbf{default } F$$

that abbreviates the static laws:

$$\textbf{caused } F \textbf{ if } F.$$

We can represent that the value of a fluent f remains the same unless there is a cause for its change, by the abbreviation

$$\textbf{inertial } f$$

that stands for the fluent dynamic causal law

$$\textbf{caused } f = v \textbf{ if } f = v \textbf{ after } f = v \quad (\forall v \in Dom(f)).$$

In almost all the action domains, we express that there is no cause for the occurrence of an action a, by the abbreviation

$$\textbf{exogenous } a$$

Sorts
> $latch$

Objects
> $left$, $right$ of $latch$

Variables
> L of $latch$

Simple fluent constants
> $up(latch)$, $defused$ with Boolean domains

Action constants
> $flip(latch)$ with a Boolean domain

$$\begin{aligned}&\textbf{inertial } up(L)\\&\textbf{inertial } defused\end{aligned} \tag{2.14}$$

$$\textbf{exogenous } flip(L) \tag{2.15}$$

$$\begin{aligned}&flip(L) \textbf{ causes } up(L) \textbf{ if } \neg up(L)\\&flip(L) \textbf{ causes } \neg up(L) \textbf{ if } up(L)\end{aligned} \tag{2.16}$$

$$\textbf{caused } defused \textbf{ if } up(left) \wedge up(right) \tag{2.17}$$

Figure 2.2: Bomb Disposal domain in $\mathcal{C}+$ (BD)

that abbreviates the following action dynamic laws:

$$\textbf{caused } a \textbf{ if } a$$
$$\textbf{caused } \neg a \textbf{ if } \neg a.$$

Example   An action description BD of Bomb Disposal domain is provided in Figure 2.2. Note that BD directly corresponds to the transition diagram in Figure 2.1. Also, the causal theory in Section 2.2 can be described as $T(BD, 1)$.

2.4   Answer Set Programming

Answer Set Programming (ASP) [36, 62, 7] is a logic programming paradigm based on stable model semantics. Here briefly mention its syntax and semantics.

The idea of ASP is to represent knowledge (e.g., actions of multiple robots) as a "program" and to reason about the knowledge (e.g., find a plan of robots' actions) by computing models, called "answer sets", of the program using "ASP solvers" like `iclingo` [34].

Syntax An *ASP program* Π over signature σ is a finite set of *rules* of the form

$$l_0 \ or \ \ldots or \ l_k \leftarrow l_{k+1}, \ldots, l_m, not \ l_{m+1}, \ldots, not \ l_n \qquad (2.18)$$

where $n \geq m \geq k \geq 0$. Each l_i is a literal (a propositional atom $p \in \sigma$ or its negation $\neg p$).

In such a rule, $l_0 \ or \ \ldots \ or \ l_k$ is called the *head*, and $l_{k+1}, \ldots, l_m, not \ l_{m+1}, \ldots,$ *not* l_n is called the *body* of the rule. If head of a rule is empty, then the rule is called a *constraint*. A rule with an empty body is described as a *fact*, and we generally omit the $\leftarrow$ sign in this case. Note the two sorts of negation: classical negation $\neg$ as in classical logic, and default negation *not*. A literal with or without a default negation is called an *extended literal*. 2-place connective *or* is called *epistemic disjunction*.

Semantics A consistent set of literals of an ASP program is called a *partial interpretation*. With respect to a partial interpretation I,

- a literal l is *true* if $l \in I$, *false* if $\bar{l} \in I$, and *unknown* otherwise

- an extended literal *not* l is *true* if $l \notin I$, *false* otherwise

- body of a rule (conjunction) is *true* if each element is true, *false* if at least one element is false, *unknown* otherwise

- head of a rule (disjunction) is *true* if at least one of the elements is true, *false* if all the elements are false, *unknown* otherwise.

A partial interpretation I *satisfies* a rule if the head of the rule is true whenever the body of the same rule is true with respect to I. If a partial interpretation satisfies all the rules of a program, it is called a *model* of the program. An *answer set* is a model of a program which is "minimal" in the sense of set-theoretic inclusion, among the other models of the program.

First, let us consider a *normal program* (a program that does not contain default negation) such as

$$p \leftarrow p$$

where the signature of program contains a single atom, namely p. Note that this simple program has two partial interpretations: $I_1 = \{p\}$, and $I_2 = \{\}$. Both of these interpretations are models, since they both satisfy the only rule of the program. But I_1 is not a minimal model, because a strict subset of it, namely I_2, is also a model of the program. Therefore, I_1 cannot be an answer set. On the other hand, I_2 is a model, and minimal, thus an answer set of the program.

While considering answer sets of arbitrary programs which may contain rules with default negation, we make use of a construct called "reduct". The *reduct* Π^I of a program

Π with respect to a partial interpretation I is the set of rules of the form

$$l_0 \ or \ \ldots or \ l_k \leftarrow l_{k+1}, \ldots, l_m$$

for each rule of Π of the form (2.18) where $\{l_{m+1}, \ldots, l_n\} \cap I = \emptyset$. Thus, Π^I turns out to be a normal program. A partial interpretation I is an answer set of a program Π, if it is an answer set of the reduct Π^I of the said program.

Now, let us consider another program Π with rules that contain default negation such as

$$p \leftarrow not \ q$$
$$q \leftarrow not \ p$$

where the signature of the program $\sigma_\Pi = \{p, q\}$. One of the possible partial interpretations of Π is and $I = \{p\}$. If we take the reduct of Π with respect to I, the result is

$$p \leftarrow$$

and we see that Π^I is satisfied by I. If we look at the strict subsets of I, which is only $I' = \{\}$, we see that they do not satisfy Π^I. Thus, I is a minimal model, i.e. an answer set of Π^I. That means I is also an answer set of Π. Similarly, $\{q\}$ is another answer set of Π.

Consider another program Π with the single rule

$$p \ or \ \neg p \leftarrow$$

which contains a disjunction. The only rule of this program has an empty body, so for an interpretation I to satisfy this rule, the head of the rule $(p \ or \ \neg p)$ should be true with respect to I. For a disjunction to be true with respect to I, at least one of the disjuncts should be an element of I. Partial interpretation $I_1 = \{p\}$ makes the head of the rule true since $I_1 \cap \{p, \neg p\} \neq \emptyset$, and satisfies this rule as a result. Since this is the only rule, I_1 is also a model of the program. The only strict subset of I_1 is $\{\}$, and it is not a model of the program Π. Therefore, $I_1 = \{p\}$ is an answer set of the program. Similarly, $\{\neg p\}$ is also another answer set of the program Π.

When we represent a problem in ASP, we use special constructs of the form

$$m \ \{l_1, \ldots, l_k\} \ n$$

(called *cardinality expressions*) where each l_i is a literal and m and n are nonnegative integers denoting the "lower bound" and the "upper bound" [81]. Programs using these constructs can be viewed as abbreviations for normal programs [26]. Such an expression describes the subsets of the set $\{l_1, \ldots, l_k\}$ whose cardinalities are at least m and at most n. Such expressions when used in heads of rules generate many answer sets whose

cardinality is at least m and at most n, and when used in constraints eliminate some answer sets. Rules with cardinality expressions in the head are called "choice rules".

A group of rules that follow a pattern can be often described in a compact way using "schematic variables". For instance, we can write the program

$$p_i \leftarrow \mathit{not}\ p_{i+1} \qquad (1 \leq i \leq 7)$$

as follows

$$\mathit{index}(1).\ \mathit{index}(2).\ ...\ \mathit{index}(7).$$
$$p(i) \leftarrow \mathit{not}\ p(i+1), \mathit{index}(i).$$

The auxiliary predicate $\mathit{index}(i)$ is introduced to describe the ranges of variables. ASP solvers compute an answer set for a given program that contains variables, after "grounding" the program. The "definitions" of such auxiliary predicates tell the ASP solver how to substitute specific values for variables in schematic expressions. Variables can be also used "locally" to describe the list of formulas. For instance, the rule

$$1\ \{p_1, \ldots, p_7\}\ 1$$

can be represented as follows:

$$1\ \{p(i) : \mathit{index}(i)\}\ 1.$$

Note that the semantics of ASP programs are defined above for ground programs.

2.5   A Transformation from $\mathcal{C}+$ to ASP

There are various transformations between causal logic, $\mathcal{C}+$, and ASP, such as [67] and [25]. There also exist some automated tools for this purpose, such as [33] and [3]. Unfortunately, these tools are not robust, and fail to transform some $\mathcal{C}+$ formulations, especially formulations that include causal laws with external predicates. Therefore, we choose to do the transformation manually. Let us consider the transformation as in [67].

Multi-valued signature to Boolean signature   In $\mathcal{C}+$, we are allowed to use multi-valued signatures, but the signature is Boolean in our ASP definition. Fortunately, we can transform a $\mathcal{C}+$ domain with a multi-valued signature into an analogous $\mathcal{C}+$ domain with a Boolean signature [42], and then continue forward with the transformation towards ASP.

For every multi-valued constant c with a domain $Dom(c) = \{v_1, \ldots, v_n\}$, we can replace c with $|Dom(c)|$ newly introduced constants $c'(v_1), \ldots, c'(v_n)$ such that $c'(v_i)$ corresponds to $c = v_i$. We also introduce some causal laws to make sure the new constants

behave like the old one. To ensure that no more than a single $c'(v_i)$ constant is true at any time, we say the following:

$$\textbf{caused } \neg c'(v_i) \textbf{ if } c'(v_j) \quad (\forall v_i, v_j \in Dom(c) \; st. \; v_i \neq v_j)$$

The following constraint makes sure that at least one of the $c'(v_i)$ constants is true at any time:

$$\textbf{caused } False \textbf{ if } \neg c'(v_1) \wedge \ldots \wedge \neg c'(v_n)$$

After changing the signature accordingly and making sure the functional behavior is preserved by the causal laws above, we can use $c'(v_i)$ instead of $c = v_i$ whenever needed in the Boolean representation.

$\mathcal{C}+$ **to causal logic**   While defining the semantics of $\mathcal{C}+$ in Section 2.3, we said that we could reduce a domain description in $\mathcal{C}+$ to a causal theory. To be precise, causal laws of the form (2.6), (2.7), and (2.8) can be transformed into causal rules of the form (2.9), (2.10), and (2.11) respectively.

In the resulting causal theory, some changes would help us further in the transformation to ASP. We can eliminiate any conjunctions in the head of causal rule such as

$$F \wedge G \Leftarrow H$$

by rewriting the rule as the following causal rules:

$$F \Leftarrow H$$
$$G \Leftarrow H$$

Similarly, any rule with disjunction in the body such as

$$F \Leftarrow G \vee H$$

can be eliminated by replacing it with following causal rules:

$$F \Leftarrow G$$
$$F \Leftarrow H$$

Using these changes, we can obtain a causal theory where the head of every rule is a literal, and the body of a rule does not contain any disjunctions. The resulting causal theory would be equivalent to the previous one, as described in [41, Proposition 4].

Causal logic to ASP   Once we obtain a causal theory as described above where the head of every rule is a literal and body is a conjunction, we can translate it into an ASP

program. Every causal rule of the form

$$l_0 \Leftarrow l_1, \ldots, l_n \tag{2.19}$$

can be translated into an ASP rule of the form

$$l_0 \leftarrow not\ \overline{l_1}, \ldots, not\ \overline{l_n} \tag{2.20}$$

where $\overline{l_i}$ denotes the contrary literal of l_i, and the complete answer sets of the resulting ASP program would be identical to the models of the initial causal theory, as described in [68, Proposition 6.7].

Simplifications in ASP   After obtaining a raw ASP formulation, we can apply some further simplifications using some special constructs mentioned above. For instance, instead of defining the exogeneity of a fluent f initially, as follows

$$f(0) \leftarrow not\ \neg f(0)$$
$$\neg f(0) \leftarrow not\ f(0)$$

we can make use of a choice rule as the following:

$$0\ \{f(0), \neg f(0)\}\ 1.$$

An even further simplification can be done for fluent atoms that are non-Boolean "implied functions" as described in [4, Corollary 2]. Some atoms may "act like a function" due to the rules and constraints effecting their value. For instance, a fluent atom $loc(obj, pos, t)$ denoting positions of objects effectively behaves like a function $loc : Obj \times Time \rightarrow Pos$ in a correct representation, since every object must have exactly one position any given time.

Let $f(v, t)$ be an implied function, and $val(v)$ denote the range of this implied function. We can represent the initial exogeneity of such a fluent by the following rule:

$$1\ \{f(v, 0) : val(v)\}\ 1.$$

Also, if we directly apply the transformation from $\mathcal{C}+$ to ASP as described above, the commonsense law of inertia for this fluent would be represented as follows:

$$f(v, t+1) \leftarrow not\ \neg f(v, t+1),\ not\ \neg f(v, t)$$
$$\neg f(v, t+1) \leftarrow not\ f(v, t+1),\ not\ f(v, t)$$

However, we can simplify this formulation by using the following rules instead:

$$\{f(v, t+1)\} \leftarrow f(v, t)$$
$$\leftarrow \{f(v, t) : val(v)\}\, 0$$
$$\leftarrow 2\, \{f(v, t) : val(v)\}$$

In this way, we can drop the classically negated literals of this sort of fluents in our formulations, and reduce the program size (i.e., number of ground atoms and rules).

For atoms denoting actions, we can also apply some simplifications to omit the classically negated literals. For instance, instead of defining the exogenity of an action a as follows

$$a(t) \leftarrow \textbf{\textit{not}}\ \neg a(t)$$
$$\neg a(t) \leftarrow \textbf{\textit{not}}\ a(t)$$

we can write the following choice rule:

$$0\, \{a(t)\}\, 1.$$

Example Let us transform Bomb Disposal domain in Figure 2.2 into an ASP formulation as described above. In the following, assume that the schematic variable l ranges over $\{left, right\}$, and t ranges over $\{0, 1\}$.

As explained above, the $\mathcal{C}+$ description of Figure 2.2 can be transformed into causal rules (2.1)-(2.5). Then we can express the exogeneity of initial values of fluents, (2.1), with the simplifications described above:

$$1\, \{up(l, 0), \neg up(l, 0)\}\, 1$$
$$1\, \{defused(0), \neg defused(0)\}\, 1$$

Declaration of action exogeneity in (2.2) is transformed into the following choice rule:

$$0\, \{flip(l, t)\}\, 1 \tag{2.21}$$

We transform the causal rules about commonsense law of inertia in (2.3) as follows:

$$up(l, t+1) \leftarrow \textbf{\textit{not}}\ \neg up(l, t+1),\ \textbf{\textit{not}}\ \neg up(l, t)$$
$$\neg up(l, t+1) \leftarrow \textbf{\textit{not}}\ up(l, t+1),\ \textbf{\textit{not}}\ up(l, t)$$
$$defused(t+1) \leftarrow \textbf{\textit{not}}\ \neg defused(t+1),\ \textbf{\textit{not}}\ \neg defused(t) \tag{2.22}$$
$$\neg defused(t+1) \leftarrow \textbf{\textit{not}}\ defused(t+1),\ \textbf{\textit{not}}\ defused(t)$$

The causal laws describing the direct effects of the flipping action in (2.4) are transformed

into ASP as follows:

$$up(l, t+1) \leftarrow flip(l, t), \neg up(l, t)$$
$$\neg up(l, t+1) \leftarrow flip(l, t), up(l, t)$$

(2.23)

Finally, the ramifications described in (2.5) are transformed into ASP as follows:

$$defused(t) \leftarrow up(left, t), up(right, t).$$

(2.24)

2.6 Automated Reasoners

2.6.1 CCALC

CCALC [41] is a reasoner of the action language $\mathcal{C}+$. It makes use of available transformations from $\mathcal{C}+$ to propositional logic, and then utilizes SAT solvers in the background, such as MINISAT [19]. To describe the input language of CCALC, we present $\mathcal{C}+$ description of Bomb Disposal in Figure 2.2 to CCALC, as shown in Figure 2.3.

When we present formulas to CCALC, conjunctions $\wedge$, disjunctions $\vee$, implications $\supset$, negations $\neg$, universal quantifiers $\forall$, and existential quantifiers $\exists$ are replaced with the symbols &, ++, ->>, -, /\, and \/ respectively. If we do not explicitly specify the domain of a constant, it is assumed to be Boolean by CCALC.

We can present planning problems in form of queries to CCALC. Figure 2.4 shows a simple planning problem in Bomb Disposal domain. In the initial state, bomb is not defused and the latches are facing downwards. We want to find the shortest plan that will defuse the bomb.

After presenting the domain description and planning problem to CCALC, the output shown in Figure 2.5 is obtained. It says that if we concurrently flip both latches at step 0, the bomb will be defused at step 1.

2.6.2 `iclingo`

`iclingo` [34] is an "incremental" ASP solver. Here, incremental means that the solver computes answer sets of a problem by gradually increasing the search space boundaries with respect to a predetermined variable. We present the ASP encoding of Bomb Disposal domain to `iclingo`, as shown in Figure 2.6.

Here, `#domain` is a solver directive that denotes the domain of a variable. Another directive, `#base`, is preceded before the rules that should be evaluated only in the initial step of incremental reasoning. On the other hand, directive `#cumulative t` denotes that the following rules should be evaluated for increasing values of `t` until an answer set is found.

We present `iclingo` the same planning problem that we have presented CCALC before, as shown in Figure 2.7. Here, rules succeeding `#volatile t` directive are evaluated

```
1  :- sorts
2    latch.
3
4  :- objects
5    left, right :: latch.
6
7  :- variables
8    L :: latch.
9
10  :- constants
11    flip(latch) :: exogenousAction;
12    up(latch) :: inertialFluent;
13    defused :: inertialFluent.
14
15  %% direct effects of flip action
16  flip(L) causes up(L) if -up(L).
17  flip(L) causes -up(L) if up(L).
18
19  %% ramification of flip action
20  caused defused if up(left) & up(right).
```

Figure 2.3: Presenting Bomb Disposal domain to CCALC

```
1  :- query
2    label :: 0;
3    maxstep :: 0..1;
4    0: -up(left), -up(right), -defused;
5    maxstep: defused.
```

Figure 2.4: Presenting a Bomb Disposal problem to CCALC

```
1  0:
2  ACTIONS:  flip(left)  flip(right)
3  1:  up(left)  up(right)  defused
```

Figure 2.5: A Bomb Disposal plan obtained from CCALC

20

```
1   #base.
2   %% objects
3   latch(left;right).
4
5   #cumulative t.
6   timea(0..t-1).
7   timef(0..t).
8
9   %% variables
10  #domain latch(L).
11  #domain timea(Ta).
12  #domain timef(Tf).
13
14  %% flip(latch), exogenous action
15  0 { flip(L,Ta) } 1.
16
17  %% up(latch), inertial fluent
18  1 { up(L,0), -up(L,0) } 1.
19  up(L,Ta+1) :- not -up(L,Ta+1), not -up(L,Ta).
20  -up(L,Ta+1) :- not up(L,Ta+1), not up(L,Ta).
21
22  %% defused, inertial fluent
23  1 { defused(0), -defused(0) } 1.
24  defused(Ta+1) :- not -defused(Ta+1), not -defused(Ta).
25  -defused(Ta+1) :- not defused(Ta+1), not defused(Ta).
26
27  %% direct effects of flip action
28  up(L,Ta+1) :- flip(L,Ta), -up(L,Ta).
29  -up(L,Ta+1) :- flip(L,Ta), up(L,Ta).
30
31  %% ramification of flip action
32  defused(Tf) :- up(left,Tf), up(right,Tf).
```

Figure 2.6: Presenting Bomb Disposal domain to iclingo

```
1  #base.
2  :- not -up(left,0).
3  :- not -up(right,0).
4  :- not -defused(0).
5
6  #volatile t.
7  :- not defused(t).
```

Figure 2.7: Presenting a Bomb Disposal problem to `iclingo`

```
1  Answer: 1
2  latch(left) latch(right) timea(0) timef(0) timef(1) flip(right,0)
3  flip(left,0) -up(right,0) -up(left,0) up(right,1) up(left,1) defused(1)
4  -defused(0)
5  SATISFIABLE
```

Figure 2.8: A Bomb Disposal plan obtained from `iclingo`

for increasing `t` values, and then the evaluations are discarded in the following reasoning steps.

The result obtained from `iclingo` is shown in Figure 2.8.

2.6.3 `dlvhex`

`dlvhex`[21] is an answer set solver that specializes on integration of external computational sources. Its input language is slightly different than `iclingo`'s. We present Bomb Disposal domain to `dlvhex` in Figure 2.9.

`dlvhex` does not provide directives for declaring dedicated variables, so we need to specify the domain of each variable in every single rule. `#int` is a special atom for representing nonnegative integers. The symbol v denotes *or*, the epistemic disjunction.

We present the same Bomb Disposal planning problem to `dlvhex`, as shown in Figure 2.10. Here, `#maxint` guides `#int` directive by mandating an upper limit. We look for plans of length 1, and we specify it by declaring an auxiliary atom, `const(t,1)`.

The result obtained from `dlvhex` is shown in Figure 2.11.

```
1   %% objects
2   latch(left).
3   latch(right).
4
5   timef(I) :- #int(I), const(t,C), I<=C.
6   timea(I) :- #int(I), const(t,C), I<C.
7
8   %% flip(latch), exogenous action
9   flip(L,Ta) v -flip(L,Ta) :-
10    latch(L), timea(Ta).
11
12  %% up(latch), inertial fluent
13  up(L,0) v -up(L,0) :-
14    latch(L).
15  up(L,Ti) :- not -up(L,Ti), not -up(L,Ta),
16    latch(L), timea(Ta), Ti=Ta+1.
17  -up(L,Ti) :- not up(L,Ti), not up(L,Ta),
18    latch(L), timea(Ta), Ti=Ta+1.
19
20  %% defused, inertial fluent
21  defused(0) v -defused(0).
22  defused(Ti) :- not -defused(Ti), not -defused(Ta),
23    timea(Ta), Ti=Ta+1.
24  -defused(Ti) :- not defused(Ti), not defused(Ta),
25    timea(Ta), Ti=Ta+1.
26
27  %% direct effects of flip action
28  up(L,Ti) :- flip(L,Ta), -up(L,Ta),
29    latch(L), timea(Ta), Ti=Ta+1.
30  -up(L,Ti) :- flip(L,Ta), up(L,Ta),
31    latch(L), timea(Ta), Ti=Ta+1.
32
33  %% ramification of flip action
34  defused(Tf) :- up(left,Tf), up(right,Tf),
35    timef(Tf).
```

Figure 2.9: Presenting Bomb Disposal domain to `dlvhex`

```
1  #maxint=1.
2  const(t,1).
3
4  :- not -up(left,0).
5  :- not -up(right,0).
6  :- not -defused(0).
7
8  :- not defused(C), const(t,C).
```

Figure 2.10: Presenting a Bomb Disposal problem to dlvhex

```
1  {latch(left),latch(right),-defused(0),const(t,1),-up(left,0),
2  -up(right,0),flip(right,0),flip(left,0),up(right,1),up(left,1),
3  timef(1),timef(0),timea(0),defused(1)}
```

Figure 2.11: A Bomb Disposal plan obtained from dlvhex

Chapter 3

Representing the Housekeeping Domain

In this chapter, we describe the housekeeping domain, and provide a detailed description of its representation in $\mathcal{C}+$. Then, we demonstrate how we make use of transformation techniques described in Section 2.5 to obtain a corresponding ASP formulation of the same domain.

3.1 Housekeeping Domain

Consider a house consisting of several rooms (e.g., bedroom, living room, kitchen). In each room there are some stationary obstacles (e.g., bed, sofa, wardrobe, table, bookshelf, tv stand), some movable objects (e.g., book, pillow, dish), and several autonomous robots in the house. Each room is assigned a group of robots; no robot is assigned to two different rooms. The goal is for the robots to relocate the movable objects so that the house becomes "tidy", i.e., every object is located where it should be.

The housekeeping robots are intelligent, autonomous, and rational in the sense that 1) they know where the movable objects belong to. For instance, a book should be placed on a bookshelf, or a dirty dish should be put in a dishwasher. 2) They can decide for optimal feasible plans to tidy a house by a given time, without colliding with objects. 3) They help each other when needed. Since some of the movable objects are heavier than the others, a single robot cannot carry such objects and require the assistance of another robot. In such cases, a robot may have to leave its current room, and travel to another place in the house to help others.

3.2 Representation of the Housekeeping Domain in $\mathcal{C}+$

We represent actions and change in the housekeeping domain in the action description language $\mathcal{C}+$, and compute optimal plans using reasoner CCALC. Let us first describe the representation as presented to CCALC. Full formulation can be found at Appendix A.
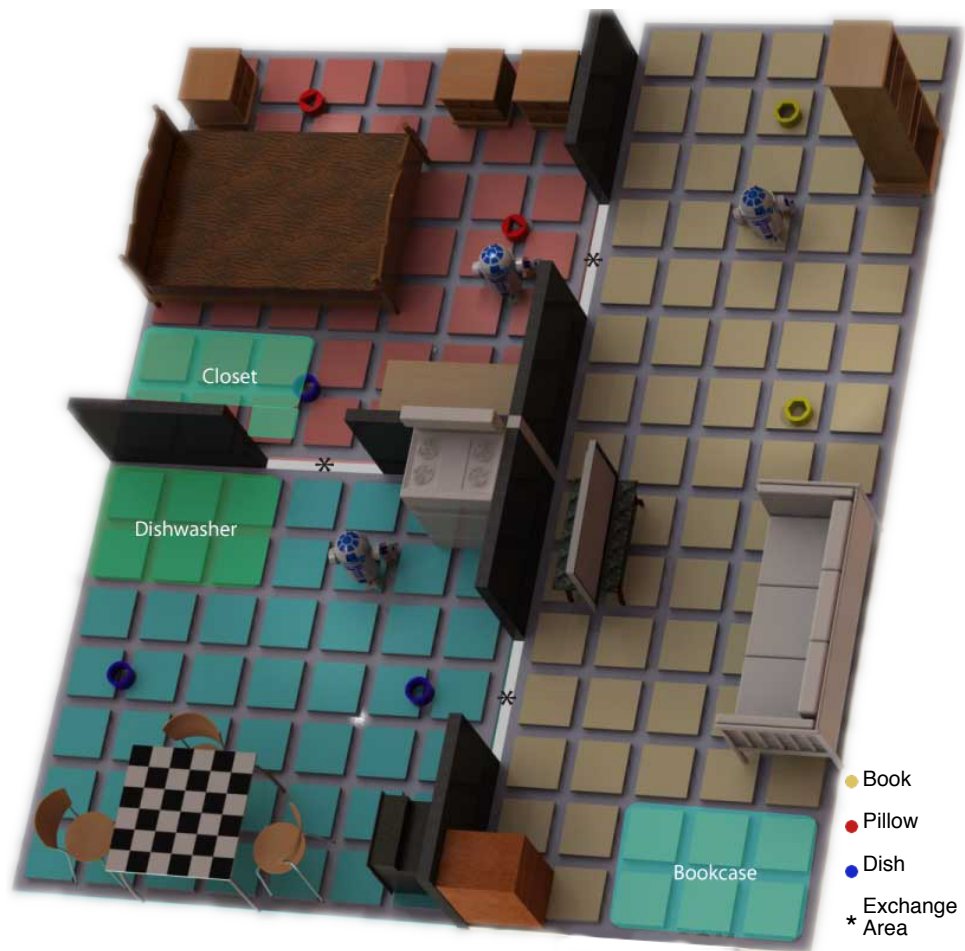
Figure 3.1: Housekeeping domain

Fluents and actions We view the house as a grid. We assume that robots and the end-points of objects are located at grid-points. We consider the fluents

- `at(TH,X,Y)` ("thing `TH` is at `(X,Y)`")

- `connected(R,EP)` ("robot `R` is connected to endpoint `EP`")

and the actions

- `goto(R,X,Y)` ("robot `R` goes to `(X,Y)`"),

- `detach(R)` ("robot `R` detaches from the object it is connected to"), and

- `attach(R)` ("robot `R` attaches to an object").

We add the commonsense law of inertia for every fluent (i.e., fluents are inertial), and express that the actions are exogenous while declaring fluent constants and action constants at the very beginning of the action description, as follows:

```
:- constants
  at(thing, x_coord, y_coord),
  connected(robot, endpoint) :: inertialFluent;
  goto(robot, x_coord, y_coord),
  detach(robot),
  attach(robot) :: exogenousAction.
```

Direct effects of actions We describe the direct effects of the actions above by causal laws of the form (2.12). For instance, the following causal law expresses the direct effect of the action of a robot `R` going to location `(X,Y)`:

```
goto(R,X,Y) causes at(R,X,Y).
```

Similarly, we can describe the direct effects of the action of a robot `R` detaching from the endpoints of an object it is connected to:

```
detach(R) causes -connected(R,EP) if connected(R,EP).
```

To describe the direct effects of the action of a robot `R` attaching to an endpoint of an object, we introduce an "attribute" `attach_point` of this action to show at which endpoint the robot is attaching.

```
:- constants
  attach_point(robot) :: attribute(endpoint) of attach(robot).
```

An attribute of an action is a useful feature of CCALC that allows us to talk about various special cases of actions without having to modify the definitions of more general actions. We can formalize the direct effect of attaching a payload ("robot `R` is connected to the endpoint `EP` of an object"):

```
attach(R) causes connected(R,EP) if attach_point(R)=EP.
```

Preconditions of actions   We describe effects of actions by causal laws of the form (2.13). For instance, we can describe that a robot R cannot go to a location (X,Y) if the robot is already at (X,Y), by the causal laws:

```
nonexecutable goto(R,X,Y) if at(R,X,Y).
```

To describe that a robot R cannot go to a location (X,Y) if that location is already occupied by a stationary object, we need to know in advance the locations of stationary objects in the house. Such knowledge is represented as the "background knowledge" in Prolog. CCALC allows to use the predicates defined as part of background knowledge, in causal laws, as follows:

```
nonexecutable goto(R,X,Y) where occupied(X,Y).
```

where occupied(X,Y) describe the locations (X,Y) occupied by stationary objects.

In general, the where parts in causal laws presented to CCALC include formulas that consist of "external predicates/functions". We will describe this term in more detail while elaborating on embedding commonsense knowledge and geometric reasoning into our domain.

Now let us present the preconditions of two other sorts of actions. Consider the action of a robot R attaching to an endpoint of an object. This action is not possible if the robot is connected to some endpoint EP of an object:

```
nonexecutable attach(R) if connected(R,EP).
```

Note that here we do not refer to the special case of the action of attaching via attributes. Also this action is not possible if the robot and the endpoint are not at the same location (X,Y):

```
nonexecutable attach(R) & attach_point(R)=EP
  if -[\/X \/Y | at(R,X,Y) & at(EP,X,Y)].
```

In the last line above, the negated expression stands for a disjunction of conjunctions at(R,X,Y) & at(EP,X,Y) over locations (X,Y).

Finally, we can describe that a robot R cannot detach from an object if it is not connected to any endpoint:

```
nonexecutable detach(R)
  if [/\EP | -connected(R,EP)].
```

the expression in the last line above stands for a conjunction of -connected(R,EP) over endpoints EP.

Ramifications We describe two ramifications of the action of a robot `R` going to a location `(X,Y)`. If the robot is connected to an endpoint of an object, then the location of the object changes as well:

```
caused at(EP,X,Y) if connected(R,EP) & at(R,X,Y).
```

Furthermore, neither the robot nor the endpoint are at their previous locations anymore:

```
caused -at(TH,X,Y) if at(TH,X1,Y1) where X\=X1 ++ Y\=Y1.
```

Here `TH` denotes a "thing" which can be either a robot or an endpoint.

Constraints We ensure that two objects do not reside at the same location by the constraint

```
caused false if at(EP,X,Y) & at(EP1,X,Y) where EP \= EP1.
```

and that a robot is not connected to two endpoints by the constraint

```
caused false
   if connected(R,EP1) & connected(R,EP)
   where EP \= EP1.
```

Some objects `OBJ` are large and have two endpoints `EP` and `EP1` one unit from each other. To be able to pick these objects, we ensure that the endpoints of the objects are located horizontally or vertically, and one unit apart from each other by the constraint:

```
caused false if at(EP1,X1,Y1) & at(EP2,X2,Y2)
   where belongs(EP1,OBJ) & belongs(EP2,OBJ)
   & Dist is sqrt((X1-X2)^2 + (Y1-Y2)^2)
   & EP1 \= EP2 & Dist \= 1.
```

Here `belongs(EP, OBJ)` is defined externally in predicate.

Finally, we need to express that a robot cannot move to a location and attach to or detach from an endpoint of an object at the same time.

```
nonexecutable goto(R,X,Y) & attach(R).
nonexecutable goto(R,X,Y) & detach(R).
```

3.3 Embedding Commonsense Knowledge into the Domain Description

In the housekeeping domain, the robots need to know that books are expected to be in the bookcase, dirty dishes in the dishwasher, and pillows in the closet. Moreover, a bookcase is expected to be in the living-room, dishwasher in the kitchen, and the closet in the bedroom. In addition, the robots should have an understanding of a tidy house to be able to clean a house autonomously: tidying a house means that the objects are at their desired locations. Also, while cleaning a house, robots should pay more attention while carrying fragile objects; for that they should have an understanding of what a fragile object is. Such commonsense knowledge is formally represented already in commonsense knowledge bases, such as CONCEPTNET.

CCALC allows us to extract and embed commonsense knowledge from these knowledge bases by means of "external predicates." External predicates are not part of the signature of the domain description (i.e., they are not declared as fluents or actions). They are implemented as functions in some programming language of the user's choice, such as C++ or Prolog. External predicates take as input not only some parameters from the domain description (e.g., the locations of robots) but also detailed information that is not a part of the action domain description (e.g., commonsense knowledge). They are used to externally check some conditions.

Expected locations of objects   We represent the expected locations of objects by a new fluent at_desired_location(EP) describing that an object EP is at its expected position in the house. Unlike the fluents above, we can define at_desired_location(EP) in terms of other fluents. This type of fluents are called "statically determined fluents". We declare this at_desired_location(EP) fluent as follows:

```
:- constants
  at_desired_location(endpoint) :: sdFluent.
```

After the declaration, the definition of the fluent is formalized as the following two causal law:

```
caused at_desired_location(EP) if at(EP,X,Y)
   where in_place(EP,X,Y).
default -at_desired_location(EP).
```

The second causal law expresses that normally the movable objects in an untidy house are not at their desired locations. The first causal law formalizes that the object EP is at its desired location if it is at some "appropriate" position (X,Y) in the right room. So,

the robots need to know that books are expected to be in the bookcase, dirty dishes in the dishwasher, and pillows in the closet. Moreover, a bookcase is expected to be in the living-room, dishwasher in the kitchen, and the closet in the bedroom. We describe such background knowledge externally as a Prolog program. For instance, the external predicate `in_place(EP,X,Y)` is defined as follows:

```
in_place(EP,X,Y) :- belongs(EP,Obj),
  type_of(Obj,Type), el(Type,Room),
  area(Room,Xmin,Xmax,Ymin,Ymax),
  X>=Xmin, X<=Xmax, Y>=Ymin, Y>=Ymax.
```

Here `belongs(EP,OBJ)`, `type_of(OBJ,Type)` describe the type `Type` of an object `Obj` that the endpoint `EP` belongs to, and `el(Type,Room)` describes the expected room of an object of type `Type`. The rest of the body of the rule above checks that the endpoint's location `(X,Y)` is a desired part of the room `Room`.

After defining `at_desired_location(EP)`, we can introduce a "macro" to define a tidy house:

```
:- macros
   tidy -> [/\EP | at_desired_location(EP)].
```

The second rule above expresses that the house is normally tidy. The first rule above describes the exceptions: when an object is not at its expected location, the house is untidy.

Acquiring commonsense knowledge   In order to acquire the knowledge of expected rooms of objects, which is represented by `el(Type,Room)`, we make use of existing commonsense knowledge bases. Specifically, we use CONCEPTNET.

CONCEPTNET [64] is a semantic network in which the nodes correspond to concepts (e.g., "human", "walking", etc.), and the edges denote relations (e.g. "capable of", "located near") between these concepts. Most of its data is obtained through Open Mind Common Sense Project [82] where thousands of volunteers manually enter trivial facts regarding the world, i.e., commonsense knowledge. The semantic network can be queried using reasoning techniques such as spreading action.

Using Python API of CONCEPTNET 4.0, we can easily query which objects are likely to be located at a specific room. The network provides a well-suited relation for our purpose, called "At Location". As the name itself suggests, it is a relation denoting the locations of objects.

For instance, we query the objects which are likely to be located in the bedroom, and automatically generate a list of facts as follows:

```
el(dresser, bedroom).
el(mirror, bedroom).
el(bed, bedroom).
el(pillow, bedroom).
el(closet, bedroom).
el(person, bedroom).
el(blanket, bedroom).
el(pillowcase, bedroom).
el(wardrobe, bedroom).
```

Figure 3.2: Commonsense knowledge about bedroom objects

```
result = Assertion.objects.filter( \
  relation=atLocation, concept2=bedroom, \
  score__gte=threshold)

for assertion in result:
  print 'el(%s, bedroom).' \
    % assertion.concept1.text
```

Here `atLocation` represents the "At Location" relation, and `bedroom` is representing the "Bedroom" concept. The query outcome denoted by `result` is a set of assertions. An assertion in CONCEPTNET is simply an object which includes two related concepts, and a relation connecting these two concepts. Assertions also have some intrinsic properties like the language of the assertion, and the frequency in which the given two concepts are related to each other by the given relation. "Score" is one of these intrinsic values which denotes the reliability of an assertion. In order to eliminate unreliable assertions, we filter out the ones with a score less than the value of `threshold`. The value of `threshold` is determined empirically, and is equal to 5 for this case.

After obtaining the query result, we simply represent it in Prolog using atoms of the form `el(Type,Room)`. Some assertions about what is expected in the bedroom are shown in Figure 3.2. We could have obtained a much longer list of facts if the `threshold` had been set to a lower value; but then we would have jeopardized the integrity of the facts. CONCEPTNET is generated automatically using the data gathered collaboratively by Open Mind Common Sense Project, and as a result, contains some unreliable knowledge. Still, eliminating the unreliability is possible as described above, with the help of the easy-to-use API of CONCEPTNET.

Note that the expected location of an object depends on where it is: for instance, the expected location of a book on the floor of the kitchen is the exchange area between the kitchen and the living room (so that the robot whose goal is to tidy the living

room can pick it up and put it in the bookcase); on the other hand, the expected location of a plate on the floor of the kitchen is the dishwasher. Therefore, the predicate `area(Room,Xmin,Xmax,Ymin,Ymax)` describes either an exchange area (if the object does not belong to the room where it is at) or a deposit area (if the object belongs to the room where it is at). Note also that knowledge about specific objects in a room as well as specific deposit and exchange areas are not common knowledge to all robots; each robot has a different knowledge of objects in their room.

3.4 Embedding Geometric Reasoning into Causal Planning

We can embed geometric reasoning in causal reasoning by making use of external predicates as well. For instance, suppose that the external predicate `path_exists(X,Y, X1,Y1)` is implemented in C++ utilizing Rapidly exploring Random Trees (RRTs) [59]; so it returns 1 if there is a collision-free path between `(X,Y)` and `(X1,Y1)`, and it returns 0 if there is no collision-free path between `(X,Y)` and `(X1,X1)`. Then, we can express that the robot `R` cannot go from `(X1,Y1)` to `(X,Y)` where `path_exists(X,Y,X1,X1)` does not hold, by the following action precondition:

```
nonexecutable goto(R,X,Y) if at(R,X1,Y1)
   where -path_exists(R,X1,Y1,X,Y).
```

Note that the parameters of the `path_exists(X,Y,X1,Y1)` external predicate are only the initial position `(X,Y)` and the goal position `(X1,Y1)`. In other words, we do not pass the positions of the movable objects, due to the limitations of CCALC. Therefore, this external predicate considers a relaxed version of the problem of finding a continous collision-free trajectory by considering only the stationary obstacles in the environment. Fortunately, we can overcome this limitation in our ASP encoding of housekeeping domain using the reasoner `dlvhex` which specializes on integration of external computation.

3.5 Representing Durative Actions

For multiple autonomous robots to complete cleaning a house by a given time, durations of actions should also be taken into account. Since the robots can help each other, when a help request is received, the robot should be able to autonomously decide whether she has enough time to complete her tasks and help the other robot by the given deadline. Since changing locations to be able to help each other takes some time, transportation delay (depending on the length of the continuous trajectory) should also be taken into account.

To represent duration of actions, we introduce another fluent `robot_time(R)` as follows:

```
:- constants
  robot_time(robot) :: simpleFluent(duration).
```

This fluent keeps the knowledge of "duration of a single action performed by the robot R". When a new action is performed by robot R, this value changes, and if no action is performed then the value of the fluent is not propagated to the next state. In other words, inertia is not a property of this fluent. Therefore, we declare this fluent not as an inertial fluent, but as a "simple fluent".

We define effects of actions on these fluents accordingly. For instance, the rule below expresses that the action of a robot R attaching to an object takes 1 unit of time:

```
attach(R) causes robot_time(R)=1.
```

Every attach performed by the robots is nearly identical; therefore, it is reliable to assign a constant value for the duration of this action. But the idea may not be applicable to other actions, such as changing the location of a robot. Since the path traveled by the robot is the main factor contributing to the duration of a movement of the robot, we tried to obtain an estimate by taking the length of the path into account using the following rule:

```
goto(R,X,Y) causes robot_time(R)=D if at(R,X1,Y1)
    where time_estimate(X1,Y1,X,Y,D).
```

Here `time_estimate(X1,Y1,X2,Y2)` is an external predicate implemented in C++. It calls a motion planner to find a trajectory from an initial position (`X1,Y1`) to a final position (`X2,Y2`), and then returns true if the duration estimate between 1 and 4 based on the total length of this trajectory is equal to `D`. Therefore, the rule expresses that if the robot R goes from one point to another, the execution time is proportional to the length of the path it follows.

There is no obligation that every robot should perform an action at every state. Therefore, we define the default value for the duration of an action to be 0:

```
default robot_time(R)=0.
```

While `robot_time(R,D)` estimates the duration of a single action, we also need a fluent to keep track of the total amount of time starting from the initial state. For that we introduce a fluent `elapsed_time(D)` as follows:

```
:- constants
  elapsed_time :: inertialFluent(duration).
```

This fluent keeps track of the time to reach a specific step of the plan, and we define it essentially by accumulating the durations of each action by the following causal law:

```
caused elapsed_time=(T+D)
   if D=robot_time(R), [/\R1 /\D1 | D1=robot_time(R1) ->> (D>=D1)]
   after T=elapsed_time
   where T+D =< timeLimit.
```

Since there can be multiple robots in the planning environment, more than one action can be performed concurrently. This causal law ensures that if concurrent actions happen, the elapsed time is incremented by the duration of the most time consuming action.

3.6 Representation of the Housekeeping Domain in ASP

We transform the $\mathcal{C}+$ representation of housekeeping domain at Appendix A into an ASP program as described in Section 2.5.

3.6.1 Presenting the Housekeeping Domain to `iclingo`

We represent the ASP encoding of the housekeeping domain in the input language of `iclingo` [34]. Full formulation translated from $\mathcal{C}+$ description without any simplification can be found at Appendix B. A further simplified formulation can be found at Appendix C. Below, we go over the simplified version.

Fluents and actions The signature is similar to the $\mathcal{C}+$ representation, but this time we need to explicitly denote state and transition identifiers in atoms corresponding to fluents and actions respectively. Therefore, a CCALC fluent such as `connected(R,EP)` is transformed into `connected(R,EP,T)` where "T" denotes the state. Similarly, an action constant such as `detach(R)` is now in the form of `detach(R,T)`, and this time "T" identifies a transition.

While we were able to define fluents as inertial in CCALC during the initial declaration, we need to explicitly specify the inertia property in ASP. For instance, inertia and initial exogeneity of `connected(R,EP,T)` fluent is defined as follows:

```
1 { connected(R,EP,0), -connected(R,EP,0) } 1.
connected(R,EP,Ta+1) :- not -connected(R,EP,Ta+1), connected(R,EP,Ta).
-connected(R,EP,Ta+1) :- not connected(R,EP,Ta+1), -connected(R,EP,Ta).
```

When we are transforming a functional fluent of CCALC formulation, we can apply further simplifications. For instance, we describe the inertia and initial exogeneity of `elapsed_time(D,T)` as follows:

```
0 { elapsed_time(D,0) } 1.
{ elapsed_time(D,Ta+1) } :- elapsed_time(D,Ta).
:- { elapsed_time(I,Tf) : duration(I) } 0.
:- 2 { elapsed_time(I,Tf) : duration(I) }.
```

Here, we save the formulation from some unnecessary atoms, specifically the negated literals -elapsed_time(D,T).

When we describe exogeneous actions, we again make use of cardinality expressions. For instance, we define the exogenity of goto(R,X,Y,T) as follows:

```
0 { goto(R,X,Y,Ta) } 1.
```

In this way, the simplified representation allow us to omit the negated literals.

Direct effects of actions We describe the direct effect of the action of robot R going to (X,Y) at step Ta as the following rule:

```
at(R,X,Y,Ta+1) :- goto(R,X,Y,Ta).
```

Similarly, the direct effect of detaching from an object can be represented as

```
-connected(R,EP,Ta+1) :- detach(R,Ta), connected(R,EP,Ta).
```

while the effect of attaching to an object can be described as the following:

```
connected(R,EP,Ta+1) :- attach(R,Ta), attach_point(R,EP,Ta).
```

Here, attach_point(R,EP,Ta) corresponds to an attribute of attach action in $\mathcal{C}+$ formalism. It is defined as follows:

```
1 { attach_point(R,Var,Ta) : endpoint(Var) } 1 :- attach(R,Ta).
```

Preconditions of actions Preconditions are represented using constraint rules of ASP. For instance, we say that "for a robot the to move, the destination should be different than its current position", using the following constraint rule:

```
:- goto(R,X,Y,Ta), at(R,X,Y,Ta).
```

Ramifications If the robot is holding an object, then the location of the object changes as the location of the robot changes. In that sense, the change of the location of the object is a ramification of the robot's action. ASP allows us to represent this ramification as follows:

```
at(EP,X,Y,Tf) :- connected(R,EP,Tf), at(R,X,Y,Tf).
```

Constraints   We have showed that we use constraints rules to represent action preconditions which can also be described as transition constraints. Similarly, the solution for representing state constraints is constraint rules of ASP. For instance, we say that two different objects cannot be present at the same grid point at the same time using the following constraint rule:

```
:- at(EP1,X,Y,Tf), at(EP2,X,Y,Tf), EP1!=EP2.
```

Embedding commonsense knowledge   Similar to our $\mathcal{C}+$ representation, we make use of some external predicates. In order to describe if an object is at a desired location, we say the following:

```
at_desired_location(EP,Tf) :- at(EP,X,Y,Tf), in_place(EP,X,Y).
```

Here, `in_place(EP,X,Y)` is an external predicate that shares the same functionality with the predicate of the same name in our $\mathcal{C}+$ formulation, i.e., the predicate checks if an object is located in an appropriate location based on the commonsense knowledge extracted from CONCEPTNET. Using `at_desired_location(EP,T)` atom, we define tidiness as follows:

```
tidy(Tf) :- at_desired_location(I,Tf) : endpoint(I).
```

Embedding geometric reasoning   Here is another example of an external predicate that decides existence of a collision-free trajectory which is used while describing an action precondition:

```
:- goto(R,X2,Y2,Ta), at(R,X1,Y1,Ta),
   @path_exists(X1,Y1,X2,Y2)==0.
```

In this way, geometric reasoning is embedded into ASP reasoning.

Representing durative actions   We represent the duration of an `attach(R,T)` action as follows:

```
robot_time(R,1,Ta+1) :- attach(R,Ta).
```

In order to describe the duration of a `goto(R,X,Y,T)` action, which varies with respect to the distance traveled by the robot, we use an external function similar to our $\mathcal{C}+$ representation, as follows:

```
robot_time(R,Da,Ta+1) :- goto(R,X2,Y2,Ta), at(R,X1,Y1,Ta),
   Da:=@time_estimate(X1,Y1,X2,Y2).
```

3.6.2 Presenting the Housekeeping Domain to `dlvhex`

`dlvhex` is an ASP solver that specializes on integration of external computation into the reasoning process [21]. Here we provide some details regarding our housekeeping domain in the input language of `dlvhex`, and the integration of high-level task planning and low-level geometric reasoning. Full formulation can be found at Appendix D.

Integration of task planning and geometric reasoning   As we have mentioned earlier while describing our CCALC and `iclingo` formulations, we use some external predicates to represent several low-level constraints. For instance, one of the external predicates in `iclingo` formulation is in the form of `@path_exists(X1,Y1,X2,Y2)`, and it checks the existence of a collision-free trajectory from the grid point `(X1,Y1)` to `(X2,Y2)`. As you can see, the only parameters taken into consideration by this predicate are the initial and goal position of the robot. The positions of movable objects denoted by the `at(EP,X,Y,T)` atoms are not given to the external predicate. This is due to the limitations of external computation interface of `iclingo`, and is not much of a choice. Unfortunately, CCALC suffers from the same limitations as well. Therefore, this external predicate in `iclingo` formulation and corresponding one in CCALC formulation solve a relax version of the motion planning problem where the stationary obstacles are considered, but the movable objects are omitted.

One might suggest changing the predicate to some other form such as

$$path\_exists(rx_1, ry_1, \ldots, rx_m, ry_m, x_1, y_1, \ldots, x_n, y_n)$$

where (x_i, y_i) corresponds to the position of ith object and (rx_j, ry_j) denotes the position of the jth robot, so that we can pass movable object positions to the predicate as well. Unfortunately, this trivial approach suffers from some serious problems. First of all, it requires fixing the number of movable objects and robots in the domain, or at least mandating an upper limit. This damages the elaboration tolerance of the logical representation. Secondly, even if we make some sacrifices in the representation and move accordingly, now the reasoners would have a hard time grounding such a predicate with so many parameters. It would require the reasoner to ground the predicate and the associated rules for nearly every possible combination of object and robot positions. Even with a modest upper limit to the number of objects and a small grid, the grounding mechanisms these solver employ make it impossible for a regular computer or a workstation in today's standards to deliver a result in a feasible time.

Fortunately, `dlvhex` provides a more versatile interface and reasoning mechanism for external computations that does not include the short-comings of `iclingo` and CCALC. In `dlvhex`, we can pass a set of atoms as a parameter to the an external predicate. This second-order property of `dlvhex` predicates allows us to integrate the actual motion plan-

ning problem with all the movable objects instead of the relaxed version. The following is the corresponding rule that describes a precondition of `goto(R,X,Y,T)` action:

```
:- goto(R,X2,Y2,Ta), at(R,X1,Y1,Ta),
   not &path_exists[X1,Y1,X2,Y2,at,Ta](),
   robot(R), xcoord(X1), xcoord(X2),
   ycoord(Y1), ycoord(Y2), timea(Ta).
```

Here, the external predicate `&path_exists[X1,Y1,X2,Y2,at,Ta]()` has a parameter named `at` which corresponds to the set of `at(TH,X,Y,T)` atoms. In this way, the predicate is able to include the movable object positions into its geometric reasoning process.

Note that `dlvhex` also does not suffer from the performance penalties of computing the external predicate for unnecessary instantiations. There is an interleaving between the grounding (which includes computation of external predicates) and solving. Therefore, the costly computations of external predicates are omitted as much as possible.

3.7 Further Use of Commonsense Knowledge

Besides the knowledge about expected locations of objects, we can integrate other sorts of commonsense knowledge to render the robots more informed about their surroundings. After all, our method of embedding commonsense knowledge is general enough to embed all sorts of commonsense knowledge. With new sorts of knowledge, reasoning capabilities of the robots can be improved.

For instance, we can extract from the commonsense knowledge base CONCEPTNET the types of objects has property of being fragile as follows:

```
result = Assertion.objects.filter(relation=hasProperty, \
  concept2=fragile, score__gte=threshold)
for assertion in result:
  print 'has_property(%s, fragile)' % assertion.concept1.text
```

and obtain a list of object sorts that are likely to be fragile. Afterwards, we can define an external predicate to specify the endpoints of fragile objects:

```
fragile(EP) :- belongs(EP,Obj),
  type_of(Obj,Sort),
  has_property(Sort,fragile).
```

and then describe that fragile objects should be handled more carefully by adding an "attribute" to the action of attaching to an object:

$$handle\_with\_care(r, t) \leftarrow$$
$$attach(r, t), attach\_point(r, ep, t), fragile(ep).$$

Exceptions to commonsense knowledge There may be some exceptions to the commonsense knowledge extracted from the commonsense knowledge bases. For instance, books normally are expected to be in the living room; however, cookbooks are exceptions (since they are expected to be in the kitchen). Such exceptions can be represented in $\mathcal{C}+$and ASP in a natural way, since the formalisms allow representation of "defaults". To be able to handle such exceptions about the locations of objects, we need to replace the rule

$$at\_desired\_location(ep, t) \leftarrow$$
$$at(ep, x, y, t), in\_place(ep, x, y)$$

with the rule

$$at\_desired\_location(ep, t) \leftarrow$$
$$at(ep, x, y, t), in\_place(ep, x, y), \boldsymbol{not}\ exception(ep)$$

and say that cookbooks are exceptions:

$$exception(ep) \leftarrow belongs(ep, obj),$$
$$type\_of(obj, cookbook)$$

3.8 Heterogenous Robots

In our description of housekeeping domain, all robots are homogenous, i.e. they share the same capabilities, perform the same set of actions. But this situation may not be the case in another scenario. For instance, some of the robots may be able to pick up objects from the table, but not from the floor due to their short robotic arms. Or another type of robot may not be able to carry heavy objects even with the help of another robot. To represent such a domain which includes different types of robots, we can define detailed action preconditions in ASP or $\mathcal{C}+$. For instance, the following is an example of preventing short-armed robots from picking up objects not located on the table:

$$\leftarrow attach(r, t),\ attach\_point(r, ep, t),$$
$$\boldsymbol{not}\ coarse\_location(ep, table, t),\ has\_property(r, ``short\ arm'')$$

Similarly, we can prevent weak robots from carrying heavy objects as follows:

$$\leftarrow attach(r, t), attach\_point(r, ep, t),$$
$$heavy(ep),\ has\_property(r, "light\ weight")$$

While concluding this chapter, we would like to emphasize one more time that by representing the domain in $\mathcal{C}+$ and in ASP, we could utilize 1) the expressivity of these formalisms (e.g., defaults, choice rules), and 2) the relevant computational methods (i.e.,

external predicates) to handle the following two challenges of the housekeeping domain: i) embedding commonsense knowledge, ii) embedding geometric reasoning.

Chapter 4

Reasoning about the Housekeeping Domain

The goal behind logic-based representation of the housekeeping domain is to be able to reason on the resulting formulation via various solvers. The specific reasoning task that we are interested in is mainly planning. Here we describe how we formulate planning problems in these logical formalisms, and provide some experimental results on solver performances.

4.1 Planning with CCALC and `iclingo` in the Housekeeping Domain

CCALC   We describe planning problems in CCALC in terms of queries. Figure 4.1 is an example query in the input language of CCALC.

In this query, `maxstep` is a variable denoting the last step of the plan. It can be assigned to a single non-negative integer or a range of non-negative integers. When it is the latter, the value of `maxstep` is initialized with the lower limit, and incremented unless a plan is found with the current value or the upper limit is reached. In this way, it can be ensured that the resulting plan is of minimum length by setting the lower limit to 0.

```
1  :- query
2    label :: 0;
3    maxstep :: 0..20;
4    0: at(r1,4,6), at(ep1,3,5), elapsed_time=0, free;
5    maxstep: tidy, free.
```

Figure 4.1: A CCALC query for a housekeeping problem

```
%% initial state
#base.
:- not at(r1,4,6,0).
:- not free(0).
:- not at(ep1,3,5,0).
:- not elapsed_time(0, 0).

%% goal condition
#volatile t.
:- not tidy(t).
:- not free(t).
```

Figure 4.2: An `iclingo` query for a housekeeping problem

A planning problem consists of an initial state and a goal condition. In Figure 4.1, the initial state is described by the fluents in fourth line. It is the complete description of the initial state that consists of positions of the robots, positions of the objects, initial time, and availability of robot end effectors. The fifth line describes the goal condition of the plan. We say that, at the end of the plan the room should be tidy, and the robot end effectors should be free.

`iclingo` Similar to CCALC, planning problems are represented in the form of queries in ASP. In this case, a query is basicly a set of constraints. The ASP query corresponding to the planning problem in Figure 4.1 is given in Figure 4.2 in the input language of `iclingo`.

In Figure 4.2, `#base` and `#volatile` are solver directives. When we want to evaluate a group of rules only once in the base step of incremental reasoning, we precede them with `#base`. That is the case with the constraint rules which represent the initial state of the planning problem, therefore we use this directive.

The constraint rules following the initial state description represent the goal condition of the planning problem. We want the goal condition to be checked at each step of the incremental reasoning, but we do not want to propagate any unsatisfiable instantiations of the these rules into to the later steps. Therefore, we precede them with the `#volatile t` directive. In this way, the goal condition is evaluated for each `t` value incrementally without minding the satisfiability for prior `t` values.

4.2 Planning with Complex Goals

One of the advantages of having a logical representation of the planning domain is to be able to represent some complex goals using logical formulas. Here we provide some

```
:- query
  label :: 0;
  maxstep :: 0..20;

  %% initial state
  0: at(r1,4,6);
  0: at(ep1,3,5);
  0: elapsed_time=0:
  0: free;

  %% goal condition
  maxstep: tidy;
  maxstep: free;
  maxstep: elapsed_time=<30.
```

```
%% initial state
#base.
:- not at(r1,4,6,0).
:- not free(0).
:- not at(ep1,3,5,0).
:- not elapsed_time(0, 0).

%% goal condition
#volatile t.
:- not tidy(t).
:- not free(t).
:- elapsed_time(D,t), D>30.
```

Figure 4.3: CCALC and `iclingo` queries with deadlines

examples of these complex goals used in housekeeping planning.

As we have said earlier, incremental plan search ensures that the resulting plan would be of minimum length in CCALC and `iclingo`. But minimum plan length does not neccessarily mean the least possible plan duration. Plans with the minimum length may differentiate between each other in terms of duration. Also, a plan with fewer steps may take more time than another plan more steps. While length of a plan is an abstract term, duration of a plan is more meaningful from the practical point of view. People would care about giving a deadline to a robot to tidy a room, but the number of steps taken to complete the task is mostly irrelevant. Here we show how we can apply a deadline in a query in Figure 4.3.

Consider a scenario with a robot, a small object, and a heavy object which require collaboration between two robots to be carried around. Since a single robot cannot carry the heavy object, the robot need to find a new plan that involves two robots, and call another robot for help to execute it. But that robot is likely to be busy working on another room. It would not be wise if the resulting plan involves some collaborative actions in the beginning, handling of the small object which does not require another robot in the middle, and then another sequence of collaborative actions in the end. A good practice would be postponing the handling of the heavy object as much as possible. Fortunately, we can write down such a complex query to obtain a desired plan quite easily. For instance, we force the plan to not have any actions involving the helper robot r2 before the last 4 steps of the plan using the queries provided in Figure 4.4. In this way, concurrent actions are postponed as well as the handling of the heavy object.

We can also prevent existence of some actions under specific circumstances from the desired steps of the plan in our queries. This kind of constraints are especially useful in case of replanning after encountering plan failures. For instance, assume that we have

```
:- query                                    %% initial state
  label :: 0;                               #base.
  maxstep :: 4..20;                         :- not at(r1,3,5,0).
                                            :- not at(r2,4,5,0).
  %% initial state                          :- not free(0).
  0: at(r1,3,5);                            :- not at(ep1,3,5,0).
  0: at(r2,4,5);                            :- not at(ep2,2,5,0).
  0: at(ep1,3,5);                           :- not at(ep3,3,6,0).
  0: at(ep2,2,5);                           :- not elapsed_time(0,0).
  0: at(ep3,3,6);
  0: elapsed_time=0;                        %% goal condition
  0: free;                                  #volatile t.
                                            :- not tidy(t).
  %% goal condition                         :- not free(t).
  maxstep: tidy;
  maxstep: free;                            %% temporal constraint
                                            :- 1 { goto(r2,I,J,0..(t-4))
  %% temporal constraint                      : xcoordinate(I)
  (S @< (maxstep - 4)) ->>                     : ycoordinate(J) }.
    (S: [/\X /\Y | -goto(r2,X,Y),          :- 1 { attach(r2,0..(t-4)) }.
      -attach(r2), -detach(r2)]).           :- 1 { detach(r2,0..(t-4)) }.
```

Figure 4.4: CCALC and `iclingo` queries with temporal constraints

a plan in which the execution fails unexpectedly when robot `r1` tries to go to `(3,6)` point from `(2,5)` in the grid. This may be due to human interference, or some unknown object blocking the path, etc. Since we do not know the exact reason, one of the things we can do is to put a constraint that prevents existence of this action in the plan, so that we may work around the problem. We can do this by adding the following constraint into a query in `iclingo`:

```
:- goto(r1,2,5,T), at(r1,3,6,T).
```

We can also append similar constraints to our query for other failures encountered during the execution cumulatively. In this way, the robot would learn from its mistakes, and the resulting plan would be more likely to succeed.

4.3 Hybrid Planning

Since geometric reasoning and temporal reasoning (via a motion planner) are embedded in the computation, the calculated plans are essentially hybrid plans, integrating discrete planning and continuous motion planning. Hybrid plans help computation of plans that are geometrically feasible as well as temporally feasible. Let us show these two advantages of hybrid planning with examples.
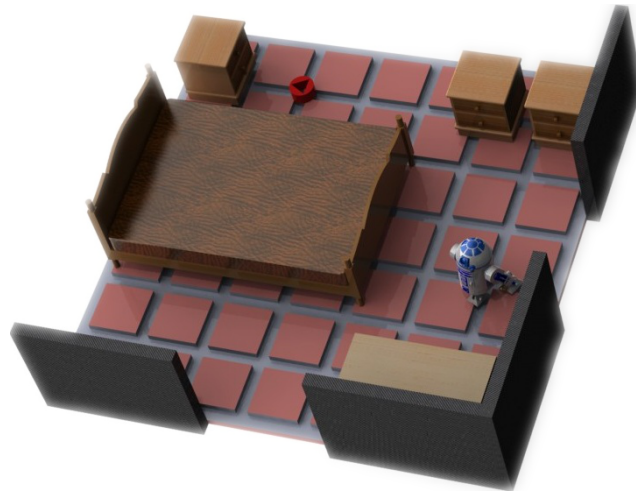
Figure 4.5: Housekeeping domain for Scenario 1

Consider Scenario 1, depicted in Figure 4.5, in which one of the nightstands together with the bed forms a narrow passage possibly blocking the robot to reach the north part of the bedroom where the red pillow is located. Note that this kind of geometric feasibility checks can only be performed using full geometric models of the robot and the room at a continuous level, and cannot be abstracted at the grid level without loosing completeness. Therefore, without geometric reasoning, naive task planning may lead to infeasible plans.

Indeed, at the grid-level, preconditions prohibit the robot to go to the locations occupied by the robot itself and occupied by the stationary objects; so, in Scenario 1, the robot can go to where the red pillow is, and given the planning problem in Figure 4.6, naive task planning using `iclingo` without geometric feasibility checks computes the geometrically infeasible plan in Figure 4.7.

However, our hybrid planning approach identifies that such a planning problem is

```
%% initial state
% Robot 1 is at (4,6) in bedroom
:- not at(r1,4,6,0).
% Robot 1 is free
:- not free(0).
% red pillow is (1,2) in bedroom
:- not at(redpillow1,1,2,0).
% initially elapsed time is 0
:- not elapsed_time(0,0).

%% goal
:- not tidy(t). % the bedroom is tidy
:- not free(t). % Robot1 finished cleaning
```

Figure 4.6: Planning problem for Scenario 1

```
% go to where the red pillow is
0. goto(r1,1,2,0)
% pick the red pillow
1. attach(r1,1)
% go to where pillow is expected to be
2. goto(r1,5,1,2)
% put the red pillow
3. detach(r1,3)
```

Figure 4.7: An infeasible plan for Scenario 1

infeasible (i.e., `iclingo` returns that no solution exists) thanks to the geometric reasoning embedded into high-level representation.

The second example shows the usefulness of embedding temporal reasoning in planning, by means of estimating the durations of actions based on the length of paths (computed by motion planners), to identify infeasible plans due to temporal constraints. Consider Scenario 2, described by the planning problem in Figure 4.8. We compute two shortest plans (of length 16) for this problem using `iclingo`: one without any temporal constraints, and one with the temporal constraint where the total duration of the plan is limited to 25 units of time. For the latter problem we add the following line to the planning problem description above:

```
% total elapsed time is <= 25 units
:- elapsed_time(D,t), D>25.
```

Table 4.1 presents these two plans. Note that, even though both plans last 16 time steps at the discrete level, the plan computed with the duration estimation obeys the temporal constraint of 25 time units, while the plan computed without duration estimation lasts for 26 time units, violating the temporal constraint. Indeed, according to the continuous trajectories computed for each "go" action, the total length of the trajectory for the plan computed without temporal constraints is 55 units, whereas the total length of the trajectory for the plan computed with temporal constraints is 51 units; assuming a constant velocity of 3 units leads to duration estimates of 26 and 25 units respectively.

Note that the hybrid planning approach can also be extended to allow for collaborative actions of multiple housekeeping robots. In particular, consider Scenario 3, a variation of Scenario 2 where there exists a second robot that is available to help Robot 2 in the kitchen. A collaborative plan for these two robots calculated by `iclingo` is presented in Table 4.2. Note that the collaboration of robots decrease the total plan duration to 14 time units.

```
%% initial state
% Robot 2 is at (2,2) in kitchen
:- not at(r1,2,2,0).
% Robot 2 is free
:- not free(0).
% location of plate1 is (5,5) in kitchen
:- not at(plate1,5,5,0).
% location of fork1 is (1,5) in kitchen
:- not at(fork1,1,5,0).
% location of spoon1 is (7,6) in kitchen
:- not at(spoon1,7,6,0).
% location of mug1 is (7,5) in kitchen
:- not at(mug1,7,5,0).
% initially elapsed time is 0
:- not elapsed_time(0,0).

%% goal
:- not tidy(t). % the kitchen is tidy
:- not free(t). % Robot1 finished cleaning
```

Figure 4.8: Planning problem for Scenario 2

Table 4.1: Plans for Scenario 2

Time Step	Plan with duration estimation	Elapsed Time	Plan without duration estimation	Elapsed Time
0	goto(r2,1,5,0)	1	goto(r2,7,6,0)	2
1	attach(r2,1)	2	attach(r2,1)	3
2	goto(r2,0,3,2)	3	goto(r2,2,3,2)	5
3	detach(r2,3)	4	detach(r2,3)	6
4	goto(r2,5,5,4)	6	goto(r2,7,5,4)	9
5	attach(r2,5)	7	attach(r2,5)	10
6	goto(r2,1,3,6)	9	goto(r2,0,0,6)	13
7	detach(r2,7)	10	detach(r2,7)	14
8	goto(r2,7,5,8)	13	goto(r2,5,5,8)	17
9	attach(r2,9)	14	attach(r2,9)	18
10	goto(r2,0,0,10)	17	goto(r2,1,2,10)	20
11	detach(r2,11)	18	detach(r2,11)	21
12	goto(r2,7,6,12)	21	goto(r2,1,5,12)	22
13	attach(r2,13)	22	attach(r2,13)	23
14	goto(r2,2,3,14)	24	goto(r2,1,0,14)	25
15	detach(r2,15)	25	detach(r2,15)	26

Table 4.2: Collaborative Plan for Scenario 3

Time Step	Plan with two robots	Elapsed Time
0	`goto(r2,7,5,0) goto(r1,7,6,0)`	3
1	`attach(r2,1) attach(r1,1)`	4
2	`goto(r2,1,0,2) goto(r1,0,3,2)`	7
3	`detach(r2,3) detach(r1,3)`	8
4	`goto(r2,5,5,4) goto(r1,1,5,4)`	10
5	`attach(r2,5) attach(r1,5)`	11
6	`goto(r2,1,3,6)`	13
7	`detach(r2,7) detach(r1,7)`	14

4.4 Experimental Evaluation

We have represented the housekeeping domain using different formalisms, $\mathcal{C}+$ and ASP. Also, for each ASP reasoner that we use, we have provided slightly different ASP formulations that make use of the differents aggregates or interfaces available. To benchmark the performance of these different approaches, we run some tests with respect to a set of planning problems. Here we provide these experimental results.

In these experiments, we use the domain representation in $\mathcal{C}+$ provided in Appendix A, and the representations in ASP provided in Appendix B, C and D. For the experiments on $\mathcal{C}+$ representation, we use CCALC 2.0 with SWI-PROLOG 5.10.4 and MINISAT 2.0. As for the ASP solvers, we use `iclingo` 3.0.3 and a development release of `dlvhex` 2.1.0. The experiments are done on a workstation with quad core Intel(R) Xeon(R) CPU E5310 @ 1.60GHz CPU and 6GB RAM. During the experiments, relevant measurements are made for each instance only once, since the high-level reasoning is deterministic, and the embedded external computation of geometric reasoning is also deterministic in practice due to the precomputation and caching techniques we have employed. Therefore, multiple runs of the same instance cannot be subject to significant differences.

In the following tables and figures, `iclingo*` denotes the results of simplified ASP formulation in Appendix C. In the meantime, `iclingo**` denotes the results obtained by applying a simple, equivalence preserving transformation to the `iclingo*` formulation. This transformation is a very simple yet highly effective workaround for reducing the grounding time of external predicates/functions that do not change their value with respect to plan steps. The following rules describe this transformation:

```
#base.
path_exists_base(X1,Y1,X2,Y2) :- @path_exists(X1,Y1,X2,Y2)==1.
time_estimate_base(X1,Y1,X2,Y2,D) :- D:=@time_estimate(X1,Y1,X2,Y2).
```

After adding these rules, we replace every occurence of external predicates/functions with

Table 4.3: Planning Experiment Problem Details

(R:Robots, O:Objects, L:Plan Length)

	Properties			Program Size				
No	R	O	L	CCALC (Atoms/Clauses)	iclingo (Atoms/Rules)	iclingo* (Atoms/Rules)	iclingo** (Atoms/Rules)	dlvhex (Atoms/Rules)
1	1	2	8	37450 287390	8231 300890	3983 156468	8821 188154	- -
2	1	3	12	58300 490819	13780 539308	7402 326850	12240 371960	- -
3	1	1	4	18308 127031	3400 117588	1489 48029	6327 66291	39411 74578
4	2	2	4	37389 260721	5791 230786	2735 154365	7573 186051	- -
5	2	3	7	66929 485297	10646 442333	5452 374325	10290 426147	- -
6	2	4	8	79391 606447	14191 592562	7175 548117	12013 606651	- -
7	2	5	11	111717 890051	20759 893773	11120 920388	15960 999084	- -
8	2	6	12	124805 1041313	24369 1077892	13191 1187415	16846 1167253	- -
9	3	6	8	125068 951336	19755 894034	10223 1176233	15065 1261655	- -
10	4	6	8	166723 1203111	22761 1046796	11941 1560487	15292 1464664	- -
11	5	6	4	105198 756694	13995 648750	7147 974601	11989 1046583	- -
12	5	6	7	182583 1285138	22824 1061856	11971 1702116	11989 1046583	- -
13	6	8	7	229921 1640663	28136 1344063	14894 2803970	19730 2808806	- -
14	4	7	8	170800 1263400	24152 1126025	12672 1907820	17508 1912656	- -
15	4	8	8	174877 1324877	25537 1206500	13397 2148943	18233 2153779	- -
16	5	7	7	187087 1342416	24107 1133597	12649 2080506	17485 2085342	- -
17	5	8	8	218598 1591766	28647 1362278	15173 2676229	20009 2681065	- -

the newly created atoms to obtain `iclingo**` formulation. In this way, the costly computation of these external predicates/functions are restricted to the base step of incremental reasoning.

The experiment consists of 17 different planning problems of various size. In each problem, there are some movable object of different sorts in an untidy condition, and a varying number of robots are present in the same room. The goal condition of each plan is to obtain a tidy room with unoccupied robots, and meet a reasonable deadline in the meantime. All instances are satisfiable. More detail about the number of robots, number of movable objects, and the minimum plan length is provided in Table 4.3.

Program size   At first, we provide some measurements for the size of each problem. In Table 4.3, program size is given in terms of atom count and rule count for `iclingo` and `dlvhex` formulations. For CCALC formulation, we give the count of atoms and clauses. In case of `iclingo`, these values quantifiy the whole process due the incremental reasoning. For the other formulations, these counts correspond to the last and therefore the largest step of each planning problem.

When we examine Table 4.3, we see that `iclingo` internally produces significantly less number of atoms than CCALC or `dlvhex` does. Also, the simplifications we have applied on the `iclingo` formulation contributed towards descreasing the size of the problem.

Computation time   In Table 4.4 and Figure 4.9, we show the computation time it takes to find a plan for each problem using different reasoner-formulation combinations. The first thing that strikes is that `dlvhex` timeouts in every problem instance except the smallest one which is Problem 3 that includes a single robot and a single movable object. The computation time given to `dlvhex` processes before forcing them to terminate is 6000 seconds.

Despite the poor performance in general, `dlvhex` outperforms all the other reasoners with a great margin in the only planning problem it can solve. When we look at this Problem 3 more closely, we see that all the reasoners except `dlvhex` spend nearly all of their time grounding this problem. This is due to the fact that these reasoners unnecessarily compute every possible external predicate ground atom. Even worse, the external predicates that we use in CCALC and `iclingo` formulations are responsible for a relaxed version of the problems which do not include the movable objects. Therefore, their values are actually time independent. Unfortunately, CCALC and `iclingo` compute the ground atoms of these predicates at each incremental step unnecessarily except `iclingo**` formulation thanks to the workaround, and this situation results in a huge performance penalty. Thus, we can say that the inverleaving between grounding and solving employed by `dlvhex` saves a lot of time in this instance. Note that the external predicates

51

in `dlvhex` formulation solve a harder problems by considering the movable objects, and still there exists a huge performance difference in favor of `dlvhex`.

Nevertheless, it is certain that `dlvhex` suffers from some serious scalability problems since it can solve only the smallest instance in a reasonable time. This may be related with the fact that `dlvhex` reasoning process takes into account significantly more atoms and rules than `iclingo` does. The huge difference can be seen in Table 4.3 for Problem 3.

Table 4.4: Planning Time

(G:Grounding, C:Completion, Sh:Shifting, Pr:Preprocessing, S:Solving)

No	CCALC (s)	iclingo (s)	iclingo* (s)	iclingo** (s)	dlvhex (s)
	G:66.74	G:96.95	G:92.11	G:14.08	
	C:18.78 Sh:0	Pr:0.29	Pr:0.52	Pr:0.40	
	S:3.63	S:0.25	S:0.36	S:0.28	
1	**98.64**	**97.49**	**93.04**	**14.80**	**t/o**
	G:68.53	G:140.98	G:142.50	G:15.05	
	C:16.55 Sh:.02	Pr:0.59	Pr:1.17	Pr:1.05	
	S:11.93	S:2.63	S:2.38	S:1.55	
2	**110.82**	**144.22**	**146.15**	**17.81**	**t/o**
	G:58.59	G:47.56	G:46.65	G:13.29	
	C:14.38 Sh:0	Pr:0.12	Pr:0.14	Pr:0.11	G:1.23
	S:1.17	S:0.00	S:0.01	S:0.01	S:0.75
3	**78.60**	**47.68**	**46.80**	**13.41**	**4.76**
	G:121.07	G:94.32	G:91.93	G:14.60	
	C:27.20 Sh:.01	Pr:0.26	Pr:0.53	Pr:0.42	
	S:2.75	S:0.04	S:0.06	S:0.03	
4	**158.35**	**94.63**	**92.53**	**15.05**	**t/o**
	G:127.19	G:166.92	G:162.76	G:15.47	
	C:38.44 Sh:0	Pr:0.56	Pr:1.45	Pr:1.24	
	S:5.97	S:2.21	S:2.37	S:0.73	
5	**182.30**	**169.71**	**166.72**	**17.55**	**t/o**
	G:151.37	G:190.13	G:186.00	G:16.18	
	C:44.31 Sh:.03	Pr:0.72	Pr:2.22	Pr:1.86	
	S:25.34	S:12.38	S:5.43	S:2.90	
6	**242.85**	**203.29**	**193.89**	**21.29**	**t/o**
	G:169.31	G:263.53	G:256.18	G:18.64	
	C:46.65 Sh:.02	Pr:1.26	Pr:4.03	Pr:3.45	
	S:50.41	S:270.49	S:44.89	S:28.90	
7	**294.06**	**535.70**	**305.78**	**52.23**	**t/o**
	G:165.71	G:272.25	G:279.21	G:19.97	
	C:48.56 Sh:.04	Pr:1.45	Pr:5.49	Pr:4.27	

To be continued on next page

8	S:93.51 **347.98**	S:492.58 **767.53**	S:125.28 **410.94**	S:46.78 **72.39**	**t/o**
9	G:260.35 C:73.63 Sh:.04 S:108.74 **489.18**	G:274.59 Pr:1.09 S:95.60 **371.58**	G:279.46 Pr:5.20 S:73.39 **358.79**	G:20.27 Pr:4.40 S:15.18 **40.92**	**t/o**
10	G:331.88 C:79.15 Sh:.05 S:346.47 **810.14**	G:370.77 Pr:1.67 S:263.94 **637.33**	G:377.35 Pr:6.98 S:87.42 **472.98**	G:21.61 Pr:5.17 S:22.44 **50.32**	**t/o**
11	G:385.14 C:112.95 Sh:.02 S:11.55 **542.97**	G:261.33 Pr:1.18 S:3.88 **266.42**	G:236.23 Pr:4.65 S:1.72 **243.02**	G:19.07 Pr:3.77 S:0.64 **23.66**	**t/o**
12	G:423.68 C:124.07 Sh:.02 S:57.24 **669.76**	G:435.16 Pr:2.11 S:1597.62 **2035.60**	G:422.49 Pr:8.67 S:122.21 **554.90**	G:18.75 Pr:3.69 S:1.11 **24.09**	**t/o**
13	G:508.35 C:144.98 Sh:.03 S:111.78 **848.59**	G:488.17 Pr:2.77 S:13918.09 **14416.59**	G:580.83 Pr:18.33 S:1285.87 **1892.42**	G:30.61 Pr:11.09 S:1419.93 **1467.08**	**t/o**
14	G:362.57 C:114.09 Sh:.04 S:798.89 **1343.93**	G:372.78 Pr:2.12 S:1007.70 **1383.91**	G:391.71 Pr:9.91 S:783.39 **1190.25**	G:24.47 Pr:7.32 S:277.33 **312.72**	**t/o**
15	G:374.01 C:108.89 Sh:.06 S:412.48 **983.37**	G:420.87 Pr:2.82 S:991.48 **1416.27**	G:396.28 Pr:11.98 S:1226.33 **1641.77**	G:25.81 Pr:8.40 S:406.60 **444.45**	**t/o**
16	G:392.92 C:108.06 Sh:.03 S:35.37 **583.60**	G:416.59 Pr:2.06 S:535.82 **955.92**	G:484.35 Pr:10.72 S:379.48 **878.54**	G:25.43 Pr:7.70 S:311.19 **347.71**	**t/o**
17	G:410.74 C:116.36 Sh:.08 S:2297.97 **2937.33**	G:463.94 Pr:2.97 S:4642.81 **5112.80**	G:594.57 Pr:16.62 S:3388.03 **4013.86**	G:29.42 Pr:12.49 S:3554.15 **3623.74**	**t/o**
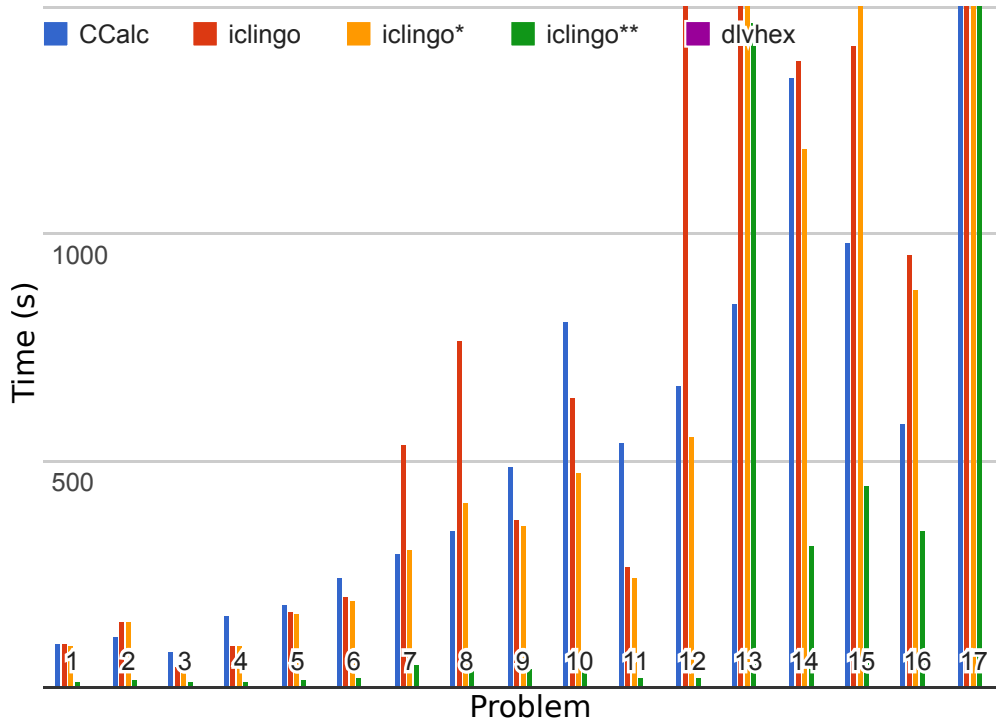
Figure 4.9: Planning Time

In general, we can say that `iclingo` shows a good performance in small instances, especially when the task is to find a plan with a short lengh. In moderate-sized instances, `iclingo` falls back a little in comparison to CCALC, mostly because of the inefficiencies in grounding. In case of large instances, CCALC shows the best performance. This is mostly because of the underlying state-of-the-art SAT solver MINISAT used by CCALC. It is fair since satisfiability solving has been an active research topic for a much longer time than ASP, and SAT solvers come from a long tradition of competitions in which solvers are tested against difficult and enormously large problems. Therefore, it is not a suprise that ASP solvers are outperformed by SAT solvers in large instances. To further exploit the advantages of SAT solvers, we have also run some tests using a parallel SAT solver called MANYSAT [47], but the difference in terms of computation time compared to MINISAT is insignificant, thus we simply omit those results here.

The simplifications we have made in `iclingo` formulation has contributed towards a better timing as well. This is expected since the program size has been largely decreased due to these simplifications made using the available aggregates. Unfortunately, `dlvhex` does not support some of these useful aggregates such as choice and cardinality expressions natively, therefore we cannot provide a simplified version of `dlvhex` domain, and test it against `iclingo`.

Memory usage In Table 4.5 and Figure 4.10, we see the memory usage of different reasoner-formulation combinations. It is seen that even in the small instances CCALC uses massive amount of memory. In comparison, `iclingo` can be identified as memory

Table 4.5: Memory Usage

No	CCALC (MB)	iclingo (MB)	iclingo* (MB)	iclingo** (MB)	dlvhex (MB)
1	2703.28	73.35	139.54	117.92	-
2	2707.04	119.93	281.00	222.70	-
3	2678.73	41.01	53.93	53.54	647.9
4	3462.54	69.62	143.07	124.48	-
5	3762.84	131.07	335.62	251.78	-
6	3789.32	156.93	413.65	326.68	-
7	3810.51	438.71	836.81	594.01	-
8	3832.42	814.89	986.85	707.67	-
9	3881.95	381.25	818.21	674.42	-
10	3933.12	866.42	1109.39	785.04	-
11	3996.43	176.89	738.70	600.25	-
12	3996.46	1336.40	1434.10	601.29	-
13	4102.37	6180.03	3498.68	4578.51	-
14	3969.23	1049.56	2250.25	1775.90	-
15	3993.76	868.79	2793.25	2014.56	-
16	4022.31	1112.26	1739.28	1991.09	-
17	4048.07	2761.21	5079.98	7593.35	-
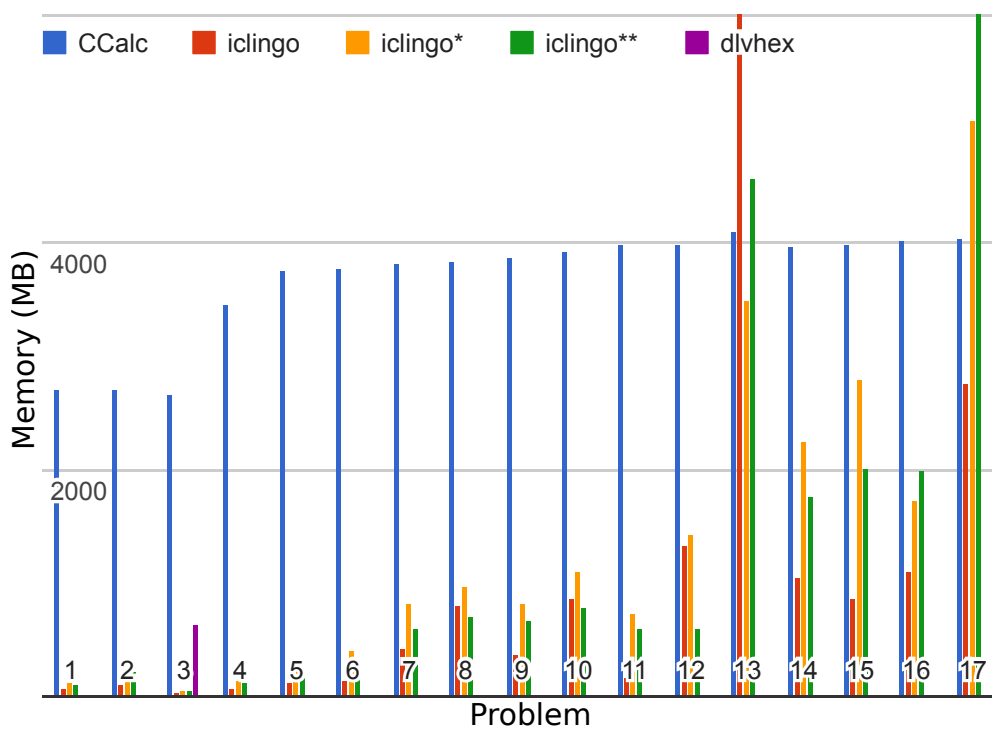
Figure 4.10: Memory Usage

efficient especially in the small instances. This is mostly because of the incremantal reasoning approach of `iclingo`. In large instances, the difference begins to fade. We suspect that this situation is due to the memory efficiency of MINISAT utilized by CCALC, rather than the grounding mechanism of CCALC which is written in Prolog and known to scale not-so-well.

4.5 Plan Optimization

As we have said earlier, our incremental reasoning approach allows us to find the shortest possible plans in terms of plan length. But we have also made it clear that a short plan do not necessarily mean a plan with the minimum duration. Fortunately, reasoners like `clingo` (non-incremental version of `iclingo`) and `dlvhex` allow some optimization directives as input while searching for an answer set. Using these directives, we can achieve an optimal plan with respect to total duration. The optimization directive that can be inserted in any desired plan query for `clingo`, would be as follows:

```
#minimize [elapsed_time(I,t):duration(I)=I@1].
```

Other optimization examples can be easily generated. For instance, we can optimize the number of occurence of a specific action in the plan. The following optization statement would maximize the number of `attach(R,T)` actions in the plan:

```
#maximize [attach(I,J) @ 1].
```

At the end of this chapter, we would like to summarize our contributions from the perspective of reasoning by saying that 1) we utilize different logic-based reasoners to find optimal hybrid plans, 2) our planning approach allows us to represent planning problems with complex temporal goals and find the corresponding plans in an efficient way, 3) we provide experimental evaluation of CCALC with SAT solver MINISAT, and ASP solvers `iclingo` and `dlvhex`.

Chapter 5

Monitoring the Plan Execution

Having plans to tidy a house does not necessarily mean the execution will always be flawless. In a real world scenario such as housekeeping, we should consider possible plan failures, and monitor the execution of the plans to make sure they succeed. Thus, we developed an execution monitoring algorithm for the housekeeping robots. Here, we provide some details about this algorithm, and show its applicability.

5.1   Execution and Monitoring of Hybrid Plans

Suppose that, once each robot obtains the current state of the world from the sensor information, she autonomously finds a hybrid plan to achieve her tasks in a given time, using the hybrid planning approach as presented in Section 4.3. Then, each robot starts executing her plan. However, a plan execution may fail, for instance, due to the unpredictable interference of a human. A human may relocate an object which is to be carried by robot, or bring a new movable object to the room. Since the robots are unaware of these dynamic changes, they may collide with these obstacles. Furthermore, while most of the moveable objects are carried with only one robot, some of these objects are heavy and their manipulation requires two robots. The robots do not know in advance which objects are heavy, but discover a heavy object only when they attempt to move it. Also, a failure may cause a delay and put the time constraint at stake. In such a case, a robot may need assistance of another robot to meet its deadline. When such incidents occur, robots can identify the cause of the failure and act accordingly, e.g., according to the planning and monitoring algorithm illustrated by the flowcharts in Figure 5.1 and 5.2.

In particular, for each group of cleaning robots in a single room, a plan is computed and executed according to Algorithm 1 (monitor). First, each monitoring instance obtains the current state s of the world from the sensor information. After that, a plan $\mathcal{P}$ of length less than or equal to a given nonnegative integer k is computed, from the observed state s_0 to a goal state (that satisfies the given goal g which includes a time constraint tc) in
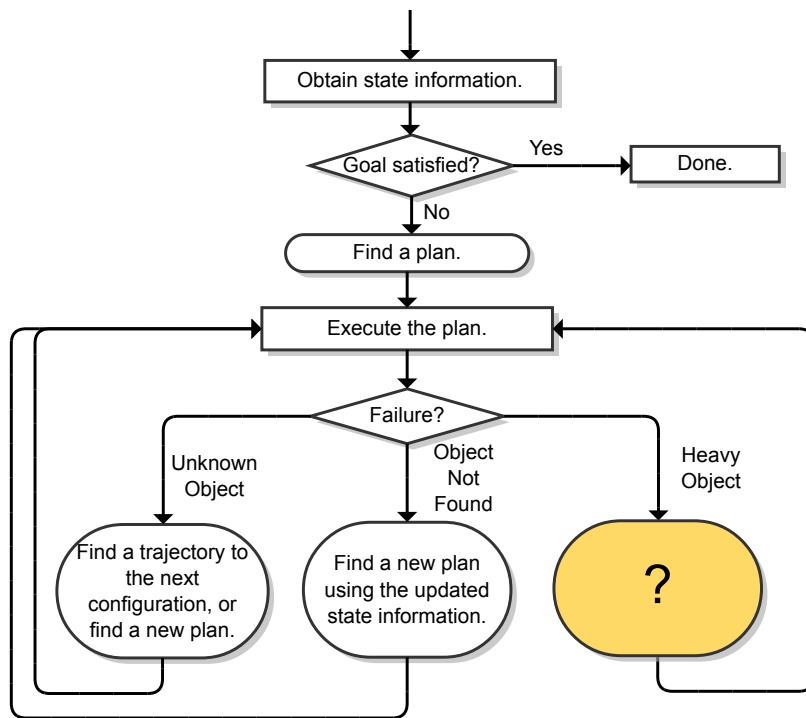
Figure 5.1: Flowchart of an execution and monitoring algorithm for the housekeeping domain
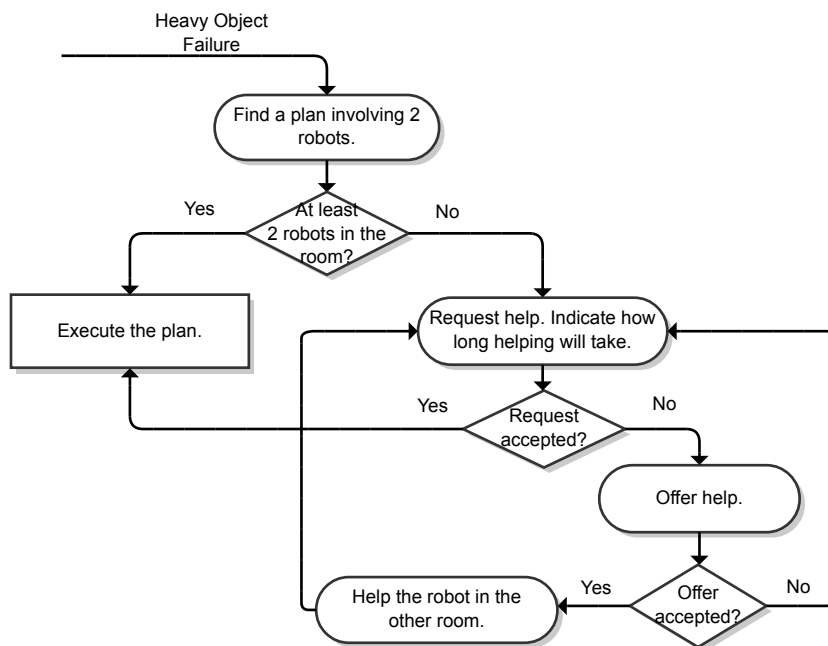
Figure 5.2: Flowchart of help offer routine in the housekeeping domain

a world described by a given action domain description $\mathcal{D}$. Plans are computed using CCALC, `iclingo` or `dlvhex` as described in Chapter 4, by the function `findP`. Once a plan $\mathcal{P}$ is computed, each monitoring instance starts executing it according to Algorithm 8 (execute). If a plan execution fails or is interrupted, the relevant instance can identify the cause of the failure/interrupt and act accordingly.

- When a robot collides with an unknown movable object while following its planned trajectory for a transition $\langle s, A, s' \rangle$ (i.e., the cause of the failure is *UNKNOWN-OBJECT*), Algorithm 2 (caseUnknownObject) is invoked. In particular, the monitoring instance tries to calculate a new, collision-free trajectory π to reach the next state s' from its current configuration. Such a trajectory π is computed by the function `findT`, which implements a motion planner based on RRTs. If no such trajectory is calculated by the motion planer, the robot goes to a safe state s (possibly the previous state) and the monitoring instance asks the reasoner to find a new plan $\mathcal{P}$ to reach the goal from s taking into account the recently discovered moveable objects.

- When a plan fails because a robot attempts to carry an object which is relocated by another agent (i.e., the cause of the failure is *OBJECT-NOT-FOUND*), then following Algorithm 3 (caseObjectNotFound), the monitoring instance observes the world and asks the reasoner to find a new plan $\mathcal{P}$ to reach the goal from the current state s taking into account the recently relocated moveable objects.

- When a plan fails because a robot attempts to manipulate a heavy object (i.e., the cause of the failure is *HEAVY-OBJECT*), there are two possible courses. If there are multiple robots in the room, then a new plan is found with respect to the updated world information, and executed by the robots which are already in the room. Otherwise, the robot has to ask for assistance from other robots so that the heavy object can be carried to its destination (see Algorithm 4 (caseHeavyObject)). However, in order not to disturb the other robots while they are occupied with their own responsibilities, the call for help is delayed as much as possible. With the observation that the manipulation of the heavy object takes 4 steps (get to the heavy object, attach to it, carry it, detach from it), this is accomplished by asking the reasoner to find a new plan $\mathcal{P}$ that manipulates the heavy object within the last $i = 4, 5, 6, ...$ steps of the plan only. Once such a plan is computed, the robot single-handedly follows the first part of the plan without asking for any help.

- As the execution continues, the monitoring instance eventually encounters a concurrent action involving multiple robots (i.e., the cause of the interrupt is *CALL-ROBOT*). At this point, the execution is paused and the monitoring instance notifies the other groups of robots that it needs help (see Algorithm 5 (caseCallRobot)), and

gives them a time estimate for how long the helping process will take by looking at the plan $\mathcal{P}$.

- Once the other monitoring instances' plan execution is interrupted by such a request (i.e., the cause of the interrupt is *HELP-REQUESTED*), they accept or reject the request of help-seeking robot by taking their own deadlines into account, and their possible behaviors can be described as follows (see Algorithms 6 (caseHelpRequested) and 7 (caseHelpOffered)):

  - If one of the monitoring instances decides that it can spare a robot to assist the help-seeking robot without violating its own time constraint by, it replies back to the help request affirmatively, pauses the execution of its own plan, send a robot to the entrance of the other room, and waits for the commands of the help-seeking monitoring instance. After they execute the collaborative part of the plan, the helper robot goes back its room, exactly to the location where it was interrupted in the first place, and the execution of the original plan resumes.

  - If all of the monitoring instances reject the help request due to the fact that they cannot meet their deadlines otherwise, then help-seeking robot offers help to the other monitoring instances. The idea is that, by offering help, the help-seeking robot may save some time in another room. Then, the help-seeking robot can ask for help one more time, and this second attempt is much more likely to be successful.

- If a monitoring instance's plan execution is interrupted by such a help offer (i.e., the cause of the interrupt is *HELP-OFFERED*), then the instance accepts the offered help if that would save the robot some time.

Message passing for asking for help, offering help, helping other robots and listening to messages are detailed in Algorithms 11 (askForHelp) –14 (listen).

Note that if a human brings some objects into the room, and these objects are not in collision course with the robots, they cannot be discovered until the robot observes the world. Therefore, in Algorithm 1 (monitor), after the execution of the plans, a final observation instance is triggered comparing current state of the room with its goal state. If there are any discrepancies, then the robots ask for new plans to reach the goal.

Plans are executed according to Algorithm 8 (execute) as follows. For each transition $\langle s, A, s' \rangle$ of the history of the plan:

- If the action A can be performed by the robots in the room then the monitoring instance executes it according to Algorithm 9 (executeAction).

- Otherwise, A is a concurrent action requiring more robots, the monitoring instance finds a helper to collaborate with and then the robots execute the concurrent action according to Algorithm 9 (executeAction).

Given the current state s and the next state s', an action A is executed at s according to Algorithm 9 (executeAction) and 10 (sendAction) as follows.

- If A is a primitive action of the form `goto(R,X,Y,T)`, then a trajectory is computed from the current configuration (at s) to the next configuration (at s') and the relevant robot follows that trajectory.

- If A is a primitive action of some other form (e.g., `attach(R,T)` or `detach(R,T)`), then the action is performed by the relevant robot.

- If A is a concurrent action of the form $\{A_1, \ldots, A_n\}$ to be executed by multiple robots independently, then the actions are executed in parallel by recursively invoking Algorithm 9 (executeAction) and 10 (sendAction).

- If A contains a concurrent action of the form $\{A_1, A_2\}$ to be executed by two robots collaboratively (i.e., each A_i is of the form `goto(R,X,Y,T)` and both robots are attached to the same object according to s), then a trajectory for each robot is computed from the current configuration (at s) to the next configuration (at s') and the robots follow these trajectories.

Algorithm 1: monitor

Input: Action description $\mathcal{D}$, goal g, maximum plan length k, identifier of the location $room$, agents in the room $robots$

```
/* Get the current state s₀ and configuration c from sensor
   information                                              */
```
$s_0, c \leftarrow$ observe();

while s_0 *does not satisfy* g **do**

    $F := true$;

    $tc \leftarrow$ extract global time constraint from g;

```
    /* Find a plan 𝒫 = ⟨s₀, a₀, s₁, ..., aₙ, sₙ₊₁⟩ to reach the goal g
       from s₀ under constraints F                          */
```

    $plan, \mathcal{P} \leftarrow$ findP$(\mathcal{D}, s_0, g, F, k)$;

    if $plan$ **then**

```
        /* Execute the plan 𝒫 starting at state s₀ and, if it
           fails, return the cause of failure (result) and the
           current state (sᵢ)                               */
```

        $result, s_i, msg \leftarrow$ execute$(\mathcal{P}, s_0, robots)$;

        while $result \neq$ NO-INTERRUPT **do**

            if $result ==$ UNKNOWN-OBJECT **then**

                $result, s_i, msg, \mathcal{P} \leftarrow$ caseUnknownObject$(s_i, \mathcal{D}, g, F, k, \mathcal{P}, robots)$;

            else if $result ==$ OBJECT-NOT-FOUND **then**

                $result, s_i, msg, \mathcal{P} \leftarrow$ caseObjectNotFound$(\mathcal{D}, g, F, k, \mathcal{P}, robots)$;

            else if $result ==$ HEAVY-OBJECT **then**

                $result, s_i, msg, \mathcal{P}, F \leftarrow$ caseHeavyObject$(\mathcal{D}, g, F, k, \mathcal{P}, robots)$;

            else if $result ==$ CALL-ROBOT **then**

                $result, s_i, msg, robots \leftarrow$ caseCallRobot$(s_i, \mathcal{P}, robots, room)$;

            else if $result ==$ HELP-REQUESTED **then**

                $result, s_i, msg \leftarrow$ caseHelpRequested$(s_i, tc, \mathcal{P}, robots, room, msg)$;

            else // $result ==$ HELP-OFFERED

                $result, s_i, msg, F, \mathcal{P}, robots \leftarrow$ caseHelpOffered$(s_i, \mathcal{D}, g, F, \mathcal{P}, tc, room, msg)$;

    else

```
        /* Check if a plan is possible with more robots, i.e.,
           the reason of the failure is missing the deadline  */
```

        $s_0 \leftarrow$ add a helper robot to s_0;

        $plan, \mathcal{P} \leftarrow$ findP$(\mathcal{D}, s_0, g, F, k)$;

        if $plan$ **then**

            $result, s_i, msg, robots \leftarrow$ caseCallRobot$(s_0, \mathcal{P}, robots, room)$;

    $s_0, c \leftarrow$ observe();

Algorithm 2: caseUnknownObject

Input : $s_i, \mathcal{D}, g, F, k, \mathcal{P}, robots$
Output: $result, s_i, msg, \mathcal{P}$
$s_0, c \leftarrow$ observe();
// Find a trajectory π from the current configuration c to the next state
$s_{i+1} \leftarrow$ extract from $\mathcal{P}$;
$c_{next} \leftarrow$ extract next configuration from s_{i+1};
$traj, \pi \leftarrow$ findT(c, c_{next});
if $traj$ **then**
    $result \leftarrow$ send the new trajectory π to the failed robot, wait for response;
    if $result ==$ NO-INTERRUPT **then**
        $result, s_i, msg \leftarrow execute(\mathcal{P}, s_{i+1}, robots)$;
else
    $plan := false$;
    while $\neg plan$ **do**
        $s_0 \leftarrow$ Find a safe state closeby;
        $plan, \mathcal{P} \leftarrow$ findP$(\mathcal{D}, s_0, g, F, k)$;
    $result, s_i, msg \leftarrow execute(\mathcal{P}, s_0, robots)$;
return $result, s_i, msg, \mathcal{P}$;

Algorithm 3: caseObjectNotFound

Input : $\mathcal{D}, g, F, k, \mathcal{P}, robots$
Output: $result, s_i, msg, \mathcal{P}$
$s_i, c \leftarrow$ observe();
$plan, \mathcal{P} \leftarrow$ findP$(\mathcal{D}, s_i, g, F, k)$;
$result, s_i, msg \leftarrow execute(\mathcal{P}, s_i, robots)$;
return $result, s_i, msg, \mathcal{P}$;

Algorithm 4: caseHeavyObject

Input : $\mathcal{D}, g, F, k, \mathcal{P}, robots$
Output: $result, s_i, msg, \mathcal{P}, F$
$s_0, c \leftarrow$ observe();
if $|robots| < 2$ **then**
    $s_0 \leftarrow$ add a helper robot to s_0;
    $i := 4$; // help requested at the last i steps
    $plan := false$;
    while $\neg plan$ **do**
        $F \leftarrow$ "the second robot should not move before the last i^{th} step";
        $plan, \mathcal{P} \leftarrow$ findP$(\mathcal{P}, s_0, g, F, k)$;
        $i := i + 1$;
else
    $plan, \mathcal{P} \leftarrow$ findP$(\mathcal{D}, s_0, g, F, k)$;
$result, s_i, msg \leftarrow execute(\mathcal{P}, s_0, robots)$;
return $result, s_i, msg, \mathcal{P}, F$;

Algorithm 5: caseCallRobot

Input : $s_i, \mathcal{P}, room, robots$
Output: $result, s_i, msg, robots$
$expectedDuration \leftarrow$ calculate using $elapsed\_time$ fluents in s_i and s_{goal};
$helpFound, helper \leftarrow \texttt{askForHelp}(room, expectedDuration)$;
if $\neg helpFound$ **then**
    $helpeeFound, helpee \leftarrow \texttt{offerHelp}(room, expectedDuration)$;
    if $helpeeFound$ **then**
        $robot \leftarrow$ Select from $robots$;
        $\texttt{help}(robot, helpee, room, helpeeRoom, s_i)$;
else
    $robots \leftarrow$ add $helper$ to available robots temporarily;
    $result, s_i, msg \leftarrow \texttt{execute}(\mathcal{P}, s_i, robots)$;
    $robots \leftarrow$ remove $helper$ from $robots$;
return $result, s_i, msg, robots$;

Algorithm 6: caseHelpRequested

Input : $s_i, tc, \mathcal{P}, robots, room, msg$
Output: $result, s_i, msg$
$confirmed := false$;
if *not receiving help* **then**
    $helpee, helpeeRoom, expectedDuration \leftarrow msg[1], msg[2], msg[3]$;
    $finishTime \leftarrow$ extract from final state s_g in $\mathcal{P}$;
    if $tc - finishTime \geq expectedDuration$ **then**
        $confirmed := true$;
        $\texttt{send}(helpee, \langle \text{``}ANS\text{''}, \text{``}affirmative\text{''} \rangle)$;
        $acknowledgement \leftarrow \texttt{receive}(helpee, \text{``}ACK\text{''})$;
        if $acknowledgement == \text{``}affirmative\text{''}$ **then**
            // Send a robot there to help
            $robot \leftarrow$ Select from $robots$;
            $c_{last} \leftarrow$ extract the configuration $robot$ from s_i;
            $c_{exchange} \leftarrow \texttt{getExchangePtConf}(room, helpeeRoom)$;
            $traj, \pi \leftarrow \texttt{findT}(c_{last}, c_{exchange})$;
            $timeInfo \leftarrow \texttt{getTimeInfo}(\pi)$;
            $\texttt{send}(robot, \langle \text{``}ACT\text{''}, \langle \text{``}GOTO\text{''}, \pi, timeInfo \rangle \rangle)$;

            $\texttt{help}(robot, helpee, room, helpeeRoom, s_i)$;

            // After help, robot comes back following the reverse of π
            $timeInfo \leftarrow \texttt{getTimeInfo}(\pi_r)$;
            $\texttt{send}(robot, \langle \text{``}ACT\text{''}, \langle \text{``}GOTO\text{''}, \pi_{reverse}, timeInfo \rangle \rangle)$;
if $\neg confirmed$ **then**
    $\texttt{send}(helpee, \langle \text{``}ANS\text{''}, \text{``}negative\text{''} \rangle)$;
$result, s_i, msg \leftarrow \texttt{execute}(\mathcal{P}, s_i, robots)$;
return $result, s_i, msg$;

Algorithm 7: caseHelpOffered

Input : $s_i, \mathcal{D}, g, F, k, \mathcal{P}, robots, tc, room, msg$

Output: $result, s_i, msg, \mathcal{P}, robots$

$confirmed := false;$

$s_0, c \leftarrow$ observe();

$s_{alt} \leftarrow$ add a helper robot to s_0;

/* Compansate the travel time of the helper robot by starting
   the plan before its arrival   */

$F_{alt} \leftarrow$ "The helper robot should not move during the first t steps";

$plan_{alt}, \mathcal{P}_{alt} \leftarrow$ findP($\mathcal{D}, s_{alt}, g, F_{alt}, k$);

if $plan_{alt}$ **then**

    $tf \leftarrow$ extract the finishing time from goal state s_g in $\mathcal{P}_{alt}$;

    $tr := tc - tf;$

    $helper, helperRoom, expectedDuration \leftarrow msg[1], msg[2], msg[3];$

    if $tr > expectedDuration$ **then**

        $confirmed := true;$

        send($helper, \langle$"ANS", "$affirmative$", $room\rangle$);

        $acknowledgement \leftarrow$ receive($helper$, "ACK");

        if $acknowledgement ==$ "$affirmative$" **then**

            $robots \leftarrow$ add $helper$ to available robots temporarily;

            //preparing helper robot to come nearby the requester

            $c_{exchange} \leftarrow$ getExchangePtConf($room, helperRoom$);

            $c_{safe} \leftarrow$ getSafePtConf($room$);

            $traj, \pi \leftarrow$ findT($c_{exchange}, c_{safe}$);

            $timeInfo \leftarrow$ getTimeInfo(π);

            send($helper, \langle$"ACT", $\langle$"$GOTO$", $\pi, timeInfo\rangle\rangle$);

            $\mathcal{P} := \mathcal{P}_{alt};$

            $result, s_i, msg \leftarrow$ execute($\mathcal{P}, s_0, robots$);

            $c_{last} \leftarrow$ extract the conf. of helper from s_{next};

            $traj, \pi \leftarrow$ findT($c_{last}, c_{exchange}$);

            $timeInfo \leftarrow$ getTimeInfo(π);

            send($helper, \langle$"ACT", $\langle$"$GOTO$", $\pi, timeInfo\rangle\rangle$);

            send($helper, \langle$"ACT", $\langle$"$DONE$"$\rangle\rangle$);

            $robots \leftarrow$ remove $helper$ from available robots;

        else

            $result, s_{next}, msg \leftarrow$ execute($\mathcal{P}, s_0, robots$);

if $\neg confirmed$ **then**

    send($id, \langle ANS$, "$negative$", $-1\rangle$);

    $result, s_{next}, msg \leftarrow$ execute($\mathcal{P}, s_0, robots$);

return $result, s_i, msg, \mathcal{P}, robots$;

Algorithm 8: execute

Input : A plan $\mathcal{P}$ (with a history $H = \langle s_0, a_0, ..., s_n, a_n, s_{goal} \rangle$), current state of plan s_i, agents in the room $robots$

Output: result of last action $result$, current state s_i, message tuple msg

$result := $ NO-INTERRUPT;

$msg := \langle \rangle$;

while $s_i \neq s_{goal}$ **do**

    $a_i, s_{next} \leftarrow$ Extract from $\mathcal{P}$ the action a_i to be executed, and next state s_{i+1};

    if a_i *is a concurrent action with* n *primitives* $\wedge |robots| < n$ **then**

        $result := $ CALL-ROBOT;

        return $result, s_i, msg$;

    $result \leftarrow$ executeAction$(s_i, a_i, s_{next}, robots)$;

    if $result \neq$ NO-INTERRUPT **then**

        return $result, s_i, msg$;

    else

        $s_i := s_{next}$;

    $result, msg \leftarrow$ listen$()$;

    if $result \neq$ NO-INTERRUPT **then**

        return $result, s_i, msg$;

return $result, s_{goal}, msg$;

Algorithm 9: executeAction

Input : current state s_i, action a_i, next state s_{next}, agents in room $robots$

Output: result of action

$result := $ NO-INTERRUPT;

sendAction$(s_i, a_i, s_{next}, robots)$;

foreach $robot$ in $robots$ participating in a_i **do**

    $result \leftarrow$ receive$(robot, RESULT)$;

    if $result \neq$ NO-INTERRUPT **then**

        break;

return $result$;

Algorithm 10: sendAction

Input : current state s_i, action a_i, next state s_{next}, agents in room $robots$

if *a_i is a primitive action* **then**
$\quad robot \leftarrow$ extract from a_i and $robots$;
$\quad timeInfo \leftarrow$ extract from s_i and s_{next};
$\quad$ **if** *a_i is a "goto" action* **then**
$\quad\quad c_i \leftarrow$ extract from s_i;
$\quad\quad c_{next} \leftarrow$ extract from s_{next};
$\quad\quad traj, \pi \leftarrow \texttt{findT}(c_i, c_{next})$;
$\quad\quad \texttt{send}(robot, \langle\text{"ACT"}, \langle\text{"GOTO"}, \pi, timeInfo\rangle\rangle)$;
$\quad$ **else**
$\quad\quad \texttt{send}(robot, \langle\text{"ACT"}, \langle a_i, timeInfo\rangle\rangle)$;
else if *a_i contains collaborating "goto" actions carrying a single object* **then**
$\quad a_{i1}, a_{i2} \leftarrow$ extract from a_i;
$\quad robot_1, robot_2 \leftarrow$ extract from a_{i1}, a_{i2}, and $robots$;
$\quad timeInfo \leftarrow$ extract from s_i and s_{next};
$\quad c_{obj} \leftarrow$ extract configuration of the object from s_i;
$\quad c_{next} \leftarrow$ extract next configuration of the object from s_{next};
$\quad traj, \pi \leftarrow \texttt{findT}(c_{obj}, c_{next})$;
$\quad \pi_1, \pi_2 \leftarrow$ add offset to π for each robot;
$\quad \texttt{send}(robot_1, \langle\text{"ACT"}, \langle\text{"GOTO"}, \pi_1, timeInfo\rangle\rangle)$;
$\quad \texttt{send}(robot_2, \langle\text{"ACT"}, \langle\text{"GOTO"}, \pi_2, timeInfo\rangle\rangle)$;
$\quad \texttt{sendAction}(s_i, a_i - \{a_{i1}, a_{i2}\}, s_{next}, robots)$;
else
$\quad$ **foreach** *primitive action a_{ij} in a_i* **do**
$\quad\quad \texttt{sendAction}(s_i, a_{ij}, s_{next}, robots)$

Algorithm 11: askForHelp

Input : $room, estimatedDuration$
Output: $helpFound, helper, helperRoom$

$helpFound := false;$
$helper := null;$

foreach *group i* **do**
  $\quad$ send$(i, \langle \text{``}HELPREQ\text{''}, room, estimatedDuration \rangle);$

$waiting := true;$

while $\neg helpFound \wedge waiting$ **do**
  $\quad$ **foreach** *group i that has a message* **do**
    $\quad\quad answer \leftarrow$ receive$(i, \text{``}ANS\text{''});$
    $\quad\quad$ **if** $answer = \text{``}affirmative\text{''}$ **then**
      $\quad\quad\quad helpFound := true;$
      $\quad\quad\quad helper := i;$
      $\quad\quad\quad$ break;

  $\quad$ **if** *All groups have responded* **then**
    $\quad\quad waiting := false;$
foreach *group i, such that $i \neq helper$* **do**
  $\quad$ send$(i, \langle \text{``}ACK\text{''}, \text{``}negative\text{''} \rangle);$

if $helpFound$ **then**
  $\quad$ send$(helper, \langle \text{``}ACK\text{''}, \text{``}affirmative\text{''} \rangle);$

return $helpFound, helper, helperRoom;$

Algorithm 12: offerHelp

Input : $room, estimatedDuration$
Output: $helpeeFound, helpee, helpeeRoom$

$helpeeFound := false$;
$helpee := null$;

foreach $group\ i$ **do**
  send($i, \langle$"$HELPOFFER$"$, room, estimatedDuration\rangle$);

$waiting := true$;

while $\neg helpeeFound \wedge waiting$ **do**
    foreach $group\ i\ that\ has\ a\ message$ **do**
        $answer, helpeeRoom \leftarrow$ receive($i,$ "ANS");
        if $answer =$ "$affirmative$" **then**
            $helpeeFound := true$;
            $helpee := i$;
            break;
    if $All\ groups\ have\ responded$ **then**
        $waiting := false$;

foreach $group\ i,\ such\ that\ i \neq helpee$ **do**
  send($i, \langle$"ACK"$,$ "$negative$"$\rangle$);

if $helpeeFound$ **then**
  send($helpee, \langle$"ACK"$,$ "$affirmative$"$\rangle$);
return $helpeeFound, helpee, helpeeRoom$;

Algorithm 13: help

Input: $robot, helpee, room1, room2, s_i$

$conf1 \leftarrow$ extract configuration of $robot$ from s_i;
$conf2 \leftarrow$ getExchangePointConf($room1, room2$);
$traj, \pi \leftarrow$ findT($conf1, conf2$);
$timeInfo \leftarrow$ getTimeInfo(π);
send($robot, \langle$"ACT"$, \langle$"$GOTO$"$, \pi, timeInfo, \rangle\rangle$);

while $true$ **do**
    $message \leftarrow$ receive($helpee, ACT$);
    if $message[0] ==$ "$DONE$" **then**
        break;
    else
        $action \leftarrow message[1]$;
        send($robot, \langle$"ACT"$, action\rangle$);
//Follow trajectory π_r, reverse of π
$timeInfo \leftarrow$ getTimeInfo(π);
send($robot, \langle$"ACT"$, \langle$"$GOTO$"$, \pi_r, timeInfo, \rangle\rangle$);

Algorithm 14: listen

Input :

Output: $type, msg$

foreach *group i* **do**

    if *group i has a help request message* **then**

        $helpeeRoom, expectedDuration \leftarrow$ receive$(i, \text{``}REQ\text{''});$

        return HELP-REQUESTED, $\langle i, helpeeRoom, expectedDuration \rangle;$

    else if *group i has a help offer message* **then**

        $helperRoomId, expectedDuration \leftarrow$ receive$(i, \text{``}OFF\text{''});$

        return HELP-OFFERED, $\langle i, helperRoom, expectedDuration \rangle;$

return NO-INTERRUPT, $\langle \rangle;$

70

Table 5.1 shows the execution of plans calculated in Section 4.3 by Robots 1 and 2. In particular, Robot 1 starts executing its plan in the Bedroom, but at time step 1, it encounters a heavy object. Then, Robot 1 asks Robot 2 for help by giving it the duration of help process, which is estimated to be 9. Considering the help duration and the cost of a round trip travel between the bedroom and the kitchen (which is $2 \times 2 = 4$), Robot

Table 5.1: Execution of the Plans

(The symbol '-' denotes "doing nothing" whereas the symbol '···' denotes
the "continuation of the execution of the previous action".)

Time	Bedroom	Kitchen
0	goto(r1,3,5)	goto(r2,3,6)
1	attach(r1,redpillow1) Heavy object failure occurred.	r2: ···
2	r1: Re-plan. Request help.	attach(r2,mug1)
3	r1: Receive acknowledgement. Offer help.	r2: Decline request due to time limit. goto(r2,2,1)
4	-	r2: ···
5	r1: Receive acknowledgement. Go to the kitchen door at (7,3).	r2: Re-plan. Accept offer. Get ready to receive help.
6	r1: ···	-
7	-	r1: Go to the rendezvous point at (2,4).
8	-	r1: ···
9	-	goto(r1,3,1) goto(r2,4,5)
10	-	attach(r1,plate1) attach(r2,spoon1)
11	-	goto(r1,1,1) goto(r2,2,3)
12	-	detach(r1) detach(r2)
13	-	r1: Go to the bedroom door at (0,3).
14	-	r1: ···
15	r1: Go back to the original position at (3,5).	-
16	r1: ···	-
17	r1: Request help.	r2: Accept request. r2: Go to the bedroom door at (0,3)
18	-	r2: ···
19	r2: Go to the rendezvous point at (4,5).	-
20	r2: ···	-
21	goto(r1,3,6) goto(r2,3,5)	-
22	attach(r1,redpillow2) attach(r2,redpillow1)	-
23	goto(r1,6,3) goto(r2,6,2)	-
24	r1: ··· r2: ···	-
25	detach(r1) detach(r2)	-
26	r2: Go to the kitchen door at (7,3).	-
27	r2: ···	-
28	-	r2: Go back to the original position at (2,3).
29	-	r2: ···

2 declines Robot 1's request, since otherwise it will miss its own deadline. Then, Robot 1 asks Robot 2 "If I help you in the kitchen, will you be able to help me back?". Robot 2 evaluates this help offer and verifies that after receiving help from Robot 1, it can, not only complete its own task on time, but also offer help to Robot 1 to complete its task. Hence, it accepts Robot 1's help offer. They tidy the kitchen together, then Robot 1 goes back to the bedroom and asks for help one more time. This time Robot 2 is unoccupied and willing to help Robot 1. The help request is accepted and the heavy object in the bedroom is carried collaboratively. Both robots successfully complete their tasks without violating their deadlines.

We have shown the applicability of the our approach to hybrid planning, in such a planning and monitoring framework with a simulation of a housekeeping domain. The implementation is done in C++, Java, and Python using Robot Operating System (ROS) tools and libraries. Plan execution is simulated using Gazebo. A video clip illustrating this simulation can be found at the following address: `http://youtu.be/YNFFuVg2tEM`

5.2 Experimental Evaluation: Hybrid Plans

To investigate the advantages and disadvantages of our hybrid planning approach, we run a series of monitoring experiments. The experiments are done using `iclingo` 3.0.3 with the simplified ASP encoding of the housekeeping domain in Appendix C, on a workstation with quad core Intel(R) Xeon(R) CPU E5310 @ 1.60GHz CPU and 6GB RAM. These experiments include 5 different planning problems of varying size in housekeeping domain. More details regarding the size of these problems are provided in Table 5.2. In these experiments, we measure the performance of monitoring algorithm under different settings, in terms of replanning count, time efficiency, and memory usage.

1) In the first experiment, we run the monitoring algorithm under two different settings: one with a domain description that includes the `path_exists(X1,Y1,X2,Y2)` external predicate (which checks the existence of a collision-free trajectory and is used in the description of `goto(R,X,Y,T)` action), and one without the external predicate. In other words, we compare the performance of our hybrid planning approach against a control group. In both of these settings, after finding plans we validate them with respect

Table 5.2: Monitoring Experiment Problem Details

Problem	Robots	Objects	Plan Length
1	2	2	4
2	2	3	8
3	2	3	8
4	2	4	8
5	2	6	12

Table 5.3: Finding A Feasible Plan with and without `path_exists` Predicate

(P:Problem, PT:Planning Time, TT:Total Time, TM:Total Memory)

P	w/o External Predicate				w/ External Predicate			
	Replans	PT(s)	TT(s)	TM(MB)	Replans	PT(s)	TT(s)	TM(MB)
1	5	7.23	320.40	186.85	0	54.23	57.74	141.85
2	192	913.42	t/o	304.71	0	109.85	118.60	251.56
3	58	275.35	3909.40	288.35	0	111.93	118.73	249.54
4	88	559.51	5344.51	348.18	0	113.26	120.20	308.54
5	94	4926.79	t/o	845.17	0	182.78	193.52	668.56

to low-level geometric constraints, i.e. we check if it is possible for the robots to perform high-level `goto(R,X,Y,T)` actions through collision free paths. If the validation process points out that the plan is not feasible, we force replanning with an updated query that restricts the last infeasible action. All instances are satisfiable.

Table 5.3 shows the results of the first experiment. We see that when we disable the `path_exists` external predicate, number of replannings due to plan failures increase, time efficiency of both the planner and the overall process decrease, and memory usage increases as well. Nearly every single measurement gets worse regardless of the problem size, but the most affected instances are the larger ones. Two of the five instances timeout before finding a feasible plan. The only positive thing in table when we disable `path_exists` predicate is that the planning time decreases in the smallest instance. But even in this case, due to costly feasibility checks, the total time elapsed before finding a feasible plan is much more compared to the setting with the external predicate. On the other hand, when we enable the external predicate, not a single replan occurs due to an infeasibility.

2) In the second experiment, there are again two different settings that we compare. We disable the `time_estimate(X1,Y1,X2,Y2)=D` external function (which estimates the duration `D` of going from `(X1,Y1)` to `(X2,Y2)`) in the first setting. In the second setting, this predicate is present. In every instance, we enforce a duration limit for the robots to accomplish the given task. The plans are validated with respect to this duration limit in the low-level. If a high-level plan does not respect this limit, it is discarded, the query is updated to disallow the previously found plan, and a new plan is searched. All instances are satisfiable, and the duration limits are managable with appropriate plans.

Table 5.4 shows the results of the second experiment. As you can see, without the `time_estimate` function, only the smallest instance can be solved within a reasonable time. Even in this case, a significant amount of replans are needed, and the total time elapsed is orders of magnitude longer compared to the setting with the external function enabled. In general, it is clear that the external function reduces the need for replanning while contributing towards time and memory monitoring.

3) Third and the final experiment is about identifying the cumulative effect of

73

Table 5.4: Finding A Feasible Plan with and without `time_estimate` Function

(P:Problem, D:Duration Limit, PT:Planning Time, TT:Total Time, TM:Total Memory)

		w/o External Function				w/ External Function			
P	D	Replans	PT(s)	TT(s)	TM(MB)	Replans	PT(s)	TT(s)	TM(MB)
1	4	92	115.39	8894.73	186.89	0	56.1	59.45	141.84
2	10	39	176.33	t/o	284.62	1	223.89	639.69	284.09
3	10	24	108.43	t/o	283.51	0	111.01	119.41	285.28
4	10	26	165.02	t/o	343.64	0	112.41	119.25	347.04
5	17	18	860.44	t/o	696.75	0	206.84	219.08	695.15

Table 5.5: Finding A Feasible Plan with and without both `path_exists` and `time_estimate`

(P:Problem, D:Duration Limit, PT:Planning Time, TT:Total Time, TM:Total Memory)

		w/o External Computation				w/ External Computation			
P	D	Replans	PT(s)	TT(s)	TM(MB)	Replans	PT(s)	TT(s)	TM(MB)
1	4	5	7.22	321.82	186.82	0	108.8	115.59	141.85
2	10	192	914.44	t/o	304.71	1	434.16	455.68	258.57
3	10	140	656.10	9002.57	297.43	0	244.09	257.78	254.15
4	10	104	662.24	6434.32	351.15	3	871.88	942.58	319.54
5	17	106	3932.28	t/o	766.62	1	695.51	727.52	617.82

both external predicates/functions. In one setting, we disable both `path_exists` and `time_estimate` at the same time, and the external computation is available in the other setting. In both settings, every plan is checked for feasibility with respect to both geometric constraints and temporal constraints.

In Table 5.5, we can see the results of the final experiment. The results are parallel to the previous two experiments, i.e. regardless of the problem size, reasoning process without the external computation produces a lot of infeasible plans which cost a lot in terms of time and memory during the validation and replanning steps. Thus, we can say that our hybrid approach improves the reasoning performance, and quality of the plans.

To conclude this chapter, we want to summarize our results in terms of execution monitoring by saying that 1) we develop a modular monitoring algorithm that can be easily extended or applied in another domain, 2) the main difference of our approach from the existing solutions to monitoring is that, rather than simply replanning in case of a failure, we identify the failure and take action accordingly by avoiding a costly replan as much as possible, 3) we embed our hybrid planning approach into our monitoring algorithm, 4) we provide experimental evaluation of our monitoring approach.

Chapter 6

Related Work

In our approach to housekeeping problem, we address a variety of challenges from execution monitoring to the integeration of low-level and high-level reasoning. Hence, it would be a good idea to examine the related studies in literature under several subtopics.

6.1 Domestic Service Robots

It is not a suprise that domestic service robotics attracts a fair amount of researchers, considering highly practical possible outcomes of the research. CAESAR, a domestic service robot capable of object manipulation that has participated in RoboCup@Home competitions for many years, is the outcome of a series of studies in this field [80, 28]. At the high-level, the CAESAR uses robot programming and planning language READYLOG [27], a GOLOG dialect that allows prioritization of parallel actions, guarded execution, and interrupts. At the low-level, it employs an A* based incremental approach for path planning that initially omits the kinematic constraints to find a coarse path and then fine-tunes it into a feasible path. The claim is that this approach improves the timing and increases the responsiveness of the robot.

Beetz et al. [5] present a robot capable of doing basic household chores. The behavior of the robot is specified in CRAM-PL [6], a robot programming and planning language, in a hierarchially structured way so that the problem specific action plans can be composed of robot's library of default plans. The plans are semantically annotated to make interpretions and runtime adjustments possible. Different reasoning mechanisms (e.g. reasoning about reachability through inverse kinematics) are incorporated into queries phrasing control decisions. Also, an external probabilistic inference mechanism is utilized for determining expected locations of objects.

Pecora et al. [76] propose a reasoning service architecture for the use of different types of human assistance robots/actuators in the house. The architecture grounds the behavior of the human agent into meaningful tasks using constraint-based and temporal

reasoning techniques on sensory data. Concurrently, plans are found to assist the human agent in that specific task via available actuators/robots in the environment.

Dornhege et al. [17] present their study on a robot capable of tidying up a house by manipulating objects, which is similar to our work to some extend. For monitoring and high-level planning, symbolic task planner Temporal Fast Downward/Modules (TFD/M) [16] is used. Geometric reasoners are integrated into the high-level planning process using semantic attachments, a concept similar to external predicates. The interface between planner and the semantic attachment modules allows the exchange of complete state information, similar to `dlvhex`. Contrary to our study, interactions between multiple robotic agents are not taken into account.

Galindo et al. [32] show that a domestic service robot can self-assign goals using semantic knowledge about the environment. The normative knowledge of the world (e.g., "Milk is kept in the refrigerator.") is represented using description logics, specifically OWL-DL [72]. Then, any violations of the norm (e.g. "A milk bottle is on the countertop.") are detected and isolated using a DL reasoner, and a goal condition that will resolve the inconsisteny in the knowledge base is automatically obtained and feeded to a Hierarchical Task Networks (HTN) based planner. To some extend, we can call it similar to automatic inference of expected location of objects using commonsense knowledge bases in our work.

[44] is a study describing a humanoid cooking robot. High-level description of cooking recipes are modeled in terms of a Hierarchial Task Network (HTN), while RRT-Connect Planner is used for generating collision-free, dynamically stable motion plans from full-body posture goals.

Kaneko et al. [55] propose a planning strategy for the task of putting away laundry. The study focuses on recognition and handling of non-solid objects, i.e clothes. It does not specify a task or motion planning method particularly, but outlines general framework for isolation, unfolding, and folding of a cloth.

An important aspect of domestic service robotics research is the interaction with the environment and human beings. As a result, there exist a fair amount of studies in the literature focusing on the low-level control and human interaction of the domestic robots. [66] is one of them, introducing a hybrid force/position controller for a robot that can perform table cleaning, and path teaching/learning. Palacin et al. [74] propose a control mechanism with low-cost components for a mobile robot capable of floor cleaning. Forlizzi et al. [31] provide an in-depth study on well-known Roomba vacuum cleaner robots in terms of human-robot interaction and ethnographic design.

6.2 Execution Monitoring

It is argued that execution monitoring is not seen as an independent research topic by robotics community [77]. Pettersson also claims that, despite the lack of interest towards execution monitoring in robotics community, the topic is covered in great detail by control theorists under the name of *fault detection and isolation* (FDI). By control theorists, execution monitoring solutions are labeled as one or more of three approaches, namely, analytical, data-driven, and knowledge driven [10]. Our approach can be identified as a knowledge-driven expert system.

Another study that can be classified under knowledge-driven methods is [1] in which an expert system is used for FDI purposes in flexible assembly system. One of the drawbacks of this monitoring approach is relative sensitivity to uncertainties due to the use of logic terms which are either true or false. This issue is addressed in [91] by using a fuzzy logic based decision mechanism for monitoring in a automated robotic assembly line.

Besides these similar approaches, there exist a whole variety of different solutions to the execution monitoring problem. [51], [14] are examples of analytical methods applied on robotic arms. Parsons et al. [75] propose a data-driven approach for mobile robots.

6.3 Integration of Symbolic and Geometric Reasoning

Integration of high-level symbolic reasoning and low-level geometric reasoning is a well-known challenge of mobile robotics, and addressed in many studies including early examples like [65], and more recent ones such as [45].

Fainekos et al. [24] target this integration by combining temporal logic and motion planning. First, the workspace of robot is discretized into several cells, then a plan is constructed on this discrete domain using temporal logic model checking tools. Afterwards, the discrete plan is detailed at low-level while the temporal constraints are preserved. However, this approach does not include an interleaving between symbolic and motion plan, thus plan failures require a complete replan.

Erdem et al. [22] follow a similar method to our approach, by using external predicates to represent low-level geometric constraints in a logic formalism. In this way, they prevent collisions between the robots in the domain at symbolic planning level. The shortcoming of this method lies in the restrictions of action language $\mathcal{C}+$, and its reasoner CCALC which does not allow the use of second-order variables. As a result, constraints about objects that may change quantity over different problem instances (e.g. "All payloads should be in non-colliding positions with respect to each other."), cannot be represented in an elegant way.

Cambon et al. [8] implement a hybrid planner called aSyMov. Besides the symbolic

and geometric data, a relation between these two are given to the planner. While extending the state space by applying symbolic actions, the low-level validity of the state is also checked with respect to the previously given relation.

Hauser et al. [48] consider the search space as a graph in which the nodes correspond to subtasks, and the edges denote precedence constraints. Starting from an initial node, they incrementally construct this graph. In each incremental step, configuration space of each newly expanded node is sampled, and the new milestones are connected to prior ones using Probabilistic Road Map (PRM) [49] planners. Therefore, an interweaving between the task planning and motion planning is achieved.

Kaelbling et al. [53] present their top-down hierarchial planning approach in which a goal-regression planner is used. The low-level details of each predecessor state is suggested by geometric reasoners during the high-level regression process, thus the obtained plan is hybrid.

Eyerich et al. [23] propose the use of semantic attachments which is just another name for external predicates/functions, to integrate low-level reasoning into classical planning systems.

Several studies on assembly planning also focus on combining geometric reasoning with higher-level elements, since the nature of the problem has requirements beyond the capabilities of motion planners. Halperin et. al. [46] use constraints to represent the set of object configurations, while Hutchinson et al. [50], Cao et al. [9], Tung et al. [84] use graphs for the same purpose.

Chapter 7

Conclusion

We have shown the usefulness of action language $\mathcal{C}+$, and the knowledge representation and reasoning paradigm Answer Set Programming (ASP), in housekeeping robotics from the point of view of three perspectives: formal representation of the domain, automated reasoning and planning over this domain, execution and monitoring of computed plans.

In particular, we have formalized a housekeeping domain that involves multiple autonomous robots, in $\mathcal{C}+$ and ASP. We have illustrated how to embed three sorts of semantic knowledge into high-level representation of housekeeping domain for intelligent reasoning: 1) commonsense knowledge automatically extracted from the commonsense knowledge base CONCEPTNET, 2) feasibility check of plans via (continuous) geometric reasoning (e.g., RRT-based motion planning), 3) estimated durations of actions computed also by means of motion planning algorithms.

We have shown how hybrid plans can be computed using the $\mathcal{C}+$ reasoner CCALC, and ASP solvers `iclingo` and `dlvhex` over this domain, taking into account temporal constraints. We have also performed some experiments that illustrated the advantage of using ASP and `iclingo` over the action language $\mathcal{C}+$ and the reasoner CCALC, in terms of computation time and memory consumption.

We have also introduced a planning and monitoring framework so that robots can recover from failures during execution of the hybrid plans. In particular, we have considered execution failures due to (i) collisions with movable objects whose presence and location are not known in advance and (ii) heavy objects that cannot be lifted alone, and introduced algorithms so that robots can identify the cause of failures and act/collaborate accordingly to recover from these failures.

Our approach to planning actions of multiple housekeeping robots can be viewed as a kind of multi-agent planning: tasks need to be allocated among these robots, robots need to collaborate with each other to complete the given tasks, and the robots need to communicate with each other for collaboration. Since our focus is more oriented towards

embedding of background/commonsense knowledge and geometric/temporal reasoning in high-level representation and reasoning, these three aspects of multi-agent planning are kept as simple as possible: we assume that tasks are already allocated among the robots, the robots collaborate with each other only when they cannot complete their tasks due to a failure (for instance, when one robot cannot carry a heavy object), we assume that the robots communicate with each other by means of requesting/offering help. However, thanks to the modular structure of our planning and monitoring framework and the generality of the methods used for hybrid planning, these three aspects of housekeeping robotics domain can be extended with the existing approaches in multi-agent planning (as described in the survey paper [12]. For instance, tasks can be allocated among the robots using a method for deciding for compatible agents, using protocols or auction based methods, such as [58], [39], [88], [89], [90], coordination of robots can be allowed at the stage of hybrid planning even when failures do not occur by following the approaches as in [18], [13], [52], [11], and the communication between robots can be extended possibly via an Agent Communication Language, such as KQML [29] or FIPA ACL [30].

Appendix A

CCALC **Formulation**

```
1  :- macros
2     durationLimit -> 40;
3     xLimit -> 7;
4     yLimit -> 7;
5     actionDurationLimit -> 4.
6
7  :- sorts
8     time;
9     action_time;
10    thing >> (robot;endpoint);
11    x_coord;
12    y_coord.
13
14 :- objects
15    0..20 :: step;
16    0..durationLimit :: duration;
17    0..actionDurationLimit :: action_duration;
18    0..xLimit :: x_coord;
19    0..yLimit :: y_coord;
20    r1 :: robot;
21    ep1 :: endpoint.
22
23 :- variables
24    S :: step;
25    T, T1, T2 :: duration;
26    A, A1, A2 :: action_duration;
27    ROBO, ROBO1, ROBO2 :: robot;
28    EP,EP1,EP2 :: endpoint;
29    TH, TH1 :: thing;
```

```
30    X, X1, X2 :: x_coord;
31    Y, Y1, Y2 :: y_coord.

32

33  :- constants
34    at(thing, x_coord, y_coord),
35    connected(robot, endpoint)  :: inertialFluent;
36    at_desired_location(endpoint) :: sdFluent;
37    robot_time(robot) :: simpleFluent(action_duration);
38    elapsed_time :: inertialFluent(duration);
39    goto(robot, x_coord, y_coord) :: exogenousAction;
40    attach(robot) :: exogenousAction;
41    detach(robot) :: exogenousAction;
42    attach_point(robot) :: attribute(endpoint) of attach(robot).

43

44  :- macros
45    tidy -> [/\EP | at_desired_location(EP)];
46    free -> [/\ROBO /\EP | -connected(ROBO,EP)].

47

48  %% goto - direct effects and preconditions
49  goto(ROBO, X, Y) causes at(ROBO, X, Y).
50  goto(ROBO, X, Y) causes robot_time(ROBO)=A if at(ROBO,X1,Y1)
51    where time_estimate(X1,Y1,X,Y,A).
52  nonexecutable goto(ROBO, X, Y)
53    where occupied(X, Y).
54  nonexecutable goto(ROBO, X, Y) if at(ROBO, X, Y).
55  nonexecutable goto(ROBO, X, Y) if at(ROBO, X1, Y1)
56    where -path_exists(X1, Y1, X, Y).

57

58  %% attach - direct effects and preconditions
59  attach(ROBO) causes connected(ROBO, EP)
60    if attach_point(ROBO)=EP.
61  attach(ROBO) causes robot_time(ROBO)=1.
62  nonexecutable attach(ROBO) if connected(ROBO, EP).
63  nonexecutable attach(ROBO) & attach_point(ROBO)=EP
64    if -[\/X \/Y | at(ROBO, X, Y) & at(EP, X, Y)].

65

66  %% detach - direct effects and preconditions
67  detach(ROBO) causes -connected(ROBO, EP)
68    if connected(ROBO, EP).
69  detach(ROBO) causes robot_time(ROBO)=1.
70  nonexecutable detach(ROBO) if [/\EP | -connected(ROBO, EP)].
```

```
71
72   %% things can be located at a single grid point.
73   caused -at(TH, X, Y) if at(TH, X1, Y1)
74       where X \= X1 ++ Y \= Y1.
75
76   %% two objects cannot share the same grid point.
77   caused false if at(EP, X, Y) & at(EP1, X, Y)
78       where EP \= EP1.
79
80   %% if a robot is attached to an endpoint
81   %% then the endpoint is wherever the robot is
82   caused at(EP, X, Y) if connected(ROBO, EP) & at(ROBO, X, Y).
83   caused false if connected(ROBO, EP1) & connected(ROBO, EP)
84       where EP \= EP1.
85
86   %% an object is at a desired location
87   %% if its endpoints are at that location
88   caused at_desired_location(EP)
89       if at(EP, X, Y)
90       where in_place(EP, X, Y).
91   default -at_desired_location(EP).
92
93   %% objects with 2 endpoints are located
94   %% horizontally or vertically
95   caused false if at(EP1, X1, Y1) & at(EP2, X2, Y2)
96       where diagonal(EP1,X1,Y1,EP2,X2,Y2).
97
98   %% at each step, elapsed_time is incremented
99   %% by the maximum robot_time
100  default robot_time(ROBO)=0.
101  caused elapsed_time=(T+A)
102      if A=robot_time(ROBO),
103        [/\ROBO1 /\A1 | A1=robot_time(ROBO1) ->> (A>=A1)]
104      after T=elapsed_time
105      where T+A =< timeLimit.
106
107  %% a robot can perform only a single action at each step.
108  nonexecutable goto(ROBO, X, Y) & attach(ROBO).
109  nonexecutable goto(ROBO, X, Y) & detach(ROBO).
110
111  %% a planning problem query
```

```
112  :- query
113  label :: 0;
114  maxstep :: 0..20;
115  0: at(r1,3,5), at(ep1,3,5), elapsed_time=0, free;
116  maxstep: tidy, free, elapsed_time=<30.
```

Appendix B

`iclingo` **Formulation**

```
1  #base.
2  #const xLimit=7.
3  #const yLimit=7.
4  #const durationLimit=40.
5  #const actionDurationLimit=4.
6
7  %% sorts
8  endpointplusnone(I) :- endpoint(I).
9  thing(I) :- robot(I).
10 thing(I) :- endpoint(I).
11
12 %% variables
13 #domain time(T).
14 #domain atime(Ta).
15 #domain robot(R;R1;R2).
16 #domain endpoint(EP;EP1;EP2).
17 #domain endpointplusnone(EPN;EPN1;EPN2).
18 #domain thing(TH;TH1;TH2).
19 #domain xcoordinate(X;X1;X2).
20 #domain ycoordinate(Y;Y1;Y2).
21 #domain duration(D;D1;D2;D3;D4).
22 #domain action_duration(Da;Da1;Da2;Da3;Da4).
23
24 %% objects
25 endpointplusnone(none).
26 xcoordinate(0..xLimit).
27 ycoordinate(0..yLimit).
28 robot(r1).
29 endpoint(ep1;ep2).
```

```
30  duration(0..durationLimit).
31  action_duration(0..actionDurationLimit).
32
33  #cumulative t.
34  time(0..t).
35  atime(0..t-1).
36
37  %% auxiliary atoms
38  -tidy(T) :- -at_desired_location(EP,T).
39  tidy(T) :- not -tidy(T).
40  -free(T) :- connected(R,EP,T).
41  free(T) :- not -free(T).
42  -free_robot(R,T) :- connected(R,EP,T).
43  free_robot(R,T) :- not -free_robot(R,T).
44  -different_loc(R,EP,T) :- at(R,X,Y,T), at(EP,X,Y,T).
45  different_loc(R,EP,T) :- not -different_loc(R,EP,T).
46
47  %% at(thing,x,y) - inertial fluent
48  at(TH,X,Y,0) :- not -at(TH,X,Y,0).
49  -at(TH,X,Y,0) :- not at(TH,X,Y,0).
50  at(TH,X,Y,Ta+1) :- not -at(TH,X,Y,Ta+1), not -at(TH,X,Y,Ta).
51  -at(TH,X,Y,Ta+1) :- not at(TH,X,Y,Ta+1), not at(TH,X,Y,Ta).
52
53  %% connected(robot,ep) - inertial fluent
54  connected(R,EP,0) :- not -connected(R,EP,0).
55  -connected(R,EP,0) :- not connected(R,EP,0).
56  connected(R,EP,Ta+1) :- not -connected(R,EP,Ta+1),
57    not -connected(R,EP,Ta).
58  -connected(R,EP,Ta+1) :- not connected(R,EP,Ta+1),
59    not connected(R,EP,Ta).
60
61  %% robot_time(robot) - simple fluent
62  robot_time(R,Da,0) :- not -robot_time(R,Da,0).
63  -robot_time(R,Da,0) :- not robot_time(R,Da,0).
64
65  %% elapsed_time - inertial fluent
66  elapsed_time(D,0) :- not -elapsed_time(D,0).
67  -elapsed_time(D,0) :- not elapsed_time(D,0).
68  elapsed_time(D,Ta+1) :- not -elapsed_time(D,Ta+1),
69    not -elapsed_time(D,Ta).
70  -elapsed_time(D,Ta+1) :- not elapsed_time(D,Ta+1),
```

```
71    not elapsed_time(D,Ta).

72

73    %% goto - exogenous action
74    goto(R,X,Y,Ta) :- not -goto(R,X,Y,Ta).
75    -goto(R,X,Y,Ta) :- not goto(R,X,Y,Ta).

76

77    %% attach - exogenous action
78    attach(R,Ta) :- not -attach(R,Ta).
79    -attach(R,Ta) :- not attach(R,Ta).

80

81    %% attach_point(robot, endpoint+none) - attribute of attach(robot)
82    attach_point(R,EPN,Ta) :- not -attach_point(R,EPN,Ta).
83    -attach_point(R,EPN,Ta) :- not attach_point(R,EPN,Ta).
84    :- not -attach(R,Ta), not -attach_point(R,none,Ta).
85    :- not attach(R,Ta), not attach_point(R,none,Ta).

86

87    %% conversion from non-boolean function
88    -attach_point(R,EPN1,Ta) :- not -attach_point(R,EPN2,Ta),
89      EPN1!=EPN2.
90    :- not attach_point(R,I,Ta) : endpointplusnone(I).

91

92    %% detach - exogenous action
93    detach(R,Ta) :- not -detach(R,Ta).
94    -detach(R,Ta) :- not detach(R,Ta).

95

96    %% goto - direct effect & precondition
97    at(R,X,Y,Ta+1) :- not -goto(R,X,Y,Ta).
98    robot_time(R,Da,Ta+1) :- not -goto(R,X1,Y1,Ta), not -at(R,X2,Y2,Ta),
99      Da:= @time_estimate(X1,Y1,X2,Y2).
100   :- not -goto(R,X,Y,Ta), occupied(X,Y).
101   :- not -goto(R,X,Y,Ta), not -at(R,X,Y,Ta).
102   :- not -goto(R,X2,Y2,Ta), not -at(R,X1,Y1,Ta),
103     @path_exists(X1,Y1,X2,Y2)==0.

104

105   %% attach - direct effect & precondition
106   connected(R,EP,Ta+1) :- not -attach(R,Ta), not -attach_point(R,EP,Ta).
107   robot_time(R,1,Ta+1) :- not -attach(R,Ta).
108   :- not -attach(R,Ta), not -connected(R,EP,Ta).
109   :- not -attach(R,Ta), not -attach_point(R,EP,Ta),
110     not -different_loc(R,EP,Ta).

111
```

```
112   %% detach - direct effect & precondition
113   -connected(R,EP,Ta+1) :- not -detach(R,Ta), not -connected(R,EP,Ta).
114   robot_time(R,1,Ta+1) :- not -detach(R,Ta).
115   :- not -detach(R,Ta), not -free_robot(R,Ta).
116
117   % an object can be present at a single grid point
118   -at(TH,X,Y,T) :- not -at(TH,X1,Y1,T), X!=X1.
119   -at(TH,X,Y,T) :- not -at(TH,X1,Y1,T), Y!=Y1.
120
121   % two object cannot be present at the same grid point
122   :- not -at(EP1,X,Y,T), not -at(EP2,X,Y,T), EP1!=EP2.
123
124   %% conversion from non-boolean function
125   -robot_time(R,Da1,T) :- not -robot_time(R,Da2,T), Da1!=Da2.
126   :- not robot_time(R,I,T) : action_duration(I).
127
128   -elapsed_time(D1,T) :- not -elapsed_time(D2,T), D1!=D2.
129   :- not elapsed_time(I,T) : duration(I).
130
131   % if a robot is attached to an endpoint
132   % then the endpoint is wherever the robot is
133   at(EP,X,Y,T) :- not -connected(R,EP,T), not -at(R,X,Y,T).
134   :- not -connected(R,EP1,T), not -connected(R,EP2,T), EP1!=EP2.
135
136   % an object is at a desired location
137   % if its endpoints are at that location
138   at_desired_location(EP,T) :- not -at(EP,X,Y,T), in_place(EP,X,Y).
139   -at_desired_location(EP,T) :- not at_desired_location(EP,T).
140
141   % objects with 2 endpoints are located horizontally or vertically
142   :- not -at(EP1,X1,Y1,T), not -at(EP2,X2,Y2,T),
143      diagonal(EP1,X1,Y1,EP2,X2,Y2).
144
145   % at each step, elapsed_time is incremented
146   % by the maximum robot_time
147   longer_time_exists(Da1,Tf) :- robot_time(R1,Da1,Tf),
148      robot_time(R2,Da2,Tf), Da1 < Da2.
149
150   elapsed_time(Da1+D2,Ta+1) :- robot_time(R1,Da1,Ta+1),
151      not longer_time_exists(Da1,Ta+1), elapsed_time(D2,Ta).
152
```

```
153   % default robot_time is 0.
154   robot_time(R,0,T) :- not -robot_time(R,0,T).
155
156   %% concurrency restrictions
157   :- not -goto(R,X,Y,Ta), not -attach(R,Ta).
158   :- not -goto(R,X,Y,Ta), not -detach(R,Ta).
159
160   %% a planning problem instance
161   %% initial state
162   #base.
163   :- not at(r1,4,6,0).
164   :- not free(0).
165   :- not at(ep1,3,5,0).
166   :- not at(ep2,3,6,0).
167   :- not elapsed_time(0, 0).
168
169   %% goal condition
170   #volatile t.
171   :- not tidy(t).
172   :- not free(t).
173   :- elapsed_time(D, t), D>=30.
```

Appendix C

`iclingo` **Formulation (Simplified)**

```
1   #base.
2   #const xLimit=7.
3   #const yLimit=7.
4   #const durationLimit=40.
5   #const actionDurationLimit=4.
6
7   %% sorts
8   thing(I) :- robot(I).
9   thing(I) :- endpoint(I).
10
11  %% objects
12  xcoordinate(0..xLimit).
13  ycoordinate(0..yLimit).
14  robot(r1).
15  endpoint(ep1;ep2).
16  duration(0..durationLimit).
17  action_duration(0..actionDurationLimit).
18
19  %% variables
20  #domain timef(Tf).
21  #domain timea(Ta).
22  #domain robot(R;R1;R2).
23  #domain endpoint(EP;EP1;EP2).
24  #domain thing(TH;TH1;TH2).
25  #domain xcoordinate(X;X1;X2).
26  #domain ycoordinate(Y;Y1;Y2).
27  #domain duration(D;D1;D2;D3;D4).
28  #domain action_duration(Da;Da1;Da2;Da3;Da4).
29
```

```
30   #cumulative t.
31   timef(0..t).
32   timea(0..t-1).
33
34   %% auxiliary atoms
35   tidy(Tf) :- at_desired_location(I,Tf) : endpoint(I).
36   free(Tf) :- not connected(I,J,Tf) : robot(I) : endpoint(J).
37
38   %% at(thing,x,y) - inertial fluent
39   0 { at(TH,X,Y,0) } 1.
40   { at(TH,X,Y,Ta+1) } :- at(TH,X,Y,Ta).
41   :- { at(TH,I,J,Tf) : xcoordinate(I) : ycoordinate(J) } 0.
42   :- 2 { at(TH,I,J,Tf) : xcoordinate(I) : ycoordinate(J) }.
43
44   %% connected(robot,ep) - inertial fluent
45   1 { connected(R,EP,0), -connected(R,EP,0) } 1.
46   connected(R,EP,Ta+1) :- not -connected(R,EP,Ta+1), connected(R,EP,Ta).
47   -connected(R,EP,Ta+1) :- not connected(R,EP,Ta+1), -connected(R,EP,Ta).
48
49   %% robot_time(robot) - simple fluent
50   0 { robot_time(R,Da,0) } 1.
51   :- { robot_time(R,I,Tf) : action_duration(I) } 0.
52   :- 2 { robot_time(R,I,Tf) : action_duration(I) }.
53
54   %% elapsed_time - inertial fluent
55   0 { elapsed_time(D,0) } 1.
56   { elapsed_time(D,Ta+1) } :- elapsed_time(D,Ta).
57   :- { elapsed_time(I,Tf) : duration(I) } 0.
58   :- 2 { elapsed_time(I,Tf) : duration(I) }.
59
60   %% goto - exogenous action
61   0 { goto(R,X,Y,Ta) } 1.
62
63   %% attach - exogenous action
64   0 { attach(R,Ta) } 1.
65
66   %% attach_point(robot, endpoint+none) - attribute of attach(robot)
67   1 { attach_point(R,Var,Ta) : endpoint(Var) } 1 :- attach(R,Ta).
68
69   %% detach - exogenous action
70   0 { detach(R,Ta) } 1.
```

```
71
72  %% goto - direct effect & precondition
73  at(R,X,Y,Ta+1) :- goto(R,X,Y,Ta).
74  robot_time(R,Da,Ta+1) :- goto(R,X2,Y2,Ta), at(R,X1,Y1,Ta),
75    Da:=@time_estimate(X1,Y1,X2,Y2).
76  :- goto(R,X,Y,Ta), occupied(X,Y).
77  :- goto(R,X,Y,Ta), at(R,X,Y,Ta).
78  :- goto(R,X2,Y2,Ta), at(R,X1,Y1,Ta),
79    not @path_exists(X1,Y1,X2,Y2).
80
81  %% attach - direct effect & precondition
82  connected(R,EP,Ta+1) :- attach(R,Ta), attach_point(R,EP,Ta).
83  robot_time(R,1,Ta+1) :- attach(R,Ta).
84  :- attach(R,Ta), connected(R,EP,Ta).
85  :- attach(R,Ta), attach_point(R,EP,Ta), at(R,X1,Y1,Ta),
86    at(EP,X2,Y2,Ta), X1!=X2.
87  :- attach(R,Ta), attach_point(R,EP,Ta), at(R,X1,Y1,Ta),
88    at(EP,X2,Y2,Ta), Y1!=Y2.
89
90  %% detach - direct effect & precondition
91  -connected(R,EP,Ta+1) :- detach(R,Ta), connected(R,EP,Ta).
92  robot_time(R,1,Ta+1) :- detach(R,Ta).
93  :- detach(R,Ta), not connected(R,I,Ta) : endpoint(I).
94
95  % two objects cannot occupy the same grid point
96  :- at(EP1,X,Y,Tf), at(EP2,X,Y,Tf), EP1!=EP2.
97
98  % if a robot is attached to an endpoint
99  % then the endpoint is wherever the robot is
100 at(EP,X,Y,Tf) :- connected(R,EP,Tf), at(R,X,Y,Tf).
101
102 % a robot cannot be connected to two different
103 % objects at the same time
104 :- connected(R,EP1,Tf), connected(R,EP2,Tf), EP1!=EP2.
105
106 % an object is at a desired location
107 % if its endpoints are at that location
108 at_desired_location(EP,Tf) :- at(EP,X,Y,Tf), in_place(EP,X,Y).
109
110 % objects with 2 endpoints are located horizontally or vertically
111 :- at(EP1,X1,Y1,Tf), at(EP2,X2,Y2,Tf),
```

```
112    EP1!=EP2, belongs(EP1,Obj), belongs(EP2,Obj),
113    ((X1-X2)**2 + (Y1-Y2)**2) != 1.
114
115  % at each step, elapsed_time is incremented
116  % by the maximum robot_time
117  elapsed_time(D1+Da2,Ta+1) :-
118    Da2=#max[ robot_time(RX,J,Ta+1):robot(RX)=J ],
119    elapsed_time(D1,Ta), D1+Da2<durationLimit.
120
121  % default robot_time is 0.
122  robot_time(R,0,Tf) :- { robot_time(R,I,Tf) : duration(I) } 0.
123
124  %% concurrency restrictions
125  :- goto(R,X,Y,Ta), attach(R,Ta).
126  :- goto(R,X,Y,Ta), detach(R,Ta).
127
128  %% a planning problem instance
129  %% initial state
130  #base.
131  :- not at(r1,4,6,0).
132  :- not free(0).
133  :- not at(ep1,3,5,0).
134  :- not at(ep2,3,6,0).
135  :- not elapsed_time(0, 0).
136
137  %% goal condition
138  #volatile t.
139  :- not tidy(t).
140  :- not free(t).
141  :- elapsed_time(D, t), D>=30.
```

Appendix D

`dlvhex` **Formulation**

```
1   %% constants
2   #maxint=40.
3   const(xLimit, 7).
4   const(yLimit, 7).
5   const(durationLimit, 40).
6   const(actionDurationLimit, 4).
7
8   %% sorts
9   thing(I) :- robot(I).
10  thing(I) :- endpoint(I).
11
12  %% objects
13  robot(r1).
14  endpoint(ep1).
15  timef(I) :- #int(I), const(t,C), I<=C.
16  timea(I) :- #int(I), const(t,C), I<C.
17  xcoord(I) :- #int(I), const(xLimit,C), I<=C.
18  ycoord(I) :- #int(I), const(yLimit,C), I<=C.
19  duration(I) :- #int(I), const(durationLimit, C), I<=C.
20  action_duration(I) :- #int(I), const(actionDurationLimit,C), I<=C.
21
22  %% auxiliary atoms
23  -tidy(Tf) :- -at_desired_location(EP,Tf),
24      endpoint(EP), timef(Tf).
25
26  tidy(Tf) :- not -tidy(Tf),
27      timef(Tf).
28
29  -free(Tf) :- connected(R,EP,Tf),
```

```
30     robot(R), endpoint(EP), timef(Tf).

31

32  free(Tf) :- not -free(Tf),
33     timef(Tf).

34

35  -free_robot(R,Tf) :- connected(R,EP,Tf),
36     robot(R), endpoint(EP), timef(TF).

37

38  free_robot(R,Tf) :- not -free_robot(R,Tf),
39     robot(R), timef(Tf).

40

41  -different_loc(R,EP,Tf) :- at(R,X,Y,Tf), at(EP,X,Y,Tf),
42     robot(R), endpoint(EP), xcoord(X), ycoord(Y), timef(Tf).

43

44  different_loc(R,EP,Tf) :- not -different_loc(R,EP,Tf),
45     robot(R), endpoint(EP), timef(Tf).

46

47  %% at(thing,x,y) - inertial fluent
48  at(TH,X,Y,0) v -at(TH,X,Y,0) :-
49     thing(TH), xcoord(X), ycoord(Y), timea(Ta).

50

51  at(TH,X,Y,Ti) :- not -at(TH,X,Y,Ti), at(TH,X,Y,Ta), Ti=Ta+1,
52     thing(TH), xcoord(X), ycoord(Y), timea(Ta).

53

54  -at(TH,X,Y,Ti) :- not at(TH,X,Y,Ti), -at(TH,X,Y,Ta), Ti=Ta+1,
55     thing(TH), xcoord(X), ycoord(Y), timea(Ta).

56

57  %% connected(robot,ep) - inertial fluent
58  connected(R,EP,0) v -connected(R,EP,0) :-
59     robot(R), endpoint(EP).

60

61  connected(R,EP,Ti) :- not -connected(R,EP,Ti), connected(R,EP,Ta),
62     Ti=Ta+1, robot(R), endpoint(EP), timea(Ta).

63

64  -connected(R,EP,Ti) :- not connected(R,EP,Ti), -connected(R,EP,Ta),
65     Ti=Ta+1, robot(R), endpoint(EP), timea(Ta).

66

67  %% robot_time(robot) - simple fluent
68  robot_time(R,Da,0) v -robot_time(R,Da,0) :-
69     robot(R), action_duration(Da).

70
```

```
71  %% elapsed_time - inertial fluent
72  elapsed_time(D,0) v -elapsed_time(D,0) :-
73    duration(D).
74
75  elapsed_time(D,Ti) :- not -elapsed_time(D,Ti), elapsed_time(D,Ta),
76    Ti=Ta+1, duration(D), timea(Ta).
77
78  -elapsed_time(D,Ti) :- not elapsed_time(D,Ti), -elapsed_time(D,Ta),
79    Ti=Ta+1, duration(D), timea(Ta).
80
81  %% goto - exogenous action
82  goto(R,X,Y,Ta) v -goto(R,X,Y,Ta) :-
83    robot(R), xcoord(X), ycoord(Y), timea(Ta).
84
85  %% attach - exogenous action
86  attach(R,Ta) v -attach(R,Ta) :-
87    robot(R), timea(Ta).
88
89  %% attach_point(robot, endpoint) - attribute of attach(robot)
90  attach_point(R,EP,Ta) v -attach_point(R,EP,Ta) :- attach(R,Ta),
91    robot(R), endpoint(EP), timea(Ta).
92
93  attach_point_exists(R,Ta) :- attach_point(R,EP,Ta),
94    robot(R), endpoint(EP), timea(Ta).
95
96  :- attach(R,Ta), not attach_point_exists(R,Ta),
97    robot(R), timea(Ta).
98
99  :- attach_point(R,EP1,Ta), attach_point(R,EP2,Ta), EP1<EP2,
100   robot(R), endpoint(EP1), endpoint(EP2), timea(Ta).
101
102 %% detach - exogenous action
103 detach(R,Ta) v -detach(R,Ta) :-
104   robot(R), timea(Ta).
105
106 %% goto - direct effect & precondition
107 at(R,X,Y,Ti) :- goto(R,X,Y,Ta),
108   Ti=Ta+1, robot(R), xcoord(X), ycoord(Y), timea(Ta).
109
110 robot_time(R,Da,Ti) :- goto(R,X2,Y2,Ta), at(R,X1,Y1,Ta),
111   &time_estimate[X1,Y1,X2,Y2,at,Ta](Da)
```

```
112     Ti=Ta+1, robot(R), xcoord(X1), xcoord(X2),
113     ycoord(Y1), ycoord(Y2), action_duration(Da), timea(Ta).
114
115  :- goto(R,X,Y,Ta), occupied(X,Y),
116     robot(R), xcoord(X), ycoord(Y), timea(Ta).
117
118  :- goto(R,X,Y,Ta), at(R,X,Y,Ta),
119     robot(R), xcoord(X), ycoord(Y), timea(Ta).
120
121  :- goto(R,X2,Y2,Ta), at(R,X1,Y1,Ta),
122     not &path_exists[X1,Y1,X2,Y2,at,Ta](),
123     robot(R), xcoord(X1), xcoord(X2),
124     ycoord(Y1), ycoord(Y2), timea(Ta).
125
126  %% attach - direct effect & precondition
127  connected(R,EP,Ti) :- attach(R,Ta), attach_point(R,EP,Ta),
128     Ti=Ta+1, robot(R), endpoint(EP), timea(Ta).
129
130  robot_time(R,1,Ti) :- attach(R,Ta),
131     Ti=Ta+1, robot(R), timea(Ta).
132
133  :- attach(R,Ta), connected(R,EP,Ta),
134     robot(R), endpoint(EP), timea(Ta).
135
136  :- attach(R,Ta), attach_point(R,EP,Ta), different_loc(R,EP,Ta),
137     robot(R), endpoint(EP), timea(Ta).
138
139  %% detach - direct effect & precondition
140  -connected(R,EP,Ti) :- detach(R,Ta), connected(R,EP,Ta),
141     Ti=Ta+1, robot(R), endpoint(EP), timea(Ta).
142
143  robot_time(R,1,Ti) :- detach(R,Ta),
144     Ti=Ta+1, robot(R), timea(Ta).
145
146  :- detach(R,Ta), free_robot(R,Ta),
147     robot(R), timea(Ta).
148
149  %% conversion from multi-valued constants
150  -robot_time(R,Da1,Tf) :- robot_time(R,Da2,Tf), Da1!=Da2,
151     robot(R), action_duration(Da1), action_duration(Da2),
152     timef(Tf).
```

```
153
154   robot_time_exists(R,Tf) :- robot_time(R,Da,Tf),
155     robot(R), action_duration(Da), timef(Tf).
156
157   :- not robot_time_exists(R,Tf),
158     robot(R), timef(Tf).
159
160   -elapsed_time(D1,Tf) :- elapsed_time(D2,Tf), D1!=D2,
161     duration(D1), duration(D2), timef(Tf).
162
163   elapsed_time_exists(Tf) :- elapsed_time(D,Tf),
164     duration(D), timef(Tf).
165
166   :- not elapsed_time_exists(Tf),
167     timef(Tf).
168
169   % things can be present only at a single
170   % grid point at a specific time
171   -at(TH,X,Y,Tf) :- at(TH,X1,Y1,Tf), X!=X1,
172     thing(TH), xcoord(X), xcoord(X1),
173     ycoord(Y), ycoord(Y1), timef(Tf).
174
175   -at(TH,X,Y,Tf) :- at(TH,X1,Y1,Tf), Y!=Y1,
176     thing(TH), xcoord(X), xcoord(X1),
177     ycoord(Y), ycoord(Y1), timef(Tf).
178
179   % two objects cannot occupy the same grid point
180   :- at(EP1,X,Y,Tf), at(EP2,X,Y,Tf), EP1<EP2,
181     endpoint(EP1), endpoint(EP2), xcoord(X), ycoord(Y), timef(Tf).
182
183   % if a robot is attached to an endpoint
184   % then the endpoint is wherever the robot is
185   at(EP,X,Y,Tf) :- connected(R,EP,Tf), at(R,X,Y,Tf),
186     robot(R), endpoint(EP), xcoord(X), ycoord(Y), timef(Tf).
187
188   % robot cannot be connected to multiple objects
189   :- connected(R,EP1,Tf), connected(R,EP2,Tf), EP1<EP2,
190     robot(R), endpoint(EP1), endpoint(EP2), timef(Tf).
191
192   % an object is at a desired location if its endpoints are at that location
193   at_desired_location(EP,Tf) :- at(EP,X,Y,Tf), in_place(EP,X,Y),
```

```
194    endpoint(EP), xcoord(X), ycoord(Y), timef(Tf).

195

196    -at_desired_location(EP,Tf) :- not at_desired_location(EP,Tf),

197       endpoint(EP), timef(Tf).

198

199    % objects with 2 endpoints are located horizontally or vertically

200    :- at(EP1,X1,Y1,Tf), at(EP2,X2,Y2,Tf), diagonal(EP1,X1,Y1,EP2,X2,Y2),

201       endpoint(EP1), endpoint(EP2), xcoord(X1), xcoord(X2),

202       ycoord(Y1), ycoord(Y2), timef(Tf).

203

204    % at each step, elapsed_time is incremented by the maximum robot_time

205    lesser_time_exists(D1,Tf) :- robot_time(R1,Da1,Tf),

206       robot_time(R2,Da2,Tf), Da1 < Da2,

207       robot(R1), robot(R2), action_duration(D1),

208       action_duration(D2), timef(Tf).

209

210    elapsed_time(DTotal, Ti) :- robot_time(R1,Da1,Ti),

211       not lesser_time_exists(Da1,Ti),

212       elapsed_time(D2,Ta), DTotal=Da1+D2, Ti=Ta+1,

213       robot(R1), action_duration(D1), duration(D2), timea(Ta).

214

215    % default robot_time is 0.

216    robot_time(R,0,Tf) :- not -robot_time(R,0,Tf),

217       robot(R), timef(Tf).

218

219    %% concurrency restrictions

220    :- goto(R,X,Y,Ta), attach(R,Ta),

221       robot(R), xcoord(X), ycoord(Y), timea(Ta).

222

223    :- goto(R,X,Y,Ta), detach(R,Ta),

224       robot(R), xcoord(X), ycoord(Y), timea(Ta).

225

226    %% a planning problem instance

227    %% initial state

228    :- not at(r1,4,6,0).

229    :- not free(0).

230    :- not at(ep1,3,5,0).

231    :- not elapsed_time(0,0).

232

233    %% goal condition

234    :- not tidy(C), const(t,C).
```

```
235  :- not free(C), const(t,C).
236  :- elapsed_time(D,C), D>=30, duration(D), const(C,T).
```

Bibliography

[1] MG Abu-Hamdan and A.S. El-Gizawy. Computer-aided monitoring system for flexible assembly operations. *Computers in Industry*, 34(1):1–10, 1997.

[2] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description logics. *Foundations of Artificial Intelligence*, 3:135–179, 2008.

[3] Joseph Babb and Joohyung Lee. Cplus2asp: Computing action language c+ in answer set programming. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, 2013.

[4] Michael Bartholomew and Joohyung Lee. Stable models of formulas with intensional functions. In *KR*, 2012.

[5] Michael Beetz, Dominik Jain, L Mosenlechner, Moritz Tenorth, Lars Kunze, Nico Blodow, and Dejan Pangercic. Cognition-enabled autonomous robot control for the realization of home chore task intelligence. *Proceedings of the IEEE*, 100(8):2454–2471, 2012.

[6] Michael Beetz, Lorenz Mosenlechner, and Moritz Tenorth. Cram—a cognitive robot abstract machine for everyday manipulation in human environments. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 1012–1017. IEEE, 2010.

[7] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[8] S. Cambon, R. Alami, and F. Gravot. A hybrid approach to intricate motion, manipulation and task planning. *The International Journal of Robotics Research*, 28(1):104–126, 2009.

[9] T. Cao and AC Sanderson. Task decomposition and analysis of robotic assembly task plans using petri nets. *Industrial Electronics, IEEE Transactions on*, 41(6):620–630, 1994.

[10] L.H. Chiang, E. Russell, and R.D. Braatz. *Fault detection and diagnosis in industrial systems*. Springer Verlag, 2001.

[11] M. De Weerdt, A. Bos, H. Tonino, and C. Witteveen. A resource logic for multiagent plan merging. *Annals of Mathematics and Artificial Intelligence*, 37(1):93–130, 2003.

[12] Mathijs de Weerdt and Brad Clement. Introduction to planning in multiagent systems. *Multiagent and Grid Systems*, 5(4):345–355, 2009.

[13] K. Decker and V.R. Lesser. Generalizing the partial global planning algorithm. *Int. J. Cooperative Inf. Syst.*, 2(2):319–346, 1992.

[14] W.E. Dixon, I.D. Walker, D.M. Dawson, and J.P. Hartranft. Fault detection for robot manipulators with parametric uncertainty: a prediction-error-based approach. *Robotics and Automation, IEEE Transactions on*, 16(6):689–699, 2000.

[15] Patrick Doherty and Jonas Kvarnström. Temporal action logics. *Foundations of Artificial Intelligence*, 3:709–757, 2008.

[16] Christian Dornhege, Patrick Eyerich, Thomas Keller, Sebastian Trüg, Michael Brenner, and Bernhard Nebel. Semantic attachments for domain-independent planning systems. In *Towards Service Robots for Everyday Environments*, pages 99–115. Springer, 2012.

[17] Christian Dornhege and Andreas Hertle. Integrated symbolic planning in the tidyup-robot project. In *2013 AAAI Spring Symposium Series*, 2013.

[18] E.H. Durfee and V.R. Lesser. Planning coordinated actions in dynamic domains. 1987.

[19] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2004.

[20] Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. A logic programming approach to knowledge-state planning, ii: The dlvk system. *Artificial Intelligence*, 144(1):157–211, 2003.

[21] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. Effective integration of declarative rules with external evaluations for semantic-web reasoning. In *The Semantic Web: Research and Applications*, pages 273–287. Springer, 2006.

[22] E. Erdem, K. Haspalamutgil, C. Palaz, V. Patoglu, and T. Uras. Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 4575–4581. IEEE, 2011.

[23] Patrick Eyerich, Thomas Keller, Bernhard Nebel, et al. Combining action and motion planning via semantic attachments. In *Proc. of Workshop on Combining Action and Motion Planning at ICAPS*. Citeseer, 2010.

[24] G.E. Fainekos, H. Kress-Gazit, and G.J. Pappas. Temporal logic motion planning for mobile robots. In *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, pages 2020–2025. IEEE, 2005.

[25] Paolo Ferraris, Joohyung Lee, Yuliya Lierler, Vladimir Lifschitz, and Fangkai Yang. Representing first-order causal theories by logic programs. *Theory and Practice of Logic Programming*, 12(3):383–412, 2012.

[26] Paolo Ferraris and Vladimir Lifschitz. Mathematical foundations of answer set programming. *We will show them*, 1:615–664, 2005.

[27] Alexander Ferrein and Gerhard Lakemeyer. Logic-based robot control in highly dynamic domains. *Robotics and Autonomous Systems*, 56(11):980–991, 2008.

[28] Alexander Ferrein, Tim Niemueller, Stefan Schiffer, and Gerhard Lakemeyer. Lessons learnt from developing the embodied ai platform caesar for domestic service robotics. In *Proc. of AAAI Spring Symposium*, 2013.

[29] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. Kqml as an agent communication language. In *Proceedings of the third international conference on Information and knowledge management*, pages 456–463. ACM, 1994.

[30] ACL Fipa. Fipa acl message structure specification. *Foundation for Intelligent Physical Agents, http://www. fipa. org/specs/fipa00061/SC00061G. html (30.6. 2004)*, 2002.

[31] J. Forlizzi and C. DiSalvo. Service robots in the domestic environment: a study of the roomba vacuum in the home. In *Proceedings of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction*, pages 258–265. ACM, 2006.

[32] Cipriano Galindo and Alessandro Saffiotti. Inferring robot goals from violations of semantic knowledge. *Robotics and Autonomous Systems*, 2013.

[33] Martin Gebser, Torsten Grote, and Torsten Schaub. Coala: a compiler from action languages to asp. In *Logics in Artificial Intelligence*, pages 360–364. Springer, 2010.

[34] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. Engineering an incremental asp solver. In *Logic Programming*, pages 190–205. Springer, 2008.

[35] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.

[36] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New generation computing*, 9(3-4):365–385, 1991.

[37] Michael Gelfond and Vladimir Lifschitz. Representing actions in extended logic programming. In *JICSLP*, volume 92, page 560, 1992.

[38] Michael Gelfond and Vladimir Lifschitz. Action languages. *Electronic Transactions on AI*, 3(16), 1998.

[39] B.P. Gerkey and M.J. Mataric. Sold!: Auction methods for multirobot coordination. *Robotics and Automation, IEEE Transactions on*, 18(5):758–768, 2002.

[40] Yolanda Gil. Description logics and planning. *AI Magazine*, 26(2):73, 2005.

[41] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1):49–104, 2004.

[42] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, and Hudson Turner. Causal laws and multi-valued fluents. In *Proceedings of Workshop on Nonmonotonic Reasoning, Action and Change (NRAC)*. Citeseer, 2001.

[43] Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *AAAI/IAAI*, pages 623–630. Citeseer, 1998.

[44] F. Gravot, A. Haneda, K. Okada, and M. Inaba. Cooking for humanoid robot, a task that needs symbolic and geometric reasonings. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 462–467. IEEE, 2006.

[45] J. Guitton and J. Farges. Geometric and symbolic reasoning for mobile robotics. In *3rd National Conf. on Control Architecture of Robots*, pages 76–97, 2008.

[46] D. Halperin, J.C. Latombe, and R.H. Wilson. A general framework for assembly planning: The motion space approach. *Algorithmica*, 26(3):577–601, 2000.

[47] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. Manysat: a parallel sat solver. *JSAT*, 6(4):245–262, 2009.

[48] Kris Hauser and Jean-Claude Latombe. Integrating task and prm motion planning: Dealing with many infeasible motion planning queries. In *Workshop on Bridging the Gap between Task and Motion Planning at ICAPS*, 2009.

[49] David Hsu, J-C Latombe, and Rajeev Motwani. Path planning in expansive configuration spaces. In *Robotics and Automation, 1997. Proceedings., 1997 IEEE International Conference on*, volume 3, pages 2719–2726. IEEE, 1997.

[50] S.A. Hutchinson and A.C. Kak. Spar: A planner that satisfies operational and geometric goals in uncertain environments. *AI magazine*, 11(1):30, 1990.

[51] R. Isermann. Estimation of physical parameters for dynamic processes with application to an industrial robot. In *American Control Conference, 1990*, pages 1396–1401. IEEE, 1990.

[52] N.R. Jennings. Specification and implementation of a belief-desire-joint-intention architecture for collaborative problem solving. *Int. Journal of Intelligent and Cooperative Information Systems*, 2(3):289–318, 1993.

[53] Leslie Pack Kaelbling and Tomas Lozano-Perez. Hierarchical planning in the now. In *Workshops at the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[54] Antonios Kakas and Rob Miller. A simple declarative language for describing narratives with actions. *The Journal of Logic Programming*, 31(1):157–200, 1997.

[55] M. Kaneko and M. Kakikura. Planning strategy for putting away laundry-isolating and unfolding task. In *Assembly and Task Planning, 2001, Proceedings of the IEEE International Symposium on*, pages 429–434. IEEE, 2001.

[56] Robert Kowalski and Marek Sergot. A logic-based calculus of events. In *Foundations of knowledge base management*, pages 23–55. Springer, 1989.

[57] Jonas Kvarnström. Talplanner and other extensions to temporal action logic. In *Linköping Studies in Science and Technology, Dissertation*. Citeseer, 2005.

[58] M.G. Lagoudakis, M. Berhault, S. Koenig, P. Keskinocak, and A.J. Kleywegt. Simple auctions with performance guarantees for multi-robot task allocation. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 1, pages 698–705. IEEE, 2004.

[59] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[60] Yves Lespérance, Hector J Levesque, Fangzhen Lin, Daniel Marcu, Raymond Reiter, and Richard B Scherl. A logical approach to high-level robot programming–a progress report. In *Control of the physical world by intelligent systems: papers from the 1994 AAAI fall symposium*, pages 79–85, 1994.

[61] Hector J Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B Scherl. Golog: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1):59–83, 1997.

[62] Vladimir Lifschitz. What is answer set programming?. In *AAAI*, volume 8, pages 1594–1597, 2008.

[63] Fangzhen Lin. Situation calculus. *Foundations of Artificial Intelligence*, 3:649–669, 2008.

[64] Hugo Liu and Push Singh. Conceptnet—a practical commonsense reasoning toolkit. *BT technology journal*, 22(4):211–226, 2004.

[65] T. Lozano-Perez, J. Jones, E. Mazer, P. O'Donnell, W. Grimson, P. Tournassoud, and A. Lanusse. Handey: A robot system that recognizes, plans, and manipulates. In *Robotics and Automation. Proceedings. 1987 IEEE International Conference on*, volume 4, pages 843–849. IEEE, 1987.

[66] F. Marrone, FM Raimondi, M. Strobel, et al. Compliant interaction of a domestic service robot with a human and the environment. In *Proceedings of the 33rd ISR (International Symposium on Robotics) October*, volume 7, page 11. Citeseer, 2002.

[67] Norman McCain and Hudson Turner. Causal theories of action and change. In *Proceedings of the National Conference on Artificial Intelligence*, pages 460–465. John Wiley & Sons ltd, 1997.

[68] Norman Clayton McCain. *Causality in commonsense reasoning about actions*. PhD thesis, University of Texas at Austin, 1997.

[69] John McCarthy. Situations, actions, and causal laws. Technical report, DTIC Document, 1963.

[70] John McCarthy. Circumscription—a form of non-monotonic reasoning. *Artificial intelligence*, 13(1):27–39, 1980.

[71] John McCarthy and Patrick Hayes. *Some philosophical problems from the standpoint of artificial intelligence*. Stanford University, 1968.

[72] Deborah L McGuinness, Frank Van Harmelen, et al. Owl web ontology language overview. *W3C recommendation*, 10(2004-03):10, 2004.

[73] International Federation of Robotics. World robotics - service robots. Technical report, 2012.

[74] J. Palacin, J.A. Salse, I. Valganon, and X. Clua. Building a mobile robot for a floor-cleaning operation in domestic environments. *Instrumentation and Measurement, IEEE Transactions on*, 53(5):1418–1424, 2004.

[75] S. Parsons, O. Pettersson, A. Saffiotti, and M. Wooldridge. Robots with the best of intentions. *Artificial Intelligence Today*, pages 329–338, 1999.

[76] Federico Pecora, Marcello Cirillo, Francesca Dell'Osa, Jonas Ullberg, and Alessandro Saffiotti. A constraint-based approach for proactive, context-aware human support. *Journal of Ambient Intelligence and Smart Environments*, 4(4):347–367, 2012.

[77] O. Pettersson. Execution monitoring in robotics: A survey. *Robotics and Autonomous Systems*, 53(2):73–88, 2005.

[78] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, 2009.

[79] Erick Sandewall. *Features and fluents: A systematic approach to the representation of knowledge about dynamical systems*. Linköping University, 1992.

[80] Stefan Schiffer, Alexander Ferrein, and Gerhard Lakemeyer. Caesar: an intelligent domestic service robot. *Intelligent Service Robotics*, 5(4):259–273, 2012.

[81] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1):181–234, 2002.

[82] Push Singh, Thomas Lin, Erik T Mueller, Grace Lim, Travell Perkins, and Wan Li Zhu. Open mind common sense: Knowledge acquisition from the general public. In *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, pages 1223–1237. Springer, 2002.

[83] Michael Thielscher. Introduction to the fluent calculus. *Electronic Transactions on Artificial Intelligence (http://www. etaij. org)*, 3, 1998.

[84] C.P. Tung and A.C. Kak. Integrating sensing, task planning, and execution for robotic assembly. *Robotics and Automation, IEEE Transactions on*, 12(2):187–201, 1996.

[85] Hudson Turner. Representing actions in logic programs and default theories a situation calculus approach. *The journal of logic programming*, 31(1):245–298, 1997.

[86] Hudson Turner. A logic of universal causation. *Artificial Intelligence*, 113(1):87–123, 1999.

[87] Hudson Turner. Nonmonotonic causal logic. In *Handbook of knowledge representation*, volume 1, pages 759–776. Elsevier Science, 2008.

[88] W.E. Walsh and M.P. Wellman. A market protocol for decentralized task allocation. In *Multi Agent Systems, 1998. Proceedings. International Conference on*, pages 325–332. IEEE, 1998.

[89] M.P. Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *arXiv preprint cs/9308102*, 1993.

[90] M.P. Wellman, W.E. Walsh, P.R. Wurman, and J.K. MacKie-Mason. Auction protocols for decentralized scheduling. *Games and Economic Behavior*, 35(1):271–303, 2001.

[91] B. Yan, T. Zhang, and C. Xie. Fuzzy expert system for fault diagnosis of robotic assembly. In *Intelligent Control and Automation, 2002. Proceedings of the 4th World Congress on*, volume 1, pages 445–449. IEEE, 2002.