Advancing Planning-as-Satisfiability

by

Nathan Robinson

Bachelor of Computer Systems Engineering (Honours 1st Class),

Griffith University, Australia (2006)

A thesis submitted in fulfillment

of the requirements of the degree of

Doctor of Philosophy

Institute for Integrated and Intelligent Systems

Faculty of Engineering and Information Technology

Griffith University, Queensland

Australia

April, 2012

Abstract

Since 1992 a popular and appealing technique for solving planning problems has been to use a general purpose solution procedure for Boolean SAT(isfiability) problems. In this setting, a fixed horizon instance of the problem is encoded as a formula in propositional logic. A SAT procedure then computes a satisfying valuation for that formula, or otherwise proves it unsatisfiable. Here, the SAT encoding is *constructive* in the usual sense that there is a one-to-one correspondence between plans –i.e., solutions to the planning problem– and satisfying valuations of the formula. One of the biggest drawbacks of this approach is the enormous sized formulae generated by the proposed encodings. In this thesis we mitigate that problem by developing, implementing, and evaluating a novel encoding that uses the techniques of *splitting* and *factoring* to develop a compact encoding that is amenable to state-of-the-art SAT procedures. Overall, our approach is the most scalable, and our representation the most compact amongst optimal planning procedures.

We then examine planning with numeric quantities, and in particular optimal planning with action costs. SAT-based procedures have previously been proposed in this setting for the fixed horizon case, where there is a given limit on plan length, however a key challenge has been to develop a SAT-based procedure that can achieve horizon-independent optimal solutions – i.e., the least costly plan irrespective of length. Meeting that challenge, in this thesis we develop a novel horizon-independent optimal procedure that poses partially weighted MaxSAT problems to our own cost-optimising *conflict-driven clause learning* (CDCL) procedure. We perform a detailed empirical evaluation of our approach, detailing the types of problem structures where it dominates.

Finally, we take the insights gleaned for the classical propositional planning case, and develop a number of encodings of problems that are described using control-knowledge. That control-knowledge expresses domain-dependent constraints which: (1) constrain the space of admissible plans, and (2) allow the compact specification of constraints on plans that cannot be naturally or efficiently specified in classical propositional planning. Specifically, in this thesis we develop encodings for planning using temporal logic based constraints, procedural knowledge written in a language based on ConGolog, and Hierarchical Task Network based constraints. Our compilations use the technique of splitting to achieve relatively compact encodings compared to existing compilations. In contrast to similar work in the field of answer set planning, our compilations admit plans in the parallel format, a feature that is crucial for the performance of SAT-based planning.

Statement of Originality

This work has not previously been submitted for a degree or diploma to any university. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made in the thesis itself.

[illegible]

Nathan Robinson

April, 2012

Contents

List of Figures

List of Tables

Acknowledgements

I would first like to acknowledge the generous support of NICTA. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program. As a PhD student hosted at the NICTA Brisbane research laboratory, I would also like to acknowledge the Queensland state government's generous support for the NICTA lab in Brisbane. I would also like to acknowledge the generous financial assistance of Griffith University, and in particular the Institute for Integrated and Intelligent Systems; Griffith University's centre of excellence for ICT research. I would also like to extend a special thank to the Griffith Research Computing Services for the use of thousands of hours of computing time. I would like to thank the University of Toronto Department of Computer Science for hosting me for some time during the last year of my PhD candidature. Finally, one of my Associate Supervisors (Dr Gretton) was supported by the EC FP7-IST grant 215181-CogX during part of my candidature, and I therefore would like to acknowledge that EU funding body for supporting a significant component of my PhD supervision.

There are many people who have extended me manifold support during my PhD candidature. First, I would like to thank my PhD supervisor Abdul Sattar, and my Associate Supervisors Charles Gretton and Duc Nghia Pham. They have provided me with invaluable academic mentoring, were excellent collaborators in my research project, and moreover provided me with much of the psychological support that is required to complete a PhD project. I would like to thank my mentor, Sheila McIlraith, who shared her many wisdoms with me during my stay at the University of Toronto. I would also like to thank my close colleagues at NICTA, Silvia Richter, and at Toronto, Christian Musie, for extending me their friendship. Lastly, I would like to thank the anonymous assessors of my thesis for their careful reading of my work and their comments which significantly helped in preparation of my final manuscript.

I would also like to thank my very close friends for supporting me during my candidature. To Bec for the many wonderful years of cohabitation, and for putting up with me while I was writing the majority of this thesis. To Elliott and Sally for always being there, even when I have not been. To Sam for years of virtual and then real friendship. To Christian and Kristie for putting me up and generally just being amazingly kind.

Finally, I would like to thank my family. For always supporting my choices, right and wrong, and for loving me unconditionally. I love you all, especially Mum, Dad, Scott, and Annaliese.

Chapter 1

Introduction

Automated planning is a core problem of artificial intelligence. Broadly speaking, planning is reasoning about how to achieve goals through action. Automated planning techniques have been applied to a wide range of practical applications including: power supply restoration [71]; the routing of oil-derivative liquids through pipeline networks [46]; control systems, such as the Schindler Miconic-10 elevator control system; and logistics problems. The solution to a planning problem is a plan, that is an action strategy to reach a goal. For example, consider a logistics problem where a fleet of trucks are required to pick-up and deliver a set of packages. If we have a single truck located at a depot that has to pick-up a single package from a warehouse, deliver it to a customer, and return back to the depot, then the following sequence of actions is representative of a valid plan:

$$\langle \texttt{Drive}(T_1, L_1, L_2), \texttt{Pick-up}(T_1, P_1, L_2), \texttt{Drive}(T_1, L_2, L_3),$$
$$\texttt{Drop-off}(T_1, P_1, L_3), \texttt{Drive}(T_1, L_3, L_1)\rangle$$

In this case, `Drive`, `Pick-up`, and `Drop-off` represent actions performed by a truck T_1. P_1 is a package transported by T_1 and L_1, L_2, and L_3 are locations. In the above plan, the locations L_1, L_2, and L_3 may represent, for example, a truck depot, a storage warehouse, and a customer.

A planning problem is defined by a set of propositions which characterise a set of world states, a set of actions that can be executed to invoke transitions between states, a single starting state, and a goal condition that corresponds to a small set of state characterising propositions. A valid plan is then a structure of actions that, when executed from the start state, transitions the system to a state satisfying the goal condition. A number of languages for specifying planning problems exist, however recently the first-order declarative PDDL (Planning Domain Definition Language)

[20, 16] has become the *de facto* standard and has been used in the International Planning Competitions from 1998 until 2011. PDDL allows the definition of STRIPS-style actions [19], where actions have preconditions and postconditions that are conjunctions of facts. PDDL also permits more advanced features including those of ADL (Action Description Language) [51]. In this thesis our focus is the classical propositional planning problem, a conceptually simple and well studied form of planning, where we are restricted to STRIPS-style actions and where the underlying world can be represented by a deterministic state transition system. We also only consider the setting where plans are required to be finite, and therefore restrict our attention to the *classical planning* problem which is PSPACE-complete [10].

Attending to the structure of solutions to a planning problem, there are a number of formats that plans can take. We consider the *linear* and *parallel* formats. A plan in the linear format is a sequence of steps with exactly one action assigned to each step. A plan in the parallel format has a sequence of steps, with a set of actions assigned to be executed in parallel at each step. Many settings require parallel action execution to be realistically modelled. For example, in the previous logistics example it is reasonable to assume that multiple trucks can deliver packages in parallel. In addition to considerations of plan format, there are many ways that the quality of a plan may be judged and therefore many ways of expressing a preference relation over plans. Plan quality metrics we consider in this thesis include the number of actions in a plan, the number of steps, or horizon of a plan, and the cost of a plan, that is the sum of the costs assigned to actions in a plan.

Fixed-horizon classical planning may be encoded as a Boolean SAT(isfiability) problem. SAT problems were the first problem to be proved NP-complete [13]. In this case, we can solve planning problems by solving a sequence of fixed-horizon SAT queries. This SAT-based planning setting, first proposed by Kautz and Selman [35], has been a fruitful approach to classical planning, particularly for step-optimal planning, where a minimal-length plan is sought. Numerous SAT-based planners have been developed that use both the linear and parallel plan formats. The planner MEDIC [18] was the first to automatically compile STRIPS planning problems into SAT and provides a number of different linear encodings that vary in the size of the resulting SAT problems that must be solved.

Currently, the best approaches for domain-independent step-optimal planning are SAT-based. The winner of the optimal track in the 2004 International Planning Competition (IPC-4) was SATPLAN-04, and in the 2006 competition (IPC-5) SATPLAN-06 [39] and MAXPLAN [12] tied for first place. More recently, there have been significant advances in performance shown by SAT-based systems, in particular our planning system SOLE [60], and the systems SASE [30] and SMP [67]. These solvers, all descended from BLACKBOX [38][1], encode the problem posed by an n-step *plangraph* [8] into a *conjunctive normal form* (CNF) propositional formula. A plan is then computed by a dedicated SAT solver such as RSAT [52] or PRECOSAT [7]. Encoding is usually fairly direct. Generally, each variable in a CNF corresponds to whether an action is executed at a step, or whether a proposition is true at a step. The planning constraints, such as *frame axioms*, *mutual exclusion*, and that *an action implies its preconditions and postconditions*, are encoded naturally in terms of those variables. Consequently, one of the biggest drawbacks to BLACKBOX successors is the enormous sized CNFs they generate. For example, the 19-step decision problem generated by SATPLAN-06 for problem 22 from the IPC-5 PIPESWORLD domain has over 20 million clauses and 47 thousand variables. Theoretically, in the worst case the number of clauses needed for each step is polynomially dominated by the number of action variables for that step.

One way to tackle this size blowup is to use a *split* representation of actions [35], as in the planner MEDIC [18]. In early split encodings, variables representing actions were replaced by conjunctions of variables, where each variable represents a parameter of the action. Reuse of these variables between actions can reduce the number of variables required to represent actions overall. Additionally, planning constraints may be *factored* by omitting irrelevant parts of action conjuncts from constraints. In the linear case this can dramatically reduce the size of the resulting SAT problems and increase the overall efficiency of planning. Due to the problem of *interference*, standard split representations of actions cannot be used for parallel planning. We addressed this shortcoming with our planner PARA-L [59], which uses a split action representation and allows parallel action execution where interference does not occur. This permits planning somewhere between the parallel and the serial case. For our planner SOLE [60] we introduced a novel split encoding that was the first compatible with parallel step-optimal planning. An alternate take on step-optimal planning with a split representation has since been provided by Huang et al. [30].

Step-optimality is the primary optimality criteria addressed by existing SAT-based planning systems. For a given plan format, a plan is step-optimal if no plan exists with fewer steps. Often step-optimality does not reflect the optimality criteria required in practice because, in general, the number of steps in a plan is unrelated to the number of actions in a plan, the cost of executing a plan, or its total execution time. The optimal deterministic tracks of the 2008 and 2011 Inter-

[1]SASE is the exception and is based on the SAS+ formalism [4].

national Planning Competitions require plans to be optimal with respect to the sum of the costs of their actions, independent of solution horizon. The current best approaches to cost-optimal planning are based on state space search, for example the *de facto* winning entry of the optimal deterministic track at the 2008 IPC was BASELINE, an A* search [25] (a best first search) with a constant heuristic function.

An optimal solution to an instance of fixed-horizon planning with action costs can be obtained by encoding the problem in Partial Weighted MaxSAT [21]. Recently, there has been considerable work on SAT-based cost-optimal planning with a fixed horizon [23, 63] and with metric resource constraints [28], but it remains largely an open problem to effectively apply SAT-based techniques to the horizon-independent case. Our SAT-based planners CO-PLAN [58] and COS-P [62] plan in the horizon-independent cost-optimal case and outperform state-space search planners on several domains. CO-PLAN, a best first search algorithm that uses SAT planning to find an initial cost bound, entered the optimal deterministic track of the 2008 IPC and placed 4th.

Domain-dependent control knowledge can be used to tailor planning systems to individual planning domains to (1) make planning easier; and (2) allow the specification of constraints on acceptable plans beyond the standard goal representation. As an example, in a logistics domain a control-rule might specify that once a package is at its goal location it may never be moved. While control knowledge of this form may be represented by classical planning with ADL extensions, it becomes increasingly unwieldy and inefficient to represent as the complexity of the constraints increases. Planning with domain-dependent control knowledge has been a successful approach to planning. To highlight just a few examples consider the planners TLPLAN [3] and TALPLAN [15] that both use a simple temporal logic and both performed well at international planning competitions. In SAT-based planning Kautz and Selman [37] integrated a number of domain-specific constraints hand-coded in propositional logic into existing SAT-based planning encodings. In the related field of ASP (answer set programming) planning Son et al. [69] provide a more satisfying solution by providing compilations from a number of expressive control knowledge formalisms into ASP. In particular, they handle control knowledge specified as temporal logic, as Golog [40] programs, and as hierarchical task networks (HTNs) [64]. Their approach prevents the need to tailor constraints written in propositional logic to individual planning problems and instead allows control knowledge to be naturally expressed for whole planning domains. Their formulations of control knowledge prohibit parallel action execution, which reduces the usefulness of their work in a SAT-based planning context.

1.1 Research Problems

We highlight the following research problems in SAT-based planning that we address in this thesis:

- Finding SAT encodings of planning that avoid the problem of size blowup to allow the encoding of realistically sized problems [18, 33];
- Finding SAT encodings that work effectively with modern SAT solvers, specifically encodings that exploit the advanced clause learning mechanisms responsible for the performance of state-of-the-art conflict learning driven solvers [56];
- Improving the performance of optimal SAT-based planning in the fixed-horizon case [63];
- Making SAT-based planning applicable to horizon-independent cost-optimal planning, or to optimising other planning metrics, irrespective of planning horizon[2];
- Making SAT-based planning practical for planning in real world planning problems by easily allowing the integration of expressive control knowledge [37, 44]; and
- Allowing the specification of constraints on plans in SAT-based planning that go beyond the usual conjunctive goal of classical planning [69, 44].

1.2 Thesis Outline

The remainder of this thesis is organised as follows: In Chapter 2 we define the main concepts of automated planning. In Chapter 3 we formally describe SAT-based planning and present a general literature review of SAT-based planning encodings and techniques.

In Chapter 4 we present a general framework for split action representations. We describe a number of encodings of problems represented in this framework into SAT for both linear and parallel planning, that represent existing split encodings. Using this framework we introduce two novel encodings. The first, along the lines of Rintanen et al. [57], increases the efficiency of planning by relaxing constraints on plan optimality. The other encoding, the *precisely split* encoding, is the first split encoding that can be used for parallel step-optimal planning. We provide

[2]The optimal deterministic track of the 2008 and 2011 International Planning Competitions requires solution plans to be horizon-independent cost-optimal.

a detailed empirical comparison of a planning system that uses this encoding with existing SAT-based planning approaches and find that our precisely split encoding is competitive with the state-of-the-art SAT-based planners for parallel step-optimal planning. Notably, our approach produces encodings that are sometimes an order of magnitude smaller than existing encodings in the same setting.

In Chapter 5 we consider the problem using SAT-based planning systems to compute optimal plans for propositional planning problems with action costs. We first introduce Partial Weighted MaxSAT and present a new Partial Weighted MaxSAT solver PWMaxSAT that outperforms existing solvers on the problems we generate. We then describe an encoding of fixed-horizon cost-optimal planning into Partial Weighted MaxSAT. State-based forms of cost-optimal planning generally require admissible heuristics to estimate the cost of reaching the goal from the current state. We explore how admissible relaxed planning heuristics can be computed with Partial Weighted MaxSAT. We empirically compare the performance of our optimal relaxed planning system based on Partial Weighted MaxSAT against a state-of-the-art optimal planner and find that our system is more efficient than the optimal planner and solves more relaxed problems. Finally, we explore how SAT-based planning systems can be applied to horizon-independent cost-optimal planning. We present the first planning system that uses SAT for a proof of horizon-independent cost-optimality. This planning system incorporates the computation of an admissible relaxed planning heuristic. We empirically evaluate our approach against a state-of-the-art cost optimal planner and find that our system is most efficient in two domains, but not competitive overall. However, the current system suggests many promising avenues for future research.

In Chapter 6 we present a number of different high-level control knowledge formalisms and novel split encodings of these formalisms into SAT. The control knowledge formalisms we examine are temporal state constraints, Golog [40] and ParaGolog, a novel variant of ConGolog [22] that has a semantics which allows true parallel action execution. We additionally present a novel formulation of Hierarchical Task Networks (HTNs) based on the SHOP planning system [50]. We allow these formalisms to be nested and hybridised. The encodings we present split a number of predicates on both operator arguments and steps. We present a preliminary empirical proof of concept of some of our formalisms.

Finally, in Chapter 7 we summarise and discuss future research directions.

Chapter 2

Background

In this chapter we formally define the automated planning problem, the basis of the work presented in this thesis. We first describe classical propositional planning and then describe quality metrics that are used to rank solution plans. We also describe the plangraph, a structured problem representation that can be efficiently compiled for typical problem descriptions. The problems posed by that graph are the focus of the rest of this thesis.

2.1 Propositional Planning

Here we define the propositional planning problem. Informally, in planning we aim to find a structure of actions to achieve a specified goal from a given starting state. We will illustrate concepts presented in this section with problems from two planning domains, briefly introduced here. The PDDL definitions for both of these domains is given in Appendix A. The first example, SIMPLE-LOGISTICS, based on IPC-1 domain LOGISTICS, has a set of locations connected by roads, a set of trucks, and a set of packages, where each package can be either at a location or in a truck. There is a set of actions which allow trucks to drive between locations and to pick-up and drop-off packages. A valid plan uses the trucks to deliver packages to the required locations. The second example, called BLOCKS-DIRECT, is based on the IPC-2 domain BLOCKS. There is a table of unbounded size and a set of blocks that can be stacked, either upon the table, or upon other blocks. There are actions that move a block between the table and the top of another block. There are also actions that move a block between the tops of other blocks. A valid plan moves the blocks into a goal configuration.

We first introduce the various components of planning problems and then planning itself.

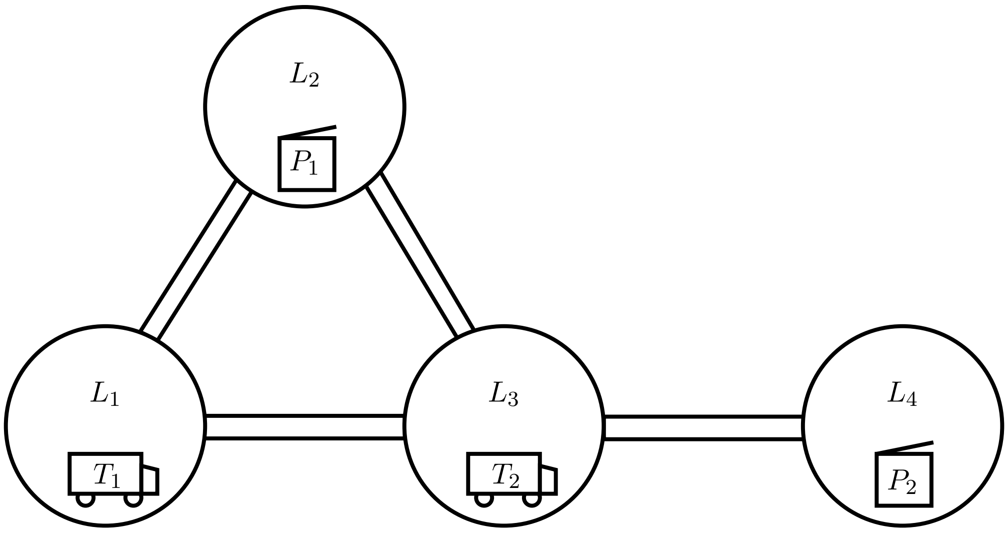


Figure 2.1: A small SIMPLE-LOGISTICS example world state with locations L_1, L_2, L_3, and L_4; trucks T_1 and T_2; and packages P_1 and P_2.

Definition 2.1.1. Let Σ be a set of typed objects. We write $X - \texttt{type}$ for concrete object X labeled with *type*, though we may omit types if they are clear from the context. □

Definition 2.1.2. Let $\texttt{D}(type) \subseteq \Sigma$ be the set of objects in Σ labelled with *type*. □

Figure 2.1 shows a graphical view of a SIMPLE-LOGISTICS problem, where nodes are locations and arcs represent roads between locations. In this example the set of objects $\Sigma :=$

$$\{L_1 - location,\ L_2 - location,\ L_3 - location,\ L_4 - location,$$
$$P_1 - package,\ P_2 - package,\ T_1 - truck,\ T_2 - truck\}$$

Definition 2.1.3. Let Λ be a set of domain predicates with typed arguments. We write $?x - \texttt{type}$ for a variable x labeled with *type* that is an argument of a predicate $\lambda \in \Lambda$. □

Continuing the previous example, $\Lambda :=$

$$\{\texttt{at}(?t - truck, ?l - location),\ \texttt{at}(?p - package, ?l - location),$$
$$\texttt{in}(?p - package, ?t - truck),\ \texttt{road}(?l_1 - location, ?l_2 - location)\}$$

Definition 2.1.4. An *operator* o is an expression of the form $O(\vec{x})$, where O is an operator name and $\vec{x} := \langle ?x_1 - type_1, ..., ?x_k - type_k \rangle$ is a list of typed variable symbols. An operator o then has the following *conditions*:

- *pre*(o) is a set of preconditions, representing facts that must be true before an action instantiating the operator can be executed;
- *post*$^+(o)$ is a set of positive postconditions (add effects), representing facts that become true when an action instantiating the operator is executed; and
- *post*$^-(o)$ is a set of negative postconditions (delete effects), representing facts that become false when an action instantiating the operator is executed. □

For example, our running SIMPLE-LOGISTICS example has the following operators:

$$\begin{aligned}
o_1 &:= \texttt{Drive}(?t - \mathit{truck}, ?\mathit{from} - \mathit{location}, ?\mathit{to} - \mathit{location})\\
&\quad \mathit{pre}(o_1) := \{\texttt{at}(?t, ?\mathit{from}),\ \texttt{road}(?\mathit{from}, ?\mathit{to})\}\\
&\quad \mathit{post}^+(o_1) := \{\texttt{at}(?t, ?\mathit{to})\}\\
&\quad \mathit{post}^-(o_1) := \{\texttt{at}(?t, ?\mathit{from})\}\\
o_2 &:= \texttt{Pick-up}(?t - \mathit{truck}, ?l - \mathit{location}, ?p - \mathit{package})\\
&\quad \mathit{pre}(o_2) := \{\texttt{at}(?t, ?l),\ \texttt{at}(?p, ?l)\}\\
&\quad \mathit{post}^+(o_2) := \{\texttt{in}(?p, ?t)\}\\
&\quad \mathit{post}^-(o_2) := \{\texttt{at}(?p, ?l)\}\\
o_3 &:= \texttt{Drop-off}(?t - \mathit{truck}, ?l - \mathit{location}, ?p - \mathit{package})\\
&\quad \mathit{pre}(o_3) := \{\texttt{at}(?t, ?l),\ \texttt{in}(?p, ?t)\}\\
&\quad \mathit{post}^+(o_3) := \{\texttt{at}(?p, ?l)\}\\
&\quad \mathit{post}^-(o_3) := \{\texttt{in}(?p, ?t)\}
\end{aligned}$$

Definition 2.1.5. $\mathcal{P}$ is the set of propositions that characterise problem states. Each $p \in \mathcal{P}$ is an instance of a predicate in Λ, grounded with objects in Σ, that respects the type terms in the predicate argument list. □

For example, the SIMPLE-LOGISTICS example has the following set of propositions $\mathcal{P} :=$

$$\begin{array}{cccc}
\{\texttt{at}(T_1, L_1), & \texttt{at}(T_1, L_2), & \texttt{at}(T_1, L_3), & \texttt{at}(T_1, L_4),\\
\texttt{at}(T_2, L_1), & \texttt{at}(T_2, L_2), & \texttt{at}(T_2, L_3), & \texttt{at}(T_2, L_4),\\
\texttt{at}(P_1, L_1), & \texttt{at}(P_1, L_2), & \texttt{at}(P_1, L_3), & \texttt{at}(P_1, L_4),\\
\texttt{at}(P_2, L_1), & \texttt{at}(P_2, L_2), & \texttt{at}(P_2, L_3), & \texttt{at}(P_2, L_4),\\
\texttt{in}(P_1, T_1), & \texttt{in}(P_1, T_2), & \texttt{in}(P_2, T_1), & \texttt{in}(P_2, T_2),\\
\texttt{road}(L_1, L_2), & \texttt{road}(L_2, L_1), & \texttt{road}(L_1, L_3) & \texttt{road}(L_3, L_1)\\
\texttt{road}(L_2, L_3), & \texttt{road}(L_3, L_2), & \texttt{road}(L_3, L_4) & \texttt{road}(L_4, L_3)\}
\end{array}$$

Definition 2.1.6. Formally, a *state* is a set of propositions – i.e., for state s, $s \subseteq \mathcal{P}$. If $p \in s$, then we say that p is true in s (also sometimes that s is characterised by p). □

Definition 2.1.7. $\mathcal{S}$ is the set of all states characterised by propositions in $\mathcal{P}$, including the starting state s_0 – i.e. $\mathcal{S} \subseteq 2^{\mathcal{P}}$. □

For example, in the running SIMPLE-LOGISTICS example the state shown in Figure 2.1 is represented by the following set of propositions:

$$\begin{array}{llll} \{\texttt{at}(T_1, L_1), & \texttt{at}(T_2, L_3), & \texttt{at}(P_1, L_2), & \texttt{at}(P_2, L_4), \\ \texttt{road}(L_1, L_2), & \texttt{road}(L_2, L_1), & \texttt{road}(L_1, L_3) & \texttt{road}(L_3, L_1) \\ \texttt{road}(L_2, L_3), & \texttt{road}(L_3, L_2), & \texttt{road}(L_3, L_4) & \texttt{road}(L_4, L_3)\} \end{array}$$

We are now ready to formally define a propositional planning problem.

Definition 2.1.8. A *propositional planning problem* is a 5-tuple $\Pi := \langle \Sigma, \Lambda, \mathcal{O}, s_0, \mathcal{G} \rangle$, where:

- Σ is a finite set of typed objects;
- Λ is a finite set of predicates with typed arguments;
- $\mathcal{O}$ is a finite, and usually small, set of operators of the form $\langle o, \mathit{pre}(o), \mathit{post}^+(o), \mathit{post}^-(o) \rangle$;
- $s_0 \in \mathcal{S}$ is the starting state; and
- $\mathcal{G} \subseteq \mathcal{S}$ is the goal description, a set of fluents where a state $s \in \mathcal{S}$ is a *goal state* iff $\mathcal{G} \subseteq s$.

□

Definition 2.1.9. An *action* a is a grounding of an operator o, with ground arguments $\vec{x}$. Action a has a set of ground preconditions $\mathit{pre}(a)$, positive postconditions $\mathit{post}^+(a)$, and negative postconditions $\mathit{post}^-(a)$. Each of these sets consists of propositions from $\mathcal{P}$ and is built by grounding the appropriate set of predicates in $\mathit{pre}(o)$, $\mathit{post}^+(o)$, or $\mathit{post}^-(o)$ with the objects in $\vec{x}$. □

For example, the action $a := \texttt{Drive}(T_1, L_1, L_2)$ has the following preconditions and postconditions:

$$\begin{array}{lcl} \mathit{pre}(a) & := & \{\texttt{at}(T_1, L_1),\ \texttt{road}(L_1, L_2)\} \\ \mathit{post}^+(a) & := & \{\texttt{at}(T_1, L_2)\} \\ \mathit{post}^-(a) & := & \{\texttt{at}(T_1, L_1)\} \end{array}$$

Definition 2.1.10. For a problem Π, $\mathcal{A}_o$ is the set of actions which instantiate operator o with the objects in Σ, respecting types. We then have $\mathcal{A} := \bigcup_{o \in \mathcal{O}} \mathcal{A}_o$. □

Continuing the running example, for the operator $o := \texttt{Drive}(?t, ?from, ?to)$, we have the following set of actions $\mathcal{A}_o :=$

$$\begin{aligned}\{&\texttt{Drive}(T_1, L_1, L_2),\ \texttt{Drive}(T_1, L_2, L_1),\ \texttt{Drive}(T_1, L_1, L_3)\ \texttt{Drive}(T_1, L_3, L_1),\\ &\texttt{Drive}(T_1, L_2, L_3),\ \texttt{Drive}(T_1, L_3, L_2),\ \texttt{Drive}(T_1, L_3, L_4)\ \texttt{Drive}(T_1, L_4, L_3),\\ &\texttt{Drive}(T_2, L_1, L_2),\ \texttt{Drive}(T_2, L_2, L_1),\ \texttt{Drive}(T_2, L_1, L_3)\ \texttt{Drive}(T_2, L_3, L_1),\\ &\texttt{Drive}(T_2, L_2, L_3),\ \texttt{Drive}(T_2, L_3, L_2),\ \texttt{Drive}(T_2, L_3, L_4)\ \texttt{Drive}(T_2, L_4, L_3)\}\end{aligned}$$

Now that we have defined actions and how they relate to propositions via conditions we need to define similar sets for the propositions.

Definition 2.1.11. For a planning problem Π and a proposition $p \in \mathcal{P}$, we define the following sets:

- $\mathit{pre}(p) := \{a | a \in \mathcal{A}, p \in \mathit{pre}(a)\}$;
- $\mathit{post}^{+}(p) := \{a | a \in \mathcal{A}, p \in \mathit{post}^{+}(a)\}$; and
- $\mathit{post}^{-}(p) := \{a | a \in \mathcal{A}, p \in \mathit{post}^{-}(a)\}$. $\square$

To describe how actions may be executed in a state, and the state that results from this execution, we require the notion of an action execution semantics.

Definition 2.1.12. For a problem Π, an action execution semantics defines the following:

- $\mathcal{A}(s) \subseteq \mathcal{A}$, the set of actions that can be executed individually at each state $s \in \mathcal{S}$. Under all execution semantics covered in this thesis, for all states $s \in \mathcal{S}$, $\mathcal{A}(s) := \{a \mid \mathit{pre}(a) \subseteq s, a \in \mathcal{A}\}$;
- A boolean relation $\mu : \mathcal{S} \times \mathcal{A} \times \mathcal{A}$ (a mutex or mutual exclusion relation) such that for some state s, and pair of distinct actions $\{a_1, a_2\} \subseteq \mathcal{A}(s) \times \mathcal{A}(s)$, $(s, a_1, a_2) \in \mu$, iff a_1 and a_2 cannot be executed in parallel in s; and
- The state $s' := s(\mathcal{A})$ that results from executing a set of non-mutex actions $\mathcal{A}' \subseteq \mathcal{A}(s)$ in parallel in state s. Under all execution semantics covered in this thesis, when $a \in \mathcal{A}(s)$ is executed in s, the resulting state $s' := \{s \backslash \mathit{post}^{-}(a) \cup \mathit{post}^{+}(a)\}$. Unless otherwise specified, we assume that the result of a valid execution of a set of actions $\mathcal{A}'$ in a state s is the state that results by first applying the delete effects of all actions, followed by the add effects of all actions. $\square$

It can be the case than an action a has a proposition p as both a positive and a negative postcondition – that is $\{\mathit{post}^{+}(a) \cap \mathit{post}^{-}(a)\} \neq \emptyset$. The obvious intent of including a pair of postconditions of this form is to enforce mutex relationships between actions. For example, in the IPC-6 domain ROVERS, the actions to communicate data add and delete a fluent `chanel-free`(c) (see Definition 2.1.32), ensuring that only a single communication can take place on a data channel at any one time.

Definition 2.1.13. For an action a, $\mathit{post}^{-}_{enf}(a)$ is the set of negative postconditions referring to propositions that actually end up false after the execution of an action. That is, $\mathit{post}^{-}_{enf}(a) := \{\mathit{post}^{-}(a) \backslash \mathit{post}^{+}(a)\}$. $\square$

Early planning-as-SAT approaches such as Kautz and Selman [35] used a serial execution semantics where a single action is executed at each step. Parallel action execution is often desirable, both to model parallelism inherent in planning domains, but also to reduce the number of steps required for a valid plan. In general, parallel action execution semantics are based on a notion of *conflicting* actions.

Definition 2.1.14. Let $\mu_c : \mathcal{A} \times \mathcal{A}$ be a Boolean relation that defines when actions *conflict*. Distinct actions a_1 and a_2 conflict, that is $(a_1, a_2) \in \mu_c$, iff $a_1 \neq a_2$ and:

- $\{\mathit{post}^{-}(a_1) \cap (\mathit{pre}(a_2) \cup \mathit{post}^{+}(a_2))\} \neq \emptyset$; or
- $\{\mathit{post}^{-}(a_2) \cap (\mathit{pre}(a_1) \cup \mathit{post}^{+}(a_1))\} \neq \emptyset$. $\square$

We can now define an execution semantics based on conflict.

Definition 2.1.15. *Conflict* execution semantics, extend Definition 2.1.12 with the following mutex relation. For a state s and two distinct actions $\{a_1, a_2\} \in \mathcal{A}(s) \times \mathcal{A}(s)$, $(s, a_1, a_2) \in \mu$ iff $(a_1, a_2) \in \mu_c$. $\square$

Under conflict mutex, all serial executions of a set of actions that can be executed in parallel in a state s are valid and lead to the same successor state s'.

A number of other parallel action execution semantics exist. In particular, a number of encodings will be explored in this work that use mutex relationships between actions based on the plangraph (Section 2.1.2). Additionally, along the lines of the $\exists$-step semantics of Rintanen et al. [57] and Wehrle and Rintanen [73], a novel parallel execution semantics that does not guarantee that optimal solutions are admitted is presented in Chapter 4.

We can now formally define a solution to a planning problem, known as a *plan*.

Definition 2.1.16. A *plan* π for a problem Π is a sequence of step-indexed sets of actions $\langle \mathcal{A}_0, \mathcal{A}_1, ..., \mathcal{A}_{h-1} \rangle$. The actions in $\mathcal{A}_0$ are executed at s_0 according to a given execution semantics, such as conflict mutex, and then each subsequent set of actions is similarly executed. A plan is valid, that is it is a solution to Π, iff the final state that results from the execution of the actions in $\mathcal{A}_{h-1}$ satisfies the goal $\mathcal{G}$. □

We are interested in representing propositional planning-as-SAT, where a planning problem is translated into a finite propositional formula. We are therefore required to bound the horizon of plans therefore define a variant of the problem bounded by this horizon.

Definition 2.1.17. Let the *horizon* $h := h(\pi)$ be the number of steps of plan π. □

Definition 2.1.18. Let Π_h be the h-step-bounded variant of planning problem Π. A problem Π_h only admits a plan π as a solution if $h(\pi) = h$. □

Continuing the running SIMPLE-LOGISTICS example, if the problem presented in Figure 2.1 is taken as a start state and we have as a goal $\mathcal{G} := \{\texttt{at}(P_1, L_4),\ \texttt{at}(P_2, L_1)\}$, then the following is a serial plan with horizon 9, where the numbers preceding actions represent the steps they are executed in:

$$
\begin{aligned}
0 &: \texttt{Drive}(T_1, L_1, L_2) \\
1 &: \texttt{Pick-up}(T_1, L_2, P_1) \\
2 &: \texttt{Drive}(T_1, L_2, L_3) \\
3 &: \texttt{Drive}(T_1, L_2, L_4) \\
4 &: \texttt{Drop-off}(T_1, L_4, P_1) \\
5 &: \texttt{Pick-up}(T_1, L_4, P_2) \\
6 &: \texttt{Drive}(T_1, L_4, L_3) \\
7 &: \texttt{Drive}(T_1, L_3, L_1) \\
8 &: \texttt{Drop-off}(T_1, L_1, P_2)
\end{aligned}
$$

Using conflict mutex, the following is a possible parallel plan for horizon 5:

$$
\begin{aligned}
0 &: \texttt{Drive}(T_1, L_1, L_2);\ \texttt{Drive}(T_2, L_3, L_4) \\
1 &: \texttt{Pick-up}(T_1, L_2, P_1);\ \texttt{Pick-up}(T_2, L_4, P_2) \\
2 &: \texttt{Drive}(T_1, L_2, L_3);\ \texttt{Drive}(T_2, L_4, L_3) \\
3 &: \texttt{Drive}(T_1, L_3, L_4);\ \texttt{Drive}(T_2, L_3, L_1) \\
4 &: \texttt{Drop-off}(T_1, L_4, P_1);\ \texttt{Drop-off}(T_1, L_1, P_2)
\end{aligned}
$$

When we solve planning problems there are a number of criteria we can use to determine the plans that will be accepted as solutions.

Definition 2.1.19. A *satisficing* planner returns any valid plan as a solution for planning problem Π (or Π_h). $\square$

Definition 2.1.20. An *optimal* planner returns a valid plan π^* as a solution for planning problem Π (or Π_h) such that there is no other valid plan π with $\xi(\pi) < \xi(\pi^*)$, where ξ is a bound function that assigns a value to each plan according to a given optimisation criteria. $\square$

2.1.1 Plan Quality Metrics

Given a choice for an action execution semantics, a number of different metrics for plan quality can be defined. Consider the two plans from the example in the previous section. The serial plan contains 9 actions and has a horizon of 9, while the parallel plan contains 10 actions but has a horizon of 5. The horizon of a plan is, in general, unrelated to its total execution time, or any other interesting quality metric. In particular, we may want to take into account the cost of executing a plan. If the cost of executing a plan is important and all actions cost the same to execute, then the number of actions in a plan can be used as a metric, otherwise actions need to be explicitly labelled with a cost.

Definition 2.1.21. Let $\mathcal{C} : \mathcal{A} \rightarrow \Re_0^+$ be a bounded cost function that assigns a non-negative cost-value to each action. This value corresponds to the cost of executing the action. $\square$

Definition 2.1.22. Where $\mathcal{A}^i$ is the set of actions at step i of a plan $\pi := \langle \mathcal{A}^0, \mathcal{A}^1, .., \mathcal{A}^{h-1} \rangle$, the cost of π, written $\mathcal{C}(\pi)$, is:

$$\mathcal{C}(\pi) := \sum_{i=0}^{h-1} \sum_{a \in \mathcal{A}^i} \mathcal{C}(a)$$ $\square$

In planning often not all goals can be achieved simultaneously. Such a problem is said to be *over-subscribed*. In such a case we can define a set of *soft* goals in addition to a set of regular *hard* goals. We measure the overall plan quality in terms of its *net-benefit*. That is, the sum of the utility of the achieved soft goals minus the cost of the actions executed to achieve them. In this case a valid plan must still achieve all regular goals.[1]

[1]Other, more expressive, formulations of net-benefit planning are possible – i.e. Russell and Holden [63].

Definition 2.1.23. Where $\mathcal{G}_{\mathcal{B}}$ is a set of soft goals, function $\mathcal{B} : \mathcal{G}_{\mathcal{B}} \rightarrow \Re_0^+$ is a bounded benefit function that assigns a utility to each proposition in $\mathcal{G}_{\mathcal{B}}$. This value corresponds to the utility of achieving a particular soft goal proposition. □

Definition 2.1.24. Where $\mathcal{A}^i$ is the set of actions at step i of a plan $\pi := \langle \mathcal{A}^0, \mathcal{A}^1, .., \mathcal{A}^{h-1} \rangle$, which entails a final state s^h at horizon h, the net benefit of π, written $\mathcal{B}(\pi)$, is

$$\mathcal{B}(\pi) := (\sum_{p \in \{s^h \cap \mathcal{G}_{\mathcal{B}}\}} \mathcal{B}(p)) - \mathcal{C}(\pi) \qquad \square$$

In the framework as described so far, a number of different conditions for plan optimality can be defined.

Definition 2.1.25. A plan π^* that is a solution to a problem Π_h is *step-optimal* iff there is no plan π with the same action execution semantics that is a solution to $\Pi_{h'}$, for $h' < h$. If the action execution semantics require that at most one action is executed at each plan step, then π^* is said to be *serial step-optimal*, otherwise it is *parallel step-optimal*. □

Next, we define notions of *cost-optimality* and *net-benefit-optimality* for both the fixed-horizon and the horizon-independent cases.

Definition 2.1.26. A plan π^* that is a solution to a problem Π_h is:

- *fixed-horizon cost-optimal* iff there is no plan π that is a solution to $\Pi_{h'}$, for any $h' \leq h$, such that $\mathcal{C}(\pi) < \mathcal{C}(\pi^*)$;
- *fixed-horizon net-benefit-optimal* iff there is no plan π that is a solution to $\Pi_{h'}$, for any $h' \leq h$, such that $\mathcal{B}(\pi) < \mathcal{B}(\pi^*)$.

A plan π^* that is a solution to a problem Π is:

- *horizon-independent cost-optimal* iff there is no plan π that is a solution to Π, such that $\mathcal{C}(\pi) < \mathcal{C}(\pi^*)$;
- *horizon-independent net-benefit-optimal* if there is no plan π that is a solution to Π, such that $\mathcal{B}(\pi) < \mathcal{B}(\pi^*)$. □

Finally, we draw attention to the fact that the definitions of horizon-independent cost-optimality and net-benefit-optimality are not dependent on the plan format (execution semantics). One key observation to be made is that the above optimality criteria are often conflicting. For example, a plan with minimal planning horizon is not guaranteed to be cost- or net-benefit-optimal. Indeed, in the general case there is no link between the planning horizon and plan cost or benefit.

2.1.2 The Plangraph

To increase the efficiency of representing and reasoning about planning problems, a number of preprocessing techniques may be applied to a problem before planning takes place. Of particular importance to this work is the *plangraph*, a data-structure that was devised for the efficient GRAPHPLAN framework of planning in STRIPS domains with no negative preconditions [8]. The plangraph captures necessary conditions for goal achievement, and is computationally cheap to build, taking polynomial time and space in $|\mathcal{P}|$ and $|\mathcal{A}|$. Before we define the plangraph itself we need to define a class of special no-operation or *noop* actions.

Definition 2.1.27. Let $noop_p$ be the *noop* action for proposition p. $noop_p$ has p as both a precondition and a positive postcondition. □

Definition 2.1.28. The *plangraph* is a directed *layered* graph for a planning problem Π_h. It has the following layers:

- For each step $t \in \{0, ..., h\}$, a layer labelled with t that contains a set of propositions $\mathcal{P}^t$, such that elements in the powerset of $\mathcal{P}^t$ are states which might be reachable in t steps from s_0; and
- For each step $t \in \{0, ..., h-1\}$, a layer labelled with t that contains the set of actions $\mathcal{A}^t$ that might be executable t steps into a plan. $\mathcal{A}^t$ also contains a noop action $noop_p^t$ for each proposition $p^t \in \mathcal{P}^t$.

The graph then has the following arcs:

- There is an arc from each action at layer t to each of its preconditions in layer t;
- There is an arc labelled *positive* from each action $a^t \in \mathcal{A}^t$ to the positive postconditions of a in layer $t+1$;
- There is an arc labelled *negative* from each action $a^t \in \mathcal{A}^t$ to each of the negative postconditions of a in layer $t+1$; and
- There is an arc between each pair of mutex actions at layer t and between each pair of mutex propositions at layer t.

□

Algorithm 1 The construction of a plangraph.

```
1: Input:
   - A planning problem $\Pi_h$;
2: $\mathcal{P}^0 \leftarrow s_0$;
3: **for** $t \in \{0, ..., h-1\}$ **do**
4:     $\mathcal{A}^t \leftarrow \{a^t | a \in \mathcal{A}, \mathit{pre}(a) \subseteq \mathcal{P}^t\} \cup \{noop_p^t | p^t \in \mathcal{P}^t\}$;
5:     $\mathcal{P}^{t+1} \rightarrow \{p^{t+1} | p \in \mathit{post}^+(a), a \in \mathcal{A}^t\}$;
6: **end for**
```

The nodes of a plangraph without mutex are constructed as shown in Algorithm 1. Arcs are then added between actions and their preconditions and postconditions for every layer.
The size of each layer of the plangraph monotonically increases until all reachable ground facts are present. The plangraph is made leaner (i.e., is characterised using fewer actions and fewer state propositions) by computing and exploiting state-dependent mutex relations between actions and propositions.

Definition 2.1.29. Let $\mu_{pg-a} : \mathbb{N}_0 \times \mathcal{A} \times \mathcal{A}$ be a Boolean relation that defines when actions are mutex at a layer of the plangraph. Distinct actions a_1 and a_2 at layer t are mutex in a plangraph, that is $(t, a_1, a_2) \in \mu_{pg-p}$ iff $(a_1, a_2) \in \mu_c$, or if a precondition of a_1 is mutex with a precondition of a_2 at t. Let $\mu_{pg-p} : \mathbb{N}_0 \times \mathcal{P} \times \mathcal{P}$ be a Boolean relation that defines when propositions are mutex at a layer of the plangraph. Distinct propositions p_1 and p_2 at step t are mutex, that is $(t, p_1, p_2) \in \mu_{pg-p}$, iff there is no action at t that adds both p_1 and p_2, and every action at t which adds p_1 is mutex with every action at t which adds p_2. $\square$

While constructing the plangraph, we can omit any action a at step t, if there is a pair of propositions in its precondition that are mutex at t. Another type of processing that may be performed to further prune the plangraph is called *neededness analysis*.

Definition 2.1.30. We determine *needed* actions as follows:

1. Any action $a^{h-1} \in \mathcal{A}^{h-1}$ is needed iff $\mathcal{G} \cap \mathit{post}^+(a) \neq \emptyset$ and $\{goal \cap \mathit{post}^-(a)\} = \emptyset$; and

2. For $t \in \langle h-1, ..., 0\rangle$:

    (a) A proposition $p^t \in \mathcal{P}^t$ is needed iff $p \in \mathit{pre}(a)$ for some needed action a^t; and

    (b) An action $a^t \in \mathcal{A}^t$ is needed iff there is a needed proposition p^t such that $p^t \in \mathit{post}^+(a)$.

Any vertices (and neighbouring arcs) that are not-needed can be safely removed from the plangraph. Finally, after computing the plangraph a number of *static* propositions in $\mathcal{P}$ may be removed.

Definition 2.1.31. A proposition p is *static* if $p \in s_0$ and for all $t \in 1, ..., h-1$, $\neg \exists a \in \mathcal{A}^t$ such that $p \in \mathit{post}^-(a)$. □

Definition 2.1.32. A proposition is a *fluent* if it is not *static*. □

Definition 2.1.33. $\mathcal{F}^t \subseteq \mathcal{P}^t$ refers to the set of fluents at layer t of a plangraph. □

Continuing our running SIMPLE-LOGISTICS example, the propositions which instantiate the predicate $\texttt{road}(?\mathit{from}, ?\mathit{to})$ are static and all other propositions are fluents.

The plangraph is a useful source for a compilation from planning to SAT because the mutex constraints yielded by plangraph analysis are: (1) useful, and feature, in state-of-the-art satisfiability procedures [33], and (2) are not redundant, because such mutex relations are not deduced independently by modern SAT procedures [56]. The plangraph forms the basis of many encodings from planning to SAT examined in this thesis, including those described in Section 3.5 and Chapters 4, 5, and 6.

Chapter 3

Planning-as-Satisfiability

SAT-based planning approaches solve a propositional planning problem Π by constructing and querying a sequence of CNF (conjunctive normal form) propositional formulae. Each CNF formula ϕ_h corresponds to a bounded version of the problem Π_h and admits solutions, if any, that correspond to h-step plans. The horizons used for generating CNFs are selected according to a *query strategy*. Informally, in the case of satisficing planning, a query strategy is used to find some horizon at which a plan exists, while in the case of optimal planning, additional queries may be required to ensure plan optimality. The propositional variables in each CNF ϕ_h represent, for each step from 0 to h, the truth of each fluent and, for each step from 0 to $h - 1$, the actions that are executed. The planning constraints are then posed in terms of these variables.

In this section we first briefly introduce SAT and then formally define the planning-as-SAT paradigm. We then examine a number of encodings from planning to SAT and a number of SAT-based planning systems.

3.1 Propositional Satisfiability (SAT)

The propositional satisfiability problem (SAT) requires determining if there is a valuation of the variables of a Boolean formula, usually in CNF, such that formula evaluates to *true*. A formula for which such a valuation exists is *satisfiable*, otherwise it is *unsatisfiable*.

Definition 3.1.1. A SAT problem $\langle V, \phi \rangle$ consists of the following:

- A set of n propositional variables $V := \{x_1, ..., x_n\}$; and
- A CNF propositional formula ϕ on the variables in V. □

Definition 3.1.2. A CNF formula corresponds to a conjunction over clauses, each of which corresponds to a disjunction over literals. A literal is either a propositional variable or its negation. Throughout this thesis we may also consider a CNF to be a set of clauses and a clause to be a set of literals. □

For example, we can have a SAT problem with the variables $\{x_1, x_2, x_3\}$ and a CNF $\phi \equiv$

$$((x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3))$$

Definition 3.1.3. A *valuation* $\mathcal{V}$ to a set of variables V is a set of assignments, one from each variable in V to either $true$ or $false$. □

Definition 3.1.4. A SAT problem $\langle V, \phi \rangle$ is satisfiable iff there exists a valuation $\mathcal{V}$ over the variables in V such that $\mathcal{V} \models \phi$. □

For example, the formula from the previous example is satisfied by the valuation:

$$\{x_1 \leftarrow true, x_2 \leftarrow true, x_3 \leftarrow false\}$$

A large number of solvers exist for SAT. These include local search solvers such as WALKSAT [65] and complete DPLL solvers such as RSAT [52] and PRECOSAT [7]. Early SAT-based planning approaches – such as Kautz and Selman [35] – experimented with local search solvers. Most modern SAT-based planning approaches have opted for complete DPLL based solvers, as they have generally outperformed local search solvers on the highly structured SAT problems produced by encoding planning problems. Complete SAT solvers also have the advantage of easily being able to prove unsatisfiability, and therefore have been useful for optimal planning.

Throughout this thesis we use the SAT solver PRECOSAT and a number of solvers based on RSAT. Early versions of PRECOSAT won both gold and silver medals in the applications track of the 2009 International SAT Competition. RSAT won the gold medals for the SAT+UNSAT and UNSAT problems of the 2007 International SAT Competition.

3.2 Planning-as-SAT Framework

We now present a framework for planning-as-SAT.

Definition 3.2.1. Let $\mathcal{T}$ be an *encoding*, that is a function which takes a bounded planning problem Π_h and returns a CNF formula. We say that $\mathcal{T}$ reduces Π_h to SAT and that $\phi_h := \mathcal{T}(\Pi_h)$ is an encoding of Π_h iff a valid h-step plan can be extracted from each satisfying valuation of ϕ_h and that ϕ_h is unsatisfiable only if there is no such plan. $\square$

In most existing approaches to SAT-based planning [38, 32, 39, 57] the compilations from planning to SAT are *direct* in the sense that a variable in the CNF corresponds to whether an action is executed at a plan step, or whether a fluent is true at a plan step. Other encoding schemes, such as those using split action representations [35, 18, 59, 60] (explored in Section 3.4 and Chapter 4); state based encodings [36, 34]; and causal encodings [34] use indirect encodings of actions, but importantly have the property that they, like the direct encodings, are *constructive* [34].[1]

Definition 3.2.2. A SAT encoding $\mathcal{T}$ is *constructive* iff for every planning problem Π and horizon h we have: (i) there is an algorithm that runs in at-most polynomial time in the size of $\mathcal{T}(\Pi_h)$ such that it will extract a valid plan for Π_h given a satisfying valuation for $\mathcal{T}(\Pi_h)$; and (ii) every valid plan π for Π_h corresponds to a satisfying valuation of $\mathcal{T}(\Pi_h)$. $\square$

Corollary 3.2.1. *For a constructive SAT encoding $\mathcal{T}$, $\mathcal{T}(\Pi_h)$ is unsatisfiable only if there is no plan for Π_h.*

The clauses in a planning CNF are used to encode planning constraints such as frame axioms, mutual exclusion axioms, and action preconditions and postconditions. In the earliest SAT-based planning approaches [35, 36] CNF encodings of planning problems were hand crafted. Later, SAT-based planning approaches generally compiled planning problems, usually supplied in PDDL, automatically into CNF. Some of these latter approaches (Sections 3.3 and 3.4) compile PDDL directly into CNF, while others (Section. 3.5) first compile PDDL into a plangraph, and then translate this graph and the additional constraints it implies into CNF.

Most current SAT-based planning systems plan in the step-optimal case, though some SAT-based planning systems exist for satisficing planning – i.e. Rintanen et al. [57], Wehrle and Rintanen [73], and Robinson et al. [59]. In either case, a scheme, known as a *query strategy*, is required to select the horizons at which CNFs are generated. In the satisficing case, a scheme must select horizons until a horizon h is found at which there exists a plan. In the step-optimal case a query strategy needs to find a horizon h^* at which a plan exists and prove that there is no horizon $h' < h$ at which a plan exists. The *de facto* standard planning strategy is the *ramp-up* query strategy in which a SAT solver is instantiated at incremental horizon lengths – i.e. $h = 1, 2, 3, ...$ It has been observed by Streeter and Smith [70], among others, that large refutation proofs – i.e

[1]For many planning domains state-based encodings are constructive. This is not, however, the case in general.

proving CNFs with large horizons unsatisfiable – are often the most time consuming part of step-optimal planning with SAT-based systems. This observation led to an alternative query strategy, $h = h_{\texttt{big}}, h_{\texttt{big}-1}, ..., 1$. This is known as *ramp-down*, also called the *backwards level reduction*. Here $h_{\texttt{big}}$ is obtained in practice by querying a satisficing planner. Better efficiency has been demonstrated for interleaved and parallel query strategies described by Rintanen [55], and for efficient variants of binary search described by Streeter and Smith [70].

3.3 Serial Encodings

The earliest encodings of planning as SAT allowed for serial (linear) planning, where exactly one action is executed at each step, up to the horizon h. In 1992, Kautz and Selman [35] introduced the first encoding of planning-as-SAT, a hand-crafted set of axioms that introduced many of the constraints that would appear in later automatically generated encodings. As well as constraints describing classical frame axioms, action preconditions and postconditions, mutual exclusion, and the start state and goal, they also used a set of domain-specific constraints to rule out certain unwanted state trajectories. Kautz and Selman [37] later returned to SAT-based planning with domain-specific control knowledge, a subject which is covered extensively in Chapter 6. They also experimented with using a split representation of actions to reduce the size of their encodings, an idea which will be covered in the next section and in more detail in Chapter 4. This work was followed by two papers in 1996 that introduced a set of compilations from planning problems described in the STRIPS formalism to SAT [36, 34]. As well as exploring split, parallel, and causal encodings, these papers presented new serial encodings that make use of *explanatory frame axioms* and also encodings that have either the action or state variables compiled away.

It has been noted [34] that via clausal resolution, occurrences of action variables (resp. fluent variables) may be completely removed from a planning CNF, leading to a state- (resp. action) based encoding. The resulting increase in the size of the encoding depends on the axioms used. This will be explored more in Section 3.5, when plangraph based encodings are examined. Along similar lines Giunchiglia, et al. [24] note that in certain encodings – i.e. those which include postconditions and full frame axioms – the values of the action variables in a planning CNF completely determine the values of the state variables. In this case SAT solvers can be directed to only branch on action variables. Serial planning was explored further in 1997 by Ernst, et al. [18] who presented the first fully automatic planning system MEDIC and empirically explored a number of serial encodings of planning-as-SAT that made use of different split action representations and different frame axioms.

For a bounded problem Π_h, an encoding $\mathcal{T}$ is given in terms of a set of axiom schemata. Each individual schema specifies how a set of planning constraints in $\phi_h = \mathcal{T}(\Pi_h)$ are derived from the actions and fluents of Π_h. The schemata make use of the following propositional variables, where $\mathcal{A}^t$ and $\mathcal{F}^t$ represent the actions and fluents respectively of the problem at step t after any preprocessing has been performed:

- For each step $t \in \{0, ..., h-1\}$, there is a propositional variable a^t for each action $a \in \mathcal{A}^t$; and
- For each step $t \in \{0, ..., h\}$, there is a propositional variable f^t for each fluent $f \in \mathcal{F}^t$.

There are then the following schemata.

Schema 3.3.1. Start State: For every fluent $f \in s_0$, there is the following clause:[2]

$$f^0 \qquad \square$$

Schema 3.3.2. Goal: For every $f \in \mathcal{G}$, there is the following clause:

$$f^h \qquad \square$$

Schema 3.3.3. Preconditions: For each step $t \in \{0, ..., h-1\}$ and each action $a \in \mathcal{A}^t$, there are the following clauses:

$$a^t \rightarrow \bigwedge_{f \in pre(a)} f^t \qquad \square$$

Schema 3.3.4. Postconditions: For each step $t \in \{0, ..., h-1\}$, and each action $a \in \mathcal{A}^t$, there are the following clauses:

$$(a^t \rightarrow \bigwedge_{f \in post^+(a)} f^{t+1}) \wedge (a^t \rightarrow \bigwedge_{f \in post^-_{enf}(a)} \neg f^{t+1}) \qquad \square$$

[2]This schema assumes that some reachability pruning has been performed, or at least that fluents not at the start state are not included in $\mathcal{F}^0$ and an action a_i^0 is omitted if $\{pre(a_i^0) \cup s_0\} = \emptyset$.

In the case that there is an action with a positive and a negative postcondition referring to the same fluent, then, according to the semantics of PDDL, we do not assert the negative postcondition. In the parallel case, such a negative postcondition may be used to enforce additional mutex relationships. In serial encodings, mutex must be enforced between all actions. This can be achieved by explicitly ruling out the parallel execution of all pairs of actions. This may be achieved with fewer clauses by recognising those pairs of actions that are prevented from executing in parallel due to conflicting effects.

Definition 3.3.1. Let $\mu_{ce} : \mathcal{A} \times \mathcal{A}$ be a Boolean relation such that distinct actions $(a_1, a_2) \in \mu_{ce}$ iff:

$$\{post^+(a_1) \cap post^-_{enf}(a_2)\} \cup \{post^+(a_2) \cap post^-_{enf}(a_1)\} \neq \emptyset \qquad \square$$

To assert action mutex we require one of the following two schemata.

Schema 3.3.5. Full Action Mutex Strong: For each step $t \in \{0, ..., h-1\}$, and every pair of distinct actions $\{a_1, a_2\} \subseteq \mathcal{A}^t \times \mathcal{A}^t$, there is the following clause:

$$\neg a_1^t \vee \neg a_2^t \qquad \square$$

Schema 3.3.6. Full Action Mutex Weak: For each step $t \in \{0, ..., h-1\}$, and every pair of distinct actions $\{a_1, a_2\} \subseteq \mathcal{A}^t \times \mathcal{A}^t$, where $(a_1, a_2) \in \mu_{ce}$ we have a clause:

$$\neg a_1^t \vee \neg a_2^t \qquad \square$$

Schema 3.3.7. At Least One Action: For each step $t \in \{0, ..., h-1\}$, there is the following clause:

$$\bigvee_{a^t \in \mathcal{A}^t} a^t \qquad \square$$

The following frame axioms ensure that fluents only change truth value if caused to do so by an action.

Schema 3.3.8. Classical Frame Axioms: For each step $t \in \{0, ..., h-1\}$, each action $a \in \mathcal{A}^t$ and each fluent $f \in \mathcal{F}^t$, such that $f \notin \{post^+(a) \cup post^-(a)\}$, there are the following clauses:

$$[(a^t \wedge p^t) \rightarrow p^{t+1}] \ \wedge \ [(a^t \wedge \neg p^t) \rightarrow \neg p^{t+1}] \qquad \square$$

Classical frame axioms may be replaced by, the generally more compact, *explanatory frame axioms*, which explain the change in truth value of a fluent by requiring the execution of an appropriate action.

Schema 3.3.9. Positive Explanatory Frame Axioms: For each step $t \in \{1, ..., h\}$ and each fluent $f \in \mathcal{F}^t$, there is the following clause:

$$f^t \rightarrow (f^{t-1} \vee \bigvee_{a \in post^{+}(f)} a^{t-1}) \qquad \square$$

Schema 3.3.10. Negative Explanatory Frame Axioms: For each step $t \in \{1, ..., h\}$ and each fluent $f \in \mathcal{F}^t$, there is the following clause:

$$\neg f^t \rightarrow (\neg f^{t-1} \vee \bigvee_{a \in post^{-}_{enf}(f)} a^{t-1}) \qquad \square$$

Given the preceding schemata, we define the following two serial encodings of a bounded planning problem Π_h:

1. Serial explanatory encoding $\mathcal{T}^{Se}(\Pi_h)$, which uses the following schemata:
   - Start state and goal constraints (Schemata 3.3.1 and 3.3.2);
   - Preconditions and postconditions (Schemata 3.3.3 and 3.3.4);
   - Action mutex (Schemata 3.3.5 or 3.3.6);
   - Explanatory frame axioms (Schemata 3.3.9 and 3.3.10).
2. Serial classical encoding $\mathcal{T}^{Sc}(\Pi_h)$, which uses the following schemata:
   - Start state and goal constraints (Schemata 3.3.1 and 3.3.2);
   - Preconditions and postconditions (Schemata 3.3.3 and 3.3.4);
   - Action mutex (Schemata 3.3.5 or 3.3.6);
   - At-least-one (Schema 3.3.7); and
   - Classical frame axioms (Schema 3.3.8)

Some planning systems (such as SATPLAN-06 [39]) omit postconditions axioms. In general, postconditions (Schema 3.3.4) may be omitted if explanatory frame axioms (Schemata 3.3.9 and 3.3.10) are used. In this case, a plan may contain spurious actions which have postconditions that do not hold. These spurious actions may simply be removed from the plan. Empirically we have found that omitting postcondition constraints provides no benefit in terms of performance, so to simplify the exposition we include postcondition axioms in all relevant encodings for the rest of this thesis.

With flat serial encodings, at-least-one clauses tend to be very long – i.e. they have $|\mathcal{A}^t|$ literals. They therefore generally do not provide many opportunities for unit propagation in SAT solvers and do not contribute significantly to the efficiency of solving the resulting CNFs. Additionally, at-least-one axioms may result in no plan existing if a horizon is selected that is too large, unless an encoding includes noop actions, though we do not deal with that here. We therefore omit at-least-one axioms unless they are required by the use of classical frame axioms.

In general, as individual actions tend to affect a small number of fluents, and fluents tend to be affected by a relatively small number of actions, explanatory frame axioms are preferred to classical frame axioms. In this setting, for each step t there are $\mathrm{O}(|\mathcal{A}^t||\mathcal{F}^t|)$ classical frame axiom clauses with length 3, while there are $\mathrm{O}(|\mathcal{F}^t|)$ explanatory frame axiom clauses with length $\mathrm{O}(2 + k)$, where k is the larger of the maximum number of actions that add any fluent or the maximum number of actions that delete any fluent. As we are restricted to planning with positive goals and preconditions, negative explanatory frame axioms may be omitted. In general, this is not desirable as (i) they tend to contribute little to the overall size of a CNF and (ii) they significantly increase unit propagation during solving and therefore planning efficiency [67]. Here, we will refer to both classical frame axioms and frame axioms that include both positive and negative explanatory axioms as *full frame axioms*.

Theorem 3.3.1. *The serial explanatory encoding $\mathcal{T}^{Se}$ of planning into SAT is constructive for planning problems where we suppose a serial action execution semantics.*

Proof. Addressing condition (i), we sketch a plan extraction algorithm which takes a satisfying valuation $\mathcal{V}$ of $\mathcal{T}^{Se}(\Pi_h)$ and extracts a valid plan. A satisfying valuation $\mathcal{V}$ specifies a sequence of sets of true fluent variables $\mathcal{F}_\top := \langle \mathcal{F}^0_\top, ..., \mathcal{F}^h_\top \rangle$ and a sequence of sets of true action variables $\mathcal{A}_\top := \langle \mathcal{A}^0_\top, ..., \mathcal{A}^{h-1}_\top \rangle$. Our algorithm takes $\mathcal{A}_\top$ to represent a plan π_h, and $\mathcal{F}_\top$ to represent the sequence of states visited when π_h is realised, expressed here as $\mathcal{S}_\top := \langle s^0_\top, ..., s^h_\top \rangle$. It is clear that this algorithm runs in linear time in the size of $\mathcal{A}_\top$ and $\mathcal{F}_\top$. We therefore are left to demonstrate that π_h is a valid serial plan for Π_h. As we assume that $\mathcal{V}$ is a satisfying valuation of $\mathcal{T}^{Se}(\Pi_h)$, all clauses implied by the Schemata of $\mathcal{T}^{Se}$ are satisfied. From Definition 2.1.16 we require that:

1. $s^0_{\top} = s_0$. This holds because for each $f_i \in s_0$, we have a unit clause (Schema 3.3.1) asserting that $f^0_i \in \mathcal{V}$;

2. $\mathcal{G} \subseteq s^h_{\top}$. This holds because we have a unit clause (Schema 3.3.2) asserting that for every fluent $f_i \in \mathcal{G}$, $f^h_i \in \mathcal{V}$;

3. For all steps t, the set of actions $\mathcal{A}^t_{\top}$ obeys the serial execution semantics and therefore $|\mathcal{A}^t_{\top}| \leq 1$. If $\mathcal{T}^{Se}$ employs strong mutex clauses (Schema 3.3.5), we have $|\mathcal{A}^t_{\top}| \leq 1$ because $\mathcal{T}^{Se}$ prescribes a clause $\neg a_1 \vee \neg a_2$ for any two distinct actions a_1 and a_2. If $\mathcal{T}^{Se}$ employs weak mutex clauses (Schema 3.3.6), we also have that $|\mathcal{A}^t_{\top}| \leq 1$ because every pair of distinct actions is either ruled out by a mutex clause (as in the case of strong mutex clauses) or by postcondition clauses (Schema 3.3.4), according to Definition 3.3.1; and

4. For all steps t, every action $a^t \in \mathcal{A}^t_{\top}$ is applicable in the state $s^t_{\top}$. This is implied by Schema 3.3.3, which prescribes $\mathit{pre}(a^t) \subseteq s^t_{\top}$. Where $\mathcal{A}^t_{\top} := \{a^t\}$, we require that state $s^{t+1}_{\top} := \{\{s^t_{\top} \backslash \mathit{post}^-_{enf}(a^t)\} \cup \mathit{post}^+(a^t)\}$. This holds because the explanatory frame axioms (Schemata 3.3.9 and 3.3.10) specify that for each fluent $f \in \mathcal{F}$, $f_i \in s^{t+1}_{\top}$ iff $f_i \in \mathit{post}^+(a^t)$ or both $f_i \in s^t_{\top}$ and $f_i \notin \mathit{post}^-_{enf}(a^t)$;

It remains to show (ii), that every valid sequential plan π_h for Π_h can be translated into a satisfying valuation $\mathcal{V}$ of $\mathcal{T}^{Se}(\Pi_h)$. Following the logic of the above proof for (i), it is easy to see that the assignment $\mathcal{V}$ that represents the fluents $\mathcal{F}_{\top}$ and actions $\mathcal{A}_{\top}$, implied by the plan π_h, satisfies the clauses in the encoding $\mathcal{T}^{Se}(\Pi_h)$. □

Theorem 3.3.2. *The serial classical frame encoding $\mathcal{T}^{Sc}$ of planning into SAT is constructive for planning problems where we suppose a serial action execution semantics.*

Proof. This proof is similar to the proof of Theorem 3.3.1, with the following modifications. $\mathcal{T}^{Sc}$ includes Schema 3.3.7, which modifies point 3 of (i) in Theorem 3.3.1 to ensure that $|\mathcal{A}^t_{\top}| = 1$ for each $\mathcal{A}^t_{\top} \in \mathcal{A}_{\top}$. This implies that a plan π_h extracted from $\mathcal{T}^{Sc}(\Pi_h)$ will be exactly of length h, which is consistent with the serial execution semantics. Point 4 of (i) in Theorem 3.3.1 shows that with explanatory frame axioms each set of actions $\mathcal{A}^t_{\top} \in \mathcal{A}_{\top}$ reaches the state $s^{t+1}_{\top}$ when applied to $s^t_{\top}$. Here, from Schema 3.3.7, we have that $\mathcal{A}^t_{\top} := \{a^t\}$. The postcondition axioms (Schema 3.3.4) ensure that fluents in $\mathit{post}^-_{enf}(a^t) \cup \mathit{post}^+(a^t)$ take the correct value in $s^{t+1}_{\top}$ and the classical frame axioms (Schema 3.3.8) ensure that a fluent $f_i \in \mathcal{F} \backslash \{\mathit{post}^-_{enf}(a) \cup \mathit{post}^+(a)\}$ are in $s^{t+1}_{\top}$ iff $f_i \in s^t_{\top}$. □

3.4 Split Encodings

A fundamental limitation of SAT-based planning approaches is the often prohibitively sized CNFs the generate when solving many non-trivial problems. This limitation is particularly severe when using serial encodings like those presented in the previous section, due to the large horizon that is often required to find plans. To mitigate this size blow-up, the MEDIC system [18] uses a split representation of actions [35]. Compared with flat representations, for direct-encodings of the serial planning problem, fewer variables are required to describe actions. This section describes a number of existing split-action representations for serial planning. Chapter 4 demonstrates how split action representations can be used with parallel action execution. In decreasing order in the number of variables required for serial planning, existing action representations include: (1) *simply split*, (2) *overloaded split*, and (3) *bitwise*.

In the *simply split* case, action execution is encoded by variables that represent assignments to individual operator parameters. For the rest of this thesis, we have the following definitions.

Definition 3.4.1. Let $?x{\leftarrow}X$ mean that operator argument $?x$ is assigned to object X. □

For example, $\texttt{Drive}(T, L_1, L_2)$, which instantiates the operator $\texttt{Drive}(?t, ?from, ?to)$, can be represented by the following set of assignments:

$$\{?t{\leftarrow}T, ?from{\leftarrow}L_1, ?to{\leftarrow}L_2\}$$

Then, writing $\texttt{Drive}\{?t{\leftarrow}T\}$ for the propositional variable representing the assignment $?t{\leftarrow}T$ for operator $\texttt{Drive}$, we have that $\texttt{Drive}(T, L_1, L_2)$ is represented by the conjunction:

$$\texttt{Drive}\{?t{\leftarrow}T\} \wedge \texttt{Drive}\{?from{\leftarrow}L_1\} \wedge \texttt{Drive}\{?to{\leftarrow}L_2\}$$

Ignoring type restrictions on assignments, in the split case grounding an n-ary operator over Σ results in $\mathrm{O}(n|\Sigma|)$ variables compared to $\mathrm{O}(|\Sigma|^n)$ in the direct case.

In the *overloaded split* case, all actions share assignment variables, regardless of which operator they instantiate. First, there is a set of arguments $\{?x_0, ..., ?x_n\}$, shared amongst all operators, where n is the maximum arity of any operator in the problem description. Here $?x_0$ represents the operator of the executed action, while $?x_1, ..., ?x_n$ represent the arguments in position 1 through n, of each operator. If an operator has less than n arguments, then the additional shared arguments are ignored. Each shared argument $?x_i \in ?x_1, ..., ?x_n$ has a set of assignments such that each shared assignment $?x_i{\leftarrow}X$ is mapped to a grounding of the ith parameter of each operator

$o \in \mathcal{O}$. This encoding requires $\mathrm{O}((n+1)|\Sigma|)$ variables to encode all operators at a step, where n is the maximum arity of any operator.

Finally for the *bitwise* case, each action is mapped to an integer n in base-2. In particular, for increasing $i = 0, ..., \lceil \log_2(|\mathcal{A}|) \rceil$, taking p_i to be the ith significant bit of n we represent $5 = \texttt{Move}(A, B, C)$ with conjunct $p_0 \wedge \neg p_1 \wedge p_2$. A bitwise encoding requires the fewest variables to represent actions, and naively should produce the easiest SAT problem among the three representations. However, Ernst et al. [18] and others have found that the performance of encodings based on bitwise representation is generally worse than those based on other representations. This is mainly due to the fact that constraints in bitwise encodings cannot easily be *factored*. Due to the poor performance of the bitwise encoding scheme, and due to the similarity of the simple and overloaded splitting schemes and the relative simplicity of the simple splitting scheme, for the remainder of this section we will restrict our attention to the simply split case.

For all of the previously described split action representations, the conjunctions that describe actions participate in the existing constraints in place of the original action variables. For example, using a simply split action representation, at each step t we have the following precondition and postcondition constraints for the action $\texttt{Drive}(T, L_1, L_2)$:

$$\begin{aligned}
(\texttt{Drive}\{?t{\leftarrow}T\}^t \wedge \texttt{Drive}\{?\mathit{from}{\leftarrow}L_1\}^t \wedge \texttt{Drive}\{?\mathit{to}{\leftarrow}L_2\}^t) &\rightarrow \texttt{at}(T, L_1)^t \\
(\texttt{Drive}\{?t{\leftarrow}T\}^t \wedge \texttt{Drive}\{?\mathit{from}{\leftarrow}L_1\}^t \wedge \texttt{Drive}\{?\mathit{to}{\leftarrow}L_2\}^t) &\rightarrow \texttt{at}(T, L_2)^{t+1} \\
(\texttt{Drive}\{?t{\leftarrow}T\}^t \wedge \texttt{Drive}\{?\mathit{from}{\leftarrow}L_1\}^t \wedge \texttt{Drive}\{?\mathit{to}{\leftarrow}L_2\}^t) &\rightarrow \neg\texttt{at}(T, L_1)^{t+1}
\end{aligned}$$

Some constraints require exponentially many clauses to represent when actions are substituted for conjunctions of groundings. In particular the explanatory frame axioms (Schemata 3.3.9 and 3.3.10) and the at-least-one axiom (Schema 3.3.7). With the explanatory frame axioms this problem is almost entirely eliminated by factoring constraints. In the case of the at-least-one axiom, this problem can be eliminated by replacing the existing schema with the following:

Schema 3.4.1. Split At-least-One Axiom: For each step $t \in \{1, ..., h\}$, each operator $o := O(?x_1 - \mathit{type}_1, ..., ?x_k - \mathit{type}_k) \in \mathcal{O}$, and each $?x_i \in \{?x_1 - \mathit{type}_1, ..., ?x_k - \mathit{type}_k\}$, there is the following clause:

$$\bigvee_{X \in \mathrm{D}(\mathit{type}_i)} O\{?x_i{\leftarrow}X\}^t \qquad \square$$

Using a simply split action representation, a constraint which mentions a fluent f can be *factored* by including, for an action conjunct, only those groundings of arguments that occur in

f. For example, for the operator $\texttt{Drive}(?t, ?from, ?to)$ the preconditions:

$$\begin{aligned}
\texttt{Drive}(T, L_1, L_2) &\rightarrow \texttt{at}(T, L_1)^t \\
\texttt{Drive}(T, L_1, L_3) &\rightarrow \texttt{at}(T, L_1)^t \\
\texttt{Drive}(T, L_1, L_4) &\rightarrow \texttt{at}(T, L_1)^t \\
&\ldots
\end{aligned}$$

can all be represented with a single clause:

$$(\texttt{Drive}\{?t{\leftarrow}T\}^t \wedge \texttt{Drive}\{?from{\leftarrow}L_1\}^t) \rightarrow \texttt{at}(T, L_1)^t$$

Splitting and factoring can greatly reduce the size of a CNF encoding of serial planning. For example, to encode problem 29 from the IPC-2 domain BLOCKS with the serial encoding described in the previous section[3] for horizon 38, it requires more than 17 thousand variables and more than 650 thousand clauses. To encode the same problem with the serial version of the simply split encoding from Chapter 4 for the same horizon, it takes only 9.2 thousand variables and 181.8 thousand clauses. In the parallel case, the previously described split representations suffer from the problem of *interference* and therefore may not be used to find step-optimal plans. This is explored in detail in Section 4.1.3.

3.5 Plangraph-Based Encodings

From the perspective of efficiency, the linearity of the previously described direct and split encodings is a disadvantage. Plans for many non-trivial planning problems include many hundreds or thousands of actions. Due to memory constraints, it is often impractical to generate a CNF for the horizons required for such plans. This problem can be mitigated by using an action execution semantics which allows certain actions to be executed in parallel. The parallel execution of actions may also be useful in modelling parallelism inherent in some planning domains.[4] Kautz and Selman [36] and Kautz, et al. [34] introduced a number of parallel encodings of planning based on the plangraph. These encodings formed the basis of the BLACKBOX planning system [38] and its successors SATPLAN-04 and SATPLAN-06 [39], LRP [48], MAXPLAN [12], CRICKET [53], SOLE [60], amongst others. Some of these planning systems, particularly BLACKBOX, SATPLAN-04, and SATPLAN-06 allow a number of different encodings, including action-based

[3]With no noop actions, action postconditions included, full explanatory frame axioms, and weak serial mutex.

[4]Under most parallel action execution semantics, parallel plans may be post-serialised, if required.

and state-based encodings.[5] State-based encodings are generally impractical, as compiling away actions can lead to an exponential blowup in the size of the encoding [34]. Action-based encodings will not be covered in this exposition. This is primarily because throughout this thesis we consistently found no benefit in excluding state variables. Additionally, all of the existing action-based encodings, while state variables are omitted, either noop actions, or equivalent *maintain* actions for fluents are included. In an encoding of this form, when a state fluent f^t is true, the action $noop_f^{t-1}$ is logically equivalent to the variable f^{t-1}. The only practical effect, then, of resolving away state-variables in these encodings is to derive a set of additional causes that relate actions in consecutive steps. These constraints are already implied by *unit propagation* and therefore offer no additional efficiency in most modern SAT procedures.

We next present a set of schemata for parallel encodings that are based on the plangraph. For a bounded problem Π_h, we first build a plangraph such that there are actions $\mathcal{A}^t$ for $t \in \{0, ..., h-1\}$ and fluents $\mathcal{F}^t$ for $t \in \{0, ..., h\}$. We then have a propositional variable for each action a in each $\mathcal{A}^t$. and a propositional variable for each fluent f in each $\mathcal{F}^t$. The inclusion of noop actions in these encodings is optional and does not change the following schemata, unless it is stated otherwise.

Schema 3.5.1. Start State: For all $f \in s_0$, there is the following clause:

$$f^0 \qquad \square$$

Schema 3.5.2. Goal: For all $f \in \mathcal{G}$, there is the following clause:

$$f^h \qquad \square$$

Schema 3.5.3. Preconditions: For each step $t \in \{0, ..., h-1\}$ and every action $a \in \mathcal{A}^t$, there are the following clauses:

$$a^t \rightarrow \bigwedge_{f \in pre(a)} f^t \qquad \square$$

Schema 3.5.4. Postconditions: For each step $t \in \{0, ..., h-1\}$, and every action a, there are the following clauses:

$$(a^t \rightarrow \bigwedge_{f \in post^+(a)} f^{t+1}) \wedge (a^t \rightarrow \bigwedge_{f \in post^-_{enf}(a)} \neg f^{t+1}) \qquad \square$$

[5] Here, state-based encodings refers to domain-independent encodings that do not explicitly make use of action variables, rather than the domain-specific state-based encodings of Kautz and Selman [36].

As described for the serial case, some delete effects may not be enforced. Additionally, we have two mutex schemata which, while they encode the same mutex relations, require differing number of clauses. The first, described here as strong mutex, explicitly represents as binary clauses all mutex relations inferred from the plangraph. The second, described here as weak mutex, represents only those mutex relations that are not already enforced by other constraints. Its definition is therefore dependent on the constraints selected to be part of an encoding. The definition below assumes that constraints are included for fluent mutex and for action effects and is based on the definition in Sideris and Dimopoulos [67].

Definition 3.5.1. Let $\mu_{enfs} : \mathbb{N}_0 \times \mathcal{A} \times \mathcal{A}$ be a Boolean relation such that, for step t and distinct actions a_1 and a_2, $(t, a_1, a_2) \in \mu_{enfs}$ iff:

- $\{\mathit{pre}(a_1) \cap \mathit{post}^-(a_2)\} \neq \emptyset$;
- There is a pair of distinct fluents f_1, f_2 such that $f_1 \in \mathit{pre}(a_1)$, $f_2 \in \mathit{pre}(a_2)$, and $(t, f_1, f_2) \in \mu_{pg-p}$; and
- $\{\mathit{post}^+(a_1) \cap \mathit{post}^-_{enf}(a_2)\} \neq \emptyset$. □

Schema 3.5.5. Action Mutex Strong: For each step $t \in \{0, ..., h-1\}$, and every pair of distinct actions a_1 and a_2, where $(t, a_1, a_2) \in \mu_{enfs}$, there is the following clause:

$$\neg a_1^t \wedge \neg a_2^t \qquad \square$$

Definition 3.5.2. Let $\mu_{enfw} : \mathbb{N}_0 \times \mathcal{A} \times \mathcal{A}$ be a Boolean relation such that, for step t and distinct actions a_1 and a_2, $(t, a_1, a_2) \in \mu_{enfw}$ iff:

- $\{\mathit{pre}(a_1) \cap \mathit{post}^-(a_2)\} \neq \emptyset$;
- There is no pair of distinct fluents f_1 and f_2 such that $f_1 \in \mathit{pre}(a_1)$, $f_2 \in \mathit{pre}(a_2)$, and $(t, f_1, f_2) \in \mu_{pg-p}$;
- There is no pair of distinct fluents f_1 and f_2 such that $f_1 \in \mathit{post}^+(a_1)$, $f_2 \in \mathit{post}^+(a_2)$, and $(t+1, f_1, f_2) \in \mu_{pg-p}$; and
- $\{\mathit{post}^+(a_1) \cap \mathit{post}^-_{enf}(a_2)\} = \emptyset$. □

Schema 3.5.6. Action Mutex Weak: For each step $t \in \{0, ..., h-1\}$, and every pair of actions a_1 and a_2 , such that $(t, a_1, a_2) \in \mu_{enfw}$, there is the following clause:

$$\neg a_1^t \wedge \neg a_2^t \qquad \square$$

Schema 3.5.7. Fluent Mutex: For each step $t \in \{0, ..., h\}$, and every pair of fluents f_1 and f_2, such that $(t, f_1, f_2) \in \mu_{pg-p}$, there is the following clause:

$$\neg f_1^t \vee \neg f_2^t \qquad \square$$

Schema 3.5.8. Positive Explanatory Frame Axioms: For each step $t \in \{1, ..., h\}$ and all $f \in \mathcal{F}^t$, there is the following clause:

$$f^t \rightarrow (f^{t-1} \vee \bigvee_{a \in \mathtt{make}(f)} a^{t-1}) \qquad \square$$

Schema 3.5.9. Negative Explanatory Frame Axioms: For each step $t \in \{1, ..., h\}$, and all $f \in \mathcal{F}^t$, there is the following clause:

$$\neg f^t \rightarrow (\neg f^{t-1} \vee \bigvee_{a \in \mathtt{break}(f)} a^{t-1}) \qquad \square$$

The following schema produces constraints similar to the positive explanatory frame axioms, except that it includes noop actions instead of referring to fluents in a previous step. This schema therefore requires the inclusion of noop actions.

Schema 3.5.10. Backwards-Chaining Frame Axioms: For each step $t \in \{1, ..., h\}$, and all fluents $f \in \mathcal{F}^t$, there is the following clause:

$$f^t \rightarrow \bigvee_{a \in \mathtt{make}(f)} a^{t-1} \qquad \square$$

For an encoding $\mathcal{T}(\Pi_h)$ of a bounded planning problem Π_h to be valid we require that the following axioms be included:

- Start state and goal constraints (Schemata 3.5.1 and 3.5.2);
- Preconditions and postconditions (Schemata 3.5.3 and 3.5.4)[6]
- Action mutex (Schemata 3.5.5 or 3.5.6);
- Fluent mutex (Schemata 3.5.7);

[6]As in the serial case, effect axioms are optional. We do not consider omitting them here.

- Frame axioms. If the encoding uses noop actions, then backwards chaining frame axioms (Schema 3.5.10) may be used, otherwise explanatory frame axioms (Schemata 3.3.9 and 3.3.10) are included. As in the serial case, negative explanatory frame axioms may be omitted, but this is generally not desirable [67].

Rather than describing all existing plangraph-based encodings of planning-as-SAT, we note that Sideris and Dimopoulos [67] describe many and present an encoding, which importantly includes negative explanatory frame axioms. Their encoding uses noop actions and includes Schemata 3.5.1; 3.5.2; 3.5.3; 3.5.4; 3.5.6; 3.5.7; 3.5.10; and 3.3.10. They note that in their encoding strong action mutex constraints (Schema 3.5.5) and *londex* (long distance mutual exclusion constraints) [12] are redundant – i.e. are entailed by *unit propitiation.*

It is easy to see that a similar encoding, the one that we use throughout this thesis, where noop actions are omitted and positive explanatory frame axioms (Schema 3.3.9) are used in place of backwards-chaining frame axioms (Schema 3.5.10), has the same property. We denote such an encoding $\mathcal{T}^{Pe}$ and prove that it is constructive.

Theorem 3.5.1. *The parallel explanatory encoding $\mathcal{T}^{Pe}$ of planning into SAT is constructive for planning problems where we suppose a conflict action execution semantics.*

Proof. Addressing condition (i), we sketch a plan extraction algorithm which takes a satisfying valuation $\mathcal{V}$ of $\mathcal{T}^{Pe}(\Pi_h)$ and extracts a valid plan. Here $\mathcal{V}$ specifies a sequence of sets of true fluent variables $\mathcal{F}_\top := \langle \mathcal{F}_\top^0, ..., \mathcal{F}_\top^h \rangle$, and a sequence of sets of true action variables $\mathcal{A}_\top := \langle \mathcal{A}_\top^0, ..., \mathcal{A}_\top^{h-1} \rangle$. Our algorithm takes $\mathcal{A}_\top$ to represent a plan π_h, and $\mathcal{F}_\top$ to represent the sequence of states visited when that plan is realised, expressed here as $\mathcal{S}_\top := \langle s_\top^0, ..., s_\top^h \rangle$. It is clear that this algorithm runs in linear time in the size of $\mathcal{A}_\top$ and $\mathcal{F}_\top$. We now are left to demonstrate that π_h is a valid parallel plan for Π_h. As we assume that $\mathcal{V}$ is a satisfying valuation of $\mathcal{T}^{Se}(\Pi_h)$, all clauses implied by the Schemata of $\mathcal{T}^{Pe}$ are satisfied. From Definition 2.1.16 we require that:

1. $s_\top^0 = s_0$. This holds because for each $f_i \in s_0$, we have a unit clause (Schema 3.5.1) asserting that $f_i^0 \in \mathcal{V}$;

2. $\mathcal{G} \subseteq s_\top^h$. This holds because we have a unit clause (Schema 3.5.2) asserting that for every fluent $f_i \in \mathcal{G}$, $f_i^h \in \mathcal{V}$;

3. For all steps t, the set of actions $\mathcal{A}_\top^t$ obeys the conflict execution semantics (Definition 2.1.15). We require that for any two distinct actions a_1 and a_2 in $\mathcal{A}_\top^t$ $(a_1, a_2) \notin \mu_c$. The

postcondition clauses (Schema 3.5.4) and the mutex clauses (Schema 3.5.6) (by Definition 3.5.2) prevent both a_1 and a_2 from being in $\mathcal{A}^t_\top$ iff $(a_1, a_2) \in \mu_c$; and

4. For each step t and set of acions $\mathcal{A}^t_\top$, we have that each action $a^t \in \mathcal{A}^t_\top$ is applicable in the state $s^t_\top$ according to Definition 2.1.12. This is true as Schema 3.5.3 ensures that $\mathit{pre}(a) \subseteq s^t_\top$. We also require that $\mathcal{A}^t_\top$ reaches the state $s^{t+1}_\top$ when applied to $s^t_\top$. First, let $\mathit{post}^-_{\mathit{enf}}(\mathcal{A}^t_\top) := \{\mathit{post}^-_{\mathit{enf}}(a^t) | a^t \in \mathcal{A}^t_\top\}$ and $\mathit{post}^+(\mathcal{A}^t_\top) := \{\mathit{post}^+(a^t) | a^t \in \mathcal{A}^t_\top\}$. According to our execution semantics $\{\mathit{post}^-_{\mathit{enf}}(\mathcal{A}^t_\top) \cap \mathit{post}^+(\mathcal{A}^t_\top)\} = \emptyset$ and the state $s^{t+1}_\top$ must be $\{\{s^t_\top \backslash \mathit{post}^-_{\mathit{enf}}(\mathcal{A}^t_\top)\} \cup \mathit{post}^+(a^t_\top)\}$. These are enforced by the explanatory frame axioms (Schemata 3.5.8 and 3.3.10), which ensure that for each fluent $f_i \in s^{t+1}_\top$ iff $f_i \in s^t_\top$ and $f_i \notin \mathit{post}^-_{\mathit{enf}}(a^t_\top)$ or $f_i \in \mathit{post}^+(a^t_\top)$.

It remains to show (ii), that every valid parallel plan π_h for Π_h can be translated into a satisfying valuation $\mathcal{V}$ of $\mathcal{T}^{Pe}(\Pi_h)$. Following the logic of the above proof for (i), it is easy to see that that the assignment $\mathcal{V}$ that represents the fluents $\mathcal{F}_\top$ and actions $\mathcal{A}_\top$, implied by the plan π_h, satisfies the clauses in the encoding $\mathcal{T}^{Pe}(\Pi_h)$. □

3.6 Other Encodings

A number of other encodings of planning as SAT have been proposed that fall outside the scope of the previous sections. These include encodings such as those of Rintanen, et al. [57], which use a relaxed action execution semantics to allow increased action parallelism, similar to the first encoding presented later in Section 4.2.2. Their work also addresses the problem of size-blowup in planning CNFs and presented the first direct SAT encoding with size linearly bounded by the size of the problem at hand. Later Wehrel and Rintanen adapted this work from its original setting, ADL planning, to the STRIPS case, also further relaxing constraints on action parallelism and availability.

These approaches both exploit the concept of *post-serialisability* from Dimopoulos et al. [14], thereby allowing a set of conflicting actions to be prescribed at a single step provided a scheme, computed *a priori*, is available for generating a valid serial execution. In their encoding, *conflict exclusion* axioms – i.e., constraints that prevent conflicting actions being executed in parallel – are exchanged for axioms ensuring that conflicting parallel executions respect the serialisation scheme. Consequently, a lower horizon is usually required using ramp-up compared with other existing encodings. This means that fewer SAT problems are usually required to be solved. On the downside, the length of solution plans are often relatively long [59].

Significant work has also been carried out to retain learned constraints between SAT queries. Nabeshima et al. [48] developed the Lemma-Reusing Planner (LRP) that carries conflict clauses learned in refuting plan existence at $h - 1$ steps to the problem with h steps. Along similar lines, Ray and Ginsberg [53] introduce the SAT-based system CRICKET, which uses modified SAT procedures that exploit a predefined branching order guaranteeing optimality whenever the queried horizon is greater than or equal to the step optimal horizon – i.e., using ramp-down strategy, CRICKET makes a single call to a SAT procedure to produce a step-optimal solution. In practice Ray and Ginsberg [53] used a geometric ramp-up strategy – i.e., for some $\alpha > 1$, $h_1 = 1, h_i = \alpha h_{i-1}$. Overall, these developments in the optimal setting have not addressed the problem of size blow-up, which in these works remains a hindrance to scalability and efficiency.

Finally, we draw attention to the causal encodings of Kautz et al. [34] as they have provided inspiration for the causal elements of the encodings used in Chapter 5. In these encodings there are a number of steps and there are predicates to attach actions to these steps. There are a set of variables to describe the ordering of steps and constraints that ensure that orderings are transitive and anti-symmetric and that causal support for fluents and actions obeys the orderings. In addition they present a lifted version of their encoding which encodes the problem in *lifted SAT* and then reduces this to SAT. Their encodings are restricted to the serial case and while they are asymptotically small, we have found that they do allow for efficient planning in general.

Chapter 4

Compact and Efficient Split Action Representations

Split action representations (Section 4.1) are effective for step-optimal SAT-based planning with a serial execution semantics. In general, split action representations allow for more compact encodings than the corresponding flat serial encodings and for faster planning overall. As well as allowing better memory efficiency, the encodings have *factored* axioms that are better exploited by conflict-driven clause learning procedures such as PRECOSAT [7]. Recent successful SAT-based planning systems [12, 39, 53, 60, 67] use a parallel execution semantics. As well as allowing the modelling of parallelism inherent in planning domains, this usually allows planning at a shorter horizon than in the corresponding serial case. Until IPC-6 and IPC-7 (held in 2008 and 2011 respectively) the optimal propositional track of International Planning Competition accepted solution plans that were parallel step-optimal. Due to the problem of interference, the split action representations introduced in Section 3.4 are not suitable for step-optimal planning with a parallel execution semantics. The main contribution of this chapter is a general framework for split action representations and a number of encodings within this framework that, for the first time, enable split action representations to be used for parallel step-optimal planning. We call these encodings *precisely split*. They are more compact and scalable than existing encodings for parallel SAT-based planning.

In this framework, an action – i.e. an instantiation of an operator, is represented by a set of assignments to the operator's arguments. In a flat encoding a single propositional variable is created for each set of assignments that represents an action. In a simply split encoding one propositional variable is created for each assignment to each individual operator argument. Our precise encodings rely on the fact that there is a middle ground between the flat and simply split cases. We allow the definition of *argument subset schemes* (or just *subset schemes* for brevity) which define subsets of an operator's arguments, known here as *argument subsets*. An assignment to the arguments in an argument subset is known as an *assignment set*. Each assignment set is represented in the resulting CNF encoding by a single propositional variable. An action is then represented by a conjunction of such propositional variables, with each conjunct representing an assignment set for a distinct argument subset in a subset scheme. For example, in a flat encoding the operator $\texttt{Drive}(?t, ?from, ?to)$ has the subset scheme: $\{\{?t, ?from, ?to\}\}$. The action $\texttt{Drive}(T, L_1, L_2)$ is then represented by the following single element set of assignment sets $\{\{?t \leftarrow T, ?from \leftarrow L_1, ?to \leftarrow L_2\}\}$ and at step t by the following propositional variable: $\texttt{Drive}\{\{?t{\leftarrow}T, ?from{\leftarrow}L_1, ?to{\leftarrow}L_2\}\}^t$. The corresponding simply split representation of the action has the subset scheme: $\{\{?t\}, \{?from\}, \{?to\}\}$ and the following set of assignment sets: $\{\{?t{\leftarrow}T\}, \{?from{\leftarrow}L_1\}, \{?to{\leftarrow}L_2\}\}$. The action $\texttt{Drive}(T, L_1, L_2)^t$ is then represented by the following conjunction of propositional variables:

$$\texttt{Drive}\{?t{\leftarrow}T\}^t \wedge \texttt{Drive}\{?from{\leftarrow}L_1\}^t \wedge \texttt{Drive}\{?to{\leftarrow}L_2\}^t$$

In this chapter we formalise a notion of *conditions* from STRIPS planning. Conditions, put simply, are the preconditions and postconditions (add and delete effects) of actions and describe the way in which sets of actions interact with state fluents. First-order conditions are described for operators, known as *operator conditions*. These are then grounded using actions to form *ground conditions*. A ground condition is an annotated fluent, with annotations consisting of precondition "PRE", positive postcondition "ADD", or negative postcondition "DEL". For example, the $\texttt{Drive}(T, L_1, L_2)$ action has the following ground conditions:

$$\texttt{PRE}(\texttt{At}(T, L_1)), \texttt{DEL}(\texttt{At}(T, L_1)), \texttt{ADD}(\texttt{At}(T, L_2))$$

To represent the execution of one or more actions in a split encoding, the propositions representing assignment sets must be linked to the conditions of the actions. These links are then expressed in axiom schemata, such as precondition and postcondition axioms and successor state axioms. In a split encoding, this link can be formalised as a map from the conditions of an action to the assignment sets that represent the action. Here, we call such a map a *condition map*. For each operator, a condition map is defined at a first-order level for operator conditions and argument subsets. That map is then grounded using the instantiations of the operator, resulting in a map from ground conditions to assignment sets. Continuing the previous example, for the simply split encoding we have the following mappings:

1. Ground condition $\texttt{PRE}(\texttt{At}(T, L_1))$ maps to $\{\{?t{\leftarrow}T\}, \{?from{\leftarrow}L_1\}\}$;

2. Ground condition $\texttt{DEL}(\texttt{At}(T, L_1))$ maps to $\{\{?t{\leftarrow}T\}, \{?from{\leftarrow}L_1\}\}$;

3. Ground condition $\texttt{ADD}(\texttt{At}(T, L_2))$ maps to $\{\{?t{\leftarrow}T\}, \{?from{\leftarrow}L_2\}\}$.

Where $\texttt{Drive}\{?t{\leftarrow}T\}$ is the propositional variable for assignment set $\{?t{\leftarrow}T\}$ associated with the operator $\texttt{Drive}$, we have the following precondition and postcondition axioms for step t:

$$\begin{array}{lcl}
(\texttt{Drive}\{?t{\leftarrow}T\}^t \wedge \texttt{Drive}\{?from{\leftarrow}L_1\}^t) & \rightarrow & \texttt{At}(T, L_1)^t \\
(\texttt{Drive}\{?t{\leftarrow}T\}^t \wedge \texttt{Drive}\{?from{\leftarrow}L_1\}^t) & \rightarrow & \neg\texttt{At}(T, L_1)^{t+1} \\
(\texttt{Drive}\{?t{\leftarrow}T\}^t \wedge \texttt{Drive}\{?to{\leftarrow}L_2\}^t) & \rightarrow & \texttt{At}(T, L_2)^{t+1}
\end{array}$$

Using these concepts, this chapter describes subset schemes and the related condition mappings for operators that yield an encoding of the parallel step-optimal problem, with a split representation, that does not suffer from the problem of interference. The primary contribution of that work is the choice of argument subset schemes in which each argument subset is the set of arguments of one or more conditions of an operator. This admits a condition map that links each ground condition to exactly one assignment set. For example, with a precisely split representation the $\texttt{Drive}$ action from the previous examples has the following condition mappings:

1. Ground condition $\texttt{PRE}(\texttt{At}(T, L_1))$ maps to $\{\{?t{\leftarrow}T, ?from{\leftarrow}L_1\}\}$.

2. Ground condition $\texttt{DEL}(\texttt{At}(T, L_1))$ maps to $\{\{?t{\leftarrow}T, ?from{\leftarrow}L_1\}\}$.

3. Ground condition $\texttt{ADD}(\texttt{At}(T, L_2))$ maps to $\{\{?t{\leftarrow}T, ?from{\leftarrow}L_2\}\}$.

We then have the following precondition and postcondition axioms for step t:

$$\begin{array}{lcl}
\texttt{Drive}\{?t{\leftarrow}T, ?from{\leftarrow}L_1\}^t & \rightarrow & \texttt{At}(T, L_1)^t \\
\texttt{Drive}\{?t{\leftarrow}T, ?from{\leftarrow}L_1\}^t & \rightarrow & \neg\texttt{At}(T, L_1)^{t+1} \\
\texttt{Drive}\{?t{\leftarrow}T, ?to{\leftarrow}L_2\}^t & \rightarrow & \texttt{At}(T, L_2)^{t+1}
\end{array}$$

This action representation has the property that the conditions of non-conflicting sets of actions can be uniquely linked to the propositional variables representing assignment sets. This allows all valid state transitions to be precisely represented by propositions, each of which corresponds to an assignment set. More precisely, two actions could interfere if the union of their sets of assignment sets was related to a condition that neither of the actions possess.[1] For a set of non-conflicting actions $\mathcal{A}$, the set of conditions of the actions in $\mathcal{A}$ relates to exactly the set of assignment sets of the actions in $\mathcal{A}$. Consequently, interference does not occur in precisely split encodings and *parallel* step-optimal plans can be found.

In the remainder of this chapter: we present the general framework for split action representations (Section 4.1). We then explore the notion of interference in split representations (Section 4.1.3) and how this limitation can be overcome (Section 4.1.4). Next, a number of encodings are presented that translate planning problems represented in the split action framework to SAT (Section 4.2). We then present an empirical evaluation of these encodings (Section 4.3). Finally, some concluding remarks are made (Section 4.4).

4.1 A Framework for Split Action Representations

4.1.1 Argument Subsets and Assignment Sets

The following definitions apply to each operator at each step individually. We therefore omit steps from the discussion unless they are explicitly required. Notations and definitions relating to the following framework, in this section and subsequent sections, are summarised in Table 4.1.

Definition 4.1.1. We write θ to denote a set of typed variables, and write θ_o for a set of typed variables such that $\theta_o \subseteq \vec{x}$, where $o := O(\vec{x})$ and treating the list $\vec{x}$ as a set in the usual way. We call θ_o an *argument subset*, as it describes a subset of an operators arguments. □

For example, the operator $\texttt{Drive}(?t, ?from, ?to)$ has the following non-empty argument subsets: $\{?t, ?from, ?to\}$, $\{?t, ?from\}$, $\{?t, ?to\}$, $\{?from, ?to\}$, $\{?t\}$, $\{?from\}$, $\{?to\}$.

Definition 4.1.2. We define an *argument subset scheme* Θ_o as a set of argument subsets of o. □

For example, some possible subset schemes for the operator $\texttt{Drive}(?t, ?to, ?from)$ are:

$$\{\{?t, ?to, ?from\}\}, \quad \{\{?t\}, \{?to\}, \{?from\}\}, \quad \{\{?t, ?to\}, \{?t, ?from\}\}, \quad \{\{?t\}\}$$

In the following discussion we assume that all assignments are made respecting argument types and omit the operator name where they are clear from the context, or not related to the discussion.

Assignments may be made to the arguments in argument subsets and the resulting *assignment sets* may be used to represent the execution of actions.

[1]For a concrete example of interference see Section 3.4.

Symbol	Terminology	Description
θ_o	*argument subset*	A set of typed variables satisfying the constraint $\theta_o \subseteq \vec{x}$, where $o := O(\vec{x})$.
Θ_o	*argument subset scheme*	A set of elements of the type θ_o.
γ_θ	*assignment set*	A set of assignments of arguments to objects. Each assignment has domain θ.
Γ	*assignment sets*	A set of assignment sets.
$\theta(\gamma)$	*projection* of γ to θ	$\theta(\gamma) := \{?x{\leftarrow}X \mid ?x \in \theta,\ ?x{\leftarrow}X \in \gamma\}$.
γ_a	*assignments* of a	The set of assignments that produce a.
Γ_{θ_o}	*legal assignment sets* of θ_o	$\Gamma_{\theta_o} := \{\{\theta(\gamma_a)\} \mid a \in \mathcal{A}_o\}$.
Γ_{Θ_o}	*legal assignment sets* of all $\theta_o \in \Theta_o$	$\Gamma_{\Theta_o} := \bigcup_{\theta_o \in \Theta_o} \Gamma_{\theta_o}$.
$\Gamma \models a$	Γ *entails the execution* of a	$\Gamma \models a$ iff $\gamma_a \subseteq \bigcup_{\gamma \in \Gamma} \gamma$.
$\Gamma_{\Theta_o,a}$	*assignment sets* that entail *the execution of* a	$\Gamma_{\Theta_o,a}$ has one assignment set from each Γ_{θ_o}, for $\theta_o \in \Theta_o$, and $\Gamma_{\Theta_o,a} \models a$.
$\vec{x}'$	*sub argument list*	An argument list made up of elements from $\vec{x}$, for an operator $O(\vec{x})$.
$\mathcal{C}_o{=}\tau(\lambda(\vec{x}'))$	*operator condition* of o	An operator condition for o. $\tau \in \{\texttt{PRE}, \texttt{ADD}, \texttt{DEL}\}$ and predicate $\lambda \in \Lambda$.
$\mathbb{C}_o$	*operator conditions* of o	The set of operator conditions for o.
c_o	*ground condition* of o	c_o is a grounding of some $\mathcal{C}_o \in \mathbb{C}_o$.
$C_{\mathcal{C}_o}$	*ground conditions* of $\mathcal{C}_o$	$c_o \in C_{\mathcal{C}_o}$, is $\mathcal{C}_o$ grounded with γ_a, $a \in \mathcal{A}_o$.
C_o	*ground conditions* of o	$C_o := \{C_{\mathcal{C}_o} \mid \mathcal{C}_o \in \mathbb{C}_o\}$.
γ_c	*assignments* of c	The set of assignments that produce c
C_a	*ground conditions* of a	$C_a := \{c_o \mid \gamma_{c_o} \subseteq \gamma_a, c_o \in C_{\mathcal{C}_o}, \mathcal{C}_o \in \mathbb{C}_o\}$.
$\mathcal{R}_o$	*condition map* for o	A map from each $\mathcal{C}_o \in \mathbb{C}_o$ to a subset of Θ_o.
$\mathcal{R}_o(c_o)$	*ground condition map* of o	$\mathcal{R}_o(c_o) := \{\gamma_{\theta_o} \mid \vec{x}'(\gamma_{\theta_o}) \subseteq \gamma_{c_o}, \gamma_{\theta_o} \in \Gamma_{\theta_o}, \theta_o \in \mathcal{R}_o(\mathcal{C}_o)\}$
$\Gamma \models c_o$	Γ *entails* c_o	$\Gamma \models c_o$ iff $R_o(c_o) \subseteq \Gamma$.

Table 4.1: A summary of terminology defined in Section 4.1.

Definition 4.1.3. We write γ to denote an *assignment set*, in other words a set of assignments of arguments to objects. We write γ_θ if the domain of the assignments is the argument subset θ and each argument in θ is assigned to a single object. $\square$

In our ongoing example, the argument subset $\theta := \{?t, ?from\}$ can have the assignment sets: $\gamma_{\theta,1} := \{?t{\leftarrow}T, ?from{\leftarrow}L_1\}$, $\gamma_{\theta,2} := \{?t{\leftarrow}T, ?from{\leftarrow}L_2\}$, $\gamma_{\theta,3} := \{?t{\leftarrow}T, ?from{\leftarrow}L_3\}$. In general, assignment sets may assign a single argument to more than one object. For example, $\{?t{\leftarrow}T_1, ?t{\leftarrow}T_2, ?from{\leftarrow}L_1\}$ is a valid assignment set. In the following, the descriptions of assignment sets omit the argument subset when this will not cause confusion.

Definition 4.1.4. We write $\theta(\gamma)$ to denote a *projection*[2] of assignment set γ to the variables in argument subset θ. Formally, $\theta(\gamma) := \{?x{\leftarrow}X \mid ?x \in \theta,\ ?x{\leftarrow}X \in \gamma\}$ $\square$

For example, for a partial grounding $\gamma := \{?t \leftarrow T_1, ?to \leftarrow L_1, ?from \leftarrow L_2\}$ and a set of arguments $\theta := \{?t, ?to\}$, we have $\theta(\gamma) := \{?t{\leftarrow}T_1, ?to{\leftarrow}L_1\}$.

Definition 4.1.5. For an action a that instantiates operator $O(\vec{x})$, let γ_a be the set of assignments to $\vec{x}$ that produce a. $\square$

For example, if $a := \texttt{Drive}(T, L_1, L_2)$, which instantiates the operator $\texttt{Drive}(?t, ?from, ?to)$, then $\gamma_a := \{?t{\leftarrow}T, ?from{\leftarrow}L_1, ?to{\leftarrow}L_2\}$.

Definition 4.1.6. Given an argument subset θ_o, we notate Γ_{θ_o} the set of assignment sets of θ_o that are legal according to $\mathcal{A}_o$. Formally, we have: $\Gamma_{\theta_o} := \{\{\theta_o(\gamma_a)\} | a \in \mathcal{A}_o\}$. $\square$

Definition 4.1.7. Given an subset scheme Θ_o, let Γ_{Θ_o} be the set of valid assignment sets of all argument subsets in Θ_o for all actions in the set $\mathcal{A}_o$. Formally, $\Gamma_{\Theta_o} := \bigcup_{\theta_o \in \Theta_o} \Gamma_{\theta_o}$. $\square$

To illustrate these concepts, the column labelled θ_o in Table 4.2 shows the set of possible non-empty argument subsets for a SIMPLE-LOGISTICS problem with one truck T and three locations L_1, L_2, and L_3. The column labelled Γ_{θ_o} shows the assignment sets of each argument subset θ_o.

We are now in a position to describe how assignment sets may be used to represent the execution of an action.

Definition 4.1.8. Let $\Gamma \models a$ mean that a set of assignment sets Γ entails the execution of an action a. In other words, $\Gamma \models a$ iff $\gamma_a \subseteq \bigcup_{\gamma \in \Gamma} \gamma$. $\square$

[2]In the sense of relational algebra.

θ_o	Γ_{θ_o}
$\{?t, ?from, ?to\}$	$\{\{?t{\leftarrow}T, ?from{\leftarrow}L_1, ?to{\leftarrow}L_2\}, \{?t{\leftarrow}T, ?from{\leftarrow}L_1, ?to{\leftarrow}L_3\},$
	$\{?t{\leftarrow}T, ?from{\leftarrow}L_2, ?to{\leftarrow}L_1\}, \{?t{\leftarrow}T, ?from{\leftarrow}L_2, ?to{\leftarrow}L_3\},$
	$\{?t{\leftarrow}T, ?from{\leftarrow}L_3, ?to{\leftarrow}L_1\}, \{?t{\leftarrow}T, ?from{\leftarrow}L_3, ?to{\leftarrow}L_2\}\}$
$\{?t, ?from\}$	$\{\{?t{\leftarrow}T, ?from{\leftarrow}L_1\}, \{?t{\leftarrow}T, ?from{\leftarrow}L_2\}, \{?t{\leftarrow}T, ?from{\leftarrow}L_3\}\}$
$\{?t, ?to\}$	$\{\{?t{\leftarrow}T, ?to{\leftarrow}L_1\}, \{?t{\leftarrow}T, ?to{\leftarrow}L_2\}, \{?t{\leftarrow}T, ?to{\leftarrow}L_3\}\}$
$\{?from, ?to\}$	$\{\{?from{\leftarrow}L_1, ?to{\leftarrow}L_2\}, \{?from{\leftarrow}L_1, ?to{\leftarrow}L_3\},$
	$\{?from{\leftarrow}L_2, ?to{\leftarrow}L_1\}, \{?from{\leftarrow}L_2, ?to{\leftarrow}L_3\},$
	$\{?from{\leftarrow}L_3, ?to{\leftarrow}L_1\}, \{?from{\leftarrow}L_3, ?to{\leftarrow}L_2\}\}$
$\{?t\}$	$\{\{?t{\leftarrow}T\}\}$
$\{?from\}$	$\{\{?from{\leftarrow}L_1\}, \{?from{\leftarrow}L_2\}, \{?from{\leftarrow}L_3\}\}$
$\{?to\}$	$\{\{?to{\leftarrow}L_1\}, \{?to{\leftarrow}L_2\}, \{?to{\leftarrow}L_3\}\}$

Table 4.2: The column labelled θ_o shows the set of non-empty argument subsets for the operator $o := \texttt{Drive}(?t, ?from, ?to)$ from a SIMPLE-LOGISTICS problem. The column labelled Γ_{θ_o} shows the assignment sets for each argument subset in a problem with one truck T and three locations L_1, L_2, and L_3.

For example, the execution of the action $\texttt{Drive}(T, L_1, L_2)$ is entailed by the following sets of assignment sets, among others:

$$\{\{?t{\leftarrow}T, ?from{\leftarrow}L_1, ?to{\leftarrow}L_2\}\}, \{\{?t{\leftarrow}T, ?from{\leftarrow}L_1\}, \{?t{\leftarrow}T, ?to{\leftarrow}L_2\}\},$$
$$\{\{?t{\leftarrow}T\}, \{?from{\leftarrow}L_1\}, \{?to{\leftarrow}L_2\}\}, \{\{?t{\leftarrow}T\}, \{?from{\leftarrow}L_1\}, \{?to{\leftarrow}L_2\}, \{?to{\leftarrow}L_1\}\}$$

As described later in Section 4.2, it is possible to generate a SAT encoding of planning based on an argument subset scheme by encoding assignment sets as propositional variables. If, for all operators $o_i \in \mathcal{O}$, where $o_i := O_i(\vec{x_i})$, we use the argument subset scheme $\Theta_{o_i} := \{\{\vec{x_i}\}\}$, then assignment sets will correspond directly with actions as in the flat encoding described in Section 4.2.1. Alternatively, if we use the argument subset scheme $\Theta_{o_i} := \{\{?x\} | ?x \in \vec{x_i}\}$, then assignment sets will correspond to individual argument assignments as in simply split encodings such as Kautz and Selman [35], Ernst et al. [18], and Robinson et al. [59]. A number of other argument subset schemes are also possible, including the main technical contribution of this chapter (first published in Robinson et al. [60]).

For an argument subset scheme Θ_o, for an operator o, to be used as the basis of a SAT encoding of planning, we require that the assignment sets of argument subsets in Θ_o can represent valid executions of actions in $\mathcal{A}_o$, given our chosen action execution semantics. First, let us consider serial action execution semantics.[3] In this case, we require that a set of assignment sets from an operator can be used to represent the execution of a single action. More formally, for an operator o, we can use a subset scheme Θ_o to represent the execution of individual actions in $\mathcal{A}_o$ only, if for every action $a_i \in \mathcal{A}_o$ there exists a set of assignment sets $\Gamma_i \subseteq \Gamma_{\Theta_o}$ such that $\Gamma_i \models a_i$, and there is no $a_j \in \mathcal{A}_o, a_j \neq a_i$ such that $\Gamma_i \models a_j$.

Proposition 4.1.1. *A subset scheme Θ_o, for an operator $o := O(\vec{x})$, where every operator argument $?x_i \in \vec{x}$ is included in at least one argument subset $\theta_o \in \Theta_o$, has a set of assignment sets Γ_{θ_o} which contain the assignments $\bigcup_{\gamma \in \Gamma_{\theta_o}} = \gamma_a$ of any individual action $a \in \mathcal{A}_o$.*

Proof. We will prove that it is possible to construct a set of assignment sets $\Gamma \subseteq \Gamma_{\Theta_o}$ of the argument subsets in Θ_o, where Γ contains a single assignment set from each $\theta_o \in \Theta_o$, such that $\bigcup_{\gamma \in \Gamma} = \gamma_{a_i}$, for any $a_i \in \mathcal{A}_o$. Since any action $a_j \in \mathcal{A}_o, a_j \neq a_i$, must include some argument $?x \notin \gamma_{a_i}$, Γ will entail a_i and no other action a_j. Indeed, there is exactly one assignment set $\gamma_{\theta_o} \in \Gamma_{\theta_o}$ of every argument subset $\theta_o \in \Theta_o$ such that $\gamma_{\theta_o} \subseteq \gamma_{a_i}$. If $\Gamma := \bigcup_{\gamma_{\theta_o},\, \gamma_{\theta_o} \subseteq \gamma_{a_i},\, \theta_o \in \Theta_o} \gamma_{\theta_o}$, we have $\bigcup_{\gamma \in \Gamma} \subseteq \gamma_{a_i}$. Because every operator argument is included in at least one argument subset, we have $\bigcup_{\gamma \in \Gamma} = \gamma_{a_i}$. $\square$

For the rest of this chapter we restrict our attention to subset schemes Θ_o, for an operator $o := O(\vec{x})$, where every operator argument $?x_i \in \vec{x}$ is included in at least one argument subset $\theta_o \in \Theta_o$. In other words, in the context of Proposition 4.1.1, we restrict our attention to subset schemes where it is possible to represent the execution of any individual instance of an operator.

Definition 4.1.9. For an action $a \in \mathcal{A}_o$, and a subset scheme Θ_o, let $\Gamma_{\Theta_o, a}$ be a set of argument sets such that:

- $\forall \theta_o \in \Theta_o$, $\Gamma_{\Theta_o, a}$ contains a single partial grounding from Γ_{θ_o}; and
- $\Gamma_{\Theta_o, a} \models a$. $\square$

From Definition 4.1.9 and Proposition 4.1.1 we have that there is exactly one set $\Gamma_{\Theta_o, a}$ for each action $a \in \mathcal{A}_o$ and that for each argument subset $\theta_o \in \Theta_o$ there is exactly one grounding $\gamma_{\theta_o} \in \Gamma_{\theta_o}$, such that $\bigcup_{\gamma \in \Gamma_{\theta_o}} \subseteq \gamma_a$. In the sequel, we may omit the subset scheme Θ_o from $\Gamma_{\Theta_o, a}$ if it is clear from the context.

[3]The following also applies to an action execution semantics where at most one action can be executed in parallel that instantiates each individual operator.

4.1.2 Preconditions and Postconditions

Actions interact with state fluents through their preconditions and postconditions. In our framework each precondition or postcondition is represented by relating a set of assignment sets to a state fluent. To encode this relationship we introduce the notion of a *condition*. Informally, a condition is a precondition, or a postcondition of an operator, referred to as an *operator condition*. Likewise, a *ground condition* is a precondition or a postcondition of an action.

Definition 4.1.10. For an operator $o := O(\vec{x})$, we write $\vec{x}'$ for an argument list made up of elements from $\vec{x}$. □

Definition 4.1.11. The set of operator conditions for an operator o are notated $\mathbb{C}_o$ and elements in that set $\mathcal{C}_o \in \mathbb{C}_o$, where $\mathcal{C}_o := \tau(\lambda(\vec{x}'))$. Here for a predicate $\lambda \in \Lambda$, τ denotes a condition type, either: (1) a precondition, written $\tau := \texttt{PRE}$, (2) a positive postcondition $\tau := \texttt{ADD}$, or (3) a negative postcondition $\tau := \texttt{DEL}$. □

For example, the operator $\texttt{Move}(?x_1, ?x_2, ?x_3)$, from the BLOCKS-DIRECT domain introduced in Section 2.1 and Appendix A, has the set of conditions $\mathbb{C}_o :=$

$$\{\texttt{PRE}(\texttt{On}(?x_1, ?x_2)), \texttt{PRE}(\texttt{Clear}(?x_1)), \texttt{PRE}(\texttt{Clear}(?x_3)), \texttt{DEL}(\texttt{On}(?x_1, ?x_2)),$$
$$\texttt{DEL}(\texttt{Clear}(?x_3)), \texttt{ADD}(\texttt{On}(?x_1, ?x_3)), \texttt{ADD}(\texttt{Clear}(?x_2))\}$$

We may omit the operator o if it is clear from the context.

Definition 4.1.12. A ground condition c_o is a grounding of an operator condition $\mathcal{C}_o \in \mathbb{C}_o$ with the assignments γ_a of an action $a \in \mathcal{A}_o$. □

Definition 4.1.13. Let $C_{\mathcal{C}_o}$ be the set of ground conditions which instantiate operator condition $\mathcal{C}_o$. Each $c_o \in C_{\mathcal{C}_o}$, is generated by grounding $\mathcal{C}_o$ with γ_a, for an action $a \in \mathcal{A}_o$. □

For example, the operator $o := \texttt{Move}(?x_1, ?x_2, ?x_3)$, has a condition $\mathcal{C}_o := \texttt{PRE}(\texttt{On}(?x_1, ?x_2))$. If $\mathcal{A}_o := \{\texttt{Move}(A, B, C), \texttt{Move}(A, B, D), \texttt{Move}(B, C, D)\}$, then

$$C_{\mathcal{C}_o} := \{\texttt{PRE}(\texttt{On}(A, B)), \texttt{PRE}(\texttt{On}(B, C))\}$$

Definition 4.1.14. Let C_o be the set of all ground conditions for an operator o. Formally, $C_o := \{C_{\mathcal{C}_o} | \mathcal{C}_o \in \mathbb{C}_o\}$. □

Definition 4.1.15. A ground condition c has the set of assignments γ_c. □

For example, the ground condition $c := \texttt{PRE}(\texttt{On}(A, B))$, which instantiates operator condition $\texttt{PRE}(\texttt{On}(?x_1, ?x_2))$, has the set of assignments $\gamma_c := \{?x_1 \leftarrow A, ?x_2 \leftarrow B\}$.

Definition 4.1.16. An action $a \in \mathcal{A}_o$, for an operator o, has the set of ground conditions $C_a := \{c_o \,|\, \gamma_{c_o} \subseteq \gamma_a, c_o \in C_{\mathcal{C}_o}, \mathcal{C}_o \in \mathbb{C}_o\}$. □

For example, the action $\texttt{Move}(A, B, C)$ from the BLOCKS-DIRECT domain has the set of ground conditions $C_a :=$

$$\{\texttt{PRE}(\texttt{On}(A, B)), \texttt{PRE}(\texttt{Clear}(A)), \texttt{PRE}(\texttt{Clear}(C)), \texttt{DEL}(\texttt{On}(A, B)),$$
$$\texttt{DEL}(\texttt{Clear}(C)), \texttt{ADD}(\texttt{On}(A, C)), \texttt{ADD}(\texttt{Clear}(B))\}$$

To represent the preconditions and postconditions of an operator o, we need a map from the operator conditions in $\mathbb{C}_o$ to subsets in a subset scheme Θ_o. That an operator precondition or postcondition maps to a set of argument subsets means that when a grounding of those subsets holds, a related grounding of the condition will hold.

Definition 4.1.17. For an operator o with a subset scheme Θ_o, let $\mathcal{R}_o$ be a *condition map*, that is a map from each operator condition $\mathcal{C}_o \in \mathbb{C}_o$ to a subset of Θ_o. □

To illustrate this concept, consider the following examples. First, a simply split encoding of operator $o := \texttt{Drive}(?t, ?from, ?to)$ has the set $\Theta_o := \{\{?t\}, \{?from\}, \{?to\}\}$. The condition map $\mathcal{R}_o$ that corresponds to the *factored* simply split encoding of Ernst et al. [18] is as follows:

$$\begin{aligned} \mathcal{R}_o(\texttt{PRE}(\texttt{At}(?t, ?from))) &:= \{\{?t\}, \{?from\}\} \\ \mathcal{R}_o(\texttt{DEL}(\texttt{At}(?t, ?from))) &:= \{\{?t\}, \{?from\}\} \\ \mathcal{R}_o(\texttt{ADD}(\texttt{At}(?t, ?to))) &:= \{\{?t\}, \{?to\}\} \end{aligned}$$

For the corresponding *non-factored* simply split encoding we have that each operator condition relates to every operator argument:

$$\begin{aligned} \mathcal{R}_o(\texttt{PRE}(\texttt{At}(?t, ?from))) &:= \{\{?t\}, \{?from\}, \{?to\}\} \\ \mathcal{R}_o(\texttt{DEL}(\texttt{At}(?t, ?from))) &:= \{\{?t\}, \{?from\}, \{?to\}\} \\ \mathcal{R}_o(\texttt{ADD}(\texttt{At}(?t, ?to))) &:= \{\{?t\}, \{?from\}, \{?to\}\} \end{aligned}$$

A flat encoding of the same operator has the argument subsets $\Theta_o := \{\{?t, ?from, ?to\}\}$. The corresponding condition map $\mathcal{R}_o$ is as follows:

$$\begin{aligned} \mathcal{R}_o(\texttt{PRE}(\texttt{At}(?t, ?from))) &:= \{\{?t, ?to, ?from\}\} \\ \mathcal{R}_o(\texttt{DEL}(\texttt{At}(?t, ?from))) &:= \{\{?t, ?to, ?from\}\} \\ \mathcal{R}_o(\texttt{ADD}(\texttt{At}(?t, ?to))) &:= \{\{?t, ?to, ?from\}\} \end{aligned}$$

Next, we need to translate the condition map for operator conditions to the ground case. Given a condition map $\mathcal{R}_o$ and a subset scheme Θ_o for operator $o := O(\vec{x})$, we need to define a condition map from each ground condition to a set of assignment sets.

Definition 4.1.18. Given a subset scheme Θ_o and condition map $\mathcal{R}_o$ for an operator o, we have that for each ground condition $c_o \in C_{\mathcal{C}_o}$, $\mathcal{C}_o := \tau(\lambda(\vec{x}')) \in \mathbb{C}_o$, R_o is a map from c_o to the set of assignment sets that entail c_o. Formally,

$$\mathcal{R}_o(c_o) := \{\gamma_{\theta_o} | \vec{x}'(\gamma_{\theta_o}) \subseteq \gamma_{c_o}, \gamma_{\theta_o} \in \Gamma_{\theta_o}, \theta_o \in \mathcal{R}_o(\mathcal{C}_o)\} \qquad \square$$

Continuing the previous examples, where $o := \texttt{Drive}(?t, ?from, ?to)$, in the factored case of the simply split encoding, we have the following maps for ground conditions of the action $\texttt{Drive}(T, L_1, L_2)$, regardless of the other actions in $\mathcal{A}_o$:

$$\begin{aligned}
\mathcal{R}_o(\texttt{PRE}(\texttt{At}(T, L_1))) &:= \{\{?t \leftarrow T\}, \{?from \leftarrow L_1\}\} \\
\mathcal{R}_o(\texttt{DEL}(\texttt{At}(T, L_1))) &:= \{\{?t \leftarrow T\}, \{?from \leftarrow L_1\}\} \\
\mathcal{R}_o(\texttt{ADD}(\texttt{At}(T, L_2))) &:= \{\{?t \leftarrow T\}, \{?to \leftarrow L_2\}\}
\end{aligned}$$

For the flat encoding, if we have $\mathcal{A}_o :=$

$$\begin{aligned}
\{&\texttt{Drive}(T, L_1, L_2), & \texttt{Drive}(T, L_1, L_2), \\
&\texttt{Drive}(T, L_2, L_1), & \texttt{Drive}(T, L_2, L_3), \\
&\texttt{Drive}(T, L_3, L_1), & \texttt{Drive}(T, L_3, L_1)\}
\end{aligned}$$

then we have the following maps for ground conditions:

$$\mathcal{R}_o(\texttt{PRE}(\texttt{At}(T, L_1))) := \mathcal{R}_o(\texttt{DEL}(\texttt{At}(T, L_1))) := \mathcal{R}_o(\texttt{ADD}(\texttt{At}(T, L_1)) :=$$
$$\{\{?t \leftarrow T, ?from \leftarrow L_1, ?to \leftarrow L_2\}, \{?t \leftarrow T, ?from \leftarrow L_1, ?to \leftarrow L_3\}\}$$

$$\mathcal{R}_o(\texttt{PRE}(\texttt{At}(T, L_2))) := \mathcal{R}_o(\texttt{DEL}(\texttt{At}(T, L_2))) := \mathcal{R}_o(\texttt{ADD}(\texttt{At}(T, L_2))) :=$$
$$\{\{?t \leftarrow T, ?from \leftarrow L_2, ?to \leftarrow L_1\}, \{?t \leftarrow T, ?from \leftarrow L_2, ?to \leftarrow L_3\}\}$$

$$\mathcal{R}_o(\texttt{PRE}(\texttt{At}(T, L_3))) := \mathcal{R}_o(\texttt{DEL}(\texttt{At}(T, L_3))) := \mathcal{R}_o(\texttt{ADD}(\texttt{At}(T, L_3))) :=$$
$$\{\{?t \leftarrow T, ?from \leftarrow L_3, ?to \leftarrow L_1\}, \{?t \leftarrow T, ?from \leftarrow L_3, ?to \leftarrow L_2\}\}$$

Definition 4.1.19. For argument subset scheme Θ_o for operator o, a set of assignment sets $\Gamma \subseteq \Gamma_o$ entails a ground condition c_o, written $\Gamma \models c_o$, iff $R_o(c_o) \subseteq \Gamma$. $\square$

For example, for the set of condition maps from the flat encoding in the previous set of examples, the set of assignment sets $\{\{?t \leftarrow T, ?from \leftarrow L_1, ?to \leftarrow L_2\}\}$ entails the following conditions: $\texttt{PRE}(\texttt{At}(T, L_1))$, $\texttt{DEL}(\texttt{At}(T, L_1))$, and $\texttt{ADD}(\texttt{At}(T, L_2))$.

We can now characterise when an argument subset scheme and a condition map for an operator can be used to correctly encode serial action execution.

Proposition 4.1.2. *For a planning problem Π, a set of argument subsets Θ_o, and a condition map $\mathcal{R}_o$ can correctly encode serial action execution for an operator $o \in \mathcal{O}$, if for all actions $a \in \mathcal{A}_o$, $\Gamma_a \models c$ iff $c \in C_a$.*

Within the developed framework, the previously described flat and simply split encoding schemes correctly model serial action execution. First, in the flat encoding, for an operator $o := O(\vec{x})$, we have that $\Theta_o := \{\vec{x}\}$. Therefore each assignment set entails the execution of a single action $a \in \mathcal{A}_o$. For ground condition c of o, we have that $R_o(c)$ equals the set of assignment sets, and therefore actions, with assignments that are a superset of those which entail c. We therefore have that $\Gamma_a \models c$ iff $c \in C_a$. In the simply split encoding, for an operator $o := O(\vec{x})$, $\Theta_o := \{\{x\}|x \in \vec{x}\}$. An operator condition $\mathcal{C} := \tau(\lambda(\vec{x}'))$ is related to those argument subsets that are a subset of $\vec{x}'$.

4.1.3 Parallel Action Execution and Interference

The previous section demonstrates that encoding schemes based on splitting actions into argument subsets can be used to encode serial planning. However, in the computationally easier setting of parallel planning, *interference* prevents the use of some argument subset schemes and condition relations, in particular the encoding scheme corresponding to the simply split representation of Ernst et al. [18].

Definition 4.1.20. Let $int : \mathbb{N}_0 \times \mathcal{A} \times \mathcal{A}$ be a Boolean relation that defines when actions *interfere* at a step. For a step t and operator o and set of actions $\mathcal{A}_o$, two distinct actions $\{a_1, a_2\} \subseteq \mathcal{A}_o^t \times \mathcal{A}_o^t$ interfere, that is $(t, a_1, a_2) \in int$, iff the set of assignment sets corresponding to their parallel execution $\Gamma_{a_1} \cup \Gamma_{a_2}$ entails a ground condition $c_o \in C_{a_3}$ of a third action $a_3 \in \mathcal{A}_o^t$, and $c_o \notin C_{a_1}$ and $c_o \notin C_{a_2}$. □

For example, with an encoding scheme corresponding to factored simple splitting, for the operator `Drive`$(?t, ?from, ?to)$, the parallel execution of the pair of actions `Drive`(T_1, L_1, L_2) and `Drive`(T_2, L_3, L_4) is entailed by the set of assignment sets:

$$\{\{?t \leftarrow T_1\}, \{?t \leftarrow T_2\}, \{?from \leftarrow L_1\}, \{?from \leftarrow L_3\}, \{?to \leftarrow L_2\}, \{?to \leftarrow L_4\}\}$$

This set entails the ground conditions:

$$\begin{aligned}&\texttt{PRE}(\texttt{At}(T_1, L_1)),\ \ \texttt{ADD}(\texttt{At}(T_1, L_2)),\ \ \texttt{DEL}(\texttt{At}(T_1, L_1)),\\ &\texttt{PRE}(\texttt{At}(T_1, L_3)),\ \ \texttt{ADD}(\texttt{At}(T_1, L_4)),\ \ \texttt{DEL}(\texttt{At}(T_1, L_3)),\\ &\texttt{PRE}(\texttt{At}(T_2, L_1)),\ \ \texttt{ADD}(\texttt{At}(T_2, L_1)),\ \ \texttt{DEL}(\texttt{At}(T_2, L_1)),\\ &\texttt{PRE}(\texttt{At}(T_2, L_3)),\ \ \texttt{ADD}(\texttt{At}(T_2, L_4)),\ \ \texttt{DEL}(\texttt{At}(T_2, L_3))\end{aligned}$$

The following of which do not belong to the conditions of either of the actions $\texttt{Drive}(T_1, L_1, L_2)$ or $\texttt{Drive}(T_2, L_3, L_4)$:

$$\begin{aligned}&\texttt{PRE}(\texttt{At}(T_1, L_3)),\ \ \texttt{ADD}(\texttt{At}(T_1, L_2)),\ \ \texttt{DEL}(\texttt{At}(T_1, L_1)),\\ &\texttt{PRE}(\texttt{At}(T_2, L_1)),\ \ \texttt{ADD}(\texttt{At}(T_2, L_4)),\ \ \texttt{DEL}(\texttt{At}(T_2, L_3))\end{aligned}$$

4.1.4 Splitting Without Interference

We have seen that a split action representation can suffer from the problem of interference, rendering it incapable of representing full parallel action execution. Parallel action execution is desired for at least two reasons. First, to allow planning at a lower horizon than might otherwise be required in the serial case, and therefore increasing the efficiency of planning. Second, to model the parallelism inherent in planning domains. We present two novel schemes for encoding action execution using split representations that avoid the problem of interference. The first, published in Robinson, et al. [59], uses a simply split action representation and allows parallel action execution, except where this would cause interference. The second, published in Robinson et al. [60], is a novel splitting scheme that avoids the problem of interference altogether and therefore permits step-optimal parallel planning.

For an operator o, let Θ_o be an argument subset encoding scheme, and $\mathcal{R}_o$ a condition relation, that together define a factored simply split encoding – e.g. Ernst et al. [18]. Recall that μ_{enf} defines those pairs of actions for which we are required to enforce conflict mutex and int defines those pairs of actions which interfere.

Definition 4.1.21. Let $\mu_{sp} : \mathbb{N}_0 \times \mathcal{A} \times \mathcal{A}$ be a Boolean function such that $\mu_{sp} := \mu_{enf} \cup int$. $\square$

Now, the split encoding of planning into SAT (described in Section 4.2) can use μ_{sp} in place of μ_{enf} to specify which actions should be mutually exclusive. This defines a novel action execution semantics that admits a level of parallelism somewhere between the serial and the full parallel case. Along the same lines as the relaxed encodings of Rintanen et al. [57] and Wehrle and Rintanen [73], an encoding based on this action execution semantics generally does not permit planning in the parallel step-optimal case. Another major drawback of this approach is that, in practice, the increased planning horizon required due to the reduced action parallelism can make planning harder than it would be in the standard parallel case.

We now present a split action representation Θ^*_o and $\mathcal{R}^*_o$, for an operator $o := O(\vec{x})$, that can be used for step-optimal parallel planning. As shown later (in Section 4.3) SAT-based planning approaches based on this action representation are competitive with state-of-the-art SAT-based approaches for step-optimal planning. We refer to these interference-free split representations as *precisely split*. These encoding schemes work by constructing argument subsets, and therefore assignment sets, that correspond with operator preconditions and postconditions. We begin with the following preliminary definition.

Definition 4.1.22. Let $\alpha(\vec{x})$ be any permutation of an argument list $\vec{x}$. □

We now define the subset scheme Θ^*_o that we use for precise splitting, for operator o.

Definition 4.1.23. Let Θ^*_o for an operator o be an argument subset scheme for a precisely split encoding. For every set of conditions $\{\tau_1(\lambda_1(\vec{x}')), ..., \tau_k(\lambda_k(\alpha(\vec{x}')))\} \subseteq \mathbb{C}_o$, there is exactly one argument subset $\vec{x}' \in \Theta^*_o$. □

To illustrate this we have a few examples. First, the operator $\texttt{Move}(?x_1, ?x_2, ?x_3)$ from the BLOCKS-DIRECT domain has the following argument subsets, shown with their associated conditions:

$$\begin{array}{lcl}
\{?x_1\} & : & \texttt{PRE}(\texttt{Clear}(?x_1)) \\
\{?x_2\} & : & \texttt{ADD}(\texttt{Clear}(?x_2)) \\
\{?x_3\} & : & \texttt{PRE}(\texttt{Clear}(?x_3)), \texttt{DEL}(\texttt{Clear}(?x_3)) \\
\{?x_1, ?x_2\} & : & \texttt{PRE}(\texttt{On}(?x_1, ?x_2)), \texttt{DEL}(\texttt{On}(?x_1, ?x_2)) \\
\{?x_1, ?x_3\} & : & \texttt{ADD}(\texttt{On}(?x_1, ?x_3))
\end{array}$$

Similarly for the operator $\texttt{Drive}(?t, ?from, ?to)$:

$$\begin{array}{lcl}
\{?t, ?from\} & : & \texttt{PRE}(\texttt{At}(?t, ?from)), \texttt{DEL}(\texttt{At}(?t, ?from)) \\
\{?t, ?to\} & : & \texttt{ADD}(\texttt{At}(?t, ?to))
\end{array}$$

Now, given the described argument subset scheme Θ^*_o, for a precisely split encoding, we need a map $\mathcal{R}^*_o$ from each operator condition $\mathcal{C}_o \in \mathbb{C}_o$ to a subset of Θ^*_o.

Definition 4.1.24. Let R^*_o be a precisely split encoding for operator o. For each operator condition $\mathcal{C}_o := \tau(\lambda(\vec{x}')) \in \mathbb{C}_o$, $\mathcal{R}^*_o(\mathcal{C}_o) := \{\vec{x}'\}$. Here $\vec{x}'$ is always exactly a single element from Θ^*_o. □

Continuing the previous examples, for $o := \texttt{Move}(?x_1, ?x_2, ?x_3)$, with the set:

$$\Theta_o^* := \{\{?x_1\}, \{?x_2\}, \{?x_3\}, \{?x_1, ?x_2\}, \{?x_1, ?x_3\}\}$$

the relation $\mathcal{R}_o^*$ is as follows:

$$\begin{aligned}
\mathcal{R}_o^*(\texttt{PRE}(\texttt{Clear}(?x_1))) &:= \{\{?x_1\}\}\\
\mathcal{R}_o^*(\texttt{ADD}(\texttt{Clear}(?x_2))) &:= \{\{?x_2\}\}\\
\mathcal{R}_o^*(\texttt{PRE}(\texttt{Clear}(?x_3))) &:= \{\{?x_3\}\}\\
\mathcal{R}_o^*(\texttt{DEL}(\texttt{Clear}(?x_3))) &:= \{\{?x_3\}\}\\
\mathcal{R}_o^*(\texttt{PRE}(\texttt{On}(?x_1, ?x_2))) &:= \{\{?x_1, ?x_2\}\}\\
\mathcal{R}_o^*(\texttt{DEL}(\texttt{On}(?x_1, ?x_2))) &:= \{\{?x_1, ?x_2\}\}\\
\mathcal{R}_o^*(\texttt{ADD}(\texttt{On}(?x_1, ?x_3))) &:= \{\{?x_1, ?x_3\}\}
\end{aligned}$$

Similarly for the the operator $o := \texttt{Drive}(?t, ?from, ?to)$, with the argument subsets $\Theta_o^* := \{\{?t, ?from\}, \{?t, ?to\}\}$, the relation $\mathcal{R}_o^*$ is as follows:

$$\begin{aligned}
\mathcal{R}_o^*(\texttt{PRE}(\texttt{At}(?t, ?from))) &:= \{\{?t, ?from\}\}\\
\mathcal{R}_o^*(\texttt{DEL}(\texttt{At}(?t, ?from))) &:= \{\{?t, ?from\}\}\\
\mathcal{R}_o^*(\texttt{ADD}(\texttt{At}(?t, ?to))) &:= \{\{?t, ?to\}\}
\end{aligned}$$

Assume for the rest of this section that Θ_o^* refers to the argument subset scheme as just defined for a precise encoding and $\mathcal{R}_o^*$ refers to the given condition relation. In an encoding based on Θ_o^* and $\mathcal{R}_o^*$ for an action $a := \texttt{Move}(A, B, C)$, we have that:

$$\gamma_a := \{\{?x_1{\leftarrow}A\}, \{?x_2{\leftarrow}B\}, \{?x_3{\leftarrow}C\}, \{?x_1{\leftarrow}A, ?x_2{\leftarrow}B\}, \{?x_1{\leftarrow}A, ?x_3{\leftarrow}C\}\}$$

Given the ground conditions $C_a :=$

$$\{\texttt{PRE}(\texttt{On}(A, B)), \texttt{PRE}(\texttt{Clear}(A)), \texttt{PRE}(\texttt{Clear}(C)), \texttt{DEL}(\texttt{On}(A, B)),\\ \texttt{DEL}(\texttt{Clear}(C)), \texttt{ADD}(\texttt{On}(A, C)), \texttt{ADD}(\texttt{Clear}(B))\}$$

then, regardless of the other actions in $\mathcal{A}_o$, we have the following relations:

$$\begin{aligned}
\mathcal{R}_o^*(\texttt{PRE}(\texttt{On}(A, B))) &:= \{\{?x_1{\leftarrow}A, ?x_2{\leftarrow}B\}\}\\
\mathcal{R}_o^*(\texttt{PRE}(\texttt{Clear}(A))) &:= \{\{?x_1{\leftarrow}A\}\}\\
\mathcal{R}_o^*(\texttt{PRE}(\texttt{Clear}(C))) &:= \{\{?x_3{\leftarrow}C\}\}\\
\mathcal{R}_o^*(\texttt{DEL}(\texttt{On}(A, B))) &:= \{\{?x_1{\leftarrow}A, ?x_2{\leftarrow}B\}\}\\
\mathcal{R}_o^*(\texttt{DEL}(\texttt{Clear}(C))) &:= \{\{?x_3{\leftarrow}C\}\}\\
\mathcal{R}_o^*(\texttt{ADD}(\texttt{On}(A, C))) &:= \{\{?x_1{\leftarrow}A, ?x_3{\leftarrow}C\}\}\\
\mathcal{R}_o^*(\texttt{ADD}(\texttt{Clear}(B))) &:= \{\{?x_2{\leftarrow}B\}\}
\end{aligned}$$

It should be clear that this encoding scheme can correctly encode serial action execution. Stated formally we have the following proposition:

Proposition 4.1.3. *Given a precise encoding scheme Θ^*_o and $\mathcal{R}^*_o$, for an operator o and a set of assignment sets Γ, for all actions $a \in \mathcal{A}_o$, $\Gamma_a \models c$ iff $c \in C_a$.*

Proof. For a ground condition $c_o \in C_o$, where $c_o := \tau(\lambda(\vec{x}'))$, we have that $\mathcal{R}^*(c_o) := \{\vec{x}'\}$ and therefore that $\{\vec{x}'\} \models c_o$. According to the definition of Θ^*_o, for an action $a \in \mathcal{A}_o$, $\vec{x}' \in \Gamma_{\mathcal{A}_o}$ iff $c_o \in C_a$. $\square$

Now that it has been shown that the execution of individual actions can be encoded correctly, we need to show that the parallel execution of multiple actions can be encoded without interference. We begin with an illustrative example for the operator $o := \texttt{Drive}(?t, ?from, ?to)$, where $\mathcal{A}_o := \{a_1 := \texttt{Drive}(T_1, L_1, L_2), a_2 := \texttt{Drive}(T_2, L_1, L_3), a_3 := \texttt{Drive}(T_1, L_1, L_3)\}$. We have the following condition relation $\mathcal{R}^*_o$:

$$\begin{array}{rcl}
\mathcal{R}^*_o(\texttt{PRE}(\texttt{At}(T_1, L_1))) & := & \{\{?t{\leftarrow}T_1, ?from{\leftarrow}L_1\}\} \\
\mathcal{R}^*_o(\texttt{DEL}(\texttt{At}(T_1, L_1))) & := & \{\{?t{\leftarrow}T_1, ?from{\leftarrow}L_1\}\} \\
\mathcal{R}^*_o(\texttt{ADD}(\texttt{At}(T_1, L_2))) & := & \{\{?t{\leftarrow}T_1, ?to{\leftarrow}L_2\}\} \\
\mathcal{R}^*_o(\texttt{PRE}(\texttt{At}(T_2, L_1))) & := & \{\{?t{\leftarrow}T_2, ?from{\leftarrow}L_1\}\} \\
\mathcal{R}^*_o(\texttt{DEL}(\texttt{At}(T_2, L_1))) & := & \{\{?t{\leftarrow}T_2, ?from{\leftarrow}L_1\}\} \\
\mathcal{R}^*_o(\texttt{ADD}(\texttt{At}(T_2, L_3))) & := & \{\{?t{\leftarrow}T_2, ?to{\leftarrow}L_3\}\} \\
\mathcal{R}^*_o(\texttt{ADD}(\texttt{At}(T_1, L_3))) & := & \{\{?t{\leftarrow}T_1, ?to{\leftarrow}L_3\}\}
\end{array}$$

The parallel execution of actions a_1 and a_2, which interfere in a simply split representation, is entailed by the set of assignment sets $\Gamma_{\Theta^*_o, a_1} \cup \Gamma_{\Theta^*_o, a_2} :=$

$$\{\{t{\leftarrow}T_1, from{\leftarrow}L_1\}, \{t{\leftarrow}T_1, to{\leftarrow}L_2\}, \{t{\leftarrow}T_2, from{\leftarrow}L_1\}, \{t{\leftarrow}T_2, to{\leftarrow}L_3\}\}$$

This set entails the following set of conditions:

$$\begin{array}{lll}
\{\texttt{PRE}(\texttt{At}(T_1, L_1)), & \texttt{DEL}(\texttt{At}(T_1, L_1)), & \texttt{ADD}(\texttt{At}(T_1, L_2)), \\
\texttt{PRE}(\texttt{At}(T_2, L_1)), & \texttt{DEL}(\texttt{At}(T_2, L_1)), & \texttt{ADD}(\texttt{At}(T_2, L_3))\}
\end{array}$$

This is the set of conditions $C_{a_1} \cup C_{a_2}$, showing that with this precise action representation a_1 and a_2 do not interfere. In particular, the condition $\texttt{ADD}(\texttt{At}(T_1, L_3))$ that is entailed by a_3 and by the parallel execution of actions a_1 and a_2 in the simply split relation, is not entailed in this case.

Proposition 4.1.4. *For an operator o and a precise encoding scheme Θ_o^* and $\mathcal{R}_o^*$, the parallel execution of any two non-mutex actions $\{a_1, a_2\} \subseteq \mathcal{A}_o \times \mathcal{A}_o$ does not interfere.*

Proof. Distinct actions a_1 and a_2 do not interfere if there is no ground condition $c_i \in C_o$, where $c_i \notin C_{a_1}$, $c_i \notin C_{a_2}$, such that $\{\Gamma_{\Theta_o^*,a_1} \cup \Gamma_{\Theta_o^*,a_2}\} \models c_i$. Given $\mathcal{R}_o^*$, each condition $c_i \in C_o$ is entailed by a single assignment set γ_{c_i}, which appears in the set $\Gamma_{\Theta_o^*,a_j}$ for $a_j \in \mathcal{A}_o$ iff $c_i \in C_{a_j}$. □

Corollary 4.1.1. *It follows from Proposition 4.1.4 that Θ_o^* and $\mathcal{R}_o^*$ can represent the conditions of any subset of $\mathcal{A}_o$.*

Under a precise encoding scheme a set of assignment sets always uniquely determines a set of entailed conditions, and therefore a state transition at a plan step. It is not the case though that a set of assignment sets always uniquely determine a set of executed actions. In some situations *redundant* actions may occur.

Definition 4.1.25. An action a_i which instantiates an operator o, is *redundant* with respect to a set of actions $\mathcal{A}' \subset \mathcal{A}_o$, where $a_i \notin \mathcal{A}'$, iff $C_{a_i} \subseteq \bigcup_{a_j \in \mathcal{A}'} C_{a_j}$. □

Importantly, at any state, the parallel execution of a set of actions $\mathcal{A}' \subset \mathcal{A}_o$ is indistinguishable from the parallel execution of the actions in $\mathcal{A}'$ amended with any actions redundant with respect to $\mathcal{A}'$. This follows simply from the fact that the addition of redundant actions to $\mathcal{A}'$ does not change the set $\bigcup_{a \in \mathcal{A}'} C_a$. As an example, for the operator $\texttt{Drive}(?x_1, ?x_2, ?x_3)$ where the negative postcondition is omitted, the simultaneous execution of the actions $\texttt{Drive}(T, L_1, L_2)$ and $\texttt{Drive}(T, L_3, L_4)$ is entailed by the following set of assignment sets:

$$\{\{?t{\leftarrow}T_1, ?from{\leftarrow}L_1\}, \{?t{\leftarrow}T_1, ?to{\leftarrow}L_2\}, \{?t{\leftarrow}T_1, ?from{\leftarrow}L_3\}, \{?t{\leftarrow}T_1, ?to{\leftarrow}L_4\}\}$$

This additionally implies the execution of the actions $\texttt{Drive}(T, L_1, L_4)$ and $\texttt{Drive}(T, L_2, L_3)$. If the negative postcondition was not omitted then $\texttt{Drive}(T, L_1, L_2)$ and $\texttt{Drive}(T, L_3, L_4)$ would be mutually exclusive and therefore not executable in parallel.

The consequence of the preceding discussion is that redundancy does not prevent a precisely split representation from being used to find step-optimal plans. It is possible to post-process plans found using a SAT encoding based on a precisely split action representation to remove redundant actions. For a set of actions $\mathcal{A}'$, this post-processing is performed iteratively. At each iteration a single action $a \in \mathcal{A}'$ is removed where a is redundant given the set $\mathcal{A}' \backslash a$.

If a precisely split action representation is to be used for cost-minimising planning then propositional variables can be included which represent the execution of individual actions (to which we can assign costs) and these can be related to the appropriate propositions representing sets of assignment sets. The assignment sets can be used to encode mutex relationships much more efficiently than the action variables, making the split encoding still useful. In the following section we present a number of planning encodings that use the developed framework. An encoding that includes action variables in addition to variables representing assignment sets is given in Section 4.2.3.

4.2 Encoding Planning using Split Action Representations

In this section we will examine a number of encodings of planning into SAT that are based on the previously described framework for split action representations. We will first examine encodings for the serial case followed by the parallel case. Finally, we will present split encodings for both the serial and the parallel cases that include the action variables from the flat encoding. In the following, the schemata we introduce are intended to be general and, given a bounded planning problem Π_h, a set of argument subsets Θ_o, and condition relations $\mathcal{R}_o$, for each operator $o \in \mathcal{O}$, they will produce an h-step CNF ϕ_h. In particular, we show that given suitable choices for Θ_o and R_o for each $o \in \mathcal{O}$, the encodings we present here are *constructive* and the encoding for the parallel case can be used for parallel step-optimal planning.

The compilations make use of the following propositional variables. For each fluent f occurring at step $t \in \{0, ..., h\}$ we have a variable f^t. For each step $t \in \{0, ..., h-1\}$, operator $o \in \mathcal{O}^t$, and assignment set $\gamma^t \in \Gamma^t_{\Theta_o}$, we create a propositional variable $O\gamma^t$. To indicate that $\gamma^t \in \Gamma^t_a$ for variable $O\gamma^t$, we use the notation $O\gamma^t_a$. An action a^t, which is an instantiation of an operator with name O, then has a natural representation as conjunction of propositional variables:

$$\bigwedge_{\gamma_a \in \Gamma_a} O\gamma^t_a$$

Continuing the examples from the BLOCKS-DIRECT domain in Section 4.1.4 (starting page 53), the action $\texttt{Move}(A, B, C)$, which instantiates operator $\texttt{Move}(?x_1, ?x_2, ?x_3)$, is represented by the conjunction:

$$\texttt{Move}\{?x_1 \leftarrow A\} \wedge \texttt{Move}\{?x_2 \leftarrow B\} \wedge \texttt{Move}\{?x_3 \leftarrow C\} \wedge$$
$$\texttt{Move}\{?x_1 \leftarrow A, ?x_2 \leftarrow B\} \wedge \texttt{Move}\{?x_1 \leftarrow A, ?x_3 \leftarrow C\}$$

In the following sections, we also make use of the following definitions.

Definition 4.2.1. For fluent f^t, let $\texttt{make}(f)^t := \bigcup_{o \in \mathcal{O}} \mathcal{R}^t_o(\texttt{ADD}(f))$, where $\mathcal{R}^t_o(\texttt{ADD}(f)) := \emptyset$ if there is no ground condition $\texttt{ADD}(f) \in \Gamma_{\mathcal{C}_o}$, for an operator condition $\mathcal{C}_o \in \mathbb{C}_o$. □

For example, given the previous precisely split action representation, in a SIMPLE-LOGISTICS problem with at least one action a that instantiates the operator $\texttt{Drive}(?truck, ?from, ?to)$, such that $\texttt{at}(T_1, L_1) \in post^+(a)$, we have that $\texttt{make}(\texttt{at}(T_1, L_1))^t := \{\{?truck \leftarrow T_1, ?to \leftarrow L_1\}\}$.

Definition 4.2.2. For fluent f^t, let $\texttt{break}(f^t) := \bigcup_{o \in \mathcal{O}} \mathcal{R}^t_o(\texttt{DEL}(f))$, where $\mathcal{R}^t_o(\texttt{DEL}(f)) := \emptyset$ if there is no ground condition $\texttt{DEL}(f) \in \Gamma_{\mathcal{C}_o}$, for an operator condition $\mathcal{C}_o \in \mathbb{C}_o$. □

Definition 4.2.3. For fluent f^t, let $\texttt{prec}(f^t) := \bigcup_{o \in \mathcal{O}} \mathcal{R}^t_o(\texttt{PRE}(f))$, where $\mathcal{R}^t_o(\texttt{PRE}(f)) := \emptyset$ if there is no ground condition $\texttt{PRE}(f) \in \Gamma_{\mathcal{C}_o}$, for an operator condition $\mathcal{C}_o \in \mathbb{C}_o$. □

4.2.1 Serial Encodings

This section presents a set of serial encodings based on the previously described framework for split action representations. It includes the standard flat and split serial encodings of Section 3.3 and 3.4 as special cases. First, these encodings all include the following schemata from the plangraph based encodings (Section 3.5):

- Start state axioms (Schema 3.5.1);
- Goal axioms (Schema 3.5.2); and
- Fluent mutex (Schema 3.5.7).

We then have the following schemata:

Schema 4.2.1. Precondition and postcondition axioms: These clauses encode action conditions using assignment set propositions. First, for each step $t \in \{0, ..., h-1\}$, each fluent $f^t \in \mathcal{F}^t$, and each operator $o \in \mathcal{O}$, such that $\texttt{prec}(f) \neq \emptyset$, we have the following clause:

$$(\bigwedge_{\gamma \in \texttt{prec}(f)} O\gamma^t) \rightarrow f^t$$

To represent postconditions, we have the following constraints for each step $t \in \{1, ..., h\}$, each fluent $f^t \in \mathcal{F}^t$, and each operator $o \in \mathcal{O}$. If $\texttt{make}(f) \neq \emptyset$, we have the following clause:

$$(\bigwedge_{\gamma \in \texttt{make}(f)} O\gamma^{t-1}) \rightarrow f^t$$

If $\texttt{break}(f) \neq \emptyset$, we have the following clause:

$$(\textstyle\bigwedge_{\gamma \in \texttt{break}(f)} O\gamma^{t-1}) \rightarrow \neg f^t \qquad \square$$

For example, in a SIMPLE-LOGISTICS problem, with a precisely split encoding, the assignment set $\{?truck \leftarrow T_1, ?from \leftarrow L_1\}$, for the operator $\texttt{Drive}(?truck, ?from, ?to)$, participates in the following precondition and postcondition clauses, for time $t < h$:

$$\begin{aligned}&(\texttt{Drive}\{?truck{\leftarrow}T_1, ?from{\leftarrow}L_1\}^t \rightarrow \texttt{at}(T_1, L_1)^t) \qquad \wedge\\&(\texttt{Drive}\{?truck{\leftarrow}T_1, ?from{\leftarrow}L_1\}^t \rightarrow \neg\texttt{at}(T_1, L_1)^{t+1})\end{aligned}$$

Schema 4.2.2. Explanatory Frame Axioms: These clauses explain how the truth values of fluents change between plan steps. For each step $t \in \{1, ..., h\}$ and each fluent $f^t \in \mathcal{F}^t$ we have:

$$\begin{aligned}&(f^t \rightarrow (f^{t-1} \vee \textstyle\bigvee_{\gamma \in \texttt{make}(f)^t} O\gamma^{t-1})) \qquad \wedge\\&(\neg f^t \rightarrow (\neg f^{t-1} \vee \textstyle\bigvee_{\gamma \in \texttt{break}(f)^t} O\gamma^{t-1}))\end{aligned} \qquad \square$$

These clauses ensure that if f holds (resp. $\neg f$ holds) at step t, then either it holds (resp. $\neg f$ holds) at step $t-1$, or an assignment set variable holds at $t-1$ that f relates to via a positive (resp. negative) postcondition. In combination with the other schemata, this ensures that the execution of actions account for the change in truth value of a fluent. For example, in a SIMPLE-LOGISTICS domain for a fluent $\texttt{at}(T_1, L_1)$ we have clauses, for $t > 0$:

$$\begin{aligned}&\texttt{at}(T_1, L_1)^t \rightarrow (\texttt{at}(T_1, L_1)^{t-1} \vee \texttt{Drive}\{?truck{\leftarrow}T_1, ?to{\leftarrow}L_1\}) \qquad \wedge\\&\neg\texttt{at}(T_1, L_1)^t \rightarrow (\neg\texttt{at}(T_1, L_1)^{t-1} \vee \texttt{Drive}\{?truck{\leftarrow}T_1, ?from{\leftarrow}L_1\})\end{aligned}$$

Next, there are schemata that describe full mutex constraints between actions by enforcing mutex relationships between assignment set variables. First, we have schemata that enforce mutex relationships between assignment sets from different operators. Here we have *strong* and *weak* schemata, which differ in the number of clauses produced, but which both enforce full mutex.

Schema 4.2.3. Strong inter-operator full mutex: For each step $t \in \{0, ..., h-1\}$ and every pair of assignment set variables $O_1\gamma_1^t$ and $O_2\gamma_2^t$ at step t, where O_1 and O_2 name distinct operators, we have the following clause:

$$\neg O_1\gamma_1^t \vee \neg O_2\gamma_2^t \qquad \square$$

Schema 4.2.4. Weak inter-operator full mutex: For each step $t \in \{0, ..., h-1\}$ and every pair of distinct operators o_1 and o_2, with operator names O_1 and O_2, for a single pair of argument subsets $\theta_{o,1} \in \Theta_{o_1}$ and $\theta_{o,2} \in \Theta_{o_2}$, we have a clause for every pair of assignment sets γ_1 and γ_2 such that $\gamma_1 \in \Gamma_{\theta_{o,1}}$ and $\gamma_2 \in \Gamma_{\theta_{o,2}}$:

$$\neg O_1 \gamma_1^t \vee \neg O_2 \gamma_2^t \qquad \square$$

Next, we have a single schemata that enforces mutex between assignment sets from the same operator.

Schema 4.2.5. Intra-operator full mutex: For each step $t \in \{0, ..., h-1\}$, each operator o with operator name O, and argument subset $\theta_o \in \Theta_o$, we have a clause for each distinct pair of distinct assignment sets γ_1 and γ_2 from Γ_{θ_o}:

$$\neg O \gamma_1^t \vee \neg O \gamma_2^t \qquad \square$$

Finally, we require constraints that ensure that whole instances of operators are executed, rather than individual assignment sets. For example, consider a SIMPLE-LOGISTICS problem with two trucks T_1 and T_2 and three locations L_1, L_2, and L_3. If $\texttt{Drive}\{?t{\leftarrow}T_1, ?to{\leftarrow}L_2\}$ is true, then we need constraints ensuring all assignment sets associated with either $\texttt{Drive}(T_1, L_1, L_2)$ or $\texttt{Drive}(T_1, L_3, L_2)$ are true. We call such constraints *grounding support axioms*. We develop our *grounding support* in terms of *dependency trees*. For an operator o^t, such a tree is defined for each assignment set γ where $\gamma \in R_o^t(\texttt{ADD}(f))$, for some condition $\texttt{ADD}(f)$.[4]

Definition 4.2.4. A *dependency tree* is a directed acyclic graph, where each node n represents an assignment proposition denoted by $O\gamma_n$. We then have:

- A dependency tree for an assignment set γ of an operator o is rooted at a node n_0 labelled with $O\gamma_{n_0}$;
- $\texttt{prefix}(n)$ denotes the set of nodes in the root path of dependency tree node n;
- $\texttt{parent}(n)$ denotes the parent of dependency tree node n;
- $\texttt{children}(n)$ denotes the child nodes of dependency tree node n; and

[4]Here the set of assignment sets includes auxiliary variables added for Schema 4.2.9.

- A node n labelled with $O\gamma_n$ has a set $\texttt{children}(n)$ with all of the assignment sets of some $\theta_o \in \Theta_o$, such that $\theta_o \notin \texttt{prefix}(n)$ and such that each assignment set $\gamma_i \in \Gamma_{\theta_o}$ co-occurs with γ in Γ_a for some $a \in \mathcal{A}_o$. $\square$

In practice, we build trees in a depth first manner choosing an argument subset θ to generate the children of each node n such that θ produces the fewest children for n. When generating dependency trees, we exclude nodes for assignment sets of argument subsets that are strict subsets of other argument subsets. These relationships between argument subsets are encoded independently of any dependency trees with the following set of clauses.

Schema 4.2.6. For each step $t \in \{0, ..., h-1\}$, each operator o with name O, and each pair of argument subsets $\theta_{o,1}, \theta_{o,2} \in \Theta_o$, such that $\theta_{o,1} \subset \theta_{o,1}$, we produce clauses as follows. For each assignment sets $\gamma_i \in \Gamma_{\theta_{o,1}}$ and each assignment set $\gamma_j \in \Gamma_{\theta_{o,1}}$, such that $\gamma_i \subset \gamma_j$ we have a clause:

$$O\gamma_i \rightarrow O\gamma_j$$ $\square$

For example, in the precisely split encoding of the BLOCKS-DIRECT domain, of the `Move` operator we might have a clause:

$$\texttt{Move}\{?block{\leftarrow}A, ?from{\leftarrow}B\} \rightarrow \texttt{Move}\{?block{\leftarrow}A\}$$

Intuitively, a decision tree enforces grounding support by being translated into a set of constraints that assert that, for each node n, if all of the assignment sets labelling the nodes in the set $\texttt{prefix}(n) \cup n$ are true, then at least one assignment set labelling a node in the set $\texttt{children}(n)$ must also be true. Before these constraints are described formally, consider for example, a precisely split representation of a SIMPLE-LOGISTICS problem with two trucks T_1 and T_2 and three locations L_1, L_2, and L_3. To make this example more illustrative, we assume that the representation includes the static propositions instantiating $\texttt{road}(?from, ?to)$ as preconditions of actions instantiating $\texttt{Drive}(?t, ?from, ?to)$. We also assume that all actions have an additional precondition which instantiates $\texttt{in-service}(?t)$, for truck $?t$, and which ensures that all `Pick-up`, `Drop-off`, and `Drive` actions are performed with a truck that is in service. All of the assignment sets and the associated ground conditions and actions for the `Drive` operator are presented in Table 4.3. If, for example, the proposition for assignment set $\{?t \leftarrow T_1, ?to \leftarrow L_2\}$ is true, the constraints we generate will ensure that propositions for all assignment sets associated with

Assignment sets related to $\{?t{\leftarrow}T_1, ?to{\leftarrow}L_2\}$	Conditions	Instantiations of $\texttt{Drive}(?t, ?from, ?to)$
$\gamma_0 := \{?t{\leftarrow}T_1, ?to{\leftarrow}L_2\}$	$\texttt{ADD}(\texttt{at}(T_1, L_2))$	$(T_1, L_1, L_2), (T_1, L_3, L_2)$
$\gamma_1 := \{?from{\leftarrow}L_1, ?to{\leftarrow}L_2\}$	$\texttt{PRE}(\texttt{road}(L_1, L_2))$	$(T_1, L_1, L_2), (T_2, L_1, L_2)$
$\gamma_2 := \{?from{\leftarrow}L_3, ?to{\leftarrow}L_2\}$	$\texttt{PRE}(\texttt{road}(L_3, L_2))$	$(T_1, L_3, L_2), (T_2, L_3, L_2)$
$\gamma_3 := \{?t{\leftarrow}T_1, ?from{\leftarrow}L_1\}$	$\texttt{PRE}(\texttt{at}(T_1, L_1))$, $\texttt{DEL}(\texttt{at}(T_1, L_1))$	$(T_1, L_1, L_2), (T_1, L_1, L_3)$
$\gamma_4 := \{?t{\leftarrow}T_1, ?from{\leftarrow}L_3\}$	$\texttt{PRE}(\texttt{at}(T_1, L_3))$, $\texttt{DEL}(\texttt{at}(T_1, L_3))$	$(T_1, L_3, L_2), (T_1, L_3, L_1)$
$\gamma_5 := \{?t{\leftarrow}T_1\}$	$\texttt{PRE}(\texttt{in-service}(T_1))$	$(T_1, L_1, L_2), (T_1, L_1, L_3)$ $(T_1, L_2, L_1), (T_1, L_2, L_3)$ $(T_1, L_3, L_1), (T_1, L_3, L_2)$

Table 4.3: Assignment sets and their associated conditions and actions from a modified SIMPLE-LOGISTICS instance with trucks T_1 and T_2 and locations L_1, L_2, and L_3. The left column lists assignment sets that occur in an action that instantiates $\texttt{Drive}(?t, ?from, ?to)$ with $\{?t{\leftarrow}T_1, ?to{\leftarrow}L_2\}$. The centre column gives conditions entailed by each assignment set. The right column gives instances of $\texttt{Drive}(?t, ?from, ?to)$ that each assignment set occurs in.

either $\texttt{Drive}(T_1, L_1, L_2)$ or $\texttt{Drive}(T_1, L_3, L_2)$ are true. The topmost tree of Figure 4.1 is for assignment set $\{?t{\leftarrow}T_1, ?to{\leftarrow}L_2\}$. The constraints generated from decision trees are described with the following schema.

Schema 4.2.7. Given a set of dependency trees for an operator with name O, we generate a clause for each decision tree node n. Where n_0 is the root node of the tree containing n, we generate the following clause:

$$((\bigwedge_{\substack{n_x \in \{\mathit{prefix}(n) \cup n \backslash n_0\}, \\ |\mathit{children}(\mathit{parent}(n_x))| > 1}} O\gamma_{n_x}) \wedge O\gamma_{n_0}) \rightarrow \bigvee_{n_y \in \mathit{children}(n)} O\gamma_{n_y} \qquad \square$$

For example, the top-right of Figure 4.1 gives the constraints that are compiled from the *restricted* dependency tree for $\{?t{\leftarrow}T_1, ?to{\leftarrow}L_2\}$ of the operator $\texttt{Drive}(?t, ?from, ?to)$. The notion of relaxing restricted dependency trees to be useful for parallel encodings will be explored in the next section.

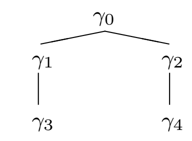
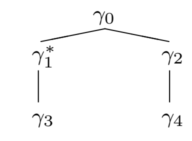
Restricted dependency tree	Corresponding constraints:- Omitting nodes: γ_5
	$\gamma_0 \rightarrow \gamma_5$ Compiling dependency tree: $\gamma_0 \rightarrow (\gamma_1 \vee \gamma_2)$; $(\gamma_0 \wedge \gamma_1) \rightarrow \gamma_3$ $(\gamma_0 \wedge \gamma_2) \rightarrow \gamma_4$
Dependency tree with condition copies	Corresponding constraints:- Omitting nodes: γ_5
	$\gamma_0 \rightarrow \gamma_5$ Compiling dependency tree: $\gamma_0 \rightarrow (\gamma_1^* \vee \gamma_2)$; $(\gamma_0 \wedge \gamma_1^*) \rightarrow \gamma_3$ $(\gamma_0 \wedge \gamma_2) \rightarrow \gamma_4$; $\gamma_1^* \rightarrow \gamma_1$

Figure 4.1: For γ_0 in Table 4.3: (above) *Restrictive* dependency tree, and (below) the same tree with a assignment set copy γ_1^*. The clauses derived from each tree are listed to their right.

The previous constraints define two encodings: split serial strong mutex $\mathcal{T}^{Split|S}$ and split serial weak mutex $\mathcal{T}^{Split|W}$. We begin by showing the constructiveness of $\mathcal{T}^{Split|S}$ and then extend the proof to cover $\mathcal{T}^{Split|S}$.

Theorem 4.2.1. *The precisely split serial encodings $\mathcal{T}^{Split|W}$ and $\mathcal{T}^{Split|S}$ of a planning problem Π into SAT are constructive for any set of argument subsets Θ_o and condition map $\mathcal{R}_o$ for $o \in \mathcal{O}$ that satisfy Proposition 4.1.2 for planning problems with a serial execution semantics.*

Proof. Our proof proceeds by taking all observations about$\mathcal{T}^{Split|W}$ to hold for $\mathcal{T}^{Split|S}$ equally, unless otherwise stated. Addressing condition (i), we sketch a plan extraction algorithm which takes a satisfying valuation $\mathcal{V}$ of $\mathcal{T}^{Split|W}(\Pi_h)$ and extracts a valid plan. Here $\mathcal{V}$ specifies a sequence of sets of true fluent variables $\mathcal{F}_\top := \langle \mathcal{F}_\top^0, ..., \mathcal{F}_\top^h \rangle$, and a sequence of sets of true assignment set variables $\Gamma_\top := \langle \Gamma_\top^0, ..., \Gamma_\top^{h-1} \rangle$. Our algorithm produces a sequence of sets of actions $\mathcal{A}_\top := \langle \mathcal{A}_\top^0, ..., \mathcal{A}_\top^{h-1} \rangle$ that represents a plan π_h, such that $\mathcal{F}_\top$ represents the sequence of states visited when that plan is realised, expressed here as $\mathcal{S}_\top := \langle s_\top^0, ..., s_\top^h \rangle$. Each set of actions $\mathcal{A}_\top^t \in \mathcal{A}_\top$ is built by including the action $a \in \mathcal{A}$ where $\Gamma_a \subseteq \Gamma_\top^t$. It is clear that this algorithm runs in linear time in the size of $\Gamma_\top$, $\mathcal{F}_\top$ and $\mathcal{A}$.

We are left to demonstrate that π_h is a valid serial plan for Π_h. As $\mathcal{V}$ is a satisfying valuation of $\mathcal{T}^{Split|W}(\Pi_h)$, all clauses implied by the Schemata of $\mathcal{T}^{Split|W}$ are satisfied. From Definition 2.1.16 we require that:

1. $s_\top^0 = s_0$. This holds because for each $f_i \in s_0$, we have a unit clause (Schema 3.3.1) asserting that $f_i^0 \in \mathcal{V}$;

2. $\mathcal{G} \subseteq s^h_\top$. This holds because we have a unit clause (Schema 3.3.2) asserting that for every fluent $f_i \in \mathcal{G}$, $f^h_i \in \mathcal{V}$;

3. For each $\Gamma^t_\top \in \Gamma_\top$ there exists at most one action $a^t_\top \in \mathcal{A}$, such that $\Gamma_{a^t_\top} \subseteq \Gamma^t_\top$ and additionally $\neg\exists c := \texttt{ADD}(f)$ such that $\Gamma^t_\top \models c$ and $c \notin C_{a^t_\top}$. These conditions encode serial action execution but allow additional groundings to be true if they do not entail a positive postcondition. This means that fluents are allowed to become false without the execution of an action. This can significantly reduce the number of constraints required to encode grounding support and does not affect the plan validity, as all goals and action preconditions are positive. If an action $a^t_\top$ instantiates operator o then by Proposition 4.1.2 there is one assignment set $\gamma \in \Gamma_{a^t_\top}$ that instantiates each $\theta \in \Theta_o$. Schema 4.2.5 ensures that for every $\theta \in \Theta_o$, at most one assignment set in Γ_θ can be in $\Gamma^t_\top$. For the encoding $\mathcal{T}^{Split|S}$, Schema 4.2.3 ensures that if any argument set for o is true no assignment set for a distinct operator o' can be in $\Gamma^t_\top$. For the encoding $\mathcal{T}^{Split|W}$, Schema 4.2.4 ensures if any argument set for o is true no assignment set for some argument of a distinct operator o' can be in $\Gamma^t_\top$. As every action consists of one assignment set for each argument subset of an operator, this prevents any action instantiating o' being true. Next, we need to show that any true assignment set that entails a positive postcondition is in the set Γ_{a^t} for some action $a^t \in \mathcal{A}^t_\top$. Such an assignment set $\gamma \in \Gamma_{\theta'_o}$ becomes the node at the base of a dependency tree according to Definition 4.2.4. Constraints generated from this tree force one assignment set in each $\{\Theta_o \backslash \theta'_o\}$ to be true. In particular, let $\langle \theta_{o'_0}, \theta_{o,1}, ..., \theta_{o,k} \rangle$ be an arbitrary ordering over the set $\{\Theta_o \backslash \theta'_o\}$. The constraints in Schemata 4.2.6 and 4.2.7 ensure that if we have true assignment sets $\gamma_0, \gamma_1, ..., \gamma_j$ instantiating $\theta'_o, \theta_{o,1}, ..., \theta_{o,j}$, for any $j < k$ then we have a true assignment set $\gamma_{j+1} \in \Gamma_{\theta_{o,j+1}}$ such that if $\gamma_{j+1} \in \Gamma_a$ then $\{\gamma_0, \gamma_1, ..., \gamma_j\} \subset \Gamma_a$. By induction, once we have one assignment set γ'_0 that entails a positive postcondition, then we must have all assignment sets for some action a that has γ'_0.

4. Let $a^t_\top$ be the single action that is entailed by $\Gamma^t_\top$. For each $\Gamma^t_\top \in \Gamma_\top$, we need to ensure that *pre*$(a^t_\top) \subseteq s^t_\top$ and $s^{t+1}_\top \subseteq \{\{s^t_\top \backslash \textit{post}^-(a^t_\top)\} \cup \textit{post}^+(a^t_\top)\}$. In the case that there is no such $a^t_\top$, we need to ensure that any assignment sets in $\Gamma^t_\top$ entail conditions such that $s^{t+1}_\top \subseteq s^t_\top$. First, the case where $\Gamma^t_\top \models a^t_\top$. Schema 4.2.1 ensures that *pre*$(a^t_\top) \subseteq s^t_\top$ by making a clause $O\gamma^t \rightarrow f^t$, where $\gamma \models \texttt{PRE}(\mathit{fluent})$ for $f \in$ *pre*$(a^t_\top)$. Schema 4.2.1 ensures that *post*$^+(a^t_\top) \subseteq s^{t+1}_\top$ and *post*$^-(a^t_\top) \cap s^{t+1}_\top = \emptyset$. It does so because for every assignment

set $\gamma \in \Gamma_{a^t_\top}$, such that $\gamma \models \texttt{ADD}(f)$, there is a clause $O\gamma^t \rightarrow f^{t+1}$. For every assignment set $\gamma \in \Gamma_{a^t_\top}$, such that $\gamma \models \texttt{DEL}(f)$ and $\gamma \not\models \texttt{ADD}(f)$, there is a clause $O\gamma^t \rightarrow \neg f^{t+1}$. The explanatory frame axioms (Schema 4.2.2) ensure that if a fluent $f \in s^t_\top$ and $f \notin s^{t+1}_\top$ then for some $\gamma \in \Gamma^i_\top$, $\gamma \models \texttt{DEL}(f)$ with a clause $(f^t \wedge \neg f^{t+1}) \rightarrow \bigvee_{\gamma' \in \texttt{break}(f)} \gamma'$. Finally, fluent $f \notin s^t_\top$ and $f \in s^{t+1}_\top$ then for some $\gamma \in \Gamma^i_\top$, $\gamma \models \texttt{ADD}(f)$ with a clause $(\neg f^t \wedge f^{t+1}) \rightarrow \bigvee_{\gamma' \in \texttt{make}(f)} \gamma'$.

It remains to show (ii), that every valid sequential plan π_h for Π_h can be translated into a satisfying valuation $\mathcal{V}$ of $\mathcal{T}^{Split|W}(\Pi_h)$ or $\mathcal{T}^{Split|S}(\Pi_h)$. Following the logic of the above proof for (i), it is easy to see that the assignment $\mathcal{V}$ that represents the fluents $\mathcal{F}_\top$ and assignment sets $\Gamma_\top$ implied by the plan π_h satisfies the clauses in the encodings $\mathcal{T}^{Split|W}(\Pi_h)$ and $\mathcal{T}^{Split|S}(\Pi_h)$. $\square$

4.2.2 Parallel Encodings

The main result of this chapter is the presentation of a split encoding that can be used for parallel step-optimal planning, which uses the previously described framework for split action representations. The previously described encoding for serial planning can be used for this purpose if the clauses that encode mutex (Schemata 4.2.3, 4.2.4, and 4.2.5) are replaced by the constraints described in this section. To encode both mutex and grounding support in the parallel case, auxiliary variables are introduced. These auxiliary variables will be described when the appropriate constraints are defined. As in the serial case, there are two distinct sets of constraints to describe mutex relationships between assignment set variables: the inter-operator mutex constraints and the intra-operator mutex constraints.

With a standard parallel action execution semantics, a pair of actions a_1 and a_2 is mutex because of at least one pair of preconditions and postconditions c_1 and c_2, such that $c_1 \in C_{a_1}$ and $c_2 \in C_{a_2}$ – i.e. a pair of preconditions to mutex fluents, a positive and a negative postcondition to the same fluent, or a precondition and a negative postcondition to the same fluent. In general, a pair of conditions may be responsible for a set of mutex relationships. For example, in SIMPLE-LOGISTICS problem, the pair $\texttt{PRE}(\texttt{At}(T_1, L_1))$ and $\texttt{DEL}(\texttt{At}(T_1, L_1))$ is responsible for the mutex relationships between actions $\texttt{Drive}(T_1, L_1, L_2)$ and $\texttt{Drive}(T_1, L_1, L_3)$, $\texttt{Drive}(T_1, L_1, L_2)$ and $\texttt{Drive}(T_1, L_1, L_4)$, etc.

As noted for the flat plangraph-based encodings described in Chapter 3, for a pair of actions a_1 and a_2 that are mutex due to a pair of mutex preconditions or an interfering positive postcondition and enforced negative postcondition, the constraints describing the relevant fluent mutex and conditions enforce the mutex relationship between a_1 and a_2. This means that explicit mutex constrains only need to be introduced for actions that are mutex only due to the interaction of a precondition and a negative postcondition or a positive postcondition and a non-enforced negative postcondition. In this case, and where a_1 instantiates an operator o_1, a_2 instantiates an operator o_2, and $o_1 \neq o_2$, mutex is enforced between a_1 and a_2 by asserting mutex relationships between the assignment sets that entail the conditions responsible for the mutex.

Schema 4.2.8. Inter-operator mutex: For each step $t \in \{0, ..., h-1\}$ and each pair of ground conditions c_1 of operator o_1 and c_2 of operator o_2, such that $o_1 \neq o_2$, we assert a mutex relationship between the assignment set variables that entail c_1 and c_2 iff both of the following hold:

- One of c_1 and c_2 is of the form $\texttt{PRE}(f)$ while the other is of the form $\texttt{DEL}(f)$; or one of c_1 and c_2 is of the form $\texttt{ADD}(f)$ while the other is of the form $\texttt{DEL}(f)$ and is not enforced by Schema 4.2.1; and
- There is a pair of actions, a_1 which instantiates o_1, and a_2 which instantiates operator o_2, such that $c_1 \in C_{a_1}$ and $c_2 \in C_{a_2}$ and there are no constraints instantiating Schemata 4.2.1 and 3.5.7 that prevent the parallel execution of a_1 and a_2.

In this case, we have two sets of assignment sets Γ_1 and Γ_2, such that $\Gamma_1 \models c_1$ and $\Gamma_2 \models c_2$. Where O_1 is the name of operator o_1 and O_2 is the name of operator o_2, we have the following clause:

$$(\bigvee_{\gamma_i \in \Gamma_1} \neg O_1 \gamma_i^t) \vee (\bigvee_{\gamma_i \in \Gamma_2} \neg O_2 \gamma_i^t) \qquad \square$$

For example, in a precisely split encoding of a SIMPLE-LOGISTICS domain, the parallel execution of any action instantiating the `Drive` operator with the negative postcondition $\texttt{DEL}(\texttt{At}(T_1, L_1))$ and any action instantiating the `Pick-up` action with the precondition $\texttt{PRE}(\texttt{At}(T_1, L_1))$ is ruled out by the following clause:

$$\neg\texttt{Drive}\{?t{\leftarrow}T_1, ?from{\leftarrow}L_1\} \vee \neg\texttt{Pick-up}\{?t{\leftarrow}T_1, ?from{\leftarrow}L_1\}$$

In the case where we need to rule out the parallel execution of actions which instantiate the same operator, the schema defined for the inter-operator case is usually inappropriate. This is because the set of assignment sets Γ that entails both of c_1 and c_2, from which a mutex clause will be made, may also prohibit the execution of a set of non-mutually exclusive actions. For example, a single assignment set $\{?t \leftarrow T_1, ?from \leftarrow L_1\}$ for the `Drive` operator entails the conditions $\texttt{PRE}(\texttt{At}(T_1, L_1))$ and $\texttt{DEL}(\texttt{At}(T_1, L_1))$. The negative unit clause that would result from the application of Schema 4.2.8: $\neg\texttt{Drive}\{?t{\leftarrow}T_1, ?from{\leftarrow}L_1\}$[5], as well as ruling out the parallel execution of all pairs of actions mutex due to the previously mentioned conditions, would also rule out the execution of the individual actions. To represent the required mutex relationships between actions which instantiate the same operator, we create a set of auxiliary assignment set variables. To represent mutex between actions a_1 and a_2 we choose two such variables $O\gamma'$ and $O\gamma''$ and enforce that $\gamma' \in \Gamma_{a_1}$ and $\gamma'' \in \Gamma_{a_2}$ with grounding support constraints. We then use these auxiliary assignment set variables to enforce mutex as described in the following Schema.

Schema 4.2.9. For each step $t \in \{0, ..., h-1\}$ and each pair of mutex actions a_1 and a_2 that instantiate the operator with name O, where auxiliary assignment set $\gamma' \in \Gamma_{a_1}$ and auxiliary assignment set $\gamma'' \in \Gamma_{a_2}$, we have a clause:

$$\neg O\gamma' \vee \neg O\gamma'' \qquad \square$$

Auxiliary assignment set variables only appear in this Schema and Schemata 4.2.7 and 4.2.10. and they do not entail any conditions. Moreover, in our implementation of the compilation, we generate a minimal number of auxiliary condition variables given mutex relations are processed sequentially. That is, we opportunistically re-use auxiliary variables to encode more than one mutex relation provided this does not cause legal parallel executions of actions to be forbidden.

Finally, as in the serial case, we require *grounding support axioms* to ensure that whole instances of operators are executed (in parallel) at each step t, rather than individual assignment sets. As the name of the top tree in Figure 4.1 suggests, the constraints are overly restrictive. In particular, in our modified SIMPLE-LOGISTICS example with a parallel execution semantics we must be able to execute $\texttt{Drive}(T_1, L_3, L_2)$ and $\texttt{Drive}(T_2, L_1, L_2)$ in parallel without the assignment set variable $\texttt{Drive}\{?t{\leftarrow}T_1, ?from{\leftarrow}L_1\}$, and hence action $\texttt{Drive}(T_1, L_1, L_2)$, being true. In order to relax the constraints developed so far, we introduce the notion of an *assignment set copy*. A tree node $O\gamma_n$ can be labelled with a copied assignment set, in which case we have an additional variable $O\gamma_n^*$. Constraints are then required to ensure that an assignment set copy implies that the original is also true.

Schema 4.2.10. For each copied assignment set variable $O\gamma_n^*$ we have a clause:

$$O\gamma_n^* \rightarrow O\gamma_n \qquad \square$$

[5]Simplified from the clause $\neg\texttt{Drive}\{?t{\leftarrow}T_1, ?from{\leftarrow}L_1\} \vee \neg\texttt{Drive}\{?t{\leftarrow}T_1, ?from{\leftarrow}L_1\}$

In compiling a dependency tree to CNF, we use the assignment set copy variable in place of the original assignment set variable. The bottom half of Figure 4.1 gives the example where γ_1 from Table 4.3 is copied for the tree rooted at γ_0.

Although the restrictiveness of the dependency trees developed thus far could be repaired by making every tree node a copy, we now describe a two phase procedure that introduces few copies while removing all spurious restrictions. The first phase proceeds bottom-up from the leaves to the root (i.e., a node is not processed until all its descendants are), labelling a node n as copied if there is a parallel execution of actions whose precisely split representation includes all the assignment sets $\texttt{prefix}(n) \cup n$ and excludes the assignment sets in $\texttt{children}(n)$.

Definition 4.2.5. When processing node n, for operator o, where we have that $\texttt{prefix}(n) \neq \emptyset$ and $\texttt{children}(n) \neq \emptyset$, for $n_i \in \{\texttt{prefix}(n) \cup n\}$ we compute the sets of actions $\mathcal{A}_n(n_i)$ defined as follows:

$$\begin{aligned}\mathcal{A}_n(n_i) := \{a| \quad & a \in \mathcal{A}_o, \gamma_{n_i} \in \Gamma_a, \\ & \forall n_y \in \texttt{children}(n), (\gamma_{n_y} \notin \Gamma_a \vee \\ & (\exists \gamma^*_{n_y} \wedge \exists n_z \in \texttt{prefix}(n_y), \gamma_{n_z} \notin \Gamma_a))\}\end{aligned} \quad \square$$

Above, $\exists O\gamma^*_n$ is true iff n is labelled as a copy. If for all i $\mathcal{A}_n(n_i)$ are non-empty, then n is labelled as a copy. Although the first phase is sufficient to make the constraints compiled from the resultant dependency trees logically correct, we perform a second top-down pass to remove some redundant copies. We apply the same test to decide if a node should be copied as in the first phase, however a copied node is interpreted as a placeholder for the conjunct of itself with the conditions in its root path – i.e., we suppose $O\gamma^*_n \equiv \bigwedge_{n_i \in \texttt{prefix}(n) \cup n} O\gamma_{n_i}$.

Definition 4.2.6. Formally, when processing node n, for operator o, we compute the set $\mathcal{A}'_n(n_i)$ for $n_i \in \{\texttt{prefix}(n) \cup n\}$ so that, $\mathcal{A}'_n(n_i) := \mathcal{A}_n(n_i)$ if $\nexists \gamma^*_{n_i}$, and otherwise:

$$\begin{aligned}\mathcal{A}'_n(n_i) := \{a| \quad & a \in \mathcal{A}_o, \\ & n_x \in \{\texttt{prefix}(n_i) \cup n_i\}, \gamma_{n_x} \in \Gamma_a, \\ & \forall n_y \in \texttt{children}(n), (\gamma_{n_y} \notin \Gamma_a \vee \\ & (\exists \gamma^*_{n_y} \wedge \exists n_z \in \texttt{prefix}(n_y), \gamma_{n_z} \notin \Gamma_a))\}\end{aligned} \quad \square$$

If $\mathcal{A}'_n(n_i)$ is empty for some i then we retract the condition copy label of n. Having been weakened according to the two passes just described, the modified dependency trees are compiled into clauses that prevent partial executions of actions while supporting all legal unique parallel executions of actions from the plangraph.

We let $\mathcal{T}^{Split||W}$ denote the encoding that results from applying the previous constraints to a problem Π using Θ^*_o and R^*_o for each operator o that define a precisely split encoding.

Theorem 4.2.2. *The precisely split parallel encoding $\mathcal{T}^{Split||W}$ of a planning problem Π_h into SAT is constructive for a set of argument subsets Θ^*_o (Definition 4.1.23) and condition map $\mathcal{R}^*_o$ (Definition 4.1.24) for $o \in \mathcal{O}$ and planning problems with a conflict action execution semantics.*

Proof. Addressing condition (i), we sketch a plan extraction algorithm which takes a satisfying valuation $\mathcal{V}$ of $\mathcal{T}^{Split||W}(\Pi_h)$, and extracts a valid plan. Here $\mathcal{V}$ specifies a sequence of sets of true fluent variables $\mathcal{F}_\top := \langle \mathcal{F}^0_\top, ..., \mathcal{F}^h_\top \rangle$, and a sequence of sets of true assignment set variables $\Gamma_\top := \langle \Gamma^0_\top, ..., \Gamma^{h-1}_\top \rangle$. Our algorithm produces a sequence of sets of actions $\mathcal{A}_\top := \mathcal{A}^0_\top, ..., \mathcal{A}^{h-1}_\top$ that represents a plan π_h, such that $\mathcal{F}_\top$ represents the sequence of states visited when that plan is realised, expressed here as $\mathcal{S}_\top := \langle s^0_\top, ..., s^h_\top \rangle$. Each set of actions $\mathcal{A}^t_\top \in \mathcal{A}_\top$ is built by adding each action $a \in \mathcal{A}$, where $\Gamma_a \subseteq \Gamma^t_\top$. The algorithm iteratively removes redundant actions according to Definition 4.1.25. It is clear that this algorithm runs in polynomial time in the size of $\mathcal{A}$.

We are left to demonstrate that π_h is a valid parallel plan for Π_h. As $\mathcal{V}$ is a satisfying valuation of $\mathcal{T}^{Split||W}(\Pi_h)$, all clauses implied by the Schemata of $\mathcal{T}^{Split||W}$ are satisfied. From Definition 2.1.16 we require that:

1. $s^0_\top = s_0$. This holds because for each $f_i \in s_0$, we have a unit clause (Schema 3.3.1) asserting that $f^0_i \in \mathcal{V}$;

2. $\mathcal{G} \subseteq s^h_\top$. This holds because we have a unit clause (Schema 3.3.2) asserting that for every fluent $f_i \in \mathcal{G}$, $f^h_i \in \mathcal{V}$;

3. For each step t, there exists set of actions $\mathcal{A}^t_\top := \{a^t_{\top,0}, ..., a^t_{\top,k}\}$, such that $\Gamma_{\mathcal{A}^t_\top} \subseteq \Gamma^t_\top$ and additionally $\neg\exists c := \texttt{ADD}(f)$ such that $\Gamma^t_\top \models c$ and $c \notin C_{\mathcal{A}^t_\top}$. These conditions encode parallel action execution, but allow additional groundings to be true if they do not entail a positive postcondition. First, we need to ensure that $\mathcal{A}^t_\top$ does not contain a pair of mutex actions. For an action a_1 to be mutex with another action a_2, either there exists a pair of fluents f_1 and f_2, such that $f_1 \in pre(a_1)$ and $f_2 \in pre(a_2)$ and $(t, f_1, f_2) \in \mu_{pg-p}$ or a_1 and a_2 conflict, that is $(a_1, a_2) \in \mu_c$, because of a fluent f. In the first case, Γ_{a_1} will

contain a grounding γ_1, such that $\gamma_1 \models c := \texttt{PRE}(f_1)$ and Γ_{a_2} will contain a grounding γ_2, such that $\gamma_2 \models c := \texttt{PRE}(f_2)$ and $(t, f_1, f_2) \in \mu_{pg-p}$. The condition relation $\mathcal{R}_o^*$ means that we will have clauses $O\gamma_1^t \rightarrow f_1^t$, $O\gamma_2^t \rightarrow f_2^t$ (Schema 4.2.1), and $\neg f_1^t \vee \neg f_2^t$ (Schema 3.5.7). These clauses imply that only one of a_1 and a_2 can be in $\mathcal{A}_\top^t$.

A similar argument can be made for two actions that conflict due to a positive postcondition and an enforced negative postcondition. Two actions a_1 and a_2 that conflict only due to a precondition and a negative postcondition to a fluent f need to be ruled out explicitly. In this case a_1 will have an auxiliary assignment set γ' and a_2 will similarly have γ''. These sets will be ruled out with a clause $\neg O\gamma' \vee \neg O\gamma''$ according to Schema 4.2.9.

In the case that a_1 and a_2 instantiate different operators and are mutex due to a precondition (entailed by γ_1) and a negative postcondition (entailed by γ_2) to a fluent f then the groundings (and therefore the actions) will be made mutex explicitly with a clause $\neg\gamma_1 \vee \neg\gamma_2$ (Schema 4.2.8).

Now that we have shown that the set of assignment sets $\Gamma_\top^t$ cannot entail a set of actions $\mathcal{A}_\top^t$ which contains mutex actions, we need to ensure that any true assignment set in $\Gamma_\top^t$ that entails a positive postcondition occurs as a part of an action in $\mathcal{A}_\top^t$. Such an assignment set $\gamma \in \Gamma_{\theta'_o}$ becomes the node at the base of a dependency tree according to Definition 4.2.4. Clauses generated from this tree force one assignment set in each $\{\Theta_o \backslash \theta'_o\}$ to be true in $\mathcal{V}$. In particular, let $\langle \theta_{o'_0}, \theta_{o,1}, ..., \theta_{o,k} \rangle$ be an arbitrary ordering over the set $\{\Theta_o \backslash \theta'_o\}$. The clauses from Schemata 4.2.6 and 4.2.7 ensure that if we have true assignment sets $\gamma_0, \gamma_1, ..., \gamma_j$ instantiating $\theta'_o, \theta_{o,1}, ..., \theta_{o,j}$, for any $j < k$, then we have a true assignment set $\gamma_{j+1} \in \Gamma_{\theta_{o,j+1}}$ such that if $\gamma_{j+1} \in \Gamma_a$ then $\{\gamma_0, \gamma_1, ..., \gamma_j\} \subset \Gamma_a$. By induction, once we have one assignment set γ'_0 that entails a positive postcondition, then we must have all assignment sets for some action a that has γ'_0. Now, let $\Gamma_\top^i = \Gamma_{a_\top^t}$ be such a set of true assignment sets indicating that action $a_\top^t$ is executed at step t. Unlike the serial case where at most one action instantiating an operator will be true, here the set $\Gamma_\top^t$ can contain assignment sets for more than one action instantiating operator o. In this case copies need to be made of dependency tree nodes according to Definitions 4.2.5 and 4.2.6. In particular if action a_1 has an assignment set γ_1 and a_2 has γ_2 such that we have a clause $(\gamma_1 \wedge \gamma 2) \rightarrow \gamma_3$, where $\gamma_3 \notin \Gamma_{a_1} \cup \Gamma_{a_2}$, then Definitions 4.2.5 and 4.2.6 ensure that a copy of either γ_1 or γ_2 is made in the relevant grounding support trees. This copy only appears in the clauses generated by Schema 4.2.7 and also imply the original assignment set (Schema 4.2.10).

4. For each step t and set of assignment sets $\Gamma^t_\top$ that entails the execution of the set of actions $\mathcal{A}^t_\top$, we need to ensure that $pre(\mathcal{A}^t_\top) \subseteq s^t_\top$ and $s^{t+1}_\top \subseteq \{\{s^t_\top \backslash post^-(\mathcal{A}^t_\top)\} \cup post^+(\mathcal{A}^t_\top)\}$. If $\mathcal{A}^t_\top = \emptyset$, then we need to ensure that any assignment sets in $\Gamma^t_\top$ entail conditions such that $s^{t+1}_\top \subseteq s^t_\top$. Schema 4.2.1 ensures that $pre(\mathcal{A}^t_\top) \subseteq s^t_\top$ by adding a clause $O\gamma^t \rightarrow f^t$, for every assignment set $\gamma \in \Gamma^t_\top$, where $\gamma \models \texttt{PRE}(fluent)$. It also ensures that $post^+(\mathcal{A}^t_\top) \subseteq s^{t+1}_\top$ by adding a clause $O\gamma^t \rightarrow f^{t+1}$ for every assignment set $\gamma \in \Gamma^t_\top$, such that $\gamma \models \texttt{ADD}(f)$. Schema 4.2.1 ensures that $post^-(\mathcal{A}^t_\top) \cap s^{t+1}_\top = \emptyset$ by making a clause $O\gamma^t \rightarrow \neg f^{t+1}$ for every assignment set $\gamma \in \Gamma^t_\top$, such that $\gamma \models \texttt{DEL}(f)$ and $\gamma \not\models \texttt{ADD}(f)$. The explanatory frame axioms (Schema 4.2.2) ensure that if a fluent $f \in s^t_\top$ and $f \notin s^{t+1}_\top$, then for some $\gamma \in \Gamma^i_\top$, $\gamma \models \texttt{DEL}(f)$ due to the clause $(f^t \wedge \neg f^{t+1}) \rightarrow \bigvee_{\gamma' \in \texttt{break}(f)} \gamma'$. Finally, if fluent $f \notin s^t_\top$ and $f \in s^{t+1}_\top$, then for some $\gamma \in \Gamma^i_\top$, $\gamma \models \texttt{ADD}(f)$ due to the clause $(\neg f^t \wedge f^{t+1}) \rightarrow \bigvee_{\gamma' \in \texttt{make}(f)} \gamma'$.

It remains to show (ii), that every valid parallel plan π_h for Π_h can be translated into a satisfying valuation $\mathcal{V}$ of $\mathcal{T}^{Split||W}(\Pi_h)$. Following the logic of the above proof for (i), it is easy to see that the assignment $\mathcal{V}$ that represents the fluents $\mathcal{F}_\top$ and assignment sets $\Gamma_\top$ implied by the plan π_h satisfies the clauses in the encoding $\mathcal{T}^{Split||W}(\Pi_h)$. □

4.2.3 Action Variables

In the encodings for split action representations presented in the previous sections, action variables are omitted and replaced by assignment set variables and, possibly, a set of auxiliary variables. It was shown previously that we can use split action representations for parallel step-optimal planning, in particular precisely split action representations. It was also noted that for the parallel case there is no direct relationship between the number of steps in a plan and the number of actions. Using a precisely split representation, due to the problem of redundant actions it may not be possible to find a satisfying valuation which minimises a particular metric, such as action cost. This problem can be overcome by reintroducing action variables in addition to the assignment set variables used to encode the problem. The reintroduction of action variables seems contrary to the stated purpose of using a split action representation – i.e. to reduce the size of encodings of planning to SAT and to increase planning efficiency. But, if the assignment set variables are used to encode action mutex, frame axioms, and condition axioms, then the intent and major benefits of the original approach are maintained. In addition, the inclusion of action variables simplifies the encoding of intra-operator mutex constraints and of grounding support. An empirical evaluation that explores the benefits of these different approaches follows in Section 4.3.

This section will present a number of schemata that use action variables to replace those of the previous sections, for both the serial and the parallel cases. Formally, in following encodings, for each operator o we include a propositional variable a^t for each action $a \in \mathcal{A}_o^t$ and each step $t \in \{0, ..., h-1\}$. In the serial case, for the encoding presented in Section 4.2.1, we omit the grounding support clauses generated from Schema 4.2.7 and replace them by the following.

Schema 4.2.11. For each step $t \in \{0, ..., h-1\}$, each operator o with name O, each action $a \in \mathcal{A}_o^t$, and each assignment set $\gamma \in \Gamma_a$, we have a clause:

$$a^t \to O\gamma^t \qquad \square$$

Schema 4.2.12. For each step $t \in \{0, ..., h-1\}$, each operator o with name O, and each assignment set $\gamma \in \Gamma_o$ we have a clause:

$$O\gamma^t \to \bigvee_{a \in \mathcal{A}_o, \gamma \in \Gamma_a} a^t \qquad \square$$

For example, for the SIMPLE-LOGISTICS instance described in Table 4.3, for the assignment set variable $\texttt{Drive}\{?t \leftarrow T_1, ?from \leftarrow L_1\}^t$, we have the following clauses:

$$\texttt{Drive}\{?t \leftarrow T_1, ?from \leftarrow L_1\}^t \to (\texttt{Drive}(T_1, L_1, L_2)^t \vee \texttt{Drive}(T_1, L_1, L_3)^t)$$
$$\texttt{Drive}(T_1, L_1, L_2)^t \to \texttt{Drive}\{?t \leftarrow T_1, ?from \leftarrow L_1\}^t$$
$$\texttt{Drive}(T_1, L_1, L_3)^t \to \texttt{Drive}\{?t \leftarrow T_1, ?from \leftarrow L_1\}^t$$

In the parallel case, the grounding support constraints of Schemata 4.2.7 and 4.2.10 are replaced by the constraints just described for action variables (Schemata 4.2.11 and 4.2.12). Constraints between assignment sets that are subsets of one another are included (Schema 4.2.6). The other change to the parallel encoding is that auxiliary variables are no longer required to encode mutex relationships between pairs of actions which instantiate the same operator, and therefore constraints from Schema 4.2.9 are omitted. For any pair of mutex actions a_1 and a_2 that both instantiate an operator o and for which an instance of Schema 4.2.9 would be created, we instead assert that a_1 and a_2 are mutex directly at each step – i.e. we create a clause as described in Schema 3.3.6.

We let $\mathcal{T}^{Split||W\mathcal{A}}$ denote the encoding that results from applying the previous constraints to a problem Π using Θ^*_o and R^*_o for each operator o that define a precisely split encoding and including the constraints in this section to add action variables.

Theorem 4.2.3. *The encoding $\mathcal{T}^{Split||W\mathcal{A}}$ of a planning problem Π_h into SAT is constructive for a set of argument subsets Θ^*_o (Definition 4.1.23) and condition map $\mathcal{R}^*_o$ (Definition 4.1.24) for $o \in \mathcal{O}$ that define a precisely split encoding.*

Proof. This proof is based on Theorem 4.2.2 and explores how this encoding differs. Addressing condition (i), we sketch a plan extraction algorithm which takes a satisfying valuation $\mathcal{V}$ of $\mathcal{T}^{Split||W\mathcal{A}}(\Pi_h)$, and extracts a valid plan. To do this it reads off the set of true fluents $\mathcal{F}_\top := \langle \mathcal{F}^0_\top, ..., \mathcal{F}^h_\top \rangle$, true assignment set variables $\Gamma_\top := \langle \Gamma^0_\top, ..., \Gamma^{h-1}_\top \rangle$, and true actions $\mathcal{A}_\top := \mathcal{A}^0_\top, ..., \mathcal{A}^{h-1}_\top$. The actions $\mathcal{A}_\top$ represent a plan π_h, such that $\mathcal{F}_\top$ represents the sequence of states visited when π_h is realised. It is clear that this algorithm runs in linear time in the size of the set of true actions $\mathcal{A}_\top$.

Addressing conditions (i) and (ii), we note that $\mathcal{T}^{Split||W\mathcal{A}}$ differs from $\mathcal{T}^{Split||W}$ by omitting constraints for intra-operator mutex (Schema 4.2.9) and grounding support trees (Schemata 4.2.7 and 4.2.10). Instead $\mathcal{T}^{Split||W\mathcal{A}}$ rules out mutex actions a_1 and a_2, which instantiate the same operator (and would be ruled out explicitly by Schema 4.2.9), by creating a clause $\neg a^t_1 \vee \neg a^t_2$ according to Schema 3.3.6. Clauses generated from grounding support trees are replaced by clauses generated from Schemata 4.2.11 and 4.2.12. These clauses ensure that if an action $a \in \mathcal{A}^t_\top$, then $\Gamma_a \subseteq \Gamma^t_\top$. They also ensure that if an assignment set $\gamma \in \Gamma^t_\top$, then some action a, such that $\gamma \in \Gamma_a$, must be in $\mathcal{A}^t_\top$. □

4.3 Empirical Evaluation

We performed a detailed experimental comparison of our approaches with state-of-the-art encodings of planning-as-SAT using benchmarks from International Planning Competitions. We find that our precisely split encoding dominates in terms of the size of the CNFs it produces and exhibits better scalability than the state-of-the-art techniques. We found the parallel simply split encodings we introduced not to be competitive. We present a comparison of the following flat and split encodings.

- $\mathcal{T}^{SMP}$: This is the encoding of Sideris and Dimopoulos [67]. It includes noop actions and Schemata: 3.5.1, 3.5.2, 3.5.3, 3.5.4, 3.5.6, 3.5.7, 3.5.10, 3.3.10.

- $\mathcal{T}^{Pe}$: This is a simple flat encoding from Section 3.5. It includes Schemata: 3.5.1, 3.5.2, 3.5.3, 3.5.4, 3.5.6, 3.5.7, 3.3.9, 3.3.10.
- $\mathcal{T}^{Split||W}$: This is the standard precisely split encoding. It includes Schemata 3.5.1, 3.5.2, 3.5.7, 4.2.1, 4.2.2, 4.2.8, 4.2.9, 4.2.6, 4.2.10, 4.2.7.
- $\mathcal{T}^{Split||W\mathcal{A}}$: This is a precisely split encoding that omits the auxiliary conditions used to encode intra-operator mutex and instead includes action variables. It includes Schemata 3.5.1, 3.5.2, 3.5.7, 4.2.1, 4.2.2, 4.2.8, 4.2.6, 4.2.11, 4.2.12, 3.3.6.
- $\mathcal{T}^{Split||W\mathcal{A}x}$: This is a precisely split encoding that includes both the auxiliary conditions used to encode intra-operator mutex and action variables. It includes Schemata 3.5.1, 3.5.2, 3.5.7, 4.2.1, 4.2.2, 4.2.8, 4.2.9, 4.2.6, 4.2.11, 4.2.12.

Here the only encoding to include noop actions is $\mathcal{T}^{SMP}$. SATPLAN-06 [39] is not included in the experimental results as it is dominated by the flat encodings $\mathcal{T}^{SMP}$ and $\mathcal{T}^{Pe}$.

All encodings were generated using a single planning system written in C++. All generated plans were verified. For all encodings, all generated CNFs are solved using PRECOSAT [7] as the underlying satisfiability procedure. Preliminary experiments were also performed with RSAT [52]. RSAT and PRECOSAT exhibit similar patterns of performance, with PRECOSAT consistently performing better than RSAT. Therefore RSAT results are omitted from the following discussion. All experiments were run on a cluster of AMD Opteron 252 2.6GHz processors, each with 2GB of RAM.

We computed optimal relaxed plans for benchmark problems from a number of International Planning Competitions: IPC-6: ELEVATORS, PEG SOLITAIRE, and TRANSPORT; IPC-5: STORAGE, and TPP; IPC-4: PIPESWORLD; IPC-3: DEPOTS, DRIVERLOG, FREECELL, ROVERS, SATELLITE, and ZENOTRAVEL; and IPC-1: BLOCKS, GRIPPER, MICONIC, and MPRIME.

Table 4.4: This table presents the solving times for a number of flat and split SAT-based planners on IPC benchmark problems. h is the step-optimal horizon for each problem. Respectively, columns "Total", $h-1$, and h report the time in seconds that RSAT spent solving: all CNFs for a problem, the CNF at $h-1$, and the CNF at h. $\#a$ is #actions in the solution plan. For each problem, RSAT was timed out after 1800.

		$\mathcal{T}^{Pe}$				$\mathcal{T}^{SMP}$				$\mathcal{T}^{Split\|\|W}$				$\mathcal{T}^{Split\|\|W\mathcal{A}}$				$\mathcal{T}^{Split\|\|W\mathcal{A}x}$			
Problem	h	Total	$h-1$	h	$\#a$	Total	$h-1$	h	$\#a$	Total	$h-1$	h	$\#a$	Total	$h-1$	h	$\#a$	Total	$h-1$	h	$\#a$
BLOCKS-29	38	8.20	0.69	1.38	38	8.63	0.84	1.39	38	5.38	0.66	1.11	38	18.1	1.74	1.90	38	14.5	1.63	2.04	38
DEPOTS-09	23	854	352	463	70	705	329	337	73	1156	673	403	80	494	284	132	80	487	383	60.1	78
DEPOTS-11	20	263	222	11.5	62	272	222	21.9	68	795	691	35.6	66	349	289	16.5	58	421	352	26.0	61
DRIVERLOG-12	16	6.01	1.95	1.59	39	7.03	2.03	2.47	47	7.16	2.65	1.82	42	7.54	2.23	1.85	40	8.31	2.84	1.25	42
DRIVERLOG-15	11	3.67	1.06	2.25	50	3.85	1.09	2.41	51	3.91	0.92	2.30	54	5.52	1.53	3.31	53	4.31	1.66	1.47	55
ELEVATORS-STRIPS-16	15	4.82	1.07	1.85	48	5.26	1.06	2.01	53	17.4	5.10	6.30	46	5.55	1.25	2.03	45	5.63	1.22	1.82	47
ELEVATORS-STRIPS-19	13	5.71	1.50	3.07	38	5.30	1.13	2.99	38	6.83	1.14	4.04	39	5.95	1.48	3.26	38	6.26	1.70	3.04	39
ELEVATORS-STRIPS-29	11	3.13	1.73	0.97	42	4.28	2.08	1.72	42	6.90	3.58	2.31	39	6.03	1.85	3.31	41	4.18	2.10	1.38	40
FREECELL-02	8	0.87	0.27	0.50	21	0.91	0.28	0.53	17	0.73	0.28	0.37	18	0.92	0.39	0.47	17	0.80	0.28	0.45	18
FREECELL-04	13	23.1	4.38	15.4	32	23.0	4.38	15.4	31	8.30	1.45	5.59	31	20.6	3.34	14.0	33	12.7	2.90	7.71	33
GRIPPER-04	19	21.9	5.05	0.16	29	24.2	7.82	0.45	29	22.6	12.2	0.28	29	43.9	12.4	0.22	29	36.5	13.5	0.36	29
GRIPPER-05	23	465	153	0.20	35	394	172	0.69	35	393	105	0.91	35	677	202	1.00	35	640	184	1.25	35
GRIPPER-06	27	Time Out				5416	1206	2.36	41	Time Out				Time Out				Time Out			
LOGISTICS98-06	13	14.2	4.14	4.89	208	15.1	4.39	5.21	223	7.49	1.84	2.95	145	15.4	4.02	5.52	136	14.2	3.85	4.76	223
LOGISTICS98-15	15	8.24	2.03	2.48	151	9.88	2.28	3.48	158	10.4	2.71	4.43	175	9.37	2.39	3.36	147	10.2	2.42	2.99	161
MICONIC-12	10	0.04	0.01	0.01	11	0.04	0.01	0.01	11	0.03	0.01	0.01	11	0.04	0.01	0.03	11	0.07	0.01	0.06	11
MICONIC-13	8	0.02	0.01	0.01	10	0.02	0.01	0.01	10	0.01	0	0.01	10	0.02	0	0.01	10	0.03	0	0.02	10
MICONIC-14	9	0.03	0.01	0.01	10	0.03	0.01	0.01	10	0.04	0.01	0.03	10	0.02	0	0.01	10	0.03	0.01	0.02	10
MPRIME-08	5	15.8	0	15.8	40	7.38	0	7.38	47	3.21	0	3.21	9	8.40	0	8.40	10	8.00	0	8.00	9
MPRIME-19	6	32.3	0	32.3	39	15.3	0	15.3	53	6.59	0	6.59	26	19.3	0	19.3	24	16.2	0	16.2	14
PEGSOL-STRIPS-17	24	1513	1174	17.3	24	1286	959	12.4	24	805	536	33.2	24	607	406	25.0	24	640	445	17.4	24
PEGSOL-STRIPS-20	22	1300	986	14.5	22	1354	1015	14.1	22	1277	909	71.6	22	801	552	29.8	22	736	461	53.1	22
PEGSOL-STRIPS-21	23	Time Out				Time Out				Time Out				2528	1383	637	23	2883	1454	870	23
PEGSOL-STRIPS-23	24	Time Out				Time Out				3590	1342	1669	24	2880	1259	1245	24	Time Out			
PEGSOL-STRIPS-25	25	Time Out				Time Out				1947	1310	114	25	2219	1278	553	25	2026	922	734	25
PIPESWORLD-08	7	6.69	0.48	5.79	14	4.57	0.49	3.67	14	6.34	2.39	3.50	14	4.63	0.42	3.89	13	2.66	1.06	1.41	13
PIPESWORLD-11	12	3.17	0.52	2.11	22	3.37	1.01	1.78	22	8.76	1.97	5.37	22	5.14	1.82	2.67	22	3.94	1.18	2.19	22

continued...

Table 4.4: (continued)

		$\mathcal{T}^{Pe}$				$\mathcal{T}^{SMP}$				$\mathcal{T}^{Split\|\|W}$				$\mathcal{T}^{Split\|\|W\mathcal{A}}$				$\mathcal{T}^{Split\|\|W\mathcal{A}x}$			
Problem	h	Total	$h-1$	h	$\#a$	Total	$h-1$	h	$\#a$	Total	$h-1$	h	$\#a$	Total	$h-1$	h	$\#a$	Total	$h-1$	h	$\#a$
PIPESWORLD-15	20	Memory Out				Memory Out				Memory Out				Memory Out				661	209	279	30
ROVERS-08	9	11.6	10.6	0.23	40	6.78	4.75	1.00	37	0.50	0.16	0.15	33	1.93	0.71	0.58	40	1.41	0.46	0.38	36
ROVERS-10	12	Time Out				587	447	3.29	55	2.53	1.03	0.25	52	14.0	7.54	0.76	52	6.78	3.32	0.54	54
ROVERS-13	13	Time Out				Time Out				21.0	11.0	0.71	55	322	199	5.66	56	44.8	27.5	1.50	56
ROVERS-17	15	Time Out				Time Out				21.4	12.3	1.21	73	Time Out				1415	1226	1.72	77
ROVERS-26	15	Time Out				Time Out				272	183	8.78	94	3061	1407	330	107	676	495	14.9	114
SCANALYZER-STRIPS-06	12	20.4	13.5	1.07	36	18.4	12.4	1.29	32	6.68	2.66	0.96	34	6.73	2.82	1.13	22	4.48	1.99	0.61	24
SCANALYZER-STRIPS-08	10	66.5	48.3	4.26	40	59.0	41.2	4.22	38	19.1	12.1	1.80	38	24.1	10.3	3.91	28	15.9	10.2	1.28	24
SOKOBAN-STRIPS-05	25	20.2	4.65	2.43	25	20.6	4.10	3.04	25	374	119	62.3	25	129	27.9	17.9	25	161	41.2	11.6	25
SOKOBAN-STRIPS-06	35	774	177	121	35	777	147	191	35	1281	239	160	35	1298	295	293	35	1258	302	164	35
STORAGE-11	11	4.44	3.18	0.38	22	3.88	2.51	0.48	20	4.22	2.19	0.65	20	5.37	3.33	0.59	20	5.97	3.66	0.68	20
STORAGE-13	18	13.4	4.12	0.95	18	15.8	5.81	2.00	18	14.1	5.20	0.90	18	17.1	5.29	1.93	18	20.5	7.36	1.24	18
STORAGE-16	11	882	732	6.31	28	819	695	2.54	28	993	885	8.66	28	Time Out				Time Out			
TPP-14	10	5.26	3.46	1.49	79	6.49	4.29	1.76	78	4.34	2.57	1.38	74	4.47	2.59	1.32	76	5.29	3.66	0.99	75
TPP-21	12	Time Out				Time Out				Time Out				Time Out				1342	1325	16.1	140
TPP-23	11	6.30	0	6.30	158	4.56	0	4.56	150	36.0	0	36.0	143	11.0	0	11.0	137	4.58	0	4.58	133
TPP-24	10	5.35	1.75	3.60	144	5.60	1.67	3.93	141	13.5	1.41	12.1	135	11.0	3.39	7.57	138	5.02	1.73	3.29	132
TPP-25	11	54.5	9.01	45.1	186	Memory Out				Memory Out				Memory Out				Memory Out			
TRANSPORT-STRIPS-05	15	30.6	12.2	8.34	30	34.9	12.6	11.9	30	92.9	29.3	45.9	30	26.5	10.4	5.08	30	30.7	12.5	6.93	30
TRANSPORT-STRIPS-06	13	133	59.6	38.1	38	190	61.4	77.8	39	155	58.2	73.8	38	208	76.5	68.3	39	227	111	43.0	39
TRANSPORT-STRIPS-26	12	85.6	37.1	34.4	35	90.8	37.2	38.7	36	80.8	26.8	41.5	35	164	89.0	45.1	34	132	70.4	39.0	35
ZENOTRAVEL-13	7	2.41	0.66	1.33	40	1.97	0.58	1.04	35	0.44	0.15	0.18	40	1.51	0.48	0.66	40	1.44	0.48	0.60	38
ZENOTRAVEL-14	6	8.14	3.16	4.77	46	6.13	2.28	3.65	44	1.09	0.39	0.51	41	4.38	1.67	2.53	43	3.34	1.35	1.78	41

Table 4.5: This table presents the encoding sizes for flat and split SAT-based planners. h is the step-optimal horizon for each problem. Respectively, columns c and v give #clauses and #variables in the CNF at h. For each problem, RSAT was timed out after 1800.

		$\mathcal{T}^{Pe}$		$\mathcal{T}^{SMP}$		$\mathcal{T}^{Split\|\|W}$		$\mathcal{T}^{Split\|\|W\mathcal{A}}$		$\mathcal{T}^{Split\|\|W\mathcal{A}x}$	
Problem	h	$\#v$	$\#c$	$\#v$	$\#c$	$\#v$	$\#c$	$\#v$	$\#c$	$\#v$	$\#c$
TPP-14	10	8504	94.4K	10.9K	99.2K	16.2K	52.7K	12.6K	74.9K	14.1K	55.8K
TPP-21	12	Time Out		Time Out		Time Out		Time Out		55.8K	266K
TPP-23	11	37.5K	914K	43.8K	926K	59.3K	228K	49.3K	786K	53.7K	250K
TPP-24	10	30.4K	645K	36.2K	656K	52.0K	187K	40.6K	540K	44.5K	202K
TPP-25	11	47.8K	1.24M	Memory Out		Memory Out		Memory Out		Memory Out	
BLOCKS-WORLD-29	38	17.5K	656K	23.9K	669K	20.4K	188K	30.1K	518K	31.3K	220K
DEPOTS-09	23	33.5K	1.82M	41.3K	1.84M	93.4K	489K	52.3K	1.36M	54.8K	468K
DEPOTS-11	20	24.1K	722K	28.9K	732K	65.1K	305K	37.1K	496K	39.7K	292K
DRIVERLOG-12	16	12.1K	140K	14.0K	144K	19.1K	61.5K	17.8K	81.0K	18.5K	78.4K
DRIVERLOG-15	11	16.1K	232K	18.2K	236K	23.4K	82.0K	22.0K	114K	22.8K	105K
ELEVATORS-STRIPS-16	15	9032	148K	10.4K	151K	17.7K	69.0K	12.9K	90.6K	14.6K	72.4K
ELEVATORS-19	13	8048	162K	9392	165K	15.3K	66.3K	11.5K	97.4K	12.9K	69.5K
ELEVATORS-29	11	6852	128K	7965	130K	14.2K	55.6K	9771	77.6K	11.1K	57.1K
FREECELL-02	8	4782	249K	5306	250K	6162	37.3K	7153	56.5K	7573	40.9K
FREECELL-04	13	16.2K	1.55M	17.6K	1.55M	11.6K	120K	22.9K	274K	24.0K	140K
FTB-25	21	20.9K	66.5K	25.3K	75.3K	26.6K	61.0K	42.8K	104K	42.8K	104K
FTB-29	25	23.9K	72.8K	28.9K	82.8K	29.7K	67.5K	48.4K	116K	48.4K	116K
GRID-01	14	3676	75.5K	4810	77.8K	5035	33.4K	6279	41.7K	6367	42.0K
GRIPPER-04	19	2303	22.2K	3088	23.9K	2872	12.3K	3928	19.4K	4346	19.0K
GRIPPER-05	23	3341	34.7K	4476	37.0K	4124	19.0K	5680	29.9K	6278	28.8K
GRIPPER-06	27	Time Out		6120	53.9K	Time Out		Time Out		Time Out	
LOGISTICS98-06	13	51.7K	1.39M	62.1K	1.42M	59.6K	530K	79.0K	658K	81.6K	609K
LOGISTICS98-15	15	31.8K	434K	38.6K	448K	42.5K	191K	47.9K	238K	51.2K	236K
MICONIC-12	10	432	2364	529	2558	630	1634	705	2180	800	2352
MICONIC-13	8	336	1860	409	2006	478	1245	535	1657	610	1793
MICONIC-14	9	388	2124	475	2298	565	1447	633	1937	718	2091
MPRIME-08	5	14.4K	1.17M	15.1K	1.17M	40.5K	1.04M	16.6K	1.32M	26.1K	1.03M
MPRIME-19	6	26.9K	2.13M	28.3K	2.14M	75.3K	1.88M	31.3K	2.13M	46.8K	1.88M
PEGSOL-17	24	6141	130K	8395	135K	15.4K	80.3K	10.6K	109K	11.4K	68.9K
PEGSOL-20	22	5517	116K	7566	120K	13.7K	71.9K	9509	96.6K	10.2K	61.8K
PEGSOL-21	23	Time Out		Time Out		Time Out		9976	101K	10.7K	64.7K
PEGSOL-23	24	Time Out		Time Out		15.2K	79.3K	10.5K	107K	Time Out	
PEGSOL-25	25	Time Out		Time Out		15.7K	81.5K	10.8K	110K	11.6K	69.9K
PIPESWORLD-08	7	11.8K	969K	12.6K	971K	65.3K	265K	12.8K	682K	15.5K	180K
PIPESWORLD-11	12	10.5K	422K	12.1K	425K	46.3K	161K	14.7K	243K	15.5K	101K
PIPESWORLD-15	20	Memory Out		Memory Out		Memory Out		Memory Out		44.6K	307K
ROVERS-08	9	2887	78.2K	3675	80.8K	3785	16.4K	4589	39.8K	5249	22.1K
ROVERS-10	12	Time Out		6161	130K	6487	26.9K	7544	60.3K	8695	36.5K
ROVERS-13	13	Time Out		Time Out		12.8K	76.7K	13.2K	270K	15.5K	95.5K
ROVERS-17	15	Time Out		Time Out		21.1K	99.7K	Time Out		28.6K	139K
ROVERS-26	15	Time Out		Time Out		53.7K	551K	70.1K	5.10M	78.2K	668K
SCANALYZER-06	12	10.7K	256K	11.4K	258K	23.9K	73.9K	12.5K	215K	12.7K	61.1K
SCANALYZER-08	10	23.5K	1.04M	24.4K	1.04M	53.2K	169K	26.4K	755K	26.7K	135K
SOKOBAN-05	25	18.4K	657K	24.3K	669K	58.2K	398K	34.0K	403K	34.2K	378K
SOKOBAN-06	35	6313	85.7K	9054	91.2K	17.4K	72.0K	12.9K	72.6K	14.0K	71.8K
STORAGE-11	11	3890	57.4K	4826	59.3K	12.1K	36.8K	6050	42.8K	6480	37.7K
STORAGE-13	18	5829	109K	7924	114K	13.1K	70.0K	10.6K	78.3K	10.9K	73.4K
STORAGE-16	11	11.1K	246K	13.0K	249K	35.8K	116K	Time Out		Time Out	
TRANSPORT-05	15	17.1K	264K	18.7K	267K	32.0K	113K	21.1K	146K	21.6K	115K
TRANSPORT-06	13	25.1K	1.05M	26.9K	1.06M	51.4K	222K	29.9K	617K	31.9K	222K

Table 4.5: (continued)

		$\mathcal{T}^{Pe}$		$\mathcal{T}^{SMP}$		$\mathcal{T}^{Split\|\|W}$		$\mathcal{T}^{Split\|\|W\mathcal{A}}$		$\mathcal{T}^{Split\|\|W\mathcal{A}x}$	
Problem	h	$\#v$	$\#c$	$\#v$	$\#c$	$\#v$	$\#c$	$\#v$	$\#c$	$\#v$	$\#c$
TRANSPORT-26	12	$23.6K$	$990K$	$25.4K$	$994K$	$47.8K$	$207K$	$28.2K$	$580K$	$30.0K$	$208K$
ZENOTRAVEL-13	7	8293	$156K$	8891	$158K$	4459	$26.9K$	$10.3K$	$52.3K$	$10.7K$	$45.1K$
ZENOTRAVEL-14	6	$25.3K$	$591K$	$26.2K$	$595K$	8019	$73.2K$	$28.7K$	$203K$	$29.5K$	$136K$

Tables 4.4 and 4.5 and Figure 4.2 compare the previously described encodings when run with the ramp-up query strategy. For each domain, we include the hardest problem solved by each encoding. We measure problem hardness in terms of the amount of time it takes the solver to yield a solution. If multiple solvers find the same problem hardest, then we also include a row for the penultimate hardest for each solver. Here the reported time for each instance is the time taken to solve all of the generated SAT problems. The time taken to generate the SAT problems is omitted as, for hard problems, the generation time is insignificant for all approaches. This decision also removes the engineering of the CNF generator as a factor in the results.

Using the same experimental data as for Tables 4.4 and 4.5, Figure 4.2 plots the cumulative number of problems solved over time by each planning system, supposing invocations of the systems on problem instances were made in parallel. In Figure 4.2 we only included data for problems that take one of the planners over 3 seconds to solve.

Briefly summarising the results shown in Figure 4.2 and Table 4.4, the performance of all approaches is similar overall, with the split approaches scaling better than the flat approaches and solving more problems overall. The flat approaches are similar in size and performance, with $\mathcal{T}^{Pe}$ having slightly fewer clauses and variables than $\mathcal{T}^{SMP}$ due to it lacking variables for noop actions.

Summarising the results from Table 4.5, comparing the flat encodings $\mathcal{T}^{SMP}$ and $\mathcal{T}^{Pe}$ to the standard precise encoding $\mathcal{T}^{Split||W}$, except for the ZENOTRAVEL domain, where individual actions have an unusually small number of assignment sets, the flat encodings generally requires fewer variables than $\mathcal{T}^{Split||W}$ to encode a problem. This disparity is due to the large number of assignment set copies we require to soundly represent grounding support constraints. For example, in the PIPESWORLD domain assignment set copy variables accounted for as much as 93% of the total number of variables and in the DEPOTS they accounted for as much as 73%.

It should be noted that assignment set copy variables do not have a significant impact on the performance of DPLL procedures. That is, the variance in solution time required for the $\mathcal{T}^{Split||W}$

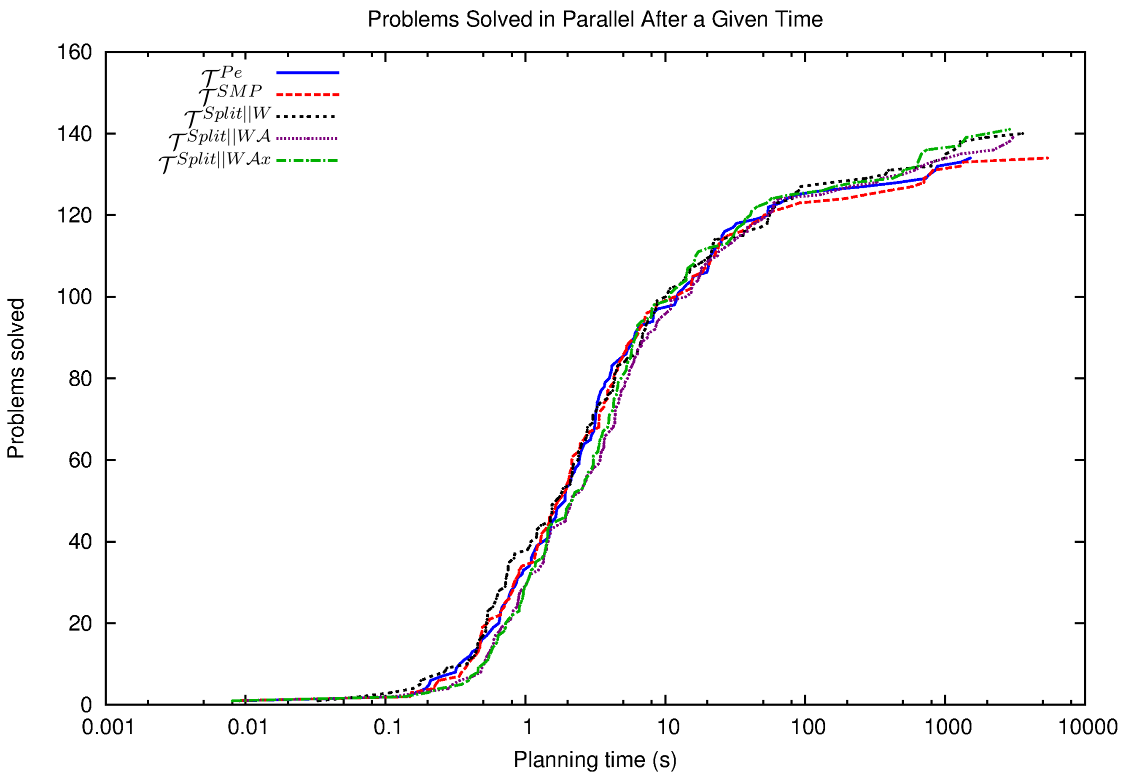


Figure 4.2: The number of "interesting" problems solved in parallel after a given planning time for a number of flat and split encodings. Interesting problems are those which take at least one approach at least 3 seconds.

encoding on different problems from the same domain seems unrelated to the percentage of variables that represent condition copies. Intuitively this is because a copy γ_n^* of an assignment set node γ_n represents the execution of at least one in a (typically) small set of actions whose precisely split representations include γ_n. More technically, γ_n^* implies the original node γ_n in Schema 4.2.10. Generally, due to mutex constraints between assignment sets, the majority of assignment set variables will be set false in any valuation explored by the SAT solver. Consequently, the majority of condition copies will be assigned false due to the application of unit propagation, a computationally cheap procedure. After the assignment set copies, the assignment set variables themselves require the most variables. The auxiliary assignment set variables introduced to efficiently encode mutex in SOLE typically accounted for about $1 - 10\%$ of the total number of variables.

Comparing the flat and precise in terms of compactness, $\mathcal{T}^{Split||W}$ usually requires far fewer clauses than the flat approaches, sometimes an order of magnitude fewer. The overwhelming majority of clauses generated by both the flat approaches represent action mutex. The relative compactness of $\mathcal{T}^{Split||W}$ follows because we factor mutex relations between actions. Indeed, the main benefit of our precisely split action representation is factoring. Because of this, compared to direct encodings $\mathcal{T}^{Split||W}$ uses significantly fewer preconditions and postcondition and mutex clauses, all expressed in 2-SAT, and usually vastly shorter successor state clauses – i.e., successor state clauses are expressed according to the small set of assignment sets (resp. large set of actions) that alter a fluents truth value. Quantifying this benefit, in problem SCANALYZER-08, $\mathcal{T}^{Pe}$ required about 882 thousand clauses to encode action mutex, whereas $\mathcal{T}^{Split||W}$ uses 1622 clauses to encode mutex (Schemata 4.2.8 and 4.2.8) and about 154 thousand clauses to encode grounding restriction (Schemata 4.2.6, 4.2.10, and 4.2.7).

The encodings $\mathcal{T}^{Split||W\mathcal{A}}$ and $\mathcal{T}^{Split||W\mathcal{A}x}$ both include action variables in addition to assignment set variables. They therefore use a simplified grounding support mechanism and avoid producing assignment set copy variables. In domains such as PIPESWORLD, in which assignment set copies constitute the overwhelming majority of the variables, the introduction of action variables can significantly reduce the size of the generated CNF. For example, on PIPESWORLD-11 $\mathcal{T}^{Split||W}$ requires about 161 thousand clauses while $\mathcal{T}^{Split||W\mathcal{A}x}$, which uses auxiliary variables to efficiently encode mutex, requires about 101 thousand. Typically though, in most domains the encoding of grounding support decision trees is more compact than the encodings that make use of action variables. This is due to the large number of binary clauses required to state that an action variable implies the assignment sets that are used to represent it. The encoding $\mathcal{T}^{Split||W\mathcal{A}}$ omits auxiliary assignment sets typically used to encode mutex between actions, and instead directly encodes these mutex relationships using the action variables, as in flat encodings. As expected, this encoding is typically less compact than then other split encodings, though due to the factoring of other planning constraints, it is still more compact than the flat encodings.

Despite requiring more variables, in general the split encodings outperform the flat encodings, with a split encoding being the most efficient (in terms of solution times) 68% of the time on hard problems (those in Table 4.4). Looking at individual encodings, the best performing split encoding was the $\mathcal{T}^{Split||W}$ encoding, which solved 20 problems from Table 4.4, while the best performing flat encoding was $\mathcal{T}^{Pe}$, which solved 13. As shown in Figure 4.2, the split approaches solve more “interesting” problems given our experimental time-out. The performance advantage of using a split encoding shown here is significantly less than was demonstrated in Robinson et al. [60] against SATPLAN-06. This is primarily due to the fact that, unlike the flat encodings used here, SATPLAN-06 does not include negative frame axioms (Schema 3.3.10) and includes a large number of redundant action mutex clauses. Negative frame axioms were included in our precisely split encodings [59, 60] and their importance was shown conclusively by Sideris and Dimopoulos [67].

In the domains where the split encodings significantly outperform the flat encodings, such as PEGSOL and ROVERS, the flat approaches tend to perform poorly when attempting refutation proofs, while the split approaches tend to perform somewhat better at this task. For example, on problem ROVERS-10 $\mathcal{T}^{SMP}$ takes 447 seconds to solve the SAT instance at $h-1$, while the worst of the split encodings, $\mathcal{T}^{Split||W\mathcal{A}}$ takes only 7.54 seconds. In the ROVERS domain, the grounding support constraints of $\mathcal{T}^{Split||W\mathcal{A}}$ require relatively few assignment set copies, typically less than 15% of propositional variables, and this encoding is the most efficient. In the PEGSOL domain, significantly more copies are required to represent grounding support (around 50% of variables) and in this case the encodings that include action variables ($\mathcal{T}^{Split||W\mathcal{A}}$ and $\mathcal{T}^{Split||W\mathcal{A}x}$) perform better. The only domain where the split approaches perform significantly worse than the flat approaches is the SOKOBAN domain. In this domain, relatively few clauses are required for the flat encoding of action mutex.

We believe that the performance advantages shown by the precisely split encodings, come not only from their compactness, but also from the factored conflict clauses learnt by PRECOSATwhen solving these instances. In particular, in the split encodings PRECOSAT learns conflicts between *assignment sets*, whereas in the flat encodings it has to learn conflict between *individual actions* in the flat encodings. Each conflict clause learned between propositions representing assignment sets represents a set of conflict clauses learned between actions. Due to this factoring, the decision procedure PRECOSAT is usually more efficient given precisely split encodings because: (1) we have relatively few mutex constraints, and (2) because PRECOSAT is required to learn and then exploit far fewer conflict clauses. This benefit is often offset by the need for complicated grounding support constraints. Finally, despite the problem of redundant actions highlighted in Section 4.1.4, when the precisely split encodings are solved, they produce plans with a comparable number of actions to the flat approaches.

4.4 Conclusion

In the spirit of leveraging advances in general-purpose automated reasoning in a planning setting, we have developed a framework to describe a range of split action representations and formally defined conditions under which these can be used for step-optimal serial and parallel planning. This framework is general enough to describe all current split action representations used for SAT-based planning. Furthermore, we have described a number of general compilations for a problem represented in this framework into propositional SAT formulae. In particular, we describe a novel *precisely split* action representation which is the first split action representation that can encode parallel step-optimal planning. This representation of actions results in more compact encodings of planning in SAT, and a more scalable and efficient SAT-based planning procedure over the state-of-the-art. We perform an experimental evaluation, and find that compactness chiefly derives from having a factored representation of mutex relations between actions. The gains in scalability and efficiency in precise splitting essentially follows from compactness and factoring. We also find that clause learning techniques implemented in modern decision procedures benefit from having factored representation of actions, resulting in further efficiency gains.

A pressing item for future work is to examine the benefits of our compact representation for optimal planning using more advanced query strategies [70, 55], and using a ramp-down query strategy in the CRICKET setting [53]. Additionally, we plan to explore the relationship between our encodings and new SAT encodings of planning such as SASE [30]. Finally, we plan to explore the space of sound encodings that are representable in the described encoding framework.

Chapter 5

Cost-Optimal SAT-Based Planning

In this chapter we consider the problem of computing optimal plans for propositional planning problems with action costs. Existing SAT-based planning procedures are limited to step-optimal and *fixed-horizon* settings. In the latter case, valid optimal solutions are constrained to be of, or less than, a fixed length. There has recently been significant work in fixed-horizon optimal SAT-based planning. For example, the work of Hoffman *et al.* [28] answers a key challenge from Kautz [33] by demonstrating how existing SAT-based planning techniques can be made effective solution procedures for fixed-horizon planning with metric resource constraints. In the same vein, Russell & Holden [63] and Giunchiglia & Maratea [23] develop optimal SAT-based procedures for *net-benefit* planning in fixed-horizon problems. Following that work, all of the planning techniques for cost-optimal planning presented in this chapter can straight-forwardly be extended to the net-benefit case.

The restriction of SAT-based techniques to the step-optimal and the fixed-horizon optimal cases limits their usefulness in practice. Optimal SAT-based planning procedures were unable to compete effectively in the deterministic track of the 2008 and 2011 International Planning Competitions due to the adoption of *horizon-independent* cost-optimality as the optimisation criteria. In the spirit of leveraging advances in general-purpose automated reasoning, we develop an approach for horizon-independent cost-optimal planning that operates by solving a sequence of Partial Weighted MaxSAT problems, each of which corresponds to a fixed-horizon variant of the problem at hand. Our approach is the first SAT-based approach in which a proof of horizon-independent cost-optimality is obtained using a MaxSAT procedure. It is also the first SAT-based planning system incorporating an admissible planning heuristic.

In the remainder of this chapter we will first describe Partial Weighted MaxSAT and present a new solver based on RSAT [52] that is tailored specifically for planning problems. We will then examine fixed-horizon cost-optimal SAT-based planning and how delete-relaxation planning heuristics can be computed with Partial Weighted MaxSAT. We present a novel method for finding horizon-independent cost-optimal plans based on Partial Weighted MaxSAT, which incorporates planning heuristics. Finally, we present a detailed empirical evaluation of our work on a number of benchmark problems.

5.1 Partial Weighted MaxSAT

The following definitions build upon the definition of SAT presented in Section 3.1.

5.1.1 MaxSAT

A Boolean MaxSAT problem is an optimisation problem related to SAT, typically expressed in CNF. However, in this case, the objective is to compute a valuation that maximises the number of satisfied clauses.

Definition 5.1.1. A Boolean MaxSAT problem $\langle V, \phi \rangle$ consists of the following:

- A set of n propositional variables $V := \{x_1, ..., x_n\}$; and
- A CNF propositional formula ϕ on the variables in V. □

Definition 5.1.2. For a valuation $\mathcal{V}$ over a set of variables V and a clause κ with variables in V, let $\mathcal{V} \models \kappa$ have numeric value 1 when valid, and 0 otherwise. □

Definition 5.1.3. For a MaxSAT problem $\langle V, \phi \rangle$, let the score of a valuation $\mathcal{V}$ over V be:

$$\omega(\mathcal{V}) := \sum_{\kappa \in \phi} (\mathcal{V} \models \kappa) \qquad \square$$

Definition 5.1.4. For a MaxSAT problem $\langle V, \phi \rangle$, a optimal valuation $\mathcal{V}^*$ over V has the property:

$$\mathcal{V}^* := \arg\max_{\mathcal{V}} \omega(\mathcal{V}) \qquad \square$$

5.1.2 Partial MaxSAT

The Partial MaxSAT problem is a variant of MaxSAT that distinguishes between *hard* and *soft* clauses. In these problems a solution is valid iff it satisfies all hard clauses. An optimal solution satisfies all hard clauses and maximises the number of satisfied soft clauses.

Definition 5.1.5. A Partial MaxSAT problem $\langle V, \phi := \{\phi^{hd} \cup \phi^{st}\}\rangle$ consists of the following:

- A set of n propositional variables $V := \{x_1, ..., x_n\}$;
- A CNF propositional formula ϕ on the variables in V, which consists of a set of hard clauses ϕ^{hd} and a set of soft clauses ϕ^{st}. □

Definition 5.1.6. A Partial MaxSAT problem $\langle V, \phi := \{\phi^{hd} \cup \phi^{st}\}\rangle$ is satisfiable iff there exists a valuation $\mathcal{V}$ over V, such that $\mathcal{V} \models \kappa$ for all hard clauses $\kappa \in \phi^{hd}$. □

Definition 5.1.7. For a Weighted MaxSAT problem $\langle V, \phi := \{\phi^{hd} \cup \phi^{st}\}\rangle$, and a satisfying valuation $\mathcal{V}$ over V, let the score of the valuation be:

$$\omega(\mathcal{V}) := \sum_{\kappa \in \phi^{st}} (\mathcal{V} \models \kappa)$$ □

Definition 5.1.8. An optimal solution $\mathcal{V}^*$ to a Partial MaxSAT problem $\langle V, \phi := \{\phi^{hd} \cup \phi^{st}\}\rangle$ must satisfy all hard clauses in ϕ^{hd} and additionally:

$$\mathcal{V}^* := \arg\max_{\mathcal{V}} \omega(\mathcal{V})$$ □

5.1.3 Weighted MaxSAT

A Weighted MaxSAT problem [2] is a MaxSAT problem where each clause has a bounded positive numerical weight. In this case an optimal solution must maximise the sum of weights of satisfied clauses.

Definition 5.1.9. A Weighted MaxSAT problem $\langle V, \phi\rangle$ consists of the following:

- A set of n propositional variables $V := \{x_1, ..., x_n\}$; and
- A CNF propositional formula ϕ on the variables in V, where each clause $\kappa \in \phi$ has a bounded positive numerical weight $\omega(\kappa)$. □

Definition 5.1.10. For a Weighted MaxSAT problem $\langle V, \phi \rangle$ and a valuation $\mathcal{V}$ over V, let the score of the valuation be:

$$\omega(\mathcal{V}) := \sum_{\kappa \in \phi} \omega(\kappa)(\mathcal{V} \models \kappa) \qquad \square$$

Definition 5.1.11. An optimal solution $\mathcal{V}^*$ to a Weighted MaxSAT problem $\langle V, \phi \rangle$ has the property:

$$\mathcal{V}^* := \arg\max_{\mathcal{V}} \omega(\mathcal{V}) \qquad \square$$

5.1.4 Partial Weighted MaxSAT

Partial Weighted MaxSAT [21] is a variant of Partial MaxSAT and Weighted MaxSAT. Like Partial MaxSAT a distinction is made between *hard* and *soft* clauses. Like Weighted MaxSAT soft clauses are given a numerical weight. In these problems a solution is valid iff it satisfies all hard clauses. An optimal solution maximises the sum of the weights of satisfied soft clauses.

Definition 5.1.12. A Partial Weighted MaxSAT problem $\langle V, \phi := \{\phi^{hd} \cup \phi^{st}\}\rangle$ consists of the following:

- A set of n propositional variables $V := \{x_1, ..., x_n\}$;
- A CNF propositional formula ϕ on the variables in V, which consists of a set of hard clauses ϕ^{hd} and a set of soft clauses ϕ^{st}, where each clause $\kappa \in \phi^{st}$ has a bounded positive numerical weight $\omega(\kappa)$. $\square$

Definition 5.1.13. An optimal solution $\mathcal{V}^*$ to a Partial Weighted MaxSAT problem $\langle V, \phi := \{\phi^{hd} \cup \phi^{st}\}\rangle$ must satisfy all hard clauses in ϕ^{hd} and additionally:

$$\mathcal{V}^* := \arg\max_{\mathcal{V}} \omega(\mathcal{V}) \qquad \square$$

5.1.5 A New Partial Weighted MaxSAT Solver

We found that branch-and-bound procedures for Partial Weighted MaxSAT [2, 21] are ineffective at solving our direct encodings of bounded planning problems, that we describe later. Taking RSAT [52] as a starting point, we developed PWM-RSAT, a more efficient optimisation procedure for this setting. An outline of the algorithm is given in Algorithm 2. PWM-RSAT can broadly be described as a backtracking search with Boolean unit propagation. It features common enhancements from state-of-the-art SAT solvers, including conflict driven *clause learning* with *non-chronological* backtracking [47, 43], and *restarts* [29].

Algorithm 2 shows the standard variant of PWM-RSAT that solves Partial Weighted MaxSAT problems $\langle V, \phi := \{\phi^{st} \cup \phi^{hd}\}\rangle$, where the soft clauses ϕ^{st} are restricted to unit clauses.[1] This restriction, in effect, means that weights are attached to propositional variables. Our encoding, described later, uses individual variables to represent the execution of actions. The cost of executing these actions can therefore be represented by the weights attached to variables.

PWM-RSAT works as follows: At the beginning of the search, the current partial valuation $\mathcal{V}$ of truth values to variables in V is set to empty and its associated cost c is set to 0. We use $\mathcal{V}^*$ to track the best full valuation found so far for the minimum cost of satisfying ϕ^{hd} given ϕ^{st}. $\hat{c}$ is the cost associated with $\mathcal{V}^*$. Initially, $\mathcal{V}^*$ is empty and $\hat{c}$ is set to an input non-negative weight bound $\hat{c}^I$ (if none is known then $\hat{c} \leftarrow \hat{c}^I \leftarrow \infty$). Note that the set of *asserting clauses* Γ is initiated to empty as no clauses have been learned yet. The solver then repeatedly tries to expand the partial valuation $\mathcal{V}$ until either the optimal solution is found or ϕ is proved unsatisfiable (line 4-21). At each iteration, a call to *SatUP*$(\mathcal{V}, \phi, \kappa)$ applies unit propagation to a unit clause $\kappa \in \phi$ and adds new variable assignments to $\mathcal{V}$. If κ is not a unit clause, *SatUP*$(\mathcal{V}, \phi, \kappa)$ returns 1 if κ is satisfied by $\mathcal{V}$, and 0 otherwise. The current cost c is also updated (line 5). If $c \geq \hat{c}$, then the solver will perform a backtrack-by-cost to a previous point where $c < \hat{c}$ (line 6-8).

During the search, if the current valuation $\mathcal{V}$ violates any clause in $(\phi^{hd} \wedge \Gamma)$, then the solver will either (i) restart if required (line 10), or (ii) try to learn the conflict (line 11) and then backtrack (line 12). If the backtracking causes all assignments in $\mathcal{V}$ to be undone, then the solver has successfully proved that either (i) $(\mathcal{V}^*, \hat{c})$ is the optimal solution, or (ii) ϕ is unsatisfiable if $\mathcal{V}^*$ remains empty (line 13-16). Otherwise, if $\mathcal{V}$ does not violate any clause in $(\phi^{hd} \wedge \Gamma)$ (line 17), then the solver will heuristically add a new variable assignment to $\mathcal{V}$ (line 22) and repeat the main loop (line 4). Note that if $\mathcal{V}$ is already complete, the better solution is stored in $\mathcal{V}^*$ together with the new lower cost $\hat{c}$ (line 19). The solver also performs a backtrack by cost (line 20) before trying to expand $\mathcal{V}$ in line 22.

[1] PWM-RSAT may be used to solve general Partial Weighted MaxSAT problems by amending the soft clauses with auxiliary variables that encode the clause weight. In more detail, for each non-unit soft clause $\kappa \in \phi^{st}$ create an auxiliary variable x_κ and add $\{\kappa \cup \neg x_\kappa\}$ to ϕ^{hd}. Finally, ϕ^{st} should consist of all original soft unit clauses and a soft unit clause for each auxiliary variable x_κ, with weight $\omega(\kappa)$.

Algorithm 2 Cost-Optimal RSat —- PWM-RSAT

```
 1: Input:
    - A given non-negative weight bound $\hat{c}^I$; If none is known then $\hat{c}^I := \infty$;
    - A CNF formula $\phi$ consists of the *hard* clause set $\phi^{hd}$ and the *soft* clause set $\phi^{st}$;
 2: $c \leftarrow 0$; $\hat{c} \leftarrow \hat{c}^I$;
 3: $\mathcal{V}, \mathcal{V}^* \leftarrow []$; $\Gamma \leftarrow \emptyset$;
 4: while $true$ do
 5:   $c \leftarrow \sum_{\kappa \in \phi^{st}} \omega(\kappa) \mathit{SatUP}(\mathcal{V}, \phi, \kappa)$;
 6:   if $c \geq \hat{c}$ then
 7:     Pop elements from $\mathcal{V}$ until $c < \hat{c}$; continue;
 8:   end if
 9:   if $\exists \kappa \in (\phi^{hd} \wedge \Gamma)$ s.t. $\neg \mathit{SatUP}(\mathcal{V}, \phi^{hd} \wedge \Gamma, \kappa)$ then
10:     if restart then $\mathcal{V} \leftarrow []$; continue;
11:     Learn clause with assertion level $m$; Add it to $\Gamma$;
12:     Pop elements from $\mathcal{V}$ until $|\mathcal{V}| = m$;
13:     if $\mathcal{V} = []$ then
14:       if $\mathcal{V}^* \neq []$ then return $\langle \mathcal{V}^*, \hat{c} \rangle$ as the solution;
15:       else return UNSATISFIABLE;
16:     end if
17:   else
18:     if $\mathcal{V}$ is a full assignment then
19:       $\mathcal{V}^* \leftarrow \mathcal{V}$; $\hat{c} \leftarrow c$;
20:       Pop elements from $\mathcal{V}$ until $c < \hat{c}$;
21:     end if
22:     Add a new variable assignment to $\mathcal{V}$;
23:   end if
24: end while
```

5.2 Fixed-Horizon Optimal Planning as Partial Weighted MaxSAT

In fixed-horizon optimal planning we aim to find an optimal plan for a bounded planning problem Π_h. We encode a bounded planning problem Π_h into Partial Weighted MaxSAT and find an optimal satisfying valuation, which corresponds to an optimal plan for Π_h. Here, we restrict ourselves to the cost-optimal case, but the techniques described can straight-forwardly be extended to the net-benefit case as in Russell and Holden [63] and Giunchiglia and Maratea [23]. In this case, we say that a plan π^{h*} is fixed-horizon cost-optimal for horizon h, iff there is no cheaper plan π', such that $h(\pi') \leq h(\pi^{h*})$. In practice, with the encoding described in the following section, we only need to prove that there is no cheaper plan at h and $h-1$.

5.2.1 Weighted MaxSAT Encoding of Cost-Optimal Planning

We define the encoding of the problem Π_h into Partial Weighted MaxSAT as follows.

Definition 5.2.1. Let $\mathcal{T}^c$ be a function which takes a bounded planning problem Π_h and returns a Partial Weighted MaxSAT problem. We say that $\mathcal{T}^c$ reduces Π_h to Partial Weighted MaxSAT and that $\phi := \mathcal{T}^c(\Pi_h)$ is an encoding of Π_h, iff a valid plan can be extracted from the optimal satisfying valuation $\mathcal{V}^*$ of ϕ and $\omega(\mathcal{V}^*)$ is the cost of the optimal plan. Finally we require that ϕ is unsatisfiable only if there is no such plan. $\square$

We now describe an encoding that is direct and constructive. The encoding uses the variables and constraints of the plangraph-based encoding of Section 3.5. In particular, there are the following propositional variables:

- For each step $t \in \{0, ..., h-1\}$ and each action $a \in \mathcal{A}^t$, we have a variable a^t; and
- For each step $t \in \{0, ..., h\}$ and each fluent $f \in \mathcal{F}^t$, we have a variable f^t.

We have the following hard constraints:

- Start state and goal constraints (Schemata 3.5.1 and 3.5.2);
- Preconditions (Schema 3.5.3);
- Postconditions (Schema 3.5.4);
- Weak action mutex (Schema 3.5.6);
- Fluent mutex (Schema 3.5.7); and
- Full explanatory frame axioms (Schemata 3.3.9 and 3.3.10).

Finally, we also have the following soft constraints:

Schema 5.2.1. Action cost axioms (soft): For each step $t \in \{0, ..., h-1\}$ and action variable a^t, such that $\mathcal{C}(a) > 0$, we have a soft unit clause $\kappa_i :=$

$$\neg a^t$$

with weight $\omega(\kappa_i) := \mathcal{C}(a)$. □

We now formally show the constructiveness of the encoding $\mathcal{T}^c$.

Theorem 5.2.1. *The parallel encoding $\mathcal{T}^c$ of a fixed-horizon planning problem Π_h into SAT is constructive.*

Proof. Addressing conditions (i) and (ii), we note that $\mathcal{T}^c$ is the encoding $\mathcal{T}^{Pe}$ with additional soft constraints that encode action costs. From Theorem 3.5.1 we have that every satisfying valuation of $\mathcal{T}^c$ represents a plan for Π_h and vice versa. Schema 5.2.1 ensures that a satisfying valuation $\mathcal{V}$ of $\mathcal{T}^c(\Pi_h)$ will have a weight $\omega(\mathcal{V})$ that is exactly the sum of the costs of the false actions in $\mathcal{V}$. The optimal satisfying valuation maximizes this weight and therefore minimises action cost. □

5.3 Computing Planning Heuristics with Partial Weighted MaxSAT

Heuristic state-space search is a common and powerful approach to optimal planning, and was used by the top three (non-baseline) entries in the optimal deterministic track of the 2008 International Planning Competition – i.e. GAMER [17] and two variants of HSP [26]. Heuristic search planners maintain a frontier of states which are expanded by selecting actions based on heuristic information about, for example, the cost of goal achievement. Generally, there is a trade-off between the informativeness of a heuristic and the cost of computing it. In optimal planning a more informative heuristic means less search nodes need to be expanded to find and prove an optimal solution. It is therefore of great interest to find new efficient methods of generating heuristic information. *Delete relaxation* heuristics are one widely used class of heuristics.

Definition 5.3.1. The *delete relaxation* Π^+ of planning problem Π is achieved by removing all delete effects from operators in Π. In particular, the set of operators $\mathcal{O}^+$ for Π^+ has an operator o^+ for each $o \in \mathcal{O}$ from Π, modified so that *post*$^-(o^+) := \emptyset$. The set of actions corresponding to groundings of this set is denoted $\mathcal{A}^+$ and individual actions in this set are denoted a^+. □

The relaxed problem has two key properties of interest here. First, the cost of an optimal plan from any reachable state in Π is greater than or equal to the cost of the optimal plan from that state in Π^+. Consequently relaxed planning can yield a useful admissible heuristic in search. For example, a best-first search such as A^* can be heuristically directed towards an optimal solution by using the costs of relaxed plans to arrange the priority queue of nodes to expand. Second, although it is NP-hard to solve the relaxed problem Π^+ in general [10], in practice it is often the case that optimal solutions to Π^+ are more easily computed than for Π.

5.3.1 Weighted MaxSAT Encoding of Relaxed Planning

We now describe a direct compilation of a relaxed planning problem Π^+ into a Partial Weighted MaxSAT problem. The cost of an optimal solution to this problem represents an admissible heuristic estimate of the cost of reaching the goal $\mathcal{G}$ from the start state s_0.

Definition 5.3.2. Let $\mathcal{T}^+$ be a function which takes a delete relaxation of a planning problem Π^+ and returns a Partial Weighted MaxSAT problem. We say that $\mathcal{T}^+$ reduces Π^+ to Partial Weighted MaxSAT and that $\phi^+ := \mathcal{T}^+(\Pi^+)$ is an encoding of Π^+ iff a valid relaxed plan can be extracted from each satisfying valuation of ϕ^+. Finally, we required that ϕ^+ is unsatisfiable only if there is no such relaxed plan. □

Our encoding of the delete relaxation is *causal*, in the sense developed by Kautz, et al. [34] for their ground parallel causal encoding of propositional planning into SAT. Our encoding has the following propositional variables. For each fluent $f \in \mathcal{F}$ and relaxed action $a^+ \in \mathcal{A}^+$ we have corresponding variables f^+ and a^+. For each $a^+ \in \mathcal{A}^+$ and $f \in add(a)$, such that $f \notin s_0$, we have a variable $add(a, f)^+$. A number of preprocessing techniques, such as the reachability analysis employed during plangraph generation, may be applied when populating the sets $\mathcal{F}$ and $\mathcal{A}^+$.

That f_i^+ is true in a valuation $\mathcal{V}$ intuitively means that $f_i \in \mathcal{G}$, or f_i^+ causally supports another fluent f_j^+ that is true in $\mathcal{V}$. That a^+ is true in $\mathcal{V}$ means that a is executed in the corresponding relaxed plan. That $add(a, f)^+$ is true in $\mathcal{V}$ means that a is the action considered to have added f.[2] Finally, we also require a set of causal link variables.

Definition 5.3.3. Let our set of causal link variables be defined by the set S^∞, recursively defined

[2]These variables are required because if multiple actions that add a single fluent are true in a relaxed plan, an encoding that fails to designate which of them add the fluent may be inconsistent. This was overlooked in Robinson, et al. [62].

by the two following definitions:

$$S^0 := \{\mathcal{K}(f_i, f_j)^+ | a^+ \in \mathcal{A}^+, f_i \in pre(a), f_j \in post^+(a_i)\}$$
$$S^{i+1} := S^i \cup \{\mathcal{K}(f_j, f_l)^+ | \mathcal{K}(f_j, f_k)^+, \mathcal{K}(f_k, f_l)^+ \in S^i\} \qquad \square$$

For each $\mathcal{K}(f_i, f_j)^+ \in S^\infty$ we have a corresponding propositional variable. Intuitively, if the variable $\mathcal{K}(f_i, f_j)^+$ is true in a valuation then f_i is the cause of f_j in the corresponding relaxed plan. We then have the following axiom schemata that use these variables.

Schema 5.3.1. Relaxed goal axioms: For each fluent $f \in \mathcal{G}$, where $f \notin s_0$, we assert that f is achieved using a relaxed action in $\mathcal{A}^+$. This is expressed with a unit clause:

$$f^+ \qquad \square$$

Schema 5.3.2. Relaxed fluent support axioms: For each fluent $f \in \mathcal{F}$, where $f \notin s_0$, there is the following clause:

$$f^+ \rightarrow (\bigvee_{a \in make(f)} add(a, f)^+) \qquad \square$$

Schema 5.3.3. Add variable axioms: For each variable $add(a, f)^+$, there are the following clauses:

$$add(a, f)^+ \rightarrow (a^+ \wedge f^+) \qquad \square$$

Schema 5.3.4. Add variable mutex axioms: For each pair of add variables $add(a_1, f)^+$ and $add(a_2, f)^+$, such that $a_1 \neq a_2$, there is the following clause:

$$\neg add(a_1, f)^+ \vee \neg add(a_2, f)^+ \qquad \square$$

Schema 5.3.5. Causal link axioms: For all fluents $f_i \in \mathcal{F}$, where $f_i \notin s_0$, all actions $a \in make(f_i)$, and all fluents $f_j \in pre(a)$, where $f_j \notin s_0$, there is the following clause:

$$add(a, f_i)^+ \rightarrow \mathcal{K}(f_j, f_i)^+ \qquad \square$$

Schema 5.3.6. Causality implies cause and effect axiom: For each causal link variable $\mathcal{K}(f_1, f_2)^+$, there is the following clause:

$$\mathcal{K}(f_1, f_2)^+ \rightarrow (f_1^+ \wedge f_2^+) \qquad \square$$

Schema 5.3.7. Causal transitive closure axioms: For each pair of causal link variables $\mathcal{K}(f_1, f_2)^+$ and $\mathcal{K}(f_2, f_3)^+$, there is the following clause:

$$(\mathcal{K}(f_1, f_2)^+ \wedge \mathcal{K}(f_2, f_3)^+) \rightarrow \mathcal{K}(f_1, f_3)^+ \qquad \square$$

Schema 5.3.8. Causal anti-reflexivity axioms: For each causal link variable of the form $\mathcal{K}(f, f)^+$, there is the following clause:

$$\neg\mathcal{K}(f, f)^+ \qquad \square$$

Together, the previous two schemata produce constraints which imply that causal support is anti-symmetric. Finally, we have a set of soft constraints to encode action costs.

Schema 5.3.9. Relaxed action cost axioms (soft): For each relaxed action $a^+ \in \mathcal{A}^+$, such that $\mathcal{C}(a) > 0$, there is a unit clause $\kappa_i :=$

$$\neg a^+$$

and have $\omega(\kappa_i) = \mathcal{C}(a)$. $\square$

The schemata we have given thus far are theoretically sufficient for our purpose. However, we introduce a number of optional schemata which have been shown to improve the efficiency of solving these Partial Weighted MaxSAT problems. First, we can rule out some *redundant* actions.

Definition 5.3.4. For a relaxed planning problem Π^+, relaxed action $a_i^+ \in \mathcal{A}^+$ is *redundant* in an optimal solution, if there exists a relaxed action $a_j^+ \in \mathcal{A}^+$ such that:

1. $cost(a_j) \leq cost(a_i)$;
2. $pre(a_j) \backslash s_0 \subseteq pre(a_i) \backslash s_0$; and
3. $post^+(a_i) \backslash s_0 \subseteq post^+(a_j) \backslash s_0$. $\square$

We then use the following schema.

Schema 5.3.10. Relaxed action cost dominance axioms: For each relaxed action $a^+ \in \mathcal{A}^+$ that is *redundant*, there is the following clause:

$$\neg a^+ \qquad \square$$

In a causal encoding of a delete relaxation of a planning problem, most causal links are not relevant to the relaxed cost of reaching the goal from a particular state. For example, in a SIMPLE-LOGISTICS problem, if a truck T at location L_1 needs to be moved directly to location L_2, then the fact that the truck is at any other location should not support it being at L_2 – i.e. $\neg\mathcal{K}(\text{at}(T, L_3), \text{at}(T, L_2))^+$, where $L_3 \neq L_1$.

In an attempt to restrict sub-optimal options for causal support, we introduce schemata that provide a number of *layers* that actions and fluents in the relaxed suffix can be assigned to. Fluents and actions are forced to occur as early in the set of layers as possible and are only assigned to a layer if all supporting actions and fluents occur at earlier layers. The orderings of fluents in the relaxed layers is used only to restrict the truth values of the causal link variables. The admissibility of the heuristic estimate of the relaxed suffix is therefore independent of the number of relaxed layers.

We pick a horizon $k > 0$ and generate a copy a^{+l} of each relaxed action $a^+ \in \mathcal{A}^+$ at each layer $l \in \{0, ..., k-1\}$ and a copy f^{+l} of each fluent $f \in \mathcal{F}$ at each layer $l \in \{0, ..., k\}$, where $f \notin s_0$.[3] We also have an auxiliary variable $aux(f^{+l})$ for each fluent f^{+l} at each layer $\{1, ..., k\}$. Intuitively, $aux(f^{+l})$ says that f is false at every layer in the relaxed suffix from 0 to l. We have the following schemata to encode these layers.

Schema 5.3.11. Layered relaxed action axioms: For each layered relaxed action variable a^{+l}, there is the following clause:

$$a^{+l} \rightarrow a^+ \qquad \square$$

Schema 5.3.12. Layered relaxed actions only once axioms: For each relaxed action $a^+ \in \mathcal{A}^+$ and pair of layers $\{l_1, l_2\} \subseteq \{0, ..., k-1\} \times \{0, ..., k-1\}$, where $l_1 \neq l_2$, we have:

$$\neg a^{+l_1} \vee \neg a^{+l_2} \qquad \square$$

Schema 5.3.13. Layered relaxed action precondition axioms: For each layered relaxed action variable a^{+l_1} and each fluent $f \in \mathit{pre}(a)$, such that $f \notin s_0$, we have a clause:

$$a^{+l_1} \rightarrow \bigvee_{l_2 \in \{0, ..., l_1\}} f^{+l_2} \qquad \square$$

[3]There are no cost constraints associated with the layered copies of relaxed action variables.

Schema 5.3.14. Layered relaxed action effect axioms: For each layered relaxed action variable a^{+l_1} and each fluent $f \in \mathit{post}^+(a)$, such that $f \notin s_0$, there is a clause:

$$(a^{+l_1} \wedge f^+) \rightarrow \bigvee_{l_2 \in \{1,...,l+1\}} f^{+l_2} \qquad \square$$

Schema 5.3.15. Layered relaxed action as early as possible axioms: For each layered relaxed action variable a^{+l_1}, there is the following clause:

$$a^+ \rightarrow (\bigvee_{l_2 \in \{0,...,l_1\}} a^{+l_2}) \vee (\bigvee_{f \in \mathit{pre}(a), f \notin s_0} aux(f^{+l_1})) \qquad \square$$

Schema 5.3.16. Auxiliary variable axioms: For each auxiliary variable $aux(f^{+l_1})$, there is a set of clauses:

$$aux(f^{+l_1}) \longleftrightarrow \bigwedge_{l_2 \in \{0,...,l_1\}} \neg f^{+l_2} \qquad \square$$

Schema 5.3.17. Layered fluent axioms: For each layered fluent variable f^{+l}, there is a clause:

$$f^{+l} \rightarrow f^+ \qquad \square$$

Schema 5.3.18. Layered fluent frame axioms: For each layered fluent variable f^{+l}, there is a clause:

$$f^{+l} \rightarrow \bigvee_{a \in make(f)} a^{+l-1} \qquad \square$$

Schema 5.3.19. Layered fluent as early as possible axioms: For each layered fluent variable f^{+l_1}, there is a set of clauses:

$$f^{+l_1} \rightarrow \bigwedge_{a \in make(f)} \bigwedge_{l_2 \in \{0,...,l_1-2\}} \neg a^{+l_2} \qquad \square$$

Schema 5.3.20. Layered fluent only once axioms: For each fluent $f \in \mathcal{F}$, where $f \notin s_0$, and pair of layers $\{l_1, l_2\} \subseteq \{1, ..., k\} \times \{1, ..., k\}$, where $l_1 \neq l_2$, there is a clause:

$$\neg f^{+l_1} \vee \neg f^{+l_2} \qquad \square$$

Schema 5.3.21. Layered fluents prohibit causal links axioms: For each layered fluent variable $f_1^{+l_1}$ and fluent $f_2 \in \mathcal{F}$, where $f_2 \notin s_0$, such that $f_1 \neq f_2$ and there exists a causal link variable $\mathcal{K}(f_2, f_1)^+$, there is a clause:

$$f_1^{+l_1} \rightarrow (\bigvee_{l_2 \in \{1,...,l-1\}} f_2^{+l_2}) \vee \neg \mathcal{K}(f_2, f_1)^+ \qquad \square$$

In the following, let $\mathcal{T}^+$ refer to the above encoding without the optional schemata – i.e. without Schemata 5.3.10 - 5.3.21. Let $\mathcal{T}^{+R}$ refer to $\mathcal{T}^+$ with the addition of constraints that prohibit redundant actions – i.e. including Schema 5.3.10. Let $\mathcal{T}^{+L}$ refer to $\mathcal{T}^+$ with the addition of constraints that represent relaxed layers – i.e. Schemata 5.3.11 - 5.3.21. Finally, let $\mathcal{T}^{+RL}$ refer to $\mathcal{T}^+$ with the addition of constraints both for prohibiting redundant actions and for representing relaxed layers.

We now formally show the constructiveness of the encodings $\mathcal{T}^+$ and $\mathcal{T}^{+R}$ and discuss the constructiveness of $\mathcal{T}^{+L}$ and $\mathcal{T}^{+RL}$.

Theorem 5.3.1. *The encoding $\mathcal{T}^+$ of a relaxed planning problem Π^+ into SAT is constructive.*

Proof. Addressing condition (i), we sketch a plan extraction algorithm which takes an optimal satisfying valuation $\mathcal{V}$ of $\mathcal{T}^+(\Pi^+)$, and extracts a valid plan. A $\mathcal{V}$ specifies a set of true fluent variables $\mathcal{F}_\top$, a set of true action variables $\mathcal{A}_\top$, a set of true add variables $Add_\top$, and a set of true causal link variables $\mathcal{K}_\top$. Our algorithm produces a plan π^+ that consists of a sequence of sets of actions $\langle \mathcal{A}^0_\top, ..., \mathcal{A}^{h-1}_\top \rangle$. It proceeds by building a directed acyclic graph where nodes are actions and edges represent causal support. For each add relation $add(a_1, f) \in Add_\top$, where there is a distinct action a_2 with $f \in pre(a_2)$, we add an edge from a_1 to a_2. The graph is acyclic. This can be observed because a cyclic graph would violate the clauses generated by $\mathcal{T}^+$. Let $Add_\top$ have $add(a_1, f_1)$, $add(a_2, f_2)$, and $add(a_3, f_3)$ and there be the precondition relations $f_1 \in pre(a_2)$, $f_2 \in pre(a_3)$, and $f_3 \in pre(a_1)$. By Schema 5.3.3, we have $\{f_1, f_2, f_3\} \subseteq \mathcal{F}_\top$ and $\{a_1, a_2, a_3\} \subseteq \mathcal{A}_\top$. By Schema 5.3.5, we have $\{\mathcal{K}(f_3, f_1), \mathcal{K}(f_1, f_2), \mathcal{K}(f_2, f_3)\} \subseteq \mathcal{K}_\top$. By Schema 5.3.7, we have that $\{\mathcal{K}(f_3, f_2), \mathcal{K}(f_3, f_3)\} \subseteq \mathcal{K}_\top$. By Schema 5.3.8, we have a clause $\neg\mathcal{K}(f_3, f_3)^+$ and therefore a contradiction.

Now that we have a directed acyclic graph that encodes causal support between actions, we annotate nodes with layers. If a node has no incoming arcs then it is labelled with layer 0. Other nodes are labelled with a number corresponding to the distance of the longest path on an incoming edge from a node labelled with 0. As the graph is acyclic this procedure terminates and labels all nodes. From this labelled graph we build the sets $\pi^+ := \langle \mathcal{A}^0_\top, ..., \mathcal{A}^{h-1}_\top \rangle$, where actions are put into the set that corresponds to their label and h is determined to be $1+$ the largest such label. It is clear that such an algorithm runs in polynomial time in the size of $\mathcal{A}$.

We show that π^+ is a valid plan for Π^+ by demonstrating that:

1. The goal is supported by the execution of actions in π^+. First, Schema 5.3.1 ensures that

$\mathcal{F}_\top$ contains all goal fluents and Schemata 5.3.2 and 5.3.3 ensure that actions executed add these fluents; and

2. Every action in π^+ has preconditions that are supported by actions at an earlier step in π^+ or by the start state. Schemata 5.3.5 and 5.3.6 ensure that if an action a_1 adds a fluent f_1 and has a precondition f_2, then $f_2 \in \mathcal{F}_\top$ and $\mathcal{K}(f_1, f_2) \in \mathcal{K}_\top$. Because we now have $f_2 \in \mathcal{F}_\top$, by Schema 5.3.2, we will require an action a_3 to add f_2 unless $f_2 \in s_0$. As we have shown earlier that support between actions and fluents is acyclic, a_3 will be labelled with an earlier layer than a_2.

It remains to show (ii), that every valid relaxed plan π^+ for Π^+ can be translated into a satisfying valuation $\mathcal{V}$ of $\mathcal{T}^+(\Pi^+)$. Following the logic of the above proof for (i), it is easy to see that the valuation $\mathcal{V}$ that represents the fluents $\mathcal{F}_\top$, actions $a_\top$, add variables $Add_\top$, and causal link variables $\mathcal{K}_\top$ implied by the plan π^+ satisfies the clauses in the encoding $\mathcal{T}^+(\Pi^+)$. □

Theorem 5.3.2. *The encoding $\mathcal{T}^{+R}$ of a relaxed planning problem Π^+ into SAT is constructive.*

Proof. Addressing conditions (i) and (ii), we note that $\mathcal{T}^{+R}$ is equal to $\mathcal{T}^+$ with the addition of Schema 5.3.10, which prevents a number of actions that are redundant from appearing in any satisfying valuation $\mathcal{V}$ of $\mathcal{T}^{+R}(\Pi^h)$. By Definition 5.3.4, redundant actions are never needed in any optimal plan. we therefore have from Theorem 5.3.1 that $\mathcal{T}^{+R}$ is constructive. □

We do not formally show the constructiveness of the encodings $\mathcal{T}^{+L}$ and $\mathcal{T}^{+RL}$ as these encodings do not prove to be empirically useful in the following section. We do however note that the clauses from Schemata 5.3.11 - 5.3.21 for the relaxed layers only interact with the underlying encoding via Schema 5.3.21, which prohibits a causal link $\mathcal{K}(f_1, f_2)$ in the case that f_2 is put into a relaxed layer before f_1. Actions and fluents need to occur in the relaxed layers as early as possible, but only if there is enough layers. As such these constraints do not rule out any valid plans in the underlying encoding and can easily be shown to be constructive.

5.3.2 Empirical Evaluation

We implemented a translator in C++ for the encodings $\mathcal{T}^+$, $\mathcal{T}^{+R}$, $\mathcal{T}^{+L}$, and $\mathcal{T}^{+RL}$, described above. The translator first builds a plangraph representation of the problem to perform reachability and neededness pruning. It then generates Partial Weighted MaxSAT problems using the sets of actions and fluents reachable at a stable plangraph layer. The generated Partial Weighted

MaxSAT problems were solved with PWM-RSAT. We computed optimal relaxed plans for benchmark problems from a number of International Planning Competitions: IPC-6: ELEVATORS, PEG SOITAIRE, and TRANSPORT; IPC-5: STORAGE, and TPP; IPC-4: PIPESWORLD; IPC-3: DEPOTS, DRIVERLOG, FREECELL, ROVERS, SATELLITE, and ZENOTRAVEL; and IPC-1: BLOCKS, GRIPPER, MICONIC, and MPRIME. We also developed our own domain, called FTB, that demonstrates the effectiveness of the factored problem representations employed by SAT-based systems such as our solver COS-P, described in Section 5.4.1. This domain has the following important properties: (1) it has exponentially many states in the number of problem objects, (2) if there are h objects, then the branching factor is such that a breadth-first search encounters all the states at depth h, and (3) all plans have length h, and plan optimality is determined by the first and last actions (only) of the plan. This domain cripples state-based systems such as BASELINE and GAMER.

We compute relaxed plans from the start states of the test problems. The performance exhibited in this task should be indicative of the performance exhibited at other reachable states of the problem. We compare our approach to BASELINE, the de facto winning entry at the 2008 IPC. BASELINE is an A* search [25] (a best first search) with a constant heuristic function that uses the search code of the LAMA [54] planner.

Experiments were run on a cluster of AMD Opteron 252 2.6GHz processors, each with 2GB of RAM. All plans computed were verified as correct and computed within a timeout of 30 minutes. The times reported for BASELINE represent the time required to find a plan as reported by the planner. The time reported for each SAT-based approach represents the time reported by PWM-RSAT required to solve the generated Partial Weighted MaxSAT problem. Except in the smallest instances, the time required to generate the SAT encodings was insignificant compared to the time taken by PWM-RSAT to solve them.

The results of our experiments are summarised in Table 5.1. This table includes the hardest problem solved by each approach for each domain. The results are also summarised in Figure 5.1, which shows the number of problems solved in parallel by each approach after a given planning time. The first thing to note about the results is that the SAT-based approaches significantly outperform BASELINE, solving far more instances overall. The SAT-based approaches outperform BASELINE on all domains except the FREECELL, GRID, and PIPESWORLD domains. These are domains that are typically hard for SAT-based planning systems and that tend to require many clauses and variables to encode causal support. For example, the problem GRID-1 has more than

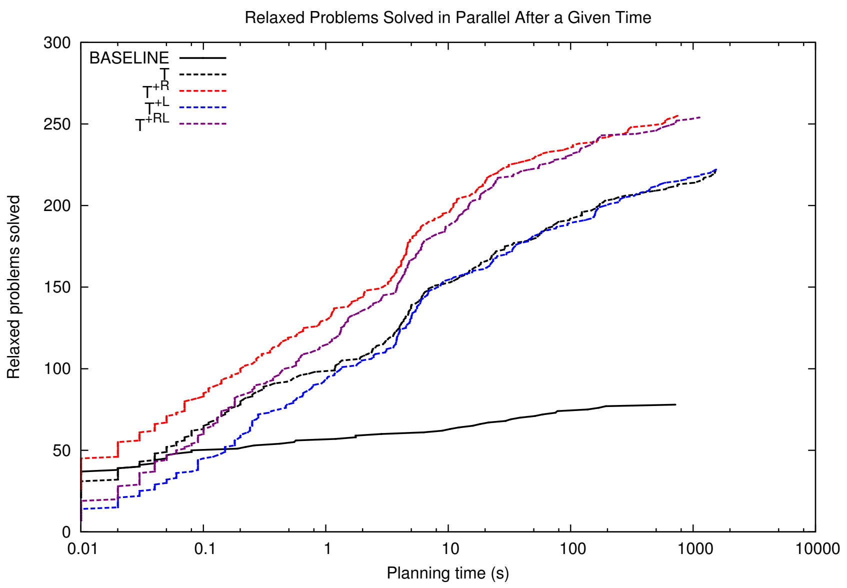


Figure 5.1: The number of relaxed problems solved optimally in parallel after a given planning time for BASELINE and Partial Weighted MaxSAT-based planners using the encodings $\mathcal{T}^{+}$, $\mathcal{T}^{+R}$, $\mathcal{T}^{+L}$, and $\mathcal{T}^{+RL}$. Problems were solved with a time-out of 30 minutes and $2GB$ of RAM.

76 thousand support variables and requires more than 20 million clauses to represent causal transitive closure. While complex encodings of causal support make planning difficult in the SAT-based cases, compact encodings of causal support are no guarantee that these approaches will succeed. For example, while the problem MICONIC-29 requires only 294 support variables and 3528 transitive closure clauses it cannot be solved within the timeout by the SAT-based approach with encoding $\mathcal{T}^{+}$. Though in this case, the addition of clauses to rule out redundant actions (as appear in encodings $\mathcal{T}^{+R}$ and $\mathcal{T}^{+RL}$) allows the problem to be solved in a small fraction of a second.

The SAT-based approach that uses the encoding $\mathcal{T}^{+R}$ is the most efficient overall. The approach that also includes relaxed layers $\mathcal{T}^{+RL}$ also performs well, but as the results in table Table 5.1 indicate, it is often not worth including the relaxed layers. In some problems most actions are redundant. For example, in the problem MICONIC-29 121 out of 144 (84%) of actions are redundant. Even in problems where a relatively small percentage of actions are redundant, removing the redundant actions can have a big effect on performance. For example, in the problem TRANSPORT-14 only 15% of actions are redundant, yet removing them improves the performance of the planner by two orders of magnitude. The effectiveness of removing redundant actions suggests that this approach could be further enhanced by improving the redundancy checking mechanism. In particular, actions may be found to be redundant if a sequence of other actions can more cheaply achieve the same effects.

Finally, the effectiveness of the SAT-based approaches at solving the relaxed planning problems versus BASELINE is not necessarily indicative of the superiority of SAT-based techniques for solving this problem. The state-spaces of relaxed problems have a very high branching factor which poses a problem for the BASELINE, which lacks an effective heuristic. A state-based approach with a better heuristic would likely perform better. By the same token, there are likely to be many further enhancements that can be made to the SAT-based approaches and we plan to examine these in the future.

5.4 Horizon-Independent Cost-Optimality

Existing SAT-based approaches to cost-optimal planning are limited to the fixed-horizon case. In this case, a plan Π^{h*} is optimal for horizon h iff there is no cheaper plan at any horizon $h' \leq h$. In general, that a plan π^{*h} is fixed-horizon optimal for horizon h does not preclude there being a cheaper plan at a horizon greater than h. For example, in the SIMPLE-LOGISTICS problem shown in Figure 2.1, let the following actions have the costs indicated:

$$\begin{aligned}
\mathcal{C}(\mathit{Drive}(T, L_1, L_2)) &:= x \\
\mathcal{C}(\mathit{Drive}(T, L_1, L_3)) &:= 3x \\
\mathcal{C}(\mathit{Drive}(T, L_2, L_3)) &:= x
\end{aligned}$$

If a planning problem requires truck T to drive from L_1 to L_3, then there is a plan with horizon 1 that achieves the goal with cost $3x$:

$$0 : \mathit{Drive}(T, L_1, L_3)$$

If the horizon is increased to 2 then there is a plan that achieves the goal with cost $2x$:

$$\begin{aligned}
0 &: \mathit{Drive}(T, L_1, L_2) \\
1 &: \mathit{Drive}(T, L_2, L_3)
\end{aligned}$$

Step-optimality and fixed-horizon-optimality are often not suitable for use as practical definitions of optimality because the number of steps in a plan is, in general, not linked to any meaningful aspect of plan quality, such as execution time, cost, or benefit. This fact is reflected in the choice of optimality criteria at the 2008 and 2011 International Planning Competitions. For the optimal deterministic track this was horizon-independent cost-optimality.

Definition 5.4.1. A plan π^* for a problem Π is *horizon-independent cost-optimal* if there is no cheaper plan at any horizon $h \in \{1, 2, ...\}$. □

In general, for a horizon h, if we have a plan π^{h*} that is the cheapest plan at any horizon $\{0, ..., h\}$, then to prove that π^{h*} is horizon-independent cost-optimal, we need to prove that there is no horizon $h' > h$ such that there is a plan $\pi^{h'}$, where $\mathcal{C}(\pi^{h'}) < \mathcal{C}(\pi^{h*})$. As it is impossible to prove that there is no plan cheaper than π^{h*} at every horizon, an upper bound on the horizon is required. A very weak bound can be obtained by assuming that a plan never visits the same state twice. If $\mathcal{S}$ is the set of states that can be formed from the fluents in $\mathcal{F}$, then we have a maximum upper bound of $|\mathcal{S}| - 1$.

Improving this bound, if we have $\mathcal{C}(\pi^{h*})$ and a is the cheapest action in $\mathcal{A}$, then we know that a plan cheaper than the optimal must take less than $\lfloor \mathcal{C}(\pi^{h*})/\mathcal{C}(a) \rfloor$ steps. This bound may be improved even further with additional reasoning, in particular by detecting that certain cheap actions always occur as a part of planning macros – i.e. always occur in conjunction with other actions. This is a promising area for future research. In general, where there is a large difference between the cost of the most and least expensive action, determining such a bound becomes difficult without sophisticated reasoning. We therefore explore a number of SAT-based horizon-independent cost-optimal planning methods that do not require determining a horizon bound *a priori*.

5.4.1 Horizon-Independent Cost-Optimal Planning System

We now describe COS-P, our sound and complete planning system that finds horizon-independent cost-optimal plans by alternately solving two variants of h-step-bounded instances of a planning problem for successively larger horizons h. Solutions to the intermediate step-bounded instances are obtained by compiling them into equivalent Partial Weighted MaxSAT problems, and then using a modified version of our own solver PWM-RSAT (Section 5.1.5) to compute their optimal solutions.

The two variants are characterised in terms of their optimal solutions. For a bounded problem Π_h, VARIANT-I admits optimal solutions that correspond to cost-optimal fixed-horizon parallel plans Π_h. VARIANT-II admits optimal plans with a prefix corresponding to h *sets* of actions from Π_h – i.e., an h-step plan prefix in the parallel format – and an arbitrary length suffix (including length 0) comprised of actions from the delete relaxation Π^+.

VARIANT-II instances are solved with a modified version of PWM-RSAT. In particular, the following lines are inserted between lines 4 and 5 of Algorithm 2.

```
1: if solving Variant-II && duplicating-layers($\mathcal{V}$) then
2:     pop elements from $\mathcal{V}$ until $\neg$duplicating-layers($\mathcal{V}$); continue;
3: end if
```

These lines prevent PWM-RSAT from exploring valuations implying that the same state occurs at more than one planning layer in the plan prefix, that is in layers $\{0, ..., h\}$.

Both variants can be categorised as *direct*, and *constructive*. They are *constructive* in the sense that every plan with cost $\mathcal{C}$ in the planning problem has a corresponding satisfying model with cost $\mathcal{C}$ in the Partial Weighted MaxSAT encoding and *vice versa*. This permits two key observations about VARIANT-I and VARIANT-II. First, when both variants yield an optimal solution to a fixed-horizon planning problem Π_h, and both those solutions have identical cost, then the solution to VARIANT-I is a cost-optimal plan for Π. Second, if Π is soluble, then there exists some h for which the observation of horizon-independent cost-optimality shall be made by COS-P. Algorithm 3 describes COS-P. Note that lines 4-6 of Algorithm 3 ensure that COS-P is complete by bounding the maximum horizon at which a plan can exist. As described previously, without further reasoning this bound can be set at $|\mathcal{S}|$.

For the remainder of this section we give the Partial Weighted MaxSAT compilation for VARIANT-I and VARIANT-II. First, VARIANT-I uses exactly the encoding for fixed-horizon cost-optimal planning presented in Section 5.2. VARIANT-II includes the constraints from VARIANT-I, with the constraints describing goals (Schema 3.5.2) omitted. It also includes additional constraints for the relaxed suffix, which are a modification of the constraints for planning in the delete relaxation (Section 5.3). Formally, a VARIANT-II encoding is defined as follows.

Definition 5.4.2. Let $\mathcal{T}^{II}$ be a function which takes a bounded planning problem Π_h and returns a Partial Weighted MaxSAT VARIANT-II instance. We say that $\mathcal{T}^{II}$ reduces Π_h to SAT and that $\phi_h := \mathcal{T}^{II}(\Pi_h)$ is an encoding of Π_h iff a plan $\pi = \langle \pi', \pi^+ \rangle$ with the following structure can be extracted from each satisfying valuation of ϕ_h:

- There is a valid parallel plan π' with horizon h that reaches some $s \in \mathcal{S}$; and
- Either $\mathcal{G} \subseteq s$, or there is a valid relaxed plan π^+ from s to a goal state.

Finally, we require that ϕ_h is unsatisfiable only if there is no such plan. □

Algorithm 3 Cost-Optimal RSat – COS-P

```
1: Input:
   - A planning problem $\Pi$;
   - A given non-negative weight bound $\hat{c}^I$. If none is known: $\hat{c}^I := \infty$;
   - An absolute horizon bound $h_{abs}$ such as $|\mathcal{S}|$;
2: $h \leftarrow 1$; $\hat{c} \leftarrow \hat{c}^I$; $\hat{\pi} \leftarrow []$;
3: while true do
4:   if $h \geq h_{abs}$ then
5:     return NO-PLAN;
6:   end if
7:   Solve a VARIANT-I instance of $\Pi_h$ with the initial bound $\hat{c}$ to get plan $\Pi^{*h}$;
8:   if UNSATISFIABLE then
9:     $h \leftarrow h + 1$;
10:  else
11:    $\hat{\pi} \leftarrow \Pi^{*h}$; $\hat{c} \leftarrow \mathcal{C}(\Pi^{*h})$;
12:    Solve a VARIANT-II instance of $\Pi_h$ with the initial bound $\hat{c}$ to get plan $\Pi^*$;
13:    if UNSATISFIABLE then
14:      return $\hat{\pi}$;
15:    else if $\Pi^*$ is a valid plan then
16:      return $\Pi^*$;
17:    end if
18:  end if
19: end while
```

We now formally describe the encoding $\mathcal{T}^{II}$. First, it uses the variables and constraints of the encoding for fixed-horizon cost-optimal planning (Section 5.2.1) except that the goal constraints (Schema 3.5.2) are omitted. Next, it uses the variables and constraints for computing planning heuristics (Section 5.3) with the following modifications. In this case the encoding is attached to the fluents at horizon h of the prefix, rather than to a fixed start state. First, Schemata 5.3.1 and 5.3.2, are identical except that the restriction that the fluent $f \notin s_0$ is lifted. Schema 5.3.5 is replaced by the following:

Schema 5.4.1. VARIANT-II **causal link axioms:** For all fluents $f_i \in \mathcal{F}$, actions $a \in make(f_i)$, and fluents $f_j \in \mathit{pre}(a)$, we have the following clause:

$$add(a, f_i)^+ \rightarrow (f_j^h \vee \mathcal{K}(f_j, f_i)^+) \qquad \square$$

Next, Schemata 5.3.6- 5.3.9 are used unchanged. As the relaxed suffix does not have a fixed start state, we need to modify the definition of redundancy and the schema that prohibits redundant actions.

Definition 5.4.3. For a relaxed planning problem Π^+ and a set $\overrightarrow{P}$ of non-mutex fluents at horizon h, relaxed action $a_i^+ \in \mathcal{A}^+$ is *conditionally redundant* in an optimal solution, iff the fluents in $\overrightarrow{P}$ are true at horizon h and there exists a relaxed action $a_j^+ \in \mathcal{A}^+$, such that:

1. $cost(a_j) \leq cost(a_i)$;
2. $\mathit{pre}(a_j) \backslash \overrightarrow{P} \subseteq \mathit{pre}(a_i) \backslash \overrightarrow{P}$; and
3. $\mathit{post}^+(a_i) \backslash \overrightarrow{P} \subseteq \mathit{post}^+(a_j) \backslash \overrightarrow{P}$.

Schema 5.3.10 is replaced by the following schema, which prohibits actions redundant in the relaxed suffix based on the state at horizon h.

Schema 5.4.2. VARIANT-II **relaxed action cost dominance axioms:** For relaxed action $a^+ \in \mathcal{A}^+$ that is *conditionally redundant* for $\overrightarrow{P_1} \subseteq \mathcal{F}$ and not conditionally redundant for any $\overrightarrow{P_2} \subseteq \mathcal{F}$, where $\overrightarrow{P_2} \subset \overrightarrow{P_1}$ there is the following clause:[4]

$$(\bigwedge_{f \in \overrightarrow{P_1}} f^h) \rightarrow \neg a^+ \qquad \square$$

Schemata 5.3.11- 5.3.21 are used unchanged except that the relaxed suffix layer 0 refers to layer h in the prefix. Finally, we have the following two new schemata.

Schema 5.4.3. Variant-II relaxed goal axioms: For each fluent $f \in \mathcal{G}$, we assert that it is either achieved at the planning horizon h, or using a relaxed action in $\mathcal{A}^+$. This is expressed with a clause:

$$f^h \vee f^+ \qquad \square$$

Schema 5.4.4. Variant-II only necessary relaxed fluent axioms: For each fluent $f \in \mathcal{F}$ we have a constraint:

$$\neg f^+ \vee \neg f^h \qquad \square$$

[4]In practice we limit $|\overrightarrow{P_1}|$ to 2.

To show the correctness of our algorithm COS-P, we first need to show that optimal solutions to Variant-I instances represent optimal plans to fixed-horizon planning problems. The encoding used for Variant-I instances is simply the encoding $\mathcal{T}^c$, which is shown to be constructive by Theorem 5.2.1. Next, we need to show that an encoding $\mathcal{T}^{II}(\Pi_h)$ (a Variant-II encoding) of a fixed-horizon planning problem Π_h, can be used to prove cost-optimality.

Theorem 5.4.1. *If a CNF $\phi_h^{II} \equiv \mathcal{T}^{II}(\Pi_h)$ of a fixed-horizon planning problem Π_h, is unsatisfiable given an initial cost bound c^*, then no plan exists for Π with less than cost c^* for any horizon greater than h. If there is a plan with a cost less than c^* then ϕ_h^{II} must be satisfiable.*

Proof. A variant-II instance consists of two parts $\phi_h^{II} \equiv (\phi^{pref} \wedge \phi^{suff})$, a prefix ϕ^{pref} that consists of the encoding $\mathcal{T}^c$ without the goal clauses (Schema 3.5.2) and a suffix ϕ^{suff} that uses the encoding $\mathcal{T}^{+RL}$ of relaxed planning except that the start state, instead of being fixed, is taken to be the fluents at layer h of the encoding ϕ^{pref}. As the suffix encoding used in ϕ_h^{II} performs similarly in practise with the simpler suffix encoding $\mathcal{T}^{+R}$, for brevity we use the simpler encoding in this proof.

It is easy to see from the constructiveness of a similar encoding $\mathcal{T}^{Pe}$ (Theorem 3.5.1, that a satisfying valuation $\mathcal{V}$ of ϕ^{pref} represents a parallel plan π' with horizon h that reaches some $s' \in \mathcal{S}$. Let this plan have a cost c'. Turning our attention to the encoding suffix ϕ^{suff}, Schema 5.4.3 requires that any fluent $f \in \mathcal{G}$, where $f \notin s'$ must be true in the suffix, representing the goal constraints of $\mathcal{T}^{+R}$. Schema 5.4.4 ensures that any true fluent at h cannot be true in the relaxed suffix. Schema 5.4.1 modifies the related Schema 5.3.3 in $\mathcal{T}^{+R}$ by saying that if $add(a, f_1)$ and $f_2 \in \mathit{pre}(a)$ then either f_2 is true at h or in the relaxed suffix. Previously this constraint would not be created if $f_2 \in s_0$. It should be clear from the proof of constructiveness of $\mathcal{T}^{+R}$ (Theorem 5.3.1) and the modifications described here, that a satisfying valuation to ϕ^{suff}, given a state s' at horizon h, represents a (possibly-empty) relaxed plan from s' that achieves $\mathcal{G}$. Let this satisfying valuation have cost c''. Now a satisfying valuation to ϕ_h^{II} has a total cost $c' + c''$ which must be less than c^*.

In the case where no such satisfying valuation exists then there is no plan with h steps to a state s', followed by a relaxed plan from s' that satisfies the goal of the problem in under the cost bound c^*. This implies that there is no plan at all with any horizon greater than h and a cost less than c^*. If there was such a plan it would be possible to take the first h actions of this plan and make the corresponding variables true in layers 0 through $h - 1$ in the ϕ^{pref} and make the remaining actions true in the suffix ϕ^{suff}, eliminating any duplicate actions. $\square$

Finally, we show the correctness of COS-P.

Theorem 5.4.2. *The algorithm* COS-P *is correct and complete. That is, for a problem* Π *with an optimal cost* c^**,* COS-P *will find a plan* π_h *at some horizon* h*, such that* $\mathcal{C}(\pi_h) = c^*$ *and the CNF* $\phi_h^{II} = \mathcal{T}^{II}(\Pi_h)$ *will be unsatisfiable.*

Proof. It follows from the constructiveness of Variant-I instances (Theorem 5.2.1) that there exists a horizon h for which we can find a horizon-independent cost-optimal plan π^*, that has cost c^* by generating and solving Variant-I instances. It follows from Theorem 5.4.1 that ϕ_h^{II} instances will not be unsatisfiable for a horizon h unless the optimal solution to the Variant-I instance $\mathcal{T}^c(\Pi_h)$ is a horizon-independent cost-optimal plan π^* with cost c^*. Algorithm 3 will therefore loop until we are at such a horizon h with a cost-optimal plan π^* From Theorem 5.4.1 we know that if ϕ_h^{II} is unsatisfiable given our cost bound c^*, then our plan π^* is horizon-independent cost-optimal and the algorithm can terminate. If ϕ_h^{II} is satisfiable given our cost bound c^* then the algorithm will increase h and repeat the process.

It remains to be shown that there always a horizon h where ϕ_h^{II} is unsatisfiable given our cost bound c^*. This follows from a property of the MaxSAT solver used (Algorithm 2) when solving Variant-II instances. Algorithm 2 ensures that any two states s_t and $s_{t'}$ at different layers t and t', implied by a satisfying valuation to ϕ_h^{pref}, must be different. As the number of states in our problem $|\mathcal{S}|$ is finite, as the horizon h increases, ϕ_h^{pref} and therefore ϕ_h^{II} must be unsatisfiable and the algorithm must terminate. □

5.4.2 Empirical Evaluation

We now discuss our experimental comparison of COS-P with IPC baseline planner BASELINE and a version of COS-P called H-ORACLE. We implemented both COS-P and H-ORACLE in C++. The latter is provided with the shortest horizon that yields a horizon-independent cost-optimal plan. Experiments were performed using benchmark problems from a number of International Planning Competitions: IPC-6: ELEVATORS, PEG SOITAIRE, and TRANSPORT; IPC-5: STORAGE, and TPP; IPC-4: PIPESWORLD; IPC-3: DEPOTS, DRIVERLOG, FREECELL, ROVERS, SATELLITE, and ZENOTRAVEL; and IPC-1: BLOCKS, GRIPPER, MICONIC, and MPRIME. Experiments were run on a cluster of AMD Opteron 252 2.6GHz processors, each with 2GB of RAM. All plans computed by COS-P, H-ORACLE, and BASELINE were independently verified and computed within a timeout of 30 minutes.

The results of our experiments are summarised in Table 5.2. For each domain there is one row for the hardest problem solved by each of the three planners. Here, we measure problem hardness as the time it takes each solver to return the optimal plan. In some domains we also include additional problems. Using the same experimental data as for Table 5.2, Figure 5.2 plots the cumulative number of problems solved over time by each planning system, supposing invocations of the systems on problems are made in parallel. It is important to note that in most domains the size of the CNF encodings required by COS-P (and H-ORACLE) are not prohibitively large – i.e, where the SAT-based approaches fail, this is typically because they exceed the 30 minutes timeout, and not because they exhaust system memory. The sizes of the SAT problems generated for problems in Table 5.2 are shown in Table 5.3.

COS-P outperforms the BASELINE in the BLOCKS and FTB domains. For example, on BLOCKS problem 18 BASELINE takes 39.15 seconds while COS-P takes only 3.47 seconds. In other domains BASELINE outperforms COS-P, sometimes by several orders of magnitude. For example, on problem ZENOTRAVEL problem 4 BASELINE takes 0.04 seconds while COS-P takes 841.2 seconds. BASELINE can also solve significantly more problems overall. Importantly, we discovered that in COS-P it is relatively easy to find a cost-optimal solution compared to proving its optimality. For example, on MICONIC problem 23 COS-P took 0.53 seconds to find the optimal plan but spent 1453 seconds proving cost-optimality. More generally, this observation is indicated by the performance of H-ORACLE, which significantly outperforms COS-P.

The difficulty of solving VARIANT-II instances is not surprising, given their structure. First, the prefix, which lacks goal constraints, has a set of satisfying valuations that represent acyclic state trajectories to all states reachable h parallel steps from the start state and within the cost bound. In most planning domains there will be an overwhelming number of such trajectories. The purpose of the relaxed suffix is to attempt to restrict the subset of these trajectories that need to be explored while solving a VARIANT-II instance by imposing a heuristic estimate of the cost to reach the goal from their end states. In practice this does little to alleviate the problem as there are often many trajectories leading to the same end state. One possible solution to this is to require that plans in the prefix have a canonical form, for example by using a process execution semantics [57].

Examining Table 5.3, one interesting thing to note about the relative sizes of the encoding components is that the size of the relaxed suffix can vary significantly compared to the rest of the encoding. For example, for the problem DEPOTS-7, more than 2 million clauses required to

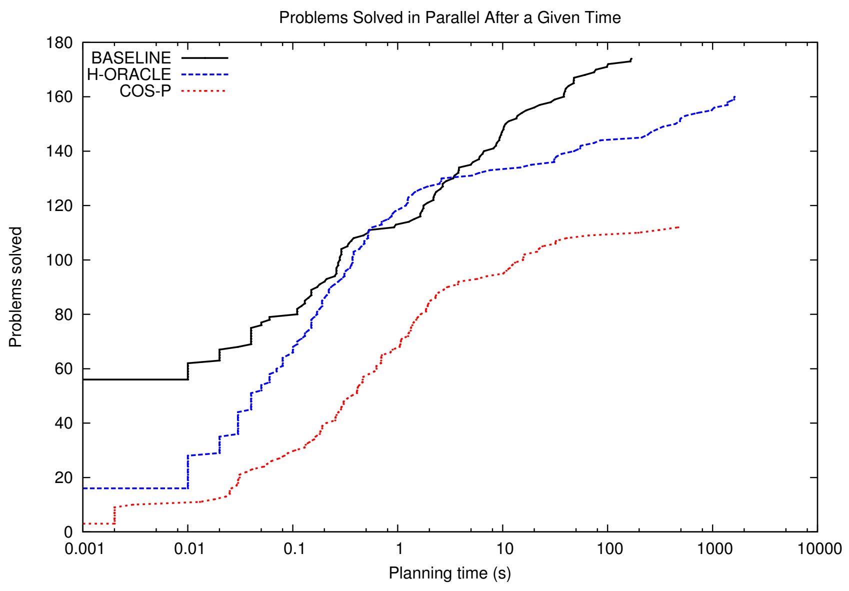


Figure 5.2: The number of horizon-independent cost-optimal problems solved in parallel after a given planning time for BASELINE, H-ORACLE, and COS-P.

encode the relaxed suffix, while only 177 thousand clauses are required for the prefix. Whereas in MICONIC-36 about 7 thousand clauses are required for the suffix, while about 415 thousand are required for the prefix. In general, the overwhelming majority of suffix clauses enforce transitivity in causal support. In line with the results for relaxed planning (Section 5.3.2), the number of clauses required to enforce causal support is not a good predictor of solving performance. See, for example, the MICONIC problems in Table 5.2.

Overall, we find that clause learning procedures in PWM-RSAT cannot exploit the presence of the effective delete relaxation heuristic from Π^+. Consequently, a serious bottleneck of our approach stems from the time required to solve VARIANT-II instances. On a positive note, those proofs are possible, and in domains such as BLOCKS and FTB, where the branching factor is high and useful plans long, the factored problem representations and corresponding solution procedures in the SAT-based setting are worthwhile. Moreover, in fixed-horizon cost-optimal planning, the SAT approach continues to show good performance characteristics in many domains.

5.5 Conclusion

In this chapter we demonstrated that a general theorem proving technique, particularly a DPLL procedure for Boolean SAT, can be modified to find cost-optimal solutions to propositional planning problems encoded as SAT.[5] In particular, we modified the SAT solver RSAT to create PWM-RSAT, an effective partial weighted MaxSAT procedure for problems where all *soft* constraints are unit clauses. This forms the underlying optimisation procedure in COS-P, our cost-optimal planning system that, for successive horizon lengths, uses PWM-RSAT to establish a candidate solution at that horizon, and then to determine if that candidate is horizon-independent cost-optimal. Each candidate is a minimal-cost step-bounded plan for the problem at hand. That a candidate is horizon-independent optimal is known if no step-bounded plan with a relaxed suffix has lower cost. To achieve that, we developed a MaxSAT encoding of bounded planning problems with a relaxed suffix and proved that this encoding is constructive. This encoding constitutes the first application of the causal representations of planning [34] to prove cost-optimality.

Additionally, we applied the solver PWM-RSAT to optimally solving relaxed encodings of planning problems and find that it significantly outperforms BASELINE at this task. The ability to optimally solve relaxed problems represents the ability to compute a heuristic similar to the highly informative h^+ heuristic. Given the difficulty of solving many of the relaxed instances, it seems unlikely that a state-search planner that uses BASELINE or a SAT-based approach with PWM-RSAT as a heuristic estimator would be competitive. Our results are sufficiently promising to suggest that a SAT-based approach may one day be useful for practical heuristic estimation.

Existing work directly related to COS-P includes the hybrid solver CO-PLAN [58] and the fixed-horizon optimal system PLAN-A [11]. Those systems placed 4th and last respectively out of 10 systems at IPC-6. CO-PLAN is a hybrid planner in the sense that it proceeds in two phases, each of which applies a different search technique. The first phase is SAT-based, and identifies the cheapest step-optimal plan. PLAN-A also performs that computation, however assumes that a least cost step-optimal plan is horizon-independent optimal – Therefore PLAN-A was not competitive because it could not guarantee that the plans it returned were cost-optimal (and often they were not) and thus forfeited in many domains. The first phase of CO-PLAN and the PLAN-A system can be seen as more general and efficient versions of the system described in Buttner and Rintanen [9]. The second phase of CO-PLAN breaks from the planning-as-SAT paradigm. It corresponds to a cost-bounded anytime best-first search. The cost bound for the second phase is provided by the first phase. Although competitive with a number of other competition entries, CO-PLAN is not competitive in IPC-6 competition benchmarks with the BASELINE – The *de facto* winning entry, a brute-force A^* in which the distance-plus-cost computation always takes

[5]This was supposed to be possible, though in an impractical sense in Giunchiglia and Maratea [23].

the distance to be zero.

Other work related to COS-P leverages SAT modulo theory (SMT) procedures to solve problems with metric resource constraints [74]. SMT-solvers typically interleave calls to a *simplex* algorithm with the *decision steps* of a backtracking search, such as DPLL. Solvers in this category include the systems LPSAT [74], TM-LPSAT [66], and NUMREACH/SMT [28]. SMT-based planners also operate according to the BLACKBOX scheme, posing a series of step-bounded decision problems to an SMT solver until an optimal plan is achieved. Because they are not horizon-independent optimal, existing SMT systems are not directly comparable to COS-P.

One of the most pressing item for future work is a technique to exploit SMT —and/or branch-and-bound procedures from weighted MaxSAT— in proving the optimality of candidate solutions that PWM-RSAT yields for bounded problems. We should also exploit recent work in using useful admissible heuristics for state-based search when evaluating whether horizon h yields an optimal solution [27].

		BASELINE	$\mathcal{T}^{+}$	$\mathcal{T}^{+R}$	$\mathcal{T}^{+L}$	$\mathcal{T}^{+RL}$
Problem	C^*	t	t	t	t	t
BLOCKS-7	11	21.3	0.54	0.49	0.58	0.57
BLOCKS-28	25	-	627	557	452	629
BLOCKS-29	25	-	1205	588	1374	-
BLOCKS-30	27	-	1122	692	-	564
DEPOTS-2	14	38.5	2.93	1.56	5.54	1.70
DEPOTS-3	22	-	55.3	25.7	173	13.2
DEPOTS-4	17	-	72.0	46.1	33.0	37.4
DEPOTS-10	22	-	-	164	-	179
DEPOTS-13	24	-	-	272	-	101
DRIVERLOG-4	12	78.0	0.73	0.42	0.78	0.38
DRIVERLOG-7	12	-	76.2	3.04	75.4	9.50
DRIVERLOG-8	15	-	-	621	-	693
ELEVATORS-1	32	74.9	1.19	0.66	1.04	0.89
ELEVATORS-20	51	-	1514	19.1	-	23.2
ELEVATORS-29	57	-	484	27.1	905	54.6
ELEVATORS-30	57	-	741	31.2	1170	45.4
FREECELL-1	8	194	-	-	-	-
FTB-1	201	0	0	0	0.01	0.01
FTB-40	1001	-	0.81	0.58	3.13	2.84
GRID-1	10	1.75	-	-	-	-
GRIPPER-3	17	137	0.15	0.14	0.20	0.18
GRIPPER-6	29	-	56.1	5.11	65.3	12.3
GRIPPER-7	33	-	1500	162	-	167
LOGISTICS98-1	24	-	244	2.91	158	4.71
MICONIC-17	13	0.06	0.13	0	0.28	0.02
MICONIC-20	15	0.03	123	0.01	209	0.02
MICONIC-27	19	-	-	0.02	-	0.04
MICONIC-28	19	-	-	0.01	-	0.10
MICONIC-29	18	-	-	0.01	750	0.04
MPRIME-4	7	-	42.0	85.7	154	166
MPRIME-12	5	-	277	15.5	433	90.9
PEGSOL-26	5	-	12.3	11.8	24.7	14.5
PEGSOL-29	8	-	22.4	23.2	22.0	20.1
PEGSOL-30	13	-	-	-	-	955
PIPESWORLD-2	7	0.22	11.1	2.68	7.74	2.95
PIPESWORLD-5	7	169	-	289	-	528
ROVERS-5	18	18.1	1227	0.27	872	0.37
ROVERS-9	24	-	-	202	-	344
SATELLITE-2	12	-	0.22	0.01	0.48	0.11
SATELLITE-18	31	-	-	432	-	498
STORAGE-10	12	59.2	365	104	130	66.7
STORAGE-12	9	29.3	191	310	159	732
STORAGE-15	11	-	748	591	512	718
TPP-5	17	0.57	0.03	0.03	0.05	0.05
TRANSPORT-2	119	6.29	2.41	1.01	7.08	2.04
TRANSPORT-14	382	-	1389	36.1	1444	105
TRANSPORT-24	428	-	1018	28.2	1533	121
TRANSPORT-25	549	-	-	742	-	1120
ZENOTRAVEL-6	11	711	593	0.37	273	1.12
ZENOTRAVEL-8	10	-	-	50.0	-	117

Table 5.1: The hardest relaxed problems solved by BASELINE and the SAT-based planners using the encodings $\mathcal{T}^{+}$, $\mathcal{T}^{+R}$, $\mathcal{T}^{+L}$, and $\mathcal{T}^{+RL}$ and the Partial Weighted MaxSAT solver PWM-RSAT. C^* is the optimal cost of each problem and t the time taken for each approach. Problems were solved with a time-out of 30 minutes. '-' indicates that a solver either timed out or ran out of memory.

		BASELINE	H-ORACLE		COS-P			
Problem	C^*	t	h	t	h	t_t	t_π	t_*
BLOCKS-17	28	39.83	28	0.59	28	3.7	3.79	0
BLOCKS-18	26	39.15	26	0.53	26	3.47	3.47	0
BLOCKS-23	30	-	30	4.61	30	31.7	31.7	0
BLOCKS-25	34	-	34	3.43	34	32	32	0
DEPOTS-7	21	98.08	11	64.79	-	-	-	-
DRIVERLOG-1	7	0.01	7	0.01	7	0.14	0.02	0.13
DRIVERLOG-6	11	9.25	5	0.046	-	-	-	-
DRIVERLOG-7	13	100.9	7	1.26	-	-	-	-
ELEVATORS-2	26	0.33	3	0.01	3	13	0.01	13
ELEVATORS-5	55	167.9	-	-	-	-	-	-
ELEVATORS-13	59	28.59	10	378.6	-	-	-	-
FREECELL-4	26	47.36	-	-	-	-	-	-
FTB-17	401	38.28	17	0.08	17	0.31	0.09	0.22
FTB-30	1001	-	25	0.7	25	2.31	0.71	1.6
FTB-38	601	-	33	0.48	33	1.85	0.48	1.38
FTB-39	801	-	33	0.7	33	2.72	0.67	2.05
GRIPPER-1	11	0	7	0.02	7	195.2	0.06	195.1
GRIPPER-3	23	0.05	15	34.23	-	-	-	-
GRIPPER-7	47	73.95	-	-	-	-	-	-
MICONIC-15	10	0	8	0.017	8	0.63	0.06	0.57
MICONIC-23	15	0.04	10	0.12	-	-	-	-
MICONIC-36	27	9.62	22	1754	-	-	-	-
MICONIC-39	28	10.61	24	484.1	-	-	-	-
PEGSOL-7	3	0	12	0.08	12	2.28	0.24	2.04
PEGSOL-9	5	0.02	15	7.07	15	468.1	12.16	455.9
PEGSOL-13	9	0.14	21	1025	-	-	-	-
PEGSOL-26	9	42.44	-	-	-	-	-	-
ROVERS-3	11	0.02	8	0.1	8	40.6	0.15	40.5
ROVERS-5	22	164.1	8	69.83	-	-	-	-
SATELLITE-1	9	0	8	0.08	8	1.04	0.27	0.77
SATELLITE-2	13	0.01	12	0.23	-	-	-	-
SATELLITE-4	17	6.61	-	-	-	-	-	-
STORAGE-7	14	0	14	0.59	14	1.33	1.33	0
STORAGE-9	11	0.2	9	643.2	-	-	-	-
STORAGE-13	18	3.47	18	112	18	308.9	308.9	0
STORAGE-14	19	60.19	-	-	-	-	-	-
TPP-4	14	0	5	0	5	0.19	0.01	0.19
TPP-5	19	0.15	7	0.01	-	-	-	-
TRANSPORT-1	54	0	5	0.02	5	0.3	0.03	0.27
TRANSPORT-4	318	47.47	-	-	-	-	-	-
TRANSPORT-23	630	0.92	9	1.28	-	-	-	-
ZENOTRAVEL-3	6	0.06	5	0.16	5	10.8	0.25	10.55
ZENOTRAVEL-6	11	8.77	7	54.35	-	-	-	-
ZENOTRAVEL-7	15	5.21	8	1600	-	-	-	-

Table 5.2: Time results for horizon-independent cost-optimal solvers. C^* is the optimal cost for each problem. All times are in seconds. For BASELINE t is the solution time. For H-ORACLE, h is the horizon returned by the oracle and t is the time taken by PWM-RSAT to find the lowest cost plan at h. For COS-P, t_t is the total time for all SAT instances, t_π is the total time for all SAT instances where the system was searching for a plan, while t_* is the total time for all SAT instances where the system is performing optimality proofs. '-' indicates that a solver either timed out or ran out of memory.

		Prefix		Suffix		Suffix Opt.	
Problem	h	$\#v$	$\#c$	$\#v$	$\#c$	$\#v$	$\#c$
BLOCKS-17	28	1973	$211K$	$10.5K$	$1.02M$	2172	$142K$
BLOCKS-18	26	1946	$211K$	$10.5K$	$1.02M$	2172	$142K$
BLOCKS-23	30	4134	$830K$	$21.5K$	$3.03M$	3180	$282K$
BLOCKS-25	34	4544	$952K$	$29.4K$	$4.89M$	3756	$382K$
DEPOTS-7	11	1465	$177K$	$18.4K$	$2.29M$	3672	$253K$
DRIVERLOG-1	7	248	3141	886	$18.7K$	912	$16.7K$
DRIVERLOG-6	5	492	$16.3K$	2871	$99.2K$	2124	$49.1K$
DRIVERLOG-7	7	708	$48.6K$	3495	$131K$	2412	$58.4K$
ELEVATORS-2	3	521	4663	4722	$221K$	2946	$78.1K$
ELEVATORS-13	10	1159	$242K$	7890	$537K$	3804	$129K$
FTB-17	17	1503	$19.4K$	1031	7866	2424	$38.6K$
FTB-30	25	7497	$112K$	6273	$71.5K$	8904	$303K$
FTB-38	33	7541	$113K$	4965	$66.3K$	7080	$212K$
FTB-39	33	$10.0K$	$153K$	7147	$98.4K$	9432	$339K$
GRIPPER-1	7	175	4217	436	7492	456	8234
GRIPPER-3	15	603	$31.2K$	1364	$44.6K$	840	$22.3K$
MICONIC-15	8	123	5330	123	717	360	4034
MICONIC-23	10	283	$47.3K$	325	2775	840	$11.5K$
MICONIC-36	22	787	$415K$	623	6993	1512	$24.2K$
MICONIC-39	24	995	$769K$	808	$10.2K$	1920	$32.8K$
PEGSOL-7	12	997	$31.2K$	6777	$469K$	1637	$92.7K$
PEGSOL-9	15	1622	$194K$	$10.7K$	$1.02M$	2226	$147K$
PEGSOL-13	21	2201	$294K$	$10.7K$	$1.02M$	2262	$148K$
ROVERS-3	8	728	$28.9K$	5897	$407K$	1626	$82.9K$
ROVERS-5	8	515	$16.0K$	3776	$210K$	1154	$53.6K$
SATELLITE-1	8	176	$12.2K$	208	1303	558	6782
SATELLITE-2	12	487	$61.4K$	546	4147	1104	$15.3K$
STORAGE-7	14	707	$61.8K$	3166	$147K$	1452	$49.3K$
STORAGE-9	9	1009	$244K$	7566	$529K$	3348	$121K$
STORAGE-13	18	2239	$528K$	$15.6K$	$1.69M$	3276	$213K$
TPP-4	5	169	769	256	1440	492	5534
TPP-5	7	375	2498	702	5683	900	$13.3K$
TRANSPORT-1	5	213	5927	764	$12.8K$	936	$16.9K$
TRANSPORT-23	9	1076	$336K$	5314	$234K$	4560	$108K$
ZENOTRAVEL-3	5	457	$56.7K$	1220	$11.8K$	2172	$34.0K$
ZENOTRAVEL-6	7	798	$255K$	1966	$20.4K$	3504	$57.2K$
ZENOTRAVEL-7	8	888	$201K$	2156	$23.3K$	3672	$60.5K$

Table 5.3: CNF sizes for COS-P. h is the horizon returned by the oracle for each instance. $\#v$ and $\#c$ represent the number of variables and clauses for the various components of a COS-P VARIANT-II instance. Prefix represents the variables and clauses for the prefix of each instance. Suffix represents the variables and clauses of the basic relaxed suffix without the redundancy clauses or the variables and clauses for the relaxed layers. Finally, Suffix Opt. represents the variables and clauses of the redundancy constraints and the relaxed layers. Details are included for instances even if they could not be solved at the indicated horizon or if no VARIANT-II instance was generated in solving the problem.

Chapter 6

Integrating Control Knowledge into SAT-Based Planning

6.1 Introduction

Classical domain-independent fixed-horizon planning is NP-complete and impractical for many real-world planning problems. To alleviate this problem, control knowledge can be used to guide planning systems and increase the efficiency of planning. Additionally, control knowledge can be used to restrict valid solutions to plans that have certain desirable properties. Kautz and Selman [37] explore how to integrate hand-coded control knowledge into SAT-based planning. In particular, they looked at encoding *invariants*, *optimality constraints*, and *simplifying constraints*. That hand-coded approach has limits in practice because constraints must be tailored specifically to each temporal constraint in the planning domain at hand.

Following on from this work, Mattmüller and Rintanen [44] describe an encoding of Linear Temporal Logic (LTL), interpreted over an infinite sequence of state, into SAT-based planning. They use this system to plan with temporally extended goals. While their system allows the specification of constraints for planning domains, rather than just individual instances, it has the drawback of being relatively inexpressive. In particular, the LTL they encode does not include the *next* operator and does not allows temporal operators that refer to the past.

In the spirit of Son et al. [69], we shall review a number of different high-level control knowledge formalisms and present novel split encodings of these formalisms and hybrids of them into SAT. The languages we describe are relatively expressive and provide a means of intuitively encoding a number of natural constraints at the domain level. We examine LTL with a finite state sequence semantics, the procedural control languages Golog [40] and ParaGolog, a novel variant of ConGolog [22] that has a semantics that allows true parallel action execution. We additionally present a novel formulation of Hierarchical Task Networks (HTNs) based on the SHOP planning system [50]. Some of the control knowledge formalisms we examine overlap in expressiveness, but often one representation is significantly more compact and efficient for a particular task. For example, the partial ordering constraints of HTNs can be represented in Golog, but this requires that all permissible partial orders be represented explicitly in the Golog program. The control knowledge formalisms that we encode are more expressive that existing formalisms that have been encoded with SAT or ASP. One reason is because we allow true parallel action execution, a feature which is essential to the efficiency of existing SAT-based planning systems [59].

For each formalism, we present a number of representational variants that are used in specific circumstances to maximise compactness of the resulting encodings. Specifically, as well as flat encodings, we present split encodings that split on operator arguments and on steps. For example, the predicate *trans*(n, δ_1, t_1, t_2), which says that Golog complex action δ with name n is executed from step t_1 to step t_2, can be replaced by two predicates *trans-s*(n, δ_1, t_1) and *trans-e*(n, δ_1, t_2), which represent the start and end (resp.) of the action. Like Son et al. [69], the constraints we generate are non-essential for planning and are conjoined to an underlying planning encoding. Any underlying planning encoding may be used as long as it correctly encodes parallel planning and has variables that directly represent actions and fluents. In particular, the encodings in Sections 3.3, 4.2.3, and 5.2 are suitable.

The control knowledge formalisms that we present here are useful for a range of planning, verification [1], and diagnosis problems [68]. When used for planning, we can encode temporarily extended goals [3] and can use the procedural knowledge to restrict plans to those that follow predefined plan sketches. The constraints we generate for this control knowledge also increase the efficiency of planning by placing tighter constraints on satisfying valuations of the resulting SAT problem.

In the rest of this chapter we formally describe the problem of planning with control knowledge, we then describe the control knowledge formalisms that we allow. We present a number of encodings for each formalism that vary in compactness and expressiveness. Throughout this exposition we describe hybrids of the presented formalisms and present encodings of all formalisms into SAT. Finally, we provide a preliminary empirical proof of concept of some of the formalisms.

6.2 Setting

Here, we consider the model of classical propositional planning presented in Chapter 2. The encodings of domain-independent control knowledge that follow can be used with a number of underlying planning encodings. In particular, encodings with the following properties:

- There are variables for individual actions and fluents. Split encodings can be used but these must also include action variables (as in Section 4.2.3);
- Effect and frame axioms must fully determine state transitions. This means we must have constraints for effects and must use *full* frame axioms;
- Standard conflict mutex must be enforced – i.e. a set of actions $\mathcal{A}$ can only be executed in parallel in a state if all serial executions of the actions in $\mathcal{A}$ are valid and lead to the same successor state.

The encodings that follow could be integrated into more expressive formulations of planning. For example, those that allow conditional effects (along the lines of Mattmüller and Rintanen [44]), arbitrary preconditions, nondeterminism, and partial observability (along the lines of Majercik [42] and Littman, et al. [41]). Additionally, it is possible to use these encodings of control knowledge in SAT-based planning systems that find cost-optimal plans (such as those in Chapter 5). In this work we use an underlying encoding similar to that of Sideris and Dimopoulos [67], in particular a plangraph-based encoding from Section 3.5 with weak action mutex (Schema 3.5.6), full frame axioms (Schemata 3.5.8 and 3.5.9), and no noop actions.

6.3 Temporal State Constraints

In classical planning, goals are specified as conjunctions of literals. In some situations we require more expressive goals, such as temporally extended goals [3], which allow the specification of constraints on the state trajectories implied by the execution of valid plans. We present a SAT encoding of Linear Temporal Logic (LTL) with past operators. The particular logic we use is interpreted over a finite horizon and is along the lines of Baier and McIlraith [5] and Bauer and Haslum[6]. The constraints we describe can be used to specify temporally extended goals and also form a part of the encodings we present of Golog (Section 6.4), ParaGolog (Section 6.4.5), and HTNs (Section 6.5). We first define non-temporal and temporal formulae. These formulae use the following elements:

- The first order quantifiers $\forall$ and $\exists$;
- The propositional connectives: *and*, *or*, and *not*;
- The temporal operators: *always*, *next*, *until*, *eventually*, *release*, *previous*, *always-past*, *until-past*, *eventually-past*, *release-past*; and
- A goal operator: *goal*.

Son, et al. [69] note that, as they are working with object types with finite domains, they can transform formula to remove existential and universal quantifiers. We do not perform this transformation, and instead encode existential and universal quantifiers directly. Encoding quantifiers using splitting can be significantly more compact than compiling the quantifiers away. We now describe the semantics of the logic we are using.

Definition 6.3.1. Let $\psi[?x \leftarrow X]$ denote formula ψ with free variable $?x$ replaced by constant X. $\square$

Definition 6.3.2. A non-temporal formula ψ is one of the following predicates, where all subformula ψ_1 and ψ_2 are also non-temporal formulae: $\forall(?x, type, \psi_1)$, $\exists(?x, type, \psi_1)$, *not*$(\psi_1)$, *and*$(\psi_1, \psi_2)$, *or*$(\psi_1, \psi_2)$, $f(?x_1, ..., ?x_k)$, and *goal*$(f(?x_1, ...?x_k))$.[1] $\square$

Non-temporal formulae are interpreted in a state s. A state is then said to *satisfy* a non-temporal formula.

Definition 6.3.3. Let $s \models \psi$ mean that s *satisfies* non-temporal formula ψ. The following defines when a state s satisfies a non-temporal formula ψ:

1. If $\psi = \forall(?x, type, \psi_1)$, then $s \models \psi$ iff $\forall_{X \in \mathrm{D}(type)} s \models \psi_1[?x \leftarrow X]$;
2. If $\psi = \exists(?x, type, \psi_1)$, then $s \models \psi$ iff $\exists_{X \in \mathrm{D}(type)} s \models \psi_1[?x \leftarrow X]$;
3. If $\psi =$ *not*(ψ_1), then $s \models \psi$ iff $s \not\models \psi_1$;
4. If $\psi =$ *and*(ψ_1, ψ_2), then $s \models \psi$ iff $s \models \psi_1$ and $s \models \psi_2$;
5. If $\psi =$ *or*(ψ_1, ψ_2), then $s \models \psi$ iff at least one of $s \models \psi_1$ or $s \models \psi_2$;
6. If $\psi = f(X_1, ..., X_k)$, then $s \models \psi$ iff $f(X_1, ..., X_k) \in s$; and

[1]Some or all of the variables for the f and *goal* predicates may be constants.

7. If $\psi = \mathit{goal}(f(X_1, ..., X_k))$, then $\models \psi$ iff $\mathcal{G} \subseteq f(X_1, ..., X_k)$ for fluent $f(X_1, ...X_k)$. $\square$

In the previous definition we introduce variables that are attached to quantifiers. Here, we require that variable names are unique. In particular, for a formula $\psi = \forall(?x, \mathit{type}, \psi_1)$ or $\psi = \exists(?x, \mathit{type}, \psi_1)$, we require that ψ_1 does not contain a sub-formula of the form $\forall(?x, \mathit{type}, \psi_i)$ or $\exists(?x, \mathit{type}, \psi_i)$. Additionally, in the previous definition fluent and goal predicates can only be interpreted in a state if they are ground. This means that we are restricted to *closed* formulae, where all variables are within the scope of a quantifier.

In addition to the predicates for non-temporal formulae there is a set of predicates to represent temporal formulae.

Definition 6.3.4. A temporal formula ψ is one of the following predicates, or any non-temporal predicate from (Definition 6.3.2) that has a temporal formula as a sub-formula:

$$\mathit{next}(\psi_1),\ \mathit{always}(\psi_1),\ \mathit{eventually}(\psi_1),\ \mathit{until}(\psi_1, \psi_2),\ \mathit{release}(\psi_1, \psi_2),\ \mathit{previous}(\psi_1),$$
$$\mathit{always\text{-}past}(\psi),\ \mathit{eventually\text{-}past}(\psi_1),\ \mathit{until\text{-}past}(\psi_1, \psi_2),\ \mathit{release\text{-}past}(\psi_1, \psi_2) \quad \square$$

The temporal operators used are usually defined for the infinite-horizon case – i.e. in Son et al. [69]. As we have a fixed horizon h, the following two assumptions are made:

1. No actions occur before step 0 – i.e. if a fluent f is *true* (resp. *false*) in state s_0 then f would also be *true* (resp. *false*) in all states at steps before 0, if such states existed; and

2. No actions occur on or after step h – i.e. if a fluent f is *true* (resp. *false*) in state s_h then f would also be *true* (resp. *false*) in all states at steps after h, if such states existed.

Bauer and Haslum [6] describe a version of LTL for infinite extensions of finite traces, that is where a temporal formula is interpreted over a finite state sequence where the final state is assumed to repeat infinitely. The two assumptions made previously make our approach very similar to theirs. They argue that their semantics for LTL provide a compelling interpretation of LTL formulae when these formulae are used for temporally extended goals in planning, as we are using them here.

The decision to interpret LTL formulae over a finite sequence of states (or a finite sequence with an infinite extension of the final state), rather than over an infinite sequence means that some natural LTL formula can never be satisfied. For example, a formula that implies alternating states: $\mathit{always}(f \leftrightarrow \mathit{next}(\neg f))$, written $\mathit{always}(\mathit{and}(\mathit{or}(\mathit{not}(f), \mathit{next}(f)), \mathit{or}(f, \mathit{next}(\mathit{not}(f)))))$ here, can never be satisfied by a finite sequence of states, under the assumptions we have made. Such a formula may be useful, for example, in the area of verification, where it is important to assert that properties of a system hold indefinitely. However, in the classical planning setting examined in this thesis, such a formula is less useful. In our setting, it may be possible to admit such formulae by introducing a *weak* next operator along the lines of Baier and McIlraith [5], where *next*(f) is always interpreted as true in the final state. We leave such considerations for future work.

Temporal formulae are interpreted over a finite sequence of states $\overrightarrow{\mathcal{S}} := \langle s_0, s_1, ..., s_h\rangle$ and a state $s_i \in \overrightarrow{\mathcal{S}}$.

Definition 6.3.5. Let $\langle s_i, \overrightarrow{\mathcal{S}}\rangle \models \psi$ mean that the sequence of states $\overrightarrow{\mathcal{S}} := \langle s_0, s_1, ..., s_h\rangle$ and state $s_i \in \overrightarrow{\mathcal{S}}$ satisfy temporal formula ψ. Non-temporal components of temporal formulae are interpreted with respect to s_i alone in the pair $\langle s_i, \overrightarrow{\mathcal{S}}\rangle$. The following defines when $\overrightarrow{\mathcal{S}}$ and s_i satisfy a temporal formula ψ:

1. If $\psi = \textit{next}(\psi_1)$, then $\langle s_i, \overrightarrow{\mathcal{S}}\rangle \models \psi$ iff $\langle s_{max(h,i+1)}, \overrightarrow{\mathcal{S}}\rangle \models \psi_1$;

2. If $\psi = \textit{always}(\psi_1)$, then $\langle s_i, \overrightarrow{\mathcal{S}}\rangle \models \psi$ iff $\forall_{j\in\{i,...,h\}}\langle s_j, \overrightarrow{\mathcal{S}}\rangle \models \psi_1$;

3. If $\psi = \textit{eventually}(\psi_1)$, then $\langle s_i, \overrightarrow{\mathcal{S}}\rangle \models \psi$ iff $\exists_{j\in\{i,...,h\}}\langle s_j, \overrightarrow{\mathcal{S}}\rangle \models \psi_1$;

4. If $\psi = \textit{until}(\psi_1, \psi_2)$, then $\langle s_i, \overrightarrow{\mathcal{S}}\rangle \models \psi$ iff
$$\exists_{j\in\{i,...,h\}}(\langle s_j, \overrightarrow{\mathcal{S}}\rangle \models \psi_2 \wedge \forall_{k\in\{i,...,j\}}\langle s_k, \overrightarrow{\mathcal{S}}\rangle \models \psi_1)$$

5. If $\psi = \textit{release}(\psi_1, \psi_2)$, then $\langle s_i, \overrightarrow{\mathcal{S}}\rangle \models \psi$ iff
$$(\exists_{j\in\{i,...,h\}}\langle s_j, \overrightarrow{\mathcal{S}}\rangle \models \psi_2 \wedge \forall_{k\in\{i,...,j\}}\langle s_k, \overrightarrow{\mathcal{S}}\rangle \models \psi_1) \vee (\forall_{k\in\{i,...,h\}}\langle s_k, \overrightarrow{\mathcal{S}}\rangle \models \psi_1))$$

6. If $\psi = \textit{previous}(\psi_1)$, then $\langle s_i, \overrightarrow{\mathcal{S}}\rangle \models \psi$ iff $\langle s_{min(0,i-1)}, \overrightarrow{\mathcal{S}}\rangle \models \psi_1$;

7. If $\psi = \textit{always-past}(\psi_1)$, then $\langle s_i, \overrightarrow{\mathcal{S}}\rangle \models \psi$ iff $\forall_{j\in\{0,...,i\}}\langle s_j, \overrightarrow{\mathcal{S}}\rangle \models \psi_1$;

8. If $\psi = \textit{eventually-past}(\psi_1)$, then $\langle s_i, \overrightarrow{\mathcal{S}}\rangle \models \psi$ iff $\exists_{j\in\{0,...,i\}}\langle s_j, \overrightarrow{\mathcal{S}}\rangle \models \psi_1$;

9. If $\psi = \textit{until-past}(\psi_1, \psi_2)$, then $\langle s_i, \overrightarrow{\mathcal{S}}\rangle \models \psi$ iff
$$\exists_{j\in\{0,...,i\}}(\langle s_j, \overrightarrow{\mathcal{S}}\rangle \models \psi_2 \wedge \forall_{k\in\{j,...,i\}}\langle s_k, \overrightarrow{\mathcal{S}}\rangle \models \psi_1)$$

10. If $\psi = \textit{release-past}(\psi_1, \psi_2)$, then $\langle s_i, \overrightarrow{\mathcal{S}}\rangle \models \psi$ iff
$$(\exists_{j\in\{0,...,i\}}\langle s_j, \overrightarrow{\mathcal{S}}\rangle \models \psi_2 \wedge \forall_{k\in\{j,...,i\}}\langle s_k, \overrightarrow{\mathcal{S}}\rangle \models \psi_1) \vee (\forall_{k\in\{0,...,i\}}\langle s_k, \overrightarrow{\mathcal{S}}\rangle \models \psi_1))$$

□

Definition 6.3.6. A *temporal constraint* is a temporal formula asserted at a step. A formula ψ is asserted at a step $t \in \{0, ..., h\}$ with the following predicate:

$$\textit{holds}(\psi, t) \qquad \square$$

Certain formulae are assumed to have a *default* step.

Definition 6.3.7. The *default step* for predicates is as follows:

- 0 – for *always*, *eventually*, *release*, and *until*;
- h – for *always-past*, *eventually-past*, *release-past*, and *until-past*; and
- none – otherwise. □

Definition 6.3.8. A temporal constraint can assert a formula ψ at its default step with the following predicate:

$$\textit{holds}(\psi) \qquad \square$$

We illustrate the previously described concepts by presenting some examples of well-formed temporal formulae:

- *Goal-independent* formula:

$$\begin{aligned}&\textit{holds}(\textit{always}(\forall(t, truck, \forall(l_1, loc, \forall(l_2, loc,\\&\quad \textit{not}(\textit{and}(\textit{eq}(l_1, l_2), \textit{and}(at(t, l_1), at(t, l_2)))))))))\end{aligned}$$

- *Goal-dependent* formula:

$$\begin{aligned}&\textit{holds}(\textit{always}(\forall(p, package, forall(l, loc,\\&\quad \textit{or}(\textit{not}(\textit{and}(at(p, l), \textit{goal}(at(p, l)))), \textit{next}(at(p, l)))))))\end{aligned}$$

- Formula with past operators:

$$\begin{aligned}&\textit{holds}(\textit{always}(\forall(p, package, \forall(l, loc, \forall(t, truck,\\&\quad \textit{or}(\textit{not}(\textit{and}(in\text{-}truck(p, t), \textit{previous}(at(p, l)))),\\&\quad \textit{always-past}(at(p, l)))))))))\end{aligned}$$

Formally, we define an encoding of a set of temporal constraints Ψ and bounded planning problem Π_h as follows.

Definition 6.3.9. Let encoding $\mathcal{T}^\tau$ be a function which takes a bounded planning problem Π_h and a set of temporal constraints Ψ and returns a SAT problem. We say that $\mathcal{T}^\tau$ reduces Π_h and Ψ to SAT and that $\phi^\tau := \mathcal{T}^\tau(\Pi_h, \Psi)$ is an encoding of Π_h and Ψ iff:

- A valid plan π can be extracted from every satisfying valuation of ϕ^τ; and
- Each temporal constraint $\mathit{holds}(\psi, t) \in \Psi$ is satisfied by $\langle s_t, \mathcal{S}_\pi \rangle$, where $\mathcal{S}_\pi$ the sequence of states visited during the execution of π and $s_t \in \mathcal{S}_\pi$ is the state in $\mathcal{S}_\pi$ for step t.

Finally, we require that ϕ^τ is unsatisfiable only if there is no such plan. □

We first present a flat encoding of the constraints for all formula constructs and then give an encoding that splits on the groundings of arguments within the scope of quantifiers. The constraints generated by the following encodings are conjoined with the constraints from an encoding of the planning problem, as described in Section 6.2.

6.3.1 Flat Encoding

First, we present a flat encoding of a set of temporal constraints Ψ, for a fixed horizon planning problem Π_h, where the groundings of variables in quantifiers are explicitly enumerated. The following schemata are recursively applied to each temporal constraint $\mathit{holds}(\psi_i, t) \in \Psi$ and a propositional variable is generated for each encountered *holds* predicate. We have the following clauses, which assert that the temporal constraints hold:

Schema 6.3.1. For each temporal constraint $\mathit{holds}(\psi_i, t) \in \Psi$, there is the following clause:

$$\mathit{holds}(\psi_i, t_i)$$ □

We then have the following schemata.

Existential quantifier: $\psi = \exists(?x, \mathit{type}, \psi_1)$

Schema 6.3.2. For all $\mathit{holds}(\exists(?x, \mathit{type}, \psi_1), t)$, there are the following clauses:

$$\mathit{holds}(\exists(?x, \mathit{type}, \psi_1, t)) \leftrightarrow \bigvee_{X \in \mathrm{D}(\mathit{type})} \mathit{holds}(\psi_1[?x \leftarrow X], t)$$ □

For all quantifier: $\psi = \forall(?x, type, \psi_1)$

Schema 6.3.3. For all $holds(\forall(?x, type, \psi_1), t)$, there are the following clauses:

$$holds(\forall(?x, type, \psi_1), t) \leftrightarrow \bigwedge_{X \in \mathrm{D}(type)} holds(\psi_1[?x \leftarrow X], t) \qquad \square$$

Not connective: $\psi = not(\psi_1)$

Schema 6.3.4. For all $holds(not(\psi_1), t)$, there are the following clauses:

$$holds(not(\psi_1), t) \leftrightarrow \neg holds(\psi_1, t) \qquad \square$$

And connective: $\psi = and(\psi_1, \psi_2)$

Schema 6.3.5. For all $holds(and(\psi_1, \psi_2), t)$, there are the following clauses:

$$holds(and(\psi_1, \psi_2), t) \leftrightarrow (holds(\psi_1, t) \wedge holds(\psi_2, t)) \qquad \square$$

Or connective: $\psi = or(\psi_1, \psi_2)$

Schema 6.3.6. For all $holds(or(\psi_1, \psi_2), t)$, there are the following clauses:

$$holds(or(\psi_1, \psi_2), t) \leftrightarrow (holds(\psi_1, t) \vee holds(\psi_2, t)) \qquad \square$$

Ground fluent: $\psi = f(X_1, ..., X_k)$

Schema 6.3.7. For all $holds(f(X_1, ..., X_k), t)$, there are the following clauses:

$$holds(f(X_1, ..., X_k), t) \leftrightarrow f(X_1, ...X_k)^t \qquad \square$$

If $f(X_1, ..., X_k)$ is static, then simplify with $holds(f(X_1, ..., X_k), t) = \top$. If $f(X_1, ..., X_k) \notin \mathcal{F}^t$, then simplify with $holds(f(X_1, ..., X_k), t) = \bot$.

Goal formula: $\psi = goal(f(X_1, ..., X_k)$

Schema 6.3.8. For all $holds(goal(f(X_1, ..., X_k)), t)$, if $f(X_1, ..., X_k) \in \mathcal{G}$, there is the following clause:

$$holds(goal(f(X_1, ..., X_k)), t)$$

otherwise, there is the following clause:

$$\neg holds(goal(f(X_1, ..., X_k)), t) \qquad \square$$

Next operator: $\psi = \mathit{next}(\psi_1)$

Schema 6.3.9. For all $\mathit{holds}(\mathit{next}(\psi_1), t)$, there are the following clauses:

$$\mathit{holds}(\mathit{next}(\psi_1), t) \leftrightarrow \mathit{holds}(\psi_1, min(t{+}1, h)) \qquad \square$$

Always operator: $\psi = \mathit{always}(\psi_1)$

Schema 6.3.10. For all $\mathit{holds}(\mathit{always}(\psi_1), t)$, if $t < h$, there are the following clauses:

$$\mathit{holds}(\mathit{always}(\psi_1), t) \leftrightarrow (\mathit{holds}(\psi_1, t) \wedge \mathit{holds}(\mathit{always}(\psi_1), t{+}1))$$

otherwise, there are the following clauses:

$$\mathit{holds}(\mathit{always}(\psi_1), h) \leftrightarrow \mathit{holds}(\psi_1, h) \qquad \square$$

Eventually operator: $\psi = \mathit{eventually}(\psi_1)$

Schema 6.3.11. For all $\mathit{holds}(\mathit{eventually}(\psi_1), t)$, if $t < h$, there are the following clauses:

$$\mathit{holds}(\mathit{eventually}(\psi_1), t) \leftrightarrow (\mathit{holds}(\psi_1, t) \vee \mathit{holds}(\mathit{eventually}(\psi_1), t{+}1))$$

otherwise, there are the following clauses:

$$\mathit{holds}(\mathit{eventually}(\psi_1), h) \leftrightarrow \mathit{holds}(\psi_1, h) \qquad \square$$

Until operator: $\psi = \mathit{until}(\psi_1, \psi_2)$

Schema 6.3.12. For all $\mathit{holds}(\mathit{until}(\psi_1, \psi_2), t)$, if $t < h$, there are the following clauses:

$$\begin{aligned}&\mathit{holds}(\mathit{until}(\psi_1, \psi_2), t) \leftrightarrow\\&(\mathit{holds}(\psi_2, t) \vee (\mathit{holds}(\psi_1, t) \wedge \mathit{holds}(\mathit{until}(\psi_1, \psi_2), t{+}1))\end{aligned}$$

otherwise, there are the following clauses:

$$\mathit{holds}(\mathit{until}(\psi_1, \psi_2), h) \leftrightarrow \mathit{holds}(\psi_2, h) \qquad \square$$

Release operator: $\psi = \mathit{release}(\psi_1, \psi_2)$

Schema 6.3.13. For all $\mathit{holds}(\mathit{release}(\psi_1, \psi_2), t)$, if $t < h$, there are the following clauses:

$$\begin{aligned}&\mathit{holds}(\mathit{release}(\psi_1, \psi_2), t) \leftrightarrow \\ &(\mathit{holds}(\psi_2, t) \vee (\mathit{holds}(\psi_1, t) \wedge \mathit{holds}(\mathit{release}(\psi_1, \psi_2), t{+}1))\end{aligned}$$

otherwise, there are the following clauses:

$$\mathit{holds}(\mathit{release}(\psi_1, \psi_2), h) \leftrightarrow (\mathit{holds}(\psi_2, h) \vee \mathit{holds}(\psi_1, h)) \qquad \square$$

Previous operator: $\psi = \mathit{previous}(\psi_1)$

Schema 6.3.14. For all $\mathit{holds}(\mathit{previous}(\psi_1), t)$, there are the following clauses:

$$\mathit{holds}(\mathit{previous}(\psi_1), t) \leftrightarrow \mathit{holds}(\psi_1, max(t{-}1, 0)) \qquad \square$$

Always-past operator: $\psi = \mathit{always\text{-}past}(\psi_1)$

Schema 6.3.15. For all $\mathit{holds}(\mathit{always\text{-}past}(\psi_1), t)$, if $t > 0$, there are the following clauses:

$$\mathit{holds}(\mathit{always\text{-}past}(\psi_1), t) \leftrightarrow (\mathit{holds}(\psi_1, t) \wedge \mathit{holds}(\mathit{always\text{-}past}(\psi_1), t{-}1))$$

otherwise, there are the following clauses:

$$holds(alwayspast(\psi_1), 0) \leftrightarrow holds(\psi_1, 0) \qquad \square$$

Eventually-past operator: $\psi = \mathit{eventually\text{-}past}(\psi_1)$

Schema 6.3.16. For all $\mathit{holds}(\mathit{eventually\text{-}past}(\psi_1), t)$, if $t > 0$, there are the following clauses:

$$\begin{aligned}&\mathit{holds}(\mathit{eventually\text{-}past}(\psi_1), t) \leftrightarrow \\ &(\mathit{holds}(\psi_1, t) \vee \mathit{holds}(\mathit{eventually\text{-}past}(\psi_1), t{-}1))\end{aligned}$$

otherwise, there are the following clauses:

$$\mathit{holds}(\mathit{eventually\text{-}past}(\psi_1), 0) \leftrightarrow \mathit{holds}(\psi_1, 0) \qquad \square$$

Until-past operator: $\psi = \textit{until-past}(\psi_1, \psi_2)$

Schema 6.3.17. For all $\textit{holds}(untilpast(\psi_1, \psi_2), t)$, if $t > 0$, there are the following clauses:

$$\textit{holds}(\textit{until-past}(\psi_1, \psi_2), t) \leftrightarrow$$
$$(\textit{holds}(\psi_2, t) \vee (\textit{holds}(\psi_1, t) \vee \textit{holds}(\textit{until-past}(\psi_1, \psi_2), t-1))$$

otherwise, there are the following clauses:

$$\textit{holds}(\textit{until-past}(\psi_1, \psi_2), 0) \leftrightarrow \textit{holds}(\psi_2, 0)$$ □

Release-past operator: $\psi = \textit{release-past}(\psi_1, \psi_2)$

Schema 6.3.18. For all $\textit{holds}(\textit{release-past}(\psi_1, \psi_2), t)$, if $t > 0$, there are the following clauses:

$$\textit{holds}(\textit{release-past}(\psi_1, \psi_2), t) \leftrightarrow$$
$$(\textit{holds}(\psi_2, t) \vee (\textit{holds}(\psi_1, t) \wedge \textit{holds}(\textit{release-past}(\psi_1, \psi_2), t-1))$$

otherwise, there are the following clauses:

$$\textit{holds}(\textit{release-past}(\psi_1, \psi_2), 0) \leftrightarrow (\textit{holds}(\psi_2, 0) \vee \textit{holds}(\psi_1, 0))$$ □

6.3.2 Split Encoding

The previous representation of temporal constraints is fully grounded with respect to fluents and goals. That is, for each quantifier, a new unique predicate is created for each grounding of the variable attached to the quantifier. For example, in the formula $\phi := \forall(?x, type, \psi_1)$, and $\mathtt{D}(type) := \{X_1, ..., X_k\}$, we create propositional variables $\textit{holds}(\psi_1[?x \leftarrow X_1]), ..., \textit{holds}(\psi_1[?x \leftarrow X_k])$. It is possible to use a split representation for the encoding of the temporal constraints with a separate set of predicates for the grounding of variables required by existential quantifiers. It is not possible to split the encoding of universal quantifiers because each variable grounding of the quantifier may require its own copy of the program structure. For example, in the formula $\forall(?x, type_x, \exists(?y, type_y, f(?x, ?y)))$, for each value that $?x$ takes $?y$ may take a different value.

The constraints for this split encoding are the same as those of the flat encoding, except that Schemata 6.3.2, 6.3.7, and 6.3.8 are omitted and replaced with the schemata in this section. Notice that here any formula ψ may contain predicates with arguments that are a mixture of constants and variables. In this setting, argument groundings are represented with predicates of the form $\textit{eq}(?x, X, t)$, which says that variable $?x$ is grounded to constant X at step t.

We have the following constraints.

Existential quantifier: $\psi = \exists(?x, type, \psi_1)$

Schema 6.3.19. For all *holds*$(\exists(?x, type, \psi_1), t)$, there are the following clauses:

$$\mathit{holds}(\exists(?x, type, \psi_1), t) \leftrightarrow \mathit{holds}(\psi_1, t) \qquad \square$$

Schema 6.3.20. For all *holds*$(\exists(?x, type, \psi_1), t)$, there is the following clause:

$$\mathit{holds}(\exists(?x, type, \psi_1), t) \rightarrow \bigvee_{X \in \mathrm{D}(type)} \mathit{eq}(?x, X, t) \qquad \square$$

Fluent: $\psi = f(?x_1, ..., ?x_k)$

Schema 6.3.21. For all $(\mathit{holds}(f(?x_1, ..., ?x_k), t)$, there is the following clause:

$$(\mathit{holds}(f(?x_1, ..., ?x_k), t) \wedge \mathit{eq}(?x_1, X_1, t) \wedge ... \wedge \mathit{eq}(?x_k, X_k, t)) \rightarrow f(X_1, ..., X_k, t) \qquad \square$$

Schema 6.3.22. For all $(\mathit{holds}(f(?x_1, ..., ?x_k), t)$, there is the following clause:

$$(\neg \mathit{holds}(f(?x_1, ..., ?x_k), t) \wedge \mathit{eq}(?x_1, X_1, t) \wedge ... \wedge \mathit{eq}(?x_k, X_k, t)) \rightarrow \neg f(X_1, ..., X_k, t) \qquad \square$$

If $f(X_1, ..., X_k)$ is static, simplify the above constraints with $f(X_1, ..., X_k, t) := \top$. In the case that $f(X_1, ..., X_k) \notin \mathcal{F}^t$ then simplify with $f(X_1, ..., X_k, t) := \bot$. Here $?x_1, ..., ?x_k$ may be either constants or variables – i.e. some arguments may be under the scope of a universal quantifier and therefore be ground.

Goal formula: $\psi = \mathit{goal}(f(x_1, ..., x_k))$

Schema 6.3.23. If $f(X_1, ..., X_k)$ is a goal, there is the following clause:

$$\mathit{holds}(\mathit{goal}(f(x_1, ..., x_k)), t) \vee \neg \mathit{eq}(x_1, X_1, t) \vee ... \vee \neg \mathit{eq}(x_k, X_k, t)$$

otherwise, there is the following clause:

$$\neg \mathit{holds}(\mathit{goal}(f(x_1, ..., x_k)), t) \vee \neg \mathit{eq}(x_1, X_1, t) \vee ... \vee \neg \mathit{eq}(x_k, X_k, t) \qquad \square$$

Here $?x_1, ..., ?x_k$ may be either constants or variables.

6.3.3 Illustrative Example

A planning system has been implemented in C++ that implements the above encoding $\mathcal{T}^{\tau}$. This system takes a planning problem represented in PDDL and, along the lines of COS-P (Chapter 5), first builds a plangraph from the input problem for a given horizon h and then generates a CNF ϕ^{τ} from the encoding $\mathcal{T}^{\tau}$. This CNF is solved using the SAT solver PRECOSAT and a plan, if it exists, is extracted from the resulting valuation. As this work is in its preliminary stages, we do not provide a thorough empirical evaluation with other planning systems. Instead, we present an illustrative example by encoding control knowledge for the SIMPLE-LOGISTICS domain that is described in Appendix A.

The control knowledge we encode is partly based on constraints suggested in Kautz and Selaman [37]. It represents a set of optimality constraints – i.e. constraints that rule out some sub-optimal solutions. We implement the following optimality constraints:

1. If a package is at its goal location, it cannot be picked up:

$$\begin{aligned}&\mathit{holds}(\mathit{always}(\forall(p, package, forall(l, location,\\&\quad \mathit{or}(\mathit{not}(\mathit{and}(at(p,l), \mathit{goal}(at(p,l)))), \mathit{next}(at(p,l)))))))\end{aligned}$$

2. If a package leaves a location, then it is never returned there:

$$\begin{aligned}&\mathit{holds}(\mathit{always}(\forall(p, package, \forall(l, location,\\&\quad \mathit{or}(\mathit{or}(\mathit{not}(at(p,l)), \mathit{next}(at(p,l))), \mathit{next}(\mathit{always}(\mathit{not}(at(p,l)))))))))\end{aligned}$$

3. If a truck drops-off a package it is carrying, it may never pick it up again:

$$\begin{aligned}&\mathit{holds}(\mathit{always}(\forall(t, truck, \forall(p, package,\\&\quad \mathit{or}(\mathit{not}(\mathit{and}(in(p,t), \mathit{next}(\exists(l, location, at(p,l))))),\\&\quad \mathit{next}(\mathit{always}(\mathit{not}(in(p,t)))))))))\end{aligned}$$

Table 6.1 shows the results of encoding two SIMPLE-LOGISTICS problems into CNF with and without the previously described optimality constraints. The table shows the encodings at the step-optimal solution horizon h, and the largest unsatisfiable horizon $h-1$. It also shows the time taken by PRECOSAT to solve the generated CNF at each horizon and cost of the solution plan at h. Finally, the table shows the size of the resulting CNFs after unit propagation was performed. The encodings examined are the flat base encoding for planning-as-SAT as described in Chapter 3, the base encoding with a flat encoding of the above optimality constraints, and the split version of the same encoding.

Encoding	h	t_h	$cost$	v_h	c_h	$h-1$	t_{h-1}	v_{h-1}	c_{h-1}
SIMPLE-LOGISTICS-12									
Base	6	0.320	20	2596	20914	5	0.036	1966	15466
Control-flat	6	0.276	19	5927	26039	5	0.056	4610	19591
Control-split	6	0.316	19	6223	26265	5	0.084	4888	19842
SIMPLE-LOGISTICS-16									
Base	9	12.45	46	14929	161441	8	8.93	12704	136421
Control-flat	9	13.15	48	300689	202085	8	11.50	28291	171712
Control-split	9	11.74	49	33412	201945	8	10.36	28866	171702

Table 6.1: The solution time and encoding size for SIMPLE-LOGISTICS problems 12 and 16 (based on DRIVERLOG problems 12 and 16 resp.). h is the step-optimal solution horizon. t_h is the time taken to solve the generated CNFs on one core of a four core Intel Xenon CPU (3.07Ghz) with 4 GB RAM running Linux at the horizon h. t_{h-1} is the time taken at $h-1$. $cost$ is the cost of the solution plan. v_h and c_h are the number of variables and clauses (resp.) in the generated CNF at h after unit propagation is performed. v_{h-1} and c_{h-1} are the same quantities at $h-1$.

The example results shown in problem Table 6.1 are indicative of results we obtained applying similar optimality constraints to other domains. The constraints often confer a slight advantage in solving efficiency when the instance is satisfiable, and a slight disadvantage when it is not. It is worth noting that stronger constraints than those we have experimented with are quite likely to yield more impact in practice. Here, we are alluding to temporal formula that rule out sub-optimal plans. For example, in the SIMPLE-LOGISTICS domain action-landmark constraints that prevent trucks from performing actions unless they will eventually move a package either to its goal, or to another truck, would likely be useful. Another important application for this planning system, is to find plans that achieve a set of temporally extended goals, or maximally satisfy a set of temporally extended preferences.

6.4 Golog

Golog [40] is a language for procedural agent control based on an extension of the Situation-Calculus [45]. We generate SAT constraints from Golog programs that restrict the set of plans that represent solutions to an underlying planning problem. The SAT encodings we present are based on the transition semantics for ConGolog [22] and the answer-set programming encodings of Son, et al. [69]. We first present encodings of Golog where an *execution trace* maps onto a set of planning steps such that there is *exactly* one *primitive action*, that is one action in the underlying planning domain, executed per step up to the planning horizon h. We then extend this work to encode programs for a novel variant of ConGolog called ParaGolog, where true concurrent action execution is allowed. We present a number of novel encodings for Golog that make use of splitting, both on the grounding of arguments and on steps.

The basic unit of a Golog program is a *complex action*. A complex action is a plan sketch, which represents a totally ordered plan when choices are made for all nondeterministic elements of the action. These choices include the grounding of nondeterministic arguments, the selection of nondeterministic actions, and the number of iterations of nondeterministic loops. Before we define complex actions, we have a renaming construct.

Definition 6.4.1. Let $\delta[?x{\leftarrow}X]$ mean that variable $?x$ is replaced by X in complex action δ. □

This construct is used for grounding variables and also for renaming variables.

Definition 6.4.2. A complex action δ with a sequence of variables $?x_1, ..., ?x_k$ is one of:

- $\delta = a(?y_1, ..., ?y_l)$ – Primitive action: a is an action in the underlying planning problem, where $?y_1, ..., ?y_l$ are free variables in $\{?x_1, ..., ?x_k\}$ or constants from the underlying domain;
- $\delta = ?(\psi)$ – Test: ψ is a temporal formula with fluents from the underlying planning problem which has arguments that are free variables in $\{?x_1, ..., ?x_k\}$ or constants from the underlying domain;
- $\delta = \textit{seq}(\delta_1, \delta_2)$ – Sequence: The complex actions δ_1 and δ_2, which have free variables from $\{?x_1, ..., x_k\}$, are executed in sequence;
- $\delta = \textit{ndet-arg}(?x, type, \delta_1)$ – Nondeterministic argument choice: First, variable $?x \notin \{?x_1, ..., ?x_k\}$. Complex action δ_1 has free variables in $\{?x_1, ..., ?x_k, ?x\}$, and exactly one instance of $\delta_1[?x{\leftarrow}X]$, for $X \in \mathtt{D}(type)$, is executed;
- $\delta = \textit{ndet-act}(\delta_1, \delta_2)$ – Nondeterministic action choice: Exactly one of the complex actions δ_1 or δ_2, which have free variables in $\{?x_1, ..., ?x_k\}$, is executed;
- $\delta = \textit{if}(\psi, \delta_1, \delta_2)$ – Conditional: if temporal formula ψ holds then complex action δ_1 is executed, otherwise δ_2 is executed. Both δ_1 and δ_2 have free variables in $\{?x_1, ..., ?x_k\}$. It is possible for δ_2 to be a $null$ action. In this case if ψ does not hold, nothing happens.

- $\delta = \textit{iter}(\delta_1)$ – Nondeterministic iteration: Complex action δ_1, which has free variables in $\{?x_1, ..., ?x_k\}$, is executed i times, where $i \geq 0$;
- $\delta = \textit{while}(\psi, \delta_1)$ – While loop: δ_1, which has free variables in $\{?x_1, ..., ?x_k\}$, is executed while temporal formula ψ holds.
- $\delta = m(?x_1, ..., ?x_k)$ – Procedure call: m is a procedure with free variables in $?x_1, ..., ?x_k$.

$\square$

Definition 6.4.3. A complex action in which all parameters are constants is said to be *ground*. $\square$

Definition 6.4.4. Let $(m(?x_1, ..., ?x_n), \delta)$ be a procedure with name m, variables $?x_1, ..., ?x_n$, and complex action δ with free variables from $?x_1, ..., ?x_n$. $\square$

Definition 6.4.5. Procedure $(m(?x_1, ..., ?x_n), \delta)$ is *nested* if δ is a procedure call. $\square$

Definition 6.4.6. Procedure $(m(?x_1, ..., ?x_n), \delta)$ is *well defined* if it does not rely on itself. Specifically, if δ and its sub-actions do not refer to m.[2] $\square$

Definition 6.4.7. A Golog program $\Gamma := (R, \delta)$, where:

- R is a set of well-defined, uniquely named, and non-nested procedures; and
- δ is a complex action. $\square$

Our formulation of Golog differs from many existing formulations by having explicit complex actions for while loops and conditionals. In related works it is more common to have while loops and conditions be represented by macros of other complex actions to simplify the presentation of the semantics. The conditional can be represented with the following macro:

$$\textit{if}(\psi, \delta_1, \delta_2) := \textit{ndet-act}(\textit{seq}(\textit{?}(\psi), \delta_1), \textit{seq}(\textit{?}(\neg\psi), \delta_2))$$

Similarly, the while loop can be represented with the following macro:

$$\textit{while}(\psi, \delta) := \textit{seq}(\textit{iter}(\textit{seq}(\textit{?}(\psi), \delta)), \textit{?}(\neg\psi))$$

This approach was not taken here to allow for more compact SAT encodings of golog programs, as described in subsequent sections. While it may be conceptually simpler to omit the

[2]This restriction could potentially be relaxed by using the finite horizon to bound the number of recursive calls made by procedures or by reasoning about the structure of particular procedures. We leave this to future work.

while loop and conditional, the aim of this work is to devise control knowledge formulations and encodings that work well with SAT-based planning systems and, as discussed extensively in Chapter 4, the size of these encoding can be particularly important to solving efficiency.

To illustrate the previous definitions, we give an example Golog program for the SIMPLE-LOGISTICS domain that directs trucks to pick-up and drop-off packages. First, there is the following procedures. The procedure $navigate(?t, ?loc)$ ensures that a truck $?t$ drives to location $?loc$:

```
(navigate(?t, ?loc),
    while(not(at(?t, ?loc)),
        ndet-arg(?l1, location,
            ndet-arg(?l2, location,
                Drive(?t, ?l1, ?l2)
    ))))
```

The next procedure $deliver(?t, ?p)$ ensures that a truck $?t$ drives to the location of a package $?p$, picks it up, and delivers it to its goal location:

```
(deliver(?t, ?p),
    ndet-arg(?l1, location,
        seq(?(at(?p, ?l1)),
            seq(navigate(?t, ?l1),
                seq(Pick-up(?t, ?p, ?l1),
                    ndet-arg(?l2, location,
                        seq(?(goal(at(?p, ?l2))),
                            seq(navigate(?t, ?l2),
                                Drop(?t, ?p, ?l2)
    ))))))))
```

Finally, we have the following complex action δ, which ensures that all packages are delivered:

```
while(∃(?p, package,
  ∃(?l, location,
      and(goal(at(?p, ?l)),
        not(at(?p, ?l))
  )),
      ndet-arg(?p, package,
        ndet-arg(?t, truck,
          deliver(?t, ?p)
  )))
```

A Golog program (R, δ) is interpreted in a pair of state-action trajectories:

$$\langle J_h := s_0 a_0 s_1, ..., a_{h-1} s_h, J_i := s_i a_i s_{i+1}, ..., a_{j-1} s_j \rangle$$

by checking if $\langle J_h, J_i \rangle$ is a *trace* of (R, δ).

Definition 6.4.8. Let $\langle J_h, J_i \rangle \models (R, \delta)$ mean that a pair of trajectories $\langle J_h := s_0 a_0 s_1, ..., a_{h-1} s_h$, $J_i := s_i a_i s_{i+1}, ..., a_{j-1} s_j \rangle$ is a trace of Golog program (R, δ). A pair trajectories $\langle J_h, J_i \rangle$ is a trace of (R, δ) iff any one of the following cases is satisfied:

1. $\delta = a$, where a is an action, $j = i + 1$, and $a_i = a$;
2. $\delta =$ *?*(ψ), where $j = i$ and temporal formula ψ holds in s_i, that is $\langle s_i, J_h \rangle \models \psi$;
3. $\delta =$ *seq*(δ_1, δ_2), there exists an l, where $i \leq l \leq j$, such that $\langle J_h, s_i a_i, ..., s_l \rangle \models (R, \delta_1)$, and $\langle J_h, s_l a_l, ..., s_j \rangle \models (R, \delta_2)$;
4. $\delta =$ *ndet-arg*$(?x, type, \delta_1)$ and $\langle J_h, J_i \rangle \models (R, \delta_1[?x \leftarrow X])$, for some $X \in \mathtt{D}(type)$;
5. $\delta =$ *ndet-act*(δ_1, δ_2) and $\langle J_h, J_i \rangle \models \delta_1$ or $\langle J_h, J_i \rangle \models \delta_2$;
6. $\delta =$ *if*$(\psi, \delta_1, \delta_2)$ and
   - If $\langle s_i, J_h \rangle \models \psi$, then $\langle J_h, J_i \rangle \models (R, \delta_1)$; otherwise
   - $\langle s_i, J_h \rangle \models$ *not*(ψ);

   If $\delta_2 = null$ then $j = i$, otherwise $\langle J_h, J_i \rangle \models (R, \delta_2)$;

7. $\delta = \textit{while}(\psi, \delta_1)$ and
   - $j = i$ and $\langle s_i, J_h \rangle \models \textit{not}(\psi)$; or
   - $\langle s_i, J_h \rangle \models \psi$ and there exists an $l, i < l \leq j$ such that $s_i a_i s_{i+1}, ..., s_l \models (R, \delta_1)$ and $s_l, ..., a_{j-1} s_j \models (R, \delta)$;
8. $\delta = \textit{iter}(\delta_1)$ and
   - $j = i$ or
   - There exists an $l, i < l \leq j$ such that $s_i a_i s_{i+1}, ..., s_l \models (R, \delta_1)$ and $s_l, ..., a_{j-1} s_j \models (R, \delta)$; or
9. $\delta = m(X_1, ..., X_k)$, then the procedure $(m(?x_1, ..., ?x_k), \delta_1) \in R$, and the trajectories $\langle J_h, J_i \rangle \models (R, \delta_1[?x_1 \leftarrow X_1, ..., ?x_k \leftarrow X_k])$. □

We can now recursively check if a pair of trajectories is a trace of a Golog program because the program is finite – i.e. because our initial complex action is finite and we have a set of procedures that are well-defined and non-nested. Note, that our definitions assume that trajectories are consistent with the underlying planning domain.

We now present a flat encoding of the constraints for all Golog constructs and then give an encoding that splits on the groundings of arguments within the scope of quantifiers, followed by encodings that split on steps. The constraints for a Golog program interact with the constraints encoding a planning problem through the simple actions and through the temporal formula. Formally, we define an encoding of a Golog program (R, δ) and a bounded planning problem Π_h as follows.

Definition 6.4.9. Let $\mathcal{T}^\Gamma$ be a function which takes a bounded planning problem Π_h and a Golog program (R, δ) and returns a SAT problem ϕ^Γ. We say that $\mathcal{T}^\Gamma$ reduces Π_h and (R, δ) to SAT and that $\phi^\Gamma := \mathcal{T}^\Gamma(\Pi_h, (R, \delta))$ is an encoding of Π_h and (R, δ) iff a valid plan π can be extracted from every satisfying valuation of ϕ^Γ and for associated trajectory $J_\pi := s_0 a_0 s_1, ..., a_{h-1} s_h$, we have that $\langle J_\pi, J_\pi \rangle \models (R, \delta)$. Finally, we require that ϕ^Γ is unsatisfiable only if there is no such plan. □

6.4.1 Flat Encoding

In the following, we create a propositional variable for each predicate $\textit{trans}(n, \delta, t_1, t_2)$, which means that complex action δ with the unique name n, is executed from step t_1 to step t_2. Here

δ is only required to be ground if it is a primitive action. The following also uses the predicate $\mathit{holds}(\psi, t)$ to mean that temporal formula ψ holds at step t as defined in Section 6.3.1. The predicate $a(X_1, ..., X_k)^t$ refers to the variable for action $a(X_1, ..., X_k)$ at step t in the CNF encoding the underlying planning domain. For a Golog program (R, δ), variables are first created for the complex action δ and then the following schemata are recursively applied to create variables for all sub-actions. In practice, we use the structure of the program and underlying domain to prune a large number of propositional variables. For example, a primitive action that occurs second in a sequence following another primitive action can never be executed at step 0. Taking this reasoning further, it is possible to use the structure of Golog programs to get a minimum and maximum bound on the planning horizon h. The exposition of these issues is left for future work. The underlying planning encoding can also be used to prune the encoding of a Golog program. Specifically, reachability pruning is performed during plangraph generation for the underlying domain and this information can be used when generating Golog programs. In the following constraints, a primitive action a, that does not exist at a step t in the underlying planning domain, is replaced by $\perp$ in the Golog program and the constraints simplified accordingly. Before we present the schemata, there is the following definition.

Definition 6.4.10. Let n be a concatenation operator and $n_1 : n_2$ be the result of concatenating names n_1 and n_2. □

Main Action:

For a Golog program (R, δ) and a horizon h we have the following constraints.

Schema 6.4.1. There is the following clause:

$$\bigvee_{t \in \{0,...,h\}} \mathit{trans}(0, \delta, 0, t)$$ □

Schema 6.4.2. For $t_1, t_2 \in \{0, ..., h\}$, where $t_1 > t_2$, there is the following clause:

$$\neg \mathit{trans}(0, \delta, 0, t_1) \vee \neg \mathit{trans}(0, \delta, 0, t_2)$$ □

To optionally assert that the program has exactly the length h we include the following schema.

Schema 6.4.3.

$$\mathit{trans}(0, \delta, 0, h)$$ □

Primitive Action: $\delta = a(X_1, ..., X_k)$

As we require closed Golog programs, the constraints for primitive actions are only ever applied to ground primitive actions.

Schema 6.4.4. For all $\mathit{trans}(n, a(X_1, ..., X_k), t, t{+}1)$, there is the following clause:

$$\mathit{trans}(n, a(X_1, ..., X_k), t, t{+}1) \rightarrow a(X_1, ..., X_k)^t \qquad \square$$

Test: $\delta = ?(\psi)$

Schema 6.4.5. For all $\mathit{trans}(n, ?(\psi), t, t)$, there is the following clause:

$$\mathit{trans}(n, ?(\psi), t, t) \rightarrow \mathit{holds}(\psi, t) \qquad \square$$

Sequence: $\delta = \mathit{seq}(\delta_1, \delta_2)$

We introduce the auxiliary predicate $\mathit{seq\text{-}m}(n, \delta_1, \delta_2, t_1, t_2, t_3)$ which says that t_2 is the middle of the sequence $\mathit{seq}(\delta_1, \delta_2)$ with name n, which starts at t_1 and ends at t_3.

Schema 6.4.6. For all $\mathit{trans}(n, \mathit{seq}(\delta_1, \delta_2), t_1, t_2)$, there are the following clauses:

$$\mathit{trans}(n, \mathit{seq}(\delta_1, \delta_2), t_1, t_2) \leftrightarrow \bigvee_{t_3 \in \{t_1, ..., t_2\}} \mathit{seq\text{-}m}(n, \delta_1, \delta_2, t_1, t_3, t_2) \qquad \square$$

Schema 6.4.7. For all $\mathit{trans}(n, \mathit{seq}(\delta_1, \delta_2), t_1, t_2)$, $t_3 \in \{t_1, ..., t_2\}$, and $t_4 \in \{t_3{+}1, ..., t_2\}$, there is the following clause:

$$\neg \mathit{seq\text{-}m}(n, \delta_1, \delta_2, t_1, t_3, t_2) \vee \neg \mathit{seq\text{-}m}(n, \delta_1, \delta_2, t_1, t_4, t_2) \qquad \square$$

Schema 6.4.8. For all $\mathit{seq\text{-}m}(n, \delta_1, \delta_2, t_1, t_2, t_3)$, there are the following clauses:

$$\mathit{seq\text{-}m}(n, \delta_1, \delta_2, t_1, t_2, t_3) \leftrightarrow (\mathit{trans}(n{:}0, \delta_1, t_1, t_2) \wedge \mathit{trans}(n{:}1, \delta_2, t_2, t_3)) \qquad \square$$

Schema 6.4.9. For all $\mathit{trans}(n{:}0, \delta_1, t_1, t_2)$, where there exists $\mathit{seq\text{-}m}(n, \delta_1, \delta_2, t_1, t_2, t_3)$, there is the following clause:

$$\mathit{trans}(n{:}0, \delta_1, t_1, t_2) \rightarrow$$
$$\bigvee_{\mathit{trans}(n:1, \delta_2, t_2, t_3),\ \text{where } \exists \mathit{seq\text{-}m}(n, \delta_1, \delta_2, t_1, t_2, t_3)} \mathit{trans}(n{:}1, \delta_2, t_2, t_3) \qquad \square$$

Schema 6.4.10. For all $\mathit{trans}(n\!:\!1, \delta_2, t_2, t_3)$, where there exists $\mathit{seq\text{-}m}(n, \delta_1, \delta_2, t_1, t_2, t_3)$, there is the following clause:

$$\mathit{trans}(n\!:\!1, \delta_2, t_2, t_3) \rightarrow \bigvee_{\mathit{trans}(n:0,\delta_1,t_1,t_2),\ \text{where } \exists \mathit{seq\text{-}m}(n,\delta_1,\delta_2,t_1,t_2,t_3)} \mathit{trans}(n\!:\!0, \delta_1, t_1, t_2) \qquad \square$$

The previous two schemata relate the two sub-sequences δ_1 and δ_2, where for each δ_1 (resp. δ_2) there can be multiple groundings of δ_2 (resp. δ_1).

Nondeterministic argument choice: $\delta = \mathit{ndet\text{-}arg}(?x, \mathit{type}, \delta_1)$

Schema 6.4.11. For all $\mathit{trans}(n, \mathit{ndet\text{-}arg}(?x, \mathit{type}, \delta_1), t_1, t_2)$, there are the following clauses:

$$\mathit{trans}(n, \mathit{ndet\text{-}arg}(?x, \mathit{type}, \delta_1), t_1, t_2) \leftrightarrow \bigvee_{X \in \mathtt{D}(\mathit{type})} \mathit{trans}(n\!:\!X, \delta_1[?x \leftarrow X], t_1, t_2) \qquad \square$$

Schema 6.4.12. For all $\mathit{trans}(n, \mathit{ndet\text{-}arg}(?x, \mathit{type}, \delta_1), t_1, t_2)$, and $X, Y \in \mathtt{D}(\mathit{type}), X \neq Y$, there is the following clause:

$$\neg \mathit{trans}(n\!:\!X, \delta_1[?x \leftarrow X], t_1, t_2) \vee \neg \mathit{trans}(n\!:\!Y, \delta_1[?x \leftarrow Y], t_1, t_2) \qquad \square$$

Nondeterministic action choice: $\delta = \mathit{ndet\text{-}act}(\delta_1, \delta_2)$

Schema 6.4.13. For all $\mathit{trans}(n, \mathit{ndet\text{-}act}(\delta_1, \delta_2), t_1, t_2)$, there is the following clause:

$$\mathit{trans}(n, \mathit{ndet\text{-}act}(\delta_1, \delta_2), t_1, t_2) \rightarrow (\mathit{trans}(n\!:\!0, \delta_1, t_1, t_2) \vee \mathit{trans}(n\!:\!1, \delta_2, t_1, t_2)) \qquad \square$$

Schema 6.4.14. For all $\mathit{trans}(n, \mathit{ndet\text{-}act}(\delta_1, \delta_2), t_1, t_2)$, there is the following clause:

$$\neg \mathit{trans}(n\!:\!0, \delta_1, t_1, t_2) \vee \neg \mathit{trans}(n\!:\!1, \delta_2, t_1, t_2) \qquad \square$$

Schema 6.4.15. For all $\mathit{trans}(n\!:\!0, \delta_1, t_1, t_2)$, where there exists $\mathit{trans}(n, \mathit{ndet\text{-}act}(\delta_1, \delta_2), t_1, t_2)$, there is the following clause:

$$\mathit{trans}(n\!:\!0, \delta_1, t_1, t_2) \rightarrow \bigvee_{\mathit{trans}(n,\mathit{ndet\text{-}act}(\delta_1,\delta_2),t_1,t_2)} \mathit{trans}(n, \mathit{ndet\text{-}act}(\delta_1, \delta_2), t_1, t_2) \qquad \square$$

Schema 6.4.16. For all $\mathit{trans}(n\!:\!1, \delta_2, t_1, t_2)$, where there exists $\mathit{trans}(n, \mathit{ndet\text{-}act}(\delta_1, \delta_2), t_1, t_2)$, there is the following clause:

$$\mathit{trans}(n\!:\!1, \delta_2, t_1, t_2) \rightarrow \bigvee_{\mathit{trans}(n,\mathit{ndet\text{-}act}(\delta_1,\delta_2),t_1,t_2)} \mathit{trans}(n, \mathit{ndet\text{-}act}(\delta_1, \delta_2), t_1, t_2) \qquad \square$$

In the previous two schemata notice that for each grounding of δ_1 (resp. δ_2) there can be multiple groundings of δ_2 (resp. δ_1).

Conditional: $\delta = \mathit{if}(\psi, \delta_1, \delta_2)$

Schema 6.4.17. For all $\mathit{trans}(n, \mathit{if}(\psi, \delta_1, \delta_2), t_1, t_2)$, there is the following clause:

$$(\mathit{trans}(n, \mathit{if}(\psi, \delta_1, \delta_2), t_1, t_2) \wedge holds(\psi, t_1)) \rightarrow \mathit{trans}(n{:}0, \delta_1, t_1, t_2) \qquad \square$$

Schema 6.4.18. For all $\mathit{trans}(n, \mathit{if}(\psi, \delta_1, \delta_2), t_1, t_2)$, if $\delta_2 \neq null$, then there is the following clause:

$$\mathit{trans}(n, \mathit{if}(\psi, \delta_1, \delta_2), t_1, t_2) \wedge \neg holds(\psi, t_1)) \rightarrow \mathit{trans}(n{:}1, \delta_2, t_1, t_2) \qquad \square$$

Schema 6.4.19. For all $\mathit{trans}(n, \mathit{if}(\psi, \delta_1, \delta_2), t_1, t_2)$, if $\delta_2 \neq null$, then there is the following clause:

$$\neg \mathit{trans}(n{:}0, \delta_1, t_1, t_2) \vee \neg \mathit{trans}(n{:}1, \delta_2, t_1, t_2) \qquad \square$$

Schema 6.4.20. For all $\mathit{trans}(n{:}0, \delta_1, t_1, t_2)$, where there exists $\mathit{trans}(n, \mathit{if}(\psi, \delta_1, \delta_2), t_1, t_2)$, there is the clause:

$$\mathit{trans}(n{:}0, \delta_1, t_1, t_2) \rightarrow \bigvee_{\mathit{trans}(n, \mathit{if}(\psi, \delta_1, \delta_2), t_1, t_2)} \mathit{trans}(n, \mathit{if}(\psi, \delta_1, \delta_2), t_1, t_2) \qquad \square$$

Schema 6.4.21. For all $\mathit{trans}(n{:}1, \delta_2, t_1, t_2)$, where there exists $\mathit{trans}(n, \mathit{if}(\psi, \delta_1, \delta_2), t_1, t_2)$, there is the clause:

$$\mathit{trans}(n{:}1, \delta_2, t_1, t_2) \rightarrow \bigvee_{\mathit{trans}(n, \mathit{if}(\psi, \delta_1, \delta_2), t_1, t_2)} \mathit{trans}(n, \mathit{if}(\psi, \delta_1, \delta_2), t_1, t_2) \qquad \square$$

Nondeterministic iteration and while loop: $\delta = \mathit{iter}(\delta_1)$ **or** $\delta = \mathit{while}(\psi, \delta_1)$

In the following, unless otherwise stated, all schemata are used for both $\delta = \mathit{iter}(\delta_1)$ and $\delta = \mathit{while}(\psi, \delta_1)$. We introduce the auxiliary predicate $\mathit{iter\text{-}it}(n, \delta, t_1, t_2)$, which corresponds to an iteration of a loop. Every iteration has at least length one, except for a final terminating iteration on every loop (This assumption is discussed in Section (6.4.4)). We also introduce the predicate $\mathit{iter\text{-}aux}(\psi, \delta, t_1, t_3, t_2)$ which states that t_3 is the middle point of iteration $\mathit{iter\text{-}it}(n, \delta, t_1, t_2)$.

Schema 6.4.22. For all $\mathit{trans}(n, \delta, t_1, t_2)$, there is the following clause:

$$\mathit{trans}(n, \delta, t_1, t_2) \rightarrow \mathit{iter}(n, \delta, t_1, t_2) \qquad \square$$

Schema 6.4.23. For all $\mathit{iter}(n, \delta, t_1, t_2)$, there is the following clause:

$$\mathit{iter}(n, \delta, t_1, t_2) \rightarrow (\mathit{trans}(n, \delta, t_1, t_2) \vee \bigvee_{t_3 \in \{0,...,t_1-1\}} \mathit{iter\text{-}aux}(n, \delta, t_3, t_1, t_2)) \qquad \square$$

Schema 6.4.24. For all $\mathit{iter}(n, \delta, t_1, t_2)$, if $t_2 > t_1$, then there are the following clauses:

$$\mathit{iter}(n, \delta, t_1, t_2) \leftrightarrow \bigvee_{t_3 \in \{t_1+1,...,t_2\}} \mathit{iter\text{-}aux}(n, \delta, t_1, t_3, t_2) \qquad \square$$

Schema 6.4.25. For all $\mathit{iter\text{-}aux}(n, \delta, t_1, t_3, t_2)$, there are the following clauses:

$$\mathit{iter\text{-}aux}(n, \delta, t_1, t_3, t_2) \rightarrow (\mathit{trans}(n, \delta_1, t_1, t_3) \wedge \mathit{iter}(n{:}0, \delta, t_3, t_2)) \qquad \square$$

Schema 6.4.26. For all $\mathit{iter\text{-}aux}(n, \delta, t_1, t_3, t_2)$, and $t_4 \in \{t_3+1, ..., t_2\}$, there is the following clause:

$$\neg \mathit{iter\text{-}aux}(n, \delta, t_1, t_3, t_2) \vee \neg \mathit{iter\text{-}aux}(n, \delta, t_1, t_4, t_2) \qquad \square$$

Schema 6.4.27. For all $\mathit{trans}(n{:}0, \delta_1, t_1, t_3)$, where $\delta = \mathit{iter}(\delta_1)$ or $\delta = \mathit{while}(\psi, \delta_1)$, there is the following clause:

$$\mathit{trans}(n{:}0, \delta_1, t_1, t_3) \rightarrow \bigvee_{\mathit{iter\text{-}aux}(n,\delta,t_1,t_3,t_2)} \mathit{iter\text{-}aux}(n, \delta, t_1, t_3, t_2) \qquad \square$$

Note that for the previous schemata, in the case where $\delta = \mathit{while}(\psi, \delta_1)$, there may be multiple groundings of $\mathit{iter\text{-}aux}(n, \delta, t_1, t_3, t_2)$ for each $\mathit{trans}(\delta_1 : 0, t_1, t_3)$, that is, ψ may have arguments that are not in δ_1.

Schema 6.4.28. For all $\mathit{iter}(n, \mathit{while}(\psi, \delta_1), t, t)$, there is the following clause:

$$\mathit{iter}(n, \mathit{while}(\psi, \delta_1), t, t) \rightarrow \neg holds(\psi, t) \qquad \square$$

Schema 6.4.29. For all $\mathit{iter}(n, \mathit{while}(\psi, \delta_1), t_1, t_2), t_1 > t_2$, there is the following clause:

$$\mathit{iter}(n, \mathit{while}(\psi, \delta_1), t_1, t_2) \rightarrow holds(\psi, t_1) \qquad \square$$

Other constraints:

Finally, we assert that actions in the underlying planning problem are only executed in part of a program trace. Note that each action may be referred to by a number of distinctly named transition predicates.

Schema 6.4.30. For all $a(X_1, ..., X_k)^t \in \mathcal{A}^t$, for $t \in \{0, ..., h-1\}$, there is the following clause:

$$a(X_1, ..., X_k)^t \rightarrow \bigvee_{\mathit{trans}(n,a(X_1,...,X_k),t,t+1)} \mathit{trans}(n, a(X_1, ..., X_k), t, t+1) \qquad \square$$

6.4.2 Split Encoding

The previous encoding of Golog is fully grounded with respect to operator arguments – i.e. for every grounding of the variable $?x$ in a nondeterministic argument choice $\textit{ndet-arg}(?x, type, \delta_1)$, a new *trans* predicate is created for the complex action representing the entire sub-program δ_1. It is possible to encode the argument choice more compactly using a split representation where argument groundings are represented separately from the main program structure.

The constraints for this encoding are the same as in the ground encoding except that the constraints for encoding primitive actions (Schema 6.4.4), planning actions (Schema 6.4.30), and the constraints representing nondeterministic argument grounding (Schemata 6.4.11 and 6.4.12) are omitted and replaced by the following constraints. Additionally, the temporal formulae of test actions may be encoded with the split representation presented in Section 6.3.2, though this is not required and the split encoding of the temporal formulae may be used with a flat encoding of Golog. For this encoding we introduce a predicate $\textit{eq}(?x, X, t)$, which means that variable $?x$ is set to constant X at step t.

Primitive Action: $\delta = a(x_1, ..., x_k)$

Schema 6.4.31. For all $\textit{trans}(n, a(?x_1, ..., ?x_k), t, t+1)$, and all groundings $X_1, ..., X_k$ of the variables $?x_1, ..., ?x_k$, where $X_1 \in \mathtt{D}(type_1), ..., X_k \in \mathtt{D}(type_k)$ for $a(?x_1 - type_1, ..., ?x_k - type_k)$, there are the following clauses:

$$(\textit{trans}(n, a(?x_1, ..., ?x_k), t, t+1) \wedge \textit{eq}(n, ?x, X_1, t) \wedge ... \wedge \textit{eq}(?x, X_k, t)) \leftrightarrow \textit{trans}(n, a(X_1, ..., X_k), t, t+1) \qquad \square$$

There are variables created for the predicate $\textit{trans}(n, a(?x_1, ..., ?x_k), t, t+1)$ and the grounded version of the predicate $\textit{trans}(n, a(X_1, ..., X_k), t, t+1)$. Some of the variables $?x_1, ..., ?x_k$ may be constants. If all are constants then there is no need to create the variable for the grounded predicate. In this latter case, the original variable is used in the following schemata.

Schema 6.4.32. For $\textit{trans}(n, a(X_1, ...X_k), t, t+1)$, there is the following clause:

$$\textit{trans}(n, a(X_1, ...X_k), t, t+1) \rightarrow a(X_1, ..., X_k)^t \qquad \square$$

Nondeterministic argument choice: $\delta = \textit{ndet-arg}(x, \textit{type}, \delta_1)$

Here we introduce the predicate $\textit{eqr}(?x, X, t_1, t_2)$ which means that $?x = X$ from step t_1 to step t_2.

Schema 6.4.33. For all $\textit{trans}(n, \textit{ndet-arg}(?x, \textit{type}, \delta_1), t_1, t_2)$, there are the following clauses:

$$\textit{trans}(n, \textit{ndet-arg}(?x, \textit{type}, \delta_1), t_1, t_2) \leftrightarrow \textit{trans}(n{:}0, \delta_1[?x \leftarrow ?x_n], t_1, t_2) \qquad \square$$

Schema 6.4.34. For all $\textit{trans}(n, \textit{ndet-arg}(?x, \textit{type}, \delta_1), t, t)$, there is the following clause:

$$\textit{trans}(n, \textit{ndet-arg}(?x, \textit{type}, \delta_1), t, t) \rightarrow \bigvee_{X \in D(\textit{type})} \textit{eq}(?x_n, X, t) \qquad \square$$

Schema 6.4.35. For all $\textit{trans}(n, \textit{ndet-arg}(?x, \textit{type}, \delta_1), t_1, t_2)$, where $t_1 > t_2$, there is the following clause:

$$\textit{trans}(n, \textit{ndet-arg}(?x, \textit{type}, \delta_1), t_1, t_2) \rightarrow \bigvee_{X \in D(\textit{type})} \textit{eqr}(?x_n, X, t_1, t_2) \qquad \square$$

Schema 6.4.36. For all $\textit{eqr}(?x, X, t_1, t_2)$, there are the following clauses:

$$\textit{eqr}(?x, X, t_1, t_2) \rightarrow \bigwedge_{t_3 \in \{t_1, \ldots, t_2 - 1\}} \textit{eq}(?x, X, t_3) \qquad \square$$

Schema 6.4.37. For all $\textit{eq}(?x, X_1, t)$ and $\textit{eq}(?x, X_1, t)$, $X_1 \neq X_2$:

$$\neg \textit{eq}(?x, X_1, t) \vee \neg \textit{eq}(?x, X_2, t) \qquad \square$$

6.4.3 Splitting on Steps

The previous encodings contain a number of predicates that have more than one step argument. It is possible to decompose each of these predicates into a set of predicates where each mentions no more than one step – i.e. it is possible to split on steps. If we are to split on steps then, due to the problem of interference, which is explored for regular planning actions in Section 4.1.3, at most one instance of any single *trans* predicate for a complex action δ, with name n, can be true in any valid trace. In the previous flat encoding, loop actions allow more than one *trans* predicate for an action δ with name n to be true at the same step. In particular, the iterations of a loop are represented by predicates that differ only in the values of their steps. To prevent these variables from interfering in the split case we copy and rename these variables for each possible loop iteration. The number of copies we need to create is bounded by the minimum possible length of each loop iteration and the horizon h. We use the program structure to compute bounds on the minimum possible length of loop iterations. The exposition of this method is left for future work.

First, in this section we present an encoding that splits only on steps and then, in the following section, one that splits on both steps and argument groundings. Even with the copies due to loops, the following encodings are more compact than the previously described encodings which do not split on steps. In the following, we use predicates *trans-s*(n, δ, t), and *trans-e*(n, δ, t) to mean that complex action δ begins at step t and ends at step t respectively. For a Golog program (R, δ) and a horizon h we have the following constraints.

Schema 6.4.38. There is the following clause:

$$\textit{trans-s}(0, \delta, 0) \qquad \square$$

Schema 6.4.39. There is the following clause:

$$\bigvee_{t \in \{0,...,h\}} \textit{trans-e}(0, \delta, t) \qquad \square$$

Schema 6.4.40. For $t_1, t_2 \in \{0, ..., h\}, t_1 > t_2$, there is the following clause:

$$\neg\textit{trans-e}(0, \delta, t_1) \vee \neg\textit{trans-e}(0, \delta, t_2) \qquad \square$$

To assert that the program has exactly the length h we have the following schemata.

Schema 6.4.41. There is the following clause:

$$\textit{trans-e}(0, \delta, h) \qquad \square$$

Primitive action: $\delta = a(X_1, ..., X_k)$

Schema 6.4.42. For all *trans-s*$(n, a(X_1, ..., X_k), t)$, $t < h$, there are the following clauses:

$$\textit{trans-s}(n, a(X_1, ..., X_k), t) \leftrightarrow \textit{trans-e}(n, a(X_1, ..., X_k), t{+}1) \qquad \square$$

Schema 6.4.43. For all *trans-s*$(n, a(X_1, ..., X_k), t)$, $t < h$, there is the following clause:

$$\textit{trans-s}(n, a(X_1, ..., X_k), t) \rightarrow a(X_1, ..., X_k)^t \qquad \square$$

Test: $\delta = ?(\psi)$

Schema 6.4.44. For all *trans-s*$(n, ?(\psi), t)$, there are the following clauses:

$$\textit{trans-s}(n, ?(\psi), t) \leftrightarrow \textit{trans-e}(n, ?(\psi), t) \qquad \square$$

Schema 6.4.45. For all *trans-s*$(n, ?(\psi), t)$, there are the following clauses:

$$\textit{trans-s}(n, ?(\psi), t) \rightarrow holds(\psi, t) \qquad \square$$

Sequence: $\delta = \textit{seq}(\delta_1, \delta_2)$

We introduce a predicate *trans-m*$(n, \textit{seq}(\delta_1, \delta_2), t)$, which means that the middle point of sequence *seq*(δ_1, δ_2) is at step t.

Schema 6.4.46. For all *trans-s*$(n, \textit{seq}(\delta_1, \delta_2), t_1)$, there is the following clause:

$$\textit{trans-s}(n, \textit{seq}(\delta_1, \delta_2), t_1) \rightarrow \bigvee_{t_2 \in \{t_1, \ldots, h\}} \textit{trans-m}(n, \textit{seq}(\delta_1, \delta_2), t_2) \qquad \square$$

Schema 6.4.47. For all *trans-s*$(n, \textit{seq}(\delta_1, \delta_2), t_1)$, there is the following clause:

$$\textit{trans-s}(n, \textit{seq}(\delta_1, \delta_2), t_1) \rightarrow \textit{trans-s}(n{:}0, \delta_1, t_1) \qquad \square$$

Schema 6.4.48. For all *trans-e*$(n, \textit{seq}(\delta_1, \delta_2), t_1)$, there is the following clause:

$$\textit{trans-e}(n, \textit{seq}(\delta_1, \delta_2), t_1) \rightarrow \bigvee_{t_2 \in \{0, \ldots, t_1\}} \textit{trans-m}(n, \textit{seq}(\delta_1, \delta_2), t_2) \qquad \square$$

Schema 6.4.49. For all *trans-e*$(n, \textit{seq}(\delta_1, \delta_2), t_1)$, there is the following clause:

$$\textit{trans-e}(n, \textit{seq}(\delta_1, \delta_2), t_1) \rightarrow \textit{trans-e}(n{:}1, \delta_2, t_1) \qquad \square$$

Schema 6.4.50. For all *trans-m*$(n, \textit{seq}(\delta_1, \delta_2), t_1)$, there are the following clauses:

$$\textit{trans-m}(n, \textit{seq}(\delta_1, \delta_2), t_1) \rightarrow (\textit{trans-e}(n{:}0, \delta_1, t_1) \wedge \textit{trans-s}(n{:}1, \delta_2, t_1)) \qquad \square$$

Schema 6.4.51. For all pairs *trans-m*$(n, \textit{seq}(\delta_1, \delta_2), t_1)$ and *trans-m*$(n, \textit{seq}(\delta_1, \delta_2), t_2)$, where $t_1 < t_2$, there is the following clause:

$$\neg\textit{trans-m}(n, \textit{seq}(\delta_1, \delta_2), t_1) \vee \neg\textit{trans-m}(n, \textit{seq}(\delta_1, \delta_2), t_2) \qquad \square$$

Schema 6.4.52. For all *trans-s*$(n{:}0, \delta_1, t_1)$, where there exists *trans-s*$(n, \textit{seq}(\delta_1, \delta_2), t_1)$, there is the following clause:

$$\textit{trans-s}(n{:}0, \delta_1, t_1) \rightarrow \bigvee_{\textit{trans-s}(n,\textit{seq}(\delta_1,\delta_2),t_1)} \textit{trans-s}(n, \textit{seq}(\delta_1, \delta_2), t_1) \qquad \square$$

Schema 6.4.53. For all *trans-e*$(n{:}0, \delta_1, t_1)$, where there exists *trans-m*$(n, \textit{seq}(\delta_1, \delta_2), t_1)$, there is the following clause:

$$\textit{trans-e}(n{:}0, \delta_1, t_1) \rightarrow \bigvee_{\textit{trans-m}(n,\textit{seq}(\delta_1,\delta_2),t_1)} \textit{trans-m}(n, \textit{seq}(\delta_1, \delta_2), t_1) \qquad \square$$

Schema 6.4.54. For all *trans-s*$(n{:}1, \delta_2, t_1)$, where there exists *trans-m*$(n, \textit{seq}(\delta_1, \delta_2), t_1)$, there is the following clause:

$$\textit{trans-s}(n{:}1, \delta_2, t_1) \rightarrow \bigvee_{\textit{trans-m}(n,\textit{seq}(\delta_1,\delta_2),t_1)} \textit{trans-m}(n, \textit{seq}(\delta_1, \delta_2), t_1) \qquad \square$$

Schema 6.4.55. For all *trans-e*$(n{:}1, \delta_2, t_1)$, where there exists *trans-e*$(n, \textit{seq}(\delta_1, \delta_2), t_1)$, there is the following clause:

$$\textit{trans-e}(n{:}1, \delta_2, t_1) \rightarrow \bigvee_{\textit{trans-e}(n,\textit{seq}(\delta_1,\delta_2),t_1)} \textit{trans-e}(n, \textit{seq}(\delta_1, \delta_2), t_1) \qquad \square$$

Nondeterministic argument choice: $\delta = \textit{ndet-arg}(?x, \textit{type}, \delta_1)$

Schema 6.4.56. For all *trans-s*$(n, \textit{ndet-arg}(?x, \textit{type}, \delta_1), t)$, there is the following clause:

$$\textit{trans-s}(n, \textit{ndet-arg}(?x, \textit{type}, \delta_1), t) \rightarrow \bigvee_{X \in \mathtt{D}(\textit{type})} \textit{eq}(?x_n, X) \qquad \square$$

Schema 6.4.57. For all *trans-s*$(n, \textit{ndet-arg}(?x, \textit{type}, \delta_1), .)$ and $X_1, X_2 \in \mathtt{D}(\textit{type})$, $X_1 \neq X_2$, there is the following clause:

$$\neg \textit{eq}(?x_n, X_1) \vee \neg \textit{eq}(?x_n, X_2) \qquad \square$$

In the previous schema, the period in the place of the step argument indicates that argument is ignored when selecting predicates – i.e. any grounding of the argument is considered a match as is the ungrounded variable. That is, if two predicates are identical except for the step argument, then only one of them is used.

Schema 6.4.58. For all $\textit{trans-s}(n, \textit{ndet-arg}(?x, type, \delta_1), t)$ and $X \in \mathtt{D}(type)$, there are the following clauses:

$$(\textit{trans-s}(n, \textit{ndet-arg}(?x, type, \delta_1), t) \wedge \textit{eq}(?x_n, X)) \leftrightarrow \textit{trans-s}(n{:}X, \delta_1[?x{\leftarrow}X], t) \qquad \square$$

Schema 6.4.59. For all $\textit{trans-e}(n, \textit{ndet-arg}(?x, type, \delta_1), t)$ and $X \in \mathtt{D}(type)$, there are the following clauses:

$$(\textit{trans-e}(n, \textit{ndet-arg}(?x, type, \delta_1), t) \wedge \textit{eq}(?x_n, X)) \leftrightarrow \textit{trans-e}(n{:}X, \delta_1[?x{\leftarrow}X], t) \qquad \square$$

Nondeterministic action choice: $\delta = \textit{ndet-act}(\delta_1, \delta_2)$

We introduce auxiliary predicate $\textit{choice}(n, \delta_1)$, which means that complex action δ_1 is selected for the argument choice with the unique name n.

Schema 6.4.60. For all $\textit{trans-s}(n, \textit{ndet-act}(\delta_1, \delta_2), .)$, there is the following clause:

$$\neg\textit{choice}(n, \delta_1) \vee \neg\textit{choice}(n, \delta_2) \qquad \square$$

Schema 6.4.61. For all $\textit{trans-s}(n, \textit{ndet-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$\textit{trans-s}(n, \textit{ndet-act}(\delta_1, \delta_2), t) \rightarrow (\textit{choice}(n, \delta_1) \vee \textit{choice}(n, \delta_2)) \qquad \square$$

Schema 6.4.62. For all $\textit{trans-s}(n, \textit{ndet-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$(\textit{trans-s}(n, \textit{ndet-act}(\delta_1, \delta_2), t) \wedge \textit{choice}(n, \delta_1)) \rightarrow \textit{trans-s}(n{:}0, \delta_1, t) \qquad \square$$

Schema 6.4.63. For all $\textit{trans-s}(n, \textit{ndet-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$(\textit{trans-s}(n, \textit{ndet-act}(\delta_1, \delta_2), t) \wedge \textit{choice}(n, \delta_2)) \rightarrow \textit{trans-s}(n{:}1, \delta_2, t) \qquad \square$$

Schema 6.4.64. For all $\textit{trans-e}(n, \textit{ndet-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$(\textit{trans-e}(n, \textit{ndet-act}(\delta_1, \delta_2), t) \wedge \textit{choice}(n, \delta_1)) \rightarrow \textit{trans-e}(n{:}0, \delta_1, t) \qquad \square$$

Schema 6.4.65. For all $\textit{trans-e}(n, \textit{ndet-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$(\textit{trans-e}(n, \textit{ndet-act}(\delta_1, \delta_2), t) \wedge \textit{choice}(n, \delta_2)) \rightarrow \textit{trans-e}(n{:}1, \delta_2, t) \qquad \square$$

Schema 6.4.66. For all *trans-s*$(n{:}0, \delta_1, t)$, where there exists *trans-s*$(n, \textit{ndet-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$\textit{trans-s}(n{:}0, \delta_1, t) \rightarrow \textit{choice}(n, \delta_1) \qquad \square$$

Schema 6.4.67. For all *trans-s*$(n{:}0, \delta_1, t)$, where there exists *trans-s*$(n, \textit{ndet-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$\textit{trans-s}(n{:}0, \delta_1, t) \rightarrow \bigvee_{\textit{trans-s}(n, \textit{ndet-act}(\delta_1, \delta_2), t)} \textit{trans-s}(n, \textit{ndet-act}(\delta_1, \delta_2), t) \qquad \square$$

Schema 6.4.68. For all *trans-e*$(n{:}0, \delta_1, t)$, where there exists *trans-e*$(n, \textit{ndet-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$\textit{trans-e}(n{:}0, \delta_1, t) \rightarrow \bigvee_{\textit{trans-e}(n, \textit{ndet-act}(\delta_1, \delta_2), t)} \textit{trans-e}(n, \textit{ndet-act}(\delta_1, \delta_2), t) \qquad \square$$

Schema 6.4.69. For all *trans-s*$(n{:}1, \delta_2, t)$, where there exists *trans-s*$(n, \textit{ndet-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$\textit{trans-s}(n{:}1, \delta_2, t) \rightarrow \textit{choice}(n, \delta_2) \qquad \square$$

Schema 6.4.70. For all *trans-s*$(n{:}1, \delta_2, t)$, where there exists *trans-s*$(n, \textit{ndet-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$\textit{trans-s}(n{:}1, \delta_2, t) \rightarrow \bigvee_{\textit{trans-s}(n, \textit{ndet-act}(\delta_1, \delta_2), t)} \textit{trans-s}(n, \textit{ndet-act}(\delta_1, \delta_2), t) \qquad \square$$

Schema 6.4.71. For all *trans-e*$(n{:}1, \delta_2, t)$, where there exists *trans-e*$(n, \textit{ndet-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$\textit{trans-e}(n{:}1, \delta_2, t) \rightarrow \bigvee_{\textit{trans-e}(n, \textit{ndet-act}(\delta_1, \delta_2), t)} \textit{trans-e}(n, \textit{ndet-act}(\delta_1, \delta_2), t) \qquad \square$$

Conditional: $\delta = \textit{if}(\psi, \delta_1, \delta_2)$

Here we introduce auxiliary predicate *choice*(n, δ_1), which means that δ_1 is selected for the conditional with the unique name n.

Schema 6.4.72. For all $trans(n, \textit{if}(\psi, \delta_1, \delta_2), .)$, there is the following clause:

$$\neg\textit{choice}(n, \delta_1) \vee \neg\textit{choice}(n, \delta_2) \qquad \square$$

Schema 6.4.73. For all $\mathit{trans}\text{-}s(n, \mathit{if}(\psi, \delta_1, \delta_2), t)$, there is the following clause:

$$(\mathit{trans}\text{-}s(n, \mathit{if}(\psi, \delta_1, \delta_2), t) \wedge \mathit{holds}(\psi, t)) \rightarrow \mathit{choice}(n, \delta_1) \qquad \square$$

Schema 6.4.74. For all $\mathit{trans}\text{-}s(n, \mathit{if}(\psi, \delta_1, \delta_2), t)$, there is the following clause:

$$(\mathit{trans}\text{-}s(n, \mathit{if}(\psi, \delta_1, \delta_2), t) \wedge \mathit{choice}(n, \delta_1)) \rightarrow \mathit{trans}\text{-}s(n{:}0, \delta_1, t) \qquad \square$$

Schema 6.4.75. For all $\mathit{trans}\text{-}s(n, \mathit{if}(\psi, \delta_1, \delta_2), t)$, there is the following clause:

$$(\mathit{trans}\text{-}s(n, \mathit{if}(\psi, \delta_1, \delta_2), t) \wedge \neg \mathit{holds}(\delta, t)) \rightarrow \mathit{choice}(n, \delta_2) \qquad \square$$

Schema 6.4.76. For all $\mathit{trans}\text{-}s(n, \mathit{if}(\psi, \delta_1, \delta_2), t)$, there is the following clause:

$$(\mathit{trans}\text{-}s(n, \mathit{if}(\psi, \delta_1, \delta_2), t) \wedge \mathit{choice}(n, \delta_2)) \rightarrow \mathit{trans}\text{-}s(n{:}1, \delta_2, t) \qquad \square$$

Schema 6.4.77. For all $\mathit{trans}\text{-}e(n, \mathit{if}(\psi, \delta_1, \delta_2), t)$, there is the following clause:

$$(\mathit{trans}\text{-}e(n, \mathit{if}(\psi, \delta_1, \delta_2), t) \wedge \mathit{choice}(n, \delta_1)) \rightarrow \mathit{trans}\text{-}e(n{:}0, \delta_1, t) \qquad \square$$

Schema 6.4.78. For all $\mathit{trans}\text{-}e(n, \mathit{if}(\psi, \delta_1, \delta_2), t)$, there is the following clause:

$$(\mathit{trans}\text{-}e(n, \mathit{if}(\psi, \delta_1, \delta_2), t) \wedge \mathit{choice}(n, \delta_2)) \rightarrow \mathit{trans}\text{-}e(n{:}1, \delta_2, t) \qquad \square$$

Schema 6.4.79. For all $\mathit{trans}\text{-}s(n{:}0, \delta_1, t)$, where there exists $\mathit{trans}\text{-}s(n, \mathit{if}(\psi, \delta_1, \delta_2), t)$, there is the following clause:

$$\mathit{trans}\text{-}s(n{:}0, \delta_1, t) \rightarrow \mathit{choice}(n, \delta_1) \qquad \square$$

Schema 6.4.80. For all $\mathit{trans}\text{-}s(n{:}0, \delta_1, t)$, where there exists $\mathit{trans}\text{-}s(n, \mathit{if}(\psi, \delta_1, \delta_2), t)$, there is the following clause:

$$\mathit{trans}\text{-}s(n{:}0, \delta_1, t) \rightarrow \bigvee_{\mathit{trans}\text{-}s(n, \mathit{if}(\psi, \delta_1, \delta_2), t)} \mathit{trans}\text{-}s(n, \mathit{if}(\psi, \delta_1, \delta_2), t) \qquad \square$$

Schema 6.4.81. For all $\mathit{trans}\text{-}e(n{:}0, \delta_1, t)$, where there exists $\mathit{trans}\text{-}e(n, \mathit{if}(\psi, \delta_1, \delta_2), t)$, there is the following clause:

$$\mathit{trans}\text{-}e(n{:}0, \delta_1, t) \rightarrow \bigvee_{\mathit{trans}\text{-}e(n, \mathit{if}(\psi, \delta_1, \delta_2), t)} \mathit{trans}\text{-}e(n, \mathit{if}(\psi, \delta_1, \delta_2), t) \qquad \square$$

Schema 6.4.82. For all $\mathit{trans\text{-}s}(n\!:\!1, \delta_2, t)$, where there exists $\mathit{trans\text{-}s}(n, \mathit{if}(\psi, \delta_1, \delta_2), t)$, there is the following clause:

$$\mathit{trans\text{-}s}(n\!:\!1, \delta_2, t) \rightarrow \mathit{choice}(n, \delta_2)$$ □

Schema 6.4.83. For all $\mathit{trans\text{-}s}(n\!:\!1, \delta_2, t)$, where there exists $\mathit{trans\text{-}s}(n, \mathit{if}(\psi, \delta_1, \delta_2), t)$, there is the following clause:

$$\mathit{trans\text{-}s}(n\!:\!1, \delta_2, t) \rightarrow \bigvee_{\mathit{trans\text{-}s}(n, \mathit{if}(\psi, \delta_1, \delta_2), t)} \mathit{trans\text{-}s}(n, \mathit{if}(\psi, \delta_1, \delta_2), t)$$ □

Schema 6.4.84. For all $\mathit{trans\text{-}e}(n\!:\!1, \delta_2, t)$, where there exists $\mathit{trans\text{-}e}(n, \mathit{if}(\psi, \delta_1, \delta_2), t)$, there is the following clause:

$$\mathit{trans\text{-}e}(n\!:\!1, \delta_2, t) \rightarrow \bigvee_{\mathit{trans\text{-}e}(n, \mathit{if}(\psi, \delta_1, \delta_2), t)} \mathit{trans\text{-}e}(n, \mathit{if}(\psi, \delta_1, \delta_2), t)$$ □

If $\delta_2 = null$, then we have the following schema:

Schema 6.4.85. For all $\mathit{trans\text{-}s}(n\!:\!1, null, t)$, there is the following clause:

$$\mathit{trans\text{-}s}(n\!:\!1, null, t) \leftrightarrow \mathit{trans\text{-}e}(n\!:\!1, null, t)$$ □

Nondeterministic iteration and while loop: $\delta = \mathit{iter}(\delta_1)$ or $\delta = \mathit{while}(\psi, \delta_1)$

As in the non-split case, unless otherwise stated, all schemata are used for both $\delta = \mathit{iter}(\delta_1)$ and $\delta = \mathit{while}(\psi, \delta_1)$. Here, iterations of loops are labelled explicitly with a number. A while loop has at most $N = h$ iterations. We can obtain better bounds on N by considering the program structure and planning domain. The details of this are left for future work. As in the flat encoding, there is a zero length iteration terminating every loop. We introduce the predicate $\mathit{iter}(n, \delta, i, t)$, which means that iteration i of the loop δ, with name n, starts at t. The predicate $\mathit{iter\text{-}aux}(n, \delta, i, t)$ means that the action of iteration i of the loop δ with name n ends at t and iteration $i+1$ starts at t.

Schema 6.4.86. For all $\mathit{trans\text{-}s}(n, \delta, t_1)$, there are the following clauses:

$$\mathit{trans\text{-}s}(n, \delta, t_1) \leftrightarrow \mathit{iter}(n, \delta, 0, t_1)$$ □

Schema 6.4.87. For all $\mathit{trans\text{-}e}(n, \delta, t_2)$, where $t_2 < t_1$, there is the following clause:

$$\mathit{trans\text{-}s}(n, \delta, t_1) \rightarrow \neg \mathit{trans\text{-}e}(n, \delta, t_2)$$ □

Schema 6.4.88. For all *iter*(n, δ, i, t_1), where $i < N$, there is the following clause:

$$\textit{iter}(n, \delta, i, t_1) \rightarrow (\textit{trans-e}(n, \delta, t_1) \vee \textit{trans-s}(n{:}i, \delta_1, t_1)) \qquad \square$$

Schema 6.4.89. For all *iter*(n, δ, i, t_1), where $i < N$, there is the following clause:

$$\textit{iter}(n, \delta, i, t_1) \rightarrow (\textit{trans-e}(n, \delta, t_1) \vee \bigvee_{t_2 \in \{t_1+1, \ldots, h\}} \textit{iter-aux}(n, \delta, i, t_2)) \qquad \square$$

Schema 6.4.90. For all *iter*(n, δ, i, t_1), where $i < N$ and $\delta = \textit{iter}(\delta_1)$, there is the following clause:

$$\textit{trans-s}(n{:}i, \delta_1, t_1) \rightarrow \textit{iter}(n, \textit{iter}(\delta_1), i, t_1) \qquad \square$$

Schema 6.4.91. For all *iter*(n, δ, N, t_1), there is the following clause:

$$\textit{iter}(n, \delta, N, t_1) \rightarrow \textit{trans-e}(n, \delta, t_1) \qquad \square$$

Schema 6.4.92. For all *iter-aux*(n, δ, i, t), where $i < N$, there is the following clause:

$$\textit{iter-aux}(n, \delta, i, t) \rightarrow \textit{trans-e}(n{:}i, \delta_1, t) \qquad \square$$

Schema 6.4.93. For all *iter-aux*(n, δ, i, t), where $i < N$, there are the following clauses:

$$\textit{iter-aux}(n, \delta, i, t) \leftrightarrow \textit{iter}(n, \delta, i{+}1, t) \qquad \square$$

Schema 6.4.94. For all *iter-aux*(n, δ, i, t), where $i < N$, if $\delta = \textit{iter}(\delta_1)$, there is the following clause:

$$\textit{trans-e}(n{:}i, \delta_1, t) \rightarrow \textit{iter-aux}(n, \delta, i, t) \qquad \square$$

Schema 6.4.95. For all *iter-aux*(n, δ, i, t), where $i < N$, and all *trans-e*(n, δ, t_2), where $t_2 < t_1$, there is the following clause:

$$\textit{iter-aux}(n, \delta, i, t) \rightarrow \neg\textit{trans-e}(n, \delta, t_2) \qquad \square$$

Schema 6.4.96. For all *iter*$(n, \textit{while}(\psi, \delta_1), i, t)$, there is the following clause:

$$(\textit{iter}(n, \textit{while}(\psi, \delta_1), i, t) \wedge \textit{holds}(\psi, t)) \rightarrow \neg\textit{trans-e}(n, \textit{while}(\psi, \delta_1), t) \qquad \square$$

Schema 6.4.97. For all $\textit{iter}(n, \textit{while}(\psi, \delta_1), i, t)$, there is the following clause:

$$(\textit{iter}(n, \textit{while}(\psi, \delta_1), i, t) \wedge \neg \textit{holds}(\psi, t)) \rightarrow \textit{trans-e}(n, \textit{while}(\psi, \delta_1), t) \qquad \square$$

Schema 6.4.98. For all $\textit{trans-s}(n : i, \delta_1, t)$, where there exists $\textit{iter}(n, \textit{while}(\psi, \delta_1), i, t)$, there is the following clause:

$$\textit{trans-s}(n{:}i, \delta_1, t) \rightarrow \bigvee_{\textit{iter}(n, \textit{while}(\psi, \delta_1), i, t)} \textit{iter}(n, \textit{while}(\psi, \delta_1), i, t) \qquad \square$$

Schema 6.4.99. For all $\textit{trans-e}(n{:}i, \delta_1, t)$, where there exists $\textit{iter-aux}(n, \textit{while}(\psi, \delta_1), i, t)$, there is the following clause:

$$\textit{trans-e}(n{:}i, \delta_1, t) \rightarrow \bigvee_{\textit{iter-aux}(n, \textit{while}(\psi, \delta_1), i, t)} \textit{iter-aux}(n, \textit{while}(\psi, \delta_1), i, t) \qquad \square$$

Finally, we need a set of additional constraints to assert that actions from the underlying planning problem are only executed as a part of a valid program trace.

Schema 6.4.100. For all $a(X_1, ..., X_k)^t \in \mathcal{A}^t$, for $t \in \{0, ..., h\}$, there is the following clause:

$$a(X_1, ..., X_k)^t \rightarrow \bigvee_{\textit{trans-s}(n, a(X_1, ..., X_k), t)} \textit{trans-s}(n, a(X_1, ..., X_k), t) \qquad \square$$

We now present an encoding that splits on both steps and on argument groundings. In the previous encoding, for each grounding X of a variable $?x$ in a nondeterministic argument choice $\textit{ndet-arg}(?x, type, \delta)$, a copy of δ, $\delta[?x \leftarrow X]$ is created. Along the lines of the standard split encoding, this encoding does not make these copies. Instead it separates the grounding of operator arguments from the main control structure. To do this we introduce a predicate $\textit{eq}(?x, X)$ which asserts that argument $?x$ is grounded to constant X.

The constraints for this encoding are the same as in the previous encoding except that the schemata for encoding primitive actions (Schemata 6.4.31 and 6.4.32) and the schemata representing argument grounding (Schemata 6.4.56, 6.4.57, 6.4.58, and 6.4.59) are omitted and replaced by the schemata that follow.

Primitive action: $\delta = a(?x_1, ..., x_k)$

Schema 6.4.101. For all *trans-s*$(n, a(?x_1, ..., ?x_k), t)$, where $t < h$, there are the following clauses:

$$\textit{trans-s}(n, a(?x_1, ..., ?x_k), t) \leftrightarrow \textit{trans-e}(n, a(?x_1, ..., ?x_k), t{+}1) \qquad \Box$$

Schema 6.4.102. For all *trans-s*$(n, a(?x_1, ..., ?x_k), t)$, where step $t < h$, and all groundings $X_1, ..., X_k$ of the variables $?x_1, ..., ?x_k$, where $X_1 \in \mathtt{D}(\textit{type}_1), ..., X_k \in \mathtt{D}(\textit{type}_k)$ for the action $a(?x_1 - \textit{type}_1, ..., ?x_k - \textit{type}_k)$, there are the following clauses:

$$(\textit{trans-s}(n, a(?x_1, ..., x_k), t) \wedge \textit{eq}(?x_1, X_1) \wedge ... \wedge \textit{eq}(?x_k, X_k)) \leftrightarrow \textit{trans-s}(n, a(X_1, ..., X_k), t)$$

$\Box$

Schema 6.4.103. For all *trans-s*$(n, a(X_1, ..., X_k), t)$, there is the following clause:

$$\textit{trans-s}(n, a(X_1, ..., X_k), t) \rightarrow a(X_1, ..., X_k)^t \qquad \Box$$

Nondeterministic argument choice: $\delta = \textit{ndet-arg}(?x, \textit{type}, \delta_1)$

Schema 6.4.104. For all *trans-s*$(n, \textit{ndet-arg}(?x, \textit{type}, \delta_1), .)$, and $X_1, X_2 \in \mathtt{D}(\textit{type})$, where $X_1 \neq X_2$, there is the following clause:

$$\neg\textit{eq}(?x_n, X_1) \vee \neg\textit{eq}(?x_n, X_2) \qquad \Box$$

Schema 6.4.105. For all *trans-s*$(n, \textit{ndet-arg}(?x, \textit{type}, \delta_1), t)$, there is the following clause:

$$\textit{trans-s}(n, \textit{ndet-arg}(?x, \textit{type}, \delta_1), t) \rightarrow \bigvee_{X \in \mathtt{D}(\textit{type})} \textit{eq}(?x_n, X) \qquad \Box$$

Schema 6.4.106. For all *trans-s*$(n, \textit{ndet-arg}(?x, \textit{type}, \delta_1), t)$, there is the following clause:

$$\textit{trans-s}(n, \textit{ndet-arg}(?x, \textit{type}, \delta_1), t) \leftrightarrow \textit{trans-s}(n{:}0, \delta_1[x_n {\leftarrow} X], t) \qquad \Box$$

Schema 6.4.107. For all *trans-e*$(n, \textit{ndet-arg}(?x, \textit{type}, \delta_1), t)$, there are the following clauses:

$$\textit{trans-e}(n, \textit{ndet-arg}(?x, \textit{type}, \delta_1), t) \leftrightarrow \textit{trans-e}(n{:}0, \delta_1[x_n {\leftarrow} X], t) \qquad \Box$$

6.4.4 Zero Length Loop Iterations

There is a subtle difficulty with the encoding of loops regarding iterations of length zero. The definition of a trace (Definition 6.4.8) restricts while loop and nondeterministic iteration actions to iterations with length ≥ 1. If a loop could have iterations of length zero, then there is the possibility of an unbounded number of iterations of this loop running. As we are using a finite CNF encoding of Golog programs, and because we require *trans* predicates to be fixed to a rigid sequence of steps, we require an upper bound on the number of possible iterations of any loop. Contrast this with the case where an iteration is of length $k \geq 1$ where we can bound the number of iterations with the planning horizon $\lfloor h/k \rfloor$.

Later, in Section 6.4.5 we allow concurrent action execution. In that case the restriction to loop iterations of length ≥ 1 is non-trivial, as there may be valid and interesting program structures that contingently have zero length loop iterations. For example, consider a program with the complex action $\delta = \mathit{while}(\psi_1, \mathit{if}(\psi_2, \delta_1, null))$. In our formalisation of Golog, in the serial case the only way that a loop iteration can be of length zero is if it consists only of one or more tests, $null$ actions from conditionals, or loops. Zero length iterations of loops do not change the underlying planning domain – i.e. no planning actions are executed in them. We can use this fact to eliminate *trivial* loops and to identify loops that may contingently have zero length iterations.

Definition 6.4.11. Let a loop complex action δ (where $\delta = \mathit{iter}(\delta_1)$ or $\delta = \mathit{while}(\psi_1, \delta_1)$) be called *trivial* if its iterations are always of length zero. □

A trivial nondeterministic iteration can place no restriction on the set of valid program traces and as such is replaced by a $null$ action. For non-trivial nondeterministic iterations, we can bound the number of iterations using the planning horizon and consider the loop to have stopped when the first zero length iteration occurs. This can be done precisely because the SAT solver could have chosen *not* to include the zero length iteration.

For while loops the situation is complicated by the fact that a loop iteration must run if the relevant condition holds. If we have a trivial while loop $\delta = \mathit{while}(\psi, \delta_1)$, then this is replaced by $?(\mathit{not}(\psi))$ because if the loop were to start then it could never end, preventing a trajectory where it occurred from being a valid trace. For non-trivial while loops that may contingently have zero length loop iterations, the constraints we have described force the loop to have iterations of length one or greater if the condition holds. If the iterations are forced to be of length zero when the condition holds, then the instance will be unsatisfiable. This is acceptable, as in this case the loop would not terminate and the underlying trajectory that caused the situation could never be a trace of the program.

6.4.5 Concurrent Action Execution

The language ConGolog [22] extends Golog to allow concurrency. ConGolog includes a number of complex actions to introduce concurrency into action execution. These include the concurrent execution of a number of complex actions, with and without associated execution priorities, concurrent iteration where loop iterations run concurrently, and interrupts that trigger whenever a condition holds. ConGolog has an *interleaved concurrency semantics*.

Definition 6.4.12. In an *interleaved concurrency semantics* for ConGolog, if there are two program traces running in parallel, then a valid transition steps one of the traces while leaving the other unchanged. □

We describe two concurrent actions that can be used to introduce concurrency into our formulation of Golog. First, we have the ConGolog complex action that permits a set of complex actions to run in parallel. Second, we have a complex action that permits each grounding of a nondeterministic argument choice to be run in parallel. These complex actions use a novel *parallel concurrency semantics*. We call the language that results *ParaGolog* for Parallel Concurrent Golog.

Definition 6.4.13. In a *parallel concurrency semantics* for ConGolog, if there are two program traces running in parallel, then a valid transition does one of the following:

1. Steps one of the traces while leaving the other unchanged; or
2. Steps both traces in parallel. □

We do not explicitly enforce that transitions respect the mutex constraints of the underlying planning domain. Mutex is enforced externally (in the SAT encoding for the underlying planning domain, for example) and only trajectories which respect mutex constraints are considered to be traces.

The concurrent iteration complex action of ConGolog was not implemented due to the fact that it potentially allows an unbounded number of instances of a complex action to be run in parallel. In our case, with a finite horizon and a finite planning domain, we could bound the number of concurrent iterations of each concurrent loop, but this potentially requires the possibility of an impractically large number of iterations to be encoded. Instead, we introduce a constraint that allows concurrency over the grounding of nondeterministic argument choices. This captures much of the intuition behind concurrency in planning domains. For example, in a program for a SIMPLE-LOGISTICS domain that allows the selection, first of a truck, and then of movement locations, concurrency should intuitively be introduced over the selection of the truck, but not over its movement locations. That is, we can move more than one truck concurrently but we cannot concurrently move a truck to or from more than one location.

Continuing the previous example, the concurrent iteration complex action of ConGolog allows the concurrent selection of more than one value for each location involved in a truck movement. This can coherently be used in a plan, specifically where the duplicate concurrent iterations are executed in sequence. If desired, this behaviour can be replicated using our complex action by introducing a nondeterministic iteration inside the concurrent choice of groundings for a variable representing a truck. It can be argued that the approach of using concurrent argument groundings is more intuitive from a modelling perspective, as it more easily allows the restriction of concurrency to the selection of particular planning objects, generally those that exhibit independent agency or autonomy.

One area for possible future extension is the introduction of a language construct functionally equivalent to semaphores. This would allow the restriction of certain resources (nondeterministic argument choices) to individual concurrent traces. For example, if trucks are selected concurrently to deliver packages, then it seems natural to assume that individual packages be restricted to a single truck.

Formally, the complex actions we introduce are as follows:

Definition 6.4.14. For ParaGolog a complex action is defined according to Definition 6.4.2 with the two additional cases:

- $\delta = \textit{con-arg}(?x, type, \delta_1)$ – Concurrent argument choice: Variable $?x \notin \{?x_1, ..., ?x_k\}$ and complex action δ_1 has free variables in $\{?x_1, ..., ?x_k, ?x\}$ and any number of instances of $\delta_1[?x \leftarrow X]$, for $X \in \mathtt{D}(type)$, are executed concurrently; and
- $\delta = \textit{con-act}(\delta_1, \delta_2)$ – Concurrent action execution: Complex actions δ_1 and δ_2, with free variables from $\{?x_1, ..., x_k\}$, are executed concurrently. $\square$

Definition 6.4.15. A ParaGolog program $\Gamma = (R, \Delta)$ where:

- R is a set of *coherent* procedures – i.e. well-defined, uniquely named, and non-nested; and
- Δ is a set of complex actions that are to be executed in parallel. □

The Golog program for the SIMPLE-LOGISTICS domain in Section 6.4 can be modified to allow trucks to act concurrently, by replacing the main complex action with the following:

$$\begin{aligned}
&\textit{while}(\exists(?p, package,\\
&\quad \exists(?l, location,\\
&\qquad \textit{and}(\textit{goal}(\texttt{At}(?p, ?l)),\\
&\qquad\quad \textit{not}(\texttt{At}(?p, ?l))\\
&\quad)),\\
&\qquad \textit{con-arg}(?t, truck,\\
&\qquad\quad \textit{ndet-arg}(?p, package,\\
&\qquad\qquad deliver(?t, ?p)\\
&\quad)))
\end{aligned}$$

In the previously described encodings, which do not allow concurrency, a program trace is fixed rigidly to a sequence of steps. That is, every action must begin as early as possible after the preceding action. This rules out stepping one concurrent trace while leaving others unchanged. In the encodings that follow this restriction is relaxed. In particular, the execution of primitive actions and test actions are modified to represent execution envelopes. Additionally, as in Son et al. [69] conditionals and while loops need to be *synchronised*. That is, it needs to be enforced that their conditions hold at the time of executing the *first* primitive actions or checking the test condition for the *first* test actions of each iteration of either the loop or of the selected branch of the conditional. This modification is required to avoid a situation where two or more conditionals or loops trigger concurrently and require the execution of incompatible actions.

Definition 6.4.16. Where δ^n represents complex action δ with name n, let the set of *first actions* $first(\delta^n)$ be a set defined recursively as follows:

- If $\delta = a(X_1, ..., X_k)$, then the $first(\delta^n) := \{\delta^n\}$;
- If $\delta = ?(\psi)$, then $first(\delta^n) := \{\delta^n\}$;
- If $\delta = \textit{ndet-act}(\delta_1, \delta_2)$ or $\delta = \textit{con-act}(\delta_1, \delta_2)$, then $first(\delta^n) := \{first(\delta_1^{n:0}) \cup first(\delta_2^{n:1})\}$;

- If $\delta =$ *ndet-arg*$(x, type, \delta_1)$ or $\delta =$ *con-arg*$(?x, type, \delta_1)$, then if argument groundings are not split $first(\delta^n) := first(\delta_1^n)$, otherwise $first(\delta^n) := \{first(\delta_1^{n:X}) | X \in D(x)\}$;
- If $\delta =$ *seq*(δ_1, δ_2), then $first(\delta^n) := first(\delta_1^{n:0})$;
- If $\delta =$ *if*$(\psi, \delta_1, \delta_2)$, then $first(\delta^n) := first(\delta_1^{n:0}) \cup first(\delta_2^{n:1})$. If $\delta_2 = null$, then $first(\delta^n) := \{first(\delta_1^{n:0}) \cup ?(\textit{not}(\psi))^n\}$;
- If $\delta =$ *iter*(δ_1), then $first(\delta^n) := \{first(\delta_1^{n:0}) \cup \delta_{iter}^n\}$. Here δ_{iter}^n is a special action representing the zero length transition of the loop. This special action covers the case where there are no loop iterations;
- If $\delta =$ *while*(ψ, δ_1), then $first(\delta^n) := \{first(\delta_1^{n:0}) \cup ?(not(\psi))^n\}$. The test covers the case where there are no loop iterations. □

A ParaGolog program (R, Δ) is interpreted in a pair of state action trajectories that allow parallel action execution:

$$\langle J_h := s_0 a_{0,0}, ..., a_{0,k_0} s_1, ..., a_{h-1,0} a_{h-1,k_{h-1}} s_h, J_i := s_i a_{i,0}, ..., a_{i,k_0} s_{i+1}, ..., a_{j-1,0} a_{j-1,k_{j-1}} s_j \rangle$$

by checking if $\langle J_h, J_i \rangle$ is a trace of (R, Δ).

Definition 6.4.17. Let $\langle J_h, J_i \rangle \models (R, \Delta)$ mean that the following pair of trajectories $\langle J_h := s_0 a_{0,0}, ..., a_{0,k_0} s_1, ..., a_{h-1,0} a_{h-1,k_{h-1}} s_h,\ J_i := s_i a_{i,0}, ..., a_{i,k_0} s_{i+1}, ..., a_{j-1,0} a_{j-1,k_{j-1}} s_j \rangle$ is a trace of the ParaGolog program (R, Δ). A pair of trajectories $\langle J_h, J_i \rangle$ is a trace of (R, Δ) iff, for all complex actions $\delta \in \Delta$, one of the following cases is satisfied:

1. $\delta = a$, where a is an action and $a \in J_i$;
2. $\delta = ?(\psi)$, where ψ holds in a state $s_l \in J$, that is $\exists s_l \in J_i$, such that $\langle s_l, J_h \rangle \models \psi$;
3. $\delta =$ *seq*(δ_1, δ_2) and there exists an l, where $i \leq l \leq j$, such that $\langle J_h, s_i a_i, ..., s_l \rangle \models (R, \delta_1)$ and $\langle J_h, s_l a_l, ..., s_j \rangle \models (R, \delta_2)$;
4. $\delta =$ *ndet-arg*$(?x, type, \delta_1)$ and $\langle J_h, J_i \rangle \models (R, \delta_1[?x \leftarrow X])$, for some $X \in \mathtt{D}(type)$;
5. $\delta =$ *ndet-act*(δ_1, δ_2) and $\langle J_h, J_i \rangle \models \delta_1$ or $\langle J_h, J_i \rangle \models \delta_2$;
6. $\delta =$ *if*$(\psi, \delta_1, \delta_2)$ and one of the following holds:

- $\langle J_h, J_i \rangle$ is a trace of (R, δ_1) and ψ holds when the *first* action of δ_1 is executed – i.e. there exists l, $i \leq l \leq j$, such that the *first* action $a \in a_{l,0}, ..., a_{l,k_l}$ and $\langle s_l, J_h \rangle \models \psi$; or
- $\langle J_h, J_i \rangle$ is a trace of (R, δ_2) and $not(\psi)$ holds when the *first* action of δ_2 is executed; or
- $\delta_2 = null$ and $i = j$ and $\langle s_i, J_h \rangle \models$ *not*(ψ);

7. $\delta =$ *while*(ψ, δ_1) and
   - $j = i$ and $\langle s_i, J_h \rangle \models$ *not*(ψ); or
   - There exists an $l, i < l \leq j$ such that $s_i a_i s_{i+1}, ..., s_l \models (R, \delta_1)$ and $s_l, ..., a_{j-1} s_j \models (R, \delta)$ and there exists an $m, i \leq m \leq l$ and the *first* action of δ_1, $a \in a_{m,0}, ..., a_{m,k_m}$ and $\langle s_m, J_h \rangle \models \psi$;
8. $\delta =$ *iter*(δ_1) and
   - $j = i$; or
   - There exists an $l, i < l \leq j$ such that $s_i a_i s_{i+1}, ..., s_l \models (R, \delta_1)$ and $s_l, ..., a_{j-1} s_j \models (R, \delta)$;
9. $\delta = m(X_1, ..., X_k)$ and the procedure $(m(?x_1, ..., ?x_k), \delta_1) \in R$ and the trajectories $\langle J_h, J_i \rangle \models (R, \delta_1[?x_1 \leftarrow X_1, ..., ?x_k \leftarrow X_k])$;
10. $\delta =$ *con-act*(δ_1, δ_2) and $\langle J_h, J_i \rangle$ is a trace of $(R, \{\delta_1, \delta_2\})$; or
11. $\delta =$ *con-arg*$(?x, type, \delta_1)$ and $\langle J_h, J_i \rangle$ is a trace of $(R, \delta_1[?x \leftarrow X])$ for all $X \in \mathtt{D}(type)$.

□

We can now recursively check if trajectory is a trace of a ParaGolog program for the same reasons as in the non-concurrent case. Formally, we define an encoding of a ParaGolog program (R, Δ) and a bounded planning problem Π_h as follows.

Definition 6.4.18. Let $\mathcal{T}^{P\Gamma}$ be a function which takes a bounded planning problem Π_h and a ParaGolog program (R, Δ) and returns a SAT problem. We say that $\mathcal{T}^{P\Gamma}$ reduces Π_h and (R, Δ) to SAT and that $\phi^{P\Gamma} := \mathcal{T}^{P\Gamma}(\Pi_h, (R, \Delta))$ is an encoding of Π_h and (R, Δ) iff a valid plan π can be extracted from every satisfying valuation of $\phi^{P\Gamma}$ and for the associated trajectory $J_\pi :=$

$s_0 a_{0,0}, ..., a_{0,k_0} s_1, ..., a_{h-1,0} a_{h-1,k_{h-1}} s_h$, we have that $\langle J_\pi, J_\pi \rangle \models (R, \Delta)$. Finally we require that $\phi^{P\Gamma}$ is unsatisfiable only if there is no such plan. □

We now present an encoding for ParaGolog. We only present schemata for an encoding that splits on steps as it is generally more compact than a similar encoding which does not. The encoding we present is based on the encodings for Golog presented previously. The schemata that we present are appropriate for both encodings presented in Section 6.4.3, the one that splits on argument groundings and the one that does not. In the following, constraints are introduced for the two new program constructs. Additionally, the following schemata are omitted and replaced by the constraints that follow:

- Primitive action: Schemata 6.4.42 and 6.4.43 in the encoding that splits only on steps and Schemata 6.4.101, 6.4.102, and 6.4.103 in the fully split encoding;
- Test: Schemata 6.4.44 and 6.4.45 for both encodings; and
- Planning actions are part of a trace: Schema 6.4.100 for both encodings.

Finally, we will discuss how to encode the synchornisation of the tests of while loops and conditionals.

Primitive Action: $\delta = a(?x_1, ..., ?x_k)$

The parameters $?x_1, ..., ?x_k$ in δ can be either constants or variables. In the ground case they will be constants and in the split case those arguments that come from standard argument choices will be variables, while those which come from concurrent argument choices will be constants.

Schema 6.4.108. For all *trans-s*$(n, a(?x_1, ..., ?x_k), t_1)$, there is the following clause:

$$\textit{trans-s}(n, a(?x_1, ..., ?x_k), t_1) \rightarrow \bigvee_{t_2 \in \{t_1, ..., h-1\}} \textit{trans-e}(n, a(?x_1, ..., ?x_k), t_2) \qquad \square$$

Schema 6.4.109. For all *trans-e*$(n, a(?x_1, ..., ?x_k), t_1)$, there is the following clause:

$$\textit{trans-e}(n, a(?x_1, ..., ?x_k), t_1) \rightarrow \bigvee_{t_2 \in \{0, ..., t_1\}} \textit{trans-s}(n, a(?x_1, ..., ?x_k), t_2) \qquad \square$$

Schema 6.4.110. For all *trans-s*$(n, a(?x_1, ..., ?x_k), t_1)$, and *trans-e*$(n, a(?x_1, ..., ?x_k), t_2)$, where $t_2 > t_1$, and each grounding $X_1, ..., X_j$ of all variables $\{Y_1, ..., Y_j\}$ from $\{?x_1, ..., ?x_k\}$, there is the following clause:

$$\begin{gathered}(\textit{trans-s}(n, a(?x_1, ..., ?x_k), t_1) \wedge \textit{trans-e}(n, a(?x_1, ..., ?x_k), t_2) \wedge ?y_1{=}X_1 \wedge ... \wedge ?y_j{=}X_j) \\ \rightarrow \bigvee\nolimits_{t_3 \in \{t_1, ..., t_2-1\}} \textit{trans}(n, a(X_1, ..., X_k), t_3)\end{gathered} \qquad \square$$

Schema 6.4.111. For all $\mathit{trans}(n, a(X_1, ..., X_k), t_1)$, there is the following clause:

$$\mathit{trans}(n, a(X_1, ..., X_k), t_1) \rightarrow a(X_1, ..., X_k)^t \qquad \square$$

Schema 6.4.112. For all $\mathit{trans}(n, a(X_1, ..., X_k), t_1)$, there is the following clause:

$$\mathit{trans}(n, a(X_1, ..., X_k), t_1) \rightarrow \bigvee_{t_2 \in \{0,...,t_1\}} \mathit{trans\text{-}s}(n, a(X_1, ..., X_k), t_2) \qquad \square$$

Schema 6.4.113. For all $\mathit{trans}(n, a(X_1, ..., X_k), t_1)$, there is the following clause:

$$\mathit{trans}(n, a(X_1, ..., X_k), t_1) \rightarrow \bigvee_{t_2 \in \{t_1+1,...,h\}} \mathit{trans\text{-}e}(n, a(X_1, ..., X_k), t_2) \qquad \square$$

Test: $\delta = ?(\psi)$

Schema 6.4.114. For all $\mathit{trans\text{-}s}(n, ?(\psi), t_1)$, there is the following clause:

$$\mathit{trans\text{-}s}(n, ?(\psi), t_1) \rightarrow \bigvee_{t_2 \in \{t_1,...,h\}} \mathit{trans\text{-}e}(n, ?(\psi), t_2) \qquad \square$$

Schema 6.4.115. For all $\mathit{trans\text{-}e}(n, ?(\psi), t_1)$, there is the following clause:

$$\mathit{trans\text{-}e}(n, ?(\psi), t_1) \rightarrow \bigvee_{t_2 \in \{0,...,t_1\}} \mathit{trans\text{-}s}(n, ?(\psi), t_2) \qquad \square$$

Schema 6.4.116. For all $\mathit{trans\text{-}s}(n, ?(\psi), t_1)$ and $\mathit{trans\text{-}e}(n, ?(\psi), t_2)$, where $t_1 \leq t_2$, there is the following clause:

$$(\mathit{trans\text{-}s}(n, ?(\psi), t_1) \wedge \mathit{trans\text{-}e}(n, ?(\psi), t_2)) \rightarrow \bigvee_{t_3 \in \{t_1,...,t_2\}} \mathit{trans}(n, ?(\psi), t_3) \qquad \square$$

Schema 6.4.117. For all $\mathit{trans}(n, ?(\psi), t)$, there is the following clause:

$$\mathit{trans}(n, ?(\psi), t) \rightarrow \mathit{holds}(\psi, t) \qquad \square$$

Schema 6.4.118. For all $\mathit{trans}(n, ?(\psi), t_1)$, there is the following clause:

$$\mathit{trans}(n, ?(\psi), t_1) \rightarrow \bigvee_{t_2 \in \{0,...,t_1\}} \mathit{trans\text{-}s}(n, ?(\psi), t_2) \qquad \square$$

Schema 6.4.119. For all $\mathit{trans}(n, ?(\psi), t_1)$, there is the following clause:

$$\mathit{trans}(n, ?(\psi), t_1) \rightarrow \bigvee_{t_2 \in \{t_1,...,h\}} \mathit{trans\text{-}e}(n, ?(\psi), t_2) \qquad \square$$

Concurrent action execution: $\delta = \textit{con-act}(\delta_1, \delta_2)$

Schema 6.4.120. For all $\textit{trans-s}(n, \textit{con-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$\textit{trans-s}(n, \textit{con-act}(\delta_1, \delta_2), t) \rightarrow \textit{trans-s}(n{:}0, \delta_1, t) \qquad \square$$

Schema 6.4.121. For all $\textit{trans-s}(n, \textit{con-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$\textit{trans-s}(n, \textit{con-act}(\delta_1, \delta_2), t) \rightarrow \textit{trans-s}(n{:}1, \delta_2, t) \qquad \square$$

Schema 6.4.122. For all $\textit{trans-e}(n, \textit{con-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$\textit{trans-e}(n, \textit{con-act}(\delta_1, \delta_2), t) \rightarrow \textit{trans-e}(n{:}0, \delta_1, t) \qquad \square$$

Schema 6.4.123. For all $\textit{trans-e}(n, \textit{con-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$\textit{trans-e}(n, \textit{con-act}(\delta_1, \delta_2), t) \rightarrow \textit{trans-e}(n{:}1, \delta_2, t) \qquad \square$$

Schema 6.4.124. For all $\textit{trans-s}(n{:}0, \delta_1, t)$, where there exists $\textit{trans-s}(n, \textit{con-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$\textit{trans-s}(n{:}0, \delta_1, t) \rightarrow \bigvee_{\textit{trans-s}(n, \textit{con-act}(\delta_1, \delta_2), t)} \textit{trans-s}(n, \textit{con-act}(\delta_1, \delta_2), t) \qquad \square$$

Schema 6.4.125. For all $\textit{trans-e}(n{:}0, \delta_1, t)$, where there exists $\textit{trans-e}(n, \textit{con-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$\textit{trans-e}(n{:}0, \delta_1, t) \rightarrow \bigvee_{\textit{trans-e}(n, \textit{con-act}(\delta_1, \delta_2), t)} \textit{trans-e}(n, \textit{con-act}(\delta_1, \delta_2), t) \qquad \square$$

Schema 6.4.126. For all $\textit{trans-s}(n{:}1, \delta_2, t)$, where there exists $\textit{trans-s}(n, \textit{con-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$\textit{trans-s}(n{:}1, \delta_2, t) \rightarrow \bigvee_{\textit{trans-s}(n, \textit{con-act}(\delta_1, \delta_2), t)} \textit{trans-s}(n, \textit{con-act}(\delta_1, \delta_2), t) \qquad \square$$

Schema 6.4.127. For all $\textit{trans-e}(n{:}1, \delta_2, t)$, where there exists $\textit{trans-e}(n, \textit{con-act}(\delta_1, \delta_2), t)$, there is the following clause:

$$\textit{trans-e}(n{:}1, \delta_2, t) \rightarrow \bigvee_{\textit{trans-e}(n, \textit{con-act}(\delta_1, \delta_2), t)} \textit{trans-e}(n, \textit{con-act}(\delta_1, \delta_2), t) \qquad \square$$

Concurrent argument choice: $\delta = \textit{con-arg}(?x, \textit{type}, \delta_1)$

Schema 6.4.128. For all $\textit{trans-s}(n, \textit{con-arg}(?x, \textit{type}, \delta_1), t_1)$, and all $X \in \mathtt{D}(\textit{type})$, there is the following clause:

$$\textit{trans-s}(n{:}X, \delta_1[?x{\leftarrow}X], t_1) \rightarrow \bigvee_{t_2 \in \{0,...,t_1\}} \textit{trans-s}(n, \textit{con-arg}(?x, \textit{type}, \delta_1), t_2) \qquad \square$$

Schema 6.4.129. For all $\textit{trans-s}(n, \textit{con-arg}(?x, \textit{type}, \delta_1), t_1)$, all $X \in \mathtt{D}(\textit{type})$, and all $t_2 \in \{t_1 + 1, ..., h\}$, there is the following clause:

$$\neg\textit{trans-s}(n{:}X, \delta_1[?x{\leftarrow}X], t_1) \vee \neg\textit{trans-s}(n{:}X, \delta_1[?x{\leftarrow}X], t_2) \qquad \square$$

Schema 6.4.130. For all $\textit{trans-e}(n, \textit{con-arg}(?x, \textit{type}, \delta_1), t_1)$ and all $X \in \mathtt{D}(\textit{type})$, there is the following clause:

$$\textit{trans-e}(n{:}X, \delta_1[?x{\leftarrow}X], t_1) \rightarrow \bigvee_{t_2 \in \{t_1,...,h\}} \textit{trans-e}(n, \textit{con-arg}(?x, \textit{type}, \delta_1), t_2) \qquad \square$$

Schema 6.4.131. For all $\textit{trans-e}(n, \textit{con-arg}(?x, \textit{type}, \delta_1), t_1)$, all $X \in \mathtt{D}(\textit{type})$, and all $t_2 \in \{t_1 + 1, ..., h\}$: there is the following clause:

$$\neg\textit{trans-e}(n{:}X, \delta_1[?x{\leftarrow}X], t_1) \vee \neg\textit{trans-e}(n{:}X, \delta_1[?x{\leftarrow}X], t_2) \qquad \square$$

We also have the following additional constraints for the program.

Schema 6.4.132. For all $a(X_1, ...X_k) \in \mathcal{A}^t$ for $t \in \{0, ..., h-1\}$, we have the following clause:

$$a(X_1, ..., X_k)^t \rightarrow \bigvee_{\textit{trans}(n, a(X_1,...,X_k), t)} \textit{trans}(n, a(X_1, ..., X_k), t) \qquad \square$$

We now have to deal with the synchronising of conditional and while loops. To do this we no longer use the existing schemata that enforce the test conditions and instead introduce schemata that use the previous definition of *first actions* to enforce the conditions. In particular for a complex action δ we omit the following constraints:

- If $\delta = \textit{if}(\psi, \delta_1, \delta_2)$: Schema 6.4.73 and if $\delta_2 \neq \textit{null}$ Schema 6.4.75; and
- For while loops: Schemata 6.4.96 and 6.4.97.

For conditionals, where we have an action $\delta^n = \mathit{if}(\psi, \delta_1, \delta_2)$, we have a set of first actions $\mathit{first}(\delta^n)$ as defined in Definition 6.4.16. Each first action of δ^n is ether a primitive action, a test action or a nondeterministic iteration action with zero length. We have the following schemata for δ^n and each $\delta_i^{n'} \in \mathit{first}(\delta^n)$.
Where $\delta_i^{n'} = a(X_1, ..., X_k)$, or $\delta_i^{n'} = ?(\psi)$:

Schema 6.4.133. For all $\mathit{trans\text{-}s}(n, \mathit{if}(\psi, \delta_1, \delta_2), t_1)$ and all $\mathit{trans}(n, \delta_i, t_2)$, where $t_2 \geq t_1$, there is the following clause:

$$(\mathit{trans\text{-}s}(n, \mathit{if}(\psi, \delta_1, \delta_2), t_1) \wedge \mathit{trans}(n, \delta_i, t_2) \wedge \mathit{holds}(\psi, t_2)) \rightarrow \mathit{choice}(n, \delta_1) \qquad \square$$

If $\delta_2 \neq null$ the we have the following schema.

Schema 6.4.134. For all $\mathit{trans\text{-}s}(n, \mathit{if}(\psi, \delta_1, \delta_2), t_1)$, for all $\mathit{trans}(n, \delta_i, t_2)$, where $t_2 \geq t_1$, there is the following clause:

$$(\mathit{trans\text{-}s}(n, \mathit{if}(\psi, \delta_1, \delta_2), t_1) \wedge \mathit{trans}(n, \delta_i, t_2) \wedge \neg\mathit{holds}(\psi, t_2)) \rightarrow \mathit{choice}(n, \delta_2) \qquad \square$$

Where $\delta_i^{n'} = \mathit{iter}(\delta_j)$, we have the following schema if $\delta_2 \neq null$.

Schema 6.4.135. For all $\mathit{trans\text{-}s}(n, \mathit{if}(\psi, \delta_1, \delta_2), t_1)$, for all $\mathit{trans}(n, \mathit{iter}(\delta_j, t_2), t_2)$, where $t_2 \geq t_1$, there is the following clause:

$$\begin{gathered}(\mathit{trans\text{-}s}(n, \mathit{if}(\psi, \delta_1, \delta_2), t_1) \wedge \mathit{trans\text{-}s}(n, \mathit{iter}(\delta_j, t_2), t_2) \wedge \\ \mathit{trans\text{-}e}(n, \mathit{iter}(\delta_j, t_2), t_2) \wedge \neg\mathit{holds}(\psi, t_2)) \rightarrow \mathit{choice}(n, \delta_2)\end{gathered} \qquad \square$$

For while loops, where we have an action $\delta^n = \mathit{while}(\psi, \delta_1)$, we similarly have a set of first actions $\mathit{first}(\delta^n)$ and again each first action of δ^n is ether a primitive action, a test action or a nondeterministic iteration. We then have the following schemata for δ and each $\delta_i^{n'} \in \mathit{first}(\delta^n)$.
Where $\delta_i^{n'} = a(X_1, ..., X_k)$, or $\delta_i^{n'} = ?(\psi)$:

Schema 6.4.136. For all $\mathit{iter}(n, \mathit{while}(\psi, \delta_1), k, t_1)$, and all $\mathit{trans}(n, \delta_i, t_2)$, where $t_2 \geq t_1$, there is the following clause:

$$(\mathit{iter}(n, \mathit{while}(\psi, \delta_1), k, t_1) \wedge \mathit{trans}(n, \delta_i, t_2) \wedge \mathit{holds}(\psi, t_2)) \rightarrow \neg\mathit{trans\text{-}e}(n, \mathit{while}(\psi, \delta_1), t_1) \square$$

Schema 6.4.137. For all $\mathit{iter}(n, \mathit{while}(\psi, \delta_1), k, t_1)$, and all $\mathit{trans}(n, \delta_i, t_2)$, where $t_2 \geq t_1$, there is the following clause:

$$(\mathit{iter}(n, \mathit{while}(\psi, \delta_1), k, t_1) \wedge \mathit{trans}(n, \delta_i, t_2) \wedge \neg\mathit{holds}(\psi, t_2)) \rightarrow \mathit{trans\text{-}e}(n, \mathit{while}(\psi, \delta_1), t_1) \square$$

Where $\delta_i^{n'} = \mathit{iter}(\delta_j)$, we have the following schema.

Schema 6.4.138. For all $\mathit{iter}(n, \mathit{while}(\psi, \delta_1), k, t_1)$, for all $\mathit{trans}(n, \mathit{iter}(\delta_j, t_2), t_2)$, where $t_2 \geq t_1$, there is the following clause:

$$(\mathit{iter}(n, \mathit{while}(\psi, \delta_1), k, t_1) \wedge \mathit{trans\text{-}s}(n, \mathit{iter}(\delta_j, t_2), t_2) \wedge$$
$$\mathit{trans\text{-}e}(n, \mathit{iter}(\delta_j, t_2), t_2) \wedge \neg \mathit{holds}(\psi, t_2)) \rightarrow \neg \mathit{trans\text{-}e}(n, \mathit{while}(\psi, \delta_1), t_1) \qquad \square$$

The encodings presented here have been tested on a number of problems. In the future we intend to provide a proof of their correctness and a thorough empirical evaluation.

6.5 HTN Planning

A Hierarchical Task Network (HTN) [64] consists of an initial *task network*, which consists of a set of *tasks* and a set of constraints on these tasks. The tasks can either be *primitive* or *non-primitive*. Primitive tasks correspond to actions in an underlying planning domain, while non-primitive tasks must be decomposed by an applicable *method*. Decomposing a non-primitive task with a method replaces the task with one of the task networks associated with the method. A solution to an HTN corresponds to a choice of a method to decompose each non-primitive task (including those introduced to decompose another task), such that all HTN constraints are satisfied, the goals of the underlying problem are achieved, and all mechanics of the underlying planning domain are respected. A number of planning systems have used HTNs such as UMCP [72] and SHOP [50] [49].

We approach HTN planning along the same lines as Son, et al. [69], who define the notion of *General Programs*. In a General Program we have elements of HTNs and of Golog programs. In particular, task networks are considered to be complex actions in Golog programs and complex actions may be used as tasks in HTNs. Son, et al. [69] encode their General Programs in ASP and use these constraints to restrict the set of valid plans in an underlying planning domain. Our formulation, that combines Golog and HTNs, is richer than existing formulations as it allows the combination of HTNs with ParaGolog. Additionally, we allow temporal formulae to be used to impose constraints on tasks and complex actions, as opposed to the non-temporal formulae used in Son, et al. [69]. We first present an encoding of a version of HTNs into SAT and show how HTNs constructs can be integrated into Golog and ParaGolog programs. Our encodings use a representation that splits on steps and as such are more compact than the ASP encodings of Son et al. [69].

First, we assume that an HTN is applied to a planning problem Π, which specifies the start state, goal, and the actions which constitute the set of primitive tasks. We formally define HTNs by first defining all of the components of an HTN, beginning with tasks. In the following we assume that all variables are typed and we omit types where they are clear from the context.

Definition 6.5.1. A *task* $\tau(?x_1 - type_1, ..., x_k - type_k)$ has a name τ and a set of parameters $?x_1, ..., ?x_k$ which are either variables or constants of the appropriate type from the underlying planning domain. □

Definition 6.5.2. A task is *primitive* if it or a grounding of it corresponds to an action from $\mathcal{A}$ in the underlying domain, otherwise it is *non-primitive* and must be decomposed by a method. □

We can now define *constraints* that apply to tasks.

Definition 6.5.3. For a set of tasks U, a *constraint c* on the tasks in U can either be a *partial ordering constraint* or a *truth constraint*. For distinct tasks $u_i, u_j \in U$, a *partial ordering constraint* $u_i < u_j$ asserts that task u_i must occur before task u_j. Where ψ is a temporal formula, *a truth constraint c* is one of the following:

- (ψ, u_i) – formula ψ must hold immediately before the execution of task u;
- (u, ψ) – formula ψ must hold immediately after the execution of task u;
- (u_i, ψ, u_j) – formula ψ must hold at every step from the end of u_i to the beginning of u_j (this also implies $u_i < u_j$);
- *pre*(ψ) is a formula that must hold immediately before the first task in U is executed;
- *post*(ψ) is a formula that must hold immediately after the last task in U is executed. □

We can now define *task networks*.

Definition 6.5.4. A *task network* $N := \langle U, C \rangle$ where:

- U is a set of primitive or non-primitive, ground or non-ground tasks; and
- C is a set of constraints on the tasks in U. □

Next, we define *methods*, which describe how to decompose ground non-primitive tasks into a set of sub-tasks.

Definition 6.5.5. An HTN *method*

$$\begin{aligned} m \; := \; \langle \textit{name}(m), \textit{task}(m), \; & \textit{subtasks}_1(m), \textit{constraints}_1(m), \textit{pre}_1(m), \\ & \textit{subtasks}_2(m), \textit{constraints}_2(m), \textit{pre}_2(m) \\ & ...\rangle \end{aligned}$$

where:

- $\textit{name}(m)$ is the name $n(?x_1 - type_1, ..., ?x_k - type_k)$ of the method, where n is unique and $?x_1, ..., ?x_k$ are variables that appear in the method;
- $\textit{task}(m)$ is the non-primitive task that this method decomposes;
- $\textit{subtasks}_i(m)$ is a set of subtasks that $task(m)$ is decomposed into if the precondition constraints in $\textit{pre}_i(m)$ hold and there are no $subtasks_j(m), j < i$, with preconditions that hold;
- $\textit{constraints}_i(m)$ is the set of ordering and truth constraints for $\textit{subtasks}_i(m)$; and
- $\textit{pre}_i(m)$ is a set of precondition constraints of the form $(\psi, \textit{task}(m))$. □

The variables that appear in a method $m := n(?x_1, ..., ?x_k)$ need not be in the set $\{?x_1, ..., ?x_k\}$. That is, there may be *local variables* that appear in subtasks, constraints, and preconditions. These, like other variables, are typed, and when decomposition occurs they are nondeterministically instantiated from the objects in the underlying planning domain. We say that a method can decompose a task if the method is *relevant* to the task.

Definition 6.5.6. A method m is *relevant* to a task $\tau(?x_1, ..., ?x_k)$ if there is a substitution σ such that $\sigma(\textit{task}(m)) := \tau(?x_1, ..., ?x_k)$. □

Definition 6.5.7. For a method m, let σ be a substitution such that $\sigma(\textit{task}(m)) := \tau(?x_1, ..., ?x_k)$ for some task $\tau(?x_1, ..., ?x_k)$. Then $\sigma(m)$ is m renamed according to σ – i.e. the substitution is applied to $\textit{name}(m)$, $\textit{task}(m)$, and every $\textit{subtasks}_i(m)$, and $\textit{constraints}_i(m)$. □

Definition 6.5.8. For a task $\tau(?x_1, ..., ?x_k)$, let $\textit{rel}(\tau(?x_1, ..., ?x_k))$ be the set of renamed methods that are relevant to $\tau(?x_1, ..., ?x_k)$. □

We can now formally define a HTN instance.

Definition 6.5.9. An HTN instance (N_0, M), for a planning problem Π, consists of:

- An initial task network N_0, where all tasks in N_0 are ground; and
- A set of methods M. □

To encode an HTN instance as SAT we need to encode all decompositions of all non-primitive tasks with all relevant methods. As we need a finite propositional formula to represent the HTN we require that HTN decompositions are finite. That is, if methods allow for recursion – i.e. a method m can be applied to a task u_1 where there exists another task u_2 that is in both the ancestors of u_1 and the subtasks of the renamed m – we require that the number of recursions be bounded. In practice, this bound can be computed by considering the minimum execution length of the tasks of the method and the planning horizon. We leave the exposition of the details of this computation for future work.

To illustrate the previous concepts, an example HTN for the SIMPLE-LOGISTICS domain (Appendix A) follows. In the following, the preconditions of the execution of primitive tasks is not encoded in the HTN as this is encoded in the underlying planning domain. First, we have the following methods M:

Method $m_0 := \texttt{Deliver}(?p - \mathit{package})$ with the following:

$$\begin{aligned}
\mathit{task}(m_0) &:= \texttt{Deliver}(?p)\\
\mathit{pre}_0(m_0) &:= \{\texttt{At}(?p, ?l_1),\ \mathit{goal}(\texttt{At}(?p, ?l_1)\}\\
\mathit{subtasks}_0(m_0) &:= \{\}\\
\mathit{constraints}_0(m_0) &:= \{\}\\
\mathit{pre}_1(m_0) &:= \{\texttt{At}(?t, ?l_1),\ \texttt{At}(?p, ?l_2),\ \mathit{goal}(\texttt{At}(?p, ?l_3))\}\\
\mathit{subtasks}_1(m_0) &:= \{u_{0,1,0} := \texttt{Navigate}(?t, ?l_1, ?l_2),\ u_{0,1,1} := \texttt{Pick-up}(?t, ?p, ?l_2),\\
&\qquad u_{0,1,2} := \texttt{Navigate}(?t, ?l_2, ?l_3),\ u_{0,1,3} := \texttt{Drop}(?t, ?p?, ?l_3)\}\\
\mathit{constraints}_1(m_0) &:= \{u_{0,1,0} < u_{0,1,1},\ u_{0,1,1} < u_{0,1,2},\ u_{0,1,2} < u_{0,1,3}\}
\end{aligned}$$

Method $m_1 := \texttt{Navigate}(?t - truck, ?l_1 - location, ?l_2 - location)$ with the following:

$$\begin{aligned}
\textit{task}(m_1) &:= \texttt{Navigate}(?t, ?l_1, ?l_2) \\
\textit{pre}_0(m_1) &:= \{\texttt{At}(?t, ?l_2)\} \\
\textit{subtasks}_0(m_1) &:= \{\} \\
\textit{constraints}_0(m_1) &:= \{\} \\
\textit{pre}_1(m_1) &:= \{\texttt{road}(?l_1, ?l_2)\} \\
\textit{subtasks}_1(m_1) &:= \{\texttt{Drive}(?t, ?l_1, ?l_2)\} \\
\textit{constraints}_1(m_1) &:= \{\} \\
\textit{pre}_2(m_1) &:= \{\texttt{road}(?l_1, ?l_3)\} \\
\textit{subtasks}_2(m_1) &:= \{u_{1,2,0} := \texttt{Drive}(?t, ?l_1, ?l_3),\ \ u_{1,2,1} := \texttt{Navigate}(?l_3, ?l_2)\} \\
\textit{constraints}_2(m_1) &:= \{u_{1,2,0} < task_{1,2,1},\ \ \textit{post}(\texttt{At}(?t, ?l_2))\}
\end{aligned}$$

We can then have a following initial task network $N_0 := \langle U, C\rangle$, where:

$$\begin{aligned}
U &:= \{\texttt{Deliver}(P_1),\ \texttt{Deliver}(P_2),\ ...\} \\
C &:= \{\}
\end{aligned}$$

An HTN instance (N_0, M), is interpreted in a pair of state-action trajectories

$$\langle J_h := s_0a_{0,0}, ..., a_{0,k_0}s_1, ..., a_{h-1,0}a_{h-1,k_{h-1}}s_h, J_i := s_ia_{i,0}, ..., a_{i,k_0}s_{i+1}, ..., a_{j-1,0}a_{j-1,k_{j-1}}s_j\rangle$$

by checking recursively if $\langle J_h, J_i\rangle$ is a *trace* of (N_0, M).

Before formally defining a trajectory of an HTN instance, we need a few preliminary definitions. Informally, a task u can be *executed* in a state-action trajectory J in the following cases:

- A primitive task u is *executed* in J if the action it represents is in J; A non-primitive task u is *executed* in J if it has a valid recursive decomposition such that all of the primitive tasks in the decomposition are run in J.

We say that a primitive task is executed at the step at which the related action appears and a non-primitive task is executed over a step envelope that encapsulates all of its recursively generated primitive sub-tasks.

Definition 6.5.10. Given a trajectory J and a task u executed in J, let *start*(u) be the start step of u in J and *end*(u) be the end step of u. If u is a primitive task, *start*(u) is the step of the action it represents and *end*(u) is one step later. If u is a non-primitive task *start*(u) and *end*(u) define the boundaries of the envelope that u is executed over. □

Definition 6.5.11. A trajectory $J := s_0 a_{0,0}, ..., a_{0,k_0} s_1, ..., a_{h-1,0} a_{h-1,k_{h-1}} s_h$, has a sub-trajectory $J[i, j]$, for $0 \leq i < j \leq h$, that is a subsequence of J from step i to step j. □

Definition 6.5.12. Let $\langle J_h, J_i \rangle \models (N, M)$ mean that pair of trajectories

$$\langle J_h := s_0 a_{0,0}, ..., a_{0,k_0} s_1, ..., a_{h-1,0} a_{h-1,k_{h-1}} s_h, J_i := s_i a_{i,0}, ..., a_{i,k_0} s_{i+1}, ..., a_{j-1,0} a_{j-1,k_{j-1}} s_j \rangle$$

is a trace of HTN program (N, M). Where $N := \langle U, C \rangle$ is a ground task network, a trajectory $\langle J_h, J_i \rangle$ is a trace of (N, M) iff the following conditions are met:

1. For all primitive tasks $u_l \in U$, where u corresponds to a planning action a_l, we have that $a_l \in J_i$;

2. For each non-primitive task $u_l \in U$, there is a method $m \in M$ that is applied to u_l such that $\mathit{subtasks}_l(m)$ and $\mathit{constraints}_l(m)$ are used to decompose u_l because $\mathit{pre}_l(m)$ holds for J_l and no $\mathit{pre}_{l'}(m)$, for any $l' < l$, and that the pair of trajectories $\langle J_h, J_l[\mathit{start}(u_l), \mathit{end}(u_l)] \rangle \models (\langle \mathit{subtasks}_l(m), \mathit{constraints}_l(m) \rangle, M)$;

3. For tasks $u_u, u_v \in U$, where there is an ordering constraint $u_u < u_v \in C$ we have that $\mathit{end}(u_u) \leq \mathit{start}(u_v)$;

4. For constraint $c \in C$, there are the following cases:

   - $c = (\psi, u_u)$: $\langle s_{\mathit{start}(u_u)}, J_h \rangle \models \psi$;
   - $c = (u, \psi)$: $c = (\psi, u_u)$: $\langle s_{\mathit{end}(u_u)}, J_h \rangle \models \psi$;
   - $c = (u_u, \psi, u_v)$: $\forall_{w \in \{\mathit{end}(u_u), ..., \mathit{start}(u_v)\}}, \langle s_w, J_h \rangle \models \psi$;
   - $c = \mathit{pre}(\psi)$: Where u is the minimum $\mathit{start}(u)$ for all tasks $u \in U$, we have $\langle s_u, J_h \rangle \models \psi$;
   - $c = \mathit{post}(\psi)$: Where u is the maximum $\mathit{end}(u)$ for all tasks $u \in U$, we have $\langle s_u, J_h \rangle \models \psi$. □

6.5.1 Encoding

Formally, we define an encoding of an HTN instance (N, M) and a bounded planning problem Π_h as follows.

Definition 6.5.13. Let $\mathcal{T}^{HTN}$ be a function which takes a bounded planning problem Π_h and an HTN instance (N, M) and returns a SAT problem. We say that $\mathcal{T}^{HTN}$ reduces Π_h and (N, M) to SAT and that $\phi := \mathcal{T}^{HTN}(\Pi_h, (N, M))$ is an encoding of Π_h and (N, M) iff a valid plan π can be extracted from every satisfying valuation of ϕ and given the associated trajectory $J_\pi := s_0 a_{0,0}, ..., a_{0,k_0} s_1, ..., a_{h-1,0} a_{h-1,k_{h-1}} s_h$, the pair $\langle J_\pi, J_\pi \rangle \models (N, M)$. Finally, we require that ϕ is unsatisfiable only if there is no such plan. □

In the following we use a set of predicates that describe the execution of tasks and the application of methods. These are described as they are introduced in the following axiom schemata. The predicates for tasks and methods are generated by starting with the initial task network and then recursively applying all relevant methods to non-primitive tasks.

For ground task $\tau(X, ..., X_k)$, we have the predicate *task-s*$(\tau(X_1, ..., X_k), n, t)$ which indicates the start of the task $\tau(X_1, ..., X_k)$ with the unique name n is at step t. We also have the predicate *task-e*$(\tau(X_1, ..., X_k), n, t)$ which similarly indicates the end of the task. The unique name parameter is required because the same task may be used in more than one decomposition. All tasks in the same network have the same unique name.

Initial task network:

Schema 6.5.1. For every task $\tau(X_1, ..., X_k)$ in N_0, there is the following clause:

$$\bigvee_{t \in \{0,...,h-1\}} \textit{task-s}(\tau(X_1, ..., X_k), 0, t)$$ □

Schema 6.5.2. For all *task-s*$(\tau(X_1, ..., X_k), 0, t_1)$, there is the following clause:

$$\textit{task-s}(\tau(X_1, ..., X_k), 0, t_1) \rightarrow \bigvee_{t_2 \in \{t_1+1,...,h\}} \textit{task-e}(\tau(X_1, ..., X_k), 0, t_2)$$ □

Schema 6.5.3. For all *task-s*$(\tau(X_1, ..., X_k), 0, t_1)$ and $t_2 \in \{t_1, ..., h-1\}$, there is the following clause:

$$\neg\textit{task-s}(\tau(X_1, ..., X_k), 0, t_1) \vee \neg\textit{task-s}(\tau(X_1, ..., X_k), 0, t_2)$$ □

Schema 6.5.4. For all *task-e*$(\tau(X_1, ..., X_k), 0, t_1)$ and all $t_2 \in \{t_1, ..., h\}$, there is the following clause:

$$\neg\textit{task-e}(\tau(X_1, ..., X_k), 0, t_1) \vee \neg\textit{task-e}(\tau(X_1, ..., X_k), 0, t_2)$$ □

Schema 6.5.5. For all *task-s*$(\tau(X_1, ..., X_k), 0, t_1)$, and all $t_2 \in \{0, ..., t_1 - 1\}$, there is the following clause:

$$\textit{task-s}(\tau(X_1, ..., X_k), 0, t_1) \rightarrow \neg \textit{task-e}(\tau(X_1, ..., X_k), 0, t_2) \qquad \square$$

The constraints C for the initial task network are described later with the truth and ordering constraints for other task networks.

Primitive tasks:

For primitive tasks we need to assert that the tasks have exactly length 1, that the relevant planning action is executed, and that actions imply that some primitive task executes them. If a primitive task refers to an action which is not in $\mathcal{A}^t$ from Π_h, then the task cannot be executed. Let $a(X_1, ..., X_k)$ be a planning action from $\mathcal{A}$.

Schema 6.5.6. For all *task-s*$(a(X_1, ..., X_k), n, t)$, there is the following clause:

$$\textit{task-s}(a(X_1, ..., X_k), n, t) \rightarrow a(X_1, ..., X_k)^t \qquad \square$$

Schema 6.5.7. For all *task-s*$(a(X_1, ..., X_k), n, t)$, there is the following clause:

$$\textit{task-s}(a(X_1, ..., X_k), n, t) \leftrightarrow \textit{task-e}(a(X_1, ..., X_k), n, t + 1) \qquad \square$$

Schema 6.5.8. For all actions $a(X_1, ..., X_k) \in \mathcal{A}$ and all $t \in \{0, ..., h - 1\}$, there is the following clause:

$$a(X_1, ..., X_k)^t \rightarrow \bigvee_{\textit{task-s}(a(X_1, ..., X_k), n, t)} \textit{task-s}(a(X_1, ..., X_k), n, t) \qquad \square$$

Non-primitive tasks:

Let $\tau(X_1, ..., X_k)$ be a ground non-primitive task.

Schema 6.5.9. For all *task-s*$(\tau(X_1, ..., X_k), n, t)$, there are the following clauses:

$$\textit{task-s}(\tau(X_1, ..., X_k), n, t) \leftrightarrow \bigvee_{m(y_1, ..., y_l) \in \textit{rel}(\tau(X_1, ..., X_k))} \textit{method}(m(Y_1, ..., Y_l), n) \qquad \square$$

Schema 6.5.10. For all *task-s*$(\tau(X_1, ..., X_k), n, t)$, and all $m(Y_1, ..., Y_l) \in \textit{rel}(\tau(X_1, ..., X_k))$, there are the following clauses:

$$(\textit{task-s}(\tau(X_1, ..., X_k), n, t) \wedge \textit{method}(m(Y_1, ..., Y_l)), n) \leftrightarrow \textit{method-s}(m(Y_1, ..., Y_l), n, t) \quad \square$$

Schema 6.5.11. For all $\mathit{task\text{-}e}(\tau(X_1, ..., X_k), n, t)$ and $m(Y_1, ..., Y_l) \in \mathit{rel}(\tau(X_1, ..., X_k))$, there are the following clauses:

$$(\mathit{task\text{-}e}(\tau(X_1, ..., X_k), n, t) \wedge \mathit{method}(m(Y_1, ..., Y_l)), n) \leftrightarrow \mathit{method\text{-}e}(m(Y_1, ..., Y_l), n, t) \quad \square$$

Schema 6.5.12. For all non-primitive tasks $\tau(X_1, ..., X_k)$, and all $m_1(Y_1, ..., Y_l), m_2(Z_1, ..., Z_i) \in \mathit{rel}(\tau(X_1, ..., X_k)), m_1 \neq m_2$, there is the following clause:

$$\neg \mathit{method}(m_1(Y_1, ..., Y_l), n) \vee \neg \mathit{method}(m_2(X_1, ..., Z_i), n) \quad \square$$

Methods:

Let $m(X_1, ..., X_k)$ be a ground method with the ordered list of decompositions:

$$\langle \mathit{subtasks}_1(m), \mathit{constraints}_1(m), \mathit{pre}_1(m), ..., \mathit{subtasks}_l(m), \mathit{constraints}_l(m), \mathit{pre}_l(m) \rangle$$

Let $\mathit{decomp}_i(m(X_1, ..., X_k), n)$ be a predicate that represents that $\mathit{task}(m)$ of the method $m(X_1, ..., X_k)$, with name n, is being decomposed by $\mathit{subtasks}_i(m)$ and $\mathit{constraints}_i(m)$. Finally, let the predicates $\mathit{decomp\text{-}s}_i(m(X_1, ..., X_k), n, t)$ and $\mathit{decomp\text{-}e}_i(m(X_1, ..., X_k), n, t)$ indicate the start and end steps of $m(X_1, ..., X_k)$, with name n.

Schema 6.5.13. For all $\mathit{method\text{-}s}(m(X_1, ..., X_k), n, t)$, there is the following clause:

$$\mathit{method\text{-}s}(m(X_1, ..., X_k), n, t) \rightarrow \bigvee_{i \in \{1, ..., l\}} \mathit{decomp}_i(m(X_1, ..., X_k), n) \quad \square$$

Schema 6.5.14. For all $\mathit{method\text{-}s}(m(X_1, ..., X_k), n, t)$, and all $\mathit{decomp}_i(m(X_1, ..., X_k), n)$, there are the following clauses:

$$(\mathit{method\text{-}s}(m(X_1, ..., X_k), n, t) \wedge \mathit{decomp}_i(m(X_1, ..., X_k), n)) \leftrightarrow$$
$$\mathit{decomp\text{-}s}_i(m(X_1, ..., X_k), n, t) \quad \square$$

Schema 6.5.15. For all $\mathit{method\text{-}e}(m(X_1, ..., X_k), n, t)$ and $\mathit{decomp}_i(m(X_1, ..., X_k), n)$, there are the following clauses:

$$(\mathit{method\text{-}e}(m(X_1, ..., X_k), n, t) \wedge \mathit{decomp}_i(m(X_1, ..., X_k), n)) \leftrightarrow$$
$$\mathit{decomp\text{-}e}_i(m(X_1, ..., X_k), n, t) \quad \square$$

Schema 6.5.16. For all $\mathit{decomp}_i(m(X_1, ..., X_k), n)$ and $\mathit{decomp}_j(m(X_1, ..., X_k), n)$, where $i > j$, there is the following clause:

$$\neg \mathit{decomp}_i(m(X_1, ..., X_k), n) \vee \neg \mathit{decomp}_j(m(X_1, ..., X_k), n) \qquad \square$$

Every decomposition $\mathit{decomp}_i(m(X_1, ..., X_k), n)$ has a predicate $\mathit{pre}_i(m(X_1, ..., X_k), n)$ to represent that its preconditions hold. The schemata that describe the preconditions will be presented with the truth and ordering constraints. Here we need constraints that assert that a decomposition can only be selected if none of the preconditions of lower numbered decompositions hold. If there are no preconditions for decomposition i then $\mathit{pre}_i(m(X_1, ..., X_k), n)$ is asserted true.

Schema 6.5.17. For all $\mathit{decomp}_i(m(X_1, ..., X_k), n)$, where $i > 0$ and $j \in \{0, ..., i-1\}$, there is the following clause:

$$\mathit{decomp}_i(m(X_1, ..., X_k), n) \rightarrow \neg \mathit{pre}_j(m(X_1, ..., X_k), n) \qquad \square$$

We now need to assert that each decomposition implies that all relevant tasks are executed within the step bounds of the method.

Schema 6.5.18. For all $\mathit{decomp\text{-}s}_i(m(X_1, ..., X_k), n, t_1)$, and all $\tau(Y_1, ..., Y_l) \in \mathit{subtasks}_i(m(X_1, ..., X_k))$, there is the following clause:

$$\mathit{decomp\text{-}s}_i(m(X_1, ..., X_k), n, t_1) \rightarrow \bigvee_{t_2 \in \{t_1, ..., h-1\}} \mathit{task\text{-}s}(\tau(Y_1, ..., Y_l), n{:}m{:}i, t_2) \qquad \square$$

Schema 6.5.19. For all $\mathit{decomp\text{-}s}_i(m(X_1, ..., X_k), n, t_1)$, all $\tau(Y_1, ..., Y_l) \in \mathit{subtasks}_i(m(X_1, ..., X_k))$, and all $t_2 \in \{0, ..., t_1 - 1\}$, there is the following clause:

$$\mathit{decomp\text{-}s}_i(m(X_1, ..., X_k), n, t_1) \rightarrow \neg \mathit{task\text{-}s}(\tau(Y_1, ..., Y_l), n{:}m{:}i, t_2) \qquad \square$$

Schema 6.5.20. For all $\mathit{decomp\text{-}e}_i(m(X_1, ..., X_k), n, t_1)$, and all $\tau(Y_1, ..., Y_l) \in \mathit{subtasks}_i(m(X_1, ..., X_k))$, there is the following clause:

$$\mathit{decomp\text{-}e}_i(m(X_1, ..., X_k), n, t_1) \rightarrow \bigvee_{t_2 \in \{1, ..., t_1\}} \mathit{task\text{-}e}(\tau(Y_1, ..., Y_l), n{:}m{:}i, t_2) \qquad \square$$

Schema 6.5.21. For all $\mathit{decomp\text{-}e}_i(m(X_1, ..., X_k), n, t_1)$, all $\tau(Y_1, ..., Y_l) \in \mathit{subtasks}_i(m(X_1, ..., X_k))$, and all $t_2 \in \{t_1 + 1, ..., h\}$, there is the following clause:

$$\mathit{decomp\text{-}e}_i(m(X_1, ..., X_k), n, t_1) \rightarrow \neg \mathit{task\text{-}e}(\tau(Y_1, ..., Y_l), n{:}m{:}i, t_2) \qquad \square$$

Truth and ordering constraints:

Finally, we need to assert that each decomposition of a non-primitive task, including the initial task network as a special case, implies that all related constraints hold. The following describes clauses for constraints in *constraints*$_i(m(X_1, ..., X_k))$ and *pre*$_i(m(X_1, ..., X_k))$ that apply to the tasks from *subtasks*$_i(m(X_1, ..., X_k))$ when decomposition *decomp*$_i(m(X_1, ..., X_k), n)$ is performed. Here we refer to the tasks $\tau_1(Y_1, ..., Y_j), \tau_2(Z_1, ..., Z_l) \in$ *subtasks*$_i(m(X_1, ..., X_k))$ where the arguments $\{Y_1, ..., Y_j\} \in \{X_1, ..., X_k\}$ and $\{Z_1, ..., Z_l\} \in \{X_1, ..., X_k\}$. First, a partial ordering constraint $\tau_1(Y_1, ..., Y_j) < \tau_2(Z_1, ..., Z_l)$ asserts that task $\tau_1(Y_1, ..., Y_j)$ must occur before task $\tau_2(Z_1, ..., Z_l)$.

Schema 6.5.22. For all $\tau_1(Y_1, ..., Y_j) < \tau_2(Z_1, ..., Z_l) \in$ *constraints*$_i(m(X_1, ..., X_k))$ and $t_1 \in \{0, ..., h\}$ and $t_2 \in \{0, ..., t_1 - 1\}$, there is the following clause:

$$\textit{task-e}(\tau_1(Y_1, ..., Y_j), n, t_1) \rightarrow \neg\textit{task-s}(\tau_2(Z_1, ..., Z_l), n, t_2) \qquad \square$$

Next, we have schemata for the truth constraints.

Schema 6.5.23. For all $(\psi, \tau_1(Y_1, ..., Y_j)) \in$ *constraints*$_i(m(X_1, ..., X_k))$ and all $t \in \{0, ..., h\}$, there is the following clause:

$$\textit{task-s}(\tau_1(Y_1, ..., Y_j), n, t) \rightarrow holds(\psi, t) \qquad \square$$

Schema 6.5.24. For all $(\tau_1(Y_1, ..., Y_j), \psi) \in$ *constraints*$_i(m(X_1, ..., X_k))$ and all $t \in \{0, ..., h\}$, there is the following clause:

$$\textit{task-e}(\tau_1(Y_1, ..., Y_j), n, t) \rightarrow holds(\psi, t) \qquad \square$$

For truth constraints of the form $(\tau_1(Y_1, ..., Y_j), \psi, \tau_2(Z_1, ..., Z_l))$, we introduce a new predicate *until*$(\psi, \tau_2(Z_1, ..., Z_l), t)$, which means that ψ must hold from t until the start of $\tau_2(Z_1, ..., Z_l)$.

Schema 6.5.25. For all $(\tau_1(Y_1, ..., Y_j), \psi, \tau_2(Z_1, ..., Z_l)) \in$ *constraints*$_i(m(X_1, ..., X_k))$, and all *task*-e$(\tau_1(Y_1, ..., Y_j), n, t_1)$, where $t_1 < h$, there is the following clause:

$$\textit{task-e}(\tau_1(Y_1, ..., Y_j), n, t_1) \rightarrow \textit{until}(\psi, \tau_2(Z_1, ..., Z_l), t_1) \qquad \square$$

Schema 6.5.26. For all *until*$(\psi, \tau_2(Z_1, ..., Z_l), t_1)$, there is the following clause:

$$\textit{until}(\psi, \tau_2(Z_1, ..., Z_l), t_1) \rightarrow holds(\psi, t_1) \qquad \square$$

Schema 6.5.27. For all $\textit{until}(\psi, \tau_2(Z_1, ..., Z_l), t_1)$, there is the following clause:

$$\textit{until}(\psi, \tau_2(Z_1, ..., Z_l), t_1) \rightarrow (\textit{task-s}(\tau_2(Z_1, ..., Z_l), n, t_2) \vee \textit{until}(\psi, \tau_2(Z_1, ..., Z_l), t_1{+}1)) \quad \Box$$

To encode constraints of the form $\textit{pre}(\psi)$, we introduce the following new predicates:

- $\textit{first}(\textit{decomp}_i(m(X_1, ..., X_k), n), t)$: some task from the set $\textit{subtasks}_i(m(X_1, ..., X_k))$ of $\textit{decomp}_i(m(X_1, ..., X_k), n)$ starts at t and no task starts any earlier;
- $\textit{some-starts}(\textit{decomp}_i(m(X_1, ..., X_k), n), t)$: some task from $\textit{subtasks}_i(m(X_1, ..., X_k))$ starts at t;
- $\textit{none-earlier}(\textit{decomp}_i(m(X_1, ..., X_k), n), t)$: no tasks from $\textit{subtasks}_i(m(X_1, ..., X_k))$ start before t.

Schema 6.5.28. For all $\textit{decomp}_i(m(X_1, ..., X_k), n)$ and $t \in \{0, ..., h\}$, there are the following clauses:

$$\begin{gathered}\textit{first}(\textit{decomp}_i(m(X_1, ..., X_k), n), t) \leftrightarrow \\ (\textit{some-starts}(\textit{decomp}_i(m(X_1, ..., X_k), n), t) \wedge \\ \textit{none-earlier}(\textit{decomp}_i(m(X_1, ..., X_k), n), t))\end{gathered} \quad \Box$$

Schema 6.5.29. For all $\textit{some-starts}(\textit{decomp}_i(m(X_1, ..., X_k), n), t)$, there are the following clauses:

$$\begin{gathered}\textit{some-starts}(\textit{decomp}_i(m(X_1, ..., X_k), n), t) \leftrightarrow \\ \bigvee_{\tau(Y_1,...,Y_l) \in \textit{subtasks}_i(m(X_1,...,X_k))} \textit{task-s}(\tau(Y_1, ..., Y_l), n, t)\end{gathered} \quad \Box$$

Schema 6.5.30. For all $\textit{none-earlier}(\textit{decomp}_i(m(X_1, ..., X_k), n), t_1))$,
and all $\textit{task-s}(\tau(Y_1, ..., Y_l), n, t_2)$, where $t_2 < t_1$, there is the following clause:

$$\textit{none-earlier}(\textit{decomp}_i(m(X_1, ..., X_k), n), t_1)) \rightarrow \neg\textit{task-s}(\tau(Y_1, ..., Y_l), n, t_2) \quad \Box$$

Preconditions are then asserted with the following clauses.

Schema 6.5.31. For all $\textit{pre}(\psi) \in (\textit{constraints}_i(m(X_1, ..., X_k)) \cup \textit{pre}_i(m(X_1, ..., X_k)))$, and all $t \in \{0, ..., h\}$, there is the following clause:

$$\textit{first}(\textit{decomp}_i(m(X_1, ..., X_k), n), t) \rightarrow \textit{holds}(\psi, t) \quad \Box$$

For every $\textit{pre}_i(m(X_1, ..., X_k), n)$ created to encode the preconditions of methods for Schema 6.5.17, we need constraints to assert that the relevant preconditions hold.

Schema 6.5.32. Let $\mathit{pre}(\psi_1), ..., \mathit{pre}(\psi_l)$ be the preconditions in the set $\mathit{pre}_i(m(X_1, ..., X_k), n)$. For all $\mathit{pre}_i(m(X_1, ..., X_k), n)$ and $t \in \{0, ..., h\}$, there are the following clauses:

$$(\mathit{pre}_i(m(X_1, ..., X_k), n) \wedge \mathit{first}(\mathit{decomp}_i(m(X_1, ..., X_k), n), t)) \\ \leftrightarrow (holds(\psi_1) \wedge ... \wedge holds(\psi_l)) \qquad \square$$

Finally, the constraint $\mathit{post}(\psi)$ asserts that ψ must hold immediately after the last task from $\mathit{subtasks}_i(m(X_1, ..., X_k))$ is executed. This does not have to be at the end of the decomposition $\mathit{decomp}_i(m(X_1, ..., X_k), n)$. As with preconditions we introduce a set of new predicates: $\mathit{last}(\mathit{decomp}_i(m(X_1, ..., X_k), n), t)$, $\mathit{some\text{-}ends}(\mathit{decomp}_i(m(X_1, ..., X_k), n), t)$, and also $\mathit{none\text{-}later}(\mathit{decomp}_i(m(X_1, ..., X_k), n), t)$. The introduced predicates have meanings mirroring those introduced for the preconditions. We then have the following schemata.

Schema 6.5.33. For all $\mathit{decomp}_i(m(X_1, ..., X_k), n)$ and all $t \in \{0, ..., h\}$, there are the following clauses:

$$\mathit{last}(\mathit{decomp}_i(m(X_1, ..., X_k), n), t) \leftrightarrow \\ (\mathit{some\text{-}ends}(\mathit{decomp}_i(m(X_1, ..., X_k), n), t) \wedge \\ \mathit{none\text{-}later}(\mathit{decomp}_i(m(X_1, ..., X_k), n), t)) \qquad \square$$

Schema 6.5.34. For all $\mathit{post}(\psi) \in \mathit{constraints}_i(m(X_1, ..., X_k))$ and all $t \in \{0, ..., h\}$, there is the following clause:

$$\mathit{last}(\mathit{decomp}_i(m(X_1, ..., X_k), n), t) \rightarrow holds(\psi, t) \qquad \square$$

6.5.2 Integrating HTN and Golog

It is possible to integrate Golog programs and HTN instances to achieve control knowledge specifications that can practically represent both procedural knowledge and partial ordering constraints. We outline an expressive formalism that goes further than previous approaches (such as Son et al. [69]), and combines ParaGolog and the HTN formalism described in the previous section. Son, et al. [69] define the notion of General Programs with *general complex actions*, where a general complex action is either a complex action of Golog or a set of procedures over which there are a set of HTN ordering and truth constraints. The semantics of their programs are built from their semantics for Golog using the notion of a trace. Their semantics results in general programs being executed sequentially, despite the partially ordered nature of HTNs. We present a similar formalism that is based on our definitions of ParaGolog and HTNs that allows parallel concurrency. We call such programs *Parallel Programs*. We present an encoding of Parallel Programs to SAT that is based on the previously presented encodings of ParaGolog and HTNs. To ensure compactness, our encoding is split on both steps and argument groundings. Parallel Programs are defined as follows.

Definition 6.5.14. A *Parallel Program* (R, M, δ), for a planning problem Π has the following elements:

- A set of procedures R;
- A set of methods M; and
- A complex action δ that is as defined in Definition 6.4.14 or is a task-network as defined in Definition 6.5.4. $\square$

There are then the following considerations:

- Non-primitive tasks are now either decomposable by methods in M or are complex actions;
- Previously HTN constraints were defined for the ground case. Now we require that any non-ground tasks only have variables under the scope of a ParaGolog nondeterministic or concurrent argument choice;
- We require that the definition of the initial complex action is non-recursive, that is we can unroll it into a finite structure.

A parallel program (R, M, δ), is interpreted in a pair of state-action trajectories

$$\langle J_h := s_0 a_{0,0}, ..., a_{0,k_0} s_1, ..., a_{h-1,0} a_{h-1,k_{h-1}} s_h, J_i := s_i a_{i,0}, ..., a_{i,k_0} s_{i+1}, ..., a_{j-1,0} a_{j-1,k_{j-1}} s_j \rangle$$

by checking recursively if $\langle J_h, J_i \rangle$ is a trace of (R, M, δ).

Definition 6.5.15. Let $\langle J_h, J_i \rangle \models (R, M, \delta)$ mean that trajectory

$$\langle J_h := s_0 a_{0,0}, ..., a_{0,k_0} s_1, ..., a_{h-1,0} a_{h-1,k_{h-1}} s_h, J_i := s_i a_{i,0}, ..., a_{i,k_0} s_{i+1}, ..., a_{j-1,0} a_{j-1,k_{j-1}} s_j \rangle$$

is a trace of parallel program (R, M, δ). A trajectory $\langle J_h, J_i \rangle \models (R, M, \delta)$ iff the following conditions are met:

1. If δ is a complex action of ParaGolog then Definition 6.4.17 applies except that we are now dealing with parallel programs and complex actions as in Definition 6.5.14;

2. Otherwise $\delta := \langle U, C\rangle$, in which case Definition 6.5.15 applies, except that we are now dealing with parallel programs and non-primitive tasks that represent complex actions that are interpreted according to the previous point. $\square$

Formally, we define an encoding of a parallel program (R, M, δ) and a bounded planning problem Π_h as follows.

Definition 6.5.16. Let $\mathcal{T}^{PP}$ be a function which takes a bounded planning problem Π_h and an HTN instance (R, M, δ) and returns a SAT problem. We say that $\mathcal{T}^{PP}$ reduces Π_h and (R, M, δ) to SAT and that $\phi := \mathcal{T}^{PP}(\Pi_h, (R, M, \delta))$ is an encoding of Π_h and (R, M, δ) iff a valid plan π can be extracted from every satisfying valuation of ϕ and, given the associated trajectory $J_\pi := s_0 a_{0,0}, ..., a_{0,k_0} s_1, ..., a_{h-1,0} a_{h-1,k_{h-1}} s_h$, the pair $\langle J_\pi, J_\pi\rangle \models (R, M, \delta)$. Finally, we require that ϕ is unsatisfiable only if there is no such plan. $\square$

We then have the following constraints.

Task Network Complex Action: $\delta = \langle U, C\rangle$

Here we treat δ as the application of a method $m(?x_1, ..., ?x_k)$ with a single decomposition *decomp*$_1$ with *subtasks*$_1(m)$, *constraints*$_1(m)$ and an empty precondition *pre*$_1(m)$. We then have the following schemata.

Schema 6.5.35. For all *trans-s*(n, δ, t), there are the following clauses:

$$\textit{trans-s}(n, \delta, t) \rightarrow \textit{method-s}(m(?x_1, ..., ?x_k), n, t)$$ $\square$

Schema 6.5.36. For all *trans-e*(n, δ, t), there are the following clauses:

$$\textit{trans-e}(n, \delta, t) \rightarrow \textit{method-e}(m(?x_1, ..., ?x_k), n, t)$$ $\square$

Schema 6.5.37. For all *decomp*$_i(m(?x_1, ..., ?x_k), n)$ and all $t \in \{0, ..., h\}$, there are the following clauses:

$$\textit{decomp}_i(m(?x_1, ..., ?x_k), n)$$ $\square$

Then Schemata 6.5.13 - 6.5.21 and 6.5.22 - 6.5.34 apply.

Complex Action as a Non-Primitive Task $u = \delta$

Schema 6.5.38. For all *task-s*$(\tau(?x_1, ..., ?x_k), n, t)$, there are the following clauses:

$$\textit{task-s}(\tau(?x_1, ..., ?x_k), n, t) \leftrightarrow \textit{trans-s}(n, \delta, t) \qquad \square$$

Schema 6.5.39. For all *task-e*$(\tau(?x_1, ..., ?x_k), n, t)$, there are the following clauses:

$$\textit{task-e}(\tau(?x_1, ..., ?x_k), n, t) \leftrightarrow \textit{trans-e}(n, \delta, t) \qquad \square$$

Primitive Task:

Schema 6.5.40. For all *task-s*$(a(?x_1, ..., ?x_k), n, t_1)$, there is the following clause:

$$\textit{task-s}(a(?x_1, ..., ?x_k), n, t_1) \rightarrow \bigvee_{t_2 \in \{t_1+1, ..., h\}} \textit{task-e}(a(?x_1, ..., ?x_k), n, t_2) \qquad \square$$

Schema 6.5.41. For all *task-e*$(a(?x_1, ..., ?x_k), n, t_1)$, there is the following clause:

$$\textit{task-e}(a(?x_1, ..., ?x_k), n, t_2) \rightarrow \bigvee_{t_2 \in \{t_1+1, ..., h\}} \textit{task-s}(a(?x_1, ..., ?x_k), n, t_1) \qquad \square$$

Schema 6.5.42. For all *task-s*$(a(?x_1, ..., ?x_k), n, t_1)$, and all *task-e*$(a(?x_1, ..., ?x_k), n, t_2)$, where $t_2 > t_1$, and all groundings $X_1, ..., X_k$ of the variables in $?x_1, ..., ?x_k$, where $X_1 \in \mathtt{D}(\mathit{type}_1)$, ..., $X_k \in \mathtt{D}(\mathit{type}_k)$ for $a(?x_1 - \mathit{type}_1, ..., ?x_k - \mathit{type}_k)$, there are the following clauses:

$$\begin{aligned}&(\textit{task-s}(a(?x_1, ..., ?x_k), n, t_1) \wedge \textit{task-e}(a(?x_1, ..., ?x_k), n, t_2) \wedge \\ &\textit{eq}(?x_1, X_1) \wedge ... \wedge \textit{eq}(?x_k, X_k)) \leftrightarrow \textit{trans}(n, a(X_1, ..., X_k), t)\end{aligned} \qquad \square$$

We then have that Schemata 6.4.111 - 6.4.113 apply to the generated predicates of the form *trans*$(n, a(X_1, ..., X_k), t)$.

The encodings presented here have been tested on a number of example problems. In the future we intend to provide a proof of their correctness and a thorough empirical evaluation.

6.6 Conclusion

In this chapter we have extended the work of Son et al. [69] and presented a number of novel expressive control knowledge formalisms encompassing temporal logic, Golog programs, and HTNs. We have presented a number of novel and compact split encodings of these formalism into SAT. For temporal logic, we use past operators and use an efficient split encoding that does not ground quantifiers. For Golog we allow, for the first time, the use of temporal formulae, rather than standard boolean formulae, in test actions and the guards of conditional and while loop actions. We also introduce novel Golog constructs to allow concurrent action execution. We introduce a concurrent action execution semantics that allows true parallel action execution and call the resulting language ParaGolog. We provide a number of highly compact encodings of ParaGolog to SAT that split on steps and on operator arguments. We also introduce a version of HTN planning based on the SHOP [50] planning system and present a split encoding of this. Finally, we describe how to integrate these HTN programs with ParaGolog programs.

Chapter 7

Conclusion

In this chapter, we summarise the main contributions to the field of SAT-based planning presented in this thesis. We then discuss potential future research problems.

7.1 Summary of Contributions

In this thesis, we have advanced the state-of-the-art in SAT-based planning by addressing several of the approaches key shortfalls. Flat SAT encodings of planning suffer from the problem of size blowup. We addressed this problem by developing a framework in which to describe split encodings of planning as SAT. Using that framework we developed several novel encodings that are compact and amenable to state-of-the-art SAT procedures [59, 60]. Another key challenge has been to develop a SAT-based procedure that can achieve horizon-independent optimal plans. Meeting that challenge, we developed a novel *horizon-independent cost-optimal* procedure that poses PWMaxSAT problems to our own cost-optimising *conflict-driven clause learning* (CDCL) procedure. The encoding we developed [61, 62] constitutes the first application of the causal representations of planning to prove cost-optimality. Finally, taking insights gleaned for the classical propositional planning case, we have developed a number of split encodings of planning control-knowledge to SAT. We give encodings for temporal logic based constraints, procedural knowledge written in a fragment of Golog, a language with a true concurrent execution semantics similar to ConGolog, and also HTN based constraints. This work demonstrates promising approaches to guide SAT-based plan search with expressive domain-specific knowledge for real world planning problems.

Describing the contributions of this thesis in more detail, in Chapter 4 we presented a general framework for split action representations and a general encoding of split actions represented in our framework to SAT for both linear and parallel planning. Our framework can represent existing split encodings and we present a novel action representation that we call *precise splitting*. The encoding that results from our representation is the first split encoding that can be used for parallel step-optimal planning [60]. We show the constructiveness of this encoding and evaluate it empirically. Notably, our precisely split encodings are more compact than existing parallel encodings of planning as SAT in the same setting, sometimes an order of magnitude smaller – e.g. for the problem FREECELL-04 the flat encoding SMP requires 1.55 million clauses, while our precisely split-encoding requires only 120 thousand. We find that the compactness chiefly derives from having a factored representation of mutex relations between actions. We provide a detailed empirical comparison and find that our precisely split encoding is competitive with the flat state-of-the-art SAT-based planners for parallel step-optimal planning.

In Chapter 5 we considered the problem of using SAT-based planning systems to compute optimal plans for propositional planning problems with action costs. We modified the SAT solver RSAT to create PWM-RSAT, an effective Partial Weighted MaxSAT procedure that outperforms existing procedures for the problems we generate [62]. We explored how admissible relaxed planning heuristics can be computed with PWMaxSAT. We empirically compared the performance of our optimal relaxed planning system based on Partial Weighted MaxSAT against a state-space search-based optimal planner and find that our system is more efficient than that planner and solves more relaxed problems. We explored how SAT-based planning systems can be applied to horizon-independent cost-optimal planning [61, 62]. We presented the first planning system that uses SAT for a proof of horizon-independent cost-optimality. Our planning system incorporates the computation of an admissible relaxed planning heuristic. We empirically evaluated our approach against a state-space search based cost-optimal planner and found that our system solved the most problems in two domains where the branching factor is high and useful plans are long.

In Chapter 6 we review a number of different high-level control knowledge formalisms and novel split encodings of these formalisms into SAT. The control knowledge formalisms we examine are Linear Temporal Logic with a finite execution semantics, Golog [40] and ParaGolog, a novel variant of ConGolog [22] that has a semantics which allows true parallel action execution. We additionally presented a novel formulation of HTNs based on the SHOP planning system [50]. We allow these formalisms to be nested and hybridised. The encodings we present split a number of predicates on both operator arguments and time steps. The resulting SAT formulae have the property that they restrict the set of valid models of an underlying SAT encoding of planning, rather than being essential for planning itself.

7.2 Future Work

A pressing item for future work is to examine the benefits of our precisely split encodings (Section 4.2) for optimal planning using more advanced query strategies. In particular, query strategies such as those in Rintanen [55], Streeter and Smith [70], and Ray and Ginsberg [53]. Additionally, we plan to explore the relationship between our precisely split encodings and recent SAT encodings of planning such as the SAS+ encoding used in SASE [30], which initial results suggest exhibits similar performance to our approach. Finally, we plan to explore the space of split action representations that are describable in our framework and examine the performance of the resulting encodings. Also, along the lines of Juma et al. [31], we plan to use our precisely split encoding for cost-optimal planning.

The application of SAT-based techniques to horizon-independent cost-optimal planning represents one of the most interesting, challenging, and important areas of future work. We plan to further develop the SAT-based approach for horizon-independent cost-optimal planning described in Chapter 5. To do this we plan to exploit SMT —and/or branch-and-bound procedures from weighted MaxSAT— in proving the optimality of candidate solutions generated by our planner. We should also exploit recent work in using useful admissible heuristics for state-based search when evaluating whether horizon h yields an optimal solution [27]. We also plan to explore other SAT-based methods for computing horizon-independent cost-optimal plans, such as finding good upper bounds on the horizons we need to query to find cost-optimal plans.

Finally, we plan to more deeply explore the encodings of control knowledge we specify in Chapter 6, prove their correctness, and provide a thorough empirical evaluation of their performance. We then plan to apply these encodings to a number of different areas such as diagnosis and verification.

Bibliography

[1] Aws Albarghouthi, Jorge A. Baier, and Sheila A. McIlraith. On the use of planning technology for verification. In *Proceedings of the 2nd Workshop on Heuristics for Domain-independent Planning (ICAPS-09), Thessaloniki, Greece*, September 2009.

[2] Josep Argelic, Chu Min Li, Felip Manya, and Jordi Planes. The first and second Max-SAT evaluations. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4(2-4):251–278, 2008.

[3] Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artifical Intelligence (AIJ)*, 116(1-2):123–191, 2000.

[4] Christer Bäckström and Bernhard Nebel. Complexity results for SAS+ planning. *Computational Intelligence (COIN)*, 11:625–656, 1995.

[5] Jorge Baier and Sheila McIlraith. Planning with temporally extended goals using heuristic search. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS-06), Lake District, Cumbria, UK*, pages 342–345. AAAI Press, June 2006.

[6] Andreas Bauer and Patrik Haslum. Ltl goal specifications revisited. In *Proceedings of The 19th European Conference on Artificial Intelligence (ECAI-10), Lisbon, Portugal*, pages 881–886. IOS Press, August 2010.

[7] Armin Biere. P{re,i}coSAT@SC09. In *SAT-2009 Competitive Events Booklet*, page 41, 2009.

[8] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. *Artificial Intelligence (AIJ)*, 90(1-2):281–300, 1997.

[9] Markus Büttner and Jussi Rintanen. Satisfiability planning with constraints on the number of actions. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS-05), Lake District, Cumbria, UK*, pages 292–299. AAAI Press, June 2005.

[10] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence (AIJ)*, 69(1-2):165–204, 1994.

[11] Yixin Chen, Qiang Lv, and Ruoyun Huang. Plan-A: A cost-optimal planner based on SAT-constrained optimization. In *Prococeedings of the 2008 Internaltional Planning Competition (IPC-6), Sydney, Australia*, September 2008.

[12] Yixin Chen, Zhao Xing, and Weixiong Zhang. Long-distance mutual exclusion for propositional planning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07), Hyderabad, India*, pages 1840–1845. Morgan Kaufmann, January 2007.

[13] Stephen A. Cook. The complexity of theorem-proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing, Shaker Heights, OH, USA*, pages 151–158, 1971.

[14] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the 4th European Conference on Planning: Recent Advances in AI Planning (ECP-97), Toulouse, France*, pages 169–181, September 1997.

[15] Patrick Doherty and Jonas Kvarnström. TALplanner: A temporal logic-based planner. *AI Magazine*, 22(3):95–102, 2001.

[16] Stefan Edelkamp and Peter Kissmann. PDDL 2.1: The language for the classical part of IPC-4. In *Proceedings of the 2004 International Planning Competition (IPC-4), Whistler, Canada*, June 2004.

[17] Stefan Edelkamp and Peter Kissmann. GAMER: Bridging planning and general game playing with symbolic search. In *Prococeedings of the 2008 Internaltional Planning Competition, (IPC-6), Sydney, Australia*, September 2008.

[18] Michael Ernst, Todd Millstein, and Daniel Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97), Nagoya, Aichi, Japan*, pages 1169–1177, August 1997.

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

[20] Maria Fox and Derek Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)*, 20:61–124, 2003.

[21] Zhaohui Fu and Sharad Malik. On solving the partial MAX-SAT problem. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT-06), Seattle, WA, USA*, pages 252–265. Springer-Verlag, August 2006.

[22] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque. Congolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence (AIJ)*, 121(1-2):109–169, 2000.

[23] Enrico Giunchiglia and Marco Maratea. Planning as satisfiability with preferences. In *Proceedings of the 22nd Conference on Artificial Intelligence (AAAI-07), Vancouver, BC, Canada*. AAAI Press, July 2007.

[24] Enrico Giunchiglia, Alessandro Massarotto, and Roberto Sebastiani. Act, and the rest will follow: Exploiting determinism in planning as satisfiability. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98), Madison, WI, USA*. AAAI Press, July 1998.

[25] Peter Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions On Systems Science And Cybernetics*, 4(2):100–107, 1968.

[26] Patrik Haslum. Additive and reversed relaxed reachability heuristics revisited. In *Prococeedings of the 2008 Internaltional Planning Competition, (IPC-6), Sydney, Australia*, September 2008.

[27] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS-09), Thessaloniki, Greece*. AAAI Press, September 2009.

[28] Jörg Hoffmann, Carla P. Gomes, Bart Selman, and Henry A. Kautz. SAT encodings of state-space reachability problems in numeric domains. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07), Hyderabad, India*, pages 1918–1923, January 2007.

[29] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07), Hyderabad, India*, pages 2318–2323. Morgan Kaufmann, January 2007.

[30] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. A novel transition based encoding scheme for planning as satisfiability. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10), Atlanta, GA, USA*. AAAI Press, July 2010.

[31] Farah Juma, Eric I. Hsu, and Sheila A. McIlraith. Exploiting MaxSAT for preference-based planning. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling Workshop on Constraint Satisfaction Techniques for Planning and Scheduling (COPLAS-11), Freiburg, Germany*, June 2011.

[32] Henry Kautz. SATPLAN04: Planning as satisfiability. In *Proceedings of the 2004 International Planning Competition (IPC-4), Whistler, BC, Canada*, June 2004.

[33] Henry Kautz. Deconstructing planning as satisfiability. In *Proceedings of The 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence Conference (IAAI-06), Boston, MA, USA*. AAAI Press, July 2006.

[34] Henry Kautz, David McAllester, and Bart Selman. Encoding plans in propositional logic. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning (KR-96), Boston, MA, USA*. AAAI Press, November 1996.

[35] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-92), Vienna, Austria*, pages 359–363. John Wiley and Sons, August 1992.

[36] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the 13th National Conference on Artificial Intelligence and 8th Innovative Applications of Artificial Intelligence Conference (IAAI-96), Portland, OR, USA*, pages 1194–1201. AAAI Press / The MIT Press, August 1996.

[37] Henry Kautz and Bart Selman. The role of domain-specific knowledge in the planning as satisfiability framework. In *Proceedings of the 4th International Conference on Artificial Intelligence Planning System (AIPS-98), Pittsburgh, PA, USA*, pages 181–189. The AAAI Press, June 1998.

[38] Henry Kautz and Bart Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99), Stockholm, Sweden*, pages 318–325. Morgan Kaufmann, August 1999.

[39] Henry Kautz, Bart Selman, and Jörg Hoffmann. SatPlan: Planning as satisfiability. In *Proceedings of the 2006 International Planning Competition (IPC-5), Lake District, Cumbira, UK*, June 2006.

[40] Hector Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming (JLP)*, 31(1-3):59–83, 1997.

[41] Michael Littman, Stephen Majercik, and Toniann Pitassi. Stochastic boolean satisfiability. *Journal of Automated Reasoning (JAR)*, 27(3):251–296, 2001.

[42] Stephen Majercik. APPSSAT: Approximate probabilistic planning using stochastic satisfiability. *International Journal of Approximate Reasoning (IJAR)*, 45(2):402–419, 2007.

[43] João Marques-Silva and Karem Sakallah. GRASP – a new search algorithm for satisfiability. In *Proceedings of The International Conference on Computer-Aided Design (ICCAD-96), San Jose, CA, USA*, pages 220–227. IEEE Press, November 1996.

[44] Robert Mattmüller and Jussi Rintanen. Planning for temporally extended goals as propositional satisfiability. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07), Hyderabad, India*, pages 1966–, January 2007.

[45] John McCarthy and Patrick Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence (MI)*, pages 463–502, 1969.

[46] Ruy Luiz Milidiú, Frederico dos Santos Liporace, and Carlos José P. de Lucena. Pipesworld: Planning pipeline transportation of petroleum derivatives. In *Proceedings of the 13th International Conference on Automated Planning and Scheduling Workshop on the Competition:*

Impact, organization, evaluation, benchmarks (ICAPS-03), Trento, Italy. AAAI Press, June 2003.

[47] Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC-01), Las Vegas, NV, USA*, pages 530–535. ACM, June 2001.

[48] Hidetomo Nabeshima, Takehide Soh, Katsumi Inoue, and Koji Iwanuma. Lemma reusing for SAT based planning and scheduling. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS-06), Lake District, Cumbria, UK*, pages 103–113. AAAI Press, June 2006.

[49] Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. SHOP2: An htn planning system. *Journal of Artificial Intelligence Research (JAIR)*, 20:379–404, 2003.

[50] Dana S. Nau, Yue Cao, Amnon Lotem, and Héctor Muñoz-Avila. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99), Stockholm, Sweden*, pages 968–975, August 1999.

[51] Edwin Pednault. Adl and the state-transition model of action. *Journal of Logic and Computation (JLC)*, 4(5):467–512, 1994.

[52] Knot Pipatsrisawat and Adnan Darwiche. RSat 2.0: SAT solver description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA, 2007.

[53] Katrina Ray and Matthew L. Ginsberg. The complexity of optimal planning and a more efficient method for finding solutions. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS-08), Sydney, Australia*, pages 280–287. AAAI Press, September 2008.

[54] Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research (JAIR)*, 39:127–177, 2010.

[55] Jussi Rintanen. Evaluation strategies for planning as satisfiability. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence (ECAI-04), Valencia, Spain*, pages 682–687. IOS Press, August 2004.

[56] Jussi Rintanen. Planning graphs and propositional clause learning. In *Proceedings of the 11th International Conference on the Principles of Knowledge Representation and Reasoning (KR-08), Sydney, Australia*, pages 535–543. AAAI Press, September 2008.

[57] Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence (AIJ)*, 170(12-13):1031–1080, 2006.

[58] Nathan Robinson, Charles Gretton, and Duc-Nghia Pham. CO-Plan: Combining SAT-based planning with forward-search. In *Prococeedings of the 2008 Internaltional Planning Competition, (IPC-6), Sydney, Australia*, September 2008.

[59] Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. A compact and efficient SAT encoding for planning. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS-08), Sydney, Australia*, pages 296–303. AAAI Press, September 2008.

[60] Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. SAT-based parallel planning using a split representation of actions. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS-09), Thessaloniki, Greece*. AAAI Press, September 2009.

[61] Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. Cost-optimal planning using weighted MaxSAT. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling Workshop on Constraint Satisfaction Techniques for Planning and Scheduling (COPLAS-10), Toronto, Canada*, May 2010.

[62] Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. Partial weighted MaxSAT for optimal planning. In *Proceedings of the 11th Pacific Rim International Conference on Artificial Intelligence (PRICAI-10), Daegu, Korea*. Springer, August 2010.

[63] Richard Russell and Sean Holden. Handling goal utility dependencies in a satisfiability framework. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS-10), Toronto, ON, Canada*, pages 145–152. AAAI Press, May 2010.

[64] Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence (AIJ)*, 5(2):115–135, 1974.

[65] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of The 12th National Conference on Artificial Intelligence (AAAI-94), Seattle, WA, USA*, pages 337–343. AAAI Press, August 1994.

[66] Ji-Ae Shin and Ernest Davis. Processes and continuous change in a SAT-based planner. *Artificial Intelligence (AIJ)*, 166(1-2):194–253, 2005.

[67] Andreas Sideris and Yannis Dimopoulos. Constraint propagation in propositional planning. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS-10), Toronto, ON, Canada*, pages 153–160. AAAI Press, May 2010.

[68] Shirin Sohrabi, Jorge A. Baier, and Sheila A. McIlraith. Diagnosis as planning revisited. In *Proceedings of the 12th International Conference on the Principles of Knowledge Representation and Reasoning (KR-10), Toronto, Canada*, pages 26–36. AAAI Press, May 2010.

[69] Tran Cao Son, Chitta Baral, Tran Hoai Nam, and Sheila A. McIlraith. Domain-dependent knowledge in answer set planning. *ACM Transactions on Computational Logic (TOCL)*, 7(4):613–657, 2006.

[70] Matt Streeter and Stephen Smith. Using decision procedures efficiently for optimization. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS-07), Providence, RI, USA*, pages 312–319. AAAI Press, September 2007.

[71] Sylvie Thiébaux and Marie-Odile Cordier. Supply restoration in power distribution systems – a benchmark for planning under uncertainty. In *Proceedings of The 6th European Conference on Planning: Recent Advances in AI Planning (ECP-01), Toledo, Spain*, pages 85–95. Springer-Verlag, September 2001.

[72] Reiko Tsuneto, James A. Hendler, and Dana S. Nau. Analyzing external conditions to improve the efficiency of htn planning. In *Proceedings of the 15th National Conference on Artificial Intelligence and the 10th Innovative Applications of Artificial Intelligence Conference (IAAI-98), Madison, WI, USA*, pages 913–920. AAAI Press, July 1998.

[73] Martin Wehrle and Jussi Rintanen. Planning as satisfiability with relaxed $\exists$-step plans. In *Proceedings of 20th Australasian Joint Conference on Artificial Intelligence (AI-07), Gold Coast, Australia*, pages 244–253. Springer, December 2007.

[74] Steven Wolfman and Daniel Weld. The LPSAT engine & its application to resource planning. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99), Stockholm, Sweden*, pages 310–317. Morgan Kaufmann, August 1999.

Appendix A

Example Planning Domains

In this appendix we provide the PDDL domain definitions for the example planning domains introduced in Section 2.1.

A.1 SIMPLE-LOGISTICS

SIMPLE-LOGISTICS is a simplified version of the IPC-1 domain LOGISTICS. There is a set of locations connected by roads, a set of trucks, and a set of packages. Each package can be either at a location or in a truck. There is a set of actions which allow trucks to drive between locations along roads and to pick-up and drop-off packages. Note that here, rather than defining two `at` predicates for trucks and packages, we define one that takes a type $locatable$ and define $truck$ and $package$ to be subtypes of $locatable$. The problem domain is represented by the PDDL in Figure A.1.

A.2 BLOCKS-DIRECT

BLOCKS-DIRECT is based on the IPC-2 domain BLOCKS. There is a table of unbounded size and a set of blocks that can be stacked, either upon the table, or upon other blocks. There are actions that move a block between the table and the top of another block and actions that move a block between the tops of other blocks. The problem domain is represented by the PDDL in Figure A.2.

```
(define (domain SIMPLE-LOGISTICS)
  (:requirements :typing :action-costs)
  (:types location locatable - object
          truck package - locatable)
  (:predicates (at ?t - locatable ?l - loaction)
               (in ?p - package   ?t - truck)
  (:functions (total-cost) - number)

  (:action Drive
   :parameters (?t - truck ?l1 - location ?l2 - location)
   :precondition (and (at ?t ?l1) (road ?l1 ?l2))
   :effect (and (not (at ?t ?l1)) (at ?t ?l2)
                (increase (total-cost) 1))
           )
  )

  (:action Pick-Up
   :parameters (?t - truck ?l - location ?p - package)
   :precondition (and (at ?t ?l) (at ?p ?l))
   :effect (and (not (at ?p ?l)) (in ?p ?t)
                (increase (total-cost) 1))
           )
  )

  (:action Drop-Off
   :parameters (?t - truck ?l - location ?p - package)
   :precondition (and (at ?t ?l) (in ?p ?t))
   :effect (and (not (in ?p ?t)) (at ?p ?l)
                (increase (total-cost) 1))
           )
  )
)
```

Figure A.1: SIMPLE-LOGISTICS PDDL domain definition.

```
(define (domain BLOCKS-DIRECT)
  (:requirements :typing :action-costs)
  (:types block)
  (:predicates (on ?x - block  ?y - block) (on-table ?x - block)
               (clear ?x - block))
  (:functions (total-cost) - number)

  (:action Move
         :parameters (?x - block ?y - block ?z - block)
         :precondition (and (clear ?x) (on ?x ?y) (clear ?z))
         :effect (and (not (on ?x ?y)) (not (clear ?z))
                      (on ?x ?z) (clear ?y)
                      (increase (total-cost) 1)
         )
  )

  (:action Move-Table-Block
         :parameters (?x - block ?y - block)
         :precondition (and (on-table ?x) (clear ?x) (clear ?y))
         :effect (and (not (on-table ?x)) (not (clear ?y))
                      (on ?x ?y)
                      (increase (total-cost) 1)
         )
  )

  (:action Move-Block-Table
         :parameters (?x - block ?y - block)
         :precondition (and (on ?x ?y) (clear ?x))
         :effect (and (not (on ?x ?y))
                      (on-table ?x) (clear ?y)
                      (increase (total-cost) 1)
         )
  )
)
```

Figure A.2: BLOCKS-DIRECT PDDL domain definition.