SAT-Based Parallel Planning Using a Split Representation of Actions

Nathan Robinson[†], Charles Gretton[‡], Duc-Nghia Pham[†], Abdul Sattar[†]
[†]ATOMIC Project, Queensland Research Lab, NICTA and
Institute for Integrated and Intelligent Systems, Griffith University, QLD, Australia
{nathan.robinson,duc-nghia.pham,abdul.sattar}@nicta.com.au
[‡] School of Computer Science, University of Birmingham
c.gretton@cs.bham.ac.uk

Abstract

Planning based on propositional SAT(isfiability) is a powerful approach to computing *step-optimal* plans given a *parallel* execution semantics. In this setting: (i) a solution plan must be minimal in the number of plan steps required, and (ii) non-conflicting actions can be executed instantaneously in parallel at a plan step. Underlying SAT-based approaches is the invocation of a decision procedure on a SAT encoding of a bounded version of the problem. A fundamental limitation of existing approaches is the size of these encodings. This problem stems from the use of a *direct* representation of actions – i.e. each action has a corresponding variable in the encoding. A longtime goal in planning has been to mitigate this limitation by developing a more compact *split* – also termed *lifted* – representation of actions in SAT encodings of parallel step-optimal problems. This paper describes such a representation. In particular, each action and each parallel execution of actions is represented uniquely as a conjunct of variables. Here, each variable is derived from action pre and post-*conditions*. Because multiple actions share *conditions*, our encoding of the planning constraints is factored and relatively compact. We find experimentally that our encoding yields a much more efficient and scalable planning procedure over the state-of-the-art in a large set of planning benchmarks.

Introduction

Optimal domain-independent planning approaches that exploit state-of-the-art propositional SAT(isfiability) procedures usually exhibit excellent performance. The winner of the optimal track at the fourth International Planning Competition (IPC-4) was SATPLAN-04, and at IPC-5 SATPLAN-06 (Kautz, Selman, & Hoffmann 2006) and MAXPLAN (Chen, Xing, & Zhang 2007) tied for first place. Based on BLACKBOX (Kautz & Selman 1999), these solvers are step-optimal in a parallel setting because: (1) a solution plan must be minimal in the number of plan steps required, and (2) non-conflicting actions can be executed instantaneously in parallel at a plan step. They operate by iteratively constructing propositional plangraphs for successive horizon lengths (e.g., $h = 1, 2, 3, ...$ plan steps), at each horizon performing the ubiquitous *reachability* and *neededness* plangraph analysis (Blum & Furst 1997). The bounded planning problem posed by the plangraph at h is compiled into a *conjunctive normal form* (CNF) propositional formula whose solutions, if any, correspond to parallel h-step plans. Successively generated CNFs are input to a satisfiability-testing procedure which determines whether the formula has any satisfying models, and if so reveals one of these. Iteration terminates once a solution is found. The compilation to SAT is *direct* in the sense that each variable in the CNF corresponds to whether an action is executed at a plan step, or whether a proposition is true at a plan step. The planning constraints, such as *frame axioms*, *conflict exclusion*, and that *an action implies its precondition and effects*, are encoded naturally in terms of those variables. Not surprisingly then, a fundamental limitation of BLACKBOX and its successors is *size blowup*, a problem that refers to the enormous sized CNFs they generate. For example, the 11-step decision problem generated by SATPLAN-06 for IPC-5 problem PIPESWORLD-9 has over 11 million clauses.

Recently there has been a significant research effort invested in approaches which improve the overall efficiency of step-optimal planning as SAT in a parallel setting. Generally speaking, such proposals have been in two directions. The first seeks query strategies that are more efficient than the *de facto* standard – i.e., where a SAT solver is instantiated at incremental horizon lengths. The second seeks to retain constraints (e.g., *conflict clauses*) "learned" during processing at horizon h at successive horizons $h + 1, h + 2, ...$ In more detail, optimal systems discussed above use simple ramp-up $h = 1, 2, 3, ...,$ and ramp-down $h = h_{\texttt{big}}, h_{\texttt{big}-1}, ..., 1$ query strategies. For ramp-down, also called *backwards level reduction*, $h_{\texttt{big}}$ is obtained in practice by querying a satisficing planner. Better efficiency has been demonstrated for interleaved and parallel query strategies described by Rintanen (2004), and for efficient variants of binary search described by Streeter & Smith (2007). In the case of constraint retention between queries, Nabeshima *et al.* (2006) 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. In a similar vein, Ray & Ginsberg (2008) introduce SAT-based system CRICKET that uses modified SAT procedures that exploit a predefined branching order guaranteeing optimality whatever the queried 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 & Ginsberg (2008) 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 blowup, which in these works remains a hindrance to scalability and efficiency.

Recent times have also seen proposed SAT-based procedures and encodings for step-optimal planning with a serial execution semantics, and approximately step-optimal planning with a parallel execution semantics. In these settings lateral approaches have been devised for delivering scalability and efficiency. In the case of serial execution semantics, only one action can be executed at a plan step. Ernst, Millstein, & Weld (1997) developed the system MEDIC which implemented a number of encodings of serial planning as SAT. The problem of CNF size blowup in MEDIC was mitigated by using a split representation of actions – An idea originally proposed by Kautz & Selman (1992). For the serial case splitting yields a much more compact representation of the planning constraints compared with a direct encoding along the lines of BLACKBOX and its successors. In practice however, parallel planning with a direct encoding is easier than serial planning with a split encoding. This is due to the comparatively long horizon required for the serial case. For example, for IPC-5 problem ROVERS-07, the optimal serial plan requires 18 steps, whereas SATPLAN-06 requires only 5 steps. To date, no proposals for a compact split representation of actions has been made for step-optimality in a parallel setting.

In the case of approximately optimal planning with a parallel execution semantics, two directions are proposed to mitigate the problem of size blowup for encoding the bounded problems that arise in a SAT-based setting. Rintanen, Heljanko, & Niemelä (2006) and Wehrle & Rintanen (2007) propose reducing the number of queries required to find a plan by relaxing constraints on action parallelism. Their works exploit the concept of *post-serialisability* from Dimopoulos, Nebel, & Koehler (1997), thereby allowing a set of conflicting actions to be prescribed at a single plan step provided a scheme, computed *a priori*, is available for generating a valid serial execution. In this case, the h step problem posed to a SAT procedure has solutions that correspond to parallel plans that require more than h steps to execute without conflict. Hence, parallel step-optimality is not guaranteed. In a somewhat opposite direction Robinson *et al.* (2008) proposed an encoding that lies between the serial and parallel case. They use the split representation of actions from MEDIC in a semi-parallel setting. In that work, legal parallel executions of non-conflicting actions are forbidden where the action representation is not sufficiently rich to describe them. Thus, again parallel step-optimality is not guaranteed.

Unlike recent research directed at improving the efficiency of step-optimal planning in a parallel setting, this paper tackles the problem of size blowup directly. In doing so, we are able to demonstrate improvements in both the efficiency and scalability of optimal planning. Our direction takes the concepts of operator splitting from the work of Ernst, Millstein, & Weld (1997) in the serial setting, and Robinson *et al.* (2008) in the approximate parallel setting, and brings them to the optimal parallel setting. The main consequence of our split representation is that we require fewer clauses in our encoding of the h horizon problem than BLACKBOX-like systems that use a *direct* encoding. For example, looking again at the 11-step IPC-5 problem PIPESWORLD-09, SATPLAN-06 required ~11M clauses whereas our split representation requires only ~670k. In practice the relative compactness of our encoding translates to better efficiency and scalability over the state-of-the-art. Finally, in our work we use the plangraph as the source of our compilation from planning to SAT. This is because plangraph analysis yields mutex constraints between pairs of actions and pairs of state propositions that are: (1) useful, and feature, in state-of-the-art satisfiability procedures (Kautz 2006), and (2) are not redundant, because such mutex relations are not deduced independently by modern SAT procedures (Rintanen 2008).

This paper is organised as follows. We summarise the propositional planning problem and introduce our notations. Following this we review existing SAT representations of actions, consider the problem of interference that occurs using existing split representations, and then develop a split representation of actions without interference. We then present a compilation of the bounded problem posed by a plangraph into SAT using our split representation of actions. We empirically evaluate our approach on planning benchmarks from a number of the International Planning Competitions. Finally we make concluding remarks and propose directions for future research.

Background and Notations

Propositional Planning

A propositional planning problem is given in terms of a finite set of objects Σ, first-order STRIPS-like planning operators of the form $\langle o, \mathbb{C} \rangle$, and predicates Π. By grounding Π over Σ we obtain the set of propositions P that characterises problem states. For example, in a blocks-world problem we can have two blocks $A, B \in \Sigma$, and a binary predicate $\texttt{On}(x_1, x_2) \in \Pi$, one grounding of which is the proposition $\texttt{On}(A, B) \in P$.

An operator o is an expression of the form $O(\overrightarrow{x})$ where O is an operator name and $\overrightarrow{x} = x_1, .., x_k$ is a list of variable symbols. Our $\mathbb{C}$ notation departs from typical expositions in planning. $\mathbb{C}$ is a set of elements which describe the operator pre-*conditions*, and negative/positive post-*conditions*. Intuitively, each element $\mathcal{C} \in \mathbb{C}$ is a composite, containing the set of conditions that range over the same argument variables. Writing $\overrightarrow{x}'$ for an argument list made up of elements from $\overrightarrow{x}$, for $\pi \in \Pi$ a basic operator condition has the form $\tau(\pi(\overrightarrow{x}'))$. Here τ denotes a condition type, either: (1) a basic precondition, written $\texttt{PRE}(\pi(\overrightarrow{x}'))$, (2) a basic negative postcondition $\texttt{DEL}(\pi(\overrightarrow{x}'))$, or (3) a basic positive postcondition $\texttt{ADD}(\pi(\overrightarrow{x}'))$. For example, the operator $\texttt{Move}(x_1, x_2, x_3)$ from blocks-world has the basic conditions: $\texttt{PRE}(\texttt{On}(x_1, x_2))$, $\texttt{PRE}(\texttt{Clear}(x_1))$, $\texttt{PRE}(\texttt{Clear}(x_3))$, $\texttt{DEL}(\texttt{On}(x_1, x_2))$, $\texttt{DEL}(\texttt{Clear}(x_3))$, $\texttt{ADD}(\texttt{On}(x_1, x_3))$, and $\texttt{ADD}(\texttt{Clear}(x_2))$. We define elements in $\mathbb{C}$ to be composites of basic conditions as follows. Let $\alpha(\overrightarrow{x}')$ be any permutation of argument list $\overrightarrow{x}'$. If there is any pair $\tau_1(\pi_1(\overrightarrow{x}'))$ and $\tau_2(\pi_2(\alpha(\overrightarrow{x}')))$ of basic conditions of an operator $\langle o, \mathbb{C}\rangle$, then there is exactly one $\mathcal{C} \in \mathbb{C}$ so that $\tau_1(\pi_1(\overrightarrow{x}')), \tau_2(\pi_2(\alpha(\overrightarrow{x}'))) \in \mathcal{C}$. For example, the previously mentioned Move operator can be described in terms of $\mathbb{C}$ as follows:

$$\begin{array}{l}\langle \texttt{Move}(x_1, x_2, x_3),\\ \quad \{\{\texttt{PRE}(\texttt{On}(x_1,x_2)), \texttt{DEL}(\texttt{On}(x_1,x_2))\},\\ \quad \{\texttt{PRE}(\texttt{Clear}(x_3)), \texttt{DEL}(\texttt{Clear}(x_3))\},\\ \quad \{\texttt{PRE}(\texttt{Clear}(x_1))\}, \{\texttt{ADD}(\texttt{On}(x_1,x_3))\}, \{\texttt{ADD}(\texttt{Clear}(x_2))\}\}\rangle\end{array}$$

Planning actions are obtained by grounding operator descriptions over object symbols. Grounding according to assignment $\{x_1 = A, x_2 = B, x_3 = C\}$ yields STRIPS action:

$$\begin{array}{lll}\multicolumn{3}{l}{\texttt{Action: } a = \texttt{Move } (A, B, C)}\\ \quad \textit{pre}(a) & := & [\texttt{On}(A,B), \texttt{Clear}(A), \texttt{Clear}(C)];\\ \quad \textit{add}(a) & := & [\texttt{Clear}(B), \texttt{On}(A,C)];\\ \quad \textit{del}(a) & := & [\texttt{On}(A,B), \texttt{Clear}(C)];\end{array}$$

We write a for the action having a set of ground preconditions *pre*(a), positive post-conditions *add*(a), and negative post-conditions *del*(a). As for the above ground Move example, these lists contain elements from P. Ground conditions associated with a are notated $\mathbb{C}_a$, and elements in that set $\mathcal{C}_a \in \mathbb{C}_a$. An action a can be executed at a state $s \subseteq P$ when *pre*$(a) \subseteq s$. We denote $\mathcal{A}$ the set of problem actions, and $\mathcal{A}(s)$ the set of actions that can be executed at state s. When $a \in \mathcal{A}(s)$ is executed at s the resultant state is $(s \cup \textit{add}(a)) \backslash \textit{del}(a)$. Actions cannot both add and delete the same proposition – i.e., *add*$(a) \cap$ *del*$(a) \equiv \emptyset$.[1] A set of actions is said to be non-conflicting if any serial execution of the actions, at any state, produces the same outcome.

A planning problem is posed in terms of a starting state $s_0 \subseteq P$, a goal $\mathcal{G} \subseteq P$, and a small set of domain operators. Supposing non-conflicting actions can be executed instantaneously in parallel, a *parallel plan* is a discrete sequence of time-indexed sets of non-conflicting actions which, when applied to the start state, lead to a goal state. For a *serial plan* (a.k.a. *linear plan*) each time-indexed set contains one action. Whether the format is parallel or serial, we say that a plan is (step-)optimal iff no other plan of the same format can reach a goal state using fewer steps.

Action Representations

A number of propositional representations of actions have been proposed for planning as satisfiability. In decreasing order in the number of variables required for serial planning, these include: (1) *direct*, (2) *simply split*, and (3) *bitwise*. In the *direct* case, each action corresponds to a Boolean variable in the SAT formula that represents whether the action is executed. All existing step-optimal systems that adopt a parallel execution semantics use this *direct* encoding. In the *simply split* case, each legal binding to each operator argument is represented by a variable. For example, writing $\texttt{Move}[k](x)$ for the predicate that says "the k'th argument of action Move is x", we have that action $\texttt{Move}(A, B, C)$ is represented by the conjunct $\texttt{Move}[1](A) \wedge \texttt{Move}[2](B) \wedge \texttt{Move}[3](C)$.[2] 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 $p_0 \wedge \neg p_1 \wedge p_2$. Hence, a bitwise encoding requires the fewest variables, and theoretically should produce the easiest SAT problem among the three representations. However, Ernst, Millstein, & Weld (1997) and others found that the performance of encodings based on bitwise representation is worse than those based on other representations.

Factoring and Interference

Simply split representations of actions have been the subject of a significant body of work in the planning as SAT paradigm (Robinson *et al.* 2008; Giunchiglia, Massarotto, & Sebastiani 1998; Ernst, Millstein, & Weld 1997; Kautz & Selman 1992). Its major advantage is compactness due to *factorisation* of constraints, whilst its major disadvantage is *interference*.

Two observations are necessary to see the main advantage of simple splitting over a direct representation. First, a simply split encoding of actions requires fewer variables than the direct case – e.g., in the split case grounding an n-ary operator over Σ results in order $n|\Sigma|$ variables compared to order $|\Sigma|^n$ in the direct case. Second, there are relatively few constraints, such as *frame axioms*, *conflict exclusion*, and that *an action implies its precondition and effects*, in the compiled problems because these can be factored. For example, precondition $\texttt{Move}(A, B, C) \rightarrow \texttt{Clear}(A)$ is written $Move[1](A) \rightarrow \texttt{Clear}(A)$. Due to these advantages, it is more efficient to plan in a simply split encoding of a serial planning problem than in its direct counterpart.

In the comparatively easier setting of parallel planning, *interference* prevents the use of a simply split representation. In particular, simple splitting is imprecise because variables cannot generally be used to encode parallel executions of actions. For example, parallel execution of 2 non-conflicting actions $\texttt{Move}(A, B, C)$ and $\texttt{Move}(D, E, F)$ interferes because it is represented by a conjunct of $\texttt{Move}[1](A) \wedge \texttt{Move}[2](B) \wedge \texttt{Move}[3](C) \wedge \texttt{Move}[1](D) \wedge \texttt{Move}[2](E) \wedge \texttt{Move}[3](F)$. This implies 6 additional instances of Move corresponding to: $\texttt{Move}(A, B, F)$, $\texttt{Move}(A, E, C)$, $\texttt{Move}(A, E, F)$, $\texttt{Move}(D, B, C)$, $\texttt{Move}(D, B, F)$, and $\texttt{Move}(D, E, C)$. In summary, interference prevents the simply split representation of actions from being used for encodings of parallel planning problems.

Splitting Actions Without Interference

Interference occurs when variables in a simply split representation are imprecise in the parallel setting. We now develop an interference free split representation called *precisely split*. Where a is an instantiation of an action with operator name O, this representation requires a condition variable OC_a for every ground condition $\mathcal{C}_a \in \mathbb{C}_a$. The action a

[1]This restriction does not apply in practice, and the case is given a special semantics. The details will not be discussed in this paper.

[2]Ernst, Millstein, & Weld (1997) also devised an overloaded variant of simple splitting. This is an important concept, however irrelevant to this discussion.

then has a natural representation as conjunct:

$$\bigwedge_{\mathcal{C}_a \in \mathbb{C}_a} O\mathcal{C}_a$$

For example, the blocks-world action $\texttt{Move}(A, B, C)$ is represented by the conjunct:

$$\begin{aligned}\texttt{Move}\{\texttt{ADD}(\texttt{Clear}(B))\} \wedge \texttt{Move}\{\texttt{PRE}(\texttt{On}(A,B)), \texttt{DEL}(\texttt{On}(A,B))\} \wedge \\ \texttt{Move}\{\texttt{PRE}(\texttt{Clear}(C)), \texttt{DEL}(\texttt{Clear}(C))\} \wedge \\ \texttt{Move}\{\texttt{PRE}(\texttt{Clear}(A))\} \wedge \texttt{Move}\{\texttt{ADD}(\texttt{On}(A,C))\}\end{aligned}$$

Before presenting our encoding we sketch two consequences of *precise splitting*. First, as required for any useful lifted representation, every action is determined by a unique conjunct. It follows that a precisely split representation can be used to encode step-optimal serial planning in SAT. Second, every parallel execution of actions is determined by a unique conjunct over conditional variables, except where redundant actions occur. In more detail, for any action a and a set of non-conflicting actions $\mathcal{A}$ so that $a \in \mathcal{A}$, we say a is *redundant* with respect to parallel execution of actions $\mathcal{A}$ iff at any state any execution of $\mathcal{A}$ is indistinguishable from any execution of $\mathcal{A} \backslash a$. Intuitively this type of redundancy occurs when the conditions representing an action are a strict subset of those representing a different set of actions. For example, suppose that ψ_1 and ψ_2 are the unique conjuncts representing two actions a_1 and a_2. Conjunct $\psi_1 \wedge \psi_2$ could imply three or more distinct actions whose representations are given by conjuncts $\psi_1, \psi_2, .., \psi_n$, corresponding to distinct actions $a_1, a_2, .., a_n$. If this were the case, for $i > 2$, actions a_i are *redundant* with respect to parallel executions a_1, a_2, a_i.

In summary, our precisely split representation yields a unique conjunct for every unique parallel execution of non-conflicting actions. Hence, a precisely split representation can be used to encode step-optimal parallel planning in SAT.

Compilation to SAT

We describe a compilation of the bounded planning problem posed by an h-step plangraph into SAT.[3] The result of compilation is a conjunctive normal form (CNF) propositional formula ϕ. Our encoding is constructive (and sound) in the sense that any satisfying model of ϕ corresponds more-or-less directly with a plan. We also have completeness because ϕ is satisfiable iff a plan exists for the original bounded planning problem.

Below we give our compilation to SAT in terms of axiom Schemata. Each individual Schema specifies how a planning constraint from the plangraph is represented in ϕ. The compilation makes use of the following propositional variables. For each propositional fluent p occurring at step $t = 1, ..., h$ we have a variable p^t.[4] We write $\mathcal{A}^t$ for the set of ground actions occurring at step t, $\mathbb{C}^t_a$ for the set of all ground conditions associated with action $a^t \in \mathcal{A}^t$, and $\mathbb{C}^t$ for the union of ground conditions occurring at step t – i.e., $\mathbb{C}^t \equiv \bigcup_{a^t \in \mathcal{A}^t} \mathbb{C}^t_a$. For our compilation we have a *condition variable* $O\mathcal{C}^t$ for each $\mathcal{C}^t \in \mathbb{C}^t$. To indicate that $\mathcal{C}^t \in \mathbb{C}^t_a$ for variable $O\mathcal{C}^t$, we use the notation $O\mathcal{C}^t_a$.

In addition to the above variables, we also introduce *auxiliary* condition variables when compiling mutex relations between instantiations of the same operator (see *Intra-operator mutex* axioms, Schema 6). Copies of condition variables, called *condition copy* variables, are introduced when constraining parallel groundings of an operator (see *Intra-operator mutex* axioms, Schema 7). Finally, we avoid annotating variables with their time index if those follow clearly from the context.

1. Start state and goal axioms: A unit clause containing p^0 occurs for every $p \in s_0$. For each $p \in \mathcal{G}$ we have a unit clause containing p^h.

2. Precondition and postcondition axioms: This Schema encodes action effects using the corresponding condition variables. For each variable $O\mathcal{C}^t$ we have the following constraint:

$$\begin{array}{ll}\bigwedge_{\texttt{PRE}(p) \in \mathcal{C}^t} O\mathcal{C}^t \rightarrow p^t & \wedge \\ \bigwedge_{\texttt{ADD}(p) \in \mathcal{C}^t} O\mathcal{C}^t \rightarrow p^{t+1} & \wedge \\ \bigwedge_{\texttt{DEL}(p) \in \mathcal{C}^t} O\mathcal{C}^t \rightarrow \neg p^{t+1} & \end{array}$$

3. Successor state axioms (i.e., frame axioms): These constrain how the truth values of fluents change over successive plan steps. For proposition p^t, let $\texttt{make}(p^t)$ be the set of conditions in $\mathbb{C}^{t-1}$ containing an element of the form $\texttt{ADD}(p)$. Then for each $p^t, t > 0$ we have:

$$p^t \rightarrow (p^{t-1} \vee \bigvee_{\mathcal{C}^{t-1} \in \texttt{make}(p^t)} O\mathcal{C}^{t-1})$$

This says that if p is true at step t, then either it was true previously, or an action was executed with an add condition that makes it true. For example, in a blocks-world domain for $\texttt{On}(A, B)$ we have a clause:

$$\texttt{On}(A,B)^t \rightarrow (\texttt{On}(A,B)^{t-1} \vee \texttt{Move}\{\texttt{ADD}(\texttt{On}(A,B)^{t-1})\})$$

4. Propositional mutex axioms: For every pair of state propositions p^t_1 and p^t_2 that occur mutex in layer t of the plangraph, we have a clause:

$$\neg p^t_1 \vee \neg p^t_2$$

5. Inter-operator mutex axioms: Schemata 2 and 4 prohibit parallel instantiation of $a, b \in \mathcal{A}$ provided either

a. Schema 2 gives constraints of the form $O_1\mathcal{C}^t_a \rightarrow p^{t+1}_1$ and $O_2\mathcal{C}^t_b \rightarrow \neg p^{t+1}_1$; or

b. Schema 2 gives constraints of the form $O_1\mathcal{C}^t_a \rightarrow p^t_1$ and $O_2\mathcal{C}^t_b \rightarrow p^t_2$, and Schema 4 gives $\neg p^t_1 \vee \neg p^t_2$.

Consequently, we do not have to enforce mutex between conditions – and therefore between their corresponding actions – where this is already implied by Schemata 2 and 4. In the remaining case we have mutex actions $a, b \in \mathcal{A}$ so that in Schema 2 $O_1\mathcal{C}^t_a \rightarrow p^t$ and $O_2\mathcal{C}^t_b \rightarrow \neg p^{t+1}$ occur. Provided $(O_1 \neq O_2)$, we have the clause:

$$\neg O_1\mathcal{C}^t_a \vee \neg O_2\mathcal{C}^t_b$$

[3]There is insufficient space to describe the plangraph here, thus we assume familiarity with (Blum & Furst 1997).

[4]A fluent is a state proposition whose truth value can be modified by executing an action.

For example, in logistics $O_1\mathcal{C}_a^t$ can be $\texttt{Load}\{\texttt{PRE}(\texttt{At}(truck_1, loc_1))\}^t$ and $O_2\mathcal{C}_b^t$ can be $\texttt{Drive}\{\texttt{PRE}(\texttt{At}(truck_1, loc_1)), \texttt{DEL}(\texttt{At}(truck_1, loc_1))\}^t$.

6. Intra-operator mutex axioms: We are again concerned with enforcing mutex not already implied by Schemata 2 and 4. Unlike Schema 5, that expressed mutex between instantiations of distinct operators, here we have mutex actions a and b that instantiate the same operator O. Mutex between these actions will be compiled in the context of mutex conditions $\mathcal{C}_a$ and $\mathcal{C}_b$ according to the following rules:

a. Suppose either: (i) $\mathcal{C}_a = \mathcal{C}_b$, or (ii) there is an action c with distinct conditions $\mathcal{C}_c$ and $\mathcal{C}'_c$ so that $\mathcal{C}_c = \mathcal{C}_a$ and $\mathcal{C}'_c = \mathcal{C}_b$. Situation i occurs in the blocks-world for $\texttt{Move}\{\texttt{PRE}(\texttt{On}(A, B)), \texttt{DEL}(\texttt{On}(A, B))\}$, which is responsible for the mutex relationship between actions $\texttt{Move}(A, B, C)$ and $\texttt{Move}(A, B, D)$, and $\texttt{Move}(A, B, C)$ and $\texttt{Move}(A, B, E)$, etc. Situation ii does not occur in practice, however could occur theoretically and is thus included here.

   Direct compilation of mutex between $\mathcal{C}_a$ and $\mathcal{C}_b$ given the above situations is not admissible. Thus, we represent mutex between a and b using auxiliary condition variables that are generated as needed. We choose two such variables $O\mathcal{C}'$ and $O\mathcal{C}''$, and later enforce $a \rightarrow O\mathcal{C}'$ and $b \rightarrow O\mathcal{C}''$ in grounding support Schema 7. Thus, mutex between a and b is expressed with the clause:

$$\neg O\mathcal{C}' \vee \neg O\mathcal{C}''$$

b. If the above case does not hold, mutex is expressed according to Schema 5.

Auxiliary condition variables only appear in this Schema and Schema 7. Thus, there are no pre- or post-conditions associated directly with them. 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.

7. Grounding support axioms: For each step $t = 1, 2, .., h$ we have constraints that ensure whole instances of operators are executed (in parallel) rather than individual conditions. For example, consider a logistics problem with two trucks T_1 and T_2 and three locations L_1, L_2, and L_3. If $\texttt{Drive}\{\texttt{ADD}(\texttt{At}(T_1, L_2))\}$ is true, then we need constraints ensuring all conditions associated with either $\texttt{Drive}(T_1, L_1, L_2)$ or $\texttt{Drive}(T_1, L_3, L_2)$ are true. Those conditions are given in Table 1 along with their associated actions.

We develop our *grounding support* in terms of *dependency trees*. Such a tree is defined for each condition that contains a basic term of the form $\texttt{ADD}(p)$.[5] The tree is rooted at a node labelled with the corresponding condition $O\mathcal{C}$ – i.e., $\texttt{ADD}(p) \in O\mathcal{C}$. Using the notation n for a tree

$\texttt{ADD}(\texttt{At}(T_1, L_2))$ related conditions $\mathcal{C}_i$ of $\texttt{Drive}$	$\texttt{Drive}(t, l_1, l_2)$
$\mathcal{C}_0 := \texttt{ADD}(\texttt{At}(T_1, L_2))$	$(T_1, L_1, L_2), (T_1, L_3, L_2)$
$\mathcal{C}_1 := \texttt{PRE}(\texttt{Road}(L_1, L_2))$	$(T_1, L_1, L_2), (T_2, L_1, L_2)$
$\mathcal{C}_2 := \texttt{PRE}(\texttt{Road}(L_3, L_2))$	$(T_1, L_3, L_2), (T_2, L_3, L_2)$
$\mathcal{C}_3 := \{\texttt{PRE}(\texttt{At}(T_1, L_1)), \texttt{DEL}(\texttt{At}(T_1, L_1))\}$	$(T_1, L_1, L_2), (T_1, L_1, L_3)$
$\mathcal{C}_4 := \{\texttt{PRE}(\texttt{At}(T_1, L_3)), \texttt{DEL}(\texttt{At}(T_1, L_3))\}$	$(T_1, L_3, L_2), (T_1, L_3, L_1)$
$\mathcal{C}_5 := \texttt{PRE}(\texttt{In-Service}(T_1))$	$(T_1, L_1, L_2), (T_1, L_1, L_3)$ $(T_1, L_2, L_1), (T_1, L_2, L_3)$ $(T_1, L_3, L_1), (T_1, L_3, L_2)$

Table 1: Conditions and their associated actions from a logistics instance with trucks T_1 and T_2 and locations L_1, L_2, and L_3. The left column lists conditions that occur in an action with $\texttt{Drive}\{\texttt{ADD}(\texttt{At}(T_1, L_2))\}$. The right column gives instances of $\texttt{Drive}$ associated with the condition.

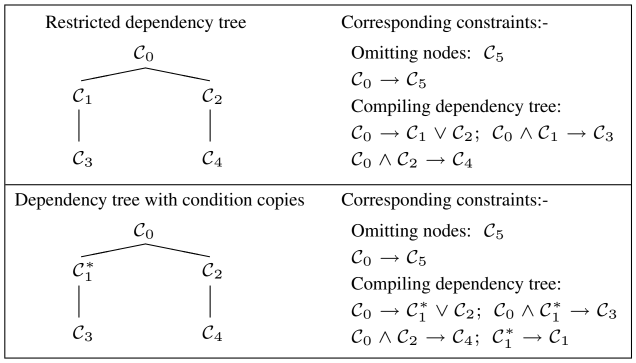
Restricted dependency tree
$\mathcal{C}_0$
$\mathcal{C}_1$
$\mathcal{C}_2$
$\mathcal{C}_3$
$\mathcal{C}_4$
Corresponding constraints:-
Omitting nodes: $\mathcal{C}_5$
$\mathcal{C}_0 \rightarrow \mathcal{C}_5$
Compiling dependency tree:
$\mathcal{C}_0 \rightarrow \mathcal{C}_1 \vee \mathcal{C}_2$; $\mathcal{C}_0 \wedge \mathcal{C}_1 \rightarrow \mathcal{C}_3$
$\mathcal{C}_0 \wedge \mathcal{C}_2 \rightarrow \mathcal{C}_4$
Dependency tree with condition copies
$\mathcal{C}_0$
$\mathcal{C}_1^*$
$\mathcal{C}_2$
$\mathcal{C}_3$
$\mathcal{C}_4$
Corresponding constraints:-
Omitting nodes: $\mathcal{C}_5$
$\mathcal{C}_0 \rightarrow \mathcal{C}_5$
Compiling dependency tree:
$\mathcal{C}_0 \rightarrow \mathcal{C}_1^* \vee \mathcal{C}_2$; $\mathcal{C}_0 \wedge \mathcal{C}_1^* \rightarrow \mathcal{C}_3$
$\mathcal{C}_0 \wedge \mathcal{C}_2 \rightarrow \mathcal{C}_4$; $\mathcal{C}_1^* \rightarrow \mathcal{C}_1$

Figure 1: For $\mathcal{C}_0$ in Table 1: (above) *Restrictive* dependency tree, and (below) the same tree with a condition copy $\mathcal{C}_1^*$. The clauses derived from each tree are listed to their right.

node, we write $\mathit{prefix}(n)$ for nodes in the root path of n, $\mathit{parent}(n)$ for the parent of n, $\mathit{children}(n)$ for children of n, and $O\mathcal{C}_n$ for the condition that labels n. A node labelled $O\mathcal{C}_n$ has a *possible* child for each ground condition that it co-occurs with in a conjunct that represents execution of whole instances of O. Where O has first-order conditions $\mathbb{C}$ (i.e., $\mathcal{C} \in \mathbb{C}$), we choose *actual* children that only instantiate, and that exhaust instantiations of, one $\mathcal{C}' \in \mathbb{C}$. We avoid loops by ensuring $\mathcal{C}'$ is not from the label of a node in $\mathit{prefix}(n) \cup n$, and choose $\mathcal{C}'$ that yields fewest children. A node n is excluded if, for some condition $O\mathcal{C}_{\rightarrow n}$ there is a dependency of the form $O\mathcal{C}_{\rightarrow n} \rightarrow O\mathcal{C}_n$ where $\mathcal{C}_{\rightarrow n}$ ranges over a superset of the argument variables in the range of $\mathcal{C}_n$. For example, such a dependency occurs in blocksworld where:

$$\texttt{Move}\{\texttt{ADD}(\texttt{On}(A, C))\} \rightarrow \\ \texttt{Move}\{\texttt{PRE}(\texttt{Clear}(C)), \texttt{DEL}(\texttt{Clear}(C))\}.$$

In excluding a node because of a dependency of the above form, the constraint $O\mathcal{C}_{\rightarrow n} \rightarrow O\mathcal{C}_n$ is included in our compilation. Making these ideas concrete, the topmost tree of Figure 1 is for condition $\texttt{Drive}\{\texttt{ADD}(\texttt{At}(T_1, L_2))\}$ from the logistics instance of Table 1.

Given a set of dependency trees for operator O, our grounding support axioms are obtained by compiling each node n into a clause of the form:

$$\bigwedge_{\substack{n_x \in \{\mathit{prefix}(n) \cup n \setminus n_0\}, \\ |\mathit{children}(\mathit{parent}(n_x))| > 1}} O\mathcal{C}_{n_x} \wedge O\mathcal{C}_{n_0} \rightarrow \bigvee_{n_y \in \mathit{children}(n)} O\mathcal{C}_{n_y}$$

[5]Here the set of conditions includes auxiliaries added for Schema 6 – i.e., if Schema 6 supposed $a \rightarrow O\mathcal{C}'$ for an auxiliary $O\mathcal{C}'$, then here $\mathcal{C}' \in \mathbb{C}_a$.

Above, we write n_0 for the root node. For example, the top-right of Figure 1 gives the constraints that are compiled from the restricted dependency tree for $\texttt{Drive}\{\texttt{ADD}(\texttt{At}(T_1, L_2))\}$. As the name of the tree suggests, these constraints are overly *restrictive*. In particular, in our logistics example 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 condition $\texttt{Drive}\{\texttt{PRE}(\texttt{At}(T_1, L_1)), \texttt{DEL}(\texttt{At}(T_1, L_1))\}$, 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 a *copied condition*. A tree node $O\mathcal{C}_n$ can be labelled as being a copied condition, written $O\mathcal{C}_n^*$, in which case we have the additional condition copy variable $O\mathcal{C}_n^*$ and the clause:

$$O\mathcal{C}_n^* \rightarrow O\mathcal{C}_n$$

In compiling the tree to CNF, we use the condition copy variable in place of the original condition. The bottom half of Figure 1 gives the example where the level 2 occurrence of $\mathcal{C}_1$ from Table 1 is copied for the tree rooted at $\mathcal{C}_0$.

Although the restrictiveness of the dependency trees developed thus far could be repaired by making every tree node a condition 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 conditions $\mathtt{prefix}(n) \cup n$ and excludes the conditions in $\mathtt{children}(n)$. Formally, in processing node n, where $\mathtt{prefix}(n) \neq \emptyset$ and $\mathtt{children}(n) \not\equiv \emptyset$, for $n_i \in \{\mathtt{prefix}(n) \cup n\}$ we compute the sets of actions $\mathcal{A}_n(n_i)$ defined as follows:

$$\begin{aligned}\mathcal{A}_n(n_i) := \{a | a \in \mathcal{A}, \mathcal{C}_{n_i} \in \mathbb{C}_a,\\ \forall n_y \in \mathtt{children}(n), (\mathcal{C}_{n_y} \notin \mathbb{C}_a \vee\\ (\exists \mathcal{C}^*_{n_y} \wedge \exists n_z \in \mathtt{prefix}(n_y), \mathcal{C}_{n_z} \notin \mathbb{C}_a))\}\end{aligned}$$

Above, $\exists O\mathcal{C}_n^*$ 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 $O\mathcal{C}_n^* \equiv \bigwedge_{n_i \in \mathtt{prefix}(n) \cup n} O\mathcal{C}_{n_i}$. Formally, in processing n we compute sets $\mathcal{A}'_n(n_i)$ for $n_i \in \mathtt{prefix}(n) \cup n$ so that, $\mathcal{A}'_n(n_i) \equiv \mathcal{A}'_n(n_i)$ if $\nexists \mathcal{C}^*_{n_i}$, and otherwise:

$$\begin{aligned}\mathcal{A}'_n(n_i) := \{a | \, a \in \mathcal{A},\\ n_x \in \{\mathtt{prefix}(n_i) \cup n_i\}, \mathcal{C}_{n_x} \in \mathbb{C}_a,\\ \forall n_y \in \mathtt{children}(n), (\mathcal{C}_{n_y} \notin \mathbb{C}_a \vee\\ (\exists \mathcal{C}^*_{n_y} \wedge \exists n_z \in \mathtt{prefix}(n_y), \mathcal{C}_{n_z} \notin \mathbb{C}_a))\}\end{aligned}$$

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.

Plan Extraction

For plan extraction we have that action a is executed at step t iff $O\mathcal{C}_a^t$ is true for all $\mathcal{C}_a^t \in \mathbb{C}_a^t$. It should be noted that auxiliary conditions from Schema 6 can be safely ignored for the purposes of plan extraction. Considering *condition copy* variables introduced in Schema 7, for plan extraction purposes we treat copies as if they were the condition from which they are derived during compilation. Finally, because the plan extraction method just described often results in many unnecessary actions being executed at each step, in practice we remove an action a at step t of the plan if that plan also includes actions $a_1, ..., a_k$ at step t, so that $i \in 1, .., k$ $a_i \neq a$ and $\mathbb{C}_a^t \subseteq \bigcup_{i \in 1,...,k} \mathbb{C}_{a_i}^t$.

Experimental Results

We developed SOLE (Step-Optimal Lifted Encoding), a SAT-based planner implemented in C++ that uses the above encoding. We now discuss an experimental comparison of SOLE with SATPLAN-06, one of the winners of the optimal track at IPC-5. For our experiments we ran SATPLAN-06 with the thin GRAPHPLAN-based encoding as that gave the best results on the benchmark domains we considered. These include IPC-5 STORAGE, and TPP; IPC-4 PIPESWORLD; IPC-3 DEPOTS, DRIVERLOG, FREECELL, ROVERS, SATELLITE, and ZENOTRAVEL; and IPC-1 LOGISTICS98, GRID, and GRIPPER. For both SOLE and SATPLAN-06 we use the complete SAT solver RSAT (Pipatsrisawat & Darwiche 2007) as the underlying satisfiability procedure. RSAT won the gold medals for the SAT+UNSAT and UNSAT problems of the 2007 International SAT Competition.[6] Although we only discuss a comparison with SATPLAN-06 here, we did perform preliminary experimentation using MAXPLAN (Chen, Xing, & Zhang 2007), finding it uncompetitive with SATPLAN-06 or indeed SOLE.[7] All experiments were run on a cluster of AMD Opteron 252 2.6GHz processors, each with 2GB of RAM. All plans computed by SOLE, SATPLAN-06, and MAXPLAN were verified by the Strathclyde Planning Group plan verifier VAL.

The results of our experiments are summarised in Table 2 and Figure 2. Table 2 compares SOLE and SATPLAN-06 when executed using the ramp-up query strategy. For each domain, there is one row for the hardest instance solved by SOLE, and one row for the hardest problem solved by SATPLAN-06. Here, we measure problem hardness in terms of the amount of time it takes the solver to yield a solution. If both solvers find the same problem hardest, then we also

[6]Since 2005, the majority of state-of-the-art DPLL SAT solvers, including RSAT, have been based on MINISAT. These MINISAT variants have dominated the crafted and industrial categories in recent SAT competitions as well as SAT-Races. In our experiments RSAT demonstrated the best performance.

[7]We believe this was due to MAXPLAN requiring a hand crafted version of the SAT solver MINISAT that is not competitive with RSAT.

		Sole						SatPlan-06					
		Time (seconds)						Time (seconds)					
Problem	h	Total	$h-1$	h	c	v	$\#a$	Total	$h-1$	h	c	v	$\#a$
DEPOTS-6	26	158.38	75.56	69.86	389268	83292	74	882.11	673.5	153.17	5311797	36527	78
DEPOTS-18	12	23.42	0.36	22.5	428712	110519	85	46.67	7.06	20.38	12355326	41981	82
DRIVERLOG-12	16	11.62	7.09	1.46	61691	21121	53	34.89	21.67	7.11	248447	14049	50
DRIVERLOG-17	13	2400.49	585.87	1777.45	277316	78401	83	Time out					
FREECELL-2	8	0.34	0.15	0.16	36539	6727	18	7.22	3.34	3.5	977707	5306	22
FREECELL-5	16	738.08	86.06	423.02	233122	25890	41	Out of memory					
GRID-1	14	0.02	-	0.02	35107	6925	14	0.08	-	0.08	173680	4317	16
GRID-2	25	12.30	2.70	6.72	530383	83481	28	76.77	19.77	15.12	17978961	54839	28
GRIPPER-4	19	26.88	11.65	0.33	11767	3099	29	85.25	58.21	0.36	34589	3088	31
GRIPPER-5	23	1998.38	452.52	1.82	18189	4445	35	Time out					
LOGISTICS98-10	13	138.9	78.71	55.08	496595	93430	138	160.32	107.4	48.49	2516447	85297	164
LOGISTICS98-23	11	20.07	0.29	19.34	222829	47339	184	27.45	0.47	26.76	506775	38994	193
PIPESWORLD-9	11	112.06	47.11	47.94	670870	143433	19	804.21	446.98	289.88	11429167	28105	22
PIPESWORLD-12	16	1424.5	844.07	359.86	408456	122958	28	Time out					
ROVERS-21	16	338.17	295.58	12.12	440662	90867	91	Time out					
ROVERS-26	15	117.81	89.84	2.26	310187	62032	120	1676.03	1157.53	8.67	16818595	57775	90
SATELLITE-12	14	56.37	31.48	1.41	140209	50940	56	163.33	86.84	1.8	1734497	53649	63
SATELLITE-13	13	54.95	19.94	2.41	180070	64790	68	79.08	35.54	10.52	2663769	67102	71
STORAGE-13	18	42.54	21.14	2.61	61016	10716	18	148.41	69.66	11.02	362345	7369	20
STORAGE-16	11	1766.34	1609.16	4.25	107223	35727	26	Time out					
TPP-21	12	3163.47	3027.91	135.56	239141	58069	154	Time out					
TPP-27	11	575.58	0.89	574.68	382160	94433	205	189.6	73.12	116.48	3963639	65235	188
ZENOTRAVEL-15	7	415.14	407.35	4.69	98097	14415	60	59.92	41.93	15.16	8956087	33259	63
ZENOTRAVEL-16	7	Time out						600.2	42.4	544.28	29994624	64078	53

Table 2: 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. Respectively, columns c and v give #clauses and #variables in the CNF at h. $\#a$ is #actions in the solution plan. For each problem, RSAT was timed out after 3600.

include a row for the penultimate hardest for each solver. Using the same experimental data as for Table 2, Figure 2 plots the cumulative number of instances solved over time by each planning system, supposing invocations of the systems on problem instances were made in parallel. For Figure 2 we only included data for instances that take one of the planners over 5 seconds to solve.

Summarising the results from Table 2 and Figure 2, except for the ZENOTRAVEL domain, where individual actions have an unusually small number of conditions, SATPLAN-06 generally requires fewer variables than SOLE to encode a problem. This disparity is due to the large number of condition copies we require to soundly represent grounding support constraints. For example, in the PIPESWORLD domain condition copy variables accounted for as much as 84% of the total number of variables and in the DEPOTS domain they accounted for as much as 68%. In addition the auxiliary condition variables in SOLE typically accounted for ~$1-3\%$ of the total number of variables. It should be noted that condition copy variables do not have a significant impact on the performance of DPLL procedures. Intuitively this is because a copy $\mathcal{C}_n^*$ represents the execution of at least one in a (typically) small set of actions whose precisely split representations include $\mathcal{C}_n$. More technically, a copy $\mathcal{C}_n^*$ implies the original $\mathcal{C}_n$ in Schema 7. Consequently, if a DPLL branches on either $\mathcal{C}_n^*$ or $\mathcal{C}_n$, it will not have to branch on the other due to the application of unit propagation. Also, even though it is not theoretically always the case (see ZENOTRAVEL for example), in practice the vast majority of assignments to copies explored by DPLL are inferred from an assignment to conditions using unit propagation, a computationally cheap procedure.

Despite requiring more variables, SOLE outperforms SATPLAN-06 in all benchmark domains tested except for TPP and ZENOTRAVEL. In these two domains, grounding support constraints (Schema 7) comprise the overwhelming majority of clauses for SOLE. We find that complicated grounding support greatly impacts the performance of RSAT when refuting plan existence given an inadequate number of steps. Examining Figure 2 we have that SOLE solves more "interesting" problems given our experimental timeout, and indeed scales better than SATPLAN-06. Moreover, SOLE does not incur any cost in terms of the number of actions in solution plans. Indeed, the number of actions in plans produced by SOLE is on a par with SATPLAN-06.

Comparing in terms of compactness of SAT representation of the bounded problems, SOLE dominates SATPLAN-06. The overwhelming majority of clauses generated by SATPLAN-06 encode action mutex. The relative compactness of SOLE 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 SOLE uses significantly fewer pre/post-condition and mutex clauses, all expressed in 2-SAT, and

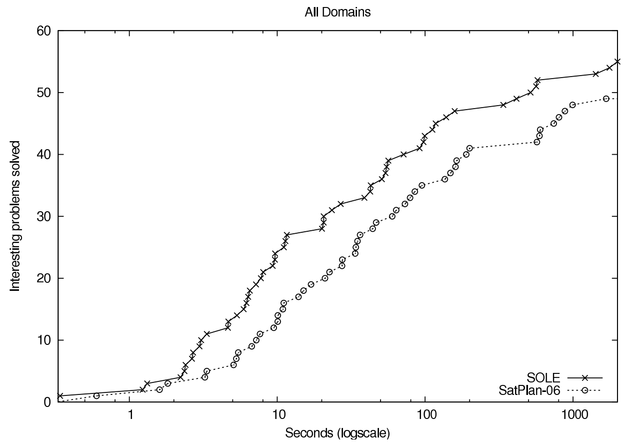

Figure 2: The number of "interesting" problems solved in parallel after a given planning time. Interesting problems are those which take at least one approach longer than 5 seconds.

usually vastly shorter successor state clauses – i.e., successor state clauses are expressed according to the small set of conditions (resp. large set of actions) that alter a fluents truth value. Quantifying this benefit, in problem DEPOTS-18, SATPLAN-06 uses ~12.1M clauses to encode action mutex, whereas SOLE uses ~293k clauses to encode both mutex (Schema 6) and grounding restrictions (Schema 7).

A deeper consequence of precise splitting results form factored conflict clauses learnt by RSAT. In particular, RSAT learns conflict between *conditions* given the SOLE encoding of the problem, whereas it has to learn conflict between *individual actions* in the SATPLAN-06 setting. Due to factoring, the decision procedure RSAT is usually more efficient given SOLE encodings because: (1) we have relatively few mutex constraints, and (2) because RSAT is required to learn and exploit far fewer conflict clauses.

Concluding Remarks

In the spirit of leveraging advances in general-purpose automated reasoning in a planning setting, we developed a compilation of step-optimal planning problems into propositional SAT formulae amenable to off-the-shelf SAT solution methods. Whereas previous approaches in this direction used *direct* representations of actions, we develop the notion of a *precisely split* representation of actions that results in more compact encodings of planning in SAT, and a more scalable and efficient SAT-based planning procedure over the state-or-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 efficiently in precise splitting essentially follow 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.

The most pressing item for future work is to examine the benefits of our compact representation for optimal planning using more advanced query strategies (Streeter & Smith 2007; Rintanen 2004), and using a ramp-down query strategy in the CRICKET setting (Ray & Ginsberg 2008). Further forward, we should explore the benefits of precisely split representations of actions for fixed horizon cost-optimal planning, and for probabilistic planning.

Acknowledgements: 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. This work was also supported by EC FP7-IST grant 215181-CogX.

References

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* (90):281–300.

Chen, Y.; Xing, Z.; and Zhang, W. 2007. Long-distance mutual exclusion for propositional planning. In Proc. *IJCAI*.

Dimopoulos, Y.; Nebel, B.; and Koehler, J. 1997. Encoding planning problems in nonmonotonic logic programs. In Proc. *ECP*.

Ernst, M.; Millstein, T.; and Weld, D. S. 1997. Automatic SAT-compilation of planning problems. In Proc. *IJCAI*.

Giunchiglia, E.; Massarotto, A.; and Sebastiani, R. 1998. Act, and the rest will follow: Exploiting determinism in planning as satisfiability. In Proc. *AAAI*.

Kautz, H., and Selman, B. 1992. Planning as satisfiability. In Proc. *ECAI*.

Kautz, H., and Selman, B. 1999. Unifying SAT-based and graph-based planning. In Proc. *IJCAI*.

Kautz, H. A.; Selman, B.; and Hoffmann, J. 2006. SatPlan: Planning as satisfiability. In Abstracts *IPC-05*.

Kautz, H. A. 2006. Deconstructing planning as satisfiability. In Proc. *AAAI*.

Nabeshima, H.; Soh, T.; Inoue, K.; and Iwanuma, K. 2006. Lemma reusing for SAT based planning and scheduling. In Proc. *ICAPS*.

Pipatsrisawat, K., and Darwiche, A. 2007. Rsat 2.0: SAT solver description. Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA.

Ray, K., and Ginsberg, M. L. 2008. The complexity of optimal planning and a more efficient method for finding solutions. In Proc. *ICAPS*.

Rintanen, J.; Heljanko, K.; and Niemelä, I. 2006. Planning as satisfiability: parallel plans and algorithms for plan search. *Artificial Intelligence* 170(12-13):1031–1080.

Rintanen, J. 2004. Evaluation strategies for planning as satisfiability. In Proc. *ECAI*.

Rintanen, J. 2008. Planning graphs and propositional clause learning. In Proc. *KR*.

Robinson, N.; Gretton, C.; Pham, D.-N.; and Sattar, A. 2008. A compact and efficient SAT encoding for planning. In Proc. *ICAPS*.

Streeter, M., and Smith, S. 2007. Using decision procedures efficiently for optimization. In Proc. *ICAPS*.

Wehrle, M., and Rintanen, J. 2007. Planning as satisfiability with relaxed e-step plans. In Proc. *Australia AI*.