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 = hbig, hbig−1, ..., 1 query strategies. For ramp-down, also called backwards level reduction, hbig 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, 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 the 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 & Rin-...
ments in $\mathbb{C}$ to be composites of basic conditions as follows. Let $\alpha(\mathcal{T}')$ be any permutation of argument list $\mathcal{T}'$. If there is any pair $\tau_1(\pi_1(\mathcal{T}'))$ and $\tau_2(\pi_2(\mathcal{T}'))$ of basic conditions of an operator $(a, \mathcal{C})$, then there is exactly one $\mathcal{C} \in \mathbb{C}$ so that $\tau_1(\pi_1(\mathcal{T}')) \land \tau_2(\pi_2(\mathcal{T}')) \in \mathcal{C}$. For example, the previously mentioned Move operator can be described in terms of $\mathcal{C}$ as follows:

$$\langle \text{Move}(x_1, x_2, x_3), \langle \{\text{PRE}(\text{On}(x_1, x_2)), \text{DEL}(\text{On}(x_1, x_2))\}, \{\text{PRE}(\text{Clear}(x_1)), \text{DEL}(\text{Clear}(x_1))\}, \{\text{PRE}(\text{Clear}(x_1)), \{\text{ADD}(\text{On}(x_1, x_2)), \{\text{ADD}(\text{Clear}(x_2))\})\rangle \rangle$$

Planning actions are obtained by grounding operator descriptions over object symbols. Grounding according to as previously mentioned Move operator can be described in terms of $\mathcal{C}$ as follows:

$$\text{Action: } a \coloneqq \text{Move}(A, B, C)$$

$$\text{pre}(a) \coloneqq \{\text{On}(A, B), \text{Clear}(A), \text{Clear}(C)\};$$

$$\text{add}(a) \coloneqq \{\text{Clear}(B), \text{On}(A, C)\};$$

$$\text{del}(a) \coloneqq \{\text{On}(A, B), \text{Clear}(C)\};$$

We write $a$ for the action having a set of ground preconditions $\text{pre}(a)$, positive post-conditions $\text{add}(a)$, and negative post-conditions $\text{del}(a)$. As for the above ground Move example, these lists contain elements from $P$. Ground conditions associated with $a$ are notated $\mathcal{C}_a$, 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 $\text{pre}(a) \subseteq s$. 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 state $s$ the resultant state is $(s \cup \text{add}(a)) \setminus \text{del}(a)$. Actions cannot both add and delete the same proposition — i.e., $\text{add}(a) \cap \text{del}(a) \equiv \emptyset$.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 $G \subseteq P$, 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 $\text{Move}[k](x)$ for the predicate that says “the $k$’th argument of action $\text{Move}$ is $x$”, we have that action $\text{Move}(A, B, C)$ is represented by the conjunct $\text{Move}[1](A) \land \text{Move}[2](B) \land \text{Move}[3](C)$.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 $i$th significant bit of $n$ we represent $\text{Move}(A, B, C)$ with conjunct $p_1 \land \neg p_1 \land p_2$. 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 $\Sigma$ 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 $\text{Move}(A, B, C) \rightarrow \text{Clear}(A)$ is written $\text{Move}[1](A) \rightarrow \text{Clear}(A)$. 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 $\text{Move}(A, B, C)$ and $\text{Move}(D, E, F)$ interferes because it is represented by a conjunct of $\text{Move}[1](A) \land \text{Move}[2](B) \land \text{Move}[3](C) \land \text{Move}[1](D) \land \text{Move}[2](E) \land \text{Move}[3](F)$. This implies 6 additional instances of $\text{Move}$ corresponding to: $\text{Move}(A, B, F)$, $\text{Move}(A, E, C)$, $\text{Move}(A, E, F)$, $\text{Move}(D, B, C)$, $\text{Move}(D, B, F)$, and $\text{Move}(D, E, C)$. In summary, interference prevents the simple 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 $\text{OC}_a$ 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:

\[ \bigwedge_{C_n \in C_w} OC_n \]

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

\[
\text{Move}\{\text{ADD}(\text{Clear}(B))\} \land \text{Move}\{\text{PRE}(\text{On}(A, B)), \text{DEL}(\text{On}(A, B))\} \land \\
\text{Move}\{\text{PRE}(\text{Clear}(C)), \text{DEL}(\text{Clear}(C))\} \land \\
\text{Move}\{\text{PRE}(\text{Clear}(A))\} \land \text{Move}\{\text{ADD}(\text{On}(A, C))\}
\]

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 \( A \) so that \( a \in A \), we say \( a \) is redundant with respect to parallel execution of actions \( A \) if at any state any execution of \( A \) is indistinguishable from any execution of \( A \backslash a \). 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 \( \psi_1 \) and \( \psi_2 \) are the unique conjuncts representing two actions \( a_1 \) and \( a_2 \). Conjoin \( \psi_1 \land \psi_2 \) 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 \( \phi \). Our encoding is constructive (and sound) in the sense that any satisfying model of \( \phi \) corresponds more-or-less directly with a plan. We also have completeness because \( \phi \) 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 \( \phi \). 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 \( A^t \) for the set of ground actions occurring at step \( t \), \( C_w^t \) for the set of all ground conditions associated with action \( a^t \in A^t \), and \( C^t \) for the union of ground conditions occurring at step \( t \) – i.e.,

\[ C^t \equiv \bigcup_{a^t \in A^t} C_w^t. \]

For our compilation we have a condition variable \( OC^t \) for each \( C^t \in C^t \). To indicate that \( C^t \in C^t \) for variable \( OC^t \), we use the notation \( OC^t \).

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 \( p \in G \) we have a unit clause containing \( p^\text{h} \).

2. **Precondition and postcondition axioms:** This Schema encodes action effects using the corresponding condition variables. For each variable \( OC^t \) we have the following constraint:

\[
\bigwedge_{\text{PRE}(p) \in \text{C}^t} \text{OC}^t \rightarrow p^t \land \\
\bigwedge_{\text{ADD}(p) \in \text{C}^t} \text{OC}^t \rightarrow p^{t+1} \land \\
\bigwedge_{\text{DEL}(p) \in \text{C}^t} \text{OC}^t \rightarrow \neg p^{t+1}
\]

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 \( \text{make}(p^t) \) be the set of conditions in \( C^{t-1} \) containing an element of the form \( \text{ADD}(p) \). Then for each \( p^t, t > 0 \) we have:

\[
p^t \rightarrow (p^{t-1} \lor \bigvee_{C^{t-1} \in \text{make}(p^t)} \text{OC}^{t-1})
\]

This says that if \( p \) is true at step \( t \), then 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 \( \text{On}(A, B) \) we have a clause:

\[
\text{On}(A, B)^t \rightarrow (\text{On}(A, B)^{t-1} \lor \text{Move}\{\text{ADD}(\text{On}(A, B)^t)\})
\]

4. **Propositional mutex axioms:** For every pair of state propositions \( p_1^t \) and \( p_2^t \) that occur mutex in layer \( t \) of the plangraph, we have a clause:

\[
\neg p_1^t \lor \neg p_2^t
\]

5. **Inter-operator mutex axioms:** Schemata 2 and 4 prohibit parallel instantiation of \( a, b \in A \) provided either

   a. Schema 2 gives constraints of the form \( O_1 C_a^t \rightarrow p_1^{t+1} \) and \( O_2 C_b^t \rightarrow \neg p_1^{t+1} \); or

   b. Schema 2 gives constraints of the form \( O_1 C_a^t \rightarrow p_1^t \) and \( O_2 C_b^t \rightarrow p_2^t \), and Schema 4 gives \( \neg p_1^t \lor \neg p_2^t \).

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 \( a, b \in A \) so that in Schema 2 \( O_1 C_a^t \rightarrow p_1^t \) and \( O_2 C_b^t \rightarrow \neg p_1^{t+1} \) occur. Provided \( O_1 \neq O_2 \), we have the clause:

\[
\neg O_1 C_a^t \lor \neg O_2 C_b^t
\]

\(^3\)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 $O_1C_a^{i}$ can be
\[
\text{Load}\{\text{PRE}{}(\text{At}(\text{truck}_1, \text{loc}_1))\}^i \quad \text{and} \quad O_2C_b^{i} \text{ can be}
\text{Drive}\{\text{PRE}{}(\text{At}(\text{truck}_1, \text{loc}_1)), \text{DEL}{}(\text{At}(\text{truck}_1, \text{loc}_1))\}^i.
\]

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 instan-
tiations 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 con-
titions $C_a$ and $C_b$ according to the following rules:

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

Direct compilation of mutex between $C_a$ and $C_b$ given the
above situations is not admissible. Thus, we represent
mutex between $a$ and $b$ using auxiliary condition vari-
able that are generated as needed. We choose two such
variables $OC'$ and $OC''$, and later enforce $a \rightarrow OC'$
and $b \rightarrow OC''$ in grounding support Schema 7. Thus, mutex
between $a$ and $b$ is expressed with the clause:

$$-OC' \lor -OC''$$

b. If the above case does not hold, mutex is expressed ac-
cording 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 implemen-
tation 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 illegal 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 op-
erators 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 $\text{Drive}\{\text{ADD}{}(\text{At}(T_1, L_2))\}$ is true, then we need
constraints ensuring all conditions associated with either
$\text{Drive}(T_1, L_1, L_2)$ or $\text{Drive}(T_1, L_3, L_2)$ are true. Those
conditions are given in Table 1 along with their associated
actions.

We develop our grounding support in terms of depen-
dency trees. Such a tree is defined for each condition that
contains a basic term of the form $\text{ADD}{}(p)$. The tree is
rooted at a node labelled with the corresponding condition
$OC$—i.e., $\text{ADD}{}(p) \in OC$. Using the notation $n$ for a tree

Table 1: Conditions and their associated actions from a logi-
cistics 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 $\text{Drive}\{\text{ADD}{}(\text{At}(T_1, L_2))\}$. The right column
gives instances of $\text{Drive}$ associated with the condition.

| Restricted dependency tree | Corresponding constraints:
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_0$</td>
<td>Omitting nodes: $C_5$</td>
</tr>
<tr>
<td>$C_1$</td>
<td>$C_0$ $\rightarrow$ $C_5$</td>
</tr>
<tr>
<td>$C_2$</td>
<td>Compiling dependency tree:</td>
</tr>
<tr>
<td>$C_3$</td>
<td>$C_0$ $\rightarrow$ $C_7$</td>
</tr>
<tr>
<td>$C_4$</td>
<td>$C_0$ $\rightarrow$ $C_7$</td>
</tr>
</tbody>
</table>

| Dependency tree with condition copies | Corresponding constraints:
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$C'_1$</td>
<td>Omitting nodes: $C_5$</td>
</tr>
<tr>
<td>$C_0$</td>
<td>$C_0$ $\rightarrow$ $C_5$</td>
</tr>
<tr>
<td>$C_1$</td>
<td>Compiling dependency tree:</td>
</tr>
<tr>
<td>$C_2$</td>
<td>$C_0$ $\rightarrow$ $C_7$</td>
</tr>
<tr>
<td>$C_3$</td>
<td>$C_0$ $\rightarrow$ $C_7$</td>
</tr>
</tbody>
</table>

Figure 1: For $C_0$ in Table 1: (above) Restrictive dependency
tree, and (below) the same tree with a condition copy $C'_1$.

The clauses derived from each tree are listed to their right.

node, we write $\text{prefix}(n)$ for nodes in the root path of $n$, $\text{parent}(n)$ for the parent of $n$, $\text{children}(n)$ for children of $n$, and $OC_n$ for the condition that labels $n$. A node labelled $OC_n$ 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 $C$ (i.e., $C \in C$), we choose actual children that only instantiate, and that exhaust instantiations of, one $C' \in C$. We avoid loops by ensuring $C'$ is not from the label of a node in $\text{prefix}(n) \cup n$, and choose $C'$ that yields fewest children. A node $n$ is excluded if, for some condition $OC_{\ldots n}$ there is a dependency of the form $OC_{\ldots n} \rightarrow OC_n$ where $C_{\ldots n}$ ranges over a superset of the argument variables in the range of $C_n$. For example, such a dependency occurs in blocks-
world where:

$$\text{Move}\{\text{ADD}{}(\text{On}(A, C))\} \rightarrow$$
$$\text{Move}\{\text{PRE}{}(\text{Clear}(C)), \text{DEL}{}(\text{Clear}(C))\}.$$
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 \( \text{Drive(ADD(At(T_1, L_2)))} \). As the name of the tree suggests, these constraints are overly restrictive. In particular, in our logistics example we must be able to execute \( \text{Drive}(T_1, L_3, L_2) \) and \( \text{Drive}(T_2, L_1, L_2) \) in parallel without the condition \( \text{Drive(\text{PRE}(At(T_1, L_1)), \text{DEL}(At(T_1, L_1)))} \), and hence action \( \text{Drive}(T_1, L_1, L_2) \), being true. In order to relax the constraints developed so far, we introduce the notion of a copied condition. A tree node \( OC_n \) can be labelled as being a copied condition, written \( OC^*_n \), in which case we have the additional condition copy variable \( OC^*_a \) and the clause:

\[
OC^*_n \rightarrow OC_n
\]

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 \( C_1 \) from Table 1 is copied for the tree rooted at \( C_0 \).

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 \( \text{prefix}(n) \cup n \) and excludes the conditions in \( \text{children}(n) \). Formally, in processing node \( n \), where \( \text{prefix}(n) \neq \emptyset \) and \( \text{children}(n) \neq \emptyset \), for \( n_i \in \{ \text{prefix}(n) \cup n \} \) we compute the sets of actions \( A_n(n_i) \) defined as follows:

\[
A_n(n_i) := \{ a | a \in A, \forall n_y \in \text{children}(n_i), (C_{n_y} \notin C_n) \lor (\exists C^*_a \land \exists n_z \in \text{prefix}(n_y), C_{n_z} \notin C_a) \}
\]

Above, \( \exists OC^*_n \) is true iff \( n \) is labelled as a copy. If for all \( i \) \( A_n(n_i) \) 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 conjunction of itself with the conditions in its root path – i.e., we suppose \( OC^*_n \equiv \bigwedge_{n_i \in \text{prefix}(n) \cup n} OC_{n_i} \). Formally, in processing \( n \) we compute sets \( A'_n(n_i) \) for \( n_i \in \text{prefix}(n) \cup n \) so that, \( A'_n(n_i) \equiv A_n(n_i) \) if \( \exists OC^*_n \), and otherwise:

\[
A'_n(n_i) := \{ a | a \in A, \forall n_y \in \text{children}(n_i), (C_{n_y} \notin C_n) \lor (\exists C^*_a \land \exists n_z \in \text{prefix}(n_y), C_{n_z} \notin C_a) \}
\]

If \( A'_n(n_i) \) 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 \( OC^*_a \) is true for all \( C^*_a \in C_a \). 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 \( a_1, \ldots, a_t \) at step \( t \), so that \( i \in 1, \ldots, k, a_i \neq a \) and \( C^*_a \subseteq \bigcup_{i \in 1, \ldots, k} C^*_a \).

**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. 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. 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
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 seconds.

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 $\sim 3\%$ 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 $C^*_n$ represents the execution of at least one in a (typically) small set of actions whose precisely split representations include $C_n$. More technically, a copy $C^*_n$ implies the original $C_n$ in Schema 7. Consequently, if a DPLL branches on either $C^*_n$ or $C_n$, 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
factoring, the decision procedure RS results in 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 ~29k 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**


