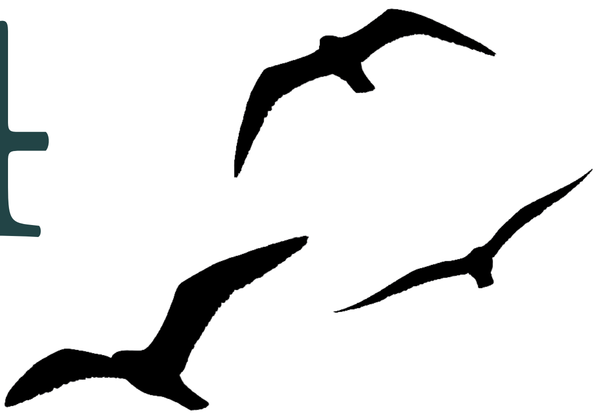
ICAPS 2014



Proceedings of the 5th Workshop on

Knowledge Engineering for Planning and Scheduling

Edited By:
Roman Barták, Simone Fratini, Lee McCluskey and Tiago Vaquero

Portsmouth, New Hampshire, USA - June 22, 2014

Organizing Committee

Roman Barták

Charles University, Czech Republic

Simone Fratini

European Space Agency, Germany

Lee McCluskey

University of Huddersfield, UK

Tiago Vaquero

University of Toronto, Canada

Program Committee

Roman Barták, Charles University, Czech Republic
Daniel Borrajo, Universidad Carlos III de Madrid, Spain
Adi Botea, IBM, Ireland
Amedeo Cesta, ISTC-CNR, Italy
Susana Fernández, Universidad Carlos III de Madrid, Spain
Simone Fratini, European Space Agency, Germany
Antonio Garrido, Universidad Politecnica de Valencia, Spain
Arturo González-Ferrer, Universidad Carlos III de Madrid, Spain
Felix Ingrand, LAAS-CNRS, France
Lee McCluskey, University of Huddersfield, UK
Ugur Kuter, SIFT, USA
Julie Porteous, Teesside University, UK
Kanna Rajan, MBARI, USA
José Reinaldo Silva, University of São Paulo, Brazil
Tiago Vaquero, University of Toronto, Canada
Dimitris Vrakas, Aristotle University of Thessaloniki, Greece
Gerhard Wickler, University of Edinburgh, Scotland

Core Security
European Coordinating Committee for Artificial Intelligence (ECCAI)
IBM Research
German Society for CS

Artificial Intelligence Journal
National ICT Australia Limited
National Science Foundation (NSF)

David E. Smith
University of New Hampshire
Adventium Labs
Smart Information Flow Technologies
TRACLabs

Foreword

The KEPS 2014 workshop aims to promote research in the areas lying between planning & scheduling technology on the one side, and practical applications and problems on the other. Despite recent advances in the area, the performance of planning & scheduling systems is still dependent to a large extent on how problems and domains are formulated, resulting in the need for careful system fine-tuning. In particular, recent work with competition benchmark problems highlights the importance of the relation between how domain knowledge is engineered within a model, and the efficiency of planning engines when input with these models.

Knowledge engineering for planning & scheduling covers a wide area, including the acquisition, formalization, design, validation and maintenance of domain models, and the selection and optimization of appropriate planning engines to work on them. Knowledge engineering processes impact directly on the success of real planning and scheduling applications. The importance of knowledge engineering techniques is clearly demonstrated by a performance gap between domain-independent planners and planners exploiting domain dependent knowledge.

This year the set of accepted papers reproduced in these proceedings reflects the wide scope of knowledge engineering within the ICAPS area. There are papers on techniques in engineering real applications, such as in clinical rehabilitation, and in hypothesis generation in medical and network domains. Two papers consider plans - one to develop the means for estimating upper bounds of plan length from planning problems, and the other in optimizing solution plans by removing redundant actions within them. Problem transformation is an area which is growing in importance, and in these proceedings we have two papers on this issue, one looking at ways of decomposing very complex planning problems to expedite solutions, and another to compile problems with soft constraints into a form usable by mainstream planning engines. Finally, we have contributions on knowledge capture - one a review of automated domain model acquisition, and the other describing a new knowledge-based language designed to enable subject-matter experts to formulate knowledge in planning applications.

Roman Barták, Simone Fratini, Lee McCluskey, Tiago Vaquero
KEPS 2014 Organizers
June 2014

Table of Contents

Mechanising Theoretical Upper Bounds in Planning 1
Mohammad Abdulaziz, Charles Gretton, Michael Norrish

Applying Problem Decomposition to Extremely Large Planning Domains 8
Masataro Asai, Alex Fukunaga

Eliminating All Redundant Actions from Plans Using SAT and MaxSAT 16
Tomáš Balyo, Lukáš Chrpa

Planning with Preferences by Compiling Soft Always Goals into STRIPS with Action Costs 23
Luca Ceriani, Alfonso Emilio Gerevini

Automated Knowledge Engineering Tools in Planning: State-of-the-art and Future Challenges 31
Rabia Jilani, Andrew Crampton, Diane Kitchin, Mauro Vallati

Goal-directed Generation of Exercise Sets for Upper-Limb Rehabilitation............. 38
José C. Pulido, José C. González, Arturo González-Ferrer, Javier García, Fernando Fernández, Antonio Bandera, Pablo Bustos, Cristina Suárez

Knowledge Engineering for Planning-Based Hypothesis Generation...................... 46
Shirin Sohrabi, Octavian Udrea, Anton V. Riabov

Creating Planning Domain Models in KEWI 54
Gerhard Wickler, Lukáš Chrpa , Thomas Leo McCluskey

Mechanising Theoretical Upper Bounds in Planning

Mohammad Abdulaziz and **Charles Gretton** and **Michael Norrish** [*]
Canberra Research Lab., NICTA
7 London Circuit, Canberra ACT 2601, Australia
{Mohammad.Abdulaziz, Charles.Gretton, Michael.Norrish}@nicta.com.au

Abstract

We examine the problem of computing upper bounds on the lengths of plans. Tractable approaches to calculating such bounds are based on a decomposition of state-variable dependency graphs (causal graphs). Our contribution follows an existing formalisation of concepts in that setting, reporting our efforts to mechanise bounding inequalities in HOL. Our primary contribution is to identify and repair an important error in the original formalisation of bounds. We also develop novel bounding results and compare them analytically with existing bounds.

Introduction

This paper develops novel insights and approaches for reasoning about upper bounds on the lengths of plans. Formally, an *upper bound* of N means that, if a plan exists, then an optimal plan comprises no more than N steps. A variety of applications for such upper bounds have been explored. If an explicit state-based search encounters a state with *upper bound* N and *lower bound* M—if a plan exists, it must be at least of length M—then if $N < M$ the state can safely be pruned. Also, given a tight upper bound N, the plan existence problem can be reduced to a fixed-horizon reachability problem—*i.e.*, is there a plan of length less-than-or-equal-to N? In that case the fixed-horizon problem can be posed as a Boolean SAT(isfiability) problem (Kautz and Selman 1996). More generally, planning problems can be solved using SAT solvers given a query strategy that focuses search effort at important horizon lengths (Rintanen 2004; Streeter and Smith 2007). Tight horizon bounds provide focus in that setting. Lastly, in situations where no plan exists, bounds have been used to identify a small subset of goal facts that cannot be achieved together. Here, plan existence for goal sets which admit short bounds are tested earliest, so that non-existence can be established quickly using relatively little search effort.

Our contributions follow (Rintanen and Gretton 2013), which describes a general procedure for computing upper bounds based on state-variable dependency information. That approach yields useful bounds in problems that exhibit a branching one-way dependency structure. A highlighted example of that type of dependency occurs in the *logistics* benchmark. To change the location of a package, vehicles must be used. The locations of vehicles can be modified irrespective of the package locations, and indeed independently of each other. In other words, each package has a one-way dependency with vehicles and otherwise all objects can be manipulated independently.

This work reports on our efforts so far to obtain mechanized proofs of the correctness of versions of the headline theorems from (Rintanen and Gretton 2013). Our work has exposed a subtle yet important error in the original formalisation of bounds. We correct the original formalisation of upper bounds and summarise the new proof of the bounds from that work. The new proof employs a constructive technique, relying on a function which builds a plan of an appropriate length given an overly long input. Our new proof of correctness is mechanised in the HOL interactive theorem proving system (Slind and Norrish 2008). We provide links to that mechanisation work which is hosted on `github`. Finally, we also develop novel inequalities which yield tighter bounds than existing approaches.

Definitions and Notations

We formalise the planning problem and give definitions of concepts related to computing bounds on solution lengths. Our definitions are functionally equivalent to standard expositions of deterministic propositional planning. We include two minor departures, used to decrease the verbosity of our proofs. In particular, our formalisation supposes there is a single goal-state, rather than a set of goal states. Also, we allow any action to be executed at any state, supposing its effects are only realised if its preconditions are satisfied at the state from which it is executed.

Definition 1 (States and Actions)**.** *A planning problem is defined in terms of* states *and* actions*:*

1. *We model states as finite maps from variables—i.e., state characterizing propositions—to Booleans.*
2. *An action π is a pair of finite maps. Each of those maps is from a subset of the problem variables to the Booleans. The domain of each of these maps does not have to be the same. The first component of the pair ($p(\pi)$) is the*

[*]NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

precondition*: each variable in the domain of the map must have the specified value if the action is to affect any state change.*[1] *The second component of the pair ($e(\pi)$) is the* effect*: each variable in the domain of the map takes on the specified value in the resulting state. We say that a variable is an* action precondition *if it is in the domain of the precondition map, and similarly that it is an* action effect *if it is in the domain of the effect map.*

3. *An action sequence $\dot{\pi}$ is a list of actions. We use the notation $\pi :: \dot{\pi}$ (a "cons") to denote the sequence which has π as its first element, followed by the actions in $\dot{\pi}$, and $[\,]$ to denote the empty sequence.*

We will write concrete examples of states and actions as sets of variables that are either bare (mapping to true), or overlined (mapping to false). For example, $\{x, \overline{y}, z\}$ is the state where state variables x and z are true, and y is false.

We now use the above concepts to define a planning *problem*.

Definition 2 (Planning Problems)**.** *A problem Π is a 3-tuple $\Pi = \langle I, A, G\rangle$, with I the initial state of the problem, G the goal state and A a set of permitted actions. When writing about the components of a problem Π, unless explicitly written otherwise we will write I, A and G for $\Pi.I$ etc., leaving the Π implicit. We will write D for the domain of the initial state; this is the domain of the problem.*

Problem Π is valid *if the goal has the same domain as the initial state, and all actions refer exclusively to variables that occur in that set. We only consider valid problems.*

Naturally, a state s is valid with respect to a planning problem Π if its domain is the same as that of the initial state I.

Definition 3 (Action Execution)**.** *When an action π is executed at state s, it causes a transition to a successor state. If the precondition is not satisfied at s, the successor is simply s once more. Otherwise, the action effects hold at the successor. We denote this operation $\mathsf{exec}(s, \pi)$. We lift that definition to sequences of executions taking an action sequence $\dot{\pi}$ as the second argument. So $\mathsf{exec}(s, \dot{\pi})$ denotes the state resulting from successively applying each of $\dot{\pi}$'s actions, starting with s.*

Key concepts in formalising bounds in planning are those of *projection* and *dependency graph*.

Definition 4 (Projection)**.** *Projecting an object (a state s, an action π, a sequence of actions $\dot{\pi}$ or a problem Π) on a variable set vs refers to restricting the domain of the object or its constituents to vs. We denote these operations as $s\downharpoonright_{vs}$, $\pi\downharpoonright_{vs}$, $\dot{\pi}\downharpoonright_{vs}$ and $\Pi\downharpoonright_{vs}$ for a state, action, action sequence and problem respectively. Note that if an action sequence $\dot{\pi}$ is projected in this way, it may come to include actions with empty preconditions and/or effects, however, actions with empty effects are removed.*

Definition 5 (Problem Dependency Graphs)**.** *The dependency graph of a problem Π is a directed graph, written G, describing variable dependencies. This graph was conceived under different guises in (Williams and Nayak 1997) and (Bylander 1994), and is also commonly referred to as a* causal graph*. That graph features one vertex for each variable in D. An edge from v_1 to v_2 records that v_2 is dependent on v_1. A variable v_2 is dependent on v_1 in a planning problem Π iff one of the following statements holds:*

1. *v_1 is the same as v_2.*
2. *There is an action π in A such that v_1 is a precondition of π and v_2 is an effect of π.*
3. *There is an action π in A such that both v_1 and v_2 are effects of π.*

We write $v_1 \rightarrow v_2$ if there is a directed arc from v_1 to v_2 in the dependency graph for Π. Also, when we illustrate a dependency graph we do not draw arcs from a variable to itself although it is dependent on itself.

We also lift the concept of dependency graphs and refer to *lifted* dependency graphs (written G_{vs}) in which each vertex represents a distinct set of problem variables. The vertices (variable sets) in lifted graphs will partition the domain D of the original problem.

Definition 6 (Variable Set Dependencies)**.** *An edge from variable set vs_1 to set vs_2 records that vs_2 is dependent on vs_1. A variable set vs_2 is dependent on vs_1 in a planning problem Π (written $vs_1 \rightarrow vs_2$) iff all of the following conditions hold:*

1. *vs_1 is disjoint from vs_2.*
2. *There exist variables $v_1 \in vs_1$ and $v_2 \in vs_2$ such that $v_1 \rightarrow v_2$.*

In revising previous work, we refer to the strongly connected components (SCCs) of a dependency graph.

Definition 7 (Strongly Connected Component)**.** *An SCC of G is a maximal subgraph in which there is a directed path from each vertex to every other vertex. We write G_S for the lifted graph which has one vertex for each SCC in G, and an edge from component (a variable set) vs_1 to component vs_2 iff $vs_1 \rightarrow vs_2$. Note that G_S will be a DAG.*

Definition 8 (Leaves, Ancestors and Children)**.** *For any DAG G, the set of leaves $\mathcal{L}(G)$ contains those vertices of G from which there are no outgoing edges. We also write $\mathcal{A}_G(n)$ to denote the set of n's ancestors in G. Alternatively, $\mathcal{A}_G(n) = \{n_0 \mid n_0 \in G \wedge n_0 \rightarrow^+ n\}$, where $\rightarrow^+$ is the transitive closure of $\rightarrow$.We also write $\mathcal{C}_G(n)$ to denote the set $\{n_0 \mid n_0 \in G \wedge n \rightarrow n_0\}$ which are the children of n in G.*

Finally, an important relation on lists (of actions) on which our work relies is the *scattered sublist*[2] relation:

Definition 9 (Scattered Sublists)**.** *List l_1 is a scattered sublist of l_2 (written $l_1 \preceq\cdot\, l_2$) if all the members of l_1 occur in the same order in l_2.*

[1]We use the word *domain* in the mathematical sense—*i.e.*, the set from which the function arguments are drawn. To avoid confusion, we shall not use this term to refer to a PDDL model.

$$
\begin{aligned}
[\,] \underset{vs}{\wr} \dot{\pi} &= \dot{\pi}\downharpoonright_{D-vs} \\
\pi_c :: \dot{\pi}_c \underset{vs}{\wr} \pi :: \dot{\pi} &= \begin{cases} \pi :: (\dot{\pi}_c \underset{vs}{\wr} \dot{\pi}) & \text{if } \pi\downharpoonright_{vs} = \pi_c \\ \pi_c :: \dot{\pi}_c \underset{vs}{\wr} \dot{\pi} & \text{o/wise} \end{cases}
\end{aligned}
$$

Figure 1: The definition of the stitching function ($\wr$).

$$\begin{tikzcd}
 & w \arrow[dl] \arrow[d] \arrow[dr] & \\
x \arrow[r, leftrightarrow] & y \arrow[r, leftrightarrow] & z
\end{tikzcd}$$

Figure 2: The dependency graph of the problem in Example 2.

Derivation of Upper Bounds

The main contribution of our work is the constructive proof technique we use to derive plan-length bounds. Our approach is the outcome of an exercise in mechanising the proofs of the results from (Rintanen and Gretton 2013) using the HOL interactive theorem proving system (Slind and Norrish 2008)[3]. Working from first principles, that exercise uncovered a subtle yet important error in the original formalisation of bounds.[4] In detail, a plan-length bound was originally formulated as the maximum of the minimum length executions between pairs of states. An important error becomes apparent in the usage of that bound, which follows the algorithm of iterative plan refinement described originally in (Knoblock 1994). Summarising the error, earlier work incorrectly assumed that an abstract plan satisfying that bound could be refined into a complete plan for the problem at hand. However, valid problems exist where no minimum length executions in the abstract model can be refined to create a valid plan. The key assumption is invalid and thus a correction is needed. In what follows we describe the error we discovered in (Rintanen and Gretton 2013), correct that error, and then describe our proofs of important bounding theorems.

Error

The headline result in (Rintanen and Gretton 2013) is described using an upper bound function that was written using the ℓ symbol. Because we are treating both the erroneous and repaired versions of the function in our work, we have chosen to write $\ell_\perp$ for the original erroneous version. We use the subscript $\perp$ symbol to emphasise that it leads to an invalid result. Intuitively, $\ell_\perp$ denotes a function which takes a planning problem Π and returns the length of the longest optimal execution between any two valid states in Π. We now formally review the definition of $\ell_\perp$, identifying and explaining two errors. Following that, we shall repair that definition in support of the upper bounds in (Rintanen and Gretton 2013) . We write $\Pi(s)$ for the set of finite lists of actions in Π that reach s from I, $\mathcal{S}$ for the set of states in Π, and $|\dot{\pi}|$ for the length of execution $\dot{\pi}$.

Definition 10.

$$\ell_\perp(\Pi) = \max_{s \in \mathcal{S}} \min_{\dot{\pi} \in \Pi(s)} |\dot{\pi}|$$

A first negative consequence of the above definition renders $\ell_\perp$ unsuitable for making statements about upper bounds. Specifically, $\ell_\perp$ is not well-defined in situations where there are no valid executions between I and a state s—*i.e.*, the function min has no well-defined output in that situation. This issue was not dealt with adequately in (Rintanen and Gretton 2013) . We illustrate this error with an example.

Example 1. *Consider the planning problem* Π *such that*

$$\Pi = < I = \{x, \overline{y}, \overline{z}\}, A = \{(\{x, \overline{y}\}, \{z\})\}, G = \{x, \overline{y}, z\} >$$

Now let $s = \{\overline{x}, \overline{y}, \overline{z}\}$ *and* $s' = \{x, \overline{y}, z\}$*. These are two valid states in* Π *but a transition from* s *to* s' *is impossible. So* $\ell_\perp$ *is thus ill-defined for* Π.

We can mitigate this ill-definedness by treating reachability explicitly, as follows. We shall see in a moment however that the proposed correction is still insufficient. Let $\dot{A}$ be the set of finite lists of actions in Π and $\Pi(\dot{\pi}, s) = \{\dot{\pi}' | \mathsf{exec}(s, \dot{\pi}) = \mathsf{exec}(s, \dot{\pi}') \wedge \dot{\pi}' \in \dot{A}\}$, *i.e.*, the set of executions from s equivalent to $\dot{\pi}$.

Definition 11.

$$\ell'_\perp(\Pi) = \max_{s \in \mathcal{S}, \dot{\pi} \in \dot{A}} \min_{\dot{\pi}' \in \Pi(\dot{\pi}, s)} |\dot{\pi}|$$

This proposed modification to $\ell'_\perp$ is insufficient to fix a deeper error. A headline results from (Rintanen and Gretton 2013) states:

Not-A-Theorem 1. *If the domain of* Π *is comprised of two disjoint variable sets* vs_1 *and* vs_2 *satisfying* $vs_2 \not\rightarrow vs_1$*, we have:*

$$\ell'_\perp(\Pi) < (\ell'_\perp(\Pi\downharpoonright_{vs_1}) + 1)(\ell'_\perp(\Pi\downharpoonright_{vs_2}) + 1)$$

This inequality is invalid with respect to Definition 11 (and Definition 10), as follows.

Example 2. *Consider the problem*

$$\Pi = \left\langle \begin{array}{l} I = \{\overline{w}, \overline{x}, \overline{y}, \overline{z}\} \\ A = \left\{ \begin{array}{l} a = (\emptyset, \{x\}), b = (\{x\}, \{\overline{x}, y\}), \\ c = (\{x, y\}, \{\overline{x}, \overline{y}, z\}), d = (\{w\}, \{x, y, z\}) \end{array} \right\} \\ G = \{\overline{w}, x, y, z\} \end{array} \right\rangle$$

[2]Our "scattered sublist" is sometimes referred to as a "subsequence" in the computer science literature. In HOL we require a distinct concept.

[3]The mechanised formalisation and proofs described in this paper can be found online at `https://github.com/mabdula/planning/`.

[4]We note that the algorithm and experimental results from (Rintanen and Gretton 2013) are presumed correct: that part of the work uses cardinality derived bounds rather than the bounding concept presented in the original formalisation.

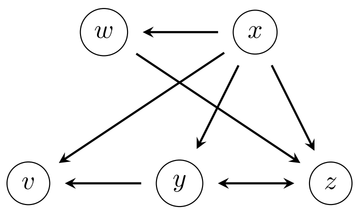


Figure 3: Dependency graph of the problem in Example 3.

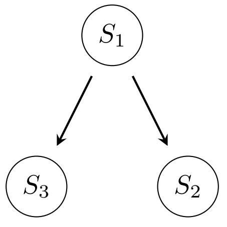


Figure 4: An SCC graph with 3 SCCs.

whose dependency graph is shown in Figure 2. Note that here we include the starting and goal conditions, however $\ell'_\perp$ is independent of those. The domain is comprised of two sets of variables $S = \{x, y, z\}$, which is an SCC, and the set $\mathcal{A}_{G_S}(S) = \{\{w\}\}$. The evaluation of $\ell'_\perp(\Pi)$ gives 7, as this is the maximal length optimal execution. Specifically, that maximal optimal execution is $[a; b; a; c; a; b; a]$ from $\{\overline{w}, \overline{x}, \overline{y}, \overline{z}\}$ to $\{\overline{w}, x, y, z\}$. Treating the abstract problems $\Pi\downharpoonright_S$ and $\Pi\downharpoonright_{\bigcup \mathcal{A}_{G_S}(S)}$, in each case we get a bound of 1. This violates Not-A-Theorem 1.

Mechanisation of Existing Bounds

In this section we provide a corrected definition of ℓ. Adopting that new definition we describe our mechanised proof of the inequality that featured in Not-A-Theorem 1. We also develop novel bounds and compare them to the inequalities suggested in (Rintanen and Gretton 2013) , showing that in some cases our novel bounds dominate.

Our definition of ℓ mitigates the problem exhibited in the definition by (Rintanen and Gretton 2013) by appealing to $\Pi^{\preceq\cdot}(\dot\pi, s) = \{\dot\pi' | \mathsf{exec}(s, \dot\pi) = \mathsf{exec}(s, \dot\pi') \wedge \dot\pi' \preceq\cdot\, \dot\pi\}$ – *i.e.*, the set of executions from s that are equivalent to $\dot\pi$ and also scattered sublists of $\dot\pi$.

Definition 12.

$$\ell(\Pi) = \max_{s \in \mathcal{S}, \dot\pi \in \dot A} \; \min_{\dot\pi' \in \Pi^{\preceq\cdot}(\dot\pi, s)} |\dot\pi|$$

It should be clear that $\ell(\Pi)$ is a valid upper bound for Π. Using ℓ we can first prove a corrected version of Not-A-Theorem 1.

Theorem 1. *If the domain of Π is comprised of two disjoint variable sets vs_1 and vs_2 satisfying $vs_2 \not\rightarrow vs_1$, we have:*

$$\ell(\Pi) < (\ell(\Pi\downharpoonright_{vs_1}) + 1)(\ell(\Pi\downharpoonright_{vs_2}) + 1)$$

Proof. To prove Theorem 1 we use a construction which, given any plan $\dot\pi$ for Π violating the stated bound, produces a shorter witness plan $\dot\pi'$ satisfying that bound. The premise $vs_2 \not\rightarrow vs_1$ implies that actions with variables from vs_2 in their effects—hereupon we shall call these vs_2-actions—never include vs_1 variables in their effects. Also, because vs_1 and vs_2 capture all problem variables, the effects of vs_1-actions after projection to the set vs_1 are unchanged. Our construction first takes the action sequence $\dot\pi\downharpoonright_{vs_2}$. Definition 12 of ℓ provides a scattered sublist $\dot\pi'_{vs_2} \preceq\cdot\, \dot\pi\downharpoonright_{vs_2}$ satisfying $|\dot\pi'_{vs_2}| \le \ell(\Pi\downharpoonright_{vs_2})$. Moreover, the definition of ℓ can guarantee that $\dot\pi'_{vs_2}$ is equivalent, in terms of the execution outcome, to $\dot\pi\downharpoonright_{vs_2}$. The stitching function described in Figure 1 is then used to remove the vs_2-actions in $\dot\pi$ whose projections on vs_2 are not in $\dot\pi'_{vs_2}$. Thus our construction arrives at a plan $\dot\pi'' = \dot\pi'_{vs_2} \underset{vs_2}{\wr} \dot\pi$ with at most $\ell(\Pi\downharpoonright_{vs_2})$ vs_2-actions. We are left to address the continuous lists of vs_1-actions in $\dot\pi''$, to ensure that in the constructed plan any such list satisfies the bound $\ell(\Pi\downharpoonright_{vs_1})$. The method by which we obtain $\dot\pi''$ guarantees that there are at most $\ell(\Pi\downharpoonright_{vs_2}) + 1$ such lists to address. The definition of ℓ provides that for any abstract list of actions $\dot\pi\downharpoonright_{vs_1}$ in $\Pi\downharpoonright_{vs_1}$, there is a list that achieves the same outcome of length at most $\ell(\Pi\downharpoonright_{vs_1})$. Our construction is completed by replacing each continuous sequence of vs_1-actions in $\dot\pi''$ with witnesses of appropriate length ($\ell(\Pi\downharpoonright_{vs_1})$). □

The above construction can be illustrated using the following concrete example.

Example 3. *Consider the valid problem*

$$\Pi = \left\langle \begin{array}{l} I = \{v, \overline{w}, \overline{x}, \overline{y}, \overline{z}\} \\ A = \left\{ \begin{array}{l} a = (\emptyset, \{x\}), b = (\{x\}, \{y\}), \\ c = (\{x\}, \{\overline{v}\}), d = (\{x\}, \{w\}), \\ e = (\{y\}, \{v\}), f = (\{w, y\}, \{z\}), \\ g = (\{\overline{x}\}, \{y, z\}) \end{array} \right\} \\ G = \{v, w, x, y, z\} \end{array} \right\rangle$$

whose dependency graph is shown in Figure 3. The domain of Π has a subset $vs_2 = \{v, y, z\}$ where vs_2 is dependent on the set $vs_1 = \{w, x\}$, and vs_1 is not dependent on vs_2.

In Π, the actions b, c, e, f, g are vs_2-actions, and a, d are vs_1-actions. A plan $\dot\pi$ for Π is $[a; a; b; c; d; d; e; f]$. When the plan $\dot\pi$ is projected on vs_2 it becomes $[b\downharpoonright_{vs_2}; c\downharpoonright_{vs_2}; e\downharpoonright_{vs_2}; f\downharpoonright_{vs_2}]$, which is a plan for $\Pi\downharpoonright_{vs_2}$. A shorter plan, $\dot\pi_c$, for $\Pi\downharpoonright_{vs_2}$ is $[b\downharpoonright_{vs_2}; f\downharpoonright_{vs_2}]$. Since $\dot\pi_c$ is a scattered sublist of $as\downharpoonright_{vs_2}$, we can use the stitching function to obtain a shorter plan for Π. In this case, $\dot\pi_c \underset{vs_2}{\wr} \dot\pi$ is $[a; a; b; d; d; f]$. The second step is to contract the pure vs_1 segments which are $[a; a]$ and $[d; d]$, which are contracted to $[a]$ and $[d]$ respectively. The final constructed witness for our bound is the plan $[a; b; d; f]$.

So far we have seen how to reason about problem bounds by treating abstract subproblems separately. We now review how subexponential bounds for planning problems are achieved by exploiting branching one-way state-variable dependencies. An example of that type of dependency structure is exhibited in Figure 4, where S_i are sets of variables each of which forms an SCC in the dependency graph, and we have both $S_1 \rightarrow S_2$ and $S_1 \rightarrow S_3$. Recall, the latter means that there is at least one edge from a variable in S_1

$$\begin{tikzcd}
S_1 \arrow[d] \arrow[dr] & S_2 \arrow[d] \\
S_3 & S_4
\end{tikzcd}$$

Figure 5: An SCC graph with 4 SCCs.

to one in S_2, and similarly between S_1 and S_3. Importantly, Figure 4 gives $S_2 \not\rightarrow S_1$ and $S_3 \not\rightarrow S_1$, and there is no dependency between any variables in S_2 and S_3. For bounding optimal plan lengths, the following theorem was suggested in (Rintanen and Gretton 2013) to exploit such structures.

Theorem 2. *Following Definition 7, G_S is the DAG of SCCs from the dependency graph for a problem Π. The upper bound $\ell(\Pi)$ satisfies the following inequality:*

$$\ell(\Pi) \leq \Sigma_{S\in\mathcal{L}(G_S)}\ell(\Pi\downharpoonleft_{S\cup(\bigcup \mathcal{A}_{G_S}(S))}) \tag{1}$$

In our work we have established sometimes superior bounds by deviating from the above inequality. The theorem that we mechanised can provide tighter bounds for some specific dependency structures; and otherwise it does not dominate and is not dominated by the bound provided by the inequality in Theorem 2. Our theorem exploits planning problems which are partitioned by two sets of state variables that are not connected in the dependency graph.

Theorem 3. *For a problem Π whose domain is partitioned by vs_1 and vs_2 such that $vs_1 \not\rightarrow vs_2$ and $vs_2 \not\rightarrow vs_1$*

$$\ell(\Pi) \leq \ell(\Pi\downharpoonleft_{vs_1}) + \ell(\Pi\downharpoonleft_{vs_2}) \tag{2}$$

Proof. The premises $vs_1 \not\rightarrow vs_2$ and $vs_2 \not\rightarrow vs_1$ implies that vs_2-actions have no variables from vs_1 in their effects or preconditions and vs_1-actions have no vs_2 variables in their effects or preconditions. This implies that removing vs_1-actions from a plan does not affect the executability of vs_2-actions in that plan. This statement applies in the other direction also, in the case of vs_2-action removal. Our construction first takes the action sequence $\dot{\pi}\downharpoonleft_{vs_2}$. Definition 12 of ℓ provides an action sequence $\dot{\pi}'_{vs_2}$ scattered sublist $\dot{\pi}'_{vs_2} \preceq \dot{\pi}\downharpoonleft_{vs_2}$ satisfying $|\dot{\pi}'_{vs_2}| \leq \ell(\Pi\downharpoonleft_{vs_2})$. Moreover, the definition of ℓ guarantees that $\dot{\pi}'_{vs_2}$ is equivalent, in terms of the execution outcome, to $\dot{\pi}\downharpoonleft_{vs_2}$. The stitching function described in Figure 1 is then used to remove the vs_2-actions in $\dot{\pi}$ whose projection on vs_2 is not in $\dot{\pi}'_{vs_2}$. Thus our construction arrives at a plan $\dot{\pi}'' = \dot{\pi}'_{vs_2} \underset{vs_2}{\wr} \dot{\pi}$ whose execution outcome is the same as $\dot{\pi}$ but in which the number of vs_2-action is at most $\ell(\Pi\downharpoonleft_{vs_2})$. The next step is to take $\dot{\pi}''\downharpoonleft_{vs_1}$. Then we obtain $\dot{\pi}'_{vs_1}$, a shorter equivalent plan to $\dot{\pi}''\downharpoonleft_{vs_1}$, which has at most $\ell(\Pi\downharpoonleft_{vs_1})$ actions. Then stitching again we obtain the plan $\dot{\pi}'_{vs_1} \underset{vs_1}{\wr} \dot{\pi}''$ which is a plan that has the same outcome as $\dot{\pi}''$ and accordingly $\dot{\pi}$ but with at most $\ell(\Pi\downharpoonleft_{vs_1}) + \ell(\Pi\downharpoonleft_{vs_2})$ actions. □

Comparison

In this section we compare different ways to decompose dependency graphs based on the results we have so far, and study the upper bounds thus obtained.

Case 1 Following Theorem 2, a bound on the problem in Figure 4 is given by :

$$\ell(\Pi) \leq \ell(\Pi\downharpoonleft_{S_2\cup S_1}) + \ell(\Pi\downharpoonleft_{S_3\cup S_1}) \tag{3}$$

Because $S_2 \not\rightarrow S_1$ and $S_3 \not\rightarrow S_1$ hold, application of the result from Theorem 1 is applicable:

$$\begin{aligned}\ell(\Pi) \;\leq\;& \ell(\Pi\downharpoonleft_{S_2})\ell(\Pi\downharpoonleft_{S_1}) + \ell(\Pi\downharpoonleft_{S_2}) + \ell(\Pi\downharpoonleft_{S_3})\ell(\Pi\downharpoonleft_{S_1}) \\ &+\ell(\Pi\downharpoonleft_{S_3}) + 2\ell(\Pi\downharpoonleft_{S_1})\end{aligned} \tag{4}$$

Alternatively, since $S_2 \cup S_3 \not\rightarrow S_1$ holds, Theorem 1 can be used, as follows:

$$\ell(\Pi) \leq \ell(\Pi\downharpoonleft_{S_2\cup S_3})\ell(\Pi\downharpoonleft_{S_1}) + \ell(\Pi\downharpoonleft_{S_2\cup S_3}) + \ell(\Pi\downharpoonleft_{S_1}) \tag{5}$$

Because $S_2 \not\rightarrow S_3$ and $S_3 \not\rightarrow S_2$ hold, thus application of Theorem 3 is admissible, yielding:

$$\begin{aligned}\ell(\Pi) \;\leq\;& \ell(\Pi\downharpoonleft_{S_2})\ell(\Pi\downharpoonleft_{S_1}) + \ell(\Pi\downharpoonleft_{S_3})\ell(\Pi\downharpoonleft_{S_1}) + \ell(\Pi\downharpoonleft_{S_2}) \\ &+\ell(\Pi\downharpoonleft_{S_3}) + \ell(\Pi\downharpoonleft_{S_1})\end{aligned} \tag{6}$$

The bound reached by the second approach is based on our new results, and is the tightest.

Case 2 Decomposing the problem using our novel bounds does not always lead to a better result. Consider the dependency graph in Figure 5. A bound derived with Theorem 2 on a problem with such a dependency graph will be:

$$\ell(\Pi) \leq \ell(\Pi\downharpoonleft_{S_3\cup S_1}) + \ell(\Pi\downharpoonleft_{S_4\cup(S_1\cup S_2)}) \tag{7}$$

Again, this bound can be decomposed further according to Theorem 1:

$$\begin{aligned}\ell(\Pi) \;\leq\;& \ell(\Pi\downharpoonleft_{S_3})\ell(\Pi\downharpoonleft_{S_1}) + \ell(\Pi\downharpoonleft_{S_3}) + \ell(\Pi\downharpoonleft_{S_1}) \\ &+\ell(\Pi\downharpoonleft_{S_4})(\ell(\Pi\downharpoonleft_{S_1\cup S_2})) + \ell(\Pi\downharpoonleft_{S_4}) \\ &+\ell(\Pi\downharpoonleft_{S_1\cup S_2})\end{aligned} \tag{8}$$

Application of Theorem 3 gives:

$$\begin{aligned}\ell(\Pi) \;\leq\;& \ell(\Pi\downharpoonleft_{S_3})\ell(\Pi\downharpoonleft_{S_1}) + \ell(\Pi\downharpoonleft_{S_4})(\ell(\Pi\downharpoonleft_{S_1})) \\ &+\ell(\Pi\downharpoonleft_{S_4})(\ell(\Pi\downharpoonleft_{S_2})) + 2\ell(\Pi\downharpoonleft_{S_1}) + \ell(\Pi\downharpoonleft_{S_2}) \\ &+\ell(\Pi\downharpoonleft_{S_3}) + \ell(\Pi\downharpoonleft_{S_4})\end{aligned} \tag{9}$$

Alternatively, decomposing the same dependency graph using Theorem 1 and Theorem 3 will lead to a different bound. Because in the dependency graph in Figure 5 $(S_3 \cup S_4) \not\rightarrow (S_1 \cup S_2)$ holds, Theorem 1 is applicable as follows:

$$\ell(\Pi) \leq \ell(\Pi\downharpoonleft_{S_3\cup S_4})\ell(\Pi\downharpoonleft_{S_1\cup S_2}) + \ell(\Pi\downharpoonleft_{S_3\cup S_4}) + \ell(\Pi\downharpoonleft_{S_1\cup S_2}) \tag{10}$$

This bound can be decomposed further using Theorem 3:

$$\begin{aligned}\ell(\Pi) \quad \leq \quad & \ell(\Pi\downharpoonright_{S_3})\ell(\Pi\downharpoonright_{S_1}) + \ell(\Pi\downharpoonright_{S_3})\ell(\Pi\downharpoonright_{S_2}) \\ & +\ell(\Pi\downharpoonright_{S_4})\ell(\Pi\downharpoonright_{S_1}) + \ell(\Pi\downharpoonright_{S_4})\ell(\Pi\downharpoonright_{S_1}) \\ & +\ell(\Pi\downharpoonright_{S_1}) + \ell(\Pi\downharpoonright_{S_2}) + \ell(\Pi\downharpoonright_{S_3}) + (\ell(\Pi\downharpoonright_{S_4}))\end{aligned} \tag{11}$$

Neither of the bounds in Inequality 9 and Inequality 11 dominate. Specifically, the first bound has an extra $\ell(\Pi\downharpoonright_{S_1})$ term while the second one has an extra $\ell(\Pi\downharpoonright_{S_2})\ell(\Pi\downharpoonright_{S_3})$ term.

A Tighter Bound

In this section we present a conjecture and an informal proof of it. To prove our conjecture we require the following lemma:

Lemma 1. *For a planning problem Π for which $\dot{\pi}$ is a solution and for a node S (i.e. an SCC) in G_S of Π, there exists a plan $\dot{\pi}'$ such that:*

- $n(S, \dot{\pi}') \leq \ell(\Pi\downharpoonright_S)(\Sigma_{C \in \mathcal{C}_{G_S}(S)} n(C, \dot{\pi}) + 1)$*, where* $n(vs, \dot{\pi})$ *is the number of* vs*-actions in* $\dot{\pi}$*, and*
- $\dot{\pi}' \preceq\cdot\, \dot{\pi}$*, and*
- $\forall\ S' \neq S.\ n(S', \dot{\pi}) = n(S', \dot{\pi}')$*.*

Proof. The proof of Lemma 1 is a constructive proof. Let $\dot{\pi}_{\overline{\mathcal{C}}}$ be a contiguous fragment of $\dot{\pi}$ that has no $\bigcup \mathcal{C}_{G_S}(S)$-actions in it. Then perform the following steps:

- By the definition of ℓ, there must be a plan $\dot{\pi}_S$ that achieves the same execution result as $\dot{\pi}_{\overline{\mathcal{C}}}\downharpoonright_S$, and satisfies $|\dot{\pi}_S| \leq \ell(\Pi\downharpoonright_S)$ and $\dot{\pi}_S \preceq\cdot\, \dot{\pi}_{\overline{\mathcal{C}}}\downharpoonright_S$.
- Because $D - S - \bigcup \mathcal{C}_{G_S}(S) \not\rightarrow S$ holds and using the same argument used in the proof of Theorem 1, $\dot{\pi}'_{\overline{\mathcal{C}}}(= \dot{\pi}_S \underset{S}{\between} \dot{\pi}_{\overline{\mathcal{C}}}\downharpoonright_{D - \bigcup \mathcal{C}_{G_S}(S)})$ achieves the same $D - \bigcup \mathcal{C}_{G_S}(S)$ assignment as $\dot{\pi}_{\overline{\mathcal{C}}}$, and at the same time it is a sublist of $\dot{\pi}_{\overline{\mathcal{C}}}$. Also, $n(S, \dot{\pi}'_{\overline{\mathcal{C}}}) \leq \ell(\Pi\downharpoonright_S)$ holds.
- Finally, because $\dot{\pi}_{\overline{\mathcal{C}}}$ has no $\bigcup \mathcal{C}_{G_S}(S)$-actions, no $\bigcup \mathcal{C}_{G_S}(S)$ variables change along the execution of $\dot{\pi}_{\overline{\mathcal{C}}}$ and accordingly any $\bigcup \mathcal{C}_{G_S}(S)$ variables in preconditions of actions in $\dot{\pi}_{\overline{\mathcal{C}}}$ always have the same assignment. This means that $\dot{\pi}'_{\overline{\mathcal{C}}} \underset{D - \bigcup \mathcal{C}_{G_S}(S)}{\between} \dot{\pi}_{\overline{\mathcal{C}}}$ will achieve the same result as $\dot{\pi}_{\overline{\mathcal{C}}}$, but with at most $\ell(\Pi\downharpoonright_S)$ S-actions.

Repeating the previous steps to each $\dot{\pi}_{\overline{\mathcal{C}}}$ fragment in $\dot{\pi}$ yields an action sequence $\dot{\pi}'$ that has at most $\ell(\Pi\downharpoonright_S)(n(\bigcup \mathcal{C}_{G_S}(S), \dot{\pi}) + 1)$ S-actions. Because $\dot{\pi}'$ is the result of consecutive applications of the stitching function, it is a scattered sublist of $\dot{\pi}$. Lastly, because during the previous steps, only S-actions were removed as necessary the number of any other S'-actions in $\dot{\pi}'$ is the same as their number in $\dot{\pi}$. □

Corrolary 1. *Let $F(S, \dot{\pi})$ be a plan that results from Lemma 1. We know then that:*

- $\mathsf{exec}(s, \dot{\pi}) = \mathsf{exec}(s, F(S, \dot{\pi}))$*, and*
- $n(S, F(S, \dot{\pi})) \leq \ell(\Pi\downharpoonright_S)(\Sigma_{C \in \mathcal{C}_{G_S}(S)} n(C, \dot{\pi}) + 1)$*, and*
- $F(S, \dot{\pi}) \preceq\cdot\, \dot{\pi}$*, and*
- $\forall\ S' \neq S.\ n(S', \dot{\pi}) = n(S', F(S, \dot{\pi}))$*.*

We now use Corollary 1 to prove the following theorem:

Theorem 4. *For a planning problem Π, the bound on the solution length for such a problem is*

$$\ell(\Pi) \leq \Sigma_{S \in G_S} N(S) \tag{12}$$

where $N(S) = \ell(\Pi\downharpoonright_S)(\Sigma_{C \in \mathcal{C}_{G_S}(S)} N(C) + 1)$*.*

Proof. Again, our proof of this theorem follows a constructive approach where we begin by assuming we have a solution $\dot{\pi}$. The goal of the proof is to find a witness plan $\dot{\pi}'$ such that $\forall\ S \in G_S.\ n(S, \dot{\pi}') \leq N(S)$. We proceed by induction on l_S,the list of nodes in G_S, assuming that it is topologically sorted. As our graph is not empty, the base case is a singleton list $[S]$. In this case the goal reduces to finding a plan $\dot{\pi}_0$ such that $n(S, \dot{\pi}_0) \leq N(S)$ and $\dot{\pi}_0 \preceq\cdot\, \dot{\pi}$. Since S has no children (as it is the only node), $N(S) = \ell(\Pi\downharpoonright_S)$ and accordingly the proof follows from the definition of ℓ.

In the step case, we assume the result holds for any problem whose non-empty node list is the topologically sorted l_S. We then show that it also holds for Π, a problem whose node list is $S :: l_S$, where S has no parents (hence its position at the start of the sorted list), and l_S is non-empty. Since the node list of $\Pi\downharpoonright_{D-S}$ is l_S, the induction hypothesis applies. Accordingly, there is a solution $\dot{\pi}_{D-S}$ for $\Pi\downharpoonright_{D-S}$ such that $\dot{\pi}_{D-S} \preceq\cdot\, \dot{\pi}\downharpoonright_{D-S}$ and $\forall\ K \in l_S.\ n(K, \dot{\pi}') \leq N(K)$. Since $S :: l_S$ is topologically sorted, $D - S \not\rightarrow S$ holds. Therefore $\dot{\pi}'_{D-S} = \dot{\pi}_{D-S} \underset{D-S}{\between} \dot{\pi}$ is a solution for Π (using the same argument used in the proof of Theorem 1). Furthermore, $\forall K \in l_S.\ n(K, \dot{\pi}'_{D-S}) \leq N(K)$ and $\dot{\pi}'_{D-S} \preceq\cdot\, \dot{\pi}$. The last step in this proof is to apply F to $(S, \dot{\pi}'_{D-S})$ to get the required witness. From Corollary 1 and because the operators $=$ and $\preceq\cdot$ are transitive, we know that

- $\mathsf{exec}(s, \dot{\pi}) = \mathsf{exec}(s, F(S, \dot{\pi}'_{D-S}))$, and
- $n(S, F(S, \dot{\pi}'_{D-S})) \leq \ell(\Pi\downharpoonright_S)(\Sigma_{C \in \mathcal{C}_{G_S}(S)} n(C, \dot{\pi}'_{D-S}) + 1)$, and
- $F(S, \dot{\pi}'_{D-S}) \preceq\cdot\, \dot{\pi}$, and
- $\forall\ K \neq S.\ n(K, \dot{\pi}'_{D-S}) = n(K, F(S, \dot{\pi}'_{D-S}))$.

Since $\forall\ K \in l_S.\ n(K, \dot{\pi}'_{D-S}) \leq N(K)$ holds, then $n(S, F(S, \dot{\pi}'_{D-S})) \leq \ell(\Pi\downharpoonright_S)(\Sigma_{C \in \mathcal{C}_{G_S}(S)} N(C)$ is true. Accordingly the plan demonstrating the needed bound is $F(S, \dot{\pi}'_{D-S})$.

□

Bounds Obtained

Using Theorem 4, we can compute bounds that are tighter than the ones obtained using Theorem 2 for the two dependency graphs in Figure 4 and Figure 5. These bounds can be obtained by computing $N(S)$ for every SCC in the graph in the reverse topological order of the SCCs and summing the results.

Case 1 For the dependency graph in Figure 4 we start by computing $N(S_2)$ and $N(S_3)$ which are going to be $\ell(\Pi\downharpoonright_{S_2})$ and $\ell(\Pi\downharpoonright_{S_3})$ because they both have no children. Then we compute $N(S_1)$ which will be $\ell(\Pi\downharpoonright_{S_1})\ell(\Pi\downharpoonright_{S_2}) + \ell(\Pi\downharpoonright_{S_1})\ell(\Pi\downharpoonright_{S_3}) + \ell(\Pi\downharpoonright_{S_1})$. Accordingly and based on Theorem 4 the bound on the problem is

$$\begin{aligned}\ell(\Pi) \quad \leq \quad & \ell(\Pi\downharpoonright_{S_1})\ell(\Pi\downharpoonright_{S_2}) + \ell(\Pi\downharpoonright_{S_1})\ell(\Pi\downharpoonright_{S_3}) + \ell(\Pi\downharpoonright_{S_1}) \\ & + \ell(\Pi\downharpoonright_{S_2}) + \ell(\Pi\downharpoonright_{S_3})\end{aligned} \tag{13}$$

This bound is tighter than the one obtained using Theorem 2 shown in Inequality 4.

Case 2 For the dependency graph in Figure 4 we start by computing $N(S_3)$ and $N(S_4)$ which equal $\ell(\Pi\downharpoonright_{S_3})$ and $\ell(\Pi\downharpoonright_{S_4})$, respectively, because they both have no children. Then we compute $N(S_1)$ which will be $\ell(\Pi\downharpoonright_{S_1})\ell(\Pi\downharpoonright_{S_2}) + \ell(\Pi\downharpoonright_{S_1})$. Finally we compute $N(S_2)$ which will be $\ell(\Pi\downharpoonright_{S_2})\ell(\Pi\downharpoonright_{S_3}) + \ell(\Pi\downharpoonright_{S_2})\ell(\Pi\downharpoonright_{S_4}) + \ell(\Pi\downharpoonright_{S_2})$. Accordingly the bound on the problem is

$$\begin{aligned}\ell(\Pi) \quad \leq \quad & \ell(\Pi\downharpoonright_{S_1})\ell(\Pi\downharpoonright_{S_3}) + \ell(\Pi\downharpoonright_{S_2})\ell(\Pi\downharpoonright_{S_3}) \\ & + \ell(\Pi\downharpoonright_{S_2})\ell(\Pi\downharpoonright_{S_4}) + \ell(\Pi\downharpoonright_{S_1}) + \ell(\Pi\downharpoonright_{S_2}) \\ & + \ell(\Pi\downharpoonright_{S_3}) + \ell(\Pi\downharpoonright_{S_4})\end{aligned} \tag{14}$$

This bound is tighter than the one obtained using Theorem 2 shown in Inequality 9.

Conclusion and Future Work

With this work, we believe we have launched a fruitful interdisciplinary collaboration between the fields of AI planning and mechanised mathematical verification. From the interactive theorem-proving community's point of view, it is gratifying to be able to find and fix errors in the modern research literature. Specifically, we found errors in the existing formalisation of bounds (errors that led to the statement of a false theorem), corrected the errors with a revised definition of the key notion, and then gave a mechanized proof of a key result relating to the calculation of upper bounds.

For planning systems to be deployed in safety critical applications and for autonomous exploration of space, they must not only be efficient, and conservative in their resource consumption, but also correct. We have therefore found it highly gratifying to be able to give the planning community strong assurance of the correctness of the formalisation we treated.

We have only scratched the surface so far. When a tight bound for a planning problem is known, the most effective technique for finding a plan is to reduce that problem to SAT (Rintanen 2012). Proposed reductions are constructive, in the sense that a plan can be constructed in linear time from a satisfying assignment to a formula. A key recent advance in the setting of planning-via-SAT has been the development of compact SAT-representations of planning problems. Such representations facilitate highly efficient plan search (Rintanen 2012; Robinson et al. 2009). In future work, we would like to verify the correctness of both the reductions to SAT, and the algorithms that subsequently construct plans from a satisfying assignment.

References

Bylander, T. 1994. The computational complexity of propositional strips planning. *Artif. Intell.* 69(1-2):165–204.

Kautz, H. A., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proc. 13th National Conf. on Artificial Intelligence*, 1194–1201. AAAI Press.

Knoblock, C. A. 1994. Automatically generating abstractions for planning. *Artif. Intell.* 68(2):243–302.

Rintanen, J., and Gretton, C. O. 2013. Computing upper bounds on lengths of transition sequences. In *IJCAI*.

Rintanen, J. 2004. Evaluation strategies for planning as satisfiability. In *Proc. 16th European Conf. on Artificial Intelligence*, 682–687. IOS Press.

Rintanen, J. 2012. Planning as satisfiability: Heuristics. *Artif. Intell.* 193:45–86.

Robinson, N.; Gretton, C.; Pham, D. N.; and Sattar, A. 2009. SAT-based parallel planning using a split representation of actions. In *ICAPS*.

Slind, K., and Norrish, M. 2008. A brief overview of HOL4. In *Theorem Proving in Higher Order Logics*, volume 5170 of *LNCS*, 28–32. Springer.

Streeter, M. J., and Smith, S. F. 2007. Using decision procedures efficiently for optimization. In *Proc. 17th Intnl. Conference on Automated Planning and Scheduling*, 312–319. AAAI Press.

Williams, B. C., and Nayak, P. P. 1997. A reactive planner for a model-based executive. In *Proc. 15th Intnl. Joint Conference on Artificial Intelligence*, 1178–1185. Morgan Kaufmann Publishers.

Applying Problem Decomposition to Extremely Large Planning Domains

Masataro Asai and **Alex Fukunaga**
Department of General Systems Studies
Graduate School of Arts and Sciences
The University of Tokyo
guicho2.71828@gmail.com, fukunaga@idea.c.u-tokyo.ac.jp

Abstract

Despite the great improvement in the existing planning technology, their ability is still limited if we try to solve extremely large problems, e.g. whose state variables are over 1000 times more than those in standard IPC benchmark problems. We propose a method that tries to reduce the problem size by decomposing a problem into a set of cyclic planning problems. We categorized the objects in a problem into groups, using Abstract Type and Abstract Task. We then solve each subproblem per group as a cyclic planning problem, which can be solved efficiently with Steady State abstraction.

1 Introduction

Recent improvements in classical planning allowed planners to solve larger and larger domains by modeling planning as satisfiability and as heuristic search. However, even this is not enough for large-scale problems such as manufacturing domains which requires hundreds or thousands of objects to be processed at once. Since STRIPS planning is PSPACE-complete [Bylander, 1994], it is impractical to directly solve such large problems with existing, search-based approaches. Empirical study for this is proposed in [Helmert, Röger, and others, 2008], addressing the same limitation in simple A^* search with "almost perfect" heuristic functions.

A recent study [Ochi et al., 2013] showed that although standard domain-independent planners were capable of generating plans for assembling a single instance of a complex product, generating plans for assembling multiple instances of a product was quite challenging. For example, generating plans to assemble 4-6 instances of a relatively simple product in a 2-arms cell assembly system pushed the limits of State-of-the-Art domain-independent planners. However, real-world CELL-ASSEMBLY applications require mass production of hundreds/thousands of instances of a product.

As another example, consider the standard IPC benchmark Elevator domain, where the task is to efficiently transport people between floors using several elevators. In the IPC benchmark instances, the number of passengers is around 7 times the number of elevators in the satisficing track, and in the optimization track, the number of passengers is comparable to the number of elevators. However, (as we demonstrate in Sec. 4) a crowded elevator scenario where the number of passengers is more than 40 times the number of elevators is beyond the capabilities of state-of-the-art planners.

These are examples of domains where the problem instances consist of sets of tasks that are basically independent and decomposable, particularly if there is no optimality requirement. In the cell assembly domain, constructing a single product is relatively easy. In the elevator domain, transporting a single passenger from its initial location to its destination is easy. While standard, search-based planners struggle to find solutions to large-scale instances of these domains, a human can easily come up with satisficing plans for these problems by decomposing the problem.

In this paper, we propose an approach to decomposing large-scale problems by identifying groups of easy subproblems. In a CELL-ASSEMBLY domain where we are given a large number of parts that must be assembled into a set of products, our system extracts a set of abstract subproblems where each subproblem corresponds to the assembly of a single type of product. Similarly, in a large-scale elevator domain, our system extracts a set of abstract subproblems where each subproblem corresponds to loading N (generic) people into an elevator on a particular floor F_I and moving them to their particular goal floor F_G. In domains such as cell assembly and elevator where there are no resource constraints, the original, large-scale problems can be solved by sequencing solutions to the subproblems.

Our approach extends the previous work which identifies "component abstractions" for the purpose of finding macro-operators [Botea, Müller, and Schaeffer, 2004]. We identify abstract tasks that consist of an abstract component, plus the initial and goal propositions that are relevant to that component. We show experimentally that this approach is capable of decomposing large-scale, CELL-ASSEMBLY problems into a batch of orders that can be solved by a cyclic planner such as the recently proposed system by [Asai and Fukunaga, 2014]. We also show that our system can be used to decompose large-scale versions of some IPC benchmark domains (Elevator, Woodworking, Rover, Barman) into more tractable subproblems.

The rest of the paper is organized as follows. First, we explain our overall approach to decomposition-based solution of large-scale problems (Sec. 2). We then explain the limitation of existing cyclic planning model further. Next we describe the notion of abstract type and *Abstract Component* originally came from [2004] and its extension (*Attribute*) in Sec. 3.1. Then we describe the notion of abstract task in Sec. 3.2. Finally, we show experimental decomposition results of extremely large PDDL domains, including both large CELL-ASSEMBLY problems as well as very large instances of standard benchmark domains.

2 Background and Motivation: Heterogeneous, Large-Scale, Repetitive Problems

Current domain-independent planners can fail to find solutions to large problems which are composed of smaller, easy problems. Consider the standard Elevator domain, where the task is to transport people between floors with a few elevators (with limited capacity). A small typical instance of the elevator domain, with 3 floors (1F, 2F, 3F) and 1 elevator, is drawn in the left side of Fig. 2. Problems of this scale are easily solved by current domain-independent planners. In the IPC benchmark instances, the number of passengers is at most 7 times the number of elevators in the satisficing track, and much fewer for the optimization track.

Now, consider the much larger instance drawn in the right side of Fig. 2, which might represent what happens at a busy office building when all but one of the elevators are shut down for maintenance. Very large instances of the Elevator problem similar to this can not be solved by current domain-independent planners (as shown in Sec. 4). It is easy to understand why: For a standard, forward-search based planner, the large elevator problem poses a serious challenge because of an extremely high branching factor (whenever the elevator door is open) and a very deep search tree.

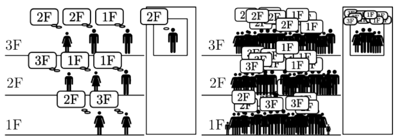


Figure 1: An usual and large Elevator problem instances

On the other hand, a human would have no trouble generating a plan to transport all the passengers to their destinations. When faced with a large problem such as this, a human would not directly search the space of possible action sequences (as current domain-independent planners do). Instead, he/she would notice that on each floor, the passengers waiting on that floor can be divided into two groups, according to their destination. For example, on the 3rd floor, there are passengers who want to go to the 1st floor, and passengers who want to go to the 2nd floor. If there are 3 floors, there can be only 6 kinds of passengers there i.e. $1F \rightarrow 2F$, $1F \rightarrow 3F$, $2F \rightarrow 1F$, $2F \rightarrow 3F$, $3F \rightarrow 1F$, $3F \rightarrow 2F$. If the number of passengers is very large, one natural and fairly efficient solution is a cyclic plan that first transports as many passengers as possible from 1F to 2F, then from 2F to 3F, then 3F to 1F, as many times as needed to transport the $1F \rightarrow 2F$, $2F \rightarrow 3F$, and $3F \rightarrow 1F$ passengers to their destinations. Then we handle the remaining passengers ($1F \rightarrow 3F$, $3F \rightarrow 2F$, $2F \rightarrow 1F$) using a cycles of moves from 1F to 3F, 3F to 2F, and 2F to 1F. More generally, if there are F floors, the passengers on each floor can be partitioned into at most $F-1$ groups, for a total of $F(F-1)$ groups, and a cyclic plan can be similarly constructed. While the cyclic plan can be suboptimal (e.g., if the number of passengers in each group is not equal), this is a reasonably efficient solution.

Other large-scale domains can be approached similarly. If we focus on one type of object (`passenger` in the above example), we can categorize them into groups, by the structural analysis of the initial state (the current floor) and the goal condition (destination floor). This may significantly abstract the basic structure of the domain especially when the number of instances of the object increases faster than the number of the groups does. Note that in our Elevator domain example with 3 floors and 1 elevator, the maximum number of groups is 6, *regardless of the number of passengers*. For a standard planner, increasing the number of passengers makes the problem instance more difficult. However, for an approach that seeks to identify subproblems that can be repeatedly solved using the same method, there is no marginal increase in problem difficulty after a certain point, i.e., the problem difficulty is "saturated".

Furthermore, the interesting things we found is that we can benefit from such categorization even in the standard IPC benchmark problems generated by the official problem generators. Although most subproblems do not form a cycle because they are too different, some *do share* their structure and can form the cycles.

2.1 Cyclic Planning for Homogeneous, Repetitive Problems

To realize these ideas, we adopted a notion of "cyclic planning" and "steady state".

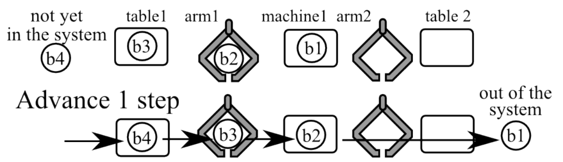


Figure 2: An example of a start/goal state of a cycle, with four products b1,b2,b3 and b4

Cyclic scheduling for robotic cell manufacturing systems, has been studied extensively in the OR literature [Dawande et al., 2005], and the general problem of cyclic scheduling has been considered in the AI literature as well [Draper et al., 1999]. This body of work focused on algorithms for generating effective cyclic schedules for specific domains, and addresses the problem: Given the stages in a robotic assembly system, compute an efficient schedule. Thus, the focus was on efficient scheduling algorithms for particular assembly scenarios.

Similarly, in a generic planning problem, a *cyclic plan* is a sequence of actions that can be performed repeatedly, and a cyclic planner can gain performance when a planning problem is reduced to a cyclic problem. For example, the most common scenario in CELL-ASSEMBLY based manufacturing plants are orders to make N instances of a particular product. In a cyclic plan, the start and end states of the cycle correspond to a "step forward" in an assembly line, where partial products start at some location/machine, and at the end of the cycle, (1) all of the partial products have advanced forward in the assembly line (2) one completed product exits the line, and (3) assembly of a new, partial product has begun (See Fig. 2). Each start/end state is called a *steady state* for a cyclic plan, which is modeled as a set of partially grounded state variables, e.g., S_i={(at b_{i+2} table), (at b_{i+1} painter), (painted b_{i+1}), (at b_i machine)}. The initial and end state of the cyclic plan are S_i and S_{i+1}, respectively.

Recently, Asai and Fukunaga developed ACP, a system which automatically formulates an efficient cyclic problem structure from a PDDL problem for assembling a single instance of a product [Asai and Fukunaga, 2014]. ACP takes as input: (1) a PDDL domain model for assembling a single product instance, (2) the type identifier associated with the end product, (3) the number of instances of the product to assemble, N, and (4) an initial state. Based only on this input (i.e., without any additional annotation on the PDDL domain), ACP generates a cyclic plan which starts at the initial state, then sets up and executes a cyclic assembly for N instances of the product. If ACP can generate a cyclic plan at all, then (because of its cyclic nature) an arbitrarily large number of instances of the target object can be produced.

While ACP provides one solution for a class of homogeneous, repetitive problems as described above, it is not a complete solution to the problem of automatically generating a cyclic formulation for repeated tasks. There are two significant limitations:

First, ACP can not handle *heterogeneous*, repetitive problems, e.g., a manufacturing order to assemble N instances of one product and M instances of another kind of product. They may or may not share the parts and the jobs, e.g. both kinds of products may require the painting but each requires a different additional treatment.

Second, ACP assumes that all instances of objects are indistinguishable. This limitation can be best explained with a concrete example: In the CELL-ASSEMBLY domain, the task is to complete many products on an assembly line with robot arms (Fig. 2). There are a number of assembly tables and machines that perform specific jobs such as painting a product or tightening a screw. In each assembly table, various kinds of `parts` are attached to a `base`, the core component of the product. For example, a problem requires two kinds of parts, `part-a`,`part-b`, to be attached to each one of `base0`,`base1`. The final products look like `base0/part-a/part-b` and `base1/part-a/part-b` . Note that in this example, each `part` is *not* assumed to be associated with any specific `base`. The two `part-a` 's are treated as if they are supplied as needed and each parts are indistinguishable. While this is acceptable in many cases in the CELL-ASSEMBLY domain, the assumption of indistinguishable instances may not be appropriate in other domains, and even in some CELL-ASSEMBLY scenarios.

Consider a CELL-ASSEMBLY domain where each part is labeled and the problem specifies which specific part instance is attached to which base. Fig. 3 describes such a problem i.e. `part-a-0` *must* be attached to `b-0` specifically and so on. Assuming the implementation of ACP system is based on the plan analysis of a unit product, which is one of `b-0` or `b-1` in this case (let it be the former), then the produced cyclic plan contains a parametrized `base` but also a static `part-a-0` object, which leads to inconsistency when we substitute the parametrized `base` with `b-1`. It shows that the information of the `base` objects is not sufficient in order to unroll a cyclic plan: there is a lack of information about the associations between the bases and the parts.

```
(:objects b-0 b-1 - base
          part-a-0 part-a-1 - part
          part-b-1 part-b-1 - part ...)
(:init (part-base part-a-0 b-0)
       (part-base part-a-1 b-1)
       (part-base part-b-0 b-0)
       (part-base part-b-1 b-1) ...)
```

Figure 3: CELL-ASSEMBLY with distinctly labeled parts.

Therefore we need to automatically detect such structure consisting of a core object and the associated objects. Once we find this composite structure, we can treat it as an abstract representation of a unit product. A cyclic plan for the abstract structure can be generated, and when the cyclic plan is "unrolled", individual instances of the parts of the composite object can be mapped to the cyclic plan.

3 Solving Large-Scale, Heterogeneous Repetitive Problems by Decomposition

Given a large-scale problem such as a heterogeneous cell assembly or large-scale Elevator problems described in the previous sections, we can try to solve them by decomposing them into subproblems that can be handled by existing approaches. Our overall approach consists of following 5 steps:

1. Divide the set of objects in a problem by extracting abstract type [2004] information, based on the structural analysis on the initial state. Instances of an abstract type are called abstract components and a PDDL object is allocated to each *slot* defined by abstract type.
2. Extract the initial state and the goal condition that is related to each component. Together with a component, they form an abstract task.
3. Compute the compatibility between tasks and categorize the tasks into groups such that each task in a same group share the same plan.
4. Solve each group of the same tasks as a cyclic planning problem. Each group can be solved separately. We finally get a set of unrolled cyclic plans.
5. Interleave the solutions to the decomposed subproblems. Unrolled plan of each group is interleaved, treated like Abstract Actions in HTN terminology.

The remainder of the paper describes an approach for partitioning of a heterogeneous, repetitive problem into groups of related, easier subproblems, i.e., steps 1-3 above. In particular, we describe a method for generating groups of subproblems that are identical when abstracted (and therefore, only 1 instance from each group needs to be solved).

After the original problem has been partitioned, then Step 4 (solution of subproblems) depends on the partitioning results. If there are numerous instances of each type of subproblem, then a cyclic planner such as ACP can be applied in order to generate an efficient plan for that group of subproblem. On the other hand, groups with a single instance member can be solved using a standard planner.

The final step, combining the subproblem solutions into a single plan for the original problem (Step 5) , is future work. In problems with no (non-replenishable) resource constraints such as the Elevator and CELL-ASSEMBLY domains, a satisficing plan can be obtained by sequentially executing the plans for the subproblems (stiching the end state of each subplan to the initial state of the next subplan may be necessary). In cases where more efficient plans are desired, or if there are resource constraints, combining the subplans is a nontrivial problem which can be at least as difficult as HTN planning. In such cases, our decomposition-based approach may not result in a feasible plan.

3.1 Component Abstraction and Attributes

Our method for identifying subproblems in large-scale, repetitive problems is based on the component abstraction method for macro generation by [Botea, Müller, and Schaeffer, 2004]. In particular, we use only the first sets of components and we extend their notion of an abstract type. This section reviews the method for extracting abstract types in [2004] and discusses the completeness of the approach.

Previous Work: Identifying Abstract Components [2004] The basic idea is to build a *static graph* of the problem and partition it into abstract components by seeding the components with one node from the graph, and then iteratively merging adjacent nodes and "growing" the component. Fig. 3.1 illustrates a possible static graph and some of its abstract components. Each small circle represents the objects appeared in Fig. 3, where "b0" and "pb0" is an abbreviation of `b-0` and `part-b-0` etc. Each color and pattern of a node implies its *type*: different colors suggest different types.

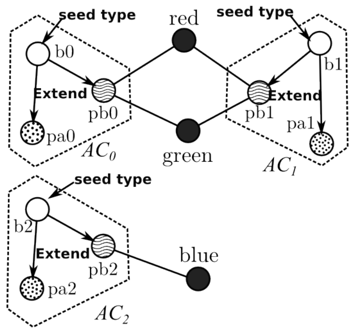


Figure 4: Example of a static graph and components

The static graph of a problem Π is an undirected graph $\langle V, E\rangle$, where nodes V are the objects in the problem and the edges E is a set of static facts in the initial state. *Static facts* are the facts (propositions) that are never added nor removed by any of the actions and only possibly appear in the preconditions. The graph may be unconnected.

Each component is a subgraph of the static graph. The decomposition into components proceeds as follows:

1. First, a *Seed Type* is selected (e.g., randomly). In the figure, `base` ○ is selected as a seed type.
2. Next, all objects of the seed type in the static graph are collected, and an abstract component is created for each selected object. In the figure, the seed objects are `b0,b1,b2`, and their corresponding abstract components are `(b0),(b1),(b2)`.
3. A *fringe* node is a node that is currently not part of any components and adjacent to some node in a component. If no fringe node exists, then it either restart the search with another randomly-selected seed type and the rest of the graph, or terminates if no seed type remains.
4. Select a set of fringe nodes simultaneously, choosing a type and then selecting all fringe nodes of that type. For example, if we choose a wave-patterned node `pb0` ▤ for the white node `b0` ○, then we simultaneously choose `pb1` ▤ for `b1` ○ and `pb2` ▤ for `b2` ○. The selection order is not specified.
5. Merge the selected nodes into the component that they are adjacent to, e.g., if we choose `pb0` ▤ first, then the components are updated from `(b0),(b1),(b2)` to `(b0 pb0),(b1 pb1),(b2 pb2)`.
6. At this point, check if the resulting components share any part of the structure, and if so, discard all the fringe nodes newly added in step 5. For example, extending the top-left AC_0 by adding `red` ● simultaneously causes top-right AC_1 to include `red`, resulting in sharing `red`. Since the objects merged in the step 5 are `red`, `green` and `blue`, they are all excluded.

Towards a Systematic Search for the Best Abstraction The original abstract type detection by Botea et al is a randomized, greedy procedure. The seed type is randomly selected in step 1 and step 3. In addition, in step 4, the type of fringe nodes is selected arbitrarily (the selection order was not specified in [2004].) While this kind of greedy algorithm suits their goal of quickly extracting macro-operators, this is not appropriate for our purpose (solving large problems by decomposition) due to two reasons.

First, depending on the choices made regarding Seed Type selection and fringe node type selection, we may fail to find a component abstraction that includes any nodes other than the seed node. In the case of macro abstraction, such a failure is not necessarily fatal because the macro system is basically a speedup mechanism, and even if a macro is not found, the base-level planner may still be able to solve the problem. On the other hand, our objective is to solve problems that are completely beyond the reach of standard planners (with hundreds of thousands of objects), so failure to find an appropriate abstract component is equivalent to failure to solve the problem at all.

Second, there may be multiple possible component abstractions for any static graph, and for our purpose (solving very large problems by decomposition), it is not sufficient to find *any* component abstraction. For example, consider the integration of component abstraction into the ACP cyclic planner. ACP extracts *lock* and *owner* predicates associated with each object. A "good" component abstraction in this context is one which results in the extraction of a large number of *locks*, which, in turn, results in efficient plans with good parallel resource usage. Since *locks* are extracted for each object and they are aggregated for each component which contains the object, large components tends to have a large number of *locks* in general.

It is straightforward to modify the abstract type detection algorithm to systematically enumerate and consider all possible component abstractions that can be extracted from a static graph. However, we don't currently perform a full enumeration. We run the abstract component detection algorithm from a fresh initial state for all possible seed types. We only use the *first* set of components that has been extended from the *first* seed type given in the initial input. Thus, the algorithm stops at step 3 if the fringe nodes exhausted. While this is more systematic than the original algorithm by Botea et al, the choice of fringe-node type is still arbitrary (simple queue with no priority). The more systematic approach to good abstract components (pruning the space of candidate component abstractions) is future work.

Attributes Conceptually, an abstract component represents an inseparable groups of objects such as a name, arms and legs of a human. The fact that an arm belongs to a person is never removed or added (or it means the transplantation or loss of the arm). Also, an arm does not belong to more than one person (again except a few cases).

In contrast, some nodes in the static graph represent *Attributes* that belong to many groups of objects, e.g. hair color, ethnicity, and gender are attributes that are shared among people. We extend the abstract type detection algorithm of [2004] above to identify such attributes. In Step 6 above, all nodes that prevented the extension are identified as *attributes*, e.g., `red,blue,green` in Fig. 3.1. Attributes are used in order to constrain the search for plan-compatible abstract tasks, described below.

3.2 Abstract Task

We define an *abstract task* as a triple consisting of (1) an abstract component AC, (2) a subset of propositions from the initial states relevant to AC, and (3) a subset of goal propositions relevant to AC.

In order to find (2) and (3), we collect the *fluent facts* in the problem description. Fluent facts are the facts which are not static. For each abstract component, we collect all fluent facts in the initial and goal states which contain one of the objects in AC in its parameters. For example, the initial state in the PDDL model in Fig. 3 may include a fact `(not-painted b0)`. It can be removed by some actions like `paint`, so it is fluent. Since `(not-painted b0)` has `b0` in its argument, it is considered as one of the initial states of the top-left component (AC_0) in Fig. 3.1. Similarly, facts like `(at b0 table1)`, `(at pa0 tray-a)` are the possible candidates for the initial states of AC_0. Likewise, the goal condition of AC_0 may be `(at b0 exit)`, `(is-painted pa0 red)`, `(assembled b0 pa0)` etc.

After all the abstract tasks are identified, we check the compatibility between the tasks. The compatibility is checked via replacing the parameters in a plan with the objects in a component. This finally allows the objects to be correctly categorized, as the people on each floor in Elevator domain were divided into groups.

Plan-wise Compatibility of Tasks In order to define the plan-wise compatibility of tasks, we first define a notion of *component plan* and its compatibility.

Let $X = \{o_0, o_1 \ldots\}$ an abstract component. A component plan of X is a result plan of *component problem* of X. Let the original planning problem is $\Pi = \langle \mathcal{D}, O, I, G\rangle$ where $\mathcal{D}$ is a domain, O is the set of objects and I, G is the initial/goal condition. Also, Let Y be another component. Then a component problem is a planning problem $\Pi_X = \langle \mathcal{D}, O_X, I_X, G_X\rangle$ where:

$$\begin{aligned} O_X =& X \cup \{O \setminus X \ni o \mid \forall Y \neq X; o \notin Y\} \\ I_X =& \{I \ni f \mid \mathit{params}(f) \cap (O \setminus O_X) = \emptyset\} \\ G_X =& \mathit{goal}(X) \end{aligned}$$

The definition of O_X specifies that it removes any objects that belong to another component Y. Note that the objects not included in any component remain as it is. I_X specifies that an initial condition is removed when it has a removed object in its parameters list. G_X is same as the goal conditions of the abstract task of X.

We solve a component problem Π_X with a domain-independent planner such as Fast Downward. However, in some domains no solutions exist in Π_X due to the removed objects and initial conditions. In such cases, we restore O and I and re-run the planner on $\langle \mathcal{D}, O, I, G_X\rangle$. This is called an *Object Restoration*. The computation of a component plan is instantaneous in most cases, but large instances tend to require more time and memory. As shown in the experiments, we found that when a large number of unused objects is restored in O, the PDDL $\rightarrow$ SAS converter in Fast Downward can become the bottleneck.

Solving component problem yields an *component plan*. The component plans P_X, P_Y are compatible if P_X can be used as a plan of Y by replacing the parameters i.e. if we replace all references to the objects in X in P_X with the corresponding objects in Y, then the modified (mapped) plan P'_X is a valid plan of Π_Y.

Finally, we define two tasks are *plan-wise compatible* when the following conditions hold:

1. The components of the two tasks are of the same abstract type.
2. The two tasks shares the same set of attributes.
3. The graph structure designated by their initial/goal condition are isomorphic, just like abstract-type, e.g. if the task of AC_0 in Fig. 3.1 has an initial state `(at b0`

table1), then a compatible task of AC_1 should also contain (at b1 table1) in its initial state.

4. One of the *component plans* of the component problem of each task are compatible.

We categorize the extracted abstract tasks according to the plan-wise compatibility. However, solving the component problems involves running a domain-independent planner, so computing component plans for every abstract task should be avoided if possible. Thus, we use attributes (condition 2 above) in order to filter the candidate pairs before checking whether their component plans are compatible.

Categorizing a group of N elements based on a equality function requires $O(N^2)$ comparisons between the elements in a naive implementation. However, in the IPC problems, most tasks are not compatible, which means they end up in many groups of only a single task in it. Clearly, a group of one task requires no further categorization. A similar observation can be made regarding condition 3.

The use of attribute-based comparison greatly reduces the number of pairs whose component plans need to be compared. Consider three similar factory assembly tasks, t_1, t_2, t_3, that require painting of an object with different colors (blue, red, green). Also, assume that the available colors in each painting machine is limited e.g. machine m_1 supports red only and m_2 supports green and blue. Now the plans for t_1 and t_2 are incompatible because they use the different machines. We can detect these incompatibilities without spending all the effort of running a planner, getting a mapped plan and validate it.

While we show experimentally that pre-categorization according to attributes is highly effective (Sec. 4.2), this method is subject to false-negatives, and can result in the compatible pairs being discarded. Continuing the previous example, plans for t_2 and t_3 is likely to be compatible because their plans use the same machine. However, once we use the condition 2, it divides t_2 and t_3 into the different groups because they use the different colors.

The number of calls to the underlying planner can also be optimized. Since the compatibility is transitive (given the condition 2 and 3. Proof omitted), we can check the compatibility between Y and Z by instead just checking the compatibility between X and Z. Then we can reuse P_X because the compatibility check requires only the plan P_X (and does not require P_Z).

4 Experimental Results

We evaluate abstract-task based problem decomposition on the CELL-ASSEMBLY domain as well as large instances of several IPC domains.

4.1 Categorization of CELL-ASSEMBLY Problems

We evaluated our decomposition method on both homogeneous and heterogeneous CELL-ASSEMBLY-EACHPARTS. Both types of problems are defined on the same manufacturing plant with the same tables and machines. However, the latter processes two completely different kinds of products at the same time. The results are shown in Fig. 5.

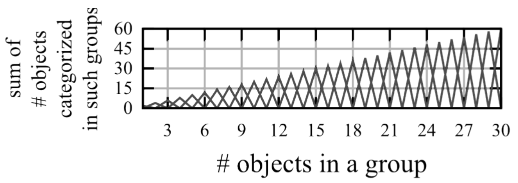


Figure 5: The categorization result of CELL-ASSEMBLY-EACHPARTS *2a2b-mixed* problems with seed base. Each problem contains n 2a-tasks and the same n 2b-tasks, where $1 \leq n \leq 30$. Each line represents one problem. The x-axis represents the number of objects in each categorized group, while the y-axis represents the total number of objects in the groups of x objects. Therefore $(x, y) = (15, 30)$ means there are 2 groups of 15 components. For each problem, the tasks are divided into 2 groups of n tasks, where each group represents *2a* and *2b*. This shows that our algorithm correctly categorized two groups of tasks in each problem, despite the combination of the mixed orders and the variation within each group (Sec. 2.1).

On the homogeneous CELL-ASSEMBLY-EACHPARTS problems, our algorithm correctly identifies all component tasks and labels them as plan-wise compatible. On the heterogeneous problems, it identifies that there are two kinds of tasks and the objects are compatible within each group. (When we say "heterogeneous $x + y$ problem", it contains x instances of one product and y instances of another product.) Decompositions based on the seed types other than base was also successful. The seed was part, and the detailed analysis suggested that the same abstract type was detected from the different seed types. Given this fact, we consider the exhaustive attempts on all seed types are necessary.

4.2 Categorization results for IPC Domains

We also evaluate our decomposition method on several IPC domains including Satellite (minimally modified, see below), woodworking, openstacks, elevators, barman, and rover. Satellite-typed is a typed variant of IPC2006 satellite. While the original domain is already "typed" by the use of 1-argument type predicates, we simply converted the type predicates to standard, explicit PDDL type annotations. This conversion has no effect on the structure of the domain or instances. Although we performed this modification manually, an automatic procedure such as TIM[Fox and Long, 1998] could have been used instead. Very large problem instances for the IPC domains were generated using either the scripts that are found in IPC 2011 result archive [1] (for Woodworking, Barman and ROVER), and our own generator for the rest.

First, in order to understand the limitations of current State-of-the-Art planners, we run the current version of Fast Downward[2] using the "seq-sat-lama-2011" satisficing configuration, which emulates the IPC2011 LAMA planner. The maximum search time is limited to 6 hours, and memory limit of 15[GB] on an Intel Xeon E5410@2.33GHz.

The table below shows the largest problem instances that were solved. Stars ($*$) mean it uses the generators and the

[1]svn://svn@pleiades.plg.inf.uc3m.es/ipc2011/data

[2]http://www.fast-downward.org

problems used in IPC. (In this table, we also summarized the seed types which are used in the later categorization.)

Domains	limit	seed type
cell-assembly	(single product, 14 bases)	base,part
-eachparts	(mixed products, 6+6 bases)	
satellite-typed	(17 satellites,39 instruments,	direction
	13 modes,310 directions)	
IPC domains		
woodworking*	p86 (227 parts and x1.2 wood)	part
openstacks	(70 orders and products)	order, product
elevators	(4 slow&fast elevators, 40 floors,	passenger
	270 passengers, 10 floors/area)	
barman-sat11*	(4 ingredients, 93 shots and cocktails)	shot,cocktail
rover*	N/A (IPC problems were solved up to p40)	objective

Table 1: Domains used in the evaluation.

Assuming that once a cyclic plan is generated, the marginal cost of processing an additional product within a group is almost *zero*, we can summarize the effect of categorization by comparing the number of components with the number of groups of components. Fig. 6(1) shows the result.

The points on the line $x = y$ in Fig. 6(1) shows that we cannot get a meaningful categorization in some configurations. The reason is twofold: Firstly, the choice of *seed* was inappropriate. Indeed, in `barman` domain, categorization based on `cocktail` did not result in components larger than the initial nodes, while that of `shot` did. This is due to the greedy nature of the abstract type detection algorithm, as described in [Botea, Müller, and Schaeffer, 2004]. The same thing applies to the results of the other seed types (though not shown here), such as the decomposition of CELL-ASSEMBLY-EACHPARTS based on `arm`, `table`, `job` and so on. This indicates that restarting the search with all seeds as described in Section 3.1 is required to ensure that a meaningful categorization is found if one exists. Secondly, in OPENSTACKS, both seeds `order` and `product` failed to get a meaningful information. In this case, the reason is attributed to the domain's characteristics itself i.e. it is possible that the domain has no inherent meaningful categorization.

Evaluation of the Optimization Methods Here we show the effect of pre-categorization described in Sec. 3.2. Though we are not able to include detailed results here, Fig. 6(2) shows the distribution of pre-categorization (based on the conditions 1 to 3, Sec. 3.2). It shows that the number of comparisons that are actually performed is far less than those required by the naive method because most categorizations are already done i.e. most tasks are already categorized into groups of a few elements (1 or 2 elements).

We also compare the actual number of calls to the underlying planner with a naive implementation. In a naive implementation, N components require N calls to the planner. Fig. 6(3) shows that the evaluation was reduced by a factor of 2. The results in all domains in Table 1 are shown at once.

Total Elapsed Time of the Categorization Elapsed time is a key factor when our method is considered as a preprocessing step for planning. We show the results in Fig. 6(4). Unfortunately, some problems are found to take very long time to compute the categorization due to the heavy object restoration (Sec. 3.2). The figure clearly shows that the most time-consuming ($t \approx 10^5$) decompositions required the component problems with object restoration $\langle \mathcal{D}, O, I, G_X \rangle$, not by $\langle \mathcal{D}, O_X, I_X, G_X \rangle$. In Fig. 6(5), we show that the current bottleneck in solving a restored problem is mainly the PDDL → SAS converter in Fast Downward. Addressing this issue remains future work.

5 Related Works

The overall approach we are pursuing, which is to decompose a large problem into relatively easy subproblems, is inspired by previous work on problem decomposition [Yang, Bai, and Qiu, 1994] and Hierarchical Task Network (HTN) planning [Erol, Hendler, and Nau, 1994]. In HTN, problem decomposition is done by *methods* which describes how an abstract-level task can be decomposed into the smaller subtasks. It naturally supports the cyclic structure by allowing self-recursive decomposition of compound tasks. Thus, decomposition gives HTN strictly more expressivity than that of classical planning [1994].

HTN differs from our approach in that *methods* are mostly written by the human experts, while ours is based on static problem analysis. However, sevaral approaches has already been made for learning the methods automatically. HTN-MAKER[Hogg, Munoz-Avila, and Kuter, 2008] learns methods from existing plans built by experts and a set of *annotated tasks*. The learning process works in a hierarchical manner. Similarly, LIGHT[Nejati, Langley, and Konik, 2006] system induces methods also from the expert plan but by *backward skill chaining* from the goal.

Our approach is related to macro abstraction systems such as Macro-FF [Botea et al., 2005], which automatically identifies reusable plan fragments, and in fact, the component abstraction framework we extended was originally used by Botea et al to identify macros [Botea, Müller, and Schaeffer, 2004]. Macro systems strive to provide a very general abstraction mechanism, but are typically limited to relatively short macros (e.g., 2-step macros in Macro-FF). Our approach, on the other hand, focuses on identifying 'very long macros" (10-30 steps per cycle) corresponding to specific abstract tasks.

Another related work is *symmetry detection* in [Fox and Long, 1999]. It builds symmetry groups of objects and actions by exploiting the initial and goal condition in a given problem. The symmetry is incrementally broken during the search and the information is used to suppress the branching factor, while our method is focused on the preprocessing before the actual search. Also, their work do not consider the structure of objects, thus shares the similar aspects with our previous approach described in Sec. 2.1.

6 Conclusions

This paper presents preliminary work on a decomposition-based approach for solving large scale planning problems with a repetitive structure in domains such as factory assembly. We presented an overview of our decomposition-based

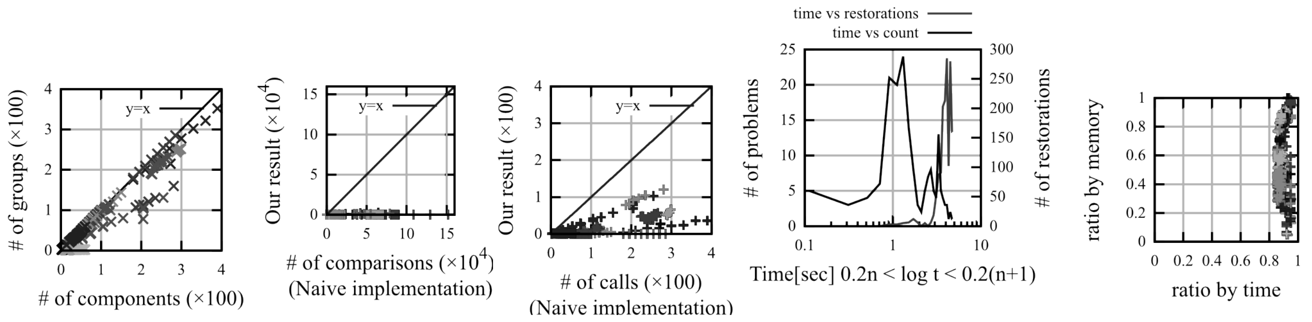


Figure 6: (1-5) The figures are given numbers from left to right. Results of all domains categorized using all seed types are shown at once. Points from the different configuration have the different density. (1) The number of components vs the number of categorized groups. (2) Number of comparisons for each problems. Our pre-categorization method significantly reduced the number of comparisons necessary to categorize the tasks. (3) Number of evaluations for each problems. Our memoization strategy significantly reduces the number of calls to the underlying planner by a factor of ≈ 2. (4) Histogram of the total elapsed time[sec], averaged in $0.2n \leq \log t \leq 0.2(n+1)$ for each integer n. (5) x-axis is a ratio of translate/(translate + preprocess + search) measured by time, while y-axis is measured by memory. We show the results from all the domains, regardless of whether the object restoration has occured or not. It clearly shows that the search in a component problem is easy because of the limited number of conditions in G_X, and the computation is usually dominated by the translator.

approach, and described a novel method for automatically detecting such a repeated structure, based on categorization of objects in a problem. We showed experimentally that on very large problem instances, our method is able to successfully decompose the problems into tasks that satisfy a particular compatibility criteria (plan-wise compatibility). We showed that plan-wise compatibility can be found not only in large instances of the factory assembly domain that is the primary motivation for this line of work, but also in large instances of IPC2011 domains.

In the overall approach to solving large-scale, heterogeneous repetitive problems outlined in Section 3, this paper addresses Steps 1 and 3. Step 4 (cyclic planning) has already been addressed in [Asai and Fukunaga, 2014]. The next step is to address the remaining steps (2, 5), which would result in a system that can fully automatically approach the problem of solving large-scale repetitive problems.

There are several directions for improving the decomposition strategy proposed in this paper. First, as discussed in Section 3.1, the abstract type detection algorithm should be made more systematic, and a focused search for larger components should be implemented. In addition, a smarter object restoration strategy which minimize the number of objects to be restored (Section 3.2) because currently all objects and initial configurations are restored again in $\langle \mathcal{D}, O, I, G_X \rangle$, causing the underlying planner to be confused by the unnecessarily large number of objects.

References

Asai, M., and Fukunaga, A. 2014. Fully automated cyclic planning for large-scale manufacturing domains. In *ICAPS*.

Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-ff: Improving ai planning with automatically learned macro-operators. *J. Artif. Intell. Res.(JAIR)* 24:581–621.

Botea, A.; Müller, M.; and Schaeffer, J. 2004. Using component abstraction for automatic generation of macro-actions. In *Proceedings of ICAPS*, 181–190.

Bylander, T. 1994. The computational complexity of propositional strips planning. *Artificial Intelligence* 69(1):165–204.

Dawande, M.; Geismar, H. N.; Sethi, S. P.; and Sriskandarajah, C. 2005. Sequencing and scheduling in robotic cells: Recent developments. *Journal of Scheduling* 8(5):387–426.

Draper, D.; Jonsson, A.; Clements, D.; and Joslin, D. 1999. Cyclic scheduling. In *Proc. IJCAI*.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. Htn planning: Complexity and expressivity. In *AAAI*, volume 94, 1123–1128.

Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*.

Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *IJCAI*, volume 99, 956–961.

Helmert, M.; Röger, G.; et al. 2008. How good is almost perfect?. In *AAAI*, volume 8, 944–949.

Hogg, C.; Munoz-Avila, H.; and Kuter, U. 2008. HTN-MAKER: Learning htns with minimal additional knowledge engineering required. In *AAAI*, 950–956.

Nejati, N.; Langley, P.; and Konik, T. 2006. Learning hierarchical task networks by observation. In *Proceedings of the 23rd international conference on Machine learning*, 665–672. ACM.

Ochi, K.; Fukunaga, A.; Kondo, C.; Maeda, M.; Hasegawa, F.; and Kawano, Y. 2013. A steady-state model for automated sequence generation in a robotic assembly system. *SPARK 2013*.

Yang, Q.; Bai, S.; and Qiu, G. 1994. A framework for automatic problem decomposition in planning. In *AIPS*, 347–352.

Eliminating All Redundant Actions from Plans Using SAT and MaxSAT

Tomáš Balyo
Department of Theoretical Computer Science
and Mathematical Logic,
Faculty of Mathematics and Physics
Charles University in Prague
biotomas@gmail.com

Lukáš Chrpa
PARK Research Group
School of Computing and Engineering
University of Huddersfield
l.chrpa@hud.ac.uk

Abstract

Satisfiability (SAT) techniques are often successfully used for solving planning problems. In this paper we show, that SAT and maximum satisfiability (MaxSAT) can be also used for post-processing optimization of plans. We will restrict ourselves to improving plans by removing redundant actions from them which is a special case of plans optimization. There exist polynomial algorithms for removing redundant actions, but none of them can remove all such actions since guaranteeing that a plan does not contain redundant actions is NP-complete. We introduce two new algorithms, based on SAT and MaxSAT, which remove all redundant actions. The MaxSAT based algorithm additionally guarantees to remove a maximum set of redundant actions. We test the described algorithms on plans obtained by state-of-the-art planners on IPC 2011 benchmarks. The proposed algorithms are very fast for these plans despite the complexity results.

Introduction

Automated Planning is an important research area for its good application potential (Ghallab, Nau, and Traverso 2004). With intelligent systems becoming ubiquitous there is a need for planning systems to operate in almost real-time. Sometimes it is necessary to provide a solution in a very little time to avoid imminent danger (e.g damaging a robot) and prevent significant financial losses. Satisficing planning engines such as FF (Hoffmann and Nebel 2001), Fast Downward (Helmert 2006) or LPG (Gerevini, Saetti, and Serina 2003) are often able to solve a given problem quickly, however, quality of solutions might be low. Optimal planning engines, which guarantee the best quality solutions, often struggle even on simple problems. Therefore, a reasonable way how to improve the quality of the solutions produced by satisficing planning engines is to use post-planning optimization techniques.

In this paper we restrict ourselves to optimizing plans by only removing redundant actions from them. Guaranteeing that a plan does not contain redundant actions is NP-complete (Fink and Yang 1992). There are polynomial algorithms, which remove most of the redundant actions, but none of them removes all such actions. We propose two new algorithms which are guaranteed to remove them all. One uses satisfiability (SAT) solving, the other one relies on maximum satisfiability (MaxSAT) solving. We compare our algorithms with a heuristic algorithm on plans obtained by state-of-the-art planners on IPC 2011 benchmarks.

Related Work

Various techniques have been proposed for post-planning plan optimization. Westerberg and Levine (2001) proposed a technique based on Genetic Programming, however, it is not clear whether it is required to hand code optimization policies for each domain as well as how much runtime is needed for such a technique. Planning Neighborhood Graph Search (Nakhost and Müller 2010) is a technique which expands a limited number of nodes around each state along the plan and then by applying Dijsktra‘s algorithm finds a better quality (shorter) plan. This technique is anytime since we can iteratively increase the limit for expanded nodes in order to find plans of better quality. AIRS (Estrem and Krebsbach 2012) improves quality of plans by identifying suboptimal subsequences of actions according to heuristic estimation (a distance between given pairs of states). If the heuristic indicates that states might be closer than they are, then a more expensive (optimal) planning technique is used to find a better sequence of actions connecting the given states. A similar approach exists for optimizing parallel plans (Balyo, Barták, and Surynek 2012). A recent technique (Siddiqui and Haslum 2013) uses plan deordering into ‘blocks’ of partially ordered subplans which are then optimized. This approach is efficient since it is able to optimize subplans where actions might be placed far from each other in a totaly ordered plan.

Determining and removing redundant actions from plans is a specific sub-category of post-planning plan optimization. An influential work (Fink and Yang 1992) defines four categories of redundant actions and provides complexity results for each of the categories. One of the categories refers to Greedily justified actions. A greedily justified action in the plan is, informally said, such an action which if it and actions dependent on it are removed from the plan, the plan becomes invalid. Greedy justification is used in the Action Elimination (AE) algorithm (Nakhost and Müller 2010) which is discussed in detail later in the text. Another of the categories refers to Perfectly Justified plans, plans in which no redundant actions can be found. Minimal reduction of plans (Nakhost and Müller 2010) is a special case of Perfectly Justified plans having minimal cost of the plan. Both Perfect Justification and Minimal reduction are NP-complete. Determining redundant pairs of inverse actions (inverse actions are those that revert each other's effects), which aims to eliminate the most common type of redundant actions in plans, has been also recently studied (Chrpa, McCluskey, and Osborne 2012a; 2012b).

Preliminaries

In this section we give the basic definitions and properties used in the rest of the paper.

Satisfiability

A *Boolean variable* is a variable with two possible values *True* and *False*. A *literal* of a Boolean variable x is either x or $\neg x$ (*positive* or *negative literal*). A *clause* is a disjunction (OR) of literals. A clause with only one literal is called a *unit clause* and with two literals a *binary clause*. An implication of the form $x \Rightarrow (y_1 \vee \cdots \vee y_k)$ is equivalent to the clause $(\neg x \vee y_1 \vee \cdots \vee y_k)$. A *conjunctive normal form (*CNF*) formula* is a conjunction (AND) of clauses. A truth assignment ϕ of a formula F assigns a truth value to its variables. The assignment ϕ satisfies a positive (negative) literal if it assigns the value True (False) to its variable and ϕ satisfies a clause if it satisfies any of its literals. Finally, ϕ satisfies a CNF formula if it satisfies all of its clauses. A formula F is said to be satisfiable if there is a truth assignment ϕ that satisfies F. Such an assignment is called a *satisfying assignment*. The satisfiability problem (SAT) is to find a satisfying assignment of a given CNF formula or determine that it is unsatisfiable.

Partial Maximum Satisfiability

A *partial maximum satisfiability (PMaxSAT) formula* is a CNF formula consisting of two kinds of clauses called *hard* and *soft* clauses. A PMaxSAT formula is satisfied under a truth assignment ϕ if it satisfies all of its hard clauses.

The *partial maximum satisfiability problem* (PMaxSAT) is to find a satisfying assignment ϕ for a given PMaxSAT formula such that ϕ satisfies as many soft clauses as possible.

Planning

In this section we give the formal definitions related to planning. We will use the multivalued SAS+ formalism (Bäckström and Nebel 1995) instead of the classical STRIPS formalism (Fikes and Nilsson 1971) based on propositional logic.

A planning task Π in the SAS+ formalism is defined as a tuple $\Pi = \{X, O, s_I, s_G\}$ where

- $X = \{x_1, \dots, x_n\}$ is a set of multivalued variables with finite domains $\mathrm{dom}(x_i)$.
- O is a set of actions (or operators). Each action $a \in O$ is a tuple $(\mathrm{pre}(a), \mathrm{eff}(a))$ where $\mathrm{pre}(a)$ is the set of preconditions of a and $\mathrm{eff}(a)$ is the set of effects of a. Both preconditions and effects are of the form $x_i = v$ where $v \in \mathrm{dom}(x_i)$.
- A state is a set of assignments to the state variables. Each state variable has exactly one value assigned from its respective domain. We denote by S the set of all states. $s_I \in S$ is the initial state. s_G is a partial assignment of the state variables (not all variables have assigned values) and a state $s \in S$ is a goal state if $s_G \subseteq s$.

An action a is *applicable* in the given state s if $\mathrm{pre}(a) \subseteq s$. By $s' = \mathrm{apply}(a, s)$ we denote the state after executing the action a in the state s, where a is applicable in s. All the assignments in s' are the same as in s except for the assignments in $\mathrm{eff}(a)$ which replace the corresponding (same variable) assignments in s.

A *(sequential) plan* P of *length* k for a given planning task Π is a sequence of actions $P = \{a_1, \dots, a_k\}$ such that $s_G \subseteq \mathrm{apply}(a_k, \mathrm{apply}(a_{k-1} \dots \mathrm{apply}(a_2, \mathrm{apply}(a_1, s_I)) \dots))$. We will denote by $|P|$ the length of the plan P.

Redundant Plans

A plan P for a planning task Π is called *redundant* if there is a subsequence P' of P ($|P'| < |P|$), such that P' is a valid plan for Π. The actions in P that are not present in P' are called *redundant actions*. A plan which is not redundant is called a *perfectly justified plan*.

A plan P for a planning task Π is called an *optimal plan* if there is no other plan P' for Π such that $|P'| < |P|$. Note, that a perfectly justified plan is not necessarily an optimal plan. On the other hand, an optimal plan is always perfectly justified.

Determining whether a plan is perfectly justified is NP-complete (Fink and Yang 1992). Nevertheless, there are several heuristic approaches, which can identify most of the redundant actions in plans in polynomial time. One of the most efficient of these approaches was introduced in (Fink and Yang 1992) under the name Linear Greedy Justification. It was reinvented in (Nakhost and Müller 2010) and called Action Elimination. In this paper we use the latter name.

Action Elimination (see Figure 1) tests for each action if it is greedily justified. An action is greedily justified if removing it and all the following actions that depend on it makes the plan invalid. One such test runs in $O(np)$ time, where $n = |P|$ and p is the maximum number of preconditions and effects any action has. Every action in the plan is tested, therefore Action Elimination runs in $O(n^2p)$ time.

There are plans, where Action Elimination cannot eliminate all redundant actions (Nakhost and Müller 2010). An interesting question is how often this occurs for the planning domains used in the planning competitions (Coles et al. 2012). To find out, first we need to design an algorithm that always eliminates all redundant actions, i.e., find perfectly justified plans. As mentioned earlier, this problem is NP-complete and therefore we find it reasonable to solve it using a SAT reduction approach. In the next section we will introduce a translation of this problem into SAT.

Satisfiability Encoding of Plan Redundancy

This section is devoted to introducing an algorithm, which given a planning task Π and a valid plan P for Π, outputs a

```
ActionElimination ($\Pi$, $P$)
$s := s_I$
$i := 1$
repeat
   mark($a_i$)
   $s' := s$
   for $j := i+1$ to $|P|$ do
      if applicable($a_j$, $s'$) then
         $s'$ := apply($a_j$, $s'$)
      else
         mark($a_j$)
   if goalSatisfied($s'$) then
      $P$ := removeMarked($P$)
   else
      unmarkAllActions()
      $s$ := apply($a_i$, $s$)
   $i := i+1$
until $i > |P|$
return $P$
```

Figure 1: Pseudo-code of the Action Elimination algorithm as presented in (Nakhost and Müller 2010).

CNF formula $F_{\Pi,P}$, such that $F_{\Pi,P}$ is satisfiable if and only if P is a redundant plan for Π.

We provide several definitions which are required to understand the concept of our approach. An action a is called a *supporting action* for a condition c if $c \in \text{eff}(a)$. An action a is an *opposing action* of a condition $c := x_i = v$ if $x_i = v' \in \text{eff}(a)$ where $v \neq v'$. The *rank* of an action a in a plan P is its order in the sequence P. We will denote by $Opps(c,i,j)$ the set of ranks of opposing actions of the condition c which have their rank between i and j ($i \leq j$). Similarly, by $Supps(c,i)$ we will mean the set of ranks of supporting actions of the condition c which have ranks smaller than i.

In our encoding we will have two kinds of variables. First, we will have one variable for each action in the plan P, which will represent whether the action is required for the plan. We will say that $a_i = True$ if the i-th action of P (the action with the rank i) is required. The second kind of variables will be option variables, their purpose and meaning is described below.

The main idea of the translation is to encode the fact, that if a certain condition c_i is required to be true at some time i in the plan, then one of the following must hold:

- The condition c_i is true since the initial state and there is no opposing action of c_i with a rank smaller than i.
- There is a supporting action a_j of c_i with the rank j and there is no opposing action of c_i with the rank between j and i.

These two kinds of properties represent the options for satisfying c_i. There is at most one option of the first kind and at most $|P|$ of the second kind. For each one of them we will use a new option variable $y_{c,i,k}$, which will be true if the condition c at time i is satisfied using the k-th option.

Now we demonstrate how to encode the fact, that we require condition c to hold at time i. If c is in the initial state, then the first option will be expressed using the following conjunction of clauses.

$$F_{c,i,0} = \bigwedge_{j \in Opps(c,0,i)} (\neg y_{c,i,0} \vee \neg a_j)$$

These clauses are equivalent to the implications below. The implications represent that if the given option is true, then none of the opposing actions can be true.

$$(y_{c,i,0} \Rightarrow \neg a_j); \forall j \in Opps(c,0,i)$$

For each supporting action a_j ($j \in Supps(c,i)$) with rank j we will introduce an option variable $y_{c,i,j}$ and add the following subformula.

$$F_{c,i,j} = (\neg y_{c,i,j} \vee a_j) \bigwedge_{k \in Opps(c,j,i)} (\neg y_{c,i,j} \vee \neg a_k)$$

These clauses are equivalent to the implications that if the given option is true, then the given supporting action is true and all the opposing actions located between them are false. Finally, for the condition c to hold at time i we need to add the following clause, which enforces at least one option variable to be true.

$$F_{c,i} = (y_{c,i,0} \bigvee_{j \in Supps(c,i)} y_{c,i,j})$$

Using the encoding of the condition requirement it is now easy to encode the dependencies of the actions from the input plan and the goal conditions of the problem. For an action a_i with the rank i we will require that if this action variable is true, then all of its preconditions must be true at time i. For an action a_i the following clauses will enforce, that if the action variable is true, then all the preconditions must hold.

$$F_{a_i} = \bigwedge_{c \in \text{pre}(a_i)} \left((\neg a_i \vee F_{c,i}) \wedge F_{c,i,0} \bigwedge_{j \in Supps(c,i)} F_{c,i,j} \right)$$

We will need to add these clauses for each action in the plan. Let us call these clauses F_A.

$$F_A = \bigwedge_{a_i \in P} F_{a_i}$$

For the goal we will just require all the goal conditions to be true in the end of the plan. Let $n = |P|$, then the goal conditions are encoded using the following clauses.

$$F_G = \bigwedge_{c \in s_G} \left(F_{c,n} \wedge F_{c,n,0} \bigwedge_{j \in Supps(c,n)} F_{c,n,j} \right)$$

The last clause we need to add is related to the redundancy property of the plan. The following clause is satisfied if at least one of the actions in the plan is omitted.

$$F_R = \left(\bigvee_{a_i \in P} \neg a_i \right)$$

Finally, the whole formula $F_{\Pi,P}$ consists of the redundancy clause, the goal clauses, and the action dependency clauses for each action in P.

$$F_{\Pi,P} = F_R \wedge F_G \wedge F_A$$

If the formula is satisfiable, we also want to use its satisfying assignment to construct a new reduced plan. A plan obtained using a truth assignment ϕ will be denoted as P_ϕ. We define P_ϕ to be a subsequence of P such that the i-th action of P is present in P_ϕ if and only if $\phi(a_i) = True$.

Lemma 1. *An assignment ϕ satisfies $F_G \wedge F_A$ if and only if P_ϕ is a valid plan for Π.*

Proof. (sketch) A plan is valid if all the actions in it are applicable when they should be applied and the goal conditions are satisfied in the end. We constructed the clauses of F_G to enforce that at least one option of satisfying each condition will be true. The selected option will then force the required action and none of its opposing actions to be in the plan. Using the same principles, the clauses in F_A guarantee that if an action is present in the plan, then all its preconditions will hold when the action is applied. □

Proposition 1. *The formula $F_{\Pi,P}$ is satisfiable if and only if P is a redundant plan for Π.*

Proof. The clause F_R is satisfied by an assignment ϕ if and only if at least one a_i is false, i.e., not present in P_ϕ which implies $|P_\phi| < |P|$. Using the previous lemma, we can conclude, that the entire formula $F_{\Pi,P} = F_R \wedge F_G \wedge F_A$ is satisfied of and only if there is a valid plan, which can be obtained from P by omitting at least one of its actions. □

Let us conclude this section by computing the following upper bound on the size of the formula $F_{\Pi,P}$.

Proposition 2. *Let p be the maximum number of preconditions of any action in P ,g the number of goal conditions of Π, and $n = |P|$. Then the formula $F_{\Pi,P}$ has at most $n^2p + ng + n$ variables and $n^3p + n^2g + np + g + 1$ clauses, from which $n^3p + n^2g$ are binary clauses.*

Proof. The are n action variables. For each required condition we have at most n option variables, since there are at most n supporting actions for any condition in the plan. We will require at most $(g + np)$ conditions for the g goal conditions and the n actions with at most p preconditions each. Therefore the total number of option variables is $n(np + g)$.

For the encoding of each condition at any time we use at most n options. Each of these options are encoded using n binary clauses (the are at most n opposing actions for any condition). Additionally we have one long clause saying that at least one of the options must be true. We have np required conditions because of the actions and g for the goal conditions. Therefore in total we have at most $(np + g)n^2$ binary clauses and $(np + g)$ longer clauses related to conditions. There is one additional long clause – the redundancy clause. □

Making Plans Perfectly Justified

In this section we describe how to use the encoding described in the previous section to convert any given plan into a perfectly justified plan.

The idea is very similar to the standard planning as SAT approach (Kautz and Selman 1992), where we repeatedly construct formulas and call a SAT solver until we find a plan. In this case we start with a plan, and keep improving it by SAT calls until it is perfectly justified.

```
RedundancyElimination ($\Pi$, $P$)
    $F_{\Pi,P}$ := encodeRedundancy($\Pi$, $P$)
    while isSatisfiable($F_{\Pi,P}$) do
        $\phi$ := getSatAssignment($F_{\Pi,P}$)
        $P := P_\phi$
        $F_{\Pi,P}$ := encodeRedundancy($\Pi$, $P$)
    return $P$
```

Figure 2: Pseudo-code of the SAT based redundancy elimination algorithm. It returns a perfectly justified plan.

The algorithm's pseudo-code is presented in Figure 2. It uses a SAT solver to determine whether a plan is perfectly justified or it can be improved. It can be improved if the formula $F_{\Pi,P}$ is satisfiable. In this case a new plan is constructed using the satisfying assignment. The while loop of the algorithm runs at most $|P|$ times, since every time at least one action is removed from P (in practice several actions are removed in each step).

The algorithm can be implemented in a more efficient manner if we have access to an incremental SAT solver. We need the simplest kind of incrementality – adding clauses.

```
IncrementalRedundancyElimination ($\Pi$, $P$)
    solver = new SatSolver
    solver.addClauses(encodeRedundancy($\Pi$, $P$))
    while solver.isSatisfiable() do
        $\phi$ := solver.getSatAssignment()
        $C := \bigvee\{\neg a_i | a_i \in P_\phi\}$
        solver.addClause($C$)
        foreach $a_i \in P$ do if $\phi(a_i) = False$ then
            solver.addClause($\{\neg a_i\}$)
        $P := P_\phi$
    return $P$
```

Figure 3: Pseudo-code of the incremental SAT based redundancy elimination algorithm.

The incremental algorithm is presented in Figure 3. It adds a new clause C in each iteration of the while loop. This clause is a redundancy clause for the actions remaining in the current plan. It will enforce, that the next satisfying assignment will remove at least one further action. The redundancy clauses added in the previous iterations could be removed, but it is not necessary. The algorithm also adds unit clauses to enforce that the already eliminated actions cannot be reintroduced.

The algorithms presented in this section are guaranteed to produce plans that are perfectly justified, i.e., it is not possible to remove any further actions from them. Nevertheless, it might be the case, that if we had removed a different set of redundant actions from the initial plan, we could have arrived at a shorter perfectly justified plan. In other words, the elimination of redundancy is not confluent. The following example demonstrates this fact.

Example 1. *Let us have a simple path planning scenario on a graph with n vertices $v_1, \ldots, v_n$ and edges (v_i, v_{i+1}) for each $i < n$ and (v_n, v_1) to close the circle. We have one agent traveling on the graph from v_1 to v_n. We have two move actions for each edge (for both directions), in total $2n$ move actions. The optimal plan for the agent is a one action plan $\{move(v_1, v_n)\}$.*

Let us assume that we are given the following plan for redundancy elimination: $\{move(v_1, v_n)$, $move(v_n, v_1)$, $move(v_1, v_2)$, $move(v_2, v_3)$, ..., $move(v_{n-1}, v_n)\}$.

The plan can be made perfectly justified by either removing all but the first action (and obtaining the optimal plan) or by removing the first two actions (ending up a with a plan of n actions). Action elimination would remove the first two actions, for the SAT algorithm we cannot tell which actions would be removed, it depends on the satisfying assignment the SAT solver returns.

The example shows us, that it matters very much in what order we remove the actions and achieving perfect justification does not necessarily mean we did a good job. What we actually want is to remove as many actions as possible. How to do this efficiently is described in the next section.

Maximum Redundancy Elimination

In the section we describe how to do the best possible redundancy elimination for a plan. The problem of *maximum redundancy elimination (MRE)* is to find a subsequence R of redundant actions in a plan P, such that there is no other subsequence R' of redundant actions which is longer than R. A similar notion (minimal reduction) was defined for plans with actions costs (Nakhost and Müller 2010).

The plan resulting from MRE is always perfectly justified, on the other hand a plan might be perfectly justified and at the same time much longer than a plan obtained by MRE (see Example 1).

The solution we propose for MRE is also based on our redundancy encoding, but instead of a SAT solver we will use a partial maximum satisfiability (PMaxSAT) solver. We will construct a PMaxSAT formula, which is very similar to the formula used for redundancy elimination.

A PMaxSAT formula consists of hard and soft clauses. The hard clauses will be the clauses we used for redundancy elimination without the redundancy clause F_R.

$$H_{\Pi,P} = F_G \wedge F_A$$

The soft clauses will be unit clauses containing the negations of the action variables.

$$S_{\Pi,P} = \bigwedge_{a_i \in P} (\neg a_i)$$

The PMaxSAT solver will find an assignment ϕ that satisfies all the hard clauses (which enforces the validity of the plan P_ϕ due to Lemma 1) and satisfies as many soft clauses as possible (which removes as many actions as possible).

```
    MaximumRedundancyEliminaion ($\Pi$, $P$)
MR1     $F$ := encodeMaximumRedundancy($\Pi$, $P$)
MR2     $\phi$ := partialMaxSatSolver($F$)
MR3     return $P_\phi$
```

Figure 4: Pseudo-code of the maximum redundancy elimination algorithm.

The algorithm (Figure 4) is now very simple and straightforward. We just construct the formula and use a PMaxSAT solver to obtain an optimal satisfying assignment. Using this assignment we construct an improved plan the same way as we did in the SAT based redundancy elimination algorithm.

Experimental Evaluation

In this section we present the results of our experimental study regarding elimination of redundant actions from plans. We implemented the Action Elimination algorithm as well as the SAT and MaxSAT based algorithms and used plans obtained by several planners for the problems of the International Planning Competition (Coles et al. 2012).

Experimental Settings

Since, our tools take input in the SAS+ format, we used Helmert's translation tool, which is a part of the Fast Downward planning system (Helmert 2006), to translate the IPC benchmark problems that are provided in PDDL.

To obtain the initial plans, we used the following state-of-the-art planners: FastDownward (Helmert 2006), Metric FF (Hoffmann 2003), and Madagascar (Rintanen 2013). Each of these planners was configured to find plans as fast as possible and ignore plan quality.

We tested four redundancy elimination methods:

- *Action Elimination (AE)* is our own Java implementation of the Action Elimination algorithm as displayed in Figure 1.
- *Action Elimination + SAT (AE+S)* is an algorithm that first runs Action Elimination on the initial plan and incremental SAT reduction (see Figure 3) on the result. We used the incremental Java SAT solver Sat4j (Berre and Parrain 2010).
- *SAT Reduction (SAT)* is using the incremental SAT reduction directly without using Action Elimination for preprocessing (same as AE+S without AE).
- *Maximum Elimination (MAX)* is a Partial MaxSAT reduction based algorithm displayed in Figure 4. We implemented the translation in Java and used the QMaxSAT (Koshimura et al. 2012) state-of-the-art MaxSAT solver written in C++ to solve the instances.

For each of these methods we measured the total runtime and the total number of removed redundant actions for each domain and planner.

Table 1: Experimental results on the plans for the IPC 2011 domains found by the planners Fast Downward, Metric FF, and Madagascar. The planners were run with a time limit of 10 minutes. The column "#Plans" contains the number of plans found and "Length" represents the sum of their lengths. By Δ_{ALG} and T_{ALG} we mean the total number of removed redundant actions and the time in seconds it took for all plans for a given algorithm ALG. The algorithms are Action Elimination (AE), Action Elimination followed by SAT reduction (AE+S), SAT reduction on the original plan (SAT), and maximum elimination using a MaxSat solver (MAX).

	Domain	#Plans	Length	Δ_{AE}	T_{AE}	Δ_{AE+S}	T_{AE+S}	Δ_{SAT}	T_{SAT}	Δ_{MAX}	T_{MAX}
Metric FF	elevators	20	4273	79	0,81	79	2,31	79	3,14	79	0,17
	floortile	2	81	10	0,02	10	0,08	10	0,10	10	0,00
	nomystery	5	107	0	0,01	0	0,16	0	0,17	0	0,00
	parking	18	1546	124	0,18	124	1,13	124	1,60	124	0,03
	pegsol	20	637	0	0,10	0	1,10	0	1,16	0	0,02
	scanalyzer	18	571	30	0,06	30	0,78	30	0,88	30	0,01
	sokoban	13	2504	6	0,39	6	2,34	6	2,52	6	0,36
	tidybot	17	1136	144	0,17	144	0,92	144	1,66	144	0,04
	transport	6	1329	164	0,34	164	0,82	164	2,15	165	0,25
	visitall	3	1137	166	0,14	166	0,47	166	0,97	172	0,08
	woodworking	19	1471	22	0,37	22	1,14	22	1,28	22	0,02
Fast Downward	barman	20	3749	528	0,52	582	3,44	596	7,18	629	0,44
	elevators	20	4625	94	0,84	94	2,41	94	3,45	94	0,19
	floortile	5	234	22	0,06	22	0,20	22	0,27	22	0,00
	nomystery	13	451	0	0,05	0	0,47	0	0,48	0	0,00
	parking	20	1494	4	0,17	4	1,21	4	1,26	4	0,03
	pegsol	20	644	0	0,11	0	1,11	0	1,18	0	0,02
	scanalyzer	20	823	26	0,10	26	1,16	26	1,33	26	0,03
	sokoban	17	5094	244	0,62	458	5,25	458	8,39	460	1,84
	tidybot	16	1046	64	0,14	64	0,91	64	1,28	64	0,03
	transport	17	4059	289	0,65	289	1,64	289	2,93	290	0,20
	visitall	20	28776	122	3,66	122	9,47	122	12,89	122	7,77
	woodworking	20	1605	27	0,41	27	1,16	27	1,33	30	0,03
Madagascar	barman	8	1785	303	0,25	303	1,59	303	3,53	318	0,30
	elevators	20	11122	2848	1,46	3017	4,13	3021	17,62	3138	2,03
	floortile	20	1722	30	0,39	30	1,05	30	1,32	30	0,03
	nomystery	15	480	0	0,06	0	0,51	0	0,53	0	0,01
	parking	18	1663	152	0,20	152	1,17	152	1,78	152	0,03
	pegsol	19	603	0	0,09	0	1,06	0	1,10	0	0,01
	scanalyzer	18	1417	232	0,24	232	0,88	232	1,61	236	0,05
	sokoban	1	121	22	0,02	22	0,13	22	0,29	22	0,01
	tidybot	16	1224	348	0,16	348	0,84	348	2,13	350	0,08
	transport	4	1446	508	0,20	539	0,40	532	1,65	553	0,16
	woodworking	20	1325	0	0,31	0	1,11	0	1,21	0	0,01

All the experiments were run on a computer with Intel Core i7 960 CPU @ 3.20 GHz processor and 24 GB of memory. The planners had a time limit of 10 minutes to find the initial plans. The benchmark problems are taken from the satisficing track of IPC 2011 (Coles et al. 2012).

Experimental Results

The results of our experiments are displayed in Table 1. We can immediately notice that the runtime of all of our methods is very low. None of the methods takes more than one second on average for any of the plans. Note, that the runtime of the MAX method is often the smallest contrary to the fact, that it is the only one which guarantees eliminating the maximum number of redundant actions (AE+S and SAT only guarantee perfect justification).

Looking at the number of removed actions in Table 1 we can make several interesting observations. For example, in the nomystery and pegsol domains no redundant actions were found in plans obtained by any planner and also Madagascar's plans for the woodworking domain were always perfectly justified. In the most cases the AE algorithm provides perfectly justified plans (this is when the values of Δ_{AE} and Δ_{AE+S} are equal). The SAT method performs better than AE+S on barman for Fast Downward and elevators for Madagascar, but removes less actions on transport for Madagascar. Although both methods reach perfect justification, the results are different since removing redundant actions is not confluent (see example 1). As expected, the MAX method removes the highest (or equal) number of actions in each case. It is strictly dominant for 11 planner domain combinations. Considering the good runtime performance of this method we can conclude, that MAX is the best way of eliminating redundant actions.

Conclusions

In this paper, we have introduced a SAT encoding for the problem of detecting redundant actions in plans and used it to build two algorithms for plan optimization. One is based on SAT solving and the other on partial MaxSAT solving. Contrary to existing algorithms, both of our algorithms guarantee, that they output a plan with no redundant actions. Additionally, the MaxSAT based algorithm always eliminates a maximum set of redundant actions. According to our experiments done on IPC benchmark problems with plans obtained by state-of-the-art planners, our newly proposed algorithms perform very well in practice.

Acknowledgments The research is supported by the Czech Science Foundation under the contract P103/10/1287 and by the Grant Agency of Charles University under contract no. 600112. This research was also supported by the SVV project number 260 104.

References

Bäckström, C., and Nebel, B. 1995. Complexity results for sas+ planning. *Computational Intelligence* 11:625–656.

Balyo, T.; Barták, R.; and Surynek, P. 2012. Shortening plans by local re-planning. In *Proceedings of ICTAI*, 1022–1028.

Berre, D. L., and Parrain, A. 2010. The sat4j library, release 2.2. *JSAT* 7(2-3):59–64.

Chrpa, L.; McCluskey, T. L.; and Osborne, H. 2012a. Determining redundant actions in sequential plans. In *Proceedings of ICTAI*, 484–491.

Chrpa, L.; McCluskey, T. L.; and Osborne, H. 2012b. Optimizing plans through analysis of action dependencies and independencies. In *Proceedings of ICAPS*, 338–342.

Coles, A. J.; Coles, A.; Olaya, A. G.; Celorrio, S. J.; López, C. L.; Sanner, S.; and Yoon, S. 2012. A survey of the seventh international planning competition. *AI Magazine* 33(1).

Estrem, S. J., and Krebsbach, K. D. 2012. Airs: Anytime iterative refinement of a solution. In *Proceedings of FLAIRS*, 26–31.

Fikes, R., and Nilsson, N. J. 1971. Strips: A new approach to the application of theorem proving to problem solving. *Artif. Intell.* 2(3/4):189–208.

Fink, E., and Yang, Q. 1992. Formalizing plan justifications. In *In Proceedings of the Ninth Conference of the Canadian Society for Computational Studies of Intelligence*, 9–14.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research (JAIR)* 20:239 – 290.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research (JAIR)* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hoffmann, J. 2003. The Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal Artificial Intelligence Research (JAIR)* 20:291–341.

Kautz, H. A., and Selman, B. 1992. Planning as satisfiability. In *Proceedings of ECAI*, 359–363.

Koshimura, M.; Zhang, T.; Fujita, H.; and Hasegawa, R. 2012. Qmaxsat: A partial max-sat solver. *JSAT* 8(1/2):95–100.

Nakhost, H., and Müller, M. 2010. Action elimination and plan neighborhood graph search: Two algorithms for plan improvement. In *Proceedings of ICAPS*, 121–128.

Rintanen, J. 2013. Planning as satisfiability: state of the art. http://users.cecs.anu.edu.au/ jussi/satplan.html.

Siddiqui, F. H., and Haslum, P. 2013. Plan quality optimisation via block decomposition. In *Proceedings of IJCAI*.

Westerberg, C. H., and Levine, J. 2001. Optimising plans using genetic programming. In *Proceedings of ECP*, 423–428.

Planning with Preferences by Compiling Soft Always Goals into STRIPS with Action Costs

Luca Ceriani and **Alfonso Emilio Gerevini***
Department of Information Engineering, University of Brescia, Italy
luca.ceriani@unibs.it, alfonso.gerevini@unibs.it (*corresponding author)

Abstract

We address the problem of planning with preferences in propositional domains focusing on soft always goals (or constraints), a basic class of temporally extended goals useful to express maintenance goals, safety conditions, or other desired conditions that affect the quality of the solution plans. Inspired by previous work of Keyder and Geffner on compiling soft goals, we propose a compilation scheme for translating a STRIPS problem with soft always constraints into an equivalent STRIPS problem with action costs. This problem transformation allows to solve propositional planning problems enriched with soft always constraints (possibly extendable also with soft goals) using several existing powerful planners, that need to support only STRIPS and action costs. An experimental analysis presented in the paper demonstrates that solving the compiled problems using existing STRIPS planners can be significantly more effective than solving the original uncompiled problems using planners supporting soft always constraints, such as HPlan-P and MIPS-XXL, in terms of satisfied always constraints and scalability.

Introduction

Planning with preferences, also called "over-subscription planning" (e.g., (van den Briel et al. 2004; Do and Kambhampati 2004; Smith 2004)), is a recent important area of automated planning concerning the generation of plans for problems involving soft goals or state-trajectory constraints, that it is desired a solution plan satisfies, but that do not have to be necessarily satisfied. The quality of a solution plan for these problems depends on the soft goals and constraints that are satisfied by the plan.

In the standard planning language PDDL3 (Gerevini et al. 2009), state-trajectory constraints are particular linear temporal logic formulae expressing temporally extended goals (Bacchus and Kabanza 1998; Baier and McIlraith 2006) restricting the intermediate states reachable by the valid plans. A basic class of state-trajectory constraints consists of *always constraints* (or *always goals*) expressing that a certain condition must hold in *every* state reached by a valid plan.[1] Adding always constraints to the standard "achievement" goals of a problem (requiring the goals to hold at the end of the plan) is useful to express safety or maintenance conditions (e.g., (Bacchus and Kabanza 1998; Weld and Etzioni 1994)) as well as desired plan properties. Examples of such conditions are "whenever a building surveillance robot is outside a room, all the room doors should be closed" or, in a logistics domain, "whenever a truck is at a location, all goods the truck has to load from that location have already been transported there" (for additional examples see, e.g., (Bacchus and Kabanza 1998; Gerevini et al. 2009; Weld and Etzioni 1994)).

[1]In the literature, always constraints/goals are also called "safety goals", "maintenance goals", "state invariants" or "state constraints".

In propositional planning, only few planners supporting soft always constraints have been developed. The most prominent of them is planner HPlan-P (Baier and McIlraith 2008), which won the "qualitative preference" track of the 5th International Planning Competition (IPC), focusing on propositional planning with soft goals and state-trajectory constraints expressed through PDDL3.

STRIPS extended with action costs is a simple language for propositional planning that many existing powerful planners support, e.g., LPG (Gerevini, Saetti, and Serina 2003), Metric-FF (Hoffmann 2003), Fast Downward (Helmert 2006), and LAMA (Richter and Westphal 2010). Handling action costs is a practically important, basic functionality that was required in the satisficing tracks of the last two IPCs. Keyder and Geffner (Keyder and Geffner 2009) showed that any STRIPS problem with soft (achievement) goals can be efficiently compiled into an equivalent STRIPS problem with action costs.

Inspired by Keyder and Geffner's work, we propose a compilation scheme for translating a STRIPS problem with soft always constraints into an equivalent STRIPS problem with action costs. This problem transformation allows to solve propositional planning problems enriched with soft always constraints (possibly extendable also with soft goals) using several existing powerful planners.

A preliminary experimental analysis presented in this paper shows that solving the compiled problems by LPG, Metric-FF or LAMA can be significantly more effective than solving the original uncompiled problems using planners supporting soft always constraints, in terms of plan quality (i.e., satisfied always constraints) and scalability.

Other approaches to compiling (soft) always constraints have been proposed. The techniques proposed in (Edelkamp 2006; Edelkamp, Jabbar, and Nazih 2006; Baier and McIlraith 2008; Gerevini et al. 2009) use a language richer than STRIPS, requiring conditional effects and numerical fluents. Their compilation schemas are very different and based on representing state-trajectory constraints through automata encoded in the compiled problem. Two of these schemas are implemented in planners HPlan-P and MIPS-XXL (Edelkamp and Jabbar 2008). In the experimental analysis of this work, we show that these planners perform generally less efficiently than the compared STRIPS planners.

Another recent planner supporting soft always constraints is LPRPG-P (Coles and Coles 2011), which is based on a hybrid heuristic using relaxed planning graphs and linear programming. Differently from this system, the investigated compilation approach has the advantage of allowing many planners (using any heuristic for classical planning) to support always constraints, possibly in addition to compiled soft goals.

The remainder of the paper is organised as follows. After some background and definitions about STRIPS with action costs and always constraints, we describe our compilation method and its properties. Then we present some experimental results and finally we give our conclusions.

STRIPS Planning with Action Costs and Soft Always Constraints

This section describes the STRIPS model of planning with action costs, introduces its extension with soft always constraints, and gives some auxiliary definitions used in the rest of the paper.

A STRIPS problem is a tuple $\langle F, I, O, G\rangle$ where F is a set of fluents, $I \subseteq F$ and $G \subseteq F$ are the initial state and goal situation, and O is a set of actions or operators defined over F as follows:

Definition 1. *Given a set of fluents F, an operator $o \in O$ is a pair $\langle Prec(o), Eff(o)\rangle$, where $Prec(o)$ is a sets of atomic formulae over F and $Eff(o)$ is a set of literals over F. $Eff(o)^+$ denotes the set of positive literals in $Eff(o)$; $Eff(o)^-$ denotes the set of negative literals in $Eff(o)$.*

An action sequence $\pi = \langle a_0, \ldots, a_n\rangle$ is applicable in a planning problem P if the actions a_i, $i \in \{0 \ldots n\}$, are all in O and there exists a sequence of states $\langle s_0, \ldots, s_{n+1}\rangle$ such that $s_0 = I$, $Prec(a_i) \subseteq s_i$ and, $s_{i+1} = s_i \cup Eff(o)^+ \setminus Eff(o)^-$ for $i \in \{0 \ldots n\}$. The applicable action sequence π achieves a fluent g, if $g \in s_{n+1}$ and is a (valid) plan for P if it achieves each goal $g \in G$, which we indicate with $\pi \models G$.

Definition 2. *A STRIPS problem with action costs (AC) is a tuple $P_c = \langle F, I, O, G, c\rangle$, where $P = \langle F, I, O, G\rangle$ is a STRIPS problem and c is a function $c : O \rightarrow \mathbb{R}_0^+$, mapping each operator $o \in O$ to a non-negative real number.*

The cost of a plan π for a problem P_c is given by:

$$c(\pi) = \sum_{i=0}^{|\pi|} c(a_i)$$

where $c(a_i)$ denotes the cost of the ith action a_i in π and $|\pi|$ is the length of π.

An *always constraint* A, also called *always goal*, is a temporally extended goal expressing a condition that must hold in the problem initial state and in every state of the sequence of states produced by applying a valid plan π (Bacchus and Kabanza 1998; Gerevini et al. 2009). If this is the case, we say that π satisfies A, and we indicate it with $\pi \models_{\mathsf{a}} A$. Without loss of generality, in the following we will assume that the condition of an always constraint is expressed in CNF form. For brevity, (soft) always constraints will be abbreviated with *constraint* or SAG (soft always goal).

Definition 3. *A STRIPS problem with action costs (AC) and soft always goals (SAG) is a tuple $P_u = \langle F, I, O, G, AG, c, u\rangle$ where:*

- *$\langle F, I, O, G, c\rangle$ is a STRIPS problem with action costs;*
- *$AG = \{A_1, \ldots, A_k \mid A_i$ CNF formula over $F, i = 1 \ldots n\}$*
- *$u : AG \rightarrow \mathbb{R}_0^+$, is an utility function mapping each always constraint $A_i \in AG$ to its utility value over R_o^+.*

In the following, STRIPS with AC is denoted by STRIPS+, and STRIPS with SAG by STRIPS+SAG. The (soft) always constraints A in a STRIPS+SAG problem is denoted with $A = ag_1 \wedge \ldots \wedge ag_n$, where each ag_i is a *clause* of A formed by literals over F.

Definition 4. *Let P_u be a STRIPS+SAG problem with a set AG of always constraints. The utility of a plan π solving P_u is the difference between the total utility obtained by the plan and its cost:*

$$u(\pi) = \left(\sum_{A_i \in AG : \pi \models_{\mathsf{a}} A_i} u(A_i) \right) - c(\pi).$$

The definition of plan utility for STRIPS+SAG is similar to the one given for STRIPS+ with soft goals by Keyder and Geffner (2009). A plan π with utility $u(\pi)$ for a STRIPS+SAG problem is optimal when no other plan π' has utility $u(\pi') > u(\pi)$. The last two IPCs featured tracks in which the objective was to find optimal plans with respect to the plan "net benefit" captured by the equation of Definition 4 applied to the utility of soft (end state) goals instead of soft always constraints.

Definition 5. *Given a clause $ag = l_1 \vee \ldots \vee l_n$, the set $L(ag) = \{l_i \mid l_i$ is a disjunct of $ag\}$ is an equivalent set-based definition for ag and $\overline{L}(ag) = \{\neg l_i \mid l_i$ is a disjunct of $ag\}$ is the literal-complement set of $L(ag)$.*

Definition 6. *Given an operator $o \in O$ of a STRIPS+SAG problem, $Z(o)$ is the set of literals defined as:*

$$Z(o) = (Prec(o) \setminus \{p \mid \neg p \in Eff(o)^-\}) \cup Eff(o)^+ \cup Eff(o)^-.$$

Note that the literals in $Z(o)$ hold in any reachable state resulting from the execution of operator o.

In a STRIPS+SAG problem, we distinguish three types of operators that are defined as follows.

Definition 7. *Given an operator o and a constraint A of a STRIPS+SAG problem, o is a* **violation** *of A if there exists a clause ag_i of A such that:*

$$\overline{L}(ag_i) \subseteq Z(o) \land \overline{L}(ag_i) \nsubseteq \textit{Prec}(o).$$

If an operator violates a constraint, the constraint is false in any state resulting from the application of the operator. The set of constraints in a STRIPS+SAG problem that are violated by an operator o is denoted with $V(o)$.

Definition 8. *Given an operator o and a constraint A of a STRIPS+SAG problem, o is a* **threat** *for A if it is not a violation and there exists a clause ag_i of A, such that:*

$$\overline{L}(ag_i) \cap Z(o) \neq \emptyset \land L(ag_i) \cap Z(o) = \emptyset \land \overline{L}(ag_i) \nsubseteq \textit{Prec}(o).$$

A clause ag_i of A satisfying the condition of Definition 8 is a *threatened clause* of A. A threatened constraint (clause) may be falsified by an operator depending on the state where the operator is applied. The set of constraint threatened by an operator o is denoted with $T(o)$; the set of constraint clauses threatened by o is denoted with $T_A(o)$. Note that the last conjunct ($\overline{L}(ag_i) \nsubseteq \textit{Prec}(o)$) in the conditions of Definitions 7-8 avoid that an operator o is considered a violation/threat when its preconditions require ag_i to be *already* violated in the state where it is applied. This can be useful to optimize the compilation procedure, because it is not necessary to treat o as a violation/threat if in the conditions of Definitions 7-8 the first two conjuncts hold and the third does not.

Definition 9. *Given an operator o and a constraint A of a STRIPS+SAG problem, o is a* **safe** *for A if for all clauses ag_i of A the following condition holds:*

$$L(ag_i) \cap Z(o) \neq \emptyset \lor \overline{L}(ag_i) \cap Z(o) = \emptyset.$$

If an operator o of a STRIPS+SAG problem P is safe for every always constraint of P, we say that the operator is *safe for P*, and we write this property with $\textit{Safe}(o, P)$.

Compilation into STRIPS+

Given a STRIPS+SAG problem P, an equivalent STRIPS+ problem P' can be derived using a transformation method inspired by (Keyder and Geffner 2009), with significant differences in the compilation of the operators. For the sake of simplicity we will assume that every always constraint of P is satisfied in the problem initial state.[2]

The operator transformation schema mentioned in the following definition is described after the definition.

Definition 10. *Given a STRIPS+SAG problem $P = \langle F, I, O, G, AG, c, u\rangle$, the compiled STRIPS+ problem of P is $P' = \langle F', I', O', G', c'\rangle$, with:*

- $F' = F \cup AV \cup D \cup C' \cup \overline{C'} \cup \{\textit{normal-mode}, \textit{end-mode}, \textit{pause}\}$;
- $I' = I \cup \overline{C'} \cup \{\textit{normal-mode}\}$;
- $G' = G \cup C'$;
- $O' = \{\textit{collect}(A), \textit{forgo}(A) \mid A \in AG\} \cup \{\textit{end}\} \cup O_{comp}$;
- $c'(o) = \begin{cases} u(A) & \text{if } o = \textit{forgo}(A) \\ c(o) & \text{if } o \in O'' \\ c''(o) & \text{if } o \in O_V \cup O_T \\ 0 & \text{otherwise} \end{cases}$

where:

- $AV = \bigcup_{i=1}^{k} \{A_i\textit{-violated}\}, k = |AG|$;
- $D = \bigcup_{i=1}^{k} \{A_i\textit{-done}_{o_1}, \ldots, A_i\textit{-done}_{o_n}\}$, $k = |AG|$ and n is the number of operators threatening or violating A_i;
- $C' = \{A' \mid A \in AG\}$;
- $\overline{C'} = \{\overline{A'} \mid A' \in C'\}$;
- $\textit{collect}(A) = \langle\{\textit{end-mode}, \neg A\textit{-violated}, \overline{A'}\}, \{A', \neg\overline{A'}\}\rangle$;
- $\textit{forgo}(A) = \langle\{\textit{end-mode}, A\textit{-violated}, \overline{A'}\}, \{A', \neg\overline{A'}\}\rangle$;
- $\textit{end} = \langle\{\textit{normal-mode}\}, \{\textit{end-mode}, \neg\textit{normal-mode}\}\rangle$;
- $O'' = \{\langle Pre(o) \cup \{\textit{normal-mode}, \neg\textit{pause}\}, \textit{Eff}(o)\rangle \mid o \in O \textit{ and } \textit{Safe}(o, P)\}$;
- $O_{comp} = O'' \cup O_V \cup O_T$
- O_V and O_T are the sets of operators generated by the operator transformation schema applied to the operators of P that violate and threaten, respectively, a constraint of P.[3]
- $c''(o)$ is the cost of an operator o in $O_V \cup O_T$ as defined in the operator transformation schema.

As in (Keyder and Geffner 2009), for each $A \in AG$ the transformation of P into P' adds a dummy hard goal A' to P' which can be achieved in two ways: with action *collect(A)*, that has cost 0 but requires A to be satisfied, i.e. *A-violated* is false in the goal-state, or with action *forgo(A)*, that has cost equal to the utility of A and can be performed when A is false or, equivalently, when *A-violated* is true. Moreover, for each constraint exactly one of {*collect(A), forgo(A)*} can appear in the plan, as both delete their shared precondition $\overline{A'}$, which no action makes true.

The remainder of this section presents the transformation schema for the operators that threaten or violate a soft always constraint. (Note that safe operators are transformed as described in Definition 10, forming operator set O''.) Threat operators are not trivial to compile because they may violate a constraint in different ways, depending on which planning state they are applied to. Each operator o such

[2]Let X be the set of constraints that are false in the initial state s. If X is not empty, it can be handled, e.g., as follows: (a) the original operator set is extended with a new operator o_{init} such that $\textit{Prec}(o_{init}) = \{\textit{FalseX}\}$, $\textit{Eff}(o_{init}) = s$ and $c(o_{init})$ is the sum of the utilities of the constraints in X; (b) s is then revised to contain only *FalseX*; (c) the constraints in X are not processed by the operator compilation scheme. Note that any valid plan is forced to contain o_{init} as the first action.

[3]If an operator is both a violation and a threat, the generated compiled operators are in O_T only.

that $T(o) \neq \emptyset$ is compiled into a set of new operators (one for each threatened constraint). More specifically, let $T(o) = \{A_1, \dots, A_m\}$, be the set of constraints threatened by o. Operator o is compiled into a sequence of $2m$ operators $O_T(o) = \{o_{A_1}, \overline{o}_{A_1}, \dots, o_{A_m}, \overline{o}_{A_m}\}$ such that in any state s where o can be applied violating a set of constraints $T(o)_s \subseteq T(o)$, the sequence $\omega_T(o)$ of m operators in $O_T(o)$ defined as follows can be applied:

$$\omega_{T(o)} = \langle o'_{A_1}, \dots, o'_{A_m} \rangle \qquad (1)$$

where $o'_{A_i} = \overline{o}_{A_i}$ if o violates A_i when applied in s, $o'_{A_i} = o_{A_i}$ if o does not violate A_i when applied in s, and $\overline{o}_{A_i}$ and o_{A_i} are mutually exclusive, for $i \in \{1 \dots m\}$. The cost $c''(o^t)$ of an operator $o^t \in O_T(o)$ is defined as follows:

$$c''(o^t) = \begin{cases} c(o) & \text{if } o \in \{o_{A_1}, \overline{o}_{A_1}\} \\ 0 & \text{otherwise} \end{cases}$$

i.e., exactly one of the operators in ω_T (the first) cost equal to the cost of the original domain operator compiled into $O_T(o)$, while the others have cost zero.

Before defining the operators forming $O_{T(o)}$ (and $\omega_{T(o)}$) for each threatening operator o, we introduce some notation useful to simplify the presentation:

Definition 11. *Given an operator o and a constraint clause ag:*

- $NA(o)_{ag} = \{l \in L(ag) \mid \neg l \in (\mathit{Eff}(o)^+ \cup \mathit{Eff}(o)^-)\}$ *is the set of literals in* $L(ag)$ *falsified by the effects of o;*
- $AA(o)_{ag} = L(ag) \setminus NA(o)_{ag}$ *is the set of literals in* $L(ag)$ *not falsified by the application of o and* $\overline{AA}(o)_{ag} = \{\neg l \mid l \in AA(o)_{ag}\}$ *is the literal-complement set of* $AA(o)_{ag}$.

For each operator o threatening a set of constraints $T(o) = \{A_1, \dots, A_m\}$, the operators of $O_{T(o)}$ are defined as follows.

- o_{A_1}:

$$\mathit{Prec}(o_{A_1}) = \mathit{Prec}(o) \cup \{\neg \mathit{pause}\} \cup \{ \bigcup_{ag \in T_A(o)} (l_1 \vee \dots \vee l_p) \mid \{l_1, \dots, l_p\} = AA_1(o)_{ag}\}$$

$$\mathit{Eff}(o_{A_1}) = \{A_1\text{-}\mathit{done}_o, \mathit{pause}\}$$

- $\overline{o}_{A_1}$:

$$\mathit{Prec}(\overline{o}_{A_1}) = \mathit{Prec}(o) \cup \{\neg \mathit{pause}\} \cup \{ \bigvee_{ag \in T_A(o)} (l_1 \wedge \dots \wedge l_q) \mid \{l_1, \dots, l_q\} = \overline{AA}_1(o)_{ag}\}$$

$$\mathit{Eff}(\overline{o}_{A_1}) = \{A_1\text{-}\mathit{done}_o, \mathit{pause}, A_1\text{-}\mathit{violated}\}$$

$\forall k \in [2 \dots m-1]$:

- o_{A_k}:

$$\mathit{Prec}(o_{A_k}) = \{A_{k-1}\text{-}\mathit{done}_o, \mathit{pause}\} \cup \{ \bigcup_{ag \in T_A(o)} (l_1 \vee \dots \vee l_p) \mid \{l_1, \dots, l_p\} = AA_k(o)_{ag}\}$$

$$\mathit{Eff}(o_{A_k}) = \{A_k\text{-}\mathit{done}_o, \neg A_{k-1}\text{-}\mathit{done}_o\}$$

- $\overline{o}_{A_k}$:

$$\mathit{Prec}(\overline{o}_{A_k}) = \{A_{k-1}\text{-}\mathit{done}_o, \mathit{pause}\} \cup \{ \bigvee_{ag \in T_A(o)} (l_1, \wedge \dots \wedge l_q) \mid \{l_1, \dots, l_q\} = \overline{AA}_k(o)_{ag}\}$$

$$\mathit{Eff}(\overline{o}_{A_k}) = \{A_k\text{-}\mathit{done}_o, A_k\text{-}\mathit{violated}, \neg A_{k-1}\text{-}\mathit{done}_o\}$$

if $k = m$:

- o_{A_m}:

$$\mathit{Prec}(o_{A_m}) = \{ \bigcup_{ag \in T_A(o)} (l_1 \vee \dots \vee l_p) \mid \{l_1, \dots, l_p\} = AA_m(o)_{ag}\} \cup \{A_{m-1}\text{-}\mathit{done}_o, \mathit{pause}\}$$

$$\mathit{Eff}(o_{A_m}) = \mathit{Eff}(o) \cup \{\neg A_{m-1}\text{-}\mathit{done}_o, \neg \mathit{pause}\}$$

- $\overline{o}_{A_m}$:

$$\mathit{Prec}(\overline{o}_{A_m}) = \{ \bigvee_{ag \in T_A(o)} (l_1, \wedge \dots \wedge l_q) \mid \{l_1, \dots, l_q\} = \overline{AA}_m(o)_{ag}\} \cup \{A_{m-1}\text{-}\mathit{done}_o, \mathit{pause}\}$$

$$\mathit{Eff}(\overline{o}_{A_m}) = \mathit{Eff}(o) \cup \{\neg A_{m-1}\text{-}\mathit{done}_o, A_m\text{-}\mathit{violated}, \neg \mathit{pause}\}.$$

The operators of $T(o)$ can be applied only in a sequence $\omega_{T(o)}$ defined above. The preconditions of any operator $\overline{o}_{A_i}$ in $\omega_{T(o)}$ require $\overline{AA}_i(o)_{ag}$ to hold in the state where $\omega_{T(o)}$ is applied for at least one $ag \in T_{A_i}(o)$. If this happens, then $\overline{o}_{A_i} \in \omega_{T(o)}$ and constraint A_i is violated by $\omega_{T(o)}$. Predicate A_i-*violated* is made true by $\overline{o}_{A_i}$ and is never falsified. After the *end* action is applied, A_i-*violated* serves as a precondition of the operator *forgo*(A_i) that has cost equal to the utility of A_i. The A_i-*done*$_o$ predicates force the planner to strictly follow the order in $\omega_{T(o)}$, avoiding repetitions. Once the planner starts sequence $\omega_{T(o)}$ for some o, no other operator $o' \notin O_{T(o)}$ is enabled before the application of $\omega_{T(o)}$ is completed. Predicate *pause* serves to this purpose, and only the last action in the sequence can falsify it.

Each domain operator o violating a constraint ($V(o) \neq \emptyset$) must be compiled as well. However, the compilation schema for o is simpler than the one shown above for a threatening operator. It is sufficient to add to the effects of o an A-*violated* predicate for each constraint $A \in V(o)$. If such a modified operator is applied, all the constraints it violates are falsified (making their *violated*-predicates true) and corresponding *forgo* actions must later be selected by the planner for every plan achieving the goals. Notice that an operator o can simultaneously be a violation and a threat of different sets of constraints. If an operator o threatening a set of constraints also violates a constraint A, effect A-*violated* is added to the first pair of operators $(o_{A_1}, \overline{o}_{A_1})$ in $O_{T(A)}$. The cost $c''(o^v)$ of an operator o^v derived by compiling an original operator o that violates a constraint (and does not threat any other constraint) is the the same cost $c(o)$ of o.

In the compilation of a threatening operator o, its preconditions can be extended by disjunctions of literals. Such disjunctions can be simplified by performing (unit) resolution over the augmented set of preconditions, discharging the operator if the empty close is generated. Moreover, since negation and disjunction are not allowed in STRIPS, a further standard "ADL-to-STRIPS" translation (Gazen and Knoblock 1997) can be executed to complete the procedure and generate a STRIPS+ problem.

Compilation Properties

Let P' be the STRIPS+ problem derived by compiling a STRIPS+SAG problem P. All plans π' solving P' have the form $\pi' = \langle \pi'', end, \pi''' \rangle$, where the prefix π'' is formed by operators of P, ω-sequences of compiled operators as defined in (1), and compiled constraint violating operators of P. Each sequence $\omega_{T(o)}$ in π' corresponds to the application of an operator $o \in O$ of P in an equivalent solution plan π for P . The two problems P and P' are equivalent in the sense that there is a correspondence between the plans for P and P', and corresponding plans are ranked in the same way by the utility functions of P and P'. More formally, the following propositions about P and P' can be proved:

Proposition 1 (Correspondence between plans). *For an applicable action sequence π in P, let $\pi' = \langle \pi'', end, \pi''' \rangle$ be any sequence of actions of P' derived as follows: the prefix-plan π'' denotes any sequence of operators of P' obtained by replacing each operator o of π threatening the constraints in $T(o)$ by a sequence $\omega_{T(o)}$ of $|T(o)|$ operators in $O_{T(o)}$; the suffix-plan π''' denotes any permutation of collect(A) and forgo(A) operators, for every always constraint A of P, when $\pi \models A$ and $\pi \not\models A$ respectively. Then:*

$$\pi \textit{ is a plan for P} \iff \pi' \textit{ is a plan for } P'.$$

Proof. (Sketch) The proof is similar to the one for Proposition 1 in (Keyder and Geffner 2009). The main difference concerns the treatment of the compiled operator sequences in π'', which are not present in the transformation of Keyder and Geffner to compile soft goals into STRIPS+. To handle them, it suffices to show that (i) the state resulting from the application of any sequence $\omega_{T(o)}$ to the state where it is applied in π'' is the same as the one that would be obtained by applying o to s, with the only exception of the auxiliary *A-done*$_o$, *A-violated* and *pause* predicates, and (ii) π''' is analogous to the plan extension π'' in the structure of the plans solving the compiled problems defined by the problem transformation for STRIPS+ with soft goals described in (Keyder and Geffner 2009). □

Proposition 2 (Correspondence between utilities and costs). *Let π_1 and π_2 be two plans for P, and let π'_1 and π'_2 be transformations of π_1 and π_2, respectively, derived as described in Proposition 1. Then:*

$$u(\pi_1) > u(\pi_2) \iff c(\pi'_1) < c(\pi'_2).$$

Proof. The proof is the same as the one for Proposition 2 in (Keyder and Geffner 2009). □

Proposition 3 (Equivalence). *Let π be a plan for P, and π' be a plan for P' derived by transforming π as described in Proposition 1. Then:*

$$\pi \textit{is an optimal plan for P} \iff \pi' \textit{is an optimal plan for} P'.$$

Proof. Direct from Propositions 1–2. □

Concerning the complexity of the proposed compilation, we have the following bound on the length of the solutions of the compiled problems:

Proposition 4. *For every solution plan π of a STRIPS+SAG problem P with n soft always constraints, there exists a solution π' of the compiled problem P' of P such that $|\pi'| = \sum_{j=1}^{|\pi|}(max(1, k_i)) + n + 1 \leq |\pi| * n + n + 1 = O(|\pi| * n)$, where $k_i \leq n$ is the number of constraints threatened by the i-th operator of π.*

Proof. (Sketch) Plan π' has three parts: π'', action end, π'''. The first, π'', is formed by the safe operators of π and an $\omega_{T(o)}$ sequence for each threatening operator o in π. The length of $\omega_{T(o)}$ is the number of constraints violated by o. The third part, π''', is a sequence of n *collect* and *forgo* actions. □

Experimental Analysis

In order to test the effectiveness of the approach, we implemented the proposed compilation scheme and compared the performance of some STRIPS planners and two planners supporting SAGs, HPlan-P and MIPS-XXL (Edelkamp, Jabbar, and Nazih 2006; Edelkamp and Jabbar 2008) (in the following abbreviated with MIPS). We considered the five domains, and corresponding test problems involving always constraints, of the "qualitative preference" track of IPC5 (Gerevini et al. 2009). Here we focus our presentation on three of them, `Rovers`, `TPP` and `Trucks` (for the other two, `Openstack` and `Storage`, HPlan-P and the compared STRIPS planners perform similarly).

For each problem, all original IPC5 SAGs were kept, while all simple soft goals and other soft temporally extended goals were removed. Additional domain-specific SAGs were added to the problems in order to make them more challenging. Concerning these additional SAGs, in `Trucks` it is requested that 50% of the packages should be delivered by half of the originally specified deadline (using discrete levels of time). In `Rovers`, each rover should always avoid a specified set of locations (given a rover `T`, a way-point is considered forbidden for `T` if it is not mentioned in any original IPC5 preference regarding `T`); moreover, we used SAGs to specify the preference that at least one rover store remains empty in every state reached by a plan. Finally, in `TPP` SAGs are used to request that each type of goods should be carried by a specific truck, which has to buy the total goal quantity of the good and unload it visiting a deposit no more than once. The association between goods and trucks is randomly decided. For each STRIPS+SAG test problem, the utility value of every SAG is one, and the cost of every domain action is zero. The CPU-time limit for each run of each tested planner was 30 minutes.

Table 1 shows results obtained by running HPlan-P and MIPS over the considered STRIPS+SAG problems, and planners LPG (Gerevini, Saetti, and Serina 2003), Metric-FF (Hoffmann 2003) and LAMA (Richter and Westphal 2010) over the equivalent compiled test problems. Concerning

Rovers	LPG			Metric-FF			LAMA			HPlan-P		MIPS-XXL	
	MV	**RA/TA**	**∆ length**	**MV**	**RA/TA**	**∆ length**	**MV**	**RA/TA**	**∆ length**	**MV**	**RA**	**MV**	**RA**
P01 (1)	0	17/20	1,18	0	11/14	1,27	0	19/23	1,21	0	18	0	8
P02 (2)	0	12/17	1,42	0	15/18	1,20	0	17/21	1,24	0	26	0	10
P03 (4)	0	14/21	1,50	0	15/21	1,40	0	20/27	1,35	0	32	0	11
P04 (4)	0	9/16	1,78	0	9/16	1,78	0	11/18	1,64	0	65	0	9
P05 (4)	0	15/26	1,73	0	24/33	1,38	0	23/32	1,39	0	132	0	23
P06 (4)	2	40/52	1,30	2	39/51	1,31	2	38/50	1,32	2	287	4	37
P07 (5)	0	21/32	1,52	0	18/29	1,61	1	21/32	1,52	0	49	0	19
P08 (6)	0	31/43	1,39	2	26/38	1,46	0	26/38	1,46	0	64	0	27
P09 (6)	0	35/41	1,17	0	37/43	1,16	1	36/42	1,17	?	?	0	35
P10 (6)	0	39/54	1,38	0	37/52	1,41	0	38/53	1,39	0	161	0	37
P11 (6)	0	36/51	1,42	2	35/51	1,46	1	36/51	1,42	?	?	?	?
P12 (6)	0	24/30	1,25	1	19/25	1,32	0	21/27	1,29	0	33	0	20
P13 (6)	0	40/66	1,65	0	45/60	1,33	0	46/62	1,35	0	1936	?	?
P14 (6)	0	35/41	1,17	0	31/37	1,19	0	46/50	1,09	?	?	0	32
P15 (6)	0	46/52	1,13	1	45/51	1,13	0	52/58	1,12	?	?	?	?
P16 (6)	0	40/65	1,63	1	47/62	1,32	0	46/61	1,33	?	?	?	?
P17 (8)	0	40/65	1,63	0	52/60	1,15	0	51/59	1,16	?	?	?	?
P18 (8)	0	50/66	1,32	0	42/58	1,38	0	47/63	1,34	?	?	?	?
P19 (8)	0	80/88	1,10	2	84/91	1,08	2	76/84	1,11	?	?	?	?
P20 (10)	0	102/112	1,10	0	91/101	1,11	1	96/107	1,11	?	?	?	?

TPP	LPG			Metric-FF			LAMA			HPlan-P		MIPS-XXL	
	MV	**RA/TA**	**∆ length**	**MV**	**RA/TA**	**∆ length**	**MV**	**RA/TA**	**∆ length**	**MV**	**RA**	**MV**	**RA**
P01 (2)	0	15/22	1,47	2	18/22	1,22	0	13/18	1,38	0	18	0	9
P02 (6)	1	13/20	1,54	1	17/24	1,41	0	15/23	1,53	0	22	0	16
P03 (8)	1	17/27	1,59	3	21/40	1,90	0	17/28	1,65	0	24	0	21
P04 (8)	1	42/51	1,21	3	38/49	1,29	0	30/41	1,37	0	57	0	34
P05 (14)	3	68/88	1,29	4	51/77	1,51	3	50/68	1,36	1	73	?	?
P06 (14)	2	40/58	1,45	5	40/60	1,50	3	35/49	1,40	2	78	0	46
P07 (14)	2	33/50	1,52	4	36/56	1,56	3	29/47	1,62	2	75	?	?
P08 (16)	2	72/95	1,32	6	42/68	1,62	4	46/68	1,48	2	87	?	?
P09 (22)	6	57/87	1,53	7	65/98	1,51	6	69/89	1,29	7	141	?	?
P10 (42)	10	51/92	1,80	6	76/152	2	8	42/84	2	3	135	?	?
P11 (42)	11	77/121	1,57	10	70/163	2,33	7	79/111	1,41	5	211	?	?
P12 (45)	14	143/212	1,48	11	88/182	2,07	9	76/124	1,63	8	269	?	?
P13 (57)	19	75/135	1,80	16	62/131	2,11	11	60/125	2,08	29	501	?	?
P14 (54)	22	93/155	1,67	19	84/149	1,77	9	83/137	1,65	30	401	?	?
P15 (57)	?	?/?	?	23	108/186	1,72	11	113/171	1,51	?	?	?	?
P16 (60)	19	117/175	1,50	25	106/184	1,74	13	113/174	1,54	?	?	?	?
P17 (69)	19	116/205	1,77	?	?/?	?	14	97/189	1,95	?	?	?	?
P18 (72)	?	?/?	?	26	117/221	1,89	16	103/208	2,02	?	?	?	?
P19 (72)	?	?/?	?	28	110/206	1,87	12	106/194	1,83	?	?	?	?
P20 (75)	?	?/?	?	25	145/267	1,84	19	125/215	1,72	?	?	?	?

Trucks	LPG			Metric-FF			LAMA			HPlan-P		MIPS-XXL	
	MV	**RA/TA**	**∆ length**	**MV**	**RA/TA**	**∆ length**	**MV**	**RA/TA**	**∆ length**	**MV**	**RA**	**MV**	**RA**
P01 (4)	0	17/22	1,29	0	17/22	1,29	0	17/23	1,35	0	17	3	17
P02 (10)	7	19/30	1,58	9	19/30	1,58	5	23/32	1,39	8	20	5	20
P03 (12)	6	26/39	1,50	11	25/39	1,56	6	25/39	1,56	9	25	8	23
P04 (15)	8	28/44	1,57	13	28/45	1,61	7	30/47	1,57	10	30	?	?
P05 (17)	9	19/48	2,53	15	43/63	1,47	8	28/48	1,71	12	35	?	?
P06 (20)	12	38/62	1,63	16	37/61	1,65	9	40/63	1,58	?	?	?	?
P07 (36)	?	?/?	?	27	37/77	2,08	18	44/76	1,73	?	?	?	?
P08 (35)	24	42/82	1,95	31	47/87	1,85	20	49/89	1,82	?	?	?	?
P09 (38)	26	46/89	1,93	29	47/89	1,89	24	51/94	1,84	?	?	?	?
P10 (42)	?	?/?	?	34	54/102	1,89	26	59/107	1,81	?	?	?	?
P11 (45)	?	?/?	?	35	57/107	1,88	27	60/111	1,85	?	?	?	?
P12 (49)	34	61/115	1,89	35	60/116	1,93	28	65/121	1,86	?	?	?	?
P13 (52)	?	?/?	?	36	64/123	1,92	30	69/128	1,86	?	?	?	?
P14 (56)	?	?/?	?	42	69/131	1,90	33	71/135	1,90	?	?	?	?
P15 (59)	?	?/?	?	43	73/140	1,92	34	77/144	1,87	?	?	?	?
P16 (81)	?	?/?	?	65	77/166	2,16	55	88/177	2,01	?	?	?	?
P17 (85)	?	?/?	?	69	73/174	2,38	59	88/184	2,09	?	?	?	?
P18 (90)	?	?/?	?	71	82/180	2,20	34	77/144	1,87	?	?	?	?
P19 (94)	?	?/?	?	74	82/177	2,16	63	96/200	2,08	?	?	?	?
P20 (99)	?	?/?	?	?	?/?	?	55	55/177	3,22	?	?	?	?

Table 1: Performance comparison about solving uncompiled and compiled STRIPS+SAG problems. **MV**: metric value of a plan indicating the number of unsatisfied SAGs; **TA**: total number of plan actions; **RA**: number of real plan actions; **∆ length**: increment factor ($\frac{\text{TA}}{\text{RA}}$) in plan length between the plans of the original and compiled problems. ?: unsolved problem. Values in brackets are total numbers of SAGs in the corresponding problems.

HPlan-P and MIPS, in the discussion of the results we focus on HPlan-P since, as shown in Table 1, HPlan-P solves more problems and performs generally better than MIPS in terms of satisfied constraints (although, when MIPS solves a problem, it tends to perform better than HPlan-P in terms of plan length).

In general, we observe that HPlan-P solves many less problems, and no one of the largest/hardest ones, indicating that the compilation into STRIPS+ is a more scalable approach. For `Rovers`, the quality of the plans generated by the considered STRIPS planners is close or equal to the optimal: LPG shows the best results finding optimal solutions for all test problems (problem p6 has two SAGs that cannot be satisfied). For `TPP`, HPlan-P solves less problems than the compared STRIPS planners (and no one of the six largest problems), although in some cases it generates plans with slightly better quality in terms of satisfied SAGs. For `Trucks`, HPlan-P can solve only five of the twenty problems, and the quality of its plans in terms of satisfied SAGs is worse than the quality of the solutions computed for the compiled problems by planners LPG and LAMA.[4]

Concerning plan length, for each generated plan, Table `Trucks` indicates the total number of actions (TA) and the number of "real actions" (RA). Given a plan $\pi' = \langle \pi'', end, \pi''' \rangle$ for a compiled problem, an equivalent plan for the original STRIPS+SAG problem P can be derived from π'' by simply replacing each ω-sequence with the corresponding operator of P. Even if in our experiments the cost function ignores domain action costs and plan length (every original domain action has cost zero), it is interesting to observe that very often HPlan-P generates solutions that are much longer than the ones produced by solving the compiled problems. This also holds when comparison is done considering the total number of plan actions instead of only the real actions.

Finally, the results about plan length in the table indicate that, for the considered benchmarks, in practice the plan length increase for the solution plans of the compiled problems is relatively modest. From the data in column "Δ length", we can see that the factor incrementing plan length ranges between 1.08 to 3.2, and on average over all domains and planners it is 1.6. Specifically, for `Rovers` it ranges between 1.08 (with Metric-FF) and 1.78 (with Metric-FF and LPG), for `TPP` between 1.21 (LPG) and 2.33 (Metric-FF), and for `Trucks` between 1.29 (Metric-FF and LPG) and 3.2 (LAMA).

Conclusions

We have presented a compilation approach to handle soft always constraints in propositional planning. The proposed compilation scheme uses only STRIPS and action costs, which makes it usable by many existing powerful planners. A preliminary experimental analysis investigating the effectiveness of the approach shows good behaviour in terms of scalability and quality of the generated plans.

In addition to a deeper experimental analysis, current and future work concerns the compilation of other types of soft and hard state-trajectory constraints. It is worth noting that the proposed compilation can be trivially adapted to compile hard always constraints by just omitting the inclusion of (a) the *forgo* actions in the compiled problem, which forces the planner to reach the goal state with all always constraints of the problem satisfied by the plan, and (b) the $\overline{o}_{A_i}$ operators ($i = 1 \dots m$) generated by the compilation of each operator o threatening a set of constraints $A_1, \dots, A_m$. An experimental analysis to evaluate our compilation scheme for dealing with hard always constraints is ongoing. Preliminary results are encouraging.

[4]Note that, while the metric values (MV) of their plans are high, indicating that many SAGs are not satisfied, for the `Trucks` problems we estimate that the optimal solutions can satisfy at most from $1/5$ to $1/3$ of the specified SAGs. (Exact optimal bounds are unknown; the planners we used could not find better solutions even when running for a longer time).

Acknoledgments

We would like to thank Simone Quaresmini for his help with the implementation of the compilation scheme.

References

Bacchus, F., and Kabanza, F. 1998. Planning for temporally extended goals. *Annals of Mathematics and Artificial Intelligence* 22(1-2):5–27.

Baier, J., and McIlraith, S. 2006. Planning with first-order temporally extended goals using heuristic search. In *Proceedings of 21st National Conference on Artificial Intelligence AAAI'06.*

Baier, F. B., and McIlraith, S. A. 2008. A heuristic search approach to planning with temporally extended preferences. In *Proceedings of 20th International Joint Conference on Artificial Intelligence (IJCAI).*

Coles, A. J., and Coles, A. 2011. LPRPG-P: Relaxed plan heuristics for planning with preferences. In *Proceedings of 21st Internaltional Conference on Automated Planning and Scheduling (ICAPS'11).*

Do, M. B., and Kambhampati, S. 2004. Partial satisfaction (over-subscription) planning as heuristic search. In *Proceedings of 5th International Conference on Knowledge Based Computer Systems (KBCS'04).*

Edelkamp, S., and Jabbar, S. 2008. MIPS-XXL: Featuring external shortest path search for sequential optimal plans and external branch-and-bound for optimal net-benefit. In *6th International Planning Competition Booklet (ICAPS'08).*

Edelkamp, S.; Jabbar, S.; and Nazih, M. 2006. Large-scale optimal pddl3 planning with MIPS-XXL. In *5th International Planning Competition Booklet (ICAPS'06).*

Edelkamp, S. 2006. On the compilation of plan constraints and preferences. In *Proceedings of 16th International Conference of Automated Planning and Scheduling (ICAPS'06)*, 374–377.

Gazen, C. B., and Knoblock, C. A. 1997. Combining the expressiveness of UCPOP with the efficiency of Graphplan. In

Steel, S., and Alami, R., eds., *Recent Advances in AI Planning: 4th European Conference on Planning, ECP'97*. New York: Springer-Verlag.

Gerevini, A.; Haslum, P.; Long, D.; Saetti, A.; and Dimopoulos, Y. 2009. Deterministic planning in the fifth international planning competition: pddl3 and experimental evaluation of the planners. *Artificial Intelligence* 173(5-6):619–668.

Gerevini, A.; Saetti, A.; and Serina, I. 2003. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research (JAIR)* 20:239–290.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Hoffmann, J. 2003. The Metric-FF planning system: Translating ”ignoring delete lists” to numeric state variables. *Journal of Artificial Intelligence Research* 20(1):291–341.

Keyder, E., and Geffner, H. 2009. Soft goals can be compiled away. *Journal of Artificial Intelligence Research (JAIR)* 36:547–556.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39(1):127–177.

Smith, D. E. 2004. Choosing objectives in over-subscription planning. In *Proceedings of 14th Internaltiona Conference on Automated Planning and Scheduling (ICAPS'04)*.

van den Briel, M.; Sanchez, R.; Do, M. B.; and Kambhampati, S. 2004. Effective approaches for partial satisfaction (over-subscription) planning. In *Proceedings of 19th National Conference on Artificial Intelligence (AAAI'04)*.

Weld, D., and Etzioni, O. 1994. The first law of robotics (a call to arms). In *Proceedings of 12th National Conference on Artificial Intelligence AAAI'94*, 1042–1047.

Automated Knowledge Engineering Tools in Planning: State-of-the-art and Future Challenges

Rabia Jilani and **Andrew Crampton** and **Diane Kitchin** and **Mauro Vallati**
School of Computing and Engineering
University of Huddersfield
United Kingdom
{U1270695, a.crampton, d.kitchin, m.vallati}@hud.ac.uk

Abstract

Intelligent agents must have a model of the dynamics of the domain in which they act. Models can be encoded by human experts or, as required by autonomous systems, automatically acquired from observation. At the state of the art, there exist several systems for automated acquisition of planning domain models.

In this paper we present a brief overview of the automated tools that can be exploited to induce planning domain models. While reviewing the literature on the existing tools for Knowledge Engineering (KE), we do a comparative analysis of them. The analysis is based on a set of criteria. The aim of the analysis is to give insights into the strengths and weaknesses of the considered systems, and to provide input for new, forthcoming research on KE tools in order to address future challenges in the automated KE area.

Introduction

Both knowledge acquisition and knowledge engineering for AI planning systems are essential to improve their effectiveness and to expand the application focus in practice. The improvement process includes the study of planning application requirements, creating a model that explains the domain, and testing it with suitable planning engines to get a final product which consists of a domain model. Domain models can be encoded by human experts or automatically learned through the observation of some existing plans (behaviours). Encoding a domain model from observations is a very complex and time-consuming task, even for domain experts. Various approaches have been used to learn domain models from plans. This is of increasing importance: domain independent planners are now being used in a wide range of applications, but they should be able to refine their knowledge of the world in order to be exploited also in autonomous systems. Automated planners require action models described using languages such as the Planning Domain Definition Language (PDDL) (Mcdermott et al. 1998).

There have been reviews of existing knowledge engineering tools and techniques for AI Planning (Vaquero, Silva, and Beck 2011; Shah et al. 2013). Vaquero et al. (2011) provided a review of tools and methods that address the challenges encountered in each phase of a design process. Their work covers all the steps of the design cycles, and is focused on tools that can be exploited by human experts for encoding domain models. Shah et al. (2013) explored the deployment of automated planning to assist in the development of domain models for different real-world applications.

Currently, there is no published comparison research on KE tools for AI planning that automatically encode a domain model from observing plan traces. In this paper, we compare and analyse different state-of-the-art, automated KE tools that automatically discover action models from a set of successfully observed plans. Our special focus is to analyse the design issues of automated KE systems, the extent of learning that can take place, the inputs that systems require and the competency in the output domain model which systems induce for dealing with complex real problems. We evaluate nine different KE tools against the following criteria: (i) Input Requirements (ii) Provided Output (iii) Language (iv) Noise in Plans (v) Refinement (vi) Operational Efficiency (vii) User Experience, and (viii) Availability. By evaluating state-of-the-art tools we can gain insight into the quality and efficiency of systems for encoding domain models, and better understand the improvements needed in the design of future supporting tools.

The rest of this paper is organised as follows. We first provide an overview of existing automated KE tools for supporting the task of encoding planning domain models. Then we discuss the criteria that are used for comparing the different encoding methods. Finally, we summarise some guidelines for future tools.

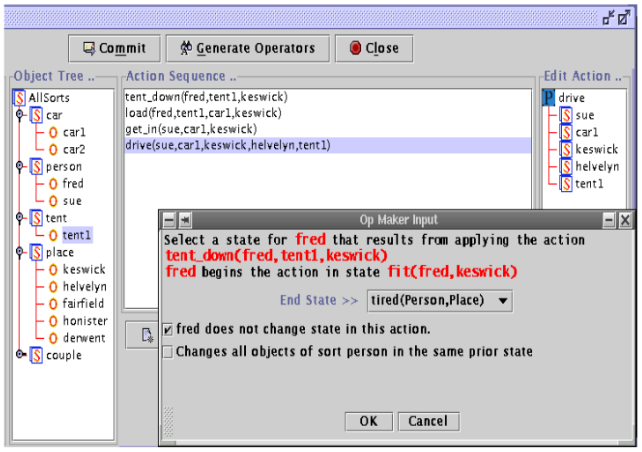
The State of the Art

In this section we provide an overview of KE tools that can be used for automatically producing planning domain models from existing plans or sequences of actions.

Opmaker

Opmaker (McCluskey, Richardson, and Simpson 2002) is a method for inducing primitive and hierarchical actions from examples, in order to reduce the human time needed for describing low level details related to operators' pre- and post-conditions.

Opmaker is an algorithm for inducing parameterized, hierarchical operator descriptions from example sequences and declarative domain knowledge (object hierarchy, object descriptions, etc.)



```
tent_down(fred,tent1,keswick)
load(fred,tent1,car1,keswick)
get_in(sue,car1,keswick)
drive(sue,car1,keswick,helvelyn,tent1)
```

Select a state for fred that results from applying the action
tent_down(fred,tent1,keswick)
fred begins the action in state fit(fred,keswick)

End State >> tired(Person,Place)

[✓] fred does not change state in this action.
[ ] Changes all objects of sort person in the same prior state

Figure 1: A screen shot of Opmaker.

The user has to specify an action and identify associated objects as being affected or unaffected by the action. The system uses static domain knowledge, the initial and goal states and a planning sequence as input. Using this knowledge, it first deduces possible state-change pathways and then uses them to induce the needed actions. These actions can then be learned or regenerated and improved according to requirement.

Opmaker extends GIPO, an integrated package for the construction of domain models, using a graphical user interface (Simpson, Kitchin, and McCluskey 2007). Figure 1 shows a screen shot of the graphical user interface.

SLAF

The SLAF (Simultaneous Learning and Filtering) algorithm (Shahaf and Amir 2006) learns action models in partially observable domains. As inputs, SLAF includes specifications of fluents, as well as partial observations of intermediate states between action executions. The pre-conditions and effects that this system generates in output includes implicit objects and unspecified relationships between objects through the use of action schema language.
As output the system learns action models (pre-conditions and effects) that also include conditional effects through a sequence of executed actions and partial observations. The action schema from this algorithm can be used in deterministic domains which involve many actions, relations and objects.
This algorithm uses a Direct Acyclic Graph representation of the formula. The results from this algorithm can be used in deterministic domains which involve many actions, relations and objects.

ARMS

ARMS (Action-Relation Modelling System) (Yang, Wu, and Jiang 2007) is a tool for learning action schema from observed plans with partial information. It is a system for automatically discovering action models from a set of observed plans where the intermediate states are either unknown or only partially known. To learn action schema, ARMS gathers knowledge on the statistical distribution of frequent sets of actions in the example plans. It then forms a weighted propositional satisfiability (weighted SAT) problem and resolves it using a weighted MAX-SAT solver. ARMS operates in two phases, where it first applies a frequent set mining algorithm to find the frequent subsets of plans that share a common set of parameters. It then applies a SAT algorithm for finding a consistent assignment of preconditions and effects.

ARMS needs partial intermediate states in addition to observed plan traces as input. The action model learnt from ARMS is not guaranteed to be completely correct, as the domain model induced is based on guesses with a minimal logical action model. This is why it can only serve as an additional component for the knowledge editors which provide advice for human users, such as GIPO (Simpson, Kitchin, and McCluskey 2007), and not as an independent, autonomous agent.

Opmaker2

Opmaker2 (an extension of Opmaker) (McCluskey et al. 2009) is a knowledge acquisition and formulation tool, which inputs a partial domain model and a training sequence, and outputs a set of PDDL operator schema including heuristics that can be used to make plan generation more efficient. It follows on from the original Opmaker idea. Its aims are similar to systems such as ARMS in that it supports the automated acquisition of a set of operator schema that can be used as input to an automated planning engine. Opmaker2 determines its own intermediate states of objects by tracking the changing states of each object in a training example sequence and making use of partial domain knowledge provided with input. Opmaker2 calls it the DetermineState procedure. The output from DetermineState is a map of states for each object in the example sequence. Parameterized operator schema are generated after applying the Opmaker algorithm for generalization of object references collected from example sequences.

LOCM

LOCM (Learning Object Centred Models) (Cresswell, McCluskey, and West 2013) is significantly different from other systems that learn action schema from examples. It requires only a set of valid plans as input to produce the required action schema as output. Valid plans should be formatted in a specific way; an example is given in Figure 2. LOCM is based on the assumption that the output domain model can be represented in an object-centred representation (Cresswell, McCluskey, and West 2013). Using an object-centred representation, LOCM outputs a set of parameterized Finite State Machines (FSMs) where each FSM represents the behaviour of each object in the learnt action schema. Such FSMs are then exploited in order to identify pre- and post-conditions of the domain operators. Although LOCM requires no background information, it usually requires many plan traces for synthesizing meaningful domain models. Moreover, LOCM is not able to automatically identify and encode static predicates.

```
sequence_task(1, [unstack(b8, b9),
stack(b8, b10), pick-up(b7), stack(b7,
b8), unstack(b9, b1), put-down(b9),
unstack(b1, b3), stack(b1, b9),
unstack(b3, b2), stack(b3, b6),
pick-up(b5), stack(b5, b3), unstack(b7,
b8), stack(b7, b2), unstack(b8, b10),
stack(b8, b7), pick-up(b10), stack(b10,
b5)], _, _).
```

Figure 2: An example of a blocksworld plan formatted as required by LOCM.

LOCM2

LOCM2 (Learning Object Centred Models 2) (Cresswell and Gregory 2011) followed on from the LOCM idea. Experiments have revealed that there are many examples that have no model in the representation used by LOCM. A common feature of domains which produce this issue is one where objects can have multiple aspects of their behaviour, and so they need multiple FSMs to represent each object's behaviour. LOCM2 generalizes the domain induction of LOCM by allowing multiple parameterised state machines to represent a single object, with each FSM characterised by a set of transitions. This enables a varied range of domain models to be fully learned. LOCM2 uses a transition-centred representation instead of the state-centred representation used by LOCM. The current LOCM and LOCM2 systems gather only the dynamic properties of a planning domain and not the static information. While domains used in planning also depend on static information, research is being carried out to fill that gap and make these systems able to induce both the dynamic and the static parts of domain models.

LSO-NIO

The system LSO-NIO (Learning STRIPS Operators from Noisy and Incomplete Observations) (Mourão et al. 2012) has been designed for allowing an autonomous agent to acquire domain models from its raw experience in the real world. In such environments, the agent's observation can be noisy (incorrect actions) and incomplete (missing actions). In order to acquire a complete STRIPS (Fikes and Nilsson 1972) domain model, the system requires a partial model, which describes objects' attributes and relations, and operators' names.

LSO-NIO exploits a two-staged approach. As a first stage, LSO-NIO learns action models by constructing a set of kernel classifiers, which are able to deal with noise and partial observability. The resulting models are "implicit" in the learnt parameters of the classifiers (Mourão, Petrick, and Steedman 2010). The implicit models act as a noise-free and fully observable source of information for the subsequent step, in which explicit action rules are extracted. The final output of LSO-NIO is a STRIPS domain model, ready to use for domain-independent planners.

RIM

RIM (Refining Incomplete Planning Domain Models) (Zhuo, Nguyen, and Kambhampati 2013) is a system designed for situations where a planning agent has an incomplete model which it needs to refine through learning. This method takes as input an incomplete model (with missing pre-conditions and effects in the actions), and a set of plans that are known to be correct. By executing given plan traces and preconditions/effects of the given incomplete model, it develops constraints and uses a MAX-SAT framework for learning the domain model (Zhuo et al. 2010). It outputs a "refined" model that not only captures additional precondition/effect knowledge about the given actions, but also "macro actions". A macro-action can be defined as a set of actions applied at a single time, that can quickly reach a goal at less depth in the search tree and thus problems which take a long time to solve might become solvable quickly.

In the first phase, it looks for candidate macros found from the plan traces, and in the second phase it learns precondition/effect models both for the primitive actions and the macro actions. Finally it uses the refined model to plan. The running time of this system increases polynomially with the number of input plan traces.

In the RIM paper the authors provide a comparison between RIM and ARMS by solving 50 different planning problems; through action models, refined and induced by the two systems. RIM uses both plans and incomplete domain models to induce a complete domain model but ARMS uses plans only, so to keep both systems' output on the same scale, RIM induces action models (used for comparison) based on plan traces only.

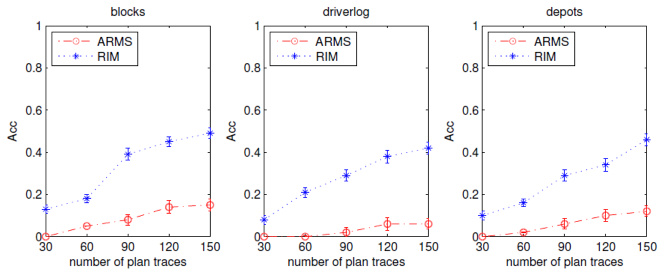


Figure 3: Comparison between RIM and ARMS (Zhuo, Nguyen, and Kambhampati 2013).

The average length of a plan is 18 when using action models learnt by ARMS; while the average length of plans (to the same problems as solved by ARMS) is 21. This is when using preferences of macro-operators learnt by RIM. Figure 3 shows a comparison of three different domains; the correctness of RIM is better than ARMS, as RIM also learns macro-operators and it uses macros to increase the accuracy of plans generated with the refined system.

AMAN

AMAN (Action-Model Acquisition from Noisy plan traces) (Zhuo and Kambhampati 2013) was designed to create domain models in situations where there is little or no possibility of collecting correct training data (plans). Usually, noisy plan traces are easier and cheaper to collect. An action is considered to be noisy if it is mistakenly observed.

AMAN works as follows. It builds a graphical model to capture the relations between actions (in plan traces) and states, and then learns the parameters of the graphical model. After that, AMAN generates a set of action models according to the learnt parameters. Specifically, AMAN first exploits the observed noisy plan traces to predict correct plan traces and the domain model based on the graphical model, and then executes the correct plan traces to calculate the reward of the predicted correct plan traces according to a predefined reward function. Then, AMAN updates the predicted plan traces and domain model based on the reward. It iteratively performs the above-mentioned steps until a given number of iterations is reached. Finally, the predicted domain model is provided.

In the AMAN paper, a comparison of AMAN and ARMS (Yang, Wu, and Jiang 2007) on noiseless inputs is provided.

Criteria for Evaluating Tools

We have identified several criteria that are useful for evaluating the existing KE automated tools for inducing domain models. Such criteria have been designed for investigating the KE tools' functionality from different perspectives: input, output, efficiency, availability and usability.

Input Requirements:

What inputs are required by a system to refine/induce a partial or full domain model? Input to the learning process could be training plans, observations, constraints, initial and goal states, predicates, and in some systems a partial domain model (with missing pre-conditions and effects in the actions).

Provided Output:

What is the extent of learning that the system can do?

Language:

What language does the system support to produce the output domain model? e.g. PDDL, STRIPS, and OCL etc.

Noise in Plans:

Is the tool able to deal with noise in plans? Noise in plans can be either incomplete plan traces (i.e., missing actions) or noisy actions. An action in a plan is considered to be noisy if it is incorrectly observed.

Refinement:

Does the tool refine existing domain models or does it build domain models from scratch?

Operational Efficiency:

How efficiently are the models produced? In general terms, the efficiency of a system could be seen as the ratio between input given to the system to do the learning process and the output domain model that we get as a result of learning.

User Experience:

Is the system/tool designed for inexperienced/beginner level planning users? Do users need to have a good knowledge of the system output language?

Availability and Usage:

Is the system available for open use? Does the system provide a user manual?

Tools Evaluation

In this section all the KE tools introduced in this paper are evaluated against the outlined criteria. Table 1 shows an overview of the comparison.

Inputs Requirements

The input to RIM, LOCM, LOCM2 and ARMS is a correct sequence of actions (training data in the form of plan traces), where each action in a plan is stated as a name and a list of objects that the action refers to. For some domains which require static knowledge, there is a need to mention static preconditions for the domain to be learnt; as LOCM and LOCM2 cannot learn static aspects of the domain. RIM in addition to a correct action sequence also requires an incomplete domain model (with missing pre-conditions and effects in the actions) as an input. ARMS makes use of background knowledge as input, comprising types, relations and initial and goal states to learn the domain model.

In comparison Opmaker2 learns from a single, valid example plan but also requires a partial domain model (declarative knowledge of objects hierarchy, descriptions, etc) as input.

AMAN and LSO-NIO, these systems learn from noisy (incorrect actions) and incomplete (missing actions) observations of real-world domains. Just like Opmaker2, LSO-NIO also requires a partial domain model, which describes objects (and their attributes and relations) as well as the name of the operators. The inputs to SLAF include specifications of fluents, as well as partial observations of intermediate states between action executions.

Provided Output

ARMS, Opmaker2, LOCM, LOCM2, LSO-NIO and RIM, the output of these systems is a complete domain model. In addition, LOCM also displays a graphical view of the finite state machines, based on which the object behaviour in the output model is learnt. To increase the efficiency of plans generated, Opmaker2 also includes heuristics while RIM also learns macro operators.

SLAF, as output the system learns an action model (preconditions and effects) that also includes conditional effects through a sequence of executed actions and partial observations.

Given a set of noisy action plans, AMAN generates multiple (candidate) domain models. To capture domain physics it produces a graphical model and learns its parameters.

Language

The domain model (also called domain description or action model) is the description of the objects, structure, states, goals and dynamics of the domain of planning (McCluskey et al. 2009).

LOCM, LOCM2 and ARMS are able to provide a PDDL domain model representation. RIM, AMAN and LSO-NIO can handle the STRIPS subset of PDDL. Opmaker and Opmaker2 use a higher level language called Object Centred Language (OCL) (McCluskey, Liu, and Simpson 2003; McCluskey and Porteous 1997) for domain modelling. Their output is an OCL domain model, but Opmaker can exploit the GIPO tool to translate the generated models into PDDL. Finally, SLAF System is able to exploit several languages to represent action schemas; starting from the most basic language SL, and then there is SL-V and SL-H (Shahaf and Amir 2006). Such languages are not usually supported by domain-independent planners.

Noise in Plans

Most of the existing KE tools require valid plans. AMAN and LSO-NIO are the systems that can deal with noisy plan traces. Moreover, LSO-NIO is also able to handle incomplete plan traces. On the other hand, also LAMP (Zhuo et al. 2010), on which RIM is based, is able to exploit partial plan traces, in which some actions are missing.

Refinement

Most existing work on learning planning models learns a complete new domain model. The only tool among all those reviewed in the paper that is able to refine an existing domain model is RIM. RIM takes in correct plan traces as well as an incomplete domain model (with missing pre-conditions and effects in the actions), to refine it by capturing required pre-condition/effects.

On the other hand, since Opmaker, Opmaker2, LSO-NIO and SLAF require as input part of the domain knowledge, to some extent they are actually refining the provided knowledge.

Operational Efficiency

The efficiency of a system could be seen as the ratio between the input given to the system for the learning process and the output domain model we get as a result of learning. As shown in the review of the considered tools, all systems have different and useful motivations behind their development. Given their relevant features of input requirements and learning extent, we can say that the system needing the least input assistance and which induces the most complete domain model is the most efficient one. On such a scale the AMAN, LOCM and LOCM2 approaches have the best performance. In order to provide a complete domain model they require only some plan traces (sequence of actions). Based on strong assumptions they output the solver-ready domain model. Similarly ARMS, though requiring richer inputs, outputs a solution which is optimal; in that it checks error and redundancy rates in the domain model and reduces them. In contrast Opmaker2 learns from a single example together with a partial domain model, and for output it not only produces a domain model, but also includes heuristics that can be used to make plan generation through the domain model more efficient.

User Experience

By experience we mean to evaluate how far the system/tool is designed for use by inexperienced/beginner level planning users. Most of these systems are built with the motivation to open up planning engines to general use. Opmaker is incorporated into GIPO as an action induction system, as GIPO is an integrated package for the construction of domain models in the form of a graphical user interface. It is used both as a research platform and in education. It has been used to support the teaching of artificial intelligence (AI) planning to students with a low-experience level (Simpson, Kitchin, and McCluskey 2007).

The other systems are being used as standalone systems, they do not provide a GUI, and require the guidance of planning experts for usage. Certain systems also require separate formats for providing inputs, e.g., LOCM requires input plan traces in Prolog while many major planning engines use PDDL as a planning language. So the conversion from PDDL to Prolog is a time consuming task and requires experienced users.

Availability and Usage

Very few systems are available on-line and open to download and practice. No systems provide documentation that make usage easy for beginners - except GIPO (Opmaker).

Guidelines and Recommendations

We will now discuss the guidelines and recommendations that we have derived from our review and assessment of the nine different state-of-the-art automated KE tools.

To create or refine an already existing domain model requires many plan examples and sometimes other inputs such as full or partial knowledge about predicates, initial, intermediate and goal states, and sometimes a partial domain model. One major concern at this stage is the way plan traces can be collected. There are three general ways to collect example plans. The first is when plans are generated through goal oriented solutions, the second through random walks and thirdly through observation of the environment by a human or by an agent. Goal oriented plan solutions are generally expensive in that a tool or a planner is needed to generate a large enough number of correct plans to be used by the system. To do this one must also have a pre-existing domain model. Observation by an agent has a high chance that noise will be introduced in the plan collection; which can clearly affect the learning process. Currently most working systems assume the input knowledge to be correct and consequently not suitable for real-world applications. To increase potential utility, systems should be able to show equal robustness to noise.

Another issue is the expressiveness of the output domain model. Observing the output of the current automated learning systems, there is a need to extend their development so that they can also learn metric domains that include durative actions, action costs and other resources. In other words, the

Criteria	**AMAN**	**ARMS**	**LOCM**	**LOCM2**	**LSO-NIO**	**Opmaker**	**Opmaker2**	**RIM**	**SLAF**
Inputs	NP	BK,P	P	P	PDM,NP	PDM,P	PDM,P	PDM,P	Pr,IS
Outputs	DM	DM	DM	DM	DM	DM	DM,H	RDM	DM
Language	STRIPS	PDDL	PDDL	PDDL	STRIPS	OCL	OCL	STRIPS	SL
Noise	$+$	$-$	$-$	$-$	$+$	$-$	$-$	$-$	$-$
Refinement	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$+$	$-$
Efficiency	$+$	i	$+$	$+$	i	$-$	i	$-$	$-$
Experience	$-$	$-$	$-$	$-$	$-$	$+$	$+$	$-$	$-$
Availability	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$

Table 1: Comparison of KE Tools. P: Plan traces; BK: Background Knowledge; PDM: Partial Domain Model; Pr: Predicates; IS: Intermediate States; NP: Noisy Plans; DM: Domain Model; RDM: Refined Domain Model; H: Heuristics. Where available, $+$, i (intermediate) or $-$ give a qualitative evaluation w.r.t. the corresponding metric.

systems should broaden the scope of domain model generation to produce more expressive versions of PDDL that can be applied to a greater range of real-world problems.
Systems which learn only from plan traces could make the output domain model more intelligible and useful by assigning meaningful names to all learnt fluents/predicates.
To enhance the potential utility of the induced domain in the real-world, error and redundancy checks should be performed in order to enhance the effectiveness of plans generated by planning engines using these domains.
To make learning systems more accessible and open to use by research students and the scientific community, these systems should be available on-line, and include a GUI and user manual for ease of use by non-planning experts. A significant extension would be to create a consistent interface across all systems for specifying inputs. Having to convert PDDL plans into Prolog, for example, is likely to inhibit the uptake of automated KE tools by non-experts rather than encourage it.

Conclusion

In order to encourage the exploitation of Automated Planning in autonomous systems, techniques for the automatic acquisition of domain models are of fundamental importance. Providing robust and expressive automated KE tools for domain model acquisition, that can be easily used by non-planning experts, will better promote the application of planning in real-world environments; particularly in applications where the actual domain model is unclear and/or too complex to design manually.

In this paper we have presented the state-of-the-art of Knowledge Engineering tools for the automatic acquisition of planning domain models. We proposed and used a set of criteria consisting of: input requirements, output, efficiency, supported language, ability to handle noisy plans, ability to refine existing models, user experience and availability. We observed that different tools require very different inputs and, usually, are designed for experienced users. We highlighted the weaknesses of existing methods and tools and we discussed the need for PDDL-inspired development in the design of future tool support.

Future work will include an experimental comparison, based on a case-study. We are also interested in improving existing KE tools for overcoming the major weaknesses that this review has highlighted.

References

Cresswell, S., and Gregory, P. 2011. Generalised domain model acquisition from action traces. In *Proceedings of The 21th International Conference on Automated Planning & Scheduling (ICAPS-11)*.

Cresswell, S. N.; McCluskey, T. L.; and West, M. M. 2013. Acquiring planning domain models using locm. *The Knowledge Engineering Review* 28(02):195–213.

Fikes, R. E., and Nilsson, N. J. 1972. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence* 2(3):189–208.

McCluskey, T. L., and Porteous, J. M. 1997. Engineering and compiling planning domain models to promote validity and efficiency. *Artificial Intelligence* 95(1):1–65.

McCluskey, T. L.; Cresswell, S.; Richardson, N. E.; and West, M. M. 2009. Automated acquisition of action knowledge. In *Proceedings of the International Conference on Agents and Artificial Intelligence (ICAART)*, 93–100.

McCluskey, T.; Liu, D.; and Simpson, R. M. 2003. Gipo ii: Htn planning in a tool-supported knowledge engineering environment. In *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS 2003)*, volume 3, 92–101.

McCluskey, T. L.; Richardson, N. E.; and Simpson, R. M. 2002. An interactive method for inducing operator descriptions. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems*, 121–130.

Mcdermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Mourão, K.; Zettlemoyer, L. S.; Mark, R. P.; and Steedman. 2012. Learning strips operators from noisy and incomplete observations. In *Proceedings of the Twenty Eighth Conference on Uncertainty in Artificial Intelligence (UAI-12)*, 614–623.

Mourão, K.; Petrick, R. P. A.; and Steedman, M. 2010. Learning action effects in partially observable domains. In

Proceedings of the 19th European Conference on Artificial Intelligence (ECAI-10), 973–974.

Shah, M.; Chrpa, L.; Jimoh, F.; Kitchin, D.; McCluskey, T.; Parkinson, S.; and Vallati, M. 2013. Knowledge engineering tools in planning: State-of-the-art and future challenges. In *Proceedings of the Knowledge Engineering for Planning and Scheduling workshop (KEPS)*.

Shahaf, D., and Amir, E. 2006. Learning partially observable action schemas. In *The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference (AAAI-06)*, 913–919.

Simpson, R. M.; Kitchin, D. E.; and McCluskey, T. 2007. Planning domain definition using gipo. *The Knowledge Engineering Review* 22(02):117–134.

Vaquero, T. S.; Silva, J. R.; and Beck, J. C. 2011. A brief review of tools and methods for knowledge engineering for planning & scheduling. In *Proceedings of the Knowledge Engineering for Planning and Scheduling workshop (KEPS)*.

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted max-sat. *Artificial Intelligence* 171(2-3):107–143.

Zhuo, H. H., and Kambhampati, S. 2013. Action-model acquisition from noisy plan traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI-13)*, 2444–2450. AAAI Press.

Zhuo, H. H.; Yang, Q.; Hu, D. H.; and Li, L. 2010. Learning complex action models with quantifiers and logical implications. *Artificial Intelligence* 174(18):1540 – 1569.

Zhuo, H. H.; Nguyen, T.; and Kambhampati, S. 2013. Refining incomplete planning domain models through plan traces. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, 2451–2457. AAAI Press.

Goal-directed Generation of Exercise Sets for Upper-Limb Rehabilitation

José C. Pulido, José C. González, Arturo González-Ferrer, Javier García, Fernando Fernández
Planning and Learning Group, Departamento de Informática
Universidad Carlos III de Madrid
{jcpulido,josgonza,artgonza,fjgpolo,ffernand}@inf.uc3m.es

Antonio Bandera
Dpto. de Tecnología Electrónica
Universidad de Málaga
ajbandera@uma.es

Pablo Bustos
Robolab
Universidad de Extremadura
pbustos@unex.es

Cristina Suárez
Grupo de Innovación Tecnológica
Hospital Virgen del Rocio, Sevilla
cristina.suarez.exts@juntadeandalucia.es

Abstract

A rehabilitation therapy usually derives from general goals set by the medical expert, who requests the patient to attend sessions during a certain time period in order to help him regaining mobility, strength and/or flexibility. The therapist must transform these general goals manually into a set of exercises distributed over different rehabilitation sessions that compose the complete therapy plan, taking into account the patient clinical conditions and a predetermined session and therapy time. This becomes a hard task and might lead to rigid schedules which not always accomplish the desired achievement level of therapeutic objectives established by the physician and could have a negative impact on the patients' engagement in the therapy. Classical and Hierarchical Task Network planning approaches have been used in this paper to compare the modelling and results of both domain formulations for the automatic generation of therapy plans for patients suffering obstetric brachial plexus palsy, in response to a given set of therapeutic objectives.

Introduction

Clinical Decision Support Systems (CDSS) have been developed in the last decades to facilitate many tasks of physicians, like helping them in implementing Clinical Practice Guidelines (CPGs) through ad-hoc computer-interpretable models (Peleg 2013). In some cases, it might happen that the protocol to treat a patient condition is not so clear, and the procedure to design the treatment pathway depends directly on a set of expected therapeutic objectives that the patient should achieve. In this case, guidelines can only give high-level recommendations on what combination of therapies to establish for a patient condition, but it still may require higher effort for the physician to deal with the configuration of the most appropriate combination of steps to maximize the expected outcome, for example according to a standard scale. This is the case of rehabilitation therapies for obstetric brachial plexus palsy (OBPP), the condition where this paper is scoped. OBPP is a serious injury that causes a loss of movement or weakness of the affected upper-limb. It is caused when the collection of nerves around the shoulder are damaged during the birth. In order to design the OBPP rehabilitation stage in the Virgen del Rocío University Hospital (Seville, Spain)[1], a set of therapeutic objectives is established after the anamnesis stage, according to the evaluated conditions of the patient. Taking these objectives into consideration, sequential, time-limited sessions of exercises that aim to achieve those goals have to be designed by the medical experts. The patients will carry out the rehabilitation sessions with periodic evaluations to check if they are progressing and achieving the expected outcome. Their progress along the therapy is measured using the Goal Attainment Scale (GAS) (Turner-Stokes 2009). In this scenario, physicians need to design combinations of exercises that contribute in a quantitatively measureable way to one or several therapeutic objectives, that might conflict among them, and that might have time, order, intensity or difficulty constraints in order to be selected.

This paper proposes to model the design of rehabilitation therapies by means of Automated Planning, which provides an automatic method to support physicians in the design of these sessions. After their clinical feasibility validation, the generated therapy plan could be projected into a programmable humanoid robotic platform that will serve as training assistant to patients, as expected in the THERAPIST project (Calderita et al. 2013). To achieve this goal, three main steps have been performed and described in this manuscript. Firstly, a domain analysis and specification have been performed with the help of physicians and therapists at Virgen del Rocío Hospital, as described in the following section. Then, we have studied how to formalize the domain with two different automated planning approaches: classical STRIPS planning and Hierarchical Task Network (HTN) Planning (Ghallab, Nau, and Traverso 2004; Erol, Hendler, and Nau 1994). Finally, we have performed an initial empirical, qualitative evaluation of such models with concrete planners to check their capabilities, including an extended discussion to highlight their strengths and weaknesses.

Related Work

There has been some work in the automatic generation of therapy plans or treatments. (Ahmed et al. 2010) present a system for the automatic generation of treatments in cancer patients. The system is concerned with the correct selection of the geometry and intensity of the irradiation to

[1]http://www.huvr.es (accessed May 20, 2014).

produce the best dose distribution. In (Morignot et al. 2010) authors use also Automated Planning for generating scenarios helping handicapped people. In (Fdez-Olivares et al. 2011; González-Ferrer et al. 2013) authors used a planning algorithm able to generate oncology treatment plans, and transforming Asbru computer-interpretable guidelines of the Hodgkin disease protocol, which include temporal constraints difficult to schedule manually by physicians. (Schimmelpfeng, Helber, and Kasper 2012) present a mixed-integer linear programming (MILP) approach to determine appointments for patients of rehab hospitals. However, they do not plan the specific exercises within each session to achieve some predetermined goals according to the requirements of the patient as detailed in our work.

Domain Analysis and Specification

There are three main actors involved in the therapeutic protocol: the therapist, the medical expert and the patient. The therapy plan is composed of sessions, each one composed of different exercises. The medical expert determines the number of sessions of the therapy and some constraints to prevent the training of certain groups of exercises depending on the patient profile. This expert also decides which general therapeutic objectives, out of five, should be trained during a rehabilitation therapy: bimanual, fine unimanual, coarse unimanual activities, arm positioning or hand positioning activities.

The main goal of the therapist is to help the patient to perform the rehabilitation sessions while evaluating the patient evolution. When planning a therapy session, the therapist selects the exercises which, according to his experience, are better to fulfill the therapeutic objectives in a fixed amount of time. The hospital that participates in our project follows general guidelines of the available exercises categorized according to affected body sites, where each group trains a certain patient capability. The therapist is also free to use his creativity to improvise new exercises in order to better accomplish the goals imposed by the medical expert. A reasonable session might be organized as follows: the initial exercises serve as warming up, the most intense exercises are performed in the central part of the session and the final exercises as cool-down phase.

The evolution of the patient is evaluated using the GAS scale (Turner-Stokes 2009). Depending on the results, the medical expert can change some therapy features, for example removing a therapeutic objective or adding another one. This system has little flexibility because it does not allow to reduce or increase the priority of the objectives by some degree. The selected exercises in a session depend greatly in the intuition of the therapist. These exercises could not be the most appropriate to achieve the therapeutic objectives and, at the end of the therapy, some of these objectives could not be completely fulfilled, putting at risk the rehabilitation success.

Having different exercises for each therapeutic objective is convenient because using an assorted exercise set may enrich the therapy quality. However, selecting the adequate exercises according to the therapeutic objectives, observing the patient profile constraints and assuring the variability of the sessions, is a time-consuming task for therapists. It causes that the therapist often do not care about finding the suitable set of exercises, so the trained sessions are usually repetitive. This may reduce the engagement of the patient in the therapy.

Model, Constraints and Requirements

Finding a plan of exercises for each session while taking into account all the requirements set by the medical expert is a search task which can be solved with Automated Planning. A database with exercises of different characteristics is available for the system to be developed. This database provides metrics to guarantee that the planned therapy fulfills all the requirements of the medical expert. To increase the flexibility when selecting exercises, the therapeutic objectives variables are graded with four adequacy values, $\{0,1,2,3\}$, as used in the mentioned GAS scale. These values will contribute to reach the *therapeutic objectives cumulative levels* (TOCL) established for a session. The system will provide the planned sessions to the therapist, that only needs to validate them. Of course, he can ultimately decide to change any exercise, if he considers it as not appropriate.

Next we show the constraints and requirements followed to plan the exercises that will be included in the rehabilitation sessions.

Goals

- Total number of sessions.
- Minimum and maximum duration of each session.
- The defined TOCL thresholds.

Exercise characteristics

- Duration (in minutes).
- Adequacy level for each therapeutic objective.
- Intensity value is associated to the average heart rate while performing the exercise.
- Difficulty for a certain patient to perform the exercise. This variable could be updated by the therapist after each session, if needed.
- Each exercise belongs to a group of exercises. These groups are related to the capabilities that patients need to perform the exercise, possibly restricted by their clinical conditions.

The next constraints are considered in order to guarantee the medical requirements and the variability of the sessions:

Basic constraints

- Each session must have three phases in the following order: warm-up, training and cool-down.
- The duration of each phase and each session must be inside a predefined range.

Variability constraints

- The repetition of a certain exercise in the same session is not allowed.
- The exercise distribution should be assorted throughout the sessions.

Patient-related constraints

- Avoiding a certain group (e.g elbow flexion) or a set of exercises (e.g. those too much intense or difficult) could be required because of patient conditions.
- Select certain types of exercises. For instance, if the patient suffers "Upper Erb OBPP", recommend only exercises for shoulder abduction, external rotation of shoulder and elbow flexion; if he suffers "Extended Erb OBP", add wrist flexion as well.
- Within a session, limit the cumulative intensity or difficulty to a given value.

With this information, the automated planner can find a suitable therapy plan if there are enough exercises in the database. In case that the available exercises are not enough, the automated planner will ask the therapist that it needs to learn a new exercise with a suggested value for some characteristics. For example, in a session plan, the planner can suggest the execution (and learning) of a new exercise with adequate level of 2 for bimanual activities. The planner assumes that the learnt exercise is performed by the patient and uses the minimum estimated values to compute the calculations for the plan. When the therapist stores the new exercise in the database, it can have higher adequacy levels for the therapeutic objectives, guaranteeing that the plan will continue being valid. In future sessions, the previously learnt exercises can be reused, minimizing the need of further learning actions and helping the therapist to fill the database with a set of useful exercises. After a session, the therapist can update the difficulty values of the exercises for a patient, if needed. The medical expert can also modify the goals with the results of the GAS scale evaluation. This updates can cause a replanning of the remaining sessions, if the previously planned therapy is no longer valid.

Methods

We propose the use of Automated Planning techniques (Ghallab, Nau, and Traverso 2004) to plan the exercises that will belong to each session. Automated Planning is an Artificial Intelligence (AI) technique that is used to find a plan of actions while respecting the model constraints. We have tested two different paradigms: classical STRIPS planning and Hierarchical Task Network (HTN) planning. In classical planning, given a model composed of an initial state, possible actions that have preconditions to be fulfilled, effects over the state and a set of goals that have to be accomplished in the final state, a planner is able to generate valid plans of actions to achieve the specified goals. HTN Planning (Erol, Hendler, and Nau 1994; Nau et al. 2003) is based a hierarchy of composed tasks and primitive actions. Composed tasks are high-level tasks that can be decomposed using methods that have to fulfill a precondition to be selected and applied, while primitive actions are modelled as in classical planning.

In order to check the viability and to measure the suitability and performance of each automated planning paradigm, two different knowledge engineers of our group addressed the presented problem using two concrete AI planners: Cost-based Planner (CBP) (Fuentetaja, Borrajo, and López 2010) for the classical paradigm and JSHOP2 (Nau et al. 2003) for the hierarchical one. Subsequent meetings with other group experts were carried out to discuss modelling approaches and results. We describe next these two models.

Classical Planning

The proposed domain for this planning model is based mainly in fluents and action costs, introduced in PDDL 2.1 (Fox and Long 2003). These requirements have yet a lack of support from many of the most current planners, but in this domain they are specially useful to operate directly with the quantitative values of the therapeutic objectives. CBP planner supports these characteristics and its search method is guided by a selection of actions extracted from a relaxed planning graph. Also, they are useful to control the session length, add specific variability restrictions and to establish a dynamic preference for certain actions. The most important design criterion that we followed in the classical model is that each individual session has to fulfill always the therapeutic objectives while observing the time duration constrains. A secondary criterion consists in forcing the variability among the therapy sessions to avoid monotony and improve the treatment engagement. This domain also has the possibility to "plan the learning" of new exercises with some suggested attributes to be executed in a session if there are not enough exercises in the database. This learning mechanism is explained in more detail in a later subsection. To clarify the further explanation, we show a plan for just one session in Figure 1. A full therapy plan will have every session planned, addressing all the dependencies among them.

```
 0: (SESSION-START)
 1: (WARMUP-PHASE)
 2: (WARMUP-DATABASE-EXERCISE E0)
 3: (TRAINING-PHASE)
 4: (TRAINING-DATABASE-EXERCISE E11)
 5: (TRAINING-DATABASE-EXERCISE E12)
 6: (TRAINING-DATABASE-EXERCISE E10)
 7: (TRAINING-DATABASE-EXERCISE E9)
 8: (LEARN-TRAINING-EXERCISE O_SPATIAL_HAND A_MEDIUM
      D_LONG I_INTENSE)
 9: (COOLDOWN-PHASE)
10: (COOLDOWN-DATABASE-EXERCISE E15)
11: (SESSION-END)
```

Figure 1: Output plan for one session.

Planning Problem

Goals The medical expert is in charge of determining the characteristics of the therapy. Firstly, he decides the total number of sessions and the minimum and maximum durations of each phase. This data is stored in the initialization part of the PDDL problem to serve as a common background for all sessions. There are other two important tasks for this expert: choose restrictions depending on the patient's profile and determine the amount of training for each therapeutic objective in each session. These are done using PDDL goals.

There is a fluent for each therapeutic objective to accumulate all the corresponding adequacy values of the planned exercises in a session. These can be defined as the amount of training for a certain therapeutic objective (the aforementioned TOCL values). An objective will be achieved if it is trained enough, so it is sufficient to assign a goal with a lower threshold for each objective to be trained. As an example, for a 30 minutes session:

```
(>= (TOCL t_bimanual) 15)
(>= (TOCL t_unimanual_fine) 7)
(>= (TOCL t_spatial_arm) 7)
```

Numeric goals in PDDL permit a great flexibility to configure the range of desired values for each fluent at the end of each session. It could be also possible to establish an upper limit for each objective (less or equal condition) or even avoid the training of a certain objective (equals to zero), but this has less medical sense because these values are just a form to represent the priority of the therapeutic objectives and they do not need to be directly related with the intensity of the exercises, for which there is a different fluent.

Exercise Database The database contains all the stored exercises. It is fully managed by the therapist, adding exercises when the system suggests it with learning actions or when the therapist finds it convenient. This information observes all the characteristics of the exercises mentioned in previous sections. The difficulty value of each exercise is stored in the patient's profile, but to simplify we consider that they are loaded in the PDDL problem file before the planning task. To assure the variability constraints, there are two additional fluents representing the session number and the position of the exercise in the the last session where it appeared. Each exercise has a predicate to be able to appear in the warm-up, training or cool-down phase.

The system assumes that the information of the database is coherent, so the therapist has to be sure that the exercises are correct when he adds them. For example, warm-up exercises should not be too intense. With these considerations, the session plan will start with soft intensity, followed by an intense training phase and ending with softer exercises again. Below there is an example of a generic therapeutic exercise (e7) modelled in PDDL:

```
(e_phase e7 p_training)
(e_group e7 g_arm_independence)
(= (e_last_session e7) 2)
(= (e_last_position e7) 4)
(= (e_intensity e7) 48)
(= (e_difficulty e7) 39)
(= (e_duration e7) 4)
(= (e_adequacy e7 t_bimanual) 0)
(= (e_adequacy e7 t_unimanual_fine) 3)
(= (e_adequacy e7 t_unimanual_coarse) 0)
(= (e_adequacy e7 t_spatial_arm) 1)
(= (e_adequacy e7 t_spatial_hand) 0)
```

Planning Domain

Actions All actions are strongly based in fluents, having numeric preconditions and action costs. There are two standard action types: to control the session flow and to add exercises from the database.

Flow control actions allow moving among warming up, training and cooling down phases or determining the start and end of a session. When the minimum time for a phase has been reached, it is possible to move to the next phase or to finish the session.

The basic way to add exercises to a session is through actions which select them from the database. They check that there is available time in the current phase and constraints like the maximum cumulative intensity. Only exercises for the current phase can be selected. To assure variability there are two restrictions modelled as preconditions in the PDDL domain:

- The exercise cannot be used in the last three sessions.
- In the training phase, an exercise cannot be trained in the same position as in the last session in which it appeared.[2]

Learning Exercises Learning actions helps the therapist to add new useful exercises to the database as the system is being used. When the system has difficulties to find a valid plan, it can ask the therapist to provide a new exercise to continue planning. Our hypothesis states that the bigger the database is, the less new learnings will be needed. It is preferable to use exercises in the database instead of learning new ones, but is not necessary to explore all the possible combinations before trying a learning action. This has been controlled using a higher action cost. The planner tries to minimize the total cost of the plan, so learning actions tend to be used few times.

To increase variability, the action cost will be higher when the new exercise allows reaching the problem goals faster. In this way, longer and less adequate exercises are preferred for the main target.

Planning Strategy

Classical planning has to deal with a major problem in this domain. Plan only one session is somewhat relatively easy, but a real therapy is composed of about 20 sessions. The first approximation was to plan the full therapy, generating plans which contain more than 250 actions. Planning multiple sessions in one run causes a non-linear complexity increase because there are dependencies among them. The planner has to do backtracking if the selected exercises for a session are not valid. The problem appears when the planner goes back further than needed, maybe many valid sessions, forcing to replan these sessions again to find a valid alternative for a later one. A smaller backtracking of just a few actions could solve the situation allowing reordering of the exercises, selecting others or planning the learning of a new one to continue onward.

We use a divide and conquer strategy to plan each session individually taking into account the dependencies of one another. In particular, the planner is called one time for each session that we want to plan. Each time that the planner returns a plan, it is parsed to determine all the database and learnt exercises planned. For the next session, a new problem file is generated to update the predicates and functions of the exercises of the last session planned, and add the new learnt exercises to the database. Then, the planner is executed again with the problem file for the next session. So for each session, a PDDL problem file is generated with the new exercises and updates of the database exercises. The experiments showed that this strategy is much faster than planning all the sessions in one run, without affecting the quality

[2]For warm-up and cool-down phases the condition is not applicable because a long exercise can reach by itself the minimum time of the phase and cannot be reordered (e.g. in the warm-up phase, such exercise will always appear in the first position).

Planned exercises →

Sessions ↓	1	2	3	4	5	6	7	8
1	e0	e9	e11	e12	e10	e7	e15	
2	e4	e2	e5	e6	L	L	L	e13
3	e1	e3	e8	L	L	L	L	e16
4	L	L	L	L	L	L	e17	
5	e0	e11	e12	e10	e9	L	e15	
6	e4	e2	e6	L19	e7	e5	L20	e13
7	e1	e3	L24	e8	L23	L22	e16	
8	L25	L26	L30	L27	L28	L29	e17	
9	e0	e12	e10	L31	e11	e9	e15	
10	e4	e2	L19	L20	e6	e7	e13	
11	e1	e3	L22	L23	L24	e8	e16	
12	L	L25	L26	L29	L30	L27	L28	e17
13	e0	e10	L21	e12	e9	e11	e15	
14	e4	e2	e7	e6	L20	L19	e13	
15	e1	e3	L24	e8	L22	L23	e16	
16	L25	L27	L31	L26	e5	L29	L30	e17
17	e0	e11	e12	L28	e10	e9	e15	
18	L32	e4	e2	L19	e6	L20	e7	e13
19	e1	e3	L22	L23	L24	e8	e16	
20	L25	L26	e5	L29	L30	e17		

Table 1: Therapy plan with few exercises in the database. An "e" or "L" with a number represents an exercise stored in the database (initial or learnt, respectively). A single "L" represents the learning and execution of a new exercise.

of the plans. The capacity to learn new exercises gives to the sessions some locality properties that can be exploited to avoid time-consuming backtracking among sessions.

Empirical Evaluation

We used the CBP automated planner (Fuentetaja, Borrajo, and López 2010) because it was specially designed to work with action costs, so its heuristics reduce the total number of new learned actions. In Table 1 there is an example of a therapy with 20 sessions. The database starts with a controlled set of exercises: 5 warm-up, 8 training and 5 cool-down exercises.

In session 1, the planner only uses exercises from the database because they are useful to reach the TOCL thresholds. In sessions 2 and 3, it needs to learn due to variability constraints. In session 4, almost all the exercises has been used in the last three sessions, so it has to continue learning new ones. In session 5, the planner can use the set of exercises of session 1 again, but it varies the order of the training phase because the exercises cannot appear in the same position as the last time. In the following sessions, the learnt exercises are reused because they continue being useful to fulfill the goals, so more learning actions are not needed. Note that the planned sessions are very different among them.

The new learnings in session 5 and 12 show that learning actions are not completely prohibited, so it is not needed to explore all the combinations in the database before using a learning action. Also, we only use the first plan returned by CBP. This planner can improve the plans iteratively if it has time, but the first plan returned is good enough to see how the system works. With this configuration, the planning time usually does not take more than five minutes. In the initial experimentation, we observed that the principal aspects that increase planning time are the number of learned actions needed and the TOCL thresholds.

HTN Planning

The automatic generation of therapies is a problem that can also be managed in a hierarchical way, where the top of the pyramid contains a task representing the whole therapy, which is divided into sessions and each session comprises a set of exercises, as shown in Figure 2. The session structure is given by the hierarchical and order relationships represented in the HTN decomposition. This approach aims to provide an easily extendible and configurable model, where human expert knowledge can be included at any time.

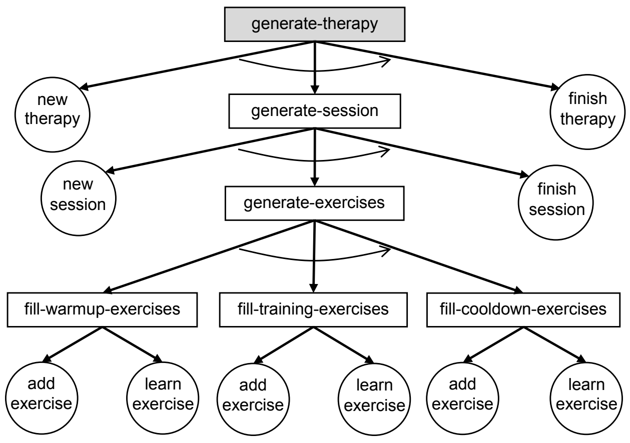


Figure 2: Hierarchical Task Network model schema.

Planning Problem

Goals As shown in Figure 2, the goal of the Hierarchical Task Network is the root level of the tree (generate-therapy). This general task comprises three arguments: number of sessions to plan, duration interval for each session and patient identifier. This task can be refined using the HTN decomposition methods until a set of primitive actions completes the plan, which should reach the TOCLs. These TOCLs are also modelled as numeric predicates in the problem description. Furthermore, with the aim to parametrize the search (time and possible exercises in a phase or session), a set of predicates is also included.

Exercise Database The exercise database has been modelled similarly to how it has been described for the classical planning approach. The only difference is purely technical due to the representation language of the HTN planner used for evaluation, described later.

Planning Domain

The HTN planning domain is organized as shown in Figure 2, where a set of exercises is generated for a number of sessions. This behaviour is modelled as a recursive task (generate-session) which receives the current session number and the total number of sessions as parameters. This current session number (?csn) is used as identifier by the planning domain and increased to generate new sessions.

```
(:method (generate-session ?csn ?tsn)
  ;main
  ((call <= ?csn ?tsn))
      ((!new-session ?csn ?tsn)
      (generate-exercises ?csn)
      (generate-session (call + ?csn 1) ?tsn))
  ;stop
  ((call > ?csn ?tsn))
      nil
 )
```

Each session is divided into three phases modelled as lower-level tasks: warm-up phase, training phase and cool-down phase. The system must distinguish which exercises are appropriate for each phase depending on its features, and decides if learning a new exercise is required during planning time.

Axioms Axioms allow to infer new predicates from the evaluation of a logical expression (abductive inference). We have defined axioms to control the time intervals of phases and to manage which exercises are more appropriate for that phase according to the parameters specified in the planning problem. For example, (cooldown-time) and (cooldown-exercise) in Figure 3 are calls to axioms.

Each session begins and ends with less intense and difficult exercises and the middle of the session consists of greater intensity and difficulty exercises. The expectation for each session (comprising three phases) is that the values of intensity and difficulty follow a Gaussian distribution, as shown later in the empirical evaluation (see Figure 4). Using axioms throughout the planning domain simplifies the modelling process of this requirement.

Tasks and Methods Methods are used to refine compound tasks into lower-level tasks or primitive actions. These methods have a precondition that needs to be fulfilled in order to be applied. In our model, we have used five tasks:

1. (generate-therapy) has a unique method with empty precondition that uses a total-order decomposition to call the lower-level task (generate-session).
2. (generate-session) is modelled as a recursive task that has a method to call lower-level task (generate-exercises) and a "nil" method that stops when the number of sessions required is reached.
3. (generate-exercises) has a unique method with empty precondition that calls a total-ordered sequence of lower-level tasks (one for each phase).
4. (fill-*phase*-exercises) are modelled with three methods. The first one checks a) that the current time is within the phase time interval, b) that the exercise is suitable for the phase and c) that the exercise selected has not been already included in the ongoing generated session plan. Figure 3 shows a high-level description of how the (fill-cooldown-exercises) task has been modelled. The first method uses a "sort-by" function that drives the planner in the order in which the variable bindings will be evaluated for the method precondition. This "sort-by" function calculates an heuristic value (?ht), modelled as follows:

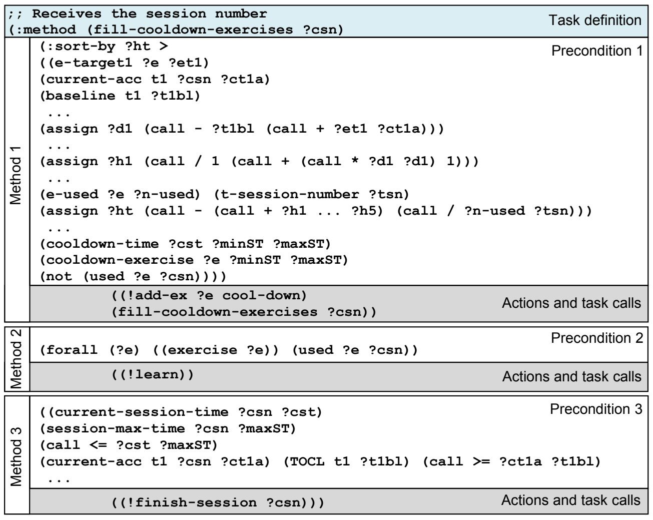
$$ht_{ex} = \sum_{i=1}^{n_{objectives}} \left(\frac{1}{{d_i}^2 + 1} - \frac{ex_{times\_used}}{num_{sessions}}\right) \quad (1)$$

where d_i, for each therapeutic objective i, is the distance (a minus operation) between the current cumulative level (if the exercise would be included) to the desired TOCL for the planned session. So, the function rewards exercises whose contribution minimizes the distance to the frontier solution. The last part of the equation penalizes the number of times an exercise has been previously used.

The second method is applied when all the possible exercises have been already included in a session, so there is no available exercises to add. In this case a new exercise needs to be acquired ("learn" action) from the therapist.

Exercises will be added taking into account the heuristic and recursive calls to (fill-cooldown-exercises) and it will be carried out till the preconditions fails. In this last case, the third method precondition is evaluated (TOCL reached within the maximum session time specified); if it is fulfilled, the plan is valid. Otherwise the planner will do backtracking to check other exercise sets, until this condition is reached.

Primitive Actions We use dummy actions to delimit start and end of sessions and therapy (see Figure 2). The action to add an exercise updates the current session time (adding the exercise duration) and the current cumulative level for the therapeutic objectives in that session. It also updates the status of the exercise to "used" and the counter of times used. At the time of writing this paper, the "learn" action establishes fixed values for the exercise attributes. Improving this behaviour is subject of future work.


```
;; Receives the session number                              Task definition
(:method (fill-cooldown-exercises ?csn)
  (:sort-by ?ht >                                            Precondition 1
  ((e-target1 ?e ?et1)
  (current-acc t1 ?csn ?ct1a)
  (baseline t1 ?t1bl)
   ...
  (assign ?d1 (call - ?t1bl (call + ?et1 ?ct1a)))
   ...
  (assign ?h1 (call / 1 (call + (call * ?d1 ?d1) 1)))
   ...
  (e-used ?e ?n-used) (t-session-number ?tsn)
  (assign ?ht (call - (call + ?h1 ... ?h5) (call / ?n-used ?tsn)))
   ...
  (cooldown-time ?cst ?minST ?maxST)
  (cooldown-exercise ?e ?minST ?maxST)
  (not (used ?e ?csn))))
          ((!add-ex ?e cool-down)                    Actions and task calls
          (fill-cooldown-exercises ?csn))

                                                             Precondition 2
  (forall (?e) ((exercise ?e)) (used ?e ?csn))
          ((!learn))                                 Actions and task calls

  ((current-session-time ?csn ?cst)                          Precondition 3
  (session-max-time ?csn ?maxST)
  (call <= ?cst ?maxST)
  (current-acc t1 ?csn ?ct1a) (TOCL t1 ?t1bl) (call >= ?ct1a ?t1bl)
   ...
          ((!finish-session ?csn)))                  Actions and task calls
```

Figure 3: JSHOP2 code for the task to include a set of exercises in the cool-down phase of a session.

Planning Strategy

Our hypothesis states that a well modelled hierarchical representation of the domain knowledge, along with parameters to drive the search appropriately, could generate successful solutions with a improved quality. In other words, we look for a parametrized design to provide a flexible configuration to the physicians. Moreover, in order to reflect the medical criteria in the resulting plan, the heuristic function is in charge of the exercises selection. As explained before, this function is also used to penalize the most repetitive exercises reducing the heuristic value, but this does not avoid the occurrence of the same exercise throughout sessions. That is why we consider the variability as soft constraint.

The HTN approach can search towards reaching general therapeutic objectives that imply interactions among sessions. These interactions can occur due to a) the exercise distribution of previous ongoing planned sessions that could affect to future ones, b) the TOCLs of subsequent sessions can be updated by the plan of earlier sessions and c) chasing possible distributions (eg. time, intensity, TOCLs) for the whole therapy predefined by physicians. This is the motivation to propose a recursive model in order to generate multiple sessions. The HTN approach preserves the capability of backtracking through past sessions without mediation of an external program.

Empirical Evaluation

We have used the SHOP2 HTN language (Nau et al. 2003) for modelling the planning domain and JSHOP2[3] to test the plan generation. The SHOP2 language is provided with a great expressiveness that allows axiomatic inference, symbolic and numerical computation, call to external programs and use of conditional quantifiers, to name some features.

In order to evaluate the behaviour of the hierarchical domain, a set of 72 exercises are included in the planning problem. This experiment has been carried out with the following configuration: 30 sessions to generate, 25-30 minutes per session, 20% of the total session time is assigned to each warm-up and cool-down phases and the remainder 60% is for training phase. The established intervals to consider an exercise as a candidate for each phase are: warm-up intensity [0-30], warm-up difficulty [0-20], training intensity [30-50], training difficulty [30-50], cool-down intensity [0-20] and cool-down difficulty [0-30]. It is assumed that an exercise could be considered as warm-up and cool-down according to their values. The effects of the exercise distribution is shown in Figure 4, where an approximate Gaussian distribution of the intensity and difficulty within the phases is achieved.

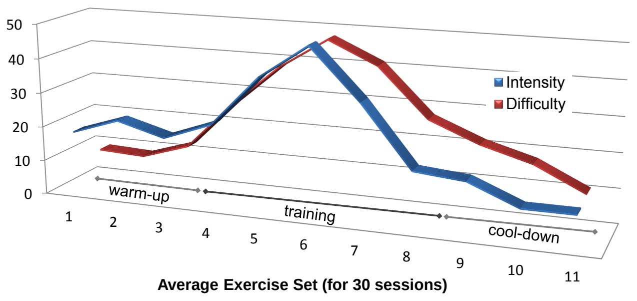


Figure 4: Represents the average values of the intensity and difficulty for 30 generated sessions. Along the different phases (warm-up, training and cool-down), the achieved progress is represented by these Gaussian distributions.

Discussion and Conclusions

To conclude this manuscript we have created a qualitative comparison of the two approaches that highlight the main topics addressed. Table 2 represents a summary of this comparison. Some further take-home messages are described next. Firstly, this work provides an original model based on numeric values of a new kind of problem which is useful to compare capabilities and behaviour among different automated planners. Secondly, we think that more flexibility for the user seems to be available in the HTN model, where more complex expert knowledge could be represented easily, however costs and preferences used in classical planning are very suitable for this problem. Third, with regard to how to achieve variability while trying to fulfil our requirements, we noted that the sort-by function used in JSHOP2 needs to arrange and order many bindings before the task decomposition is applied, which may affect its performance. This could possibly be improved by modelling the heuristic function using a java comparator function, as offered by JSHOP2. Fourth, the divide and conquer strategy used in the classical approach eliminates the backtracking to previous sessions to improve the planning time. On the other hand, the HTN model can generate plans using backtracking among sessions to solve their interactions. Finally, CBP does not use heuristics for fluents, which could be very helpful for this domain to reduce the planning time. OPTIC (Benton, Coles, and Coles 2012) provides this kind of heuristics, so it could be a future option to explore. With regard to temporal constraints, if more complex ones would be needed, the temporal representation provided by PDDL2.1 could be handled through planners like OPTIC (Benton, Coles, and Coles 2012) or the HTN planner SIADEX (Castillo et al. 2006). In the case of HTN, it would be interesting to explore new preference-based planning approaches.

To sum up, we have presented two ways to reach plans driven by general therapeutic objectives modelled numerically. We plan to do a better quantitative comparison in terms of performance, addressing the issues previously discussed.

[3]It uses a planning compilation technique to synthesize domain-dependent planners from SHOP2 domain descriptions.

Acknowledgements

This work has been partially granted by the Spanish Ministerio de Economía y Competitividad (MINECO) funds under coordinated project no. TIN2012-38079-C03-01, TIN2012-38079-C03-02 and TIN2012-38079-C03-03, and FEDER INNTERCONECTA ADAPTA 2012. We also are grateful to the medical team of the Virgen del Rocío University Hospital for their participation.

Constraints, Requirements	**Classical Planning**	**HTN Planning**
Assuring Variability	• Avoids repeating exercises in the same session. • Avoids repeating an exercise in the last three sessions. • In the training phase, an exercise cannot be planned in the same position than in the last occurrence.	• Avoids repeating exercises in the same session. • Heuristic sort-by function penalized for exercises that occur repeatedly.
Phase Selection	PDDL predicate relating each exercise to its corresponding phase.	Axioms limiting the exercise selection whose minimum and maximum duration, intensity and difficulty can be defined by physicians for each phase.
Phase Time Intervals	It is parametrized through the minimum and maximum duration for each phase, which is assigned by the medical expert.	Time is parametrized through axioms according to accumulated percentage of total time for each phase (eg. 0.2, 0.7, 1.0).
Learning new exercise	Suggests which attribute values should have a new learnt exercise, preferring exercises which can improve the session variability.	It does not suggest the minimum values yet, but it is already modelled as a new HTN method that can add new exercises during planning time.
Achieving Goals	• The automated planner has to achieve the TOCL thresholds which are the goals established in the planning problem, while observing all the constraints set by the physician. • Minimizes the total cost of the plan, where learning a new exercise has more cost than use one from the database.	• Total-order hierarchical network expressing the three phases. TOCLs and expected session time should be reached, otherwise backtracking occurs to find a suitable exercise set. • Driving exercise selection through a sort-by function (see description above).
Planning Multiple Sessions	Divide and conquer strategy which calls the planner one time per session, improving planification time without affecting the quality of the plan thanks to the learning actions.	HTN Planning is done as usual in one run, doing backtracking when exercise sets dont reach expected goals.

Table 2: Qualitative comparison of Classical and HTN approaches for the presented problem.

References

Ahmed, S.; Gozbasi, O.; Savelsbergh, M. W. P.; Crocker, I.; Fox, T.; and Schreibmann, E. 2010. An automated intensity-modulated radiation therapy planning system. *INFORMS Journal on Computing* 22(4):568–583.

Benton, J.; Coles, A. J.; and Coles, A. I. 2012. Temporal planning with preferences and time-dependent continuous costs. In *Proceedings of the Twenty Second International Conference on Automated Planning and Scheduling (ICAPS-12)*.

Calderita, L.; Bustos, P.; Suarez Mejias, C.; Fernandez, F.; and Bandera, A. 2013. Therapist: Towards an autonomous socially interactive robot for motor and neurorehabilitation therapies for children. In *7th International Conference on PervasiveHealth*, 374–377.

Castillo, L. A.; Fernández-Olivares, J.; Garcia-Perez, O.; and Palao, F. 2006. Efficiently handling temporal knowledge in an htn planner. In *ICAPS*, 63–72.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. HTN planning: Complexity and expressivity. In *AAAI*, volume 94, 1123–1128.

Fdez-Olivares, J.; Castillo, L. A.; Cózar, J. A.; and García-Pérez, Ó. 2011. Supporting clinical processes and decisions by hierarchical planning and scheduling. *Computational Intelligence* 27(1):103–122.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *J. Artif. Intell. Res.(JAIR)* 20:61–124.

Fuentetaja, R.; Borrajo, D.; and López, C. L. 2010. A look-ahead B&B search for cost-based planning. In *Current Topics in Artificial Intelligence*. Springer. 201–211.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated planning: theory & practice*. Elsevier.

González-Ferrer, A.; Ten Teije, A.; Fdez-Olivares, J.; and Milian, K. 2013. Automated generation of patient-tailored electronic care pathways by translating computer-interpretable guidelines into hierarchical task networks. *Artificial Intelligence in Medicine* 57(2):91–109.

Morignot, P.; Soury, M.; Leroux, C.; Vorobieva, H.; and Hède, P. 2010. Generating scenarios for a mobile robot with an arm: Case study: Assistance for handicapped persons. In *Proceedings of 11th International Conference on Control Automation Robotics & Vision*, 976–981. IEEE.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *J. Artif. Intell. Res.(JAIR)* 20:379–404.

Peleg, M. 2013. Computer-interpretable clinical guidelines: A methodological review. *Journal of Biomedical Informatics* 46(4):744–763.

Schimmelpfeng, K.; Helber, S.; and Kasper, S. 2012. Decision support for rehabilitation hospital scheduling. *OR spectrum* 34(2):461–489.

Turner-Stokes, L. 2009. Goal attainment scaling (GAS) in rehabilitation: a practical guide. *Clinical rehabilitation* 23(4):362–70.

Knowledge Engineering for Planning-Based Hypothesis Generation

Shirin Sohrabi Octavian Udrea Anton V. Riabov
IBM T.J. Watson Research Center
PO Box 704, Yorktown Heights, NY 10598, USA
{ssohrab, oudrea, riabov}@us.ibm.com

Abstract

In this paper, we address the knowledge engineering problems for hypothesis generation motivated by applications that require timely exploration of hypotheses under unreliable observations. We looked at two applications: malware detection and intensive care delivery. In intensive care, the goal is to generate plausible hypotheses about the condition of the patient from clinical observations and further refine these hypotheses to create a recovery plan for the patient. Similarly, preventing malware spread within a corporate network involves generating hypotheses from network traffic data and selecting preventive actions. To this end, building on the already established characterization and use of AI planning for similar problems, we propose use of planning for the hypothesis generation problem. However, to deal with uncertainty, incomplete model description and unreliable observations, we need to use a planner capable of generating multiple high-quality plans. To capture the model description we propose a language called LTS++ and a web-based tool that enables the specification of the LTS++ model and a set of observations. We also proposed a 9-step process that helps provide guidance to the domain expert in specifying the LTS++ model. The hypotheses are then generated by running a planner on the translated LTS++ model and the provided trace. The hypotheses can be visualized and shown to the analyst or can be further investigated automatically.

Introduction

Several application scenarios require the construction of hypotheses presenting alternative explanation of a sequence of possibly unreliable observations. For example, the evolution of the state of the patient over time in an Intensive Care Unit (ICU) of a hospital can be inferred from a variety of measurements. Similarly, observations from network traffic can indicate possible malware. The hypotheses, represented as a sequence of changes in patient state, aim to present an explanation for these observations, while providing deeper insight into the actual underlying causes for these observations, helping to make decisions about further testing, treatment or other actions.

Expert judgment is the primary method used for generating hypotheses and evaluating their plausibility. Automated methods have been proposed, to assist the expert, and help improve accuracy and scalability. Notably, model-based diagnosis methods can determine whether observations can be explained by a model (e.g., (Cassandras and Lafortune 1999; Sampath et al. 1995)). Recently, several researchers have proposed use of automated planning technology to address several related class of problems including diagnosis (e.g., (Sohrabi, Baier, and McIlraith 2010; Haslum and Grastien 2011)), plan recognition (Ramírez and Geffner 2009), and finding excuses (Göbelbecker et al. 2010). These problems share a common goal of finding a sequence of actions that can explain the set of observations given the model-based description of the system. However, most of the existing literature make an assumption that the observations are all perfectly reliable and should be explainable by the system description, otherwise no solution exists for the given problem. But that is not true in general. For example, even though observations resulting from the analysis of network data can be unreliable, we would still like to explain as many observations as possible with respect to our model; as a further complication, we cannot assume the model is complete.

In 2011, Sohrabi et al. established a relationship between generating explanations, a more general form of diagnosis, and a planning problem (Sohrabi, Baier, and McIlraith 2011). Recently, we extend this work to address unreliable observations and showed how to generate multiple high-quality plans or the plausible hypotheses. (Sohrabi, Udrea, and Riabov 2013)[1]. In this paper, we address knowledge engineering problems of capturing the domain knowledge.

To capture the model description we propose a language called LTS++, derived from LTS (Labeled Transition System) (Magee and Kramer 2006) for defining models for hypothesis generation, and associating observation types. LTS++ is less expressive than the general Planning Domain Definition Language (PDDL) specification of a planning problem (McDermott 1998). However, in our experience, the domain expert finds writing an LTS++ language much simpler than PDDL. To further help the domain expert, we also proposed a process that helps provide guidance in specifying the LTS++ model. Additionally, we developed a web-based tool that enables the specification of the LTS++ model and a set of observations. Our tool features syntax highlighting, error detection, and visualization of the state transition

[1]We include the description of our approach to hypothesis generation from that prior work, but LTS++ and our tools for knowledge engineering have not been previously described.

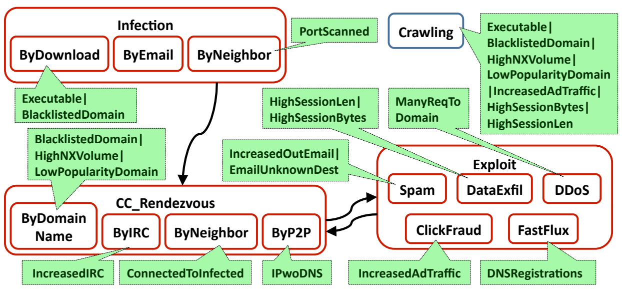

Infection
ByDownload
ByEmail
ByNeighbor
PortScanned
Crawling
Executable|BlacklistedDomain|HighNXVolume|LowPopularityDomain|IncreasedAdTraffic|HighSessionBytes|HighSessionLen
Executable|BlacklistedDomain
HighSessionLen|HighSessionBytes
ManyReqTo Domain
BlacklistedDomain|HighNXVolume|LowPopularityDomain
IncreasedOutEmail|EmailUnknownDest
Exploit
Spam
DataExfil
DDoS
CC_Rendezvous
ByDomain Name
ByIRC
ByNeighbor
ByP2P
ClickFraud
FastFlux
IncreasedIRC
ConnectedToInfected
IPwoDNS
IncreasedAdTraffic
DNSRegistrations

Figure 1 (a): Malware detection

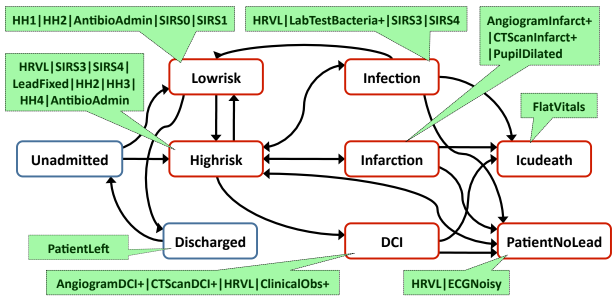

HH1|HH2|AntibioAdmin|SIRS0|SIRS1
HRVL|LabTestBacteria+|SIRS3|SIRS4
AngiogramInfarct+|CTScanInfarct+|PupilDilated
HRVL|SIRS3|SIRS4|LeadFixed|HH2|HH3|HH4|AntibioAdmin
Lowrisk
Infection
Unadmitted
Highrisk
Infarction
Icudeath
FlatVitals
Discharged
DCI
PatientNoLead
PatientLeft
AngiogramDCI+|CTScanDCI+|HRVL|ClinicalObs+
HRVL|ECGNoisy

Figure 1 (b): Intensive care

graph. The hypotheses are then generated by running a planner on the translated LTS++ model and the provided observation trace. The hypotheses can be visualized and shown to the analyst or can be further investigated automatically.

In the rest of the paper, we will first describe our two application examples in detail. We then describe the architecture of our automated hypothesis exploration problem in which hypothesis generation plays a key role. Then we describe the relationship between planning and hypothesis generation, which facilitates the use of planning technology. We show our initial experimental results in using planning. We then describe our LTS++ language, the creation process, LTS++ IDE, and show several example problems.

Application Description

In this section we introduce two example applications that illustrate our approach: intensive care delivery and malware detection. A key characteristic of these applications is that the true state of monitored patients, network hosts, or other entities, while essential for timely detection and prevention of critical conditions, is not directly observable. Instead, we must analyze the sequence of available observations to reconstruct the state. To make this possible, our approach relies on a model of the entity consisting of states, transitions between states, and many-to-many correspondence between states and observations. In the following sections we will describe how these models can be created by the domain experts and encoded in our LTS++ language.

Figure 1 shows state transition systems of intensive case and malware detection. The rounded rectangles are states. The states are associated with a type, good or bad, and drawn in blue or red respectively. The callouts are observations associated with these states. Note that the observations are obtained by analyzing raw data gathered through sensors.

In Figure 1 (a), the bad state correspond to malware lifecycle, such as the host becoming infected with malware, the bot's rendezvous with a Command and Control (C&C) machine (*botmaster*), and a number of exploits – uses of the bot for malicious activity. Each of the states can be achieved in many ways, depending on the type and capabilities of the malware. For example, the *CC_Rendezvous* state can be achieved by attempting to contact an Internet domain, or via Internet Relay Chat (IRC) on a dedicated channel. The good state in Figure 1 (a) corresponds to a "normal" lifecycle of a web crawler compressed into a single state. Note that crawler behavior can also generate a subset of the observations that malware may generate. The callouts are the observations associated with states. For example, the observation *HighNXVolume* is an observation associated with the *ByDomainName* state that corresponds to an abnormally high number of *domain does not exist* responses for Domain Name System (DNS) queries; such an observation may indicate that the bot and its botmaster are using a domain name generation algorithm, and the bot is testing generated domain names trying to find its master.

In Figure 1 (b), the bad states correspond to critical states of a patient such as *Infection*, *DCI*, or *Highrisk*. The good states are the non-critical states. Upon admission the patient is either in *Lowrisk* or in *Highrisk*. From a *Highrisk* state, they may get to the *Infection*, *Infarction*, or the *DCI* state. From *Lowrisk* they may get to the *Highrisk* state or be *Discharged* from ICU. The patient enters *Icudeath* from *Infection*, *Infarction*, or *DCI* state. The patient's condition may improve; hence the patient's state may move back to the *Lowrisk* state from for example the *Infection* state. The observations are measured based on the raw data captured by patient monitoring devices (e.g., the patient's blood pressure, heat rate, temperature) as well as other measurements and computations provided by doctors and nurses. For example, given the patient's heart rate, their blood pressure, and their temperature, which are measured continuously, their SIRS score can be computed, producing an integer between 0 to 4. Similarly, a result of CT Scan, or a lab test will indicate other possible observations about the patient.

While the complexity of the analysis involved to obtain one observation can vary, it is important to note that observations are by nature *unreliable*:

The set of observations will be incomplete. Operational constraints will prevent us running in-depth analysis on *all* of the data all the time. However, all observations are typically time stamped, and hence totally ordered.

Observations may be ambiguous. This is depicted in Figure 1, where for instance contacting a blacklisted domain may be evidence of malware activity, or maybe a crawler that reaches such a domain during normal navigation. Similarly, Heart Rate Variability Low (HRVL) may be explained by many states such as *DCI* or *Highrisk* or *Infection*.

Not all observations will be explainable. There are several reasons while some observations may remain unexplained: (i) observations are (sometimes weak) indicators of a behavior, rather than authoritative measurements; (ii) the model description is by necessity incomplete, unless we are able to design a perfect model; (iii) in the case of malware detection, malware could try to confuse detectors by either hiding in normal traffic patterns or originating extra traffic.

For Figure 1 (a) one can consider the following two observations for a host: (o_1) a download from a blacklisted domain and (o_2) an increase in traffic with ad servers. Note that according to Figure 1 (a), this sequence could be explained by two hypotheses: (a) a crawler or (b) infection by downloading from a blacklisted domain, a C&C rendezvous which we were unable to observe, and an exploit involving click fraud. In such a setting, it is normal to believe (a) is more plausible than (b) since we have no evidence of a C&C rendezvous taking place. However, take the sequence (o_1) followed by (o_3) an increase in IRC traffic followed by (o_2). In this case, it is reasonable to believe that the presence of malware – as indicated by the C&C rendezvous on IRC – is more likely than crawling, since crawlers do not use IRC. The crawling hypothesis cannot be completely discarded since it may well be that a crawler program is running in background, while a human user is using IRC to chat.

Consider the following observation sequence for the model in Figure 1 (b): *HH3*, *HRVL*. This denotes a patient with a Hunt and Hess (a grading system used to classify the severity of subarachnoid hemorrhage) score of 3, followed by *HRVL*. Since *HRVL* is an ambiguous observation – i.e., can be indicative of multiple states –, equally plausible hypotheses may be:

Unadmitted $\rightarrow$ *Highrisk* or
Unadmitted $\rightarrow$ *Highrisk* $\rightarrow$ *PatientNoLead* or
Unadmitted $\rightarrow$ *Highrisk* $\rightarrow$ *Infarction* or
Unadmitted $\rightarrow$ *Highrisk* $\rightarrow$ *DCI*.

Note, although the current state of the patient is unknown, the generated hypotheses indicate that it is one of *Highrisk*, *PatientNoLead*, *Infarction* or *DCI*.

Given a sequence of observations and the model, the hypothesis generation task infers a number of plausible hypotheses about the evolution of the entity. Practically, we have to analyze multiple hypotheses about an entity because the state transition model may be incomplete or the observations may be unreliable. The result of our automated technique can then be presented to a network administrator (or to a doctor) or to an automated system for further investigation and testing. Next, we will describe briefly all the necessary components for hypothesis exploration.

Architecture

Our work on automated exploration of hypotheses focuses on the Hypothesis Generation, which is part of a larger automated data analysis system that includes sensors, actuators, multiple analytic platforms and a *Tactical Planner*. Tactical Planner, for the purpose of this paper, should be viewed as a component responsible for execution of certain strategic actions and it can be implemented using, for example, a classical planner to compose analytics (Bouillet et al. 2009). A high-level overview of the complete system architecture is shown in Figure 2. All components of the architecture, with the exception of application-specific analytics, sensors, and actuators, are designed to be reused without modification in a variety of application domains.

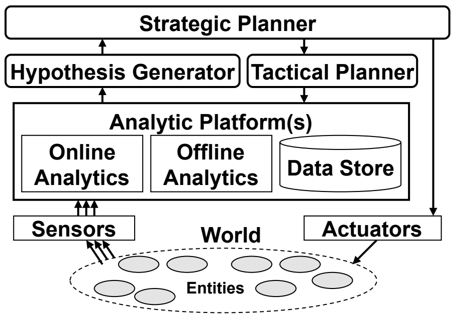


Figure 2: System architecture

The system receives input from *Sensors*, and *Analytics* translate sensor data to observations. The *Hypothesis Generator* interprets the observations received from analytics, and generates hypotheses about the state of *Entities* in the *World*. Depending on application domain, the entities may correspond to patients in a hospital, or to computers connected to a corporate network, or other kinds of objects of interest. The *Strategic Planner* evaluates these hypotheses and initiates preventive or testing actions in response. Some of the testing actions can be implemented as additional processing of data, setting new goals for the Tactical Planner, which composes and deploys analytics across multiple *Analytic Platforms*. A Hadoop cluster, for example, can be used as an analytic platform for offline analysis of historical data accumulated in one or more *Data Stores*. Alternatively, a Stream Computing cluster can be used for fast online analysis of new data received from the sensors.

Preventive actions, as well as some of the testing actions, are dispatched to *Actuators*. There is no expectation that every actuation request will succeed, or always happen instantaneously. Actuation in a hospital setting can involve dispatching alerts to doctors, or lab test recommendations.

Hypothesis Generation via Planning

In this section, we define the hypothesis generation problem and describe its relationship to planning. We also provide experimental evaluation that supports the premise of using planning for generating multiple plausible hypotheses. In the next section, we will describe how the planning model can be captured using the LTS++ language which we translate to a planning problem. Our tool, LTS++ hypothesis generator, then uses a planning to compute plausible hypothesis and present them to the user.

Following our recent work (Sohrabi, Udrea, and Riabov 2013), a *dynamical system* is defined as $\Sigma = (F, A, I)$, where F is a finite set of fluent symbols, A is a set of actions with preconditions and effects that describes actions that account for the possible transitions of the state of the entity (e.g., patient or host) as well as the *discard* action that addresses unreliable observations by allowing observations to be unexplained, and I is a clause over F that defines the initial state. The instances of the *discard* action add transitions to the system that account for leaving an observation unexplained. The added transitions ensure that we took all observations into account, but an instance of the *discard*

	Hand-crafted		10 states		50 states		100 states	
Observations	% Solved	Time	% Solved	Time	% Solved	Time	% Solved	Time
5	100%	2.49	70%	0.98	80%	5.61	30%	14.21
10	100%	2.83	90%	2.04	50%	25.09	30%	52.63
20	90%	12.31	70%	24.46	-	-	-	-
40	70%	3.92	40%	81.11	-	-	-	-
60	60%	6.19	-	-	-	-	-	-
80	50%	8.19	-	-	-	-	-	-
100	60%	11.73	10%	10.87	-	-	-	-
120	70%	20.35	20%	15.66	-	-	-	-

Table 1: The percentage of problems where the ground truth was generated, and the average time spent for LAMA.

action for a particular observation o indicates that o is not explained. Actions can be over both "good" and "bad" behaviors. This maps to "good" and "bad" states of the entity, different from a system state (i.e., set of fluents over F).

An observation formula φ is a sequence of fluents in F we refer to as *trace*. Given a trace φ, and the system description Σ, a hypothesis α is a sequence of actions in A such that α satisfies φ in the system Σ. We also define a notion of plausibility of a hypothesis. Given a set of observations, there are many possible hypotheses, but some could be stated as more plausible than others. For example, since observations are not reliable, the hypothesis α can explain a subset of observations by including instances of the *discard* action. However, we can indicate that a hypothesis that includes the minimum number of *discard* actions is more plausible. In addition, observations can be ambiguous: they can be explained by instances of "good" actions as well as "bad" actions. Similar to the diagnosis problem, a more plausible hypothesis ideally has the minimum number of "bad" or "faulty" actions. More formally, given a system Σ and two hypotheses α and α' we assume that we can have a reflexive and transitive plausibility relation $\preceq$, where $\alpha \preceq \alpha'$ indicates that α is at least as plausible as α'.

The **hypothesis generation problem** is then defined as $P = (F, A', I, \varphi)$ where A' is the set A with the addition of positive action costs that accounts for the plausibility relation $\preceq$. A hypothesis is a plan for P, and the most plausible hypothesis is the minimum cost plan. That is, if α and α' are two hypotheses, where α is more plausible than α', then $cost(\alpha) < cost(\alpha')$. Therefore, the most plausible hypothesis is the minimum cost plan.

While some class of plausibility relation can be expressed as Planning Domain Definition Language (PDDL3) (Gerevini et al. 2009) preferences, cost-based planners are (currently) more advanced than PDDL3-based planners, and so the technique proposed by Keyder and Geffner 2009 can be used to compile preferences into costs, enabling the use of cost-based planners instead.

Computing Plausible Hypotheses

To address uncertainty, the unreliability of observations and incomplete model description, we must generate multiple high-quality (or low-cost) plans that correspond to a set of plausible hypothesis. To this end, we adapt our implementation of hypothesis generation from (Sohrabi, Udrea, and Riabov 2013). We encode the plausibility notion as actions costs. In particular, we assign a high cost to the *discard* action in order to encourage explaining more observations. In addition, we assign a higher cost to all instances of the actions that represent "bad" behaviors than those that represent "good" behaviors. Furthermore, shorter/simpler plans are assumed to be more plausible. To address observations, we similarly compile them away in our encoding following a technique proposed in (Haslum and Grastien 2011).

The planning problem is described in PDDL. We used one fixed PDDL encoding of the domain, but varied the problem for each problem description, which we generate automatically in our experiments. We also developed a replanning process around LAMA (Richter and Westphal 2010) to generate multiple high-quality (or low-cost) plans that correspond to a set of plausible hypothesis. The replanning process works in such a way that after each round, the planning problem is updated to disallow finding the same set of plans in future runs of LAMA. This process continues until a time limit is reached and then all found plans are sorted by cost and shown to the user by our tool.

Experimental Evaluation

The experiments we describe in this section help evaluate the response time and the accuracy of our approach. In particular, these experiments show promise of our approach in terms of using planning. This experiments were reported in (Sohrabi, Udrea, and Riabov 2013). We evaluated performance by using both a hand-crafted description of the malware detection problem and a set of automatically generated state transition systems with 60% bad and 40% good states.

To evaluate performance, we introduce the notion of *ground truth*. In all experiments, the problem instances are generated by constructing a ground truth trace by traversing the lifecycle graph (similar to Figure 1 (a)) in a random walk, adding with small probability, missing and inconsistent observations. We then measure performance by comparing the generated hypotheses with the ground truth, and consider a problem *solved* for our purposes if the ground truth appears among the generated hypotheses.

For each size of the problem, we have generated 10 problem instances, and the measurements we present are averages. The measurements were done on a dual-core 3 GHz Intel Xeon processor and 8 GB memory, running 64-bit RedHat Linux. We used a 300 seconds time limit.

Table 1 summarizes the result. The rows and the columns indicate the problem size, measured by the number of observations and the number of states. The hand-crafted column, is the example shown in Figure 1 (a), which has 18 states. The generated problems consisted of 10, 50 and 100 states. The *% Solved* column shows the percentage of problems

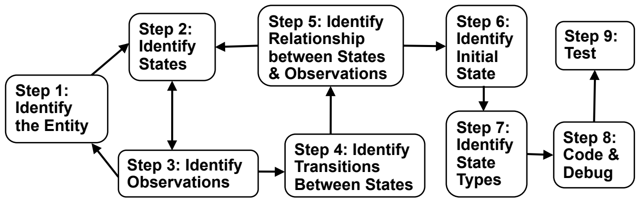

Step 1: Identify the Entity
Step 2: Identify States
Step 3: Identify Observations
Step 4: Identify Transitions Between States
Step 5: Identify Relationship between States & Observations
Step 6: Identify Initial State
Step 7: Identify State Types
Step 8: Code & Debug
Step 9: Test

Figure 3: Process for LTS++ model creation

where the ground truth was among the generated plans. The *Time* column shows the average time it took from the beginning of iterations to find the ground truth solution for the solved problems. The dash entries indicate that the ground truth was not found within the time limit.

The results show that planning can be used successfully to generate hypotheses for malware detection, even in the presence of unreliable observations, especially for smaller size problems. The correct hypothesis was generated in most experiments with up to 10 observations. However, in some of the larger instances LAMA could not find any plans. Moreover, in the smaller size problems, more replanning rounds is done within the time limit and hence more distinct plans are generated which increases the chance of finding the ground truth. The results for the hand-crafted malware example also suggest that the problems arising in practice may be easier than randomly generated ones, which had more state transitions and higher branching factor.

We believe that LAMA would have had a better chance of detecting the ground truth trace if instead of finding a set of high-quality plans it could have generated the top k plans, where k could be determined based on a particular scenario. In future work, we plan to evaluate our approach using a planner capable of finding top k plans. Nevertheless, the experiments support our findings, namely, that the use of planning is promising.

LTS++ Model

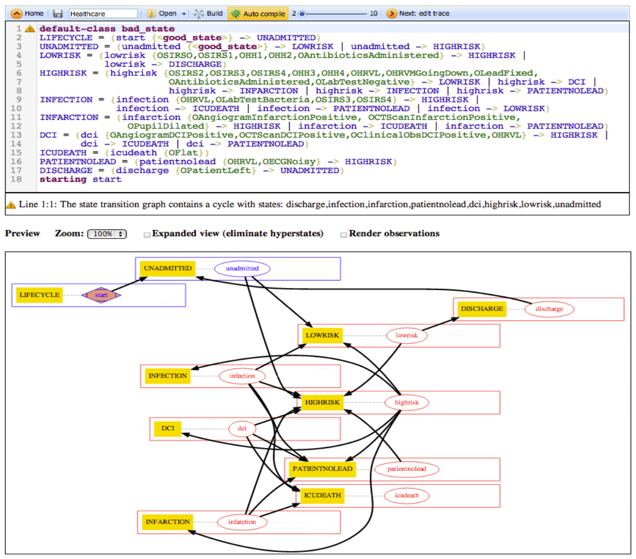
To help new users, we have built a web-based tool for generating hypotheses and developing state transition models, which we use in our experiments and applications. In particular, we have designed a language called LTS++, derived from LTS (Labeled Transition System) (Magee and Kramer 2006), for defining models for hypothesis generation, and associating observation types with states. In this section, we describe a process that the user or the domain expert might undergo in order to define an LTS++ model. We will also describe the LTS++ IDE and the LTS++ syntax.

Steps in Creating an LTS++ Model

Figure 3 shows a 9-step creation process for an LTS++ model. The arrows are intended to indicate the most typical transitions between steps: transitions that are not shown are not prohibited. This process is meant to help provide guidance to the new users in developing an LTS++ model. In this section, we will go over the first 7 steps.

In step 1, the user needs to identify the entity. This may depend on the objective of the hypothesis generator, the available data, and the available actions. For example, in the malware detection problem, the entity is the host, while in the intensive care delivery problem the entity is the patient. In step 2, the domain expert identifies the states of the entity. As we saw in the application section, the states of patient for example could be for example *DCI*, *Infection*, and *Highrisk*. Since the state transition model is manually specified and contains a fixed set of observation types, while potentially trying to model an open world with an unlimited number of possible states and observations, the model can be incomplete at any time, and may not support precise explanations for all observation sequences. To address this on the modeling side, and provide feedback to model designers about states that may need to be added, we have introduced a hierarchical decomposition of states. In some configurations, the algorithm allows designating a subset of the state transition system as a *hyperstate*. In this case, if a transition through one or several states of the hyperstate is required, but no specific observation is associated with the transition, the hyperstate itself is included as part of the hypothesis, indicating that the model may have a missing state within the hyperstate, and that state in turn may need a new observation type associated with it. In the malware detection problem, the *Infection*, *Exploit*, *CC_rendezvous* are the hyperstates.

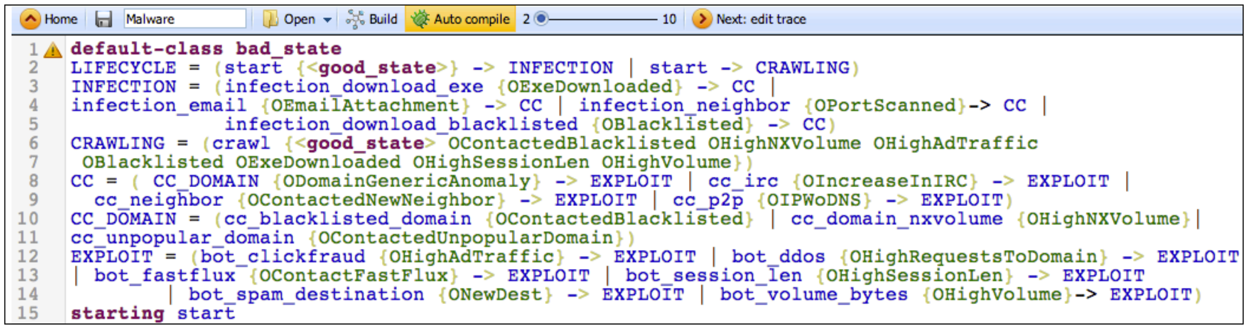
The user needs to identify a set of observations for the particular problem; this is done in step 3. The available data, the entity, and the identified states may help define and restrict the space of observations. In step 4, the domain expert has to find out all possible transitions between states. This may be a tedious task, depending on the number of states. However, one can use hyperstates to help manage these transitions. Any transition of the hyper states is carried out to its substates. In step 5, the user has to associate observations to states. This associations is shown in Figure 1 using the green callouts. In step 6, one can optionally designate a state as the starting state. The domain expert can also create a separate starting state that indicates a one of notation by transitioning to multiple states. For example, in the malware detection



```
default-class bad_state
LIFECYCLE = (start {<good_state>} -> UNADMITTED)
UNADMITTED = (unadmitted {<good_state>} -> LOWRISK | unadmitted -> HIGHRISK)
LOWRISK = (lowrisk {OSIRSO,OSIRS1,OHH1,OHH2,OAntibioticsAdministered} -> HIGHRISK |
           lowrisk -> DISCHARGE)
HIGHRISK = (highrisk {OSIRS2,OSIRS3,OSIRS4,OHH3,OHH4,OHRVL,OHRVMGoingDown,OLeadFixed,
                      OAntibioticsAdministered,OLabTestNegative} -> LOWRISK | highrisk -> DCI |
                      highrisk -> INFARCTION | highrisk -> INFECTION | highrisk -> PATIENTNOLEAD)
INFECTION = (infection {OHRVL,OLabTestBacteria,OSIRS3,OSIRS4} -> HIGHRISK |
             infection -> ICUDEATH | infection -> PATIENTNOLEAD | infection -> LOWRISK)
INFARCTION = (infarction {OAngiogramInfarctionPositive, OCTScanInfarctionPositive,
               OPupilDilated} -> HIGHRISK | infarction -> ICUDEATH | infarction -> PATIENTNOLEAD)
DCI = (dci {OAngiogramDCIPositive,OCTScanDCIPositive,OClinicalObsDCIPositive,OHRVL} -> HIGHRISK |
       dci -> ICUDEATH | dci -> PATIENTNOLEAD)
ICUDEATH = (icudeath {OFlat})
PATIENTNOLEAD = (patientnolead {OHRVL,OECGNoisy} -> HIGHRISK)
DISCHARGE = (discharge {OPatientLeft} -> UNADMITTED)
starting start
```

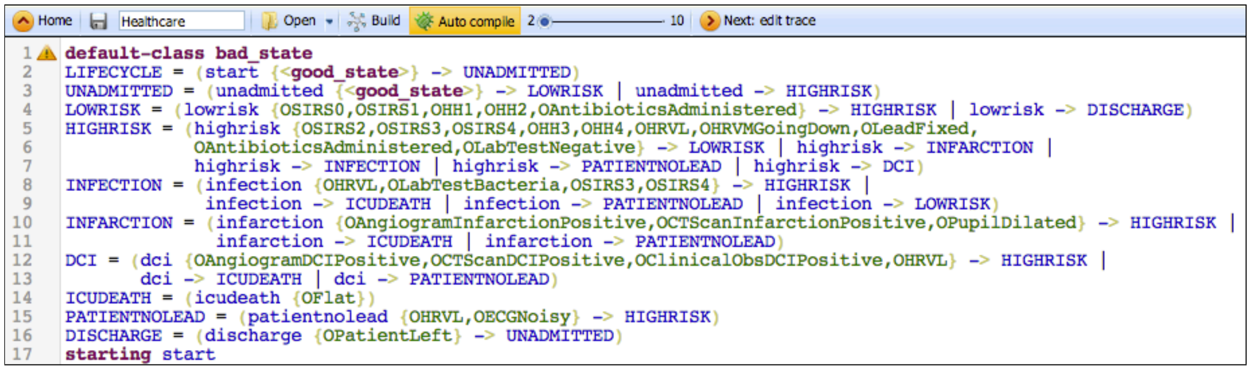
```
Line 1:1: The state transition graph contains a cycle with states: discharge,infection,infarction,patientnolead,dci,highrisk,lowrisk,unadmitted
```

Figure 4: LTS++ IDE



```
 1 default-class bad_state
 2 LIFECYCLE = (start {<good_state>} -> INFECTION | start -> CRAWLING)
 3 INFECTION = (infection_download_exe {OExeDownloaded} -> CC |
 4 infection_email {OEmailAttachment} -> CC | infection_neighbor {OPortScanned}-> CC |
 5              infection_download_blacklisted {OBlacklisted} -> CC)
 6 CRAWLING = (crawl {<good_state> OContactedBlacklisted OHighNXVolume OHighAdTraffic
 7  OBlacklisted OExeDownloaded OHighSessionLen OHighVolume})
 8 CC = ( CC_DOMAIN {ODomainGenericAnomaly} -> EXPLOIT | cc_irc {OIncreaseInIRC} -> EXPLOIT |
 9   cc_neighbor {OContactedNewNeighbor} -> EXPLOIT | cc_p2p {OIPWoDNS} -> EXPLOIT)
10 CC_DOMAIN = (cc_blacklisted_domain {OContactedBlacklisted} | cc_domain_nxvolume {OHighNXVolume}|
11 cc_unpopular_domain {OContactedUnpopularDomain})
12 EXPLOIT = (bot_clickfraud {OHighAdTraffic} -> EXPLOIT | bot_ddos {OHighRequestsToDomain} -> EXPLOIT
13 | bot_fastflux {OContactFastFlux} -> EXPLOIT | bot_session_len {OHighSessionLen} -> EXPLOIT
14         | bot_spam_destination {ONewDest} -> EXPLOIT | bot_volume_bytes {OHighVolume}-> EXPLOIT)
15 starting start
```

Figure 5: LTS++ model for malware detection



```
 1 default-class bad_state
 2 LIFECYCLE = (start {<good_state>} -> UNADMITTED)
 3 UNADMITTED = (unadmitted {<good_state>} -> LOWRISK | unadmitted -> HIGHRISK)
 4 LOWRISK = (lowrisk {OSIRS0,OSIRS1,OHH1,OHH2,OAntibioticsAdministered} -> HIGHRISK | lowrisk -> DISCHARGE)
 5 HIGHRISK = (highrisk {OSIRS2,OSIRS3,OSIRS4,OHH3,OHH4,OHRVL,OHRVMGoingDown,OLeadFixed,
 6             OAntibioticsAdministered,OLabTestNegative} -> LOWRISK | highrisk -> INFARCTION |
 7             highrisk -> INFECTION | highrisk -> PATIENTNOLEAD | highrisk -> DCI)
 8 INFECTION = (infection {OHRVL,OLabTestBacteria,OSIRS3,OSIRS4} -> HIGHRISK |
 9              infection -> ICUDEATH | infection -> PATIENTNOLEAD | infection -> LOWRISK)
10 INFARCTION = (infarction {OAngiogramInfarctionPositive,OCTScanInfarctionPositive,OPupilDilated} -> HIGHRISK |
11               infarction -> ICUDEATH | infarction -> PATIENTNOLEAD)
12 DCI = (dci {OAngiogramDCIPositive,OCTScanDCIPositive,OClinicalObsDCIPositive,OHRVL} -> HIGHRISK |
13        dci -> ICUDEATH | dci -> PATIENTNOLEAD)
14 ICUDEATH = (icudeath {OFlat})
15 PATIENTNOLEAD = (patientnolead {OHRVL,OECGNoisy} -> HIGHRISK)
16 DISCHARGE = (discharge {OPatientLeft} -> UNADMITTED)
17 starting start
```

Figure 6: LTS++ model for the intensive care

problem, the starting state "start" indicates a "one of" notation as it transitions to both *Infection* and *Crawling*.

In step 7, the user can specify state types which indicate that some states are more plausible than the others. State types are related to the "good" vs. "bad" behaviors and they influence the ranking between hypotheses. For example, the hypothesis that the host is crawling is more plausible than it being infected, given the same trace, which can be explained by both hypotheses.

LTS++ IDE

LTS++ IDE is a web-based tool that helps users create planning problems by describing LTS++ models and generate hypotheses. LTS++ IDE consists of an LTS++ editor, graphical view of the transition system, specification of the trace, and generation of hypotheses. The tool automatically generates planning problems from the LTS++ specification and entered trace. The generated hypotheses are the result of running a planner and presenting the result from top-most plausible hypothesis to the least plausible hypothesis.

Figure 4 shows the LTS++ IDE. The top part is the LTS++ language editor which allows syntax highlighting and the bottom part is the automatically generated transition graph. The transition graph can be very useful for debugging purposes. LTS++ IDE also features error detection with respect to the LTS++ syntax. The errors and warning signs are shown below the text editor. They too can be used for debugging the model creation as part of step 8.

Figure 5 and 6 shows the LTS++ model for the malware detection and intensive care applications from Figure 1 respectively. The states are shown in blue with hyperstates specified in all caps. The observations are specified within the curly brackets and are shown in green. You can specify multiple observations by using space or comma between observations (see line 6). The state types are specified within angle brackets (see line 2). The transitions between states are specified using arrows. Each transition needs to be specified within a hyperstate. Multiple transitions between states within a hyperstate can be specified using the vertical bar. The default state type is specified in line 1 and the starting state is specified in the last line.

Generating Hypotheses via LTS++ IDE

In this section, we will first explain how observations can be entered into the LTS++ IDE and then we will go through a number of examples for both of our applications, and explain how to interpret the generated results. This is the final step of the LTS++ model creation (i.e., step 9, testing).

Observations can be entered by clicking on the "Next: edit trace" from the LTS++ IDE main page shown in Figure 4. Figure 7 (a) shows an example where the first observation is selected to be a download of an executable, and the second observation is now being selected from the drop-down menu. Once the trace selection is complete, the hypotheses can be generated by clicking on "Generate hypotheses". The hypotheses are presented to the user 10 per page, and users can navigate through these pages. The next 10 hypotheses are generated once the user clicks on the "Next page". Note, the trace editor is intended mainly for testing purposes, and

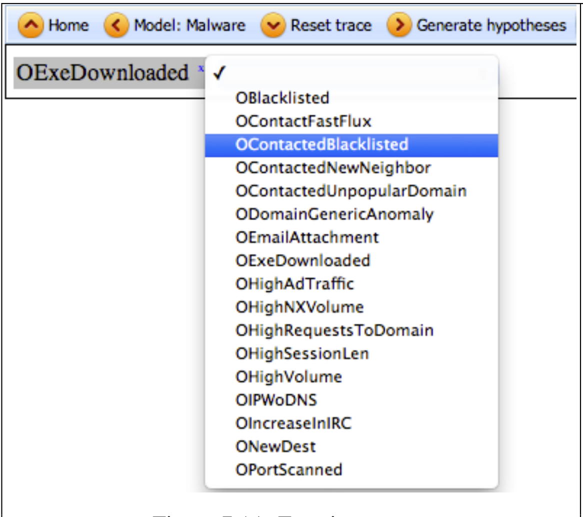


Figure 7 (a): Entering a trace

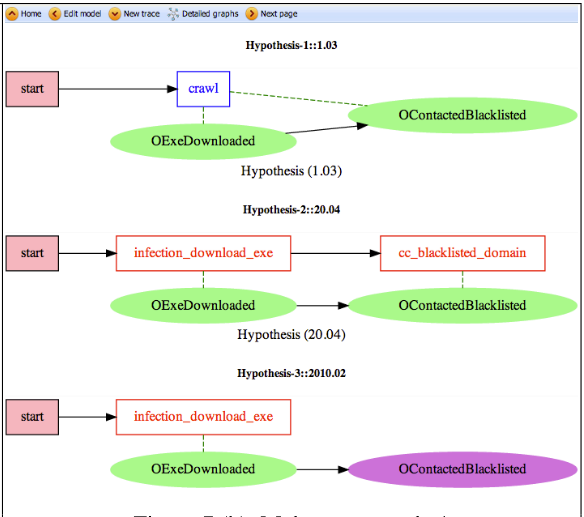


Figure 7 (b): Malware example 1

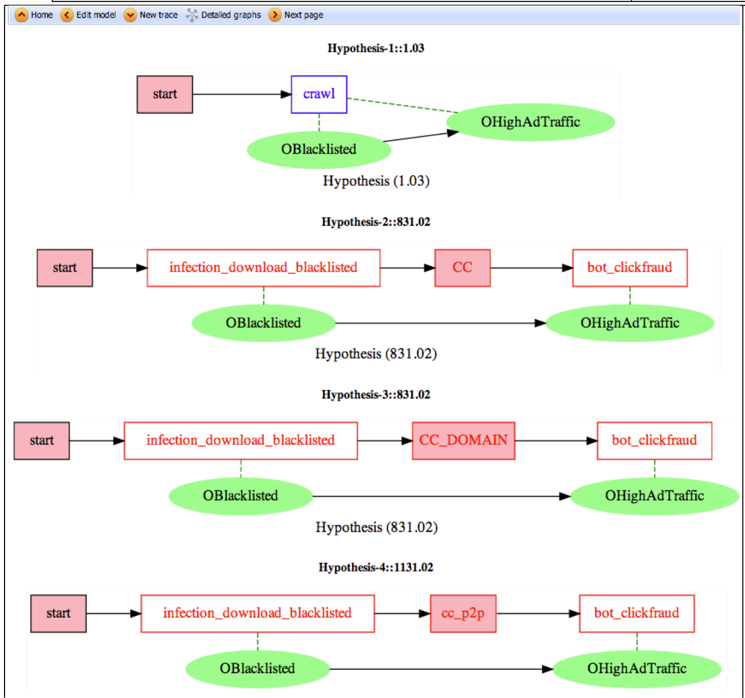


Figure 7 (c): Malware example 2

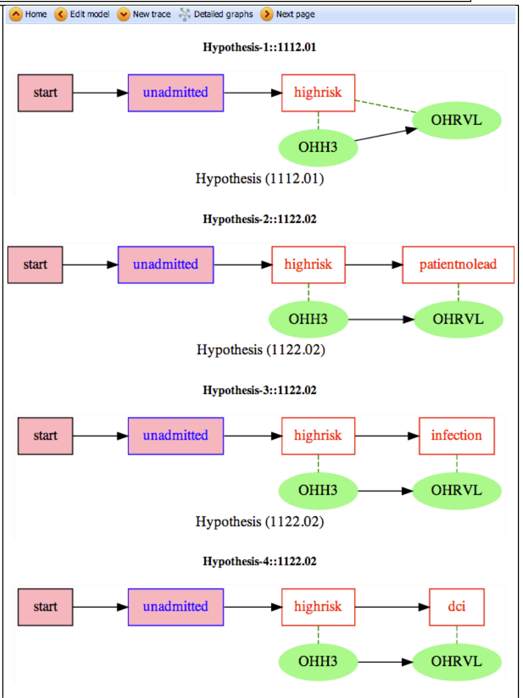


Figure 7 (d): Intensive care example 1

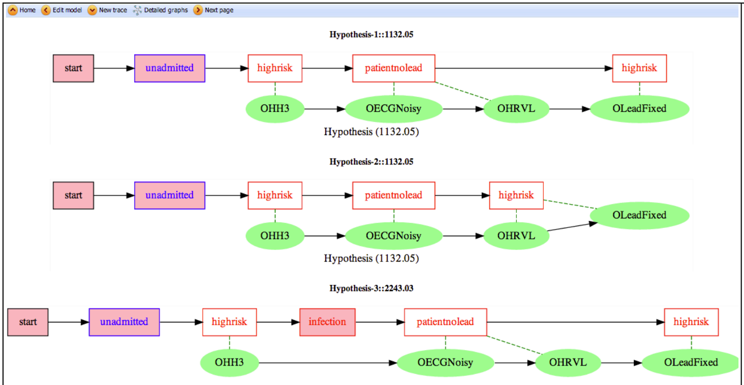


Figure 7 (e): Intensive care example 2

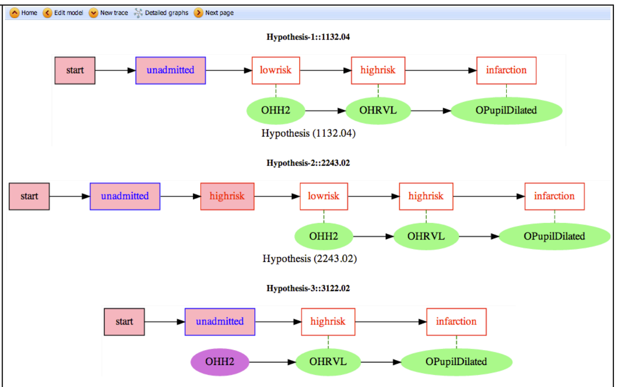


Figure 7 (f): Intensive care example 3

in operation the system will read observations automatically from an input queue.

Figure 7 (b-f) show sample example runs for the malware and intensive care examples; these results are automatically generated by our tool. Each hypothesis is shown as a sequence of states matched to observed event sequence (via green dashed lines). The observations that are explained by a state are shown in green ovals, and unexplained observations are shown in purple. The arrows between the observations show the sequence of observations in the trace. The states shown in red as before are the bad states and good states are drawn in blue. Each hypothesis is associated with a cost. The lower the cost value, the more plausible is the hypothesis.

Figure 7 (b) shows the top 3 generated hypotheses for the trace selected in Figure 7 (a). Our first hypothesis explained both observations. The second hypothesis, almost as plausible, shows infection followed by the CC state. The third hypothesis leaves the second observation unexplained. In some instances, hypotheses include states that are not linked to any observation. For example, the *CC*, *CC_Domain*, *cc_p2p* are the unobserved states in the non-crawling hypotheses in Figure 7 (c). Figure 7 (d) shows the automated generated results (the top-4) for an ambiguous observation HRVL. The result of more specific, less ambiguous observation traces are shown in Figure 7 (e,f).

Summary and Discussion

In this paper, we address the knowledge engineering problem of hypothesis generation motivated by two applications: malware detection and intensive care delivery. To this end, we proposed a modeling language called LTS++ and a web-based tool that enables the specification of a model using the LTS++ language. We also proposed a 9-step process that helps provide guidance to the domain expert in specifying the LTS++ model. Our tool, LTS++ IDE, features syntax highlighting, error detection, and visualization of the state transition graph. The hypotheses are generated by running a planner capable of generating multiple high-quality plans for the translated LTS++ model and the provided trace. The hypotheses can be visualized and shown to the analyst (doctor or network administrator), or can be further investigated automatically via the *Strategic Planner* (see the Architecture Section) to run testing or preventive actions.

In terms of evaluation of our model, we have worked with users outside of our group to develop different LTS++ models in different domains. The feedback we received from them is positive and helped us improve our tool and the creation process. Particularly, one of models developed this way is now used within a larger application.

Our approach in using planning is related to several approaches in the diagnosis literature in which the use of planners as well as SAT solvers is explored (e.g., (Grastien et al. 2007; Sohrabi, Baier, and McIlraith 2010)). In particular, the work on applying planning for the intelligent alarm processing application is most relevant (Bauer et al. 2011; Haslum and Grastien 2011). The authors have considered the case where they can encounter unexplainable observations, but have not provided a formal description of what these unexplainable observations represent or how the planning framework can model them. In this work we address this, as well provide tools for domain experts and introduce a simple language that can be used instead of PDDL.

References

Bauer, A.; Botea, A.; Grastien, A.; Haslum, P.; and Rintanen, J. 2011. Alarm processing with model-based diagnosis of discrete event systems. In *Proceedings of the 22nd International Workshop on Principles of Diagnosis (DX)*, 52–59.

Bouillet, E.; Feblowitz, M.; Feng, H.; Ranganathan, A.; Riabov, A.; Udrea, O.; and Liu, Z. 2009. Mario: middleware for assembly and deployment of multi-platform flow-based applications. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware (Middleware)*, 26:1–26:7.

Cassandras, C., and Lafortune, S. 1999. *Introduction to discrete event systems*. Kluwer Academic Publishers.

Gerevini, A.; Haslum, P.; Long, D.; Saetti, A.; and Dimopoulos, Y. 2009. Deterministic planning in the 5th international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence* 173(5-6):619–668.

Göbelbecker, M.; Keller, T.; Eyerich, P.; Brenner, M.; and Nebel, B. 2010. Coming up with good excuses: What to do when no plan can be found. In *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS)*, 81–88.

Grastien, A.; Anbulagan; Rintanen, J.; and Kelareva, E. 2007. Diagnosis of discrete-event systems using satisfiability algorithms. In *Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI)*, 305–310.

Haslum, P., and Grastien, A. 2011. Diagnosis as planning: Two case studies. In *International Scheduling and Planning Applications woRKshop (SPARK)*, 27–44.

Keyder, E., and Geffner, H. 2009. Soft Goals Can Be Compiled Away. *Journal of Artificial Intelligence Research* 36:547–556.

Magee, J., and Kramer, J. 2006. *Concurrency - state models and Java programs (2. ed.)*. Wiley.

McDermott, D. V. 1998. PDDL — The Planning Domain Definition Language. Technical Report TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

Ramírez, M., and Geffner, H. 2009. Plan recognition as planning. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, 1778–1783.

Richter, S., and Westphal, M. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39:127–177.

Sampath, M.; Sengupta, R.; Lafortune, S.; Sinnamohideen, K.; and Teneketzis, D. 1995. Diagnosability of discrete-event systems. *IEEE Transactions on Automatic Control* 40(9):1555–1575.

Sohrabi, S.; Baier, J.; and McIlraith, S. 2010. Diagnosis as planning revisited. In *Proceedings of the 12th International Conference on the Principles of Knowledge Representation and Reasoning (KR)*, 26–36.

Sohrabi, S.; Baier, J.; and McIlraith, S. 2011. Preferred explanations: Theory and generation via planning. In *Proceedings of the 25th National Conference on Artificial Intelligence (AAAI)*, 261–267.

Sohrabi, S.; Udrea, O.; and Riabov, A. 2013. Hypothesis exploration for malware detection using planning. In *Proceedings of the 27th National Conference on Artificial Intelligence (AAAI)*, 883–889.

Creating Planning Domain Models in KEWI

Gerhard Wickler
Artificial Intelligence Applications Institute
School of Informatics
University of Edinburgh
g.wickler@ed.ac.uk

Lukáš Chrpa and **Thomas Leo McCluskey**
PARK Research Group
School of Computing and Engineering
University of Huddersfield
{l.chrpa, t.l.mccluskey}@hud.ac.uk

Abstract

This paper reports on progress towards a tool for the representation of shared, procedural and declarative knowledge whose aim is to be used for various functions to do with the automation of a complex process control application - primarily to guide the response phase during an emergency situation, but also for supporting normal automated operation.

The tool is a Knowledge Engineering Web Interface called KEWI. The focus of the paper is on the conceptual model used to represent the declarative and procedural knowledge. The model consists of three layers: an ontology, a model of basic actions, and more complex methods. It is this structured conceptual model that facilitates knowledge engineering. We are aiming to evaluate the use of a *central knowledge model* for a range of planning-related functions, where the parts of the model are automatically assembled e.g. into PDDL for operational use.

Introduction

Domain-independent planning has grown significantly in recent years mainly thanks to the International Planning Competition (IPC). Besides many advanced planning engines, PDDL, a de-facto standard language family for describing planning domain and problem models, has been developed. However, encoding domain and problem models in PDDL requires a lot of specific expertise and thus it is very challenging for a non-expert to use planning engines in applications.

This paper concerns the use of AI planning technology in an organisation where (i) non-planning experts are required to encode knowledge (ii) the knowledge base is to be used for more than one planning and scheduling task (iii) it is maintained by several personnel over a long period of time, and (iv) it may have a range of potentially unanticipated uses in the future. The first concern has been a major obstacle to using AI-based, formal representations, in that the expertise required to produce such representations has normally been acquired and encoded by planning experts (e.g. as in NASA's applications (Ai-Chang et al. 2004)). The other concerns are often not covered in the planning literature: in real applications the knowledge encoding is a valuable, general asset, and one that requires a much richer conceptual representation than, for example is accorded by planner-input languages such as PDDL.

We present here a Knowledge Engineering method using a Web Interface aimed at AI Planning, called KEWI. The primary idea behind KEWI then is to ease this formalization of procedural knowledge, allowing domain experts to encode their knowledge themselves, rather than knowledge engineers having to elicit the knowledge before they formalize it into a representation. A number of frameworks exist that support the formalization of planning knowledge in shared web-based systems. Usually, such frameworks build on existing Web 2.0 technologies such as a wiki. A wiki that supports procedural knowledge is available at wikihow.com, but the knowledge remains essentially informal. A system that uses a similar approach, namely, representing procedural knowledge in a wiki is CoScripter (Leshed et al. 2008). However, their representation is not based on AI planning and thus does not support the automated composition of procedures. More recently, an AI-based representation has been used in OpenVCE (Wickler, Tate, and Hansberger 2013).

As far as we are aware, very few collaborative, domain-expert-usable, knowledge acquisition interfaces are available that are aimed at supporting the harvesting of planning knowledge within a rich language for use in a number of planning-related applications. After initial acquisition, the validation, verification, maintenance and evolution of such such knowledge is of prime importance, as the knowledge base is a valuable asset to an organisation.

Related Work

There have been several attempts to create general, user-friendly development environments for planning domain models, but they tend to be limited in the expressiveness of their underlying formalism. The Graphical Interface for Planning with Objects (GIPO) (Simpson, Kitchin, and McCluskey 2007) is based on object-centred languages *OCL* and OCL_h. These formal languages exploit the idea that a set of possible states of objects are defined first, before action (operator) definition (McCluskey and Kitchin 1998). This gives the concept of a *world state* consisting of a set of states of objects, satisfying given constraints. GIPO uses a number of consistency checks such as if the object's class hierarchy is consistent, object state descriptions satisfy invariants, predicate structures and action schema are mutually consistent and task specifications are consistent with the domain model. Such consistency checking guarantees that some types of errors can be prevented, in contrast to ad-hoc methods such as hand crafting.

itSIMPLE (Vaquero et al. 2012) provides a graphical environment that enables knowledge engineers to model planning domain models by using the Unified Modelling Language (UML). Object classes, predicates, action schema are modelled by UML diagrams allowing users to 'visualize' domain models which makes the modelling process easier. itSimple incorporates a model checking tool based on Petri Nets that are used to check invariants or analyze dynamic aspects of the domain models such as deadlocks.

The Extensible Universal Remote Operations Planning Architecture (EUROPA) (Barreiro et al. 2012), is an integrated platform for AI planning and scheduling, constraint programming and optimisation. This platform is designed to handle complex real-world problems, and the platform has been used in some of NASA's missions. EUROPA supports two representation languages, NDDL and ANML (Smith, Frank, and Cushing 2008), however, PDDL is not supported.

Besides these tools, it is also good to mention VIZ (Vodrážka and Chrpa 2010), a simplistic tool inspired by itSimple, and PDDL Studio (Plch et al. 2012), an editor which provides users a support by, for instance, identifying syntax errors or highlighting components of PDDL.

In the field of Knowledge Engineering, methodologies have been developed which centre on the creation of a precise, declarative and detailed model of the area of knowledge to be engineered, in contrast to earlier expert systems approaches which appeared to focus on the "transfer" expertise at a more superficial level. This "expertise model" contains a mix of knowledge about the "problem solving method" needed within the application and the declarative knowledge about the application. Often a key rationale for knowledge engineering is to create declarative representations of an area to act as a formalised part of some requirements, making explicit what hitherto has been implicit in code, or explicit but in documents. Knowledge Engineering modelling frameworks arose out of this, such as CommonKads (Schreiber et al. 1999), which were based on a deep modelling of an area of expertise, and emphasising a lifecycle of this model. The "knowledge model" within CommonKADS, which contains a formal encoding of task knowledge, such as problem statement(s), as well as domain knowledge, is similar to the kind of knowledge captured in KEWI. Unlike KEWI however, this model was expected to be created by knowledge engineers rather than domain experts and users.

Conceptual Model of KEWI

KEWI is a tool for encoding domain knowledge mainly by experts in the application area rather than AI planning experts. The key idea behind KEWI is to provide a user-friendly environment as well as a language which is easier to follow, especially for users who are not AI planning experts. A high-level architecture of KEWI is depicted in Figure 1. Encoded knowledge can be exported into the domain and problem description in PDDL on which standard planning

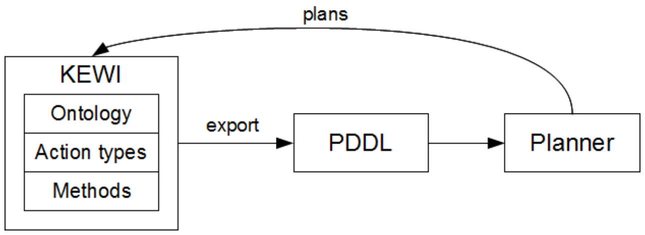


Figure 1: An architecture of KEWI.

engines can be applied, and retrieved plans can be imported back to KEWI. Hence, the user does not have to understand, or even be aware, of any PDDL encodings.

A language in which domain knowledge is encoded in KEWI has three parts, which are explained in the following subsections. First, a domain ontology is defined. The domain ontology consists of definition of classes of objects, hierarchies of classes and relations between objects. Second, action types, concretely action name, preconditions and effects, are defined. Third, methods which introduce additional ordering constraints between actions, are defined.

Ontology: Concepts, Relations and Properties

Ontological elements are usually divided into concepts and instances. Typically, the concepts are defined in a planning domain whereas the instances are defined in a planning problem. Since our focus for KEWI is on planning domains we shall mostly deal with concepts here.

Concepts A concept is represented by a unique symbol in KEWI. The formal definition of a concept is given by its super-class symbol and by a set of role constraints that define how instances of the concept may be related to other concepts. In KEWI, the definition of a concept also includes other, informal elements that are not used for formal reasoning. However, the knowledge engineering value of such informal elements must not be underestimated, much like the comments in programming often are vital for code to be understandable.

Definition 1 (KEWI Concept)**.** *A concept C in KEWI is a pair $\langle C^{sup}, R\rangle$, where:*

- *C^{sup} is the direct super-concept of C and*
- *R is a set of role constraints of the form $\langle r, n, C'\rangle$ where r is a symbolic role name, C' is a concept (denoting the role filler type), and n is a range $[n_{min}, n_{max}]$ constraining the number of different instances to play that role.*

We assume that there exists a unique root concept often referred to as *object* or *thing* that acts as the implicit super-concept for those concepts that do not have an explicit super-concept defined in the same planing domain. Thus, a concept C may be defined as $\langle \triangle, R\rangle$, meaning its super concept is implicit. This implicit super-concept has no role constraints attached.

For example, in the Dock Worker Robot (DWR) domain, the concepts `container` and `pallet` could be defined with the super-concept `stackable`, whereas the concept `crane` could be defined as a root concept with no super-concept (implicitly: $\triangle$). A role constraint can be used to define that a crane can hold at most one container as follows: $\langle \texttt{holds}, [0,1], \texttt{container} \rangle$.

Since super-concepts are also concepts, we can write a concept C as $\langle\langle\langle\triangle, R_n\rangle, \ldots, R_2\rangle, R_1\rangle$. Then we can refer to all the role constraints associated with C as $R^* = R_n \cup \ldots \cup R_2 \cup R_1$, that is, the role constraints that appear in the definition of C, the role constraints in its direct super-concept, the role constraints in its super-concepts super-concept, etc.

The reason for introducing this simple ontology of concepts is that we can now constrain the set of possible world states based on the role constraints. States are defined as sets of ground, first-order atoms over some function-free language $\mathcal{L}$. This language shall contain symbols to denote each instance of a concept defined in the ontology $(c_1, \ldots, c_L)$ where the type function τ maps each instance c_i to its type C, a concept in the ontology. The relation symbols of $\mathcal{L}$ are defined through the role constraints.

Definition 2 (Relations in $\mathcal{L}$). *Let $\langle r, n, C'\rangle$ be a role constraint of some concept C. Then the first-order language $\mathcal{L}$ that can be used to write ground atoms in a state contains a binary relation $C.r \subseteq C \times C'$.*

In what follows we shall extend the language to include further relation symbols, but for now these relations defined by the ontology are all the relations that may occur in a state. The reason why the relation name is a combination of the concept and the role is simply to disambiguate between roles of the same name but defined in different concepts. Where all role names are unique the concept may be omitted.

We can now define what it means for a state to be valid with respect to an ontology defined as a set of KEWI concepts. Essentially, for a state to be valid, every instance mentioned in the state must respect all the role constraints associated with the concepts to which the instance belongs. Since role constraints are constraints on the number of possible role fillers we need to be able to count these.

Definition 3 (Role Fillers). *Let s be a state, that is, a set of ground atoms over objects $c_1, \ldots, c_L$ using the relations in $\mathcal{L}$. Let $\langle r, n, C'\rangle$ be a role constraint of some concept C. Then we define $vals_s(C.r, c_i) = \{c_f | C.r(c_i, c_f) \in s\}, c_i \in C, c_f \in C'$, that is, the set of all constants that play role r for c_i in s.*

Definition 4 (Valid State). *Let C be a KEWI concept. Then a state s is valid if, for any instance c_i of C and any role constraint $\langle r, n, C'\rangle$ of C or one of its (direct and indirect) super-concepts, the number of ground atoms $a = C.r(c_i, *)$ must be in the range $[n_{min}, n_{max}]$, i.e. $n_{min} \leq |vals_s(C.r, c_i)| \leq n_{max}$.*

Thus, a concept definition defines a set of role constraints which can be interpreted as relations in a world state. The numeric range defines how many ground instances we may find in a valid state. This is the core of the ontological model used in KEWI.

For example, let `k1` be a crane and `ca` be a container. Then a state may contain a ground atom `crane.holds(k1,ca)`. If a state contains this atom, it may not contain another one using the same relation and `k1` as the first argument.

Relations While the relations defined through the concepts in KEWI provide a strong ontological underpinning for the representation, there are often situations where other relations are more natural, e.g. to relate more than two concepts to each other, or where a relation does not belong to a concept. In this case relations can be defined by declaring number and types (concepts) of the expected arguments.

Definition 5 (Relations in $\mathcal{L}$). *A relation may be defined by a role constraint as described above, or it may be a relation symbol followed by an appropriate number of constants. The signature of a relation R is defined as $C_1 \times \ldots \times C_R$ where C_i defines the type of the ith argument.*

A valid state may contain any number of ground instances of these relations. As long as the types of the constants in the ground atoms agree with the signature of the relation, the state that contains this atom may be valid.

Properties In reality, we distinguish three different types of role constraints: *related classes* for defining arbitrary relations between concepts, *related parts* which can be used to define a "part-of" hierarchy between concepts, and *properties* which relate instances to property values.

The first two are equivalent in the sense that they relate objects to each other. However, properties usually relate values to objects, e.g. an object may be of a given colour. While it often makes sense to distinguish all individual instances of a concept, this is not true for properties. While the paint that covers one container may not be the same paint that covers another, the colour may be the same. To allow for the representations of properties in KEWI, we allow for the definition of properties with enumerated values.

Definition 6 (Properties). *A property P is defined as a set of constant values $\{p_1, \ldots, p_P\}$.*

It is easy to see that the above definitions relating to role constraints and other relations can be extended to allow properties in place of concepts and property values in place of instances. A minor caveat is that property values are usually defined as part of a planning domain, whereas instances are usually given in a planning problem.

Action Types

Action types in KEWI are specified using an operator name with typed arguments, a set of preconditions, and a set of effects. This high-level conceptualization of action types is of course very common in AI planning formalisms. KEWI's representation is closely linked with the ontology, however. This will enable a number of features that allow for a more concise representation, allowing to reduce the redundancy contained in many PDDL planning domains.

Object References In many action representations it is necessary to introduce one variable for each object that is somehow involved in the execution of an action. This variable is declared as one of the typed arguments of the action type. The variable can then be used in the preconditions and effects to consistently refer to the same objects and express conditions on this object.

Sometimes, an action type may need to refer to specific constants in its preconditions or effects. In this case, the unique symbol can be used to identify a specific instance. In the example above, `k1` was used to refer to a crane and `ca` to refer to a container. In most planning domains, operator definitions do not refer to specific objects, but constants may be used as values of properties.

In addition to variables and constants, KEWI also allows a limited set of function terms to be used to refer to objects in an action type's preconditions and effects. Not surprisingly, this is closely linked with the ontology, specifically with the role constraints that specify a maximum of one in their range.

Definition 7 (Function Terms)**.** *Let $\langle r, n, C' \rangle$ be a role constraint of some concept C where $n_{max} = 1$. Then we shall permit the use of function terms of the form $C.r(t)$ in preconditions and effects, where t can again be an arbitrary term (constant, variable, or function term) of type C'.*

Let s be a valid state, that is, a set of ground atoms over objects $c_1, \dots, c_L$ using the relations in $\mathcal{L}$. Then the constant represented by the function term $C.r(c_i)$ is:

- *c_j if $vals_s(C.r, c_i) = \{c_j\}$, or*
- *`nothing` $(\perp)$ if $vals_s(C.r, c_i) = \emptyset$.*

Note that the set $vals_s(C.r, c_i)$ can contain at most one element in any valid state. If it contains an element, this element is the value of the function term. Otherwise a new symbol that must not be one of the constants $c_1, \dots, c_L$ will be used to denote that the function term has no value. This new constant `nothing` may also be used in preconditions as described below.

The basic idea behind function terms is that they allow the knowledge representation to be more concise; it is no longer necessary to introduce a variable for each object. Also, this style of representation may be more natural, e.g. to refer to the container held by a crane as `crane.holds(k1)` meaning "whatever crane `k1 holds`", where the role constraint tells us this must be a container. As a side effect, the generation of a fully ground planning problem could be simpler,given the potentially reduced number of action parameters.

Interestingly, a step in this direction was already proposed in PDDL 1, in which some variables were declared as parameters and others as "local" variables inside an operator. However, with no numeric constraints on role fillers or any other type of relation, it is difficult to make use of such variables in a consistent way. Similarly, state-variable representations exploit the uniqueness of a value. However, this was restricted to the case where n_{min} and n_{max} both must be one.

Condition Types The atomic expressions that can be used in preconditions and effects can be divided into two categories. Firstly, there are the explicitly defined relations. These are identical in meaning and use to PDDL and thus, there is no need to discuss these further. Secondly, there are the relations based on role constraints which have the same form as such atoms in states, except that they need not be ground.

Definition 8 (Satisfied Atoms)**.** *Let s be a valid state over objects $c_1, \dots, c_L$. Then a ground atom a is satisfied in s (denoted $s \models a$) if and only if:*

- *a is of the form $C.r(c_i, c_j)$ and $a \in s$, or*
- *a is of the form $R(c_{i_1}, \dots, c_{i_R})$ and $a \in s$, or*
- *a is of the form $C.r(c_i, \perp)$ and $vals_s(C.r, c_i) = \emptyset$.*

The first two cases are in line with the standard semantics, whereas the the last case is new and lets us express that no role filler for a given instance exists in a given state. Note that the semantics of atoms that use the symbol `nothing` in any other place than as a role filler are never satisfied in any state.

The above definition can now be used to define when an action is applicable in a state.

Definition 9 (Action Applicability)**.** *Let s be a valid state and act be an action, i.e. a ground instance of an action type with atomic preconditions $p_1, \dots, p_a$. Then act is applicable in s if and only if every precondition is satisfied in s: $\forall p \in p_1, \dots, p_a : s \models p$.*

This concludes the semantics of atoms used in preconditions. Atoms used in effects describe how the state of the world changes when an action is applied. This is usually described by the state transition function $\gamma : S \times A \to S$, i.e. it maps a state and an applicable action to a new state. Essentially, γ modifies the given state by deleting some atoms and adding some others. Which atoms are deleted and which are added depends on the effects of the action. If the action is not applicable the function is undefined.

Definition 10 (Effect Atoms)**.** *Let s be a valid state and act be an action that is applicable in s. Then the successor state $\gamma(s, a)$ is computed by:*

1. *deleting all the atoms that are declared as negative effects of the action,*
2. *for every positive effect $C.r(c_i, c_j)$ for role constraint $\langle r, n, C' \rangle$ with $n = [n_{min}, 1]$, if $C.r(c_i, c_k) \in s$ delete this atom, and*
3. *add all the atoms that are declared as positive effects of the action.*

Following this definition allows for a declaration of actions using arbitrary relations and state-variables that may have at most one value. The ontology, more specifically the numeric role constraints can be used to distinguish the two cases.

The symbol `nothing` is not allowed in effects in KEWI. Of course, it would be easy to define the semantics of such a construct as one that retracts all such atoms from the state. However, we have chosen not to go this way in KEWI for two reasons. Firstly, this construct would severely restrict the number of planners that can handle this mass retraction, although it may be possible to express this as a universally quantified effect. Secondly, it is not clear what an example of such an action would be in practise.

Methods

The definition of methods in KEWI is not yet finished. As the framework is at least partially application-driven, we may need to further refine the conceptual framework outlined (but not fully defined) below.

The approach adopted in KEWI follows standard HTN planning concepts: a method describes how a larger task can be broken down in into smaller tasks which, together, accomplish the larger task.

A method is defined by a method name with some parameters. The name usually suggests how something is to be done and the parameters have the same function as in action types; they are the objects that are used or manipulated during the instantiation of a method. Next, a method must declare the task that is accomplished by the method. This is defined by a task name usually describing what is to be done, and again some parameters. For primitive tasks, the task name will be equal to the name of an action type, in which case no further refinement is required. For non-primitive tasks, a method also includes a set of subtasks. In KEWI, the ordering constraints between subtasks are declared with the subtask, rather than as a separate component of the method. This is simply to aid readablility and does not change the expressiveness.

In addition, to these standard components, KEWI allows the specification of high-level effects and subgoal-subtasks. The aim here is to allow for a representation that supports flat, PDDL-like planning domains as well as hierarchical planning domains.

When a method declares that it achieves a high-level effect, then every decomposition of this method must result in an action sequence which will achieve the high-level effect after the last action of the sequence has been completed. This could allow a planner to use a method as if it was an action in a backward search. An alternative view is that such a method functions as a macro action type in the domain.

A method may also include subtasks that are effectively subgoals. For example, the subtask "achieve $C.r(c_i, c_j)$" may be used to state that at the corresponding point in the subtask the condition $C.r(c_i, c_j)$ must hold in the state. The idea being that a planner may revert to flat planning (such as state-space search) to find actions to be inserted into the plan at this point, until the subgoal is achieved.

This mixed approach is not new and has been used in practical planners like O-Plan (Currie and Tate 1991). However, the semantics has not been formally defined for this approach, something we shall attempt in future work.

Export to PDDL

Given that most modern planners accept planning domains and problems in PDDL syntax as their input, one of the goals for KEWI was to provide a mechanism that exports the knowledge in KEWI to PDDL. Of course, this will not include the HTN methods as PDDL does not support hierarchical planning formalisms.

Function Terms The first construct that must be removed from KEWI's representation are the function terms that may be used to refer to objects. In PDDL's preconditions and effects only variables (or symbols) may be used to refer to objects. The following function can be used to eliminate a function term of the form $C.r(t)$ that occurs in an action type O's preconditions or effects.

```
function eliminate-fterms($C.r(t), O$)
  if is-fterm($t$) then
    eliminate-fterms($t, O$)
  $v \leftarrow$ get-variable($C.r(t), O$)
  replace every $C.r(t)$ in $O$ by $v$
```

The function first tests whether the argument to the given function term is itself a function term. If this the case, it has to be eliminated first. This guarantees that, for the remainder of the function t is either a variable or a symbol. We then use the function "get-variable" to identify a suitable variable that can replace the function term. Technically, this function may return a symbol, but the treatment is identical, which is why we shall not distinguish these cases here. The identification of a suitable variable then works as follows.

```
function get-variable($C.r(v), O$)
  for every positive precondition $p$ of $O$ do
    if $p = C.r(v, v')$ then
      if is-fterm($v'$) then
        eliminate-fterms($v', O$)
      return $v'$
  retrieve $\langle r, n, C' \rangle$ from $C$
  add new parameter $v'$ of type $C'$ to $O$
  add new precondition $C.r(v, v')$ to $O$
  return $v'$
```

This function first searches for an existing, positive precondition that identifies a value for the function. Since function terms may only be used for constraints that have at most one value, there can only be at most one such precondition. If such a precondition exists, its role filler (v', a variable or a symbol) may be used as the result. If no such precondition can be found, the function will create a new one and add it to the operator. To this end, a new parameter must be added to the action type, and to know the type of the variable we need to retrieve the role filler type from the role constraint. In practise, we also use the type name to generate a suitable variable name. Then a new precondition can be added that effectively binds the function to the role filler. And finally, the new variable may be returned.

Handling `nothing` The next construct that needs to be eliminated from the KEWI representation is any precondition that uses the role filler `nothing`. Note that this symbol does not occur in states and thus cannot be bound in traditional PDDL semantics. Simply adding this symbol to the state is not an option since other preconditions that require a specific value could then be unified with this state atom. For example, if we had an explicit atom that stated `holds(k1, nothing)` in our state, then the precondition `holds(?k, ?c)` of the load action type would be unifiable with this atom. An inequality precondition may solve this problem, but only if the planner can correctly handle inequalities. The alternative approach we have implemented in KEWI is described in the following algorithm.

```
**function** eliminate-nothing($O$)
  **for every** precondition $p = C.r(v, \bot)$ **do**
    replace $p$ with $C.r.\ \bot\ (v)$
    **if** $O$ has an effect $e = C.r(v, v')$
      add another effect $\neg C.r.\ \bot\ (v)$
```

The basic idea behind this approach is to use a new predicate to keep track of state-variables that have no values in a state. This is the purpose of the new predicate "$C.r.\ \bot$", indicating the role r of concept C has no filler for the given argument. This is a common approach in knowledge engineering for planning domains. For example, in the classic blocks world we find a "holds" relation for when a block is being held, and a predicate "hand-empty" for when no block is held.

The algorithm above uses this technique to replace all preconditions that have `nothing` as a role filler with a different precondition that expresses the non-existence of the role filler. To maintain this condition, it will also be necessary to modify the effects accordingly. This is done by adding the negation of this new predicate to corresponding existing effects.

Since this is pseudo code, the algorithm actually omits a few details, e.g. the declaration of the new predicate in the corresponding section of the PDDL domain, and the fact that the planning problem also needs to be modified to account for the new predicate. Both is fairly straight forward to implement.

State-Variable Updates Finally, the cases in which the value of a state-variable is simply changed needs to be handled. The approach we have adopted here is identical to the approach described in (Ghallab, Nau, and Traverso 2004). That is, when an effect assigns a new value to a state variable, e.g. $C.r(v, v_{new})$, we need to add a precondition to get the old value, e.g. $C.r(v, v_{old})$, and then we can use this value in a new negative effect to retract the old value: $\neg C.r(v, v_{old})$.

Example

In this section we shall look at a single operator taken from the DWR domain and compare its representation in PDDL as defined in (Ghallab, Nau, and Traverso 2004) with the equivalent operator in KEWI. Note that this comparison does not include the representation of the underlying ontology, which is rather trivial in the case of the PDDL version of the domain. However, there is one fundamental difference in the two ontologies, namely, that the PDDL version uses just one symbol `pallet` to denote all the pallets which are at the bottom of each pile. We consider this epistemologically inadequate. To avoid this discussion, we shall look at the move operator here, which does not interact with piles of containers directly, and therefore this ontological difference does not show up. In the PDDL version, the operator looks as follows:

```
(:action move
  :parameters
    (?r - robot ?from ?to - location)
  :precondition (and
    (adjacent ?from ?to)
    (at ?r ?from)
    (not (occupied ?to)))
  :effect (and
    (at ?r ?to)
    (not (occupied ?from))
    (occupied ?to)
    (not (at ?r ?from))))
```

The move operator takes three arguments: the robot to be moved and the two locations involved. There are three preconditions expressing that applicability of this operator depends on the two locations being adjacent, the robot initially being at the location from which the action takes place, and the destination location currently not being occupied. Note that the latter is a negative precondition which cannot be handled by all planners. The effects come in pairs and are somewhat redundant. The robot being at the destination of course implies that this location is now occupied. Similarly, the location that has just been vacated by the robot is now not occupied and the robot is not at that location.

The equivalent operator in KEWI exploits some of the features described above. However, it is important to note that the user interface does not provide a text editor that can be used to modify a PDDL-like representation. Instead, it consists of a web-form with fields for the various components that define an action type. For comparison, we have printed the internal KEWI representation in a Lisp-like syntax, which looks as follows:

```
(:action-type move
    (:arguments ( (?robot robot)
      (?from location) (?to location) ))
    (:precondition (and
      (:relation adjacent
        ( ?from ?to ))
      (:constr location.occupied-by
        ( ?from ?robot ))
      (:constr location.occupied-by
        ( ?to nothing ))))
    (:effect (and
      (:constr location.occupied-by
        ( ?to ?robot ))
      (:constr :not location.occupied-by
        ( ?from ?robot )))))
```

The KEWI version of this operator also requires three parameters. This is because none of the parameters is uniquely implied by any of the others. However, this is the only action in the DWR domain for which this is the case. All actions involving a crane have the location of the crane as another parameter, which would not be required in the KEWI version. The number of preconditions is not reduced either in this case. However, the KEWI version of the operator does not require a negative precondition since it exploits ontological knowledge about the occupancy of locations. Thus, this precondition can be reformulated using the `nothing` symbol. The biggest difference between the two representations is in the effects, where KEWI only requires one effect for each of the pairs listed above. Again, this is due to the ontology that can be exploited, specifically to the fact that location can only be occupied by one robot at a time. Not having to represent such redundant effects reduces the risk of knowledge engineers forgetting to list such effects. Note that the second effect is negative, but could perhaps be expressed more elegantly by using the `nothing` symbol as in the preconditions. This is not permitted in the syntax at present.

Evaluation

This work is being carried out with an industrial partner with significant experience in control and automation as well as simulation, and we are using a real application of knowledge acquisition and engineering in their area of expertise. The development of KEWI is in fact work in progress, and its evaluation is ongoing, and being done in several ways: (i) An expert engineer from the industrial partner is using KEWI, in parallel with the developers, to build up a knowledge base of knowledge about artefacts, operations, procedures etc. in their domain. (ii) We have created a hand-crafted PDDL domain and problem descriptions of part of the partner's domain and for the same problem area we have generated PDDL automatically from a tool inside KEWI. We are in the process of comparing the two methods and the PDDL produced. An interface to a simulation system is being developed which will help in this aspect. (iii) We are working with another planning project in the same application, which aims to produce natural language explanations and argumentation supporting plans. In the future we believe to combine KEWI with this work, in order that (consistent with involving the user in model creation) the user will be able to better validate the planning operation.

Conclusions

In this paper we have introduced a knowledge engineering tool for building planning domain models, and given a formal account of parts of its structure and tools. Its characteristics are that (i) it has a user-friendly interface which is simple enough to support domain experts in encoding knowledge (ii) it is designed so that it can be used to acquire a range of knowledge, with links to operation via automated translators that create PDDL domain models (iii) it is designed to enable a groups of users to capture, store and maintain knowledge over a period of time, to enable the possibility of knowledge reuse.

Acknowledgements

The research was funded by the UK EPSRC Autonomous and Intelligent Systems Programme (grant no. EP/J011991/1). The University of Edinburgh and research sponsors are authorized to reproduce and distribute reprints and online copies for their purposes notwithstanding any copyright annotation hereon.

References

Ai-Chang, M.; Bresina, J. L.; Charest, L.; Chase, A.; jung Hsu, J. C.; Jónsson, A. K.; Kanefsky, B.; Morris, P. H.; Rajan, K.; Yglesias, J.; Chafin, B. G.; Dias, W. C.; and Maldague, P. F. 2004. Mapgen: Mixed-initiative planning and scheduling for the mars exploration rover mission. *IEEE Intelligent Systems* 19(1):8–12.

Barreiro, J.; Boyce, M.; Do, M.; Frank, J.; Iatauro, M.; Kichkaylo, T.; Morris, P.; Ong, J.; Remolina, E.; and Smith, T. 2012. EUROPA: A platform for AI planning, scheduling, constraint programming, and optimization. In *4th International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS).*

Currie, K., and Tate, A. 1991. O-Plan: The open planning architeture. *Artificial Intelligence* 52:49–86.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning*. Morgan Kaufmann.

Leshed, G.; Haber, E. M.; Matthews, T.; and Lau, T. A. 2008. Coscripter: automating & sharing how-to knowledge in the enterprise. In *CHI*, 1719–1728.

McCluskey, T. L., and Kitchin, D. E. 1998. A tool-supported approach to engineering HTN planning models. In *In Proceedings of 10th IEEE International Conference on Tools with Artificial Intelligence*.

Plch, T.; Chomut, M.; Brom, C.; and Barták, R. 2012. Inspect, edit and debug PDDL documents: Simply and efficiently with PDDL studio. *ICAPS12 System Demonstration* 4.

Schreiber, G.; Akkermans, H.; Anjewierden, A.; de Hoog, R.; Shadbolt, N.; de Velde, W. V.; and Wielinga, B. J. 1999. *Knowledge Engineering and Management: The CommonKADS Methodology*. Cambridge, Mass.: MIT Press, 2nd ed. edition.

Simpson, R.; Kitchin, D. E.; and McCluskey, T. 2007. Planning domain definition using gipo. *Knowledge Engineering Review* 22(2):117–134.

Smith, D. E.; Frank, J.; and Cushing, W. 2008. The anml language. *Proceedings of ICAPS-08.*

Vaquero, T. S.; Tonaco, R.; Costa, G.; Tonidandel, F.; Silva, J. R.; and Beck, J. C. 2012. itSIMPLE4.0: Enhancing the modeling experience of planning problems. In *System Demonstration – Proceedings of the 22nd International Conference on Automated Planning & Scheduling (ICAPS-12).*

Vodrážka, J., and Chrpa, L. 2010. Visual design of planning domains. In *KEPS 2010: Workshop on Knowledge Engineering for Planning and Scheduling*.

Wickler, G.; Tate, A.; and Hansberger, J. 2013. Using shared procedural knowledge for virtual collaboration support in emergency response. *IEEE Intelligent Systems* 28(4):9–17.