27th International Conference on
Automated Planning and Scheduling
June 19-23, 2017, Pittsburgh, USA

KEPS 2017

Proceedings of the Workshop on
Knowledge Engineering for Planning and Scheduling

Edited by:

Lukas Chrpa, Mauro Vallati and Tiago Vaquero

Organization

Lukas Chrpa, Czech Technical University & Charles University in Prague, Czech Republic

Mauro Vallati, University of Huddersfield, UK

Tiago Vaquero, MIT, USA

Program Committee

Dimitris Vrakas, Aristotle University of Thessaloniki, Greece

Shirin Sohrabi, IBM Research, USA

Patricia Riddle, University of Auckland, New Zealand

Julie Porteous, Teesside University, UK

Ron Petrick, Heriot-Watt University, UK

Simon Parkinson, University of Huddersfield, UK

Andrea Orlandini, National Research Council of Italy (ISTC-CNR), Italy

Lee Mccluskey, University of Huddersfield, UK

Simone Fratini, European Space Agency - ESA/ESOC, Germany

Jeremy Frank, NASA, USA

Susana Fernandez, Universidad Carlos III de Madrid, Spain

Amedeo Cesta, National Research Council of Italy (CNR), Italy

Luis Castillo, University of Granada, Spain

Adi Botea, IBM Research , Ireland

Roman Barták, Charles University, Czech Republic

Contents

StoryFramer: From Input Stories to Output Planning Models

Thomas Hayton[1], Julie Porteous[1], João F. Ferreira[1,3], Alan Lindsay[1], Jonathon Read[2]
[1]Digital Futures Institute, School of Computing, Teesside University, UK.
[2]Ocado Technology, Hatfield, UK.
[3]HASLab/INESC TEC, Universidade do Minho, 4704-553 Braga, Portugal.
firstinitial.lastname@tees.ac.uk | jonathon.read@ocado.com

Abstract

We are interested in the problem of creating narrative planning models for use in Interactive Multimedia Storytelling Systems. Modelling of planning domains has been identified as a major bottleneck in the wider field of planning technologies and this is particularly so for narrative applications where authors are likely to be non-technical. On the other hand there are many large corpora of stories and plot synopses, in natural language, which could be mined to extract content that could be used to build narrative domain models.

In this paper we describe an approach to learning narrative planning domain models from input natural language plot synopses. Our approach, called StoryFramer, takes natural language input and uses NLP techniques to construct structured representations from which we build up domain model content. The system also prompts the user for input to disambiguate content and select from candidate actions and predicates. We fully describe the approach and illustrate it with an end-to-end worked example. We evaluate the performance of StoryFramer with NL input for narrative domains which demonstrate the potential of the approach for learning complete domain models.

Introduction

Interactive Multimedia Storytelling (IS) systems allow users to interact and influence, in real-time, the evolution of a narrative as it is presented to them. This presentation can be via a range of different output media such as 2D or 3D animation (Mateas and Stern 2005; Porteous, Charles, and Cavazza 2013), filmic content (Piacenza et al. 2011) and text (Cardona-Rivera and Li 2016). In addition, a range of different interaction mechanisms have been used such as emotional speech input (Cavazza et al. 2009), gaze (Bee et al. 2010), and physiologicali measures (Gilroy et al. 2012).

AI planning has been widely used for narrative generation in IS as it provides: a natural "fit" with story plot lines represented as narrative plans; ensures causality which is important for the generation of meaningful and comprehensible narratives; and provides considerable flexibility and potential generative power. Consequently plan-based approaches have featured in many systems (e.g. as reported by (Aylett, Dias, and Paiva 2006; Riedl and Young 2010; Porteous, Charles, and Cavazza 2013)).

In this work we are interested in the problem of authoring the narrative planning domain models that are used in such IS systems. To date the authoring of narrative planning models has been handled manually, a common strategy being to build up the model via systematic consideration of alternatives around a baseline plot (Porteous, Cavazza, and Charles 2010b) and many prototype IS systems have sought inspiration from existing literary or filmic work. Examples include the Façade interactive drama which was based on "Who's Afraid of Virginia Woolf?" (Mateas and Stern 2005), The Merchant of Venice (Porteous, Cavazza, and Charles 2010a), Madame Bovary (Cavazza et al. 2009) and the tale of Aladdin (Riedl and Young 2010).

However this manual creation is extremely challenging. In this paper the problem we tackle is automation of narrative domain model creation. Our starting point is to look at natural language plot outlines as content from which to automatically induce planning models. We are developing a solution which takes natural language input (i.e. stories) and uses NLP techniques to construct structured representations of the text and keeps the user in the loop to guide refinement of the planning model. This approach represents an extension to the Framer system of (Lindsay et al. 2017) to application with narrative domain models. We have implemented the approach in a prototype system called StoryFramer.

In the paper we give an overview of the technical aspects of the approach and illustrate it with an end-to-end worked example using the tale of Aladdin taken from (Riedl and Young 2010). We present the results of an evaluation with a two domain models generated by StoryFramer and consider the potential of the approach.

Related Work

Some recent work in the area of automated domain model creation for planning has attempted to learn planner action models from natural language (NL) input.

Much of this work has attempted to map from NL input onto existing formal representations. For example, in relation to RoboCup@Home, Kollar et al. (2013) present a probabilistic approach to learning the referring expressions for robot primitives and physical locations in a region. Also Mokhtari, Lopes, and Pinho (2016) present an approach to learning action schemata for high-level robot control.

In (Goldwasser and Roth 2011) the authors present an alternative approach to learning the dynamics of the world where the NL input provides a direct lesson about part of the

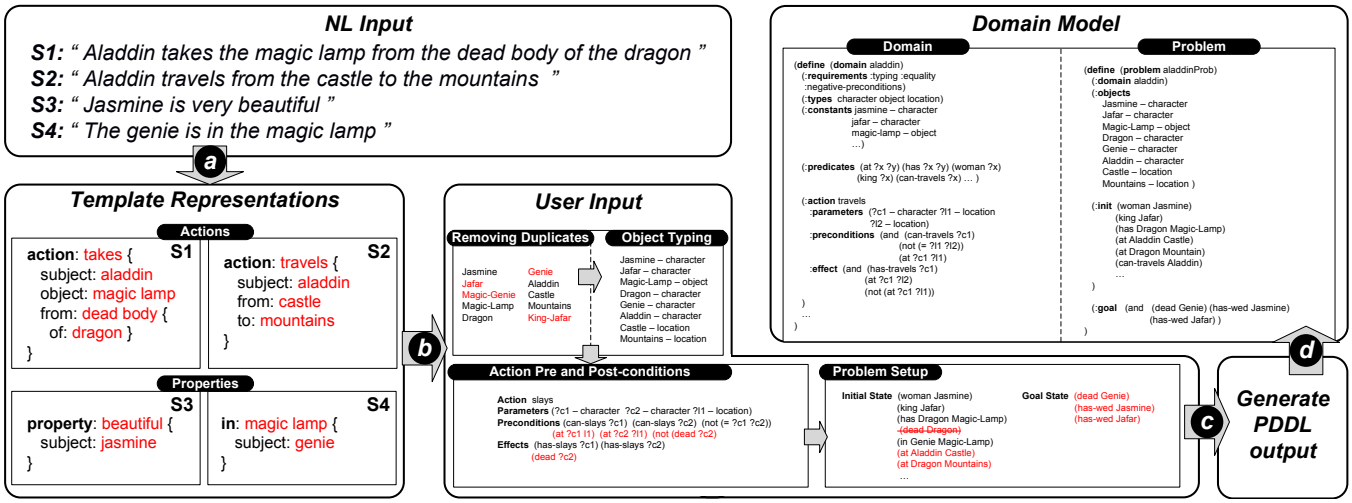


Figure 1: StoryFramer Overview: the NL input sentences and CoreNLP annotations are mapped to Template Representations ⓐ; the user disambiguates content, types objects and selects predicates for pre and post-conditions ⓑ; the different elements of the domain content are assembled ⓒ; and the domain content is output as PDDL domain model and problem file ⓓ.

dynamics of the environment. Each lesson is supported by a small training data set to support learning from the lessons. In contrast to our approach, their system relies on a representation of the states and actions, which means their NLP approach can target an existing language.

More closely related to our work are attempts to learn planning models in the absence of a target representation. These include (Sil and Yates 2011) who used text mining via a search engine to identify documents that contain words that represent target verbs or events and then uses inductive learning techniques to identify appropriate action pre- and post-conditions. Their system was able to learn action representations, although with certain restrictions such as the number of predicate arguments. Branavan et al. (2012) introduce a reinforcement learning approach which uses surface linguistic cues to learn pre-condition relation pairs from text for use during planning. The success of the learnt model relies on use of feedback automatically obtained from plan execution attempts. Yordanova (2016) presents an approach which works with input text solution plans, as a proxy for instructions, and aims to learn pre- and post-condition action representations.

A similar increase in work aimed at automated creation has been seen in research in Narrative generation for Interactive Storytelling. However, an important difference with respect to narrative domains is that they do not share the same consistency and alignment with real-world domains as do more traditional benchmark planning domains. Hence approaches have tended to focus on (semi-)automated methods to gather story content, such as crowdsourcing, weblogs and story corpora. For example, crowdsourcing was used in: the SCHERAZADE system (Li et al. 2013) to acquire typical story elements that can be assembled as plot graphs and used in a process of story generation; SCENARIOGEN (Sina, Rosenfeld, and Kraus 2014) to gather a database of scenarios of everyday activities and likely replacements for use within a serious game context; and by (Nazar and Janssen 2010) for the hand annotation of logs from user sessions with the Restaurant Game for subsequent use in automating character interactions with human participants in a speech-based narrative setting. An alternative approach aims to obtain narrative content through mining of weblogs and story corpora. For example, SAYANYTHING (Swanson and Gordon 2012) selects narrative content on-the-fly from a corpora of weblogs in response to user text-based interaction, whilst (McIntyre and Lapata 2009) attempts to generate narratives using knowledge mined from story corpora for a particular genre.

Our work complements this, with input narrative content being mined from input natural language plot synopses.

StoryFramer Overview

Our approach to domain model generation is implemented in a system called StoryFramer, the main elements of which are shown in Figure 1. The system takes as input NL narrative synopses such as the outline story of Aladdin shown in Figure 2 and outputs a PDDL domain model and problem file such that the original story can be reproduced using a planner. The target output language is PDDL1.2 (Ghallab et al. 1998).

Currently the translation from NL to a domain model is a semi-automated process with the user in the loop for disambiguation of content at a number of stages. In this section we overview the main stages in this process and then illustrate it with an end-to-end example.

Extracting template representations from NL input

The first step in the approach is the generation of frame templates which are reduced representations of the input sentences. These templates capture the main action or property that a sentence is describing, as well as the objects mentioned and an indication of their roles within the sentence.

For this extraction we use Stanford CoreNLP (Manning

There is a woman named Jasmine. There is a king named Jafar. This is a story about how King Jafar becomes married to Jasmine. There is a magic genie. This is also a story about how the genie dies. There is a magic lamp. There is a dragon. The dragon has the magic lamp. The genie is confined within the magic lamp. There is a brave knight named Aladdin. Aladdin travels from the castle to the mountains. Aladdin slays the dragon. The dragon is dead. Aladdin takes the magic lamp from the dead body of the dragon. Aladdin travels from the mountains to the castle. Aladdin hands the magic lamp to King Jafar. The genie is in the magic lamp. King Jafar rubs the magic lamp and summons the genie out of it. The genie is not confined within the magic lamp. The genie casts a spell on Jasmine making her fall in love with King Jafar. Jasmine is madly in love with King Jafar. Aladdin slays the genie. King Jafar is not married. Jasmine is very beautiful. King Jafar sees Jasmine and instantly falls in love with her. King Jafar and Jasmine wed in an extravagant ceremony. The genie is dead. King Jafar and Jasmine are married. The end.

Figure 2: Aladdin outline plot from (Riedl and Young 2010)

et al. 2014), a publicly-available and widely-used annotation pipeline for natural language analysis. Of most relevance in this work are the syntactic parsing annotations that CoreNLP produces. Syntactic analysis in CoreNLP is a two-stage process. Firstly, phrase structure trees are generated using statistical analysis of datasets containing many examples of manually annotated sentence parses (Klein and Manning 2003). Secondly, these phrase structure trees are converted to dependency parse graphs using a series of manually-curated rules based on patterns observed in the phrase structure trees (de Marneffe, MacCartney, and Manning 2006). An example of the sort of dependency graphs that are output by CoreNLP is shown in Figure 3.

For our purposes the structure of these graphs must be further simplified to move closer to a predicate logic representation. This is achieved through a recursive set of rules that crawl the dependency graph, transforming the relations based on their types. CoreNLP use the Penn Treebank Project part-of-speech tags (Marcus, Marcinkiewicz, and Santorini 1993) to annotate the text. Most importantly, the root verb, subjects and objects form actions, predicates and domain objects as follows:

- The VBZ tag denotes verb (3rd person singular present) and this forms the basis of candidate action names. For example, the action `takes` in Figure 3
- The JJ tag denotes adjectives. These form the candidate properties e.g. the property `beautiful` in Figure 4.
- The NN tag (and variants NNS, NNP, NNPS) denote nouns; singular, plural, proper noun singular and proper noun plural respectively. These form the basis candidate objects (constants) for the domain. For example, `Aladdin` in Figure 3 and `Jasmine` in Figure 4.

Conjunctions in input sentences introduce new clauses, which themselves form further predicates. Other relation types such as modifiers and compounds are used to transform the names of the predicates and arguments.

Building Action Representations

Based on the CoreNLP annotations, StoryFramer creates a temporary action template which uses the verb as the action name and includes all the associated objects. An example template action for `takes` is shown in Figure 3. It can be observed that this template contains key elements of the action that will be output, namely, the name, arguments (the characters `aladdin` and `dragon`, and the `magic-lamp` object). We discuss this in more detail later (Worked Example section) following the rest of the StoryFramer overview.

Parameters For each action template the system labels each of the associated objects as candidate parameters for the output action. During the phase of user interaction these will be typed using the generic categories of: `character, object` and `location`.

Pre- and Post-conditions Following an approach similar to (Yordanova 2016), default predicates are added to the pre- and post-conditions of template actions, named `(can-`*action* `?x)` and `(has-`*action* `?x)` to introduce a baseline level of causality, sufficient to ensure generation of a baseline plan that corresponds to the original input NL story synopsis. For example, it may be necessary to use one of these predicates as part of the goal condition (this is the case with our Aladdin worked example).

Other predicates are added by the system as the PDDL domain files are output, following user interaction.

User Interaction

At this stage StoryFramer requires user interation and hence prompts for input to be used for the following:

- Removing Duplicates:

  Anytime the same object has been referred to in different ways in the NL input, the result is that the system finds multiple different objects. In this situation the user is asked to disambiguate. At the end of this stage every object should be represented by one unique identifier.
- Typing of Objects:

  The user is asked to sort the objects into types. So far in our experiments with narrative domains we have restricted this to the following small set of narrative categories: `character, object` and `location`.
- Action Pre and Post-conditions:

  The user is asked to select between possible pre- and post-conditions for inclusion in the domain model (i.e. "Do you want to include ...?"). These are of the following types:

  1) Predicates: the user is asked to select from identified predicates, such as `beautiful` in Figure 4, and use them to populate action preconditions where appropriate.

  2) Locatedness: some conditions are commonly missing from the NL input relating to the location of characters

NL *Aladdin takes the magic lamp from the dead body of the dragon.*

CoreNLP Annotation

```
[takes/VBZ
nsubj>Aladdin/NNP
dobj>[lamp/NN det>the/DT amod>magic/JJ]
nmod:from>[body/NN
  case>from/IN
  det>the/DT
  amod>dead/JJ
  nmod:of>[dragon/NN case>of/IN det>the/DT]]
punct./.]
```

Action Template

```
action : takes {
subject : aladdin
object : magic lamp
from : dead body
of : dragon }
```

NL *Aladdin travels from the castle to the mountains.*

CoreNLP Annotation

```
[travels/VBZ
nsubj>Aladdin/NNP
nmod:from>[castle/NN case>from/IN det>the/DT
nmod:to>[mountains/NNS case>to/TO det>the/DT
punct./.]
```

Action Template

```
action : travels {
subject : aladdin
from : castle
to : mountains
```

Figure 3: Action Template Examples. The figure shows sample NL sentence input, with CoreNLP annotation and resulting action template after rewrite rules: from CoreNLP annotation, **verb**, **subject** and **object** of the sentence form the action name and arguments (see text for further details).

and objects. For the work we present here we assume that all characters must always be "at" some location and that objects can be either "at" or in the possession of a character i.e. "has". Should such predicates be missing from the NL input, then users are asked to decide whether to include them.

3) Inequality: whenever an action template has multiple parameters of the same type the system assumes that these cannot be equal and prompts the user about inclusion of a `(not (= ?x1 ?x2))` precondition.

- Problem File setup:

  The final stage of user interaction is setting up a problem file. Every predicate detected by StoryFramer that was true at some time during the story represents a potential initial state fact or plan goal. The user is shown a list of facts and asked to delete those not appropriate, as well as adding any that were missed or not mentioned. For example, this frequently requires the selection of predicates relating to the location of characters in the initial state: something frequently missing from the input NL synopses.

  The user is also prompted to select suitable goal facts for the problem file. In our experiments we have used goal

NL *Jasmine is very beautiful.*

CoreNLP Annotation

```
[beautiful/JJ
nsubj>Jasmine/NN
cop>is/VBZ advmod>very/RB
punct./.]
```

Property Template

```
property : beautiful {
subject : jasmine }
```

NL *The genie is in the magic lamp.*

CoreNLP Annotation

```
[lamp/NN
nsubj>[genie/NN det>The/DT]
cop>is/VBZ
case>in/IN
det>the/DT
amod>magic/JJ
punct./.]
```

Property Template

```
in : magic lamp {
subject : genie }
```

NL *The dragon is dead.*

CoreNLP Annotation

```
[dead/JJ
nsubj>[dragon/NN det>The/DT]
cop>is/VBZ
punct./.]
```

Property Template

```
property : dead {
subject : dragon }
```

Figure 4: Property Template Examples. The figure shows sample NL input sentences, the CoreNLP annotations for this sentence with the property and subject taken from the sentence adjective and noun (for further details see text).

conditions which enable the generation of a plan which reproduces the story outline from the input NL story.

PDDL output

The final stage in the process is the generation of the domain and problem file content which is output as PDDL.

Following the user interaction to provide type information for action parameters and domain objects the system may add additional parameters to actions at this stage. This is based on an assumption that all actions must have an associated location as this is needed in the longer term building up of a narrative domain: for example, for staging of generated story plans in a virtual environment. In practice we have observed that location details are frequently missing from synopses. Hence additional location parameters are automatically added to actions where there is no location parameter associated with the action template from the input text.

For the domain file, the detail of the actions comes from the action templates with the types of action parameters added on the basis of the user input. Pre- and post-conditions are made up of predicates extracted from the input NL, and

selected for inclusion by the user, along with system suggested predicates such as inequality testing.

Worked Example: Aladdin

In this section we present a worked example of the end-to-end process of planning domain model and problem file generation with StoryFramer. This example uses the NL plot synopsis from (Riedl and Young 2010) shown in Figure 2.

1) StoryFramer Processing

CoreNLP handles the input NL text, one sentence at a time and for each sentence returns the text with annotations. For example, for the following input sentences:

S1 "Aladdin takes the magic lamp from the dead body of the dragon"
S2 "Aladdin travels from the castle to the mountains"
S3 "Jasmine is very beautiful"
S4 "The genie is in the magic lamp"

the resulting CoreNLP annotations are as shown in Figures 3 and 4. For the action templates StoryFramer identifies the key action components and also adds the `can-X` and `has-X` predicates to action pre- and post-conditions to ensure a baseline for action causal chaining. Thus the outline actions from S1 and S2 at this stage are:

	Name	Precondition	Postcondition
S1:	`takes aladdin, magic-lamp, dragon`	`can-takes`	`has-takes`
S2:	`travels castle, mountain`	`can-travel`	`has-travel`

For the property templates, resulting from input sentence S3 and S4, the key components form the basis of predicates with name and arguments as follows:

	Name	Arguments
S3	`beautiful`	`jasmine`
S4	`in`	`magic-lamp, genie`

At the end of this first phase of automated processing with StoryFramer the sets of initial action templates, predicates and domain objects are as summarised in Figure 5.

2) User Input

Firstly, the user is asked to remove duplicate references to the same objects. For example, in Figure 5 the set of objects contains a number of duplicates, such as `Jafar` and `King-Jafar` and the user has selected unique object identifiers and the duplicates have been removed.

Once duplicates have been resolved the user sorts the set of objects into types. In our experiments to date we have restricted this to `character, object` and `location`. For the Aladdin story, the results of this phase of user interaction result in object typing as shown in Figure 5.

Next the user is prompted to select and reject predicates to populate the pre- and post-conditions of the output actions. These predicates are obtained as follows :

Actions	Predicates	Objects
confined	at has woman	Jasmine
travels	king dead	Jafar
slays	knight in	Magic-Genie
takes gives	beautiful	Magic-Lamp
rubs casts	in-love	Dragon Genie
married sees	can-travels	Aladdin
wed	has-travels	Castle
	can-slays	Mountains
	has-slays	King-Jafar
	...	

Resolved Objects

(Jafar, King-Jafar)	→ Jafar
(Genie, Magic-Genie)	→ Genie

Typing

Jafar, Jasmine, Aladdin, ...	→ character
Castle, Mountain	→ location
Magic-Lamp	→ object

Figure 5: Results of initial phase of automated StoryFramer processing: the figure shows the sets of names of actions, predicates and objects identified prior to user interaction. Duplicate Object References are highlighted (red) along with the results of user input to remove duplicates. Also shown are the results of user sorting of objects into types.

1. Predicates from properties in the input NL sentences. For Aladdin a selection of these are shown in Figure 5)
2. Locatedness predicates, `at` and `has`, introduced by StoryFramer if they are absent in the input NL and representing the location of objects of type `character` and `object`.
3. Predicates ensuring unique object grounding of multiple parameters of the same type, i.e. inequality.

The action `takes` in Figure 6 shows examples of the results of user selection of these predicates and the building of the output action.

The final phase of user interaction is selection of predicates for the setting up of a problem file: the initial state and the goal conditions. In our experiments the user selected those predicates from the initial state and goal conditions which allowed us to regenerate the plan corresponding to the input NL plot synopsis.

3) PDDL Output

The final step of the process is the outputting of the Domain Model and Planning Problem as PDDL files. For this example these can be found online:
`https://drive.google.com/drive/folders/0B6Rv1Q3KYqtMcXhUMFFSZzh1a1U?usp=sharing`

Evaluation

In this section we present an evaluation to assess how accurate StoryFramer is. We evaluate StoryFramer with two domains: the tale of Aladdin taken from (Riedl and Young

(:action **takes**
:parameters
(?c1 ?c2 - character ?o - object ?l - location)
:precondition (and
1 (dead ?c2)
2 (at ?c1 ?l1) (at ?c2 ?l1) (has ?c2 ?o1)
(can-takes ?c1) (can-takes ?c2) (can-takes ?o1)
3 (not (= ?c1 ?c2)))
:effect (and
1 (has ?c1 ?o1) (not (has ?c2 ?o1))
(has-takes ?c1) (has-takes ?c2) (has-takes ?o1))))

Figure 6: Example of StoryFramer Building for action `takes`. Following user input the parameter object names from the input NL have been replaced by variables of the appropriate types. Precondition predicates have been: selected by the user 1; system suggested locatedness and chaining predicates have been retained by the user 2; the system has introduced inequality tests for objects of the same type 3. For the postconditions the user has retained locatedness and chaining predicates as shown 2.

2010) and an old American West story taken from (Ware 2014). We selected these two sources because they provide natural language descriptions that we can use as input and they include planning domains that we can use to compare with the domains generated by StoryFramer. In particular, we evaluate StoryFramer by comparing the set of recognised actions and predicates with the actions and predicates used in the selected domains.

The domains used in the evaluation can be found online using the link provided at the end of the worked example.

Domain 1: The tale of Aladdin

We used StoryFramer with two texts describing the tale of Aladdin. Both texts were taken from (Riedl and Young 2010): one is shown here, in Figure 2; the other is a variation (see Figure 13 of Riedl's paper). Table 1 shows that all actions but one were recognised by StoryFramer. When compared to Riedl and Young's planning domain, only one action is not recognised. All the other 11 actions are recognised with four of them being named based on the same verb. The action `marry` is recognised twice: as `wed` and as `married`.

In terms of predicates, out of Riedl and Young's 24 predicates, only two predicates that are mentioned in the text are not recognised by StoryFramer: the binary predicates `married-to` and `loyal-to` (however, a unary predicate `loyal` is recognised). Nine (9) are recognised in exactly the same way: one (1) as a type, one (1) as an object, one (1) as a constant, and six (6) as predicates. Eight (8) are recognised, but with a different name from the one that Riedl and Young used: four (4) of them are minor variations (e.g. `married`/`has-married` and `loves`/`in-love`); two (2) of them are recognised as words that appear in the text: instead of `alive` and `female`, the system recognised `dead` and `woman` (resp.); and two (2) of them are recognised as types (Riedl and Young used `thing` and `place` instead of `object` and `location`; in both cases, none of the words appear in the text). Finally, there are seven (7) predicates that were not recognised: five (5) of them do not appear in the text (`scary`, `monster`, `male`, `single`, and `intends`); and finally, as mentioned above, two (2) are mentioned in the text, but are not recognised.

Output Plan	**Corresponding NL sentences**
(sees jafar jasmine castle)	King Jafar sees Jasmine and instantly falls in love with her.
(travels aladdin castle mountains)	Aladdin travels from the castle to the mountains.
(slays aladdin dragon mountains)	Aladdin slays the dragon.
(takes aladdin dragon magic-lamp mountains)	Aladdin takes the magic lamp from the dead body of the dragon
(travels aladdin mountains castle)	Aladdin travels from the mountains to the castle.
(gives aladdin jafar magic-lamp castle)	Aladdin hands the magic lamp to King Jafar.
(rubs jafar genie magic-lamp castle)	King Jafar rubs the magic lamp and summons the genie out of it.
(casts genie jasmine jafar castle)	The genie casts a spell on Jasmine making her fall in love with King Jafar.
(slays aladdin genie castle)	Aladdin slays the genie.
(wed jafar jasmine castle)	King Jafar and Jasmine wed in an extravagant ceremony.

Figure 7: Output Plan and Corresponding Input NL sentences for the tale of Aladdin: on the left hand side are the 10 actions in the output plan generated using the learned domain model; alongside each action (right hand side) are the corresponding input sentences from the original story.

Output Plans We used the StoryFramer generated domain and problem file from the original Aladdin NL input to generate an output narrative plan (using METRIC-FF (Hoffmann and Nebel 2001)). The plan consists of the 10 actions which are shown in Figure 7, along with corresponding input.

Domain 2: An old American West story

We also used StoryFramer with natural language sentences taken from (Ware 2014). These are are part of an old American West story about how a young boy named Timmy is saved (or not saved) from a deadly snakebite. His father, Hank, can save him by stealing antivenom from Carl, the town shopkeeper, but this theft causes sheriff William to hunt down Hank and dispense frontier justice.

In the thesis Ware gives seven example solution plans and translations of them into NL. It is these NL sentences which we used as input to StoryFramer. We list them here and show in brackets the action names used by Ware:

- *Timmy died. (die)*
- *Carl the shopkeeper healed Timmy using his medicine.*

Riedl and Young (2010)	Recognised by StoryFramer
travel	✓(travels)
slay	✓(slays)
pillage	✓(takes)
give	✓(gives)
summon	✓(rubs)
love-spell	✓(casts)
marry	✓(wed,married)
fall-in-love	✓(sees)
order	✗
command	✓(uses)
appear-threatening	✓(appears)

Table 1: When compared to Riedl and Young's planning domain, only one action is not recognised by StoryFramer. All the other 11 actions are recognised with four of them being named based on the same verb. The action 'marry' is recognised twice: as 'wed' and as 'married'.

(heal)

- *Hank shot his son Timmy. (shot)*
- *Hank stole antivenom from the shop, which angered Sheriff William. (steal)*
- *Hank healed his son Timmy using the stolen antivenom. (heal)*
- *Sheriff William shot Hank for his crime. (shoot)*
- *Hank intended to heal his son Timmy using the stolen antivenom. (heal)*
- *Sheriff William intended to shoot Hank for his crime. (shoot)*
- *Hank got bitten by a snake. (snakebite)*
- *Hank intended to heal himself using the stolen antivenom. (heal)*

In Table 2, we show that all the actions used by Ware were recognised by StoryFramer. Note that whilst Ware used the action `heal` to model the actions mentioned in the sentences "*Hank healed...*" and "*Hank intended to heal*", StoryFramer recognised two different actions: `heal` for the first sentence and `intended` for the second.

In terms of predicates, results were not so good as with the tale of Aladdin. The predicates generated by StoryFramer are based on the text provided as input, so, besides predicates common in general narratives (e.g. `at ?c ?l` or `has ?c ?o`), StoryFramer generated predicates associated with the recognised actions (e.g. `has-died`, `can-shot`, and `has-bitten`). It also introduced as constants all the characters mentioned (`Hank`, `Timmy`, `Carl`, and `Sheriff`) and some objects and locations used in the narrative (`Medicine`, `Antivenom`, and `Shop`). On the other hand, Ware introduced a predicate `status ?p ?s` that is to be used with one of three constants: `Healthy`, `Sick`, or `Dead`. He also introduced predicates `owns ?c ?o`, `armed ?c`, and `parent ?c1 ?c2`.

This mismatch is justified because Ware is using predicates that are not mentioned explicitly in the sentences that we used as input. For example, the sentences do not make any reference to the words or states Healthy and Sick. We discuss how this limitation can be addressed in the section on future work.

Ware (2014)	Recognised by StoryFramer
die	✓(died)
heal	✓(healed)
shoot	✓(shot)
steal	✓(stole)
snakebite	✓(bitten)
✗	intended

Table 2: All the actions used by Ware (2014) were recognised. StoryFramer also recognised an additional action ('intended').

Output Plans Even though there is a clear mismatch between the predicates recognised by StoryFramer and the ones used by Ware, the generated domains can be used to produce all the seven plans listed for the Western Domain (see (Ware 2014, Fig. 3.3) and Figure 8). However, we note that for two of these plans, F and G, it was necessary to remove reasoning about intent: despite StoryFramer correctly generating an `intends` action from the NL input. This is because intention reasoning (a feature of Ware's COPCL planner and other narrative planners in the tradition of the IPOCL planner of (Riedl and Young 2010)) requires the use of a planner capable of intentional reasoning which is beyond the scope of our current work.

With intention removed we were able to generate the same plans as reported by Ware. The results are shown in Figure 8. Although we note that in order to reproduce the same ordering of actions it was necessary to use intermediate goals as described in (Porteous, Cavazza, and Charles 2010b). For example, to enforce the ordering that a goal `(has-shot sheriff hank)` occurs before another goal `(has-died timmy)` the problem is written in PDDL3.0 using the modal operator `sometime-before`[1] and the plan is generated using a decomposition approach that solves each subgoal in turn.

Conclusions and Future Work

In the paper we have presented an overview of our approach to automated domain model generation and its implementation in the prototype system StoryFramer. We assessed the performance of the approach on a couple of publicly available narrative planning domains, for which narrative synopses are also available.

An important aspect of the approach is that it is possible to go from NL input to an output domain model and problem instance, with which it is possible to generate a plan that corresponds to the original NL input. Much of this process is automated but user input may be required for some aspects such as disambiguation of content. However there is scope to further automate the process as part of the future work.

[1]The semantics of `(sometime-before A B)` requires that application of actions in solution plans make `B` true before `A`.

Plan	**Ware Plans**	**StoryFramer Plans**
A	(die Timmy)	(died timmy shop)
B	(heal Carl Timmy)	(healed carl-shopkeeper timmy medicine shop)
C	(shoot Hank Timmy)	(shot timmy hank shop)
D	(steal Hank Antivenom Carl William) (heal Hank Antinvenom Timmy)	(stole hank sheriff-william antivenom shop) (healed hank timmy antivenom shop)
E	(steal Hank Antivenom Carl William) (heal Hank Antinvenom Timmy) (shoot William Hank)	(stole hank sheriff-william antivenom shop) (healed hank timmy antivenom shop) (shot sheriff-william hank shop)
F	(steal Hank Antivenom Carl William) (shoot William Hank) <heal Hank Antivenom Timmy> (die Timmy)	(stole hank sheriff-william antivenom shop) (shot sheriff-william hank shop) - (died timmy shop)
G	(steal Hank Antivenom Carl William) <shoot William Hank> (snakebite Hank) <heal Hank Antivenom Hank> (heal Hank Antivenom Timmy)	(stole hank sheriff-william antivenom shop) - (bitten hank shop) - (healed hank timmy antivenom shop)

Figure 8: Comparison of output plans for the Western Domain from (Ware 2014): those listed by Ware and corresponding to the NL input to StoryFramer; and those generated by StoryFramer ignoring intent. See text for detail.

Amongst our plans for future work we are keen to exploit further the part-of-speech information provided by CoreNLP in combination with other linguistic resources in order to disambiguate content. We also intend to explore the use of a commonsense reasoning engine which would enable inference of aspects such as family and social relationships (e.g. from references such as "parent" in the input).

There are also possibilities to combine this with reasoning that is able to automatically extend an existing domain model, for example via the use of antonyms to find opposite actions, as in (Porteous et al. 2015).

We may also look to use multiple story synopses as input to incrementally build up a large domain ontology. This could for example be used to learn actions from multiple episodes of a series so that generated output plans can show more variation.

References

Aylett, R.; Dias, J.; and Paiva, A. 2006. An Affectively Driven Planner for Synthetic Characters. In *Proc. of 16th Int. Conf. on Automated Planning and Scheduling (ICAPS)*.

Bee, N.; Wagner, J.; André, E.; Charles, F.; Pizzi, D.; and Cavazza, M. 2010. Multimodal interaction with a virtual character in interactive storytelling. In *the 9th International Conference on Autonomous Agents and Multi-Agent Systems, AAMAS 2010*, 1535–1536.

Branavan, S. R. K.; Kushman, N.; Lei, T.; and Barzilay, R. 2012. Learning High-level Planning from Text. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics*, ACL '12, 126–135. Stroudsburg, PA, USA: Association for Computational Linguistics.

Cardona-Rivera, R. E., and Li, B. 2016. PLOTSHOT: Generating Discourse-constrained Stories Around Photos. In *Proceedings of the 12th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*.

Cavazza, M.; Pizzi, D.; Charles, F.; Vogt, T.; and André, E. 2009. Emotional input for character-based interactive storytelling. In *Proc. of the 8th Int. Joint Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*.

de Marneffe, M.-C.; MacCartney, B.; and Manning, C. D. 2006. Generating typed dependency parses from phrase structure parses. In *Proceedings of the Conference on Language Resources and Evaluation*, 449–454.

Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; ; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language. http://www.inf.ed.ac.uk/teaching/courses/propm/papers/pddl.html.

Gilroy, S. W.; Porteous, J.; Charles, F.; and Cavazza, M. 2012. Exploring Passive User Interaction for Adaptive Narratives. In *Proc. of the 17th Int. Conf. on Intelligent User Interfaces (IUI-12)*.

Goldwasser, D., and Roth, D. 2011. Learning from natural instructions. In *Proc. of the 22nd Int. Joint Conf. on Artifical Intelligence (IJCAI)*.

Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation throughHeuristic Search. *Journal of AI Research* 14:253–302.

Klein, D., and Manning, C. D. 2003. Fast exact inference with a factored model for natural language parsing. In *Advances in Neural Information Processing Systems*, volume 15. 3–10.

Kollar, T.; Perera, V.; Nardi, D.; ; and Veloso, M. 2013. Learning Environmental Knowledge from Task-based Human-robot Dialog. In *Proceedings of IEEE International Conference on Robotics and Automation (ICRA)*.

Li, B.; Lee-Urban, S.; Johnston, G.; and Riedl, M. 2013. Story Generation with Crowdsourced Plot Graphs. In *Proc. of the 27th AAAI Conf. on Artificial Intelligence*.

Lindsay, A.; Read, J.; Ferreira, J. F.; Hayton, T.; Porteous, J.; and Gregory, P. 2017. Framer: Planning Models from Natural Language Action Descriptions. In *Proceedings of the 27th International Conference on Automated Planning and Scheduling (ICAPS)*.

Manning, C. D.; Surdeanu, M.; Bauer, J.; Finkel, J. R.; Bethard, S.; and McClosky, D. 2014. The stanford corenlp natural language processing toolkit. In *The Annual Meeting of the Association for Computational Linguistics (System Demonstrations)*, 55–60.

Marcus, M. P.; Marcinkiewicz, M. A.; and Santorini, B. 1993. Building a large annotated corpus of english: The penn treebank. *Computational linguistics* 19(2):313–330.

Mateas, M., and Stern, A. 2005. Structuring Content in the Façade Interactive Drama Architecture. In *Proc. of the 1st Conf. on AI and Interactive Digital Entertainment (AIIDE)*.

McIntyre, N., and Lapata, M. 2009. Learning to Tell Tales: A Data-driven Approach to Story Generation. In *Proceedings of 47th Meeting of the Association for Computational Linguistics (ACL)*.

Mokhtari, V.; Lopes, L. S.; and Pinho, A. J. 2016. Experience-Based Robot Task Learning and Planning with Goal Inference. In *Proc. of the 26th International Conference on Automated Planning and Scheduling (ICAPS)*.

Nazar, R., and Janssen, M. 2010. Combining resources: Taxonomy extraction from multiple dictionaries. In *Proc. of the 7th Int. Conf. on Language Resources and Evaluation*.

Piacenza, A.; Guerrini, F.; Adami, N.; Leonardi, R.; Teutenberg, J.; Porteous, J.; and Cavazza, M. 2011. Changing video arrangement for constructing alternative stories. In *Proc. of the 19th ACM International Conference on Multimedia*.

Porteous, J.; Lindsay, A.; Read, J.; Truran, M.; and Cavazza, M. 2015. Automated Extension of Narrative Planning Domains with Antonymic Operators. In *Proc. of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*.

Porteous, J.; Cavazza, M.; and Charles, F. 2010a. Narrative Generation through Characters' Point of View. In *Proc. of 9th Int. Conf. on Autonomous Agents and MultiAgent Systems (AAMAS 2010)*.

Porteous, J.; Cavazza, M.; and Charles, F. 2010b. Applying Planning to Interactive Storytelling: Narrative Control using State Constraints. *ACM Transactions on Intelligent Systems and Technology (ACM TIST)* 1(2):1–21.

Porteous, J.; Charles, F.; and Cavazza, M. 2013. NetworkING: using Character Relationships for Interactive Narrative Generation. In *Proc. of 12th Int. Conf. on Autonomous agents and multi-agent systems (AAMAS)*. IFAAMAS.

Riedl, M. O., and Young, R. M. 2010. Narrative Planning: Balancing Plot and Character. *Journal of AI Research* 39:217–267.

Sil, A., and Yates, A. 2011. Extracting strips representations of actions and events. In *Recent Advances in Natural Language Processing (RANLP)*.

Sina, S.; Rosenfeld, A.; and Kraus, S. 2014. Generating content for scenario-based serious games using CrowdSourcing. In *Proceedings of 28th AAAI Conference on Artificial Intelligence (AAAI)*.

Swanson, R., and Gordon, A. S. 2012. Say Anything: Using Textual Case-Based Reasoning to Enable Open-Domain Interactive Storytelling. *ACM Trans. Interact. Intell. Syst.* 2(3).

Ware, S. G. 2014. *A plan-based model of conflict for narrative reasoning and generation*. Ph.D. Dissertation.

Yordanova, K. 2016. From Textual Instructions to Sensor-based Recognition of User Behaviour. In *Proc. of 21st Int. Conf. on Intelligent User Interfaces*, IUI Companion. ACM.

A PDDL Representation for Contradance Composition

Richard G. Freedman and **Shlomo Zilberstein**
College of Information and Computer Sciences
University of Massachusetts Amherst
{freedman, shlomo}@cs.umass.edu

Abstract

Classical planning representations such as PDDL are primarily designed for goal-oriented problem solving, but some tasks such as creative composition lack a well defined goal. Structured performing arts, despite lacking a specific goal for their composition tasks, can be sufficiently expressed as goal-oriented problems for their performance tasks. Using contradance as an example performing art, we show how to represent individual contradances as plans such that the composition task can be compiled into a performance problem that can be expressed in PDDL. That is, by accounting for additional properties that are useful in their composition, the solutions to the performance task also serve as solutions to the composition task. We conclude with some example contradances derived using a classical planner under various composition conditions.

1 Introduction

As one of the earliest challenges in artificial intelligence, classical planning has often focused on automated problem solving where tasks have well defined goals. However, creative tasks such as artistic composition, ranging from writing music to dance and martial arts choreography, lack well defined goals outside of completing the work (to avoid an infinite loop of creation). Early research in creative artificial intelligence (Schmidhuber 2010) identified features such as constructing new things from simple patterns that cannot be easily expressed by past observations, but these features serve more as heuristics than actual approaches. Increased interest in creative machines has led to the formulation of the Lovelace Tests (Bringsjord, Bello, and Ferrucci 2001; Riedl 2015) and work in dynamic storytelling agents (Riedl and Young 2010; Amos-Binks 2017). We differentiate these tasks from the traditional ones as follows:

Definition 1 *A* goal-oriented task *in domain $\mathcal{D}$ is one for which a solution is any plan or policy π that yields a state satisfying the set of goal conditions. This includes finding optimal solutions as part of the task because any optimal solution is acceptable if more than one exists.*

Definition 2 *A* composition task *in domain $\mathcal{D}$ is one for which a solution is a plan or policy π that not only yields a state satisfying the set of goal conditions, but also satisfies specific intrinsic properties. These properties may include (but are not limited to) having particular action subsequences, following various rules describing π's structure, and expressing a desired message.*

In this work, we will investigate the creative task of contradance composition. Originating from Irish folk dancing, contradancing is a popular casual group dance in present times. The dancers form two long lines that usually break down into groups of four dancers. A caller announces a sequence of moves (called figures) that the dancers perform to reposition themselves within their group of four, sometimes mixing groups in more complicated sequences. The final figure in the sequence switches the positions of the two pairs of dancers to form new groups of four; this progression through the line continues as the sequence of figures loops until the song ends.

The steady progression and repositioning of dancers within their groups is well-structured with various mathematical properties (Peterson 2003; Copes 2003). Although these can be used to find the set of all possible contradances, composers select subsets of these contradances due to their artistic preferences and what they believe the dancers will enjoy. Thus *simply knowing how figures will alter the positions of dancers is not sufficient*. A random contradance generator created by a computer science professor (Frederking Publication Date Unavailable), which uses depth-first search to choose the next node to expand by a user-provided seed, even warns users that the dance is not guaranteed to feel right.

These systematic approaches all simply consider the rearrangement of dancers and select moves to accomplish these transitions. However, contradance composers have revealed that more than position is used when they create their sequences of figures (Dart 1995). Enumerating the state space with additional features typically leads to exponential scaling, but using a first-order logic representation of the state can help reduce these impacts for knowledge engineering. Furthermore, we can take advantage of other features of PDDL (McDermott et al. 1998; Fox and Long 2003; Gerevini et al. 2009) to properly formulate the progression as a classical planning problem such that off-the-shelf planners can find dances as sequences of figures. We begin with a background of the contradancing domain in Section 2. Section 3 uses these details to illustrate how to represent the states and actions. With the PDDL representation derived,

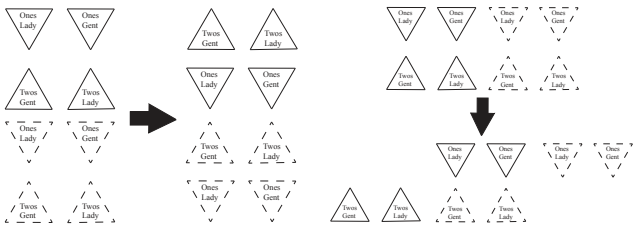


Figure 1: An illustration of progression for duple improper (left) and Becket formation (right). The couple that reaches the end changes between ones and twos to begin moving in the opposite direction with each progression.

we empirically present some of the contradances composed by off-the-shelf planners in Section 4 and discuss how this can be utilized and extended in Section 5.

2 Description of the Contradance Domain

A contradance is a well-structured community dance that has a formulaic procedure for set up and performance. Dancers are initially paired into *couples*, one *partner* in a couple is assigned the *gent* role and the other partner is assigned the *lady* role. The present-day interpretation of the roles is not indicative of who is dancing, but the lady is often located to the right of the gent and a few figures have different steps for each role working together as a couple. Depending on the choreography of the dance, couples are grouped together; the majority of dances have *duples* with two couples, and we will assume duples for our state space definitions in Section 3. In a duple, each couple faces each other where the couple facing the front of the dance hall (where the caller is located) is called the *ones* and the couple facing the back of the dance hall is called the *twos*. The ones gent and twos lady are called *neighbors* as are the ones lady and twos gent. All the duples are then joined to form two lines of dancers; it is called a *duple improper* when the neighbors are in the same line and a *Becket formation* when the partners are in the same line.

After setting up the dancers, the caller will announce a sequence of figures for everyone to perform. The steps in most figures only involve dancers within a duple so that each duple is dancing independently in parallel — some more complicated dances will have a *shadow* role for dancers between different duples who will interact with each other. However, the community still dances together even without a shadow due to the *progression*. The progression, resulting from the sequence of performed figures, moves the ones couple into the next duple closer to the front of the dance hall and the twos couple into the next duple closer to the back of the dance hall. See Figure 1 for an illustration of the progression and lines setup. After the progression, the sequence of figures is repeated using the newly formed duples; thus partners remain constant throughout the entire dance while neighbors change with each progression.

The length of time between progressions is sixty-four beats of music divided into four sets of sixteen beats. The music is divided into two sixteen-beat phrases that are each played twice during a single performance of the sequence of figures. Although this musical feature does not affect the dancers, the choreography uses this such that no figure is performed between two sets. That is, a figure must end when a musical phrase ends; this produces four partitioned subsequences of figures that each last sixteen beats. Figures vary in duration of beats from one beat to eight beats, which yields some of the variety between dances. To avoid monotony, most figures can only be repeated a specific number of times in a row. However, one figure is typically expected in every contradance: the swing. A swing is most often performed between partners or neighbors where the gent and lady hold waists/shoulders and spin clockwise about their center axis for eight beats, ending with the lady positioned to the right of the gent. Many contradances have one swing with the partner and another with the neighbor, but each swing is never done more than once per sequence of figures. After many progressions, the music eventually stops looping and ends the dance's performance.

3 Contradancing Representation in PDDL

The PDDL representation of any domain is $\mathcal{D} = \langle F, A \rangle$ where F is the set of fluents such that 2^F is the state space and A is the set of actions that can alter the states via add and delete effects $add\,(a \in A)\,, del\,(a \in A) \in F$ if their preconditions $pre\,(a \in A) \in F$ are satisfied. A problem in a given domain is represented as $\mathcal{P} = \langle \mathcal{D}, I, G \rangle$ where $I \in 2^F$ is the initial state and $G \subseteq F$ represents the goal conditions that must be satisfied for the task to be completed. When using first-order logic, we introduce the set of object types T into $\mathcal{D}$ and actual objects of type $t \in T$, $\mathcal{O}_t$, into $\mathcal{P}$ such that the fluents and actions are lifted over a tuple of parameters $params\,(x \in P \cup O) \in T^*$:

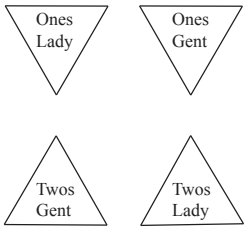
$$F = \bigcup_{p \in P} \bigotimes_{t \in params(p)} \mathcal{O}_t\ ,\ A = \bigcup_{o \in O} \bigotimes_{t \in params(o)} \mathcal{O}_t$$

where P is the set of propositions, O is the set of operators, and $*$ is the Kleene closure for any sequence of zero or more elements from a set.

For the contradance domain, we will follow the works in mathematics that view the layout of dancers in the dance hall as the state space and the figures as operators that alter this layout. Because a single caller dictates the figures and all the dancers follow these instructions, we view contradancing as a *centralized multi-agent problem*, which can be represented as a single agent problem where each action dictates what all the agents (dancers) do at once. Hence our type set T contains $dancer$, $location$, and $direction$ for each agent and the layout. We will also need T to contain $beat$ and set for temporal purposes when defining the operators.

Contradancing States

The dance hall will be laid out and connected in a manner similar to the traditional GRIDWORLD domain because contradancers are always lined up with each other before/after the performance of each figure. This gives us the propositions $adjacent$ and $same\_line$ where $params\,(adjacent) = (location, location, direction)$ and




```
(at_loc ones_lady 11)   (adjacent 11 12 left)   (same_line 11 21)
(at_loc ones_gent 12)   (adjacent 12 11 right)  (same_line 21 11)
(at_loc twos_gent 21)   (adjacent 21 22 left)   (same_line 12 22)
(at_loc twos_lady 22)   (adjacent 22 21 right)  (same_line 22 12)
(facing ones_lady 21)   (adjacent 11 21 back)
(facing ones_gent 22)   (adjacent 21 11 front)
(facing twos_gent 11)   (adjacent 12 22 back)
(facing twos_lady 12)   (adjacent 22 12 front)
```

Figure 2: A duple of dancers presented visually with its translation into PDDL.

$params\,(same\_line) = (location, location)$. For GRIDWORLD, directions are simply front, back, right, and left. Due to the lack of transitivity and symmetry between relations in PDDL, a problem's initial state must include all $2n\,(n-1)$ $same\_line$ fluents that are true to form the two lines of length n locations. Similarly, the problem must identify all pairs of locations for $adjacent$ with the directions being opposite. We present a visualization of a grid with a duple and these relations in Figure 2.

The dancers presented in the duple are positioned at locations using proposition at_loc where $params\,(at\_loc) = (dancer, location)$. Although this is sufficient for the mathematical representations, there are additional dancer features that a contradance composer keeps in mind to avoid awkward figures that the mathematical and automated approaches currently encounter. The simpler set of features identify roles because some figures require specific positioning of the gents and ladies; thus $params\,(role\_gent) = params\,(role\_lady) = params\,(role\_1s) = params\,(role\_2s) = (dancer)$ and $params\,(partner) = params\,(neighbor) = params\,(shadow) = (dancer, dancer)$ where the symmetry between these relations must again be defined.

The more complex feature to consider is based on the dancers' flow from the previous figure. If a dancer is moving forward and then told to interact with the dancer to the left by passing right shoulders, there will be some difficulty because the dancer needs to turn to the left and slow down enough to coordinate the shoulder passing — it would be more natural to pass by the left shoulder in this case due to the turn. Although computing a specific velocity in each direction can be challenging, we have found it sufficient to represent the flow by the direction that a dancer is facing; $params\,(facing) = (dancer, location)$ denotes the location in front of the specified dancer. This means a dancer may be facing at an angle if the location is not adjacent in the grid, which happens in several common figures.

Contradancing Initial and Goal States

In addition to the initial state providing the layout of locations in the dance hall, it must also set up the time (first beat in the first set), dancers, and their roles. In particular, because the lines may be formed by an arbitrary number of duples of dancers, it is only necessary to plan for the fewest number of duples who will perform the figures together, ignoring those dancing in parallel independently:

Proposition 1 *It is sufficient to define a contradance planning problem $\mathcal{P}$ containing $4\,(1+2k)$ agents/dancers where k is the number of duples between a given dancer and its respective shadow in the line, but it is only necessary to define $\mathcal{P}$ containing $4\,(1+s)$ agents/dancers where s is the specific number of shadows assigned to a dancer – usually $s \in \{0, 1\}$.*

Proposition 1 needs to consider k duples on either side of the couple because shadows become adjacent to their dancers through temporary progressions that take place within the four sets of sixteen beats. Thus the ones couple's shadows are located k duples in one direction while the twos couple's shadows are located k duples in the opposite direction because the ones and twos couples move in opposite directions during progression. Since all duples perform the same figures in parallel, it is also possible to control all the dancers at once regardless of how many duples are present in the problem. However, to reduce the overhead of keeping track of so many dancers, it is only necessary to represent one duple (for the actual choreography) and the specific dancers who serve as the shadows for each dancer in this duple (to check preconditions for figures involving shadows). Each shadow has a role in its own duple, and figures may manipulate the shadows as they affect the respective dancers of each role in the represented duple.

Proposition 2 *It is necessary and sufficient to define a contradance planning problem $\mathcal{P}$ containing $8\,(1+d)$ locations where $d = (1+2k)$ is the number of duples in Proposition 1.*

In Proposition 2, $4d$ of the locations are used for the two lines of d consecutive duples of dancers; this much of the dance hall must be present for the shadows to meet from their initial locations. The remaining $4\,(2+d)$ locations form a border around the two lines in order to accommodate *facing outwards*. The borders that are parallel to the lines do not have the $same\_line$ proposition hold true, but satisfy the respective $adjacent$ propositions. See Figure 3 for this initial state layout.

Although most the bordering locations are not guaranteed to be used in a given sequence of figures, at least a few of them are required for the goal condition of completing a progression. Figure 4 has PDDL translations of two initial and goal states for single duples without shadows (based on Figures 2 and 3). When starting from a duple improper, the couples who reach the ends of the lines should be facing

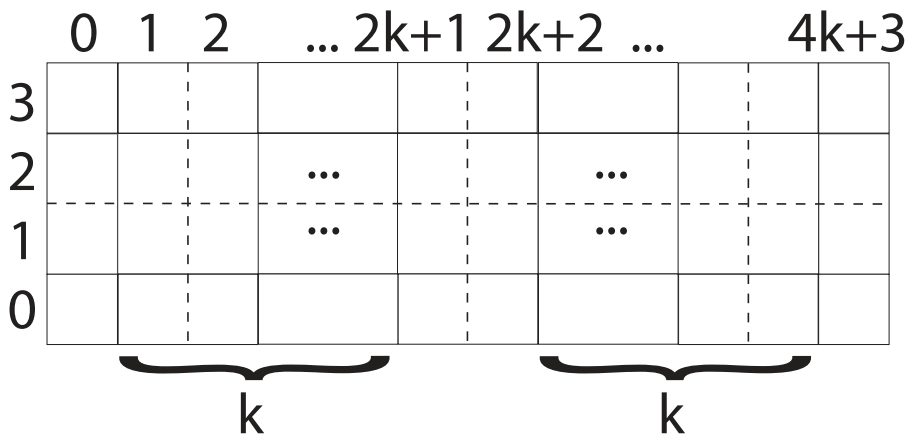


Figure 3: The grid layout of the dance hall described as the initial state. It is rotated 90 degrees counterclockwise so that the left direction is on the bottom. Thus rows 1 and 2 serve as the two lines while rows 0 and 1 and columns 0 and $4k$+3 are the border.

these border locations. When starting from a Becket formation and progressing clockwise, the couples who reach the ends of the lines should have the gent (who is to the left of the lady) at a location along this border. In addition to the changes in dancer location (although $facing$ changes for the dancers, their directions are preserved), the goal conditions should update the temporal information to the last beat of the last set (or the first beat of the set after the last, depending on implementation). This creates the task of *producing a progression in exactly the specified amount of time*, which is the goal of contradance performance.

Contradancing Actions

Given the layout of dancers throughout the dance floor, the figures will alter their locations and facing directions. Most operators represent a single figure whose parameters are the temporal beat assignment, dancers who will perform the figure, and their locations; this is usually the entire tuple, but there are figures where only two of the dancers move and the others remain still. However, dancers cannot perform a figure if they are not in the correct layout nor does a figure flow comfortably if they are facing the wrong directions with each other. This is where the preconditions are necessary to identify when a figure may be performed. The common features of operator preconditions for the contradance domain are:

- Enough beats remain in the set to perform the figure
- All the dancers in the parameters are different
- Each dancer is at the location specified in the parameters
- The locations are laid out correctly for the performance
- The dancers are facing the correct locations for flow

Likewise, the common components of operator effects (both for add and delete) for the contradance domain are:

- Increment the current beat by the duration of the figure
- Permute the dancers' locations if they change
- Change the dancers' directions if they change
- Increase the path cost if operator costs are used to guide the planner (see Section 4)

We call the set of operators representing figures $O_{figures}$, and the remaining operators are used to realign the temporal and spatial inforation $O_{realign} = O - O_{figures}$. The realignment operators do not have any cost and allow semantically equivalent representations to be used in the state space such as incrementing the set, which resets the beats from sixteen to zero to allow the next set of figures to begin, and repositioning the dancers as continuation of their flows (see Figure 5 for some examples). Although these operators could be embedded within the effects of figure operators, it is easier from a compositional perspective to observe the distinct transition in sets and movement of dancers. This also facilitates the caller's job as there is an explanation of what the dancers should do to position themselves for each figure's performance.

Counting Consecutive Figures If the number of repetitions is also considered as a constraint to reduce monotony in possible plans, then we must make several simple modifications. First the type 'repetitions' must be included in T. This allows us to include propositions of the form $consecutive\_\odot$ for each operator $\odot \in O_{figures}$ where $params\,(consecutive\_\odot) = (repetitions)$. Then each operator contains preconditions that ensure the maximum number of repetitions is not yet achieved for that figure, and the effects reset the repetition counts for all other operators while incrementing the repetition count for the performed figure.

Representation of Counting Although the use of numerical fluents (not to be confused with the domain fluents) has been considered a feature of PDDL since version 2.1 (Fox and Long 2003), many off-the-shelf classical planners limit their use to computing plan cost for optimization. So despite the ease of representation as integers with functions increment, assign, and compare to handle the beat, set, and rep-

```
(:init
 ... ;Set up the dance floor grid
 (at_loc ones_lady 11)
 (at_loc ones_gent 12)
 (at_loc twos_gent 21)
 (at_loc twos_lady 22)
 (facing ones_lady 21)
 (facing ones_gent 22)
 (facing twos_gent 11)
 (facing twos_lady 12)
 (current_set s0)
 (current_beat b0)
)
(:goal
 (at_loc ones_lady 21)
 (at_loc ones_gent 22)
 (at_loc twos_gent 11)
 (at_loc twos_lady 12)
 (facing ones_lady 31)
 (facing ones_gent 32)
 (facing twos_gent 01)
 (facing twos_lady 02)
 (current_set s4)
 (current_beat b0)
)
```

```
(:init
 ... ;Set up the dance floor grid
 (at_loc ones_lady 21)
 (at_loc ones_gent 11)
 (at_loc twos_gent 22)
 (at_loc twos_lady 12)
 (facing ones_lady 22)
 (facing ones_gent 12)
 (facing twos_gent 21)
 (facing twos_lady 11)
 (current_set s0)
 (current_beat b0)
)
(:goal
 (at_loc ones_lady 11)
 (at_loc ones_gent 01)
 (at_loc twos_gent 32)
 (at_loc twos_lady 22)
 (facing ones_lady 12)
 (facing ones_gent 02)
 (facing twos_gent 31)
 (facing twos_lady 21)
 (current_set s4)
 (current_beat b0)
)
```

Figure 4: PDDL representations of initial state and goal conditions for dancers starting from duple improper (left) and Becket formation (right). The goal conditions create a progression after the four sets of sixteen beats elapse.

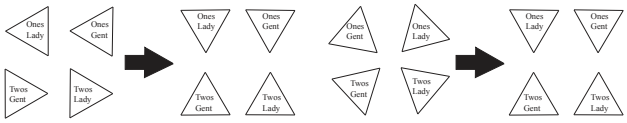


Figure 5: Realignment actions allow dancers to continue their flow for repositioning and match the preconditions of the next figure. The left image shows the flow of two dancers facing outwards to rotate towards the dancers on their left (who also rotate to face them). The right image shows the flow of dancers facing each other along the diagonals to continue moving until they face each other in the same line.

etition counts, it is not a feasible representation if we want to employ current planners to solve the contradance composition task. Therefore, we instead use conditional effects (part of the adl requirement) with the integers as domain constants of each type, and then enumerate all the precondition comparisons as a conjunction of negated equivalence checks as well as all the effect increments as a conjunction of conditional statements (hard-coding the increment). Figure 6 compares the numerical fluent and conditional effect PDDL representations.

4 Empirical Exploration of Composition

For initial evaluation of our formulation and representation, we encoded the contradancing PDDL domain with the six elementary figures detailed in Table 1. The swing is omitted because despite being a staple figure in any composition, its result is nondeterministic when considering flow — the lady ends to the right of the gent, but the direction is ambiguous depending on whether it is intended for a progression or part of the current set. We also included the realignment operator for incrementing the set after sixteen beats. When this domain was run in the state-of-the-art FastDownward Planner (Helmert 2006) with a problem that contained no shadows and started at the first beat of the first set in a duple improper formation, we received the following plan as the system's contradance composition:

```
Beat0, Set0: Do Si Do (Neighbors)
Beat4, Set0: Right and Left Through
Beat8, Set0: Right and Left Through
Beat12, Set0: Do Si Do (Neighbors)
Beat16, Set0: Update to Set1
---
Beat0, Set1: Right and Left Through
Beat4, Set1: Right and Left Through
Beat8, Set1: Do Si Do (Neighbors)
Beat12, Set1: Right and Left Through
Beat16, Set1: Update to Set2
---
Beat0, Set2: Right and Left Through
Beat4, Set2: Do Si Do (Neighbors)
Beat8, Set2: Right and Left Through
Beat12, Set2: Right and Left Through
Beat16, Set2: Update to Set3
---
```

```
(:action numeric_fluent_version
 :parameters (...)
 :precondition
 (and
   ;Must be within the time
   (< (current_beat) 13)
   ... ;Check locations, flow, and roles
 )
 :effect
 (and
   ;Add 4 beats to the set
   (increase (current_beat) 4)
   ... ;Update locations and flow
 )
)

(:action conditional_effect_version
 :parameters (... ?b - beat)
 :precondition
 (and
   ;Must be within the time
   (current_beat ?b)
   (not (= ?b b13))
   (not (= ?b b14))
   (not (= ?b b15))
   (not (= ?b b16))
   ... ;Check locations, flow, and roles
 )
 :effect
 (and
   ;Add 4 beats to the set
   (not (current_beat ?b))
   (when (= ?b b0) (current_beat b4))
   (when (= ?b b1) (current_beat b5))
   (when (= ?b b2) (current_beat b6))
   (when (= ?b b3) (current_beat b7))
   (when (= ?b b4) (current_beat b8))
   (when (= ?b b5) (current_beat b9))
   (when (= ?b b6) (current_beat b10))
   (when (= ?b b7) (current_beat b11))
   (when (= ?b b8) (current_beat b12))
   (when (= ?b b9) (current_beat b13))
   (when (= ?b b10) (current_beat b14))
   (when (= ?b b11) (current_beat b15))
   (when (= ?b b12) (current_beat b16))
   ... ;Update locations and flow
 )
)
```

Figure 6: The PDDL code for counting using numerical fluents (top) and conditional effects (bottom) when a figure has a duration of four beats.

Table 1: Contradance Figures Implemented

Name	**Duration (Beats)**	**Maximum Repetitions**
Circle to the Left	1	4
Circle to the Right	1	4
Do Si Do	4	1
Long Lines Forward and Back	8	1
Pass Through	2	3
Right and Left Through	4	2

```
Beat0, Set3: Do Si Do (Neighbors)
Beat4, Set3: Circle to the Left
Beat5, Set3: Long Lines Forward and Back
Beat13, Set3: Circle to the Left
Beat14, Set3: Pass Through (Progression)
Beat16, Set3: Update to Set4
```

The first three sets display very little variety and avoid circling to the left or right, which have the shortest duration of all the implemented figures. This decision can be explained by the fact that FastDownward is a goal-oriented optimal planner. That is, it will find the plan with least total action cost as the solution. For a formulation without defined/with uniform operator costs, this means that figures with longer duration will be preferred over figures with shorter duration because fewer figures are performed to reach the final beat of the final set. Thus the *composition cost* of a contradance operator $o \in O_{figure}$ is $cost\,(o)\,/duration\,(o)$.

Definition 3 *The* composition cost *of an action is the cost of selecting it for a plan rather than executing it. The composition cost should be lesser if it yields a greater reduction to the composition effort, has more desirable qualities for the plan requirements, etc..*

If it was possible to set up the dancers for the 'Long Lines Forward and Back' figure without performing 'Circle to the Left', then we would have seen it performed in the first three sets above because it has the cheapest composition cost $1/8$. However, the required 'Circle to the Left' would cost an additional action and offset the beat to odd parity where circling is the only figure that could return the beat to an even parity for completing the set. Thus it would cost three actions for ten beats, requiring at least two more actions to complete the set for a total of five actions. This costs more than four actions with four beats of duration each.

Because FastDownward can optimize over a total cost numeric fluent, we then re-evaluated our domain with assigned costs for each operator representing a figure. In particular, we identified composition costs for each figure using the formula above and then converted them to actual operator costs. The first case assigned all figures a uniform composition cost: $cost\,(o) = duration\,(o)$. Thus all solutions now have the same total path cost and the returned plan will depend on which solution is found first. The resulting plan found was:

```
Beat0, Set0: Right and Left Through
Beat4, Set0: Do Si Do (Neighbors)
Beat8, Set0: Right and Left Through
Beat 12, Set0: Do Si Do (Neighbors)
```

```
Beat 16, Set0: Update to Set1
---
Beat0, Set1: Right and Left Through
Beat4, Set1: Do Si Do (Neighbors)
Beat8, Set1: Right and Left Through
Beat 12, Set1: Do Si Do (Neighbors)
Beat 16, Set1: Update to Set2
---
Beat0, Set2: Right and Left Through
Beat4, Set2: Do Si Do (Neighbors)
Beat8, Set2: Right and Left Through
Beat 12, Set2: Do Si Do (Neighbors)
Beat 16, Set2: Update to Set3
---
Beat0, Set3: Right and Left Through
Beat4, Set3: Do Si Do (Neighbors)
Beat8, Set3: Right and Left Through
Beat12, Set3: Circle to the Left
Beat13, Set3: Circle to the Right
Beat14, Set3: Pass Through (Progression)
Beat16, Set3: Update to Set4
```

Lastly, to introduce a composer's preference for some figures over others, we assigned composition cost to be the duration $cost\,(o) = duration\,(o)^2$ so that figures with shorter duration have lesser cost (the opposite of the uniform operator cost case) and received the following composition (we only present the first and last set due to space — the first three sets are identical):

```
Beat0, Set0: Circle to the Left
Beat1, Set0: Circle to the Right
Beat2, Set0: Circle to the Left
Beat3, Set0: Circle to the Right
Beat4, Set0: Circle to the Left
Beat5, Set0: Circle to the Right
Beat6, Set0: Circle to the Left
Beat7, Set0: Circle to the Right
Beat8, Set0: Circle to the Left
Beat9, Set0: Circle to the Right
Beat10, Set0: Circle to the Left
Beat11, Set0: Circle to the Right
Beat12, Set0: Circle to the Left
Beat13, Set0: Circle to the Right
Beat14, Set0: Circle to the Left
Beat15, Set0: Circle to the Right
Beat16, Set0: Update to Set1
---
...
---
Beat0, Set3: Circle to the Left
Beat1, Set3: Circle to the Right
Beat2, Set3: Circle to the Left
Beat3, Set3: Circle to the Right
Beat4, Set3: Circle to the Left
Beat5, Set3: Circle to the Right
Beat6, Set3: Circle to the Left
Beat7, Set3: Circle to the Right
Beat8, Set3: Circle to the Left
Beat9, Set3: Circle to the Right
Beat10, Set3: Circle to the Left
Beat11, Set3: Circle to the Right
Beat12, Set3: Circle to the Left
Beat13, Set3: Circle to the Right
Beat14, Set3: Pass Through (Progression)
Beat16, Set3: Update to Set4
```

Using a diverse planner (Nguyen et al. 2012; Roberts, Howe, and Ray 2014) to solve these problems would be more ideal for composition tasks because there are a variety of plans with the same total cost that should be considered rather than just the first one found. Most planners have a deterministic procedure for tie-breaking during the generation of successor states; hence the plans found above are repetitive even with the consecutive figure constraints.

5 Discussion

Unlike goal-oriented tasks where the problem simply needs to be solved, composition tasks need to solve the problem with stylistic preferences. Using the performing art of contradance, we defined a state space and actions that force off-the-shelf classical planners to find sequences that are not only solutions to completing a performance, but exhibit desired intrinsic qualities such as respecting the flow of dancers, avoiding too many consecutive repetitions of figures, and including more preferred figures via composition cost. We believe that this compilation process may be used to solve other composition tasks via their goal-oriented counterparts. There are many potential applications for this work including the derivation of PDDL for similar composition tasks like square dancing, developing tools that allow human composers to receive recommendations for partially-composed dances with respect to their creative interests, and creating novel contradance figures to make unique dance patterns work (similar to Zook and Riedl's (2014) approach for developing game mechanics). These aspects will motivate and guide future research. We will also explore more complex composition cost formulas and experiment with other planners, such as diverse planners (Nguyen et al. 2012; Roberts, Howe, and Ray 2014), to investigate how they approach solving the compiled composition task.

Acknowledgements

We thank the anonymous reviewers for their feedback. This work was supported in part by NSF grant 1405550.

References

Amos-Binks, A. 2017. Problem formulation for accommodation support in plan-based interactive narratives. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence: Papers from the Doctoral Consortium*.

Bringsjord, S.; Bello, P.; and Ferrucci, D. 2001. Creativity, the Turing Test, and the (better) Lovelace Test. *Minds and Machines* 11(1):3–27.

Copes, L. 2003. Counting contra dances: From a talk given for the mathematical association of america. http://www.larrycopes.com/contra/MAA.html. [Online; posted after January-2003].

Dart, M. M. 1995. *Contra Dance Choreography: A Reflection of Social Change*. New York, New York, USA: Garland Publishing, Inc.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20(1):61–124.

Frederking, R. E. Publication Date Unavailable. Dr. Bob's random contra dances. http://www.cs.cmu.edu/∼ref/random-contras.html. [Online; last accessed 18-March-2017].

Gerevini, A. E.; 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(56):619–668. Advances in Automated Plan Generation.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26(1):191–246.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The planning domain definition language – Version 1.2. Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control, New Haven, CT, USA.

Nguyen, T. A.; Do, M.; Gerevini, A. E.; Serina, I.; Srivastava, B.; and Kambhampati, S. 2012. Generating diverse plans to handle unknown and partially known user preferences. *Artificial Intelligence* 190:1–31.

Peterson, I. 2003. Contra dances, matrices, and groups. Science News, https://www.sciencenews.org/article/contra-dances-matrices-and-groups. [Online; posted after 5-March-2003].

Riedl, M. O., and Young, R. M. 2010. Narrative planning: Balancing plot and character. *Journal of Artificial Intelligence Research* 39(1):217–268.

Riedl, M. O. 2015. The Lovelace 2.0 Test of artificial creativity and intelligence. In *Proceedings of the AAAI Workshop: Beyond the Turing Test*.

Roberts, M.; Howe, A.; and Ray, I. 2014. Evaluating diversity in classical planning. In *Proceedings of the Twenty-Fourth International Conference on Planning and Scheduling*, 253–261.

Schmidhuber, J. 2010. Formal theory of creativity, fun, and intrinsic motivation (1990-2010). *IEEE Transactions on Autonomous Mental Development* 2(3):230–247.

Zook, A., and Riedl, M. O. 2014. Automatic game design via mechanic generation. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14, 530–536. Québec City, Québec, Canada: AAAI Press.

Integrating Modeling and Knowledge Representation for Combined Task, Resource and Path Planning in Robotics

Simone Fratini, Tiago Nogueira*, Nicola Policella
European Space Agency, ESA/ESOC
Darmstadt, Germany
name.lastname@esa.int

Abstract

Robotics requires the integration of three heterogeneous planning processes: logical task planning, temporal resource allocation planning and path planning. On one hand the integration of these different classes of knowledge in a unified model introduces rich modeling features and facilitates optimization. On the other hand the different planning processes usually require, to practically solve not trivial instances of the respective problems, dedicated planners and models. This paper proposes an integration of a PDDL model in a constraint-based time-flexible temporal planning framework to smoothly coordinate task, temporal, resource and path planning. The proposal is evaluated in a domain inspired by a scenario where a UAV has to be operated to move objects in a warehouse.

Introduction

Solving robotics problems often requires reasoning on data and information that cannot be represented using symbolic planning, if not at the cost of very complex model and, therefore, inefficient solving processes. Examples include object manipulation in industrial robotics, path and motion planning in mobile robotics, or trajectory and attitude guidance and control in satellite systems.

To effectively handle planning problems in these scenarios typically involves: (1) high-level task planning to achieve mission goals, (2) middle level temporal and resource planning to properly schedule tasks and handle resource competition and (3) low level path and motion planning to plan the actual motion that the robot must follow to fulfill the higher-level plan.

Knowledge and its representation are significantly different in the three classes, and even if various attempts of bridging the gap have been done (Smith, Frank, and Cushing 2008; Bernardini and Smith 2008; Gerevini et al. 2009), existing planning systems can be framed in one of the three classes mentioned above, possibly with some capabilities of modeling and handling simple problems from other classes.

On one side the integration of the different classes of knowledge in a unified model would allow optimization and deep integration, on the other side the different planning processes usually require, to solve not trivial instances of the respective problems, dedicated planners and models. The importance of integrating in a tight loop planning and resource management is widely recognized and has been studied for quite a while (Zweben and Fox 1994; Smith, Frank, and Jonsson 2000). But also the integration of path planning with resource management and task planning is of primary importance: in fact on one side motion and path planning have a strong impact on resource balance, and on the other side the plan's feasibility often depends on platform physical configurations very difficult to be translated into the model of the task planner.

In this paper we propose an integration using modeling and solving features of a constraint-based time-flexible temporal planning framework to smoothly coordinate different planning domains and processes. Starting from a framework developed at the European Space Agency for rapid prototyping of planning and scheduling applications, we introduce modeling primitives to integrate a PDDL and a path planner. The methodology is illustrated taking as example a scenario where a UAV (Unmanned Automated Vehicle) has to be operated to move objects in a warehouse.

This scenario is inspired by the classical Blocks World domain. We took this problem as the basis to devise a more realistic warehouse domain that, while maintaining most of its original features, extends the original problem with:

- Time. We need to take into account the time necessary to process the boxes and to specify temporal windows within which we want a box to be processed;
- Resources. We have a battery on the UAV, that discharges as the UAV moves and that has to be recharged from time to time;
- Navigation. The UAV must take into account the 3D positions of the boxes and any obstacles in the warehouse when flying around;
- Uncertainty at runtime. The plan needs temporal flexibility to handle uncertainty at execution time.

Managing the box configuration is a classical planning problem, conveniently modeled by symbolic objects and actions to manipulate them. At the same time the need to account for the battery consumption and recharging requires quanti-

*This work has been co-funded by the European Space Agency Networking/Partnering Initiative (NPI) between ESA-ESOC and the Center for Telematics (Zentrum für Telematik e.V.), and by the European Research Council (ERC) Advanced Grant "NETSAT" under the Grant Agreement No. 320377.

tative time and non trivial planning capabilities for manipulating resources. Finally to move between the locations the UAV needs to navigate the warehouse, and thus the need for a path planner and the corresponding domain-specific data structure to encode the position of the boxes and obstacles.

In this proposal we acknowledge the strength of existing formalisms and solvers to address different classes of problems in real world applications, and the need of smoothly integrate them, starting from different models and then coordinating different planning processes. After a short review of classical, timeline-based and path planning, we presents our approach.

Background

Modeling is about finding the most appropriate abstraction for a problem. Choosing the right abstraction makes the problem easier to model and solve. Though the same problem can be formulated using different formalisms, some just come more natural and make the domain description more concise and understandable. PDDL for example, is quite convenient to express agent-centric precondition-effect type of reasoning. This makes PDDL very effective in avoiding the combinatorial explosion of statements that would be necessary to enumerate all the possible valid combinations of the world states. When the problem instead focuses more on time and resources, even though PDDL supports some level of temporal and numeric reasoning, other approaches turn out to be more convenient: in this case what has to be represented is a world made of numeric features evolving in time, and constraint-based approaches proved to be more natural for modeling and efficient for solving. The same can be said for path planning or any other domain-specific solving process: to cast them into an inappropriate formalism usually leads to poor models and inefficient problem solving.

Classical Planning

PDDL (Mc Dermott et al. 1998) is the most known language for classical planning based on propositional representations. A planning problem[1] is a tuple $\mathcal{P} = \langle \mathcal{F}, \mathcal{T}, \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, where $\mathcal{F}$ is a finite set of predicate symbols, $\mathcal{T}$ is a set of object types, $\mathcal{A}$ is a set of actions with preconditions, $pre(a)$, add effects, $add(a)$, delete effects, $del(a)$, $\mathcal{I} \subseteq \mathcal{F}$ defines the initial state, and $\mathcal{G} \subseteq \mathcal{F}$ defines the goal state.

A state s is a conjunction of predicates, an action a is executable in a state s if $pre(a) \subseteq s$. The successor state is defined as $\delta(a, s) = s \setminus del(a) \cup add(a)$ for the executable actions. The sequence of actions $\Pi = [a_1, \ldots, a_n]$ is a plan that achieves $\mathcal{G}$ from $\mathcal{I}$ if $\mathcal{G} \subseteq \delta(a_n, \delta(a_{n-1}, \ldots, \delta(a_1, I)))$.

Being the most widely used planning formalism, several planning systems have been proposed and developed over the years. Among the most popular we count the Fast Downward (Helmert 2006) and the Fast Forward (FF) (Hoffman and Nebel 2001) systems. Both can ingest domains defined in PDDL. In this work we use JAVAFF (Pattison 2017), a Java implementation of the FF system, with minor modifications and PDDL2.1 as modeling language.

[1] In this work we constraint the use of PDDL to the STRIPS formalisms and typed objects.

Constraint-based Temporal Planning

In constraint-based temporal planning (Muscettola 1994; Frank and Jonsson 2003), often referred as "timeline-based planning", a planning domain is modeled as a set of *timelines*, i.e. sequences of time intervals tagged with symbolic or numeric values. Timelines in this context are the primary modeling primitive to describe the decomposition of the world into sub-systems that evolve concurrently over time. Planning decisions and exogenous statements affect the behaviors of these sub-systems. The goal of the planning problem is to find a set of such statements to lead the system into a set of behaviors which satisfy some requested properties, such as feasible sequences of states or feasible resource profiles. Moreover, since these sub-systems are part of a whole model, they can be subject to *synchronization* constraints, i.e., causal and temporal relations between the values their behaviors can take and the statements that can affect these behaviors.

A formal definition of timeline-based modeling primitives and planning problems is more difficult than in classical planning (see (Frank and Jonsson 2003; Fratini and Cesta 2012; Frank 2013) for instance), but at a very high level we can define a timeline-based domain $\mathcal{D}$ as a tuple $\langle \mathcal{I}, \mathcal{TL}, \mathcal{S} \rangle$ where $\mathcal{I}$ is a time interval $[t_0, t_H), t_0, t_H \in \mathbb{N}$, $\mathcal{TL}$ is a set of *timelines* and $\mathcal{S}$ is a set of *synchronizations*. A timeline $tl \in \mathcal{TL}$ is defined by a set of assignments $\pi = \langle v, [ts, te) \rangle$ of values v in a set V_{tl} to ordered, non overlapping time intervals $[ts, te)$ in $\mathcal{I}$.

A synchronization $\sigma \in \mathcal{S}$ is a defined by a set of tuples $T_\sigma = \{\langle \nu_i, \tau_i, tl_i \rangle\}$ (where $\nu_i \subseteq V_{tl_i}$ is a value variable, $\tau_i \subseteq \mathcal{I}$ is a temporal variable and tl_i is a timeline), plus a set of *relations* R_σ on them. Given two assignments $\langle \nu, \tau, tl \rangle$ and $\langle \nu', \tau', tl' \rangle$, relations can be temporal (stating that $\tau' = f(\tau)$, where f for instance can state that τ' occurs *before* τ), on the values ($\nu' = f(\nu)$) or mixed ($\nu' = f(\tau, \nu)$).

A synchronization defines a *pattern* of valid assignments of values to timelines in a domain. A synchronization σ is *applicable* to a set of timelines $\mathcal{TL}$ if it exists in σ a tuple $\langle \nu, \tau, tl \rangle$ and in $tl \in \mathcal{TL}$ a pair $\langle v, [ts, te) \rangle$ such that ν can be unified with v and τ can be unified with $[ts, te)$[2]. A synchronization σ applicable to a set of timelines $\mathcal{TL}$ is *satisfied* by $\mathcal{TL}$ if for each tuple $\langle \nu', \tau', tl' \rangle$ in σ exists a pair $\langle v, [ts, te) \rangle$ in $tl' \in \mathcal{TL}$ such that ν' can be unified with v and τ' can be unified with $[ts, te)$.

A planning problem $\mathcal{P}$ is defined by the tuple $\langle \mathcal{D}, \mathcal{TL}_\mathcal{O}, \mathcal{TL}_\mathcal{G} \rangle$, where D is the domain, $\mathcal{TL}_\mathcal{O}$ an initial set of assignments of values to the timelines and $\mathcal{TL}_\mathcal{G}$ a final set of values for the timelines. Task of a timeline based planner is to find a set of assignments $\Pi = \{\pi_1, \ldots, \pi_n\}$ such that, given $\mathcal{TL}_\mathcal{F} = \mathcal{TL}_\mathcal{O} \cup \Pi$, the followings properties are verified: (1) $\mathcal{TL}_\mathcal{G} \subseteq \mathcal{TL}_\mathcal{F}$; (2) $\mathcal{TL}_\mathcal{F}$ satisfies all the applicable synchronizations in $\mathcal{S}$ and (3) $\mathcal{TL}_\mathcal{F}$ is *complete* (i.e., there is a value assigned to the timelines for each instant in $\mathcal{I}$).

An interesting feature of timeline-based planning stems in the fact that it does not make any conceptual difference between numeric or non-numeric variables, continuous or dis-

[2] A variable can be unified with a value in a synchronization σ if it is part of an assignment of values that verifies R_σ.

crete time, controllable or non-controllable features. In this formalism everything is modeled as a timeline. Further, this approach proved to be able to support, in a flexible way, the natural integration of planning and scheduling problems. For these reasons timeline-based planning has been often used in those contexts where the need for modeling and reasoning over quantitative time, state variables and resources is of a primary importance. This is the case, for instance, of many space planning problems from observation scheduling to on-board power, storage and data downlink management (see among others (Jonsson et al. 2000; Cesta et al. 2011; Chien et al. 2012)).

Unlike classical planning, where the PDDL is nowadays a commonly accepted language, in timeline-based planning various languages are currently in use, since historically they have been deployed together with (or on purpose for) different software platforms. Despite the syntactical differences they are all based on the notions of time intervals, values, temporal constraints and primitives of association between values and time intervals. In this work we use the ESA APSI (Advanced Planning and Scheduling Initiative) platform (Fratini and Cesta 2012) for timeline-based planning and its Domain Definition Language DDL. For an overview of other timeline-based planning platforms and languages refer, for example, to (Chien et al. 2012).

Path Planning

Path planning is concerned with the task of finding a feasible geometric path between two points in a 2D or 3D environment. A path is feasible if it respects the robot's kinematic constraints and avoids collisions with the obstacles (Ghallab, Nau, and Traverso 2004). The environment's obstacles and free spaces can be encoded directly as raw data (e.g. point clouds) or, more commonly, resorting to spatial partitioning techniques like quadtrees and octrees. A path planning problem $\mathcal{P}$ is defined by the tuple $\langle q, CS, CS_{free}, q_0, q_g \rangle$. The *configuration* q is an n-tuple that defines the n parameters required to specify the robot's position or orientation. CS is the set of all values that q may take, and is know as the *configuration space*. The *free configuration space* CS_{free} is the subset of CS configurations that are not in collision with any obstacles. q_0 and q_g are the initial and goal configurations. Planning is then about finding a path between q_0 and q_g that lies entirely in CS_{free}.

As our work focus on the interface and not on the path planner itself, we make some simplifying assumptions. The warehouse is a static environment known at the start of the planning problem with no moving object apart from the ones that are moved by the UAV itself. Further, the UAV has no kinematic constraints and can move freely in 3D space in all directions. In our warehouse domain the UAV configuration q is thus described by its position (x, y, z). We use an octree to track the position of the boxes and obstacles, and a solver based on the Rapidly-exploring Random Tree (RRT) (LaValle 1998) algorithm to plan the path.

Related Work

Our approach is at the intersection of three planning paradigms: (a) symbolic classical planning based on action-centered propositional representations; (b) constraint-based time-flexible temporal planning and (c) path planning based on domain-specific representations.

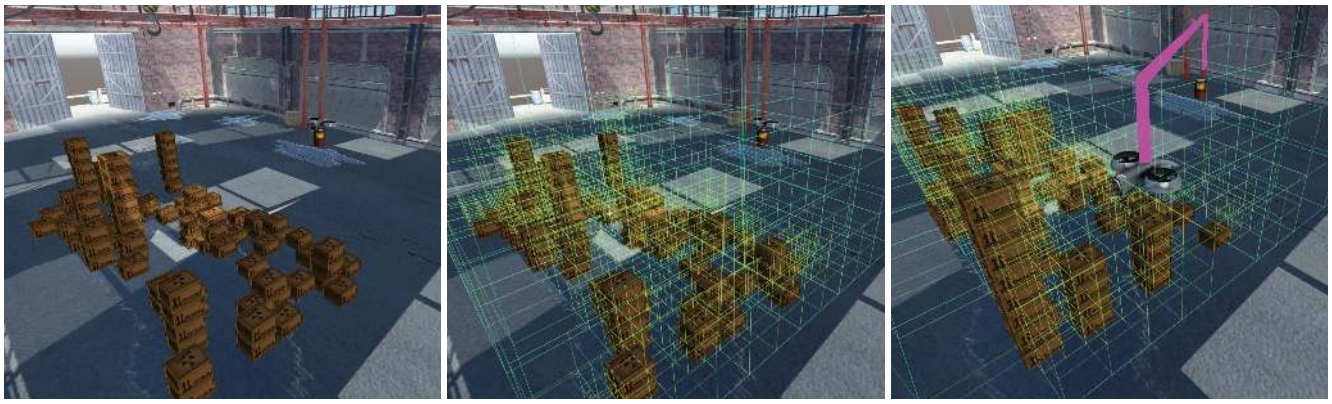
The issue of combining task with motion and object manipulation planning in robotics is well understood and many solutions have been proposed and used - e.g., (Srivastava et al. 2014; Lagriffoul et al. 2012; Erdem et al. 2011; Kaelbling and Lozano-Pérez 2011). Most of the approaches rely on symbolic action-centered STRIPS-like languages for task planning or on HTN planning approaches with no time or resource models.

When coming to more structured, domain independent solutions to integrate different classes of planning problems, we can consider two approaches: extension of a formalism or integration of two formalisms.

When addressing temporal and resource planning for instance, an approach has been to extend the classical propositional STRIPS representation to allow for durative actions, continuous effects, numeric fluents and extended goals in order to solve more realistic planning problems with temporal and resource constraints. The evolution of the PDDL language is of this an example (Gerevini et al. 2009). Notwithstanding there are very effective planners for these extensions, like (Coles et al. 2009) for instance, the original action-centered nature of this type of planning poses a strong bias on the modeling primitives, making difficult to model and reason about various features common in robotics problems: exogenous conditions and uncontrollable events for instance, complex resource models and over-subscribed scheduling, spatial models and component behaviors modeled as timed automata (to mention some). Conversely, timeline-based planning is much more effective in representing models based on timed automata, in resource driven planning and in exogenous or uncontrollable events manipulation (Muscettola 1994; Chien et al. 2012; Fratini et al. 2015), but it suffers significantly when dealing with agent/action centered models (and in spatial modeling).

Regarding the integration, traditionally reactive and layered robotics architectures have dealt with the problem of refining long term, strategic task planning with middle and low level planning and resource management. In this case different planners are integrate at a software level and the architecture provides the way for exchanging information between the different components. Examples are the architecture developed at LAAS (Alami et al. 1998) where a dedicated component in the decision layer is in charge of decomposing plans into an executable sequence based on the PRS language (Ingrand et al. 1996), T-REX (Py, Rajan, and McGann 2010) where different planning and scheduling processes are encapsulated into modules called *reactors* and coordinated by exchanging *goals* and *facts*; or ROAR (Degroote and Lacroix 2011) where the decisional layer is partitioned in separate resources, each one managed by a specific agent. In these architectures the different aspects of the problem are dealt in a loosely coupled way, an approach that presents various advantages but that does not go in the direction of integrating the different models.

Rather than attempting an extension of a particular formalism, this paper focus on the integration of the three ap-



(a) Planning domain. (b) Overlaid octree mesh. (c) UAV path.

Figure 1: Warehouse planning domain and execution environment: (1a) Planning domain showing the storage area with the stacked boxes, the loading dock on the top left, and the UAV on the charging station on the right; (1b) Octree used for path planning overlaid on the domain; (1c) UAV path on the way to pick up a box.

proaches at the modeling level. The most similar work at conceptual level is the Action Notation Modeling Language, ANML (Smith, Frank, and Cushing 2008) that brings actions, hierarchical and temporal planning together. This language is aimed at mixing the most useful features (for applicative domains) of different planning approaches, with the ambitious goal of defining a language able to represent all the different aspects of domains and problems. In fact, ANML derives features from PDDL and NDDL (a language for constraint-based temporal planning based on multi-valued variable representation) to represent actions, conditions and effects, rich temporal constraints, activities, resource usages and HTN decompositions. Our aim is similar, but instead of designing a new language, we combine different existing formalisms while retaining the original modeling primitives and, for what classical and constraint-based planning is concerned, their languages.

Last but not least, (Bernardini and Smith 2008) proposes a methodology to translate PDDL2.2 into NDDL. This is also conceptually similar to what is being proposed here to integrate PDDL into DDL, but instead of automatically extracting the timelines from the PDDL model, a methodology that preserves the theoretical soundness of the translation but does not bridge the semantic gap between the timeline and the action based formalism, we translate directly only the actions, while the PDDL state is semantically translated into timelines by means of synchronizations to map it into timeline values.

The Warehouse Domain

The domain, depicted in Figure 1, is composed of one warehouse of finite dimensions containing: (a) one storage area with a number n of boxes (of the same shape and size); (b) one loading dock; (c) one unmanned automated vehicle with a rechargeable battery and an arm that can carry one box at a time; (d) one charging station.

The warehouse storage area has pre-defined finite dimensions, constraining the maximum number of boxes that can be placed on the floor and the maximum number of boxes that can be stacked. A UAV is used to move the boxes from the storage area to the loading dock, for posterior loading for distribution. The UAV has a battery of limited capacity and an arm with a grip to pick up the boxes. The battery discharges as the UAV moves and picks up boxes. The warehouse has a charging station used by the UAV to recharge its battery as required. Finally, the warehouse has in store a number n of boxes up to a maximum number limited by the warehouse dimensions disposed in an initial configuration [3].

The task at hand involves moving a given set of boxes from the storage area to the loading dock for transport within a given time window. As they arrive in the warehouse the boxes are first placed in the storage area. The boxes must then be moved to the loading dock and arranged in a specific configuration such as to facilitate their posterior loading and delivery. The initial number of boxes, their starting and final positions as well as the time window allotted to move the boxes are given by the initial problem. The UAV initial position and battery state-of-charge are also set at the start. In this domain we can clearly identify:

- The higher level task planning to manipulate the boxes. For this problem we use a classical PDDL formulation of a blocks world domain and JAVAFF;
- The temporal and resource management problem to plan the battery recharging activities. This problem is modeled in DDL, the modeling language of the APSI platform, and we use PLASMA, a timeline based planner described in (De Maio et al. 2015);
- The path planning. For this we use an octree spacial representation and a path planner described in (LaValle 1998).

[3]The Warehouse domain and its Unity-based execution environment were first introduced in (Nogueira, Fratini, and Schilling 2017), where we describe an application for integrated evaluation of planning and execution.

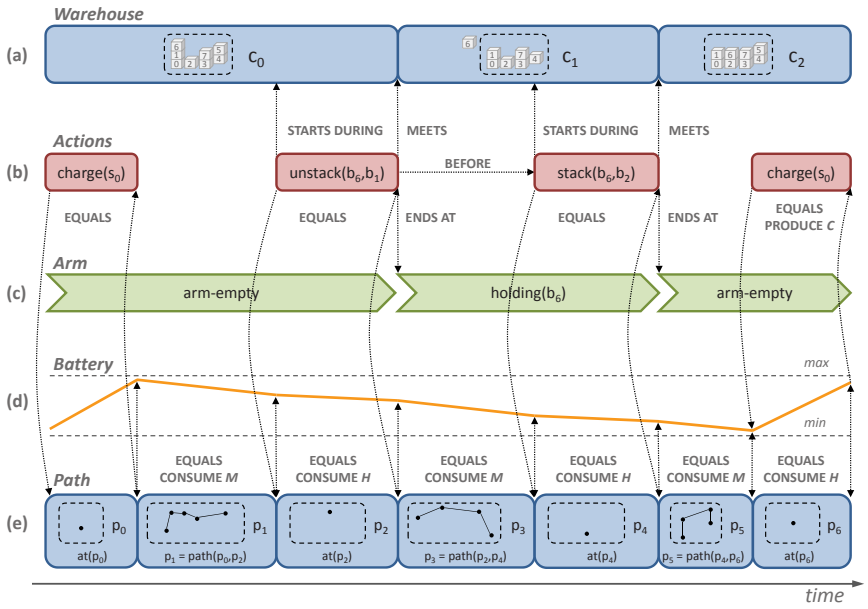


Figure 2: An example of a timeline-based plan for the warehouse domain showing some of the temporal and causal relations between timelines. From top to bottom: (a) position of the boxes and obstacles in the warehouse; (b) the UAV actions to move the boxes; (c) the UAV arm status; (d) the UAV rechargeable battery; (e) the UAV path. Note that for the sake of making the figure more understandable not all synchronizations are shown.

Modeling

At modeling level the problem is then how to smoothly integrate the three models above. We have chosen to start from a timeline-based model expressed in the APSI platform modeling language DDL as this formalism is substantially agnostic with respect to what is represented in the timelines and does not pose a strict distinction between states and actions[4].

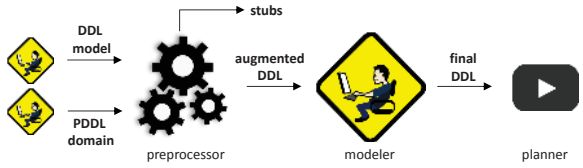


Figure 3: Modeling process overview.

We extended the syntax of DDL to allow the specification of snippets of PDDL code[5] directly into the DDL model. From these code snippets an updated set of DDL directives are automatically derived (by a preprocessor) in the form of a set of additional values for the timelines and synchronizations among them (see Figure 3). This augmented model can represent in DDL the PDDL states, actions and effects. From that the model integration is finalized with DDL synchronizations that link temporally and logically the part of the model derived from PDDL, the trajectory produced by the path planner and the rest of the native timeline based model. The result of the compilation is a DDL model to be injected into a timeline-based planner that makes calls to a PDDL and an RRT planner to solve the integrated problem.

The preprocessing step is a generic procedure that starts from a domain partially described in DDL and partially described in PDDL and produces an augmented DDL model. The preprocessing step automatically maps predicate symbols and actions to DDL timelines, and translates the effects of PDDL actions into DDL synchronizations (see example below). In addition the preprocessor guarantees that every PDDL object type has a matching enumerated, object or numeric data type in DDL. Once the full domain has been processed: (a) every PDDL predicate is mapped to one, and only one, APSI timeline; (b) every PDDL action is mapped to at least one APSI timeline. This augmented model can be further tailored by the modeler.

Timelines

In the base DDL model we have 5 timelines (see Figure 2):

- a timeline WAREHOUSE encapsulating an octree representation of the spatial configuration of the warehouse with the position of the boxes and any obstacles;
- a timeline ACTIONS to represent the actions being performed by the UAV. This timeline can take the value

[4]We take here the APSI platform modeling language as a reference but, since we are modeling with standard primitives available in any language for timeline based planning, other systems could be used as well.

[5]In the current implementation we support PDDL2.1. However, and given that time and resources are modeled directly in DDL, we limit the use of PDDL2.1 at the moment to STRIPS and typing. We directly insert PDDL code snippets within the DDL model, but it would be equally valid to keep the PDDL model separate in a different file and merge both models at compile time.

CHARGE($?s$), when the UAV is at the charging station $?s$, and a value for each action of the embedded PDDL model. In our case, we have the actions UNSTACK($?box, ?object$) and STACK($?box, ?object$), being $?box$ and $?object$ two generic DDL objects representing boxes IDs (the first) and either a box ID or a static object the second;

- a timeline ARM to represent the status of the UAV arm. The timeline can take the values ARM-EMPTY, when the arm is not holding any box, and HOLDING($?box$), $?box$ is the ID of the box currently being held by the arm;
- a timeline BATTERY to model the UAV battery level as a reservoir resource (as in (Fratini et al. 2015));
- a timeline PATH to model the actual position of the UAV and the path being followed.

The timeline WAREHOUSE models the position of the objects in a 3D space. Each object can occupy a position $\langle x, y, z\rangle$ in the space, can be added in a position $\langle x, y, z\rangle$ and can be removed from the space. Since we need to model also a box being held by the UAV arm, we conventionally use a position $\langle \overline{x}, \overline{y}, \overline{z}\rangle$ for this purpose. The values of this timeline (internally encoded as an octree) are stated but by means of two functions: $insert(?o, ?x, ?y, ?z)$ and $remove(?x, ?y, ?z)$.

Beside that, 4 more functions are defined to query the spacial position of an object given the octree representation: $getX(?o)$, $getY(?o)$ and $getZ(?o)$ to retrieve the position of an object in the space (if $?o$ is in the warehouse, otherwise the function returns -1) and $get(?x, ?y, ?z)$, to retrieve the the object in $\langle x, y, z\rangle$ (if any, otherwise the function returns -1).

The timeline PATH models the position of the UAV in the 3D space as it moves through the warehouse. A value in this timeline is either a fixed position (p_0, p_2, p_4, p_6) meaning that the UAV is *at* a given position, or a path between two fixed positions (p_1, p_3, p_5). In the current implementation the path is encoded as an ordered list of waypoints. Other timelines synchronize to this timeline by posting desired positions.

The ARM timeline, although redundant here, has been introduced in the model because it is needed to control the UAV arm at execution time (and it makes the model more understandable).

Embedded PDDL Domain

In our domain we use PDDL to model the manipulation of the objects in the warehouse as STACK and UNSTACK actions. We model two types of objects: *box* to model objects that can be moved by the STACK and UNSTACK actions; and *static* to model the floor and any other object in the warehouse that cannot be moved. The action STACK is used to place a box on the top of another object, and the action UNSTACK to remove a box from the top of another object. In both cases the object can be another box or a static object.

The following code is then embedded in the DDL model by means of the added primitive EXTERNAL_DOMAIN:

```
EXTERNAL_DOMAIN pddl21 {

  (define (domain boxes)

    (:requirements :strips :typing)

    (:types box static)

    (:predicates (clear ?x) (holding ?x) (on ?x ?y) (
        armempty) (onfloor ?x))

    (:action stack
    :parameters (?ob - box ?uo)
    :precondition (and (clear ?uo) (holding ?ob))
    :effect (and (clear ?ob) (on ?ob ?uo)
                 (not (clear ?uo))
                 (not (holding ?ob))
                 (armempty)))

    (:action unstack
    :parameters (?ob - box ?uo)
    :precondition (and (on ?ob ?uo) (clear ?ob) (armempty) )
    :effect (and (holding ?ob) (clear ?uo)
                 (not (on ?ob ?uo))
                 (not (clear ?ob))
                 (not (armempty)))))

}
```

Synchronizations

In order to integrate the results of the PDDL planner into the timelines, the actions in the PDDL model are represented directly as states of a timeline. Also the state of the PDDL planner must be represented somewhere, in order to (1) synchronize it with the rest of the model and (2) generate problems for the PDDL planner. This is done in DDL by means of the expand synchronization directive EXPAND_SYNC. This would be for example the synchronization for the STACK($?box, ?object$) action of the PDDL domain above:

```
SYNCHRONIZE actions.tl
{
  VALUE stack(?ob, ?uo)
  {
    EXPAND_SYNC boxes {warehouse.tl};
  }
}
```

This directive is embedded into a DDL synchronization and compiled by the preprocessor in Figure 3 with the following effects: (1) the set of possible values for timelines chosen to represent the PDDL actions are extended adding values to represent them;[6] (2) the set of functions to state and query values of timelines chosen to represent the PDDL state are extended adding methods to assert the effects of the PDDL actions and to query the PDDL actions' preconditions (stubs in Figure 3);[7] (3) the DDL synchronization is extended to represent the preconditions and effects of the action over the selected timelines.

In the example above we synchronize the ACTIONS timeline with the WAREHOUSE timeline, that represents the PDDL state in a shape suitable for being synchronized with the rest of the model. The synchronization is derived by

[6] Types in the PDDL domain are bound to DDL enumerated types of the same name. Objects are then defined directly in the DDL model instead of in the PDDL problem.

[7] A predicate `p(?o :type)` in the PDDL model is translated into a function `void p(?status :int, ?o :Object,?b :boolean)` and a function `boolean is_p(?o :Object)`. The function call `p(?status,?o,true)` asserts `p(?o)` in `?status`, the call to `p(?status,?o,false)` asserts $\neg$ `p(?o)`.

PDDL Predicates	DDL Timeline (+ statement)	DDL Timeline (query)
CLEAR(?o)	remove(getX(?o), getY(?o), getZ(?o)+1)	get(getX(?o), getY(?o), getZ(?o)+1) == -1
ON(?o1, ?o2)	remove(getX(?o1), getY(?o1), getZ(?o1)) ∧ insert(?o1, getX(?o2), getY(?o2), getZ(?o2)+1)	getX(?o1) == getX(?o2) ∧ getY(?o1) == getY(?o2) ∧ getZ(?o1) == (getZ(?o2) + 1)
HOLDING(?o)	insert(?o,$\overline{x}$, $\overline{y}$, $\overline{z}$)	getX(?o) == $\overline{x}$ ∧ getY(?o) == $\overline{y}$ ∧ getZ(?o) == $\overline{z}$
ARMEMPTY()	–	get($\overline{x}$, $\overline{y}$, $\overline{z}$) == -1

Table 1: Map between PDDL predicates and DDL timeline statements for the *boxes* domain.

the action schema of the the PDDL model, using the query functions of the octree timeline for the preconditions and the statement functions for the effects. In this case the compilation of this code makes that: (1) the values UNSTACK($?box, ?object$) and STACK($?box, ?object$) are added to the timeline ACTIONS. In general a value for each action defined in the PDDL model is added to the DDL model; (2) eight function stubs are added to the timeline WAREHOUSE, one for each predicate defined in the model, to translate the timeline value in a PDDL state and back. These stubs have to be implemented (in Java) to translate them into the primitives available for the target timeline. Table 1 for instance shows the implementation of the stubs in terms of the primitives available on the WAREHOUSE timeline. Not all of the statements are implemented: in fact the octree representation keeps track of the exact position of any box, and this configuration is updated only by the ON($?o1, ?o2$) and HOLDING($?o$) effects. Negative statements are not needed here for the same reason, but in the general case all the positive and negative effects, as well as all the queries should be implemented.

After running the preprocessor, the initial EXPAND_SYNC directive is translated in a set of DDL instructions. The user can then add other synchronizations as required by the domain. This is, for example, the synchronization generated for the STACK($?box, ?object$) after running the preprocessor and adding extra DDL statements to complete the model:

```
SYNCHRONIZE actions.tl
{
  VALUE stack(?ob, ?uo)
  {
    // EXPAND_SYNC: autogenerated code - START
    PRE <?> warehouse.tl.Value(?v_pre);
    EFF warehouse.tl.Value(?v_eff);
    STARTS_DURING PRE;
    MEETS EFF;

    [is_clear(?v_pre,?uo)];
    [is_holding(?v_pre,?ob)];

    ?v_eff := clear(?v_pre, ?ob, true);
    ?v_eff := on(?v_eff, ?ob, ?uo, true);
    ?v_eff := clear(?v_eff, ?uo, false);
    ?v_eff := holding(?v_eff, ?ob, false);
    ?v_eff := armempty(?v_eff, true);
    // EXPAND_SYNC: autogenerated code - END

    POS path.tl.At(?x,?y,?z);
    EQUALS POS;
    ?x := getX(?v_pre,?uo);
    ?y := getY(?v_pre,?uo);
    ?z := getZ(?v_pre,?uo);

    ARM arm.tl.ArmEmpty();
    ENDS_AT ARM;
  }
}
```

In this synchronization it is stated that a value STACK($?ob, ?uo$) on the timeline ACTIONS.TL must start during a value VALUE($?v\_pre$) of the timeline WAREHOUSE.TL and, when it finishes, a value VALUE($?v\_eff$) must start on the same timeline (see Figure 2). The temporal constraints translate the temporal semantics of precondition and effect in STRIPS, the functions on v_pre state conditions of applicability of the synchronization[8] and the constraints on v_eff define the successive configuration of the warehouse[9].

To close the loop, and as shown in Figure 2, we need to synchronize the path with the battery consumption. This is, for example, the synchronization when the UAV is holding at a given position:

```
SYNCHRONIZE path.tl
{
  VALUE At(?x,?y,?z)
  {
    CONS battery.SET_SLOPE(H);
    EQUALS CONS;
  }
}
```

Problem Solving

The DDL model described above is given as input to PLASMA, a timeline planner and scheduler. PLASMA (De Maio et al. 2015) is a planner designed as a collection of *solvers* that implements a *flaw-based* solving process (solver PLASMA). Solvers are chosen and activated on a flaw detection base. When a flaw is detected on a timeline the planner activates the corresponding solver to fix the problem.

Solver PLASMA

Input: planning problem $\mathcal{P} = \langle Problem \rangle$
procedure SOLVE($\mathcal{P}$)
  $\mathcal{TL} = Init(Problem)$
  $\Phi = CollectFlaws(\mathcal{TL})$
  while $\Phi \neq \emptyset$ **do**
    $\phi \leftarrow ChooseFlaw(\Phi)$
    $\sigma \leftarrow ChooseSolver(\phi)$
    $\mathcal{TL} \leftarrow \sigma.ChooseUpdate(\phi, \mathcal{TL})$
    $\Phi = CollectFlaws(\mathcal{TL})$
  end while
  return $\mathcal{TL}$
end procedure

[8] In DDL the <?> states that the value can't be generated on the timeline when the synchronization is applied, but must be present. The statements in between brackets [] states a guard to apply the synchronization. As a result, the action must start during an existing configuration identified by v_pre such that the two conditions hold.

[9] Some of the methods applied in the synchronization to calculate v_eff have no effects. As previously mentioned not all of them are implemented, but being this an automatic translation, all the action effects are syntactically applied.

A flaw is any type of violation in a plan. It can be a logical flaw, when unsupported actions are added to the plan, or a resource violation flaw (when a resource is over or under used) or any other impairment of temporal allocation on the values over a timeline. PLASMA uses various solvers, each one dedicated to the solution of a particular flaw, and can be configured by extending the flaws he can manage, the resolvers associated with each flaw and the priority with which each type of flaw must be resolved.

The set of solvers that PLASMA uses is extended for this domain by adding two new types: one embedding the JAVAFF (Pattison 2017) PDDL planner and one embedding an RRT path planner (LaValle 1998). The set of flaws is extended as well: a gap between two states c and c' on timeline entails the instantiation of a PDDL planner (solver PDDL) and the generation of a sequence of actions to change the disposition of the boxes from c to c'.

To extract the initial and goal states for a PDDL problem (function $ComputeStates(\phi, \mathcal{TL})$), the PDDL solver uses the queries in Table 1 to interrogate the WAREHOUSE timeline over all applicable domain objects. With reference to Figure 2, the flaw ϕ provides c_0 and c_2, the initial status s_0 is computed by applying the queries to c_0, and the final status s_F by applying them to c_2. This process allows the solver to derive the conjunction of positive propositions that make the initial and goal states. To be noted that we do not use the ARM timeline when building the PDDL states. In this domain all the knowledge needed to derive the states is encoded in the WAREHOUSE timeline.

At the beginning of a PDDL solving step, and when building the initial and goal states, the solver will query the DDL data types that match PDDL types and update its internal list of objects accordingly. By doing this at each solving step we can accommodate dynamically problems where objects can be created and destroyed during planning.

Solver PDDL choose update

1: **Input:** the flaw ϕ and the domain timeline $\mathcal{TL}$
2: **procedure** PDDL-CHOOSE-UPDATE(ϕ,$\mathcal{TL}$)
3: $\{s_0, s_F\} \leftarrow ComputeStates(\phi, \mathcal{TL})$
4: $\mathcal{P} \leftarrow GroundProblem(s_0, s_F)$
5: $\mathcal{A} \leftarrow FF(\mathcal{P})$
6: **return** $Update(\mathcal{TL}, \mathcal{A})$
7: **end procedure**

The actions generated by the PDDL planner are then added back to the UAV ACTIONS timeline (function $Update(\mathcal{TL}, \mathcal{A})$) as a set of values attached to intervals of duration [1,∞), with a simple precedence temporal constraint among them (the PDDL planner produces a pure ordered sequence of actions with no temporal duration). Again with reference to Figure 2 for instance, the values UNSTACK(b_6, b_1) BEFORE STACK(b_6, b_2). These values, added to the ACTIONS timeline (b), are then synchronized by the PLASMA planner to generate new values for timeline (a), (c) and (e).

An unjustified value added on the timeline UAV PATH entails then the generation of a flaw that instantiates an RRT-based path planner that computes the path between the two points such as to resolve the flaw. PLASMA synchronises these values with resource requests, and when resource violations arises, generates values on the ACTIONS timeline to fly the UAV to a recharging station (values CHARGE($?station$)).

Solver RRT choose update

1: **Input:** the flaw ϕ and the domain timeline $\mathcal{TL}$
2: **procedure** RRT-CHOOSE-UPDATE(ϕ,$\mathcal{TL}$)
3: $\{p_0, p_F\} \leftarrow ComputePositions(\phi, \mathcal{TL})$
4: $CS \leftarrow ConfigurationSpace(\phi, \mathcal{TL})$
5: $\mathcal{P} = \langle p_0, p_F, CS \rangle$
6: $\mathcal{A} \leftarrow RRT(\mathcal{P})$
7: **return** $Update(\mathcal{TL}, \mathcal{A})$
8: **end procedure**

It is worth pointing out that this introduces a loop centred on the resources. A resource over consumption can be fixed by adding actions to the plan that, in turn, also consume resources. Moreover, to optimise solutions in this domain we need again to reason around resource availability, and in fact the task sequence has to be optimised in order to minimise the need for recharging the UAV battery. This is the main reason for the need of a planner like PLASMA with powerful temporal and resource management capabilities to drive the instantiation of the task and path planners.

Conclusions and Future Work

We presented an approach to combine task, resource and path planning for robotics. This approach uses existing paradigms and formalisms of classical, timeline-based and path planning, that we integrate to solve in an unified way different classes of problems. Rather than handling each of the problems separately and then combining the results, the proposed implementation allows us to close the loop at modeling and solving level between task, resource and path planning. In the warehouse domain, for instance, the resource usage depends on the path traveled and the tasks performed and, at the same time, the path traveled and the tasks performed depend on the resource usage, as we have to plan for the recharge of the battery. While task, path and resource planning could probably be addressed sequentially in different scenarios, in this particular domain, and given the circular dependencies between the three problems, the proposed integration is needed to efficiently solve the problem and to optimize resource allocation.

The rationale behind this approach is to integrate as much as possible existing planners to handle real world problems, with the main objective of maintaining the knowledge encoded in the various domain as clear as possible. For this reason, we propose a two-step compilation process: first the modeler encodes the know-how using the most appropriate formalisms, then these inputs are integrated into a unified model that retains the original modeling primitives of both languages (actions, timelines and octrees). This gives the advantages of an integrated model without giving up the resolution power of the different systems. How far a given

PDDL model needs to be tailored for integration depends on how tight is the interaction between the part of the domain described in PDDL and the rest of the domain described in DDL. In this particular warehouse scenario that we are using as a case study, given that an unstack-stack sequence for a given box cannot be interrupted to, for example, insert a battery recharge operation, we can use the typical blocks world PDDL domain without any sort of tailoring. But in general the problem exists and it is currently subject to study.

The approach is currently being extended to include temporal PDDL and to consider a multi-UAV scenario, that introduces the need for global optimization of time, resources and path planning.

References

Alami, R.; Chatila, R.; Fleury, S.; Ghallab, M.; and Ingrand, F. 1998. An architecture for autonomy. *International Journal of Robotics Research* 17:315–337.

Bernardini, S., and Smith, D. 2008. Translating pddl2.2 into a constraint-based variable/value language. In *Proc. of the Workshop on Knowledge Engineering for Planning and Scheduling (KEPS), ICAPS-2008*.

Cesta, A.; Cortellessa, G.; Fratini, S.; and Oddi, A. 2011. MR-SPOCK: Steps in Developing an End-to-End Space Application. *Computational Intelligence* 27(1).

Chien, S.; Johnston, M.; Frank, J.; Giuliano, M.; Kavelaars, A.; Lenzen, C.; and Policella, N. 2012. A generalized timeline representation, services, and interface for automating space mission operations. In *Proceedings of the 12th International Conference on Space Operations, SpaceOps*. AIAA.

Coles, A.; Fox, M.; Halsey, K.; Long, D.; and Smith, A. 2009. Managing concurrency in temporal planning using planner-scheduler interaction. *Artif. Intell.* 173(1):1–44.

De Maio, A.; Fratini, S.; Policella, N.; and Donati, S. 2015. Resource driven planning with "PLASMA": the plan space multi-solver application. In *ASTRA 2015. 13^{th} Symposium on Advanced Space Technologies in Robotics and Automation*.

Degroote, A., and Lacroix, S. 2011. ROAR: resource oriented agent architecture for the autonomy of robots. In *IEEE International Conference on Robotics and Automation, ICRA 2011, Shanghai, China, 9-13 May 2011*, 6090–6095.

Erdem, E.; Haspalamutgil, K.; Palaz, C.; Patoglu, V.; and Uras, T. 2011. Combining High-Level Causal Reasoning with Low-Level Geometric Reasoning and Motion Planning for Robotic Manipulation. In *IEEE International Conference on Robotics and Automation (ICRA)*.

Frank, J., and Jonsson, A. 2003. Constraint Based Attribute and Interval Planning. *Journal of Constraints* 8(4):339–364.

Frank, J. 2013. What is a Timeline? In *Proceedings of the 4th Workshop on Knowledge Engineering for Planning and Scheduling at ICAPS-13 (KEPS-13), Rome, Italy*.

Fratini, S., and Cesta, A. 2012. The APSI Framework: A Platform for Timeline Synthesis. In *Proceedings of the 1st Workshops on Planning and Scheduling with Timelines at ICAPS-12 (PSTL-12), Atibaia, Brazil*.

Fratini, S.; Policella, N.; Faerber, N.; De Maio, A.; Donati, A.; and Sousa, B. 2015. Resource Driven Timeline-Based Planning for Space Applications. In *Proceedings of the 9^{th} International Workshop on Planning and Scheduling for Space, IWPSS15*.

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. *Artif. Intell.* 173(5-6):619–668.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory & Practice*. Elsevier.

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

Hoffman, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14(27):253–302.

Ingrand, F. F.; Chatila, R.; Alami, R.; and Robert, F. 1996. Prs: A high level supervision and control language for autonomous mobile robots. In *Proc. of the IEEE Int. Conf. on Robotics and Automation*, 43–49.

Jonsson, A.; Morris, P.; Muscettola, N.; Rajan, K.; and Smith, B. 2000. Planning in Interplanetary Space: Theory and Practice. In *AIPS-00. Proc. of the Fifth Int. Conf. on Artificial Intelligence Planning and Scheduling*, 177–186.

Kaelbling, L. P., and Lozano-Pérez, T. 2011. Hierarchical task and motion planning in the now. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, 1470–1477. IEEE.

Lagriffoul, F.; Saffiotti, A.; Karlsson, L.; Lagriffoul, F.; Dimitrov, D.; Saffiotti, A.; and Karlsson, L. 2012. Constraint propagation on interval bounds for dealing with geometric backtracking. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, number October.

LaValle, S. M. 1998. Rapidly-Exploring Random Trees: A New Tool for Path Planning. Technical report, Computer Science Dept., Iowa State University.

Mc Dermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL - the planning domain definition language. Tehnical report, CVC TR-98-003 / DCS TR-1165, Yale Center for Communicational Vision and Control.

Muscettola, N. 1994. HSTS: Integrating Planning and Scheduling. In Zweben, M. and Fox, M.S., ed., *Intelligent Scheduling*. Morgan Kauffmann.

Nogueira, T.; Fratini, S.; and Schilling, K. 2017. Autonomously Controlling Flexible Timelines: From Domain-independent Planning to Robust Execution. In *IEEE Aerospace Conference*.

Pattison, D. 2017. JavaFF. JavaFF distribution site: http://personal.strath.ac.uk/david.pattison/#software.

Py, F.; Rajan, K.; and McGann, C. 2010. A systematic agent framework for situated autonomous systems. In *AAMAS*, 583–590.

Smith, D. E.; Frank, J.; and Cushing, W. 2008. The ANML Language. In *Proceedings of ICAPS-08 (2008)*.

Smith, D.; Frank, J.; and Jonsson, A. 2000. Bridging the gap between planning and scheduling. *Knowledge Engineering Review* 15(1):47–83.

Srivastava, S.; Fang, E.; Riano, L.; Chitnis, R.; Russell, S.; and Abbeel, P. 2014. Combined task and motion planning through an extensible planner-independent interface layer. *Proceedings - IEEE International Conference on Robotics and Automation* (M):639–646.

Zweben, M., and Fox, M. 1994. *Intelligent Scheduling*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Classical Planning in Deep Latent Space: From Unlabeled Images to PDDL (and back)

Masataro Asai, Alex Fukunaga
Graduate School of Arts and Sciences
University of Tokyo

Abstract

Current domain-independent, classical planners require symbolic models of the problem domain and instance as input, resulting in a knowledge acquisition bottleneck. Meanwhile, although recent work in deep learning has achieved impressive results in many fields, the knowledge is encoded in a subsymbolic representation which cannot be directly used by symbolic systems such as planners. We propose LatPlan, an integrated architecture combining deep learning and a classical planner. Given a set of unlabeled training image pairs showing allowed actions in the problem domain, and a pair of images representing the start and goal states, LatPlan uses a Variational Autoencoder to generate a discrete latent vector from the images, based on which a PDDL model can be constructed and then solved by an off-the-shelf planner. We evaluate LatPlan using image-based versions of 3 planning domains: 8-puzzle, LightsOut, and Towers of Hanoi.

1 Introduction

Recent advances in domain-independent planning have greatly enhanced their capabilities. However, planning problems need to be provided to the planner in a structured, symbolic representation such as PDDL (McDermott 2000), and in general, such symbolic models need to be provided by a human, either directly in a modeling language such as PDDL, or via a compiler which transforms some other symbolic problem representation into PDDL. This results in the *knowledge-acquisition bottleneck*, where the modeling step is sometimes the bottleneck in the problem solving cycle. In addition, the requirement for symbolic input poses a significant obstacle to applying planning in *new, unforeseen* situations where no human is available to create such a model or a generator, e.g., autonomous spacecraft exploration. In particular this first requires generating symbols from raw sensor input, i.e., the *symbol grounding problem* (Steels 2008).

Recently, significant advances have been made in neural network (NN) deep learning approaches for perceptually-based cognitive tasks including image classification (Deng et al. 2009), object recognition (Ren et al. 2015), speech recognition (Deng, Hinton, and Kingsbury 2013), machine translation as well as NN-based problem-solving systems for problem solving (Mnih et al. 2015; Graves et al. 2016). However, the current state-of-the-art in pure NN-based systems do not yet provide guarantees provided by symbolic planning systems, such as deterministic completeness and solution optimality.

Figure 1: An image-based 8-puzzle.

Using a NN-based perceptual system to *automatically* provide input models for domain-independent planners could greatly expand the applicability of planning technology and offer the benefits of both paradigms. *We consider the problem of robustly, automatically bridging the gap between such subsymbolic representations and the symbolic representations required by domain-independent planners*.

Fig. 1 (left) shows a scrambled, 3x3 tiled version of the the photograph on the right, i.e., an image-based instance of the 8-puzzle. Even for humans, this photograph-based task is arguably more difficult to solve than the standard 8-puzzle because of the distracting visual aspects. We seek a domain-independent system which, given only a set of unlabeled images showing the valid moves for this image-based puzzle, finds an optimal solution to the puzzle. Although the 8-puzzle is trivial for symbolic planners, solving this image-based problem with a domain-independent system which (1) *has no prior assumptions/knowledge* (e.g., "sliding objects", "tile arrangement"), and (2) *must acquire all knowledge from the images*, is nontrivial. Such a system should not make assumptions about the image (e.g., "a grid-like structure"). The only assumption allowed about the nature of the task is that it can be modeled and solved as a classical planning problem.

We propose Latent-space Planner (LatPlan), an integrated architecture which uses NN-based image processing to completely automatically generate a propositional, symbolic problem representation which can be used as the input for a classical planner. LatPlan consists of 3 components: (1) a NN-based *State Autoencoder* (SAE), which provides a bidirectional mapping between the raw input of the world states and its symbolic/categorical representation, (2) an *action model generator* which generates a PDDL model of the problem domain using the symbolic representation acquired

by the SAE, and (3) a symbolic planner. Given only a set of *unlabeled images* from the domain as input, we train (unsupervised) the SAE and use it to generate D, a PDDL representation of the image-based domain. Then, given a planning problem instance as a pair of initial and goal images such as Fig. 1, LatPlan uses the SAE to map the problem to a symbolic planning instance in D, and uses the planner to solve the problem. We evaluate LatPlan using image-based versions of the 8-puzzle, LightsOut, and Towers of Hanoi domains.

2 Background

Feed Forward Neural Networks (FFN) are nonlinear function approximators consisting of layers of nodes with real-valued activations, and the nodes are connected to multiple nodes in the next layer by weighted edges. The output of each node is the weighted sum of the activations of the input nodes, transformed by a nonlinear activation function. Recent advances in deep learning have greatly increased the utility of FFNs as a powerful knowledge representation mechanism (Goodfellow, Bengio, and Courville 2016).

Although the traditional implementation of neural networks have multiple problems that make the deeply layered networks impractical, they are largely alleviated by the modern techniques: Convolutional Network is capable of learning translation-invariant representation in image-based tasks compared to the fully-connected networks; The learning speed was greatly improved by Stochastic Gradient Descent combined with batch processing and GPU acceleration; The problem of vanishing gradient in deep networks was alleviated by Rectified Linear Unit (ReLU); Dropout (Srivastava et al. 2014) and Regularization reduces the overfitting caused by the excessive representational power of the network; Finally, Batch Normalization (Ioffe and Szegedy 2015) and recent optimization algorithms such as Adam (Kingma and Ba 2014) further improves the learning speed.

An AutoEncoder (AE) is a type of FFN that uses unsupervised learning to produce an image that matches the input (Hinton and Salakhutdinov 2006). The intermediate layer has a *Latent Representation* of the input and is performing data compression. AEs are commonly used for pretraining a neural network. The performance of an AE is measured by the reconstruction loss, the distance between the input and the output vectors under a distance function such as l_1 norm, l_2 norm (euclidean distance) or binary crossentropy.

A Variational AutoEncoder (VAE) (Kingma and Welling 2013) is a type of AE that forces the *latent layer* (the most compressed layer in the AE) to follow a certain distribution (e.g., Gaussian) for given input images. Since the target random distribution prevents backpropagating the gradient, most VAE implementations use *reparametrization tricks*, which decompose the target distribution into a differentiable distribution and a purely random distribution that does not require the gradient. For example, the Gaussian distribution $N(\sigma, \mu)$ can be decomposed into $\mu + \sigma N(1, 0)$. In addition to the reconstruction loss, VAE is also tasked to minimize the variational loss, i.e. the difference between the learned distribution and the target distribution.

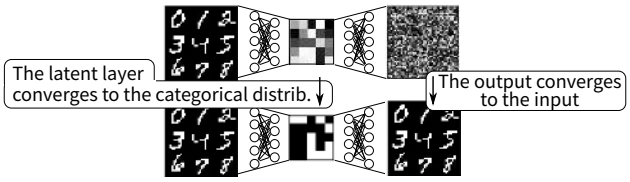


Figure 2: Step 1: Train the State Autoencoder by minimizing the sum of the reconstruction loss (binary cross-entropy between the input and the output) and the variational loss of Gumbel-Softmax (KL divergence between the actual latent distribution and the random categorical distribution as the target). As the training continues, the output of the network converges to the input images. Also, as the Gumbel-Softmax temperature τ decreases during training, the latent values approaches the discrete categorical values of 0 and 1.

Gumbel-Softmax (GS) activation is a recently proposed reparametrization trick (Jang, Gu, and Poole 2017), which continuously approximates Gumbel-Max (Maddison, Tarlow, and Minka 2014), a method for drawing samples from categorical distribution. Assume the output $z = \langle z_i \rangle$ is a k-dimensional one-hot vector, e.g. $\langle 0, 1, 0 \rangle$ represents "b" of a domain $D = \{a, b, c\}$. The input is class probabilities $\pi = \langle \pi_i \rangle$, e.g. $\langle 0.1, 0.1, 0.8 \rangle$. Gumbel-Max draws samples from D following the probabilities π as follows:

$$z_i = [i == \arg\max_j (g_j + \log \pi_j)?1 : 0]$$

where $g_1 \ldots g_k$ are i.i.d samples drawn from Gumbel$(0, 1)$ (Gumbel and Lieblein 1954). Gumbel-Softmax approximates argmax with softmax to make it differentiable:

$$z_i = \text{Softmax}((g_i + \log \pi_i)/\tau)$$

"temperature" τ controls the magnitude of approximation. τ is annealed by a schedule $\tau \leftarrow \max(0.1, \exp(-rt))$ where t is the current training epoch and r is an annealing ratio. We chose r so that $\tau = 0.1$ when the training finishes. The above schedule follows the original by Jang, Gu, and Poole (2017). Using GS in the network in place of standard activation functions (Sigmoid, Softmax, ReLU) forces the activation to converge to a discrete one-hot vector when $\tau \approx 0$.

3 LatPlan: System Architecture

This section describes the LatPlan architecture and the current implementation, LatPlanα. LatPlan works in 3 phases. In Phase 1 (symbol-grounding, Sec. 3.1), a State AutoEncoder providing a bidirectional mapping between raw data (e.g., images)[1] and symbols is learned (unsupervised) from a set of unlabeled images of representative states. In Phase 2 (action model generation, Sec. 3.2), the operators available in the domain is generated from a set of pairs of unlabeled images, and a PDDL domain model is generated. In Phase 3 (planning, Sec. 3.3), a planning problem instance is input

[1] Although the LatPlan architecture can, in principle, be applied to various unstructured data input including images, texts or low-level sensors, in the rest of the paper we refer to "images" for simplicity and also because the current implementation is image-based.

as a pair of images (i, g) where i shows an *initial state* and g shows a *goal state*. These are converted to symbolic form using the SAE, and the problem is solved by the symbolic planner. For example, an 8-puzzle problem instance in our system consists of an image of the start (scrambled) configuration of the puzzle (i), and an image of the solved state (g). Finally, the symbolic, latent-space plan is converted to a sequence of human-comprehensible images visualizing the plan (Sec. 3.4).

3.1 Symbol Grounding with a State Autoencoder

The State Autoencoder (SAE) provides a bidirectional mapping between images and a symbolic representation.

First, note that a direct 1-to-1 mapping between images and discrete objects can be trivially obtained simply by using the array of discretized pixel values as a "symbol". The model generation method of Sec. 3.2 could be applied to such "symbols". However, such a trivial SAE lacks the crucial properties of *generalization* – ability to encode/decode unforeseen world states to symbols – and *robustness* – two similar images that represent "the same world state" should map to the same symbolic representation. Thus, we need a mapping where the symbolic representation captures the "essence" of the image, not merely the raw pixel vector. The main technical contribution of this paper is the proposal of a SAE which is implemented as a Variational Autoencoder (Kingma et al. 2014) with a Gumbel-Softmax (GS) activation function (Jang, Gu, and Poole 2017).

The SAE is comprised of multilayer perceptrons combined with Dropouts and Batch Normalization in both the encoder and the decoder networks, with a GS layer in between. The input to the GS layer is the flat, last layer of the encoder network. The output is an (N, M) matrix where N is the number of categorical variables and M is the number of categories. The input is fed to a fully connected layer of size $N \times M$, which is reshaped to a (N, M) matrix and processed by the GS activation function.

Our key observation is that these categorical variables can be used directly as propositional symbols by a symbolic reasoning system, i.e., this provides a solution to the symbol grounding problem in our architecture. We obtain the propositional representation by specifying $M = 2$, effectively obtaining N propositional state variables. It is possible to specify different M for each variable and represent the world using multi-valued representation as in SAS+ (Bäckström and Nebel 1995). In this paper, we use $M = 2$ for all variables for simplicity, and also for leveraging GPU parallelism by running the computation as a matrix operation. This does not affect the expressive power in the model induced by the SAE because bitstrings of sufficient length can represent arbitrary integers in multi-valued encoding.

The trained SAE provides bidirectional mapping between the raw inputs (subsymbolic representation) to and from their symbolic representations:

- $b = Encode(r)$ maps an image r to a boolean vector b.
- $\tilde{r} = Decode(b)$ maps a boolean vector b to an image $\tilde{r}$.

$Encode(r)$ maps raw input r to a symbolic representation by feeding the raw input to the encoder network, extract the activation in the GS layer, and take the first row in the $N \times 2$ matrix, resulting in a binary vector of length N. Similarly, $Decode(b)$ maps a binary vector b back to an image by concatenating b and its complement $\bar{b}$ to obtain a $N \times 2$ matrix and feeding it to the decoder network. These are lossy compression/decompression functions, so in general, $\tilde{r} = Decode(Encode(r))$ is similar to r, but may have negligible errors from r. This is acceptable for our purposes.

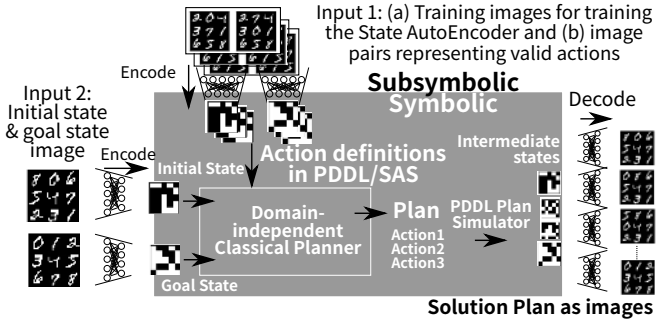


Figure 3: Classical planning in latent space: We use the learned State AutoEncoder (Sec. 3.1) to convert pairs of images $(pre, post)$ first to symbolic ground actions and then to a PDDL domain (Sec. 3.2) We also encode initial and goal state images into a symbolic ground actions and then a PDDL problem. A classical planner finds the symbolic solution plan. Finally, intermediate states in the plan are decoded back to a human-comprehensible image sequence.

It is *not* sufficient to simply use traditional activation functions such as sigmoid or softmax and round the continuous activation values in the latent layer to obtain discrete 0/1 values. As explained in Sec. 3.4, we need to map the symbolic plan back to images, so we need a decoding network trained for 0/1 values approximated by a smooth function, e.g., GS or similar approach such as (Maddison, Mnih, and Teh 2017). A rounding-based scheme would be unable to restore the images from the latent layer because the decoder network is trained using continuous activation values. Also, representing the rounding operation as a layer of the network is infeasible because rounding is non-differentiable, precluding backpropagation-based training of the network.

In some domains, an SAE trained on a small fraction of the possible states successfully generalizes so that it can $Encode$ and $Decode$ every possible state in that domain. In all our experiments below, on each domain, we train the SAE using randomly selected images from the domain. For example, on the 8-puzzle, the SAE trained on 12000 randomly generated configurations out of 362880 possible configurations is used by the domain model generator (Sec. 3.2) to $Encode$ every 8-puzzle state.

3.2 Domain Model Generation

The model generator takes as input a trained SAE, and a set R contains pairs of raw images. In each image pair $(pre_i, post_i) \in R$, pre_i and $post_i$ are images representing the state of the world before and after some action a_i is executed, respectively. In each ground action image pair,

the "action" is implied by the difference between pre_i and $post_i$. The output of the model generator is a PDDL domain file for a grounded unit-cost STRIPS planning problem.

For each $(pre_i, post_i) \in R$ we apply the learned SAE to pre_i and $post_i$ to obtain $(Encode(pre_i), Encode(post_i))$, the symbolic representations (latent space vectors) of the state before and after action a_i is executed. This results in a set of symbolic ground action instances A.

Ideally, a model generation component would induce a complete action model from a limited set of symbolic ground action instances. However, *action model learning* from a limited set of action instances is a nontrivial area of active research (Cresswell, McCluskey, and West 2013; Gregory and Cresswell 2015; Konidaris, Kaelbling, and Lozano-Pérez 2014; Mourão et al. 2012; Yang, Wu, and Jiang 2007; Celorrio et al. 2012). Since the focus of this paper is on the overall LatPlan architecture and the SAE, we leave model induction for future work.

Instead, the current implementation LatPlanα uses a trivial, baseline strategy which generates a model based on *all* ground actions, which are supposed to be easily replaced by existing off-the-shelf action model learner. In this baseline method, R contains image pairs representing all ground actions that are possible in this domain, so $A = \{Encode(r) | r \in R\}$ contains all symbolic ground actions possible in the domain. In Sec. 6, we further discuss the implication and the impact of this model.

In the experiments (Sec. 4), we generate image pairs for all ground actions using an external image generator. It is important to note that while R contains all possible actions, R is not used for training the SAE. As explained in Sec. 3.1, the SAE is trained using at most 12000 images while the entire state space is much larger.

LatPlanα compiles A directly into a PDDL model as follows. For each action $(Encode(pre_i), Encode(post_i)) \in A$, each bit $b_j (1 \leq j \leq N)$ in these boolean vectors is mapped to propositions `(b`$_j$`-true)` and `(b`$_j$`-false)` when the encoded value is 1 and 0 (resp.). $Encode(pre_i)$ is directly used as the preconditions of action a_i. The add/delete effects of action i are computed by taking the bitwise difference between $Encode(pre_i)$ and $Encode(post_i)$. For example, when b_j changes from 1 to 0, it compiles into `(and (b`$_j$`-false) (not (b`$_j$`-true)))`.

The initial and the goal states are similarly created by applying the SAE to the initial and goal images.

3.3 Planning with an Off-the-Shelf Planner

The PDDL instance generated in the previous step can be solved by an off-the-shelf planner. LatPlanα uses the Fast Downward planner (Helmert 2006). However, on the models generated by LatPlanα, the invariant detection routines in the Fast Downward PDDL to SAS translator (translate.py) became a bottleneck, so we wrote a trivial, replacement PDDL to SAS converter without the invariant detection.

LatPlan inherits all of the search-related properties of the planner which is used. For example, if the planner is complete and optimal, LatPlan will find an optimal plan for the given problem (if one exists), with respect to the portion of the state-space graph captured by the acquired model. Domain-independent heuristics developed in the planning literature are designed to exploit structure in the domain model. Although the structure in models acquired by LatPlan may not directly correspond to those in hand-coded models, intuitively, there should be some exploitable structure. The search results in Sec. 4.3 suggest that the domain-independent heuristics can reduce the search effort.

3.4 Visualizing/Executing the Plans

Since the actions comprising the plan are SAE-generated latent bit vectors, the "meaning" of each symbol (and thus the plan) is not necessarily clear to a human observer. However, we can obtain a step-by-step visualization of the world (images) as the plan is executed (e.g. Fig. 4) by starting with the latent state representation of the initial state, applying (simulating) actions step-by-step (according to the PDDL model acquired above) and $Decode$'ing the latent bit vectors for each intermediate state to images using the SAE.

In this paper, we evaluate LatPlan in Sec. 4 using puzzle domains such as the 8-puzzle, LightsOut, and Towers of Hanoi. Thus, physically "executing" the plan is not necessary, as finding the solution to the puzzles is the objective, so a "mental image" of the solution (i.e., the image sequence visualization) is sufficient. In domains where actions have effects in the world, it will be necessary to consider how actions found by LatPlan (transitions between latent bit vector pairs) can be mapped to actuation (future work).

4 Experimental Evaluation

All of the SAE networks used in the evaluation have the same network topology except the input layer which should fit the size of the input images. They are implemented with TensorFlow and Keras libraries under 5k LOC. We did not put much effort in parameter tuning. All layers except GS in the network are the very basic ones introduced in a standard tutorial.

The network consists of the following layers: [Input, GaussianNoise(0.1), fc(4000), relu, bn, dropout(0.4), fc(4000), relu, bn, dropout(0.4), fc(49x2), GumbelSoftmax, dropout(0.4), fc(4000), relu, bn, dropout(0.4), fc(4000), relu, bn, dropout(0.4), fc($input$), sigmoid]. Here, fc = fully connected layer, bn = Batch Normalization, and tensors are reshaped accordingly. The last layers can be replaced with [fc($input \times 2$), GumbelSoftmax, TakeFirstRow] for better reconstruction when we can assume that the input image is binarized. The network is trained to minimize the sum of the variational loss and the reconstruction loss (binary cross-entropy) using Adam optimizer (lr:0.001) for 1000 epochs.

The latent layer has 49 bits, which sufficiently covers the total number of states in any of the problems that are used in the following experiments. This could be reduced for each domain (made more compact) with further engineering.

4.1 Solving Various Puzzle Domains with LatPlan

MNIST 8-puzzle This is an image-based version of the 8-puzzle, where tiles contain hand-written digits (0-9) from the MNIST database (LeCun et al. 1998). Each digit is shrunk

to 14x14 pixels, so each state of the puzzle is a 42x42 image. Valid moves in this domain swap the "0" tile with a neighboring tile, i.e., the "0" serves as the "blank" tile in the classic 8-puzzle. The entire state space consists of 362880 states (9!). From any specific goal state, the reachable number of states is 181440 (9!/2). Note that the same image is used for each digit in all states, e.g., the tile for the "1" digit is the same image in all states.

Out of 362880 images, 12000 randomly selected images are used for training the SAE. This set is further divided into a training set and a validation set, each consisting of 11000 and 1000 images, where the actual backpropagation-based training of the network is performed on the training set, and the validation set is not given to the learner. Validation set represents the unseen instances: It is later used for ensuring the network is not overfitting, by computing the reconstruction loss $|r - \tilde{r}|$ of the validation set and ensure that it is comparable to the reconstruction loss of the training set. Training takes about 40 minutes with 1000 epochs on a single NVIDIA GTX-1070.

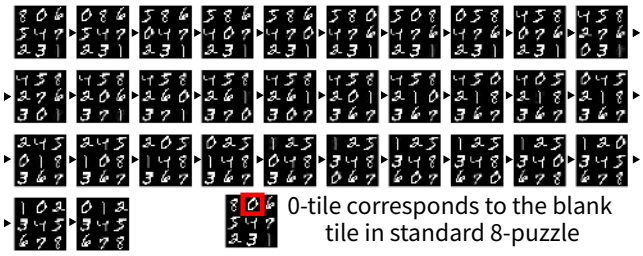


Figure 4: Output of solving the MNIST 8-puzzle instance with the longest (31 steps) optimal plan. [Reinefeld 1993]

Scrambled Photograph 8-puzzle The above MNIST 8-puzzle described above consists of images where each digit is cleanly separated from the black region. To show that LatPlan does not rely on cleanly separated objects, we solve 8-puzzles generated by cutting and scrambling real photographs (similar to sliding tile puzzle toys sold in stores). We used the "Mandrill" image, a standard benchmark in the image processing literature. The image was first converted to greyscale and then rounded to black/white (0/1) values. The same number of images as in the MNIST-8puzzle experiments are used.

Towers of Hanoi (ToH) Disks of various sizes must be moved from one peg to another, with the constraint that a larger disk can never be placed on top of a smaller disk. We generated the training and planning inputs for this task, with 3 and 4 disks. Each input image has a dimension of 24×122 and 32×146 (resp.), where each disk is presented as a 8px line segment.

Due to the smaller number of states (3^d states for d disks), we used images of all states as the set of images for training SAE. This is further divided into the training set (90%) and the validation set (10%), and we verified that the network has learned a generalized model without overfitting.

3-disk ToH is solved successfully and optimally using the default hyperparameters (Fig. 6, top). However, as the

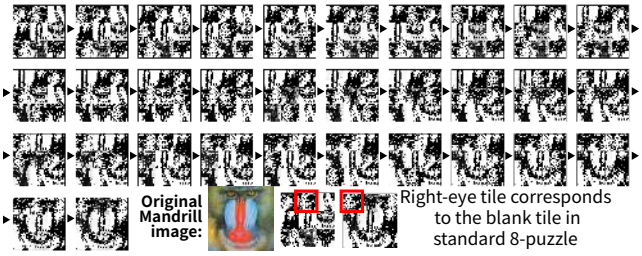


Figure 5: Output of solving a photograph-based 8-puzzle (Mandrill). We emphasize that LatPlan has no built-in notion of "sliding object", or "tile arrangement"; furthermore, the SAE is being trained completely from scratch when LatPlan is applied to this scrambled photograph puzzle – there is no transfer/reuse of knowledge from the SAE learned for the MNIST 8-puzzle above.

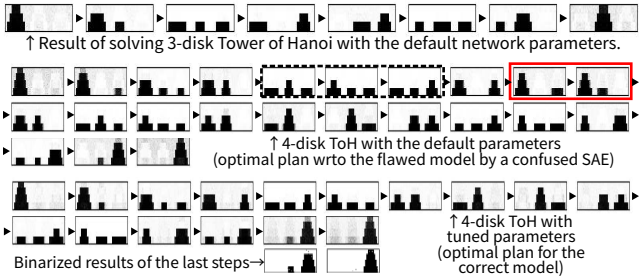


Figure 6: Output of solving ToH with 3 and 4 disks. The third picture is the result of SAE with different parameters.

images become more complex, training the SAE becomes more difficult. On 4-disks, the SAE trained with the default hyperparameters (Fig. 6, middle) is confused, resulting in a flawed model which causes the planner to choose suboptimal moves (dashed box). Sometimes, the size/existence of disks is confused (red box). Tuning the hyperparameters to reduce the SAE loss corrects this problem. Increasing the training epochs (10000) and tuning the network shape (fc(6000), $N = 29$) allows the SAE to learn a clearer model, allowing correct model generation, resulting in the optimal 15-step plan (Fig. 6, bottom).

LightsOut A video game where a grid of lights is in some on/off configuration (+: On), and pressing a light toggles its state (On/Off) as well as the state of all of its neighbors. The goal is all lights Off. Unlike the 8-puzzle where each move affects only two adjacent tiles, a single operator in 4x4 LightsOut can simultaneously flip 5/16 locations. Also, unlike 8-puzzle and ToH, the LightsOut game allows some "objects" (lights) to disappear. This demonstrates that LatPlan is not limited to domains with highly local effects and static objects.

4x4 LightsOut has $2^{16} = 65536$ states and $16 \times 2^{16} = 1048576$ transitions. Similar to the 8-puzzle instances, we used 12000 randomly selected images out of 65536 images, which is then divided into 11000 training images and 1000 validation images.

Twisted LightsOut In all of the above domains, the "ob-

Figure 7: Output of solving 4x4 LightsOut (left) and its binarized result (right). Although the goal state shows two blurred switches, they have low values (around 0.3) and disappear in the binarized image.

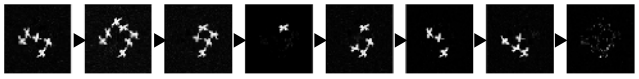


Figure 8: Output of solving 3x3 Twisted LightsOut.

jects" correspond to rectangles. To show that LatPlan does not rely on rectangular regions, we demonstrate its result on "Twisted LightsOut", a distorted version of the game where the original LightsOut image is twisted around the center. Unlike previous domains, the input images are not binarized.

4.2 Robustness to Noisy Input

We show the robustness of the system against the input noise. We corrupted the initial/goal state inputs by adding Gaussian or salt noise, as shown in Fig. 9. The system is robust enough to successfully solve the problem, because our SAE is a Denoising Autoencoder (Vincent et al. 2008) which has an internal *GaussianNoise layer* which adds a Gaussian noise to the inputs (only during training) and learn to reconstruct the original image from a corrupted version of the image.

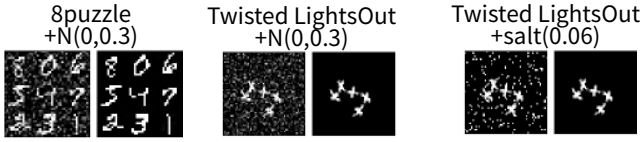


Figure 9: SAE robustness vs noise: Corrupted initial state image r and its reconstruction $Decode(Encode(r))$ by SAE on MNIST 8-puzzle and Twisted LightsOut. Images are corrupted by Gaussian noise of σ up to 0.3 for both problems, and by salt noise up to $p = 0.06$ for Twisted LightsOut. LatPlanα successfully solved the problems. The SAE maps the noisy image to the correct symbolic vector $b = Encode(r)$, conduct planning, then map b back to the denoised image $Decode(b)$.

4.3 Are Domain-Independent Heuristics Effective in Latent Space?

We compare search using a single PDB with greedy merging (Sievers, Ortlieb, and Helmert 2012) and blind heuristics (i.e., breadth-first search) in Fast Downward. The numbers of nodes expanded were:

- MNIST 8-puzzle (6 instances, mean(StdDev)): Blind 176658(25226), PDB **77811**(32978)
- Mandrill 8-puzzle (1 instance with 31-step optimal solution, corresponding to the 8-puzzle instance (Reinefeld 1993)): Blind 335378, PDB **88851**
- ToH (4 disks, 1 instance): Blind 55, PDB **17**,
- 4x4 LightsOut (1 instance): Blind 952, PDB **27**,
- 3x3 Twisted LightsOut (1 instance): Blind 522, PDB **214**

The domain-independent PDB heuristic significantly reduced node expansions. Search times (< 3 seconds for all instances) were also faster for all instances with the PDB. Although total runtimes including heuristic initialization is slightly slower than blind search, in domains where goal states and operators are the same for all instances (e.g., 8-puzzle) PDBs can be reused (Korf and Felner 2002), and PDB generation time can be amortized across many instances.

While the symbolic representation acquired by LatPlan captures the state space graph of the domain, the propositions in the latent space do not necessarily correspond to conceptual propositions in a natural, hand-coded PDDL model. Although these results show that existing heuristics for classical planning are able to reduce search effort compared to blind search, much more work is required in order to understand how the features in latent space interact with existing heuristics. In addition, a deeper understanding of the symbolic latent space may lead to new search heuristics which better exploit the properties of latent space.

5 Related Work

Konidaris, Kaelbling, and Lozano-Pérez propose a method for generating PDDL from a low-level, sensor actuator space of an agent characterized as a semi-MDP (2014). The inputs to their system are 33 variables representing structured sensor input (e.g., x/y distances between each effector and each object, light level) and categorical states (the on/off state of a button, whether the monkey has cried out). It does not explicitly deal with robustness with regard to noisy sensor input, and they focus on action model learning. In contrast, the inputs to LatPlan are unstructured images (e.g., for the 8-puzzle, 42x42=1764-dimensional arrays). We focus mostly on generating propositions from noisy, unlabeled raw images via our neural-net based SAE, and LatPlanα does not perform action model learning (Sec. 3.2). Integrating action modeling (Konidaris, Kaelbling, and Lozano-Pérez 2014; Mourão et al. 2012; Yang, Wu, and Jiang 2007) is a direction for future work.

Our approach differs from the work on learning from observation (LfO) in the robotics literature (Argall et al. 2009) in that: (1) LatPlan is trained based on image pairs showing valid before/after images of valid individual actions, while LfO work is largely based on observation (e.g., of videos) of plan executions; (2) LatPlan generates PDDL for symbolic planners which are suited for high-level (puzzle-like) tasks, while LfO focuses on tasks such as motion planning/manipulation. A closely related line of work in LfO is learning of board game play from observation of video/images (Barbu, Narayanaswamy, and Siskind 2010; Kaiser 2012; Kirk and Laird 2016). These works make relatively strong assumptions about the environment, e.g., that

there is a grid-like environment with "piece"-like objects. In contrast, as shown in Sec. 4, LatPlan does not make assumptions about the contents of the images.

There is a large body of previous work using neural networks to directly solve combinatorial search/planning tasks, starting with the well-known use of neural network to solve the TSP (Hopfield and Tank 1985). With respect to state-space search problems similar to those we consider, Neuro-solver, a neural network where each node corresponds to a state in the search space (Bieszczad and Pagurek 1998), has been used to solves Tower of Hanoi (Bieszczad and Kuchar 2015). Although such solvers use neural networks to solve the search problem, they assume a fully symbolic representation of the problem as input.

Previous work combining symbolic search algorithms and NNs embedded NNs *inside* a search algorithm to provide search control knowledge (Silver et al. 2016; Arfaee, Zilles, and Holte 2011; Satzger and Kramer 2013). In contrast, we use a NN-based SAE for symbol grounding, not for search control.

Deep Reinforcement Learning (DRL) has solved complex problems where the input is provided as images, performing well on many video games (Mnih et al. 2015). For unit-action-cost planning, LatPlan does not require a reinforcement signal (reward function). Also, LatPlan does not require *expert* solution traces, but only a random sample of the valid moves. Access to expert traces (as in the game of Go (Silver et al. 2016)) is a significant limitation of RL approach because such data may not be readily available. Finally, since LatPlanα uses a classical planner, it can provide guarantees of completeness and solution cost optimality (with regard to the acquired domain model) on deterministic, fully-observable single-agent domains. An interesting avenue for future work is extending our SAE-based approach as a symbol grounding mechanism for a symbolic, probabilistic (MDP) planner.

6 Discussion and Conclusion

We proposed LatPlan, an integrated architecture for domain model acquisition and planning which, given only a set of unlabeled images and no prior knowledge, generates a classical planning problem model, solves it with a symbolic planner, and presents the resulting plan as a human-comprehensible sequence of images. We demonstrated its feasibility using image-based versions of planning/state-space-search problems (8-puzzle, Towers of Hanoi, Lights Out). *The key technical contribution is the SAE, which leverages the Gumbel-Softmax reparametrization technique (Jang, Gu, and Poole 2017) and learns (unsupervised) a bidirectional mapping between raw images and a propositional representation usable by symbolic planners*. For example, as shown in Sec. 4, on the MNIST 8-puzzle, the "gist" of 42x42 training images are compressed into 49-bit representations that capture the essence of the images which is robust to noise.

Aside from the key assumptions that (1) the domain can be modeled and solved as a classical planning problem, and (2) the domain can be correctly inferred from the given training images, we avoid assumptions about the input domain. Thus, we have shown that domains with significantly different characteristics can all be solved by the same system, without modifying any code or manually modifying the neural network architecture. In other words, *LatPlan is a domain-independent, image-based classical planner*.

To our knowledge, LatPlan is the first system which completely automatically constructs a logical representation usable by an off-the-shelf symbolic planner from a set of unlabeled images for a diverse set of problems, with no explicit assumptions or knowledge about the nature of the domains other than the assumption that the domain can be solved by classical planning and that a sufficient set of training images is available. However, as a proof-of-concept first implementation, it has significant limitations to be addressed in future work.

6.1 Automated Validation of Plans

Since this paper focuses on demonstrating the *feasibility* of a symbolic planning system with neural perception, the experimental results included in the paper is mostly qualitative. To facilitate a more quantitative evaluation of LatPlan, one important direction for future work includes the development of a practical methods for validating plans. LatPlan should return a valid visual plan, i.e., a plan which does not violate the rules in the original input. In Sec. 4, we observed that the result plan may be invalid depending on the performance of the SAE. In classical planning, the validation of the solution plan is trivial as the plan simulator is readily available. However, validating the plan returned by LatPlan requires manual validation by human eye which checks every moves presented in the result, which does not scale to a large number of problem instances/plans, and is also prone to errors. This prevents a convenient and reliable evaluation of LatPlan. One approach to this problem is to use crowd sourcing infrastructure like Amazon Mechanical Turk, in combination with techniques for reducing the variance of the evaluation (Yuen, King, and Leung 2011). However, development of automated methods for plan validation is more preferable. Approaches such as the *inception score* for evaluating the performance of Generative Adversarial Network (Salimans et al. 2016) suggests that such automated approaches may be feasible for LatPlan as well.

6.2 Action Learning

The SAE successfully solves the problem of mapping a pair of images into a ground symbolic action using only a subset of the world state images (Sec. 3.2). On the other hand, the domain model generator in the current LatPlanα implementation does *not* perform action model learning/induction from a small set of sample actions.

Since the focus of this paper is the evaluation of SAE and not action learning, the current, baseline domain model generator requires the entire set of latent states/transitions, which in turn requires an image for each state in the state space. LatPlanα essentially constructs an explicit state space graph based on action image pairs for all ground actions in the domain, so the planner is being used to find optimal paths in an explicit graph. In other words, the preconditions of the actions generated by LatPlanα specify every

bit in a state, unlike the partial specification of a state that is common in IPC benchmark domains which allows the implicit definition of very large state spaces. This kind of trivial domain model generator which uses images for all states in the entire state space is obviously impractical in many domains. Aside from the modeling issue, the trivial explicit state-space model causes practical issues with current off-the-shelf planners, as the huge total number of actions (edges in the explicit state space model) causes major slowdowns in both the PDDL parser (Sec. 3.3), as well as the initialization runtime of heuristics (Sec. 4.3).

However, these are **not** *fundamental limitations of the LatPlan architecture*. The current primitive model generator in LatPlanα is merely a placeholder which was necessary to enable us to investigate the utility of the SAEs (our major contribution) and the overall feasibility of an end-to-end planning system based on raw images. Thus, the focus of our work and our contributions are orthogonal and complementary to the goals of previous work on domain model learning. To our knowledge, all previous planning domain model learning methods assume/require as input representations of states which are highly structured (e.g., propositional). We believe that it should be possible to replace our primitive generator with a more sophisticated generator (Cresswell, McCluskey, and West 2013; Konidaris, Kaelbling, and Lozano-Pérez 2014; Mourão et al. 2012; Yang, Wu, and Jiang 2007) which have been developed for environments with deterministic effects and fully observable states (Celorrio et al. 2012, Sec.3,Fig.2).

Another related direction for future work is how to specify the goal condition for LatPlan. Since LatPlanα assumes a single goal state as an input, developing a method for specifying a set of goal states with a partial goal specification as in IPC domains is an interesting future work. For example, one may want to tell the planner "the goal states must have tiles 0,1,2 in the correct places" in a MNIST 8-puzzle instance.

6.3 Collecting Images

LatPlan is a proof-of-concept architecture which shows the feasibility of constructing an end-to-end planning which uses raw images as input. As such, we did not address the practical issue of how the images are obtained by a sytem which uses image-based planning. However, in a real-world implementation of a system like LatPlan i.e., an image-based planner for a robot with sensors/camera and manipulators, collecting the data (images) needed to train the SAE is a practical issue. The requirement for collecting data for LatPlan would be significantly different from those that are common in learning-from-observatoin systems.

Unlike learning-from-obseration, where the learner does not know when an action starts/ends and should recognize each action from the plan traces, we assume that a robot has a lower-level manipulation capabilities of safely, randomly exploring the world by itself, deciding to initiate/terminate its own action, manipulating the state of the world and observing the consequences.[2] The robot can perform a random walk, collecting images along the way. In many domains, physical constraints will ensure that the robot can only perform legal moves (e.g., the physical tile board in 8-puzzle, the touch display in a LightsOut video game). In domains such as Towers of Hanoi where illegal moves are physically possible, we can assume either that a teacher (e.g., human) prevents the robot from making illegal moves, or that the teacher filters the training images taken by the robot. If we further assume that it is possible to periodically "reset" the world (e.g., a teacher comes and resets the world into a random configuration), then, given enough time, the robot could obtain images of the entire state space.

In domains where obtaining training images is expensive, another bottleneck in LatPlan is the number of images required to train the SAE. Developing more effective learner for minimizing the training data, such as One-shot learning methods (Lake, Salakhutdinov, and Tenenbaum 2013), is an important direction for future work. Since a better learner needs less examples, the number of required images depends on the performance of AE, which is ongoing work in the DL community.

6.4 Improving the SAE

Although we showed that LatPlanα works on several kinds of images, including MNIST handwritten digits, photographs (Mandrill), and several synthetic images (Hanoi, Lights Out), we do *not* claim that the specific implementation of SAE used in this paper works robustly on all images/data. For example, some tuning of the image (shrinking/binarization) was necessary in order to get the SAE working for the photograph-based 8-puzzles. Making a truly robust autoencoder is *not* a problem unique to LatPlan, but rather, a fundamental problem in deep learning. *A contribution of this paper is the demonstration that it is possible to leverage some existing deep learning techniques quite effectively in an integrated learning/planning system*, and future work will seek to continue leveraging further improvements in deep learning and other image processing techniques.

References

Arfaee, S. J.; Zilles, S.; and Holte, R. C. 2011. Learning Heuristic Functions for Large State Spaces. *Artificial Intelligence* 175(16-17):2075–2098.

Argall, B.; Chernova, S.; Veloso, M. M.; and Browning, B. 2009. A Survey of Robot Learning from Demonstration. *Robotics and Autonomous Systems* 57(5):469–483.

Bäckström, C., and Nebel, B. 1995. Complexity Results for SAS+ Planning. *Computational Intelligence* 11(4):625–655.

Barbu, A.; Narayanaswamy, S.; and Siskind, J. M. 2010. Learning Physically-Instantiated Game Play through Visual Observation. In *ICRA*, 1879–1886.

Bieszczad, A., and Kuchar, S. 2015. Neurosolver Learning to Solve Towers of Hanoi Puzzles. In *IJCCI*, volume 3, 28–38.

Bieszczad, A., and Pagurek, B. 1998. Neurosolver: Neuromorphic General Problem Solver. *Information Sciences* 105(1-4):239–277.

[2]Note that if the learner controls when actions are initiated/terminated, action segmentation (identifying which images indicate the start/end of individual actions) is not a issue.

Celorrio, S. J.; de la Rosa, T.; Fernández, S.; Fernández, F.; and Borrajo, D. 2012. A Review of Machine Learning for Automated Planning. *Knowledge Eng. Review* 27(4):433–467.

Cresswell, S.; McCluskey, T. L.; and West, M. M. 2013. Acquiring planning domain models using *LOCM*. *Knowledge Eng. Review* 28(2):195–213.

Deng, J.; Dong, W.; Socher, R.; Li, L.-J.; Li, K.; and Fei-Fei, L. 2009. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR*, 248–255. IEEE.

Deng, L.; Hinton, G.; and Kingsbury, B. 2013. New Types of Deep Neural Network Learning for Speech Recognition and Related Applications: An Overview. In *ICASSP*, 8599–8603. IEEE.

Goodfellow, I.; Bengio, Y.; and Courville, A. 2016. *Deep Learning*. MIT Press. `www.deeplearningbook.org`.

Graves, A.; Wayne, G.; Reynolds, M.; Harley, T.; Danihelka, I.; Grabska-Barwińska, A.; Colmenarejo, S. G.; Grefenstette, E.; Ramalho, T.; Agapiou, J.; et al. 2016. Hybrid Computing using a Neural Network with Dynamic External Memory. *Nature* 538(7626):471–476.

Gregory, P., and Cresswell, S. 2015. Domain model acquisition in the presence of static relations in the LOP system. In *ICAPS*, 97–105.

Gumbel, E. J., and Lieblein, J. 1954. Statistical theory of extreme values and some practical applications: a series of lectures.

Helmert, M. 2006. The Fast Downward Planning System. *J. Artif. Intell. Res.(JAIR)* 26:191–246.

Hinton, G. E., and Salakhutdinov, R. R. 2006. Reducing the Dimensionality of Data with Neural Networks. *Science* 313(5786):504–507.

Hopfield, J. J., and Tank, D. W. 1985. "Neural" Computation of Decisions in Optimization Problems. *Biological cybernetics* 52(3):141–152.

Ioffe, S., and Szegedy, C. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *ICML*, 448–456.

Jang, E.; Gu, S.; and Poole, B. 2017. Categorical Reparameterization with Gumbel-Softmax. In *ICLR*.

Kaiser, L. 2012. Learning Games from Videos Guided by Descriptive Complexity. In *AAAI*.

Kingma, D., and Ba, J. 2014. Adam: A Method for Stochastic Optimization. *arXiv preprint arXiv:1412.6980*.

Kingma, D. P., and Welling, M. 2013. Auto-Encoding Variational Bayes. In *ICLR*.

Kingma, D. P.; Mohamed, S.; Rezende, D. J.; and Welling, M. 2014. Semi-supervised learning with deep generative models. In *NIPS*, 3581–3589.

Kirk, J. R., and Laird, J. E. 2016. Learning General and Efficient Representations of Novel Games Through Interactive Instruction. *Advances in Cognitive Systems* 4.

Konidaris, G.; Kaelbling, L. P.; and Lozano-Pérez, T. 2014. Constructing Symbolic Representations for High-Level Planning. In *AAAI*, 1932–1938.

Korf, R. E., and Felner, A. 2002. Disjoint Pattern Database Heuristics. *Artificial Intelligence* 134(1-2):9–22.

Lake, B. M.; Salakhutdinov, R. R.; and Tenenbaum, J. 2013. One-shot Learning by Inverting a Compositional Causal Process. In *NIPS*, 2526–2534.

LeCun, Y.; Bottou, L.; Bengio, Y.; and Haffner, P. 1998. Gradient-Based Learning Applied to Document Recognition. *Proc. of the IEEE* 86(11):2278–2324.

Maddison, C. J.; Mnih, A.; and Teh, Y. W. 2017. The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables. In *ICLR*.

Maddison, C. J.; Tarlow, D.; and Minka, T. 2014. A* sampling. In *NIPS*, 3086–3094.

McDermott, D. V. 2000. The 1998 AI Planning Systems Competition. *AI Magazine* 21(2):35–55.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-Level Control through Deep Reinforcement Learning. *Nature* 518(7540):529–533.

Mourão, K.; Zettlemoyer, L. S.; Petrick, R. P. A.; and Steedman, M. 2012. Learning STRIPS Operators from Noisy and Incomplete Observations. In *UAI*, 614–623.

Reinefeld, A. 1993. Complete Solution of the Eight-Puzzle and the Benefit of Node Ordering in IDA. In *IJCAI*, 248–253.

Ren, S.; He, K.; Girshick, R.; and Sun, J. 2015. Faster R-CNN: Towards Real-time Object Detection with Region Proposal Networks. In *NIPS*, 91–99.

Salimans, T.; Goodfellow, I.; Zaremba, W.; Cheung, V.; Radford, A.; and Chen, X. 2016. Improved Techniques for Training Gans. In *NIPS*, 2226–2234.

Satzger, B., and Kramer, O. 2013. Goal Distance Estimation for Automated Planning using Neural Networks and Support Vector Machines. *Natural Computing* 12(1):87–100.

Sievers, S.; Ortlieb, M.; and Helmert, M. 2012. Efficient Implementation of Pattern Database Heuristics for Classical Planning. In *SOCS*.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529(7587):484–489.

Srivastava, N.; Hinton, G. E.; Krizhevsky, A.; Sutskever, I.; and Salakhutdinov, R. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15(1):1929–1958.

Steels, L. 2008. The Symbol Grounding Problem has been solved. So what's next? In de Vega, M.; Glenberg, A.; and Graesser, A., eds., *Symbols and Embodiment*. Oxford University Press.

Vincent, P.; Larochelle, H.; Bengio, Y.; and Manzagol, P.-A. 2008. Extracting and Composing Robust Features with Denoising Autoencoders. In *ICML*, 1096–1103. ACM.

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.

Yuen, M.-C.; King, I.; and Leung, K.-S. 2011. A Survey of Crowdsourcing Systems. In *Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on*, 766–773. IEEE.

Planning-based Scenario Generation for Enterprise Risk Management

Shirin Sohrabi and **Anton V. Riabov** and **Octavian Udrea**
IBM T.J. Watson Research Center
1101 Kitchawan Rd, Yorktown Heights, NY 10598, USA
{ssohrab, riabov, udrea}@us.ibm.com

Abstract

Scenario planning is a commonly used method that various organizations use to develop their long term plans. Scenario planning for risk management puts an added emphasis on identifying the extreme yet possible risks that are not usually considered in daily operations. While a variety of methods and tools have been proposed for this purpose, we show that formulating an AI planning problem, and applying AI planning techniques to develop the scenarios provides a unique advantage for scenario planning. Our system, the Scenario Planning Advisor (SPA), takes as input the relevant news and social media trends that characterize the current situation, where a subset of them is selected to represent key observations, as well as the domain knowledge. The domain knowledge is acquired using a graphical tool, and then automatically translated to a planning domain. We use a planner to generate multiple plans explaining the observations and projecting future states. The resulting plans are clustered and summarized to generate the scenarios for use in scenario planning. We discuss our knowledge engineering methodology, lessons learned, and the feedback received from the pilot deployment of the SPA system in a large international company. We also show our experiments that measure planning performance and how balanced and informative the generated scenarios are as we increase the complexity of the problem.

1 Introduction

Scenario planning is a commonly used method for strategic planning (Schoemaker 1995). Scenario planning involves analyzing the relationship between forces such as social, technical, economic, environmental, and political trends in order to explain the current situation in addition to providing insights about the future. A major benefit to scenario planning is that it helps businesses or policy-makers learn about the possible alternative futures and anticipate them. While the expected scenarios are interesting for verification purposes, scenarios that are surprising to the users (e.g., policy-makers businesses) are the ones that are the most important and significant (Peterson *et al.* 2003).

Risk management is a set of principles that focus on the outcome for risk-taking (Stulz 1996). A variety of methods and standards for risk management under different assumptions have been developed (Avanesov 2009). In this paper, we address scenario planning for risk management, the problem of generating scenarios with a significant focus on identifying the extreme yet possible risks that are not usually considered in daily operations. The approach we take in this paper is different from previous work in that we reason about emerging risks based on observations from the news and social media trends, and produce scenarios that both describe the current situation and project the future possible effects of these observations. Our objective is not to find a precise answer, that is to predict or forecast, but rather to project the possible alternative scenarios that may need consideration. Each scenario we produce highlights the potential *leading indicators*, the set of facts that are likely to lead to a scenario, the *scenario and emerging risk*, the combined set of consequences or effects in that scenario, and the *business implications*, a subset of potential effects of that scenario that the users (e.g., policy-makers, businesses) care about. The business implications are akin to the set of possible goals.

For example, given a high inflation observation, economic decline followed by a decrease in government spending can be the consequences or the possible effects in a scenario, while decreased client investment in the company offerings is an example of a business implication (i.e., the resulting goal). Furthermore, an increase in the cost of transportation could have been the leading indicator for that scenario. To the best of our knowledge, we are the first to apply AI planning in addressing scenario planning for enterprise risk management. We believe that AI planning provides a very natural formulation for the efficient exploration of possible outcomes required for scenario planning.

In this paper, we propose to view the scenario planning problem for enterprise risk management as a problem that can be translated to an AI planning problem. An intermediate step is a plan recognition problem, where the set of given business implications forms the set of possible goals, and the observations are selected from the news and social media trends. The domain knowledge is acquired from the domain expert via a graphical tool and is then automatically

translated to an AI planing domain. AI planning is in turn used to address the plan recognition problem (Ramírez and Geffner 2009; Sohrabi *et al.* 2016a; 2017). Top-k planning or finding a set of high-quality plans is used to generate multiple plans that can be grouped into a scenario (Riabov *et al.* 2014; Sohrabi *et al.* 2016b). The set of plans is then clustered and summarized to generate the scenarios. Hence, each scenario is a collection of plans that explain the observations and considers the possible cascading effects of the actions to identify potential future outcomes.

2 System Architecture

The system architecture for our system, Scenario Planning Adviser (SPA), is shown in Figure 1. There are three major components. The planning engine, shown under the *Scenario Generation and Presentation* component, takes as input the output of the other two components: the *News Aggregation* component and the *Domain Knowledge* component. The *News Aggregation* component deals with analyzing the raw data coming from the news and social media feeds. To this end, several text analytics are implemented in order to find the information that is relevant for a particular domain as filtered by the provided Topic Model. The Topic Model, provided by the domain expert, includes the list of important people, organization, and keywords. The result of the *News Aggregation* component is a set of relevant key observations, a subset of which can be selected by the business user and is fed into the *Scenario Generation* component. The *Domain Knowledge* component captures the necessary domain knowledge in two forms, Forces Model and Forces Impact. The Forces Model is a description of the causes and effects for a certain force, such as social, technical, economic, environmental, and political trends, and is provided by a domain expert who have little or no AI planning background. Forces Model are captured by a Mind Map (`http://freemind.sourceforge.net/wiki/`), a graphical tool that encodes concepts and relations. An example of a Mind Map for the currency depreciation force is shown in Figure 3. The Forces Impact, describes potential likelihoods and impact of a cause (i.e., concepts with an edge going into the main force) or an effect (e.g., concepts with an edge going from the main force and all other cascading concepts). The *Scenario Generation* component takes the domain knowledge and the key observations and automatically generates a planning problem whose outcome when clustered in the post-processing step generates a set of alternative scenarios.

Our system is currently deployed for an international organization. We use a company name Acme, for anonymity, in our examples. The system generates thousand plans and presents three to six scenarios to the business user. The extensive feedback we have collected has been encouraging and helpful in improving our system. We report on our knowledge engineering efforts, collected feedback, and the lessons learned in the rest of this paper.

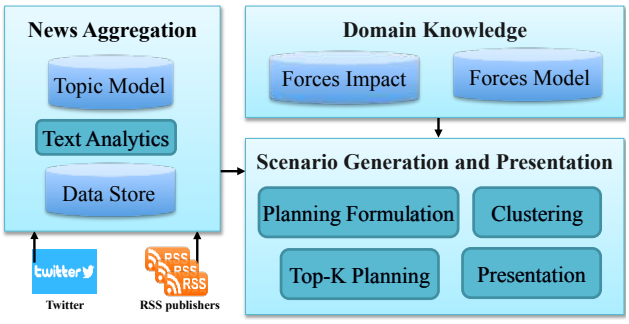


Figure 1: The SPA system architecture

3 Problem Definition

In this section, we briefly review necessary background on AI planning and Plan Recognition before defining the scenario planning for risk management problem.

Definition 1 *A planning problem is a tuple* $P = (F, A, I, G)$*, where* F *is a finite set of fluent symbols,* A *is a set of actions with preconditions,* $\text{PRE}(a)$*, add effects,* $\text{ADD}(a)$*, delete effects,* $\text{DEL}(a)$*, and action costs,* $\text{COST}(a)$*,* $I \subseteq F$ *defines the initial state, and* $G \subseteq F$ *defines the goal state.*

The solution to the planning problem, P, is a sequence of executable actions, $\pi = [a_0, ..., a_n]$ such that if executable from the initial state, I, meets the goal (i.e., $G \subseteq \delta(a_n, \delta(a_{n-1}, \ldots, \delta(a_0, I)))$, where $\delta(a, s) = ((s \backslash \text{DEL}(a)) \cup \text{ADD}(a))$ defines the successor state.

Definition 2 *A Plan Recognition (PR) problem is a tuple* $R = (F, A, I, O, \mathcal{G}, \text{PROB})$*, where* (F, A, I) *is the planning domain as defined above,* $O = \{o_1, ..., o_m\}$*, where* $o_i \in F$*,* $i \in [1, m]$ *is the set of (partially ordered) observations,* $\mathcal{G}$ *is the set of possible goals* G*,* $G \subseteq F$*, and* PROB *is a probability distribution over* $\mathcal{G}$*,* $P(G)$*.*

The solution to the PR problem is the posterior probabilities $P(\pi|O)$ and $P(G|O)$. Plan recognition problem can be transformed to an AI planning problem and the posterior probabilities can be approximated using AI planning (Ramírez and Geffner 2010; Sohrabi *et al.* 2016a). Note, the observations are said to be satisfied by an action sequence if it is either explained or discarded following the work of Sohrabi et al. 2016a. This allows for some observations to be left unexplained in particular if they are out of context with respect to the rest of the observations.

Definition 3 *A scenario planning for enterprise risk management problem is defined as a tuple* $SP = (F, A, I, O, \mathcal{G})$*, where* (F, A, I) *is the planning domain acquired by the domain experts,* $O = \{o_1, ..., o_m\}$*, where* $o_i \in F$*,* $i \in [1, m]$ *is a set of observations selected from the news and social media trends,* $\mathcal{G}$ *is a set of possible goals* $G \subseteq F$*; the set of goals are called business implications in the scenario planning problem.*

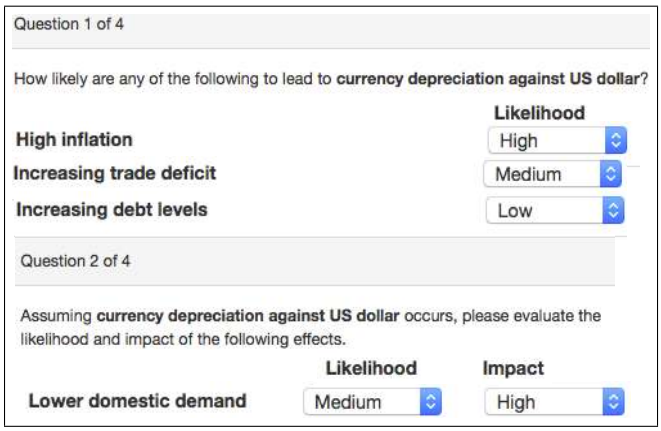


Figure 2: Sample questions

As shown in Figure 1, the input to the SPA system are raw social media posts and news articles with RSS feeds. The News Aggregation component analyzes such news and posts and suggests possible observations. In the deployment of the SPA system, we addressed unordered set of observations as input; however, in theory, the observations can be expressed in any Linear Temporal Logic (LTL) formula (Sohrabi *et al.* 2011).

The solution to the SP problem is defined as a set of scenarios, where each scenario is a collection of plans Π such that: (1) each plan $\pi = [a_0, ..., a_i, a_{i+1}, ..., a_n]$ is an action sequence that is executable from the initial state I and results in state $s = \delta(a_n, \ldots, \delta(a_0, I))$, (2) at least one of the goals is met (i.e., $\exists G \in \mathcal{G}$, where $G \subseteq s$), and (3) the set of observations is satisfied by the action sequence $[a_0, ..., a_i]$ (i.e., observations are either explained or discarded). The SP problem can be thought of as a plan recognition problem, where observations and a set of goals are given. Rather than computing $P(\pi|O)$ and $P(G|O)$, the solution to the SP problem is a set of scenarios showcasing the alternative possible outcomes.

4 Knowledge Engineering

While several knowledge engineering tools exists, most of them assume that the domain expert has some AI planning background and these tools provide the additional support in writing the domain knowledge (e.g., (Muise 2016; Simpson *et al.* 2007)). However, we anticipate the lack of proper AI planning expertise in writing the domain knowledge and the unwillingness to learn a planning language. Instead, the domain expert may choose to express their knowledge in a light-weight graphical tool and have this knowledge translated automatically to a planning language such as Planning Domain Description Language (PDDL) (McDermott 1998). In this section, we discuss the representation of the domain knowledge and its translation to planning.

As shown in Figure 1, the domain knowledge comes in two forms: Forces Model and Forces Impact. Forces Model, is the domain knowledge corresponding to the causes and effects of the different forces influencing the risks in a business organization such as the economy, currency, corruption, social unrest, and taxes. The domain experts express these relationship for each force trends (e.g., economic decline and economic growth) in separate Mind Maps. A Mind Map[1] is a graphical method that can be used to express the Forces Model in a simple way. The Mind Maps can be created in a tool such as FreeMind[2] which produces an XML representation of the Mind Maps which can serve as an input to our system. An example Mind Map is shown in Figure 3. The force in this Mind Map is the currency depreciation. The concepts with an edge going towards the force, are the possible causes, and the concepts with an outgoing edge from the force, are the possible effects. The causes and effects can appear in chains, and cascade to other causes, and effects, with a leaf concept of either a business implication (i.e., the planning goal), or another force, with its own separate Mind Map that describes it. For example, "Acme workforce capital available at better rates" is an example of a business implication, where Acme is the name of the organization. Note, one of the leafs of this Mind Map, economic decline, is another force which would be described in a separated Mind Map. Any of the concepts in the Mind Map, except for the business effects, can serve as observations in order to generate the scenarios.

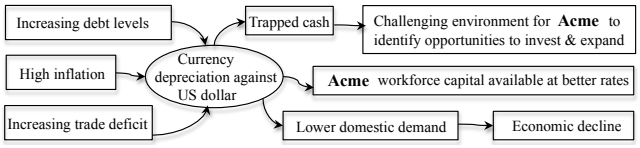


Figure 3: Part of the Mind Map for the currency depreciation against US dollar force.

Additional information on the Mind Maps is encoded through the Forces Impact, which is captured by a series of automatically generated questions based on the Mind Maps. These questions are created by a script that reads the XML encoding of the Mind Maps. Sample questions are shown in Figure 2. The domain expert is given options of low, medium, and high in addition to the option of "do not know" in which a default value is selected for them. The answers to these questions determine the weight of the edges in the Mind maps.

The domain knowledge encoded in the Mind Maps (i.e., Forces Model), together with the answers from the questionnaire (i.e., Forces Impact), is automatically translated into a planning language such as PDDL. There are at least two ways to translate the Mind Maps into a planning language. The first method, we call "ungrounded", defines one general and ungrounded set of actions in the PDDL domain file with many possible groundings of the actions based on the given Mind

[1] https://en.wikipedia.org/wiki/Mind_map
[2] http://freemind.sourceforge.net/wiki/

Maps. The domain file includes an action named "indicator" for each of the causes in a Mind Map. There would be three different "indicator" actions, one for each level (i.e., "indicator-low", "indicator-med" and "indicator-high"). The levels are determined based on the answers to the questionnaire. The domain file also includes an action named "next", and "next-bis" for each of the edges in the Mind Map. The "next" action also has three different versions, one for each level. The "next-bis" actions do not have levels and are those that end in a business implication concept (i.e., a concept that includes the name of the company).

Table 1 shows part of the planning domain. For example, the "next-med" action will be grounded by setting the parameter x1 to "increasing trade deficit" and the parameter x2 to the "currency depreciation against US dollar". Each of the "next" actions (-low, -med, -high) have a cost that maps to the importance of that edge such that lower impact/likelihood answers map to a higher cost. Hence, while the domain is fixed, based on the answers obtained by the domain experts, the actions will have a different set of possible groundings defined in the problem file. The "next-bis" action is the action that if executed, indicates that at least one of the business effects have been reached and the "bis-implication-achieved" predicate is set to true; this is the goal of the planning problem. The problem file (i.e., the initial state) will include all the possible groundings of these actions by including a grounding for the predicates "(next-med ?from ?to)", "(next-bis ?from ?to)", and "(indicator-med ?y ?x)". Note that the size of the Mind Map leads to a larger problem file, as the domain file is fixed. A successful plan maps to an execution of an "indicator" action, followed by the execution of one or more "next" actions, followed by an execution of a "next-bis" action. This maps to a path through the connected Mind Maps.

The second method to translate the Mind Maps into a planning language is called "grounded" which as the name suggests, defines one action per each edge in the Mind Map in addition to one action for each of the causes in the Mind Map in the planning domain itself. So rather than having one fixed planning domain which can get grounded by the problem file, the second approach fully specifies all the possible actions in the planning domain. We evaluate the performance of both methods in the experimental evaluation.

```
(:action next-med
 :parameters (?x1 - occ ?x2 - occ)
 :precondition (and (occur ?x1)
                    (next-med ?x1 ?x2))
 :effect (and (occur ?x2)
              (not (occur ?x1))
              (increase (total-cost) 10)))

(:action indicator-med
 :parameters (?y - force ?x - occ)
 :precondition (and (indicator-med ?y ?x))
 :effect (and (occur ?x)
              (increase (total-cost) 15)))

(:action next-bis
 :parameters (?x1 - occ ?x2 - bisimplication)
 :precondition (and (occur ?x1)
                    (next-bis  ?x1 ?x2)
 :effect (and (bis-implication-achieved)
              (increase (total-cost) 6)))
```

Table 1: Part of the planning domain.

5 Computing Plans

In the previous section, we discussed how to translate the information available in the Mind Maps into a planning domain and problem. However, we are also given the set of observations as the input and we need to compile away the observations in order to use planning. To do so we follow the work of Sohrabi et al. 2013; 2016a which adds a set of "explain" and "discard" actions for each observation. The discard action can be selected in order to leave some observations unexplained. The observations are driven from news and social media posts and not all of them are reliable; in addition, some of them could be mutually exclusive and not all of them could be explainable. Hence, it is important to have the ability to discard some observations. However, to encourage the planner to generate plans that explain as many observations as possible, a penalty is set for the "discard" action in the form of a cost. The penalty is relative to the cost of the other action in the domain; we currently set it to be five times the cost of a "next-med" action. After considering multiple options, this seemed to be good a middle-ground option between the two extremes; a high discard cost will cause the planner to consider many long and unlikely paths, while a low discard will cause the planner to discard observations without trying to explain them. In addition, to ensure all observations are considered, whether explained or discarded, a set of special predicates, one per each observation is used and must hold true for each of the "next-bis" actions. This ensures that a plan that meets one of the goals also has considered all of the observations. To disallow different permutation of the discard action, we discard observations using a fixed order.

The resulting planning problem captures both the domain knowledge that is encoded in the Mind Maps and its associated weights of the edges as well as the given set of observations, and possible set of goals, associated with the plan recognition aspect of the problem. To compute a set of high-quality plans on the transformed planning problem, we use the top-k planning approach proposed in (Riabov *et al.* 2014; Sohrabi *et al.* 2016b). Top-k planning is defined in as the problem of finding k set of plans that have the highest quality. The best known algorithm to compute the set of top-k plans is based on the k shortest paths algorithm called K^* (Aljazzar and Leue 2011) which also allows use of heuristics search. We use the K^* algorithm together with the LM-cut heuristic (Pommerening and Helmert 2012) in our system. Next, we discuss how the generated plans are post-processed into the scenarios.

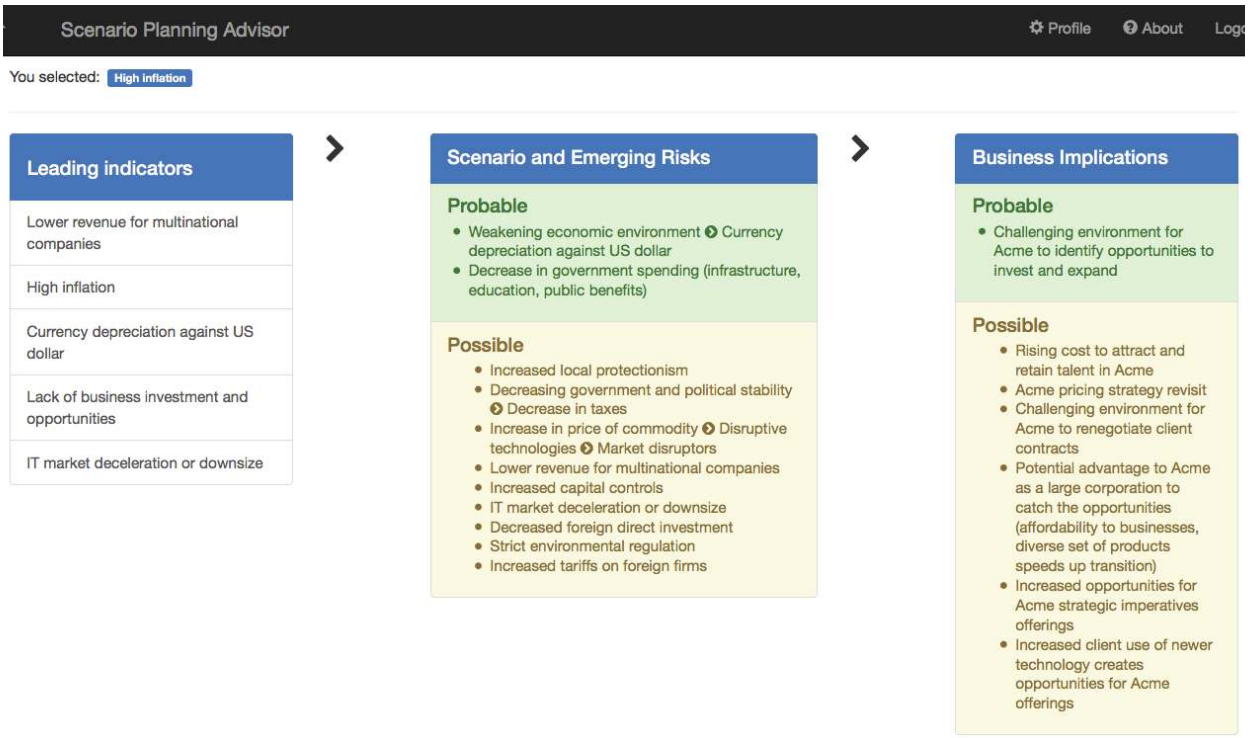


Figure 4: The screenshot of a sample generated scenario for the high inflation observation. Each scenario is divided into three parts, the leading indicators, scenario and emerging risks, and the business implications.

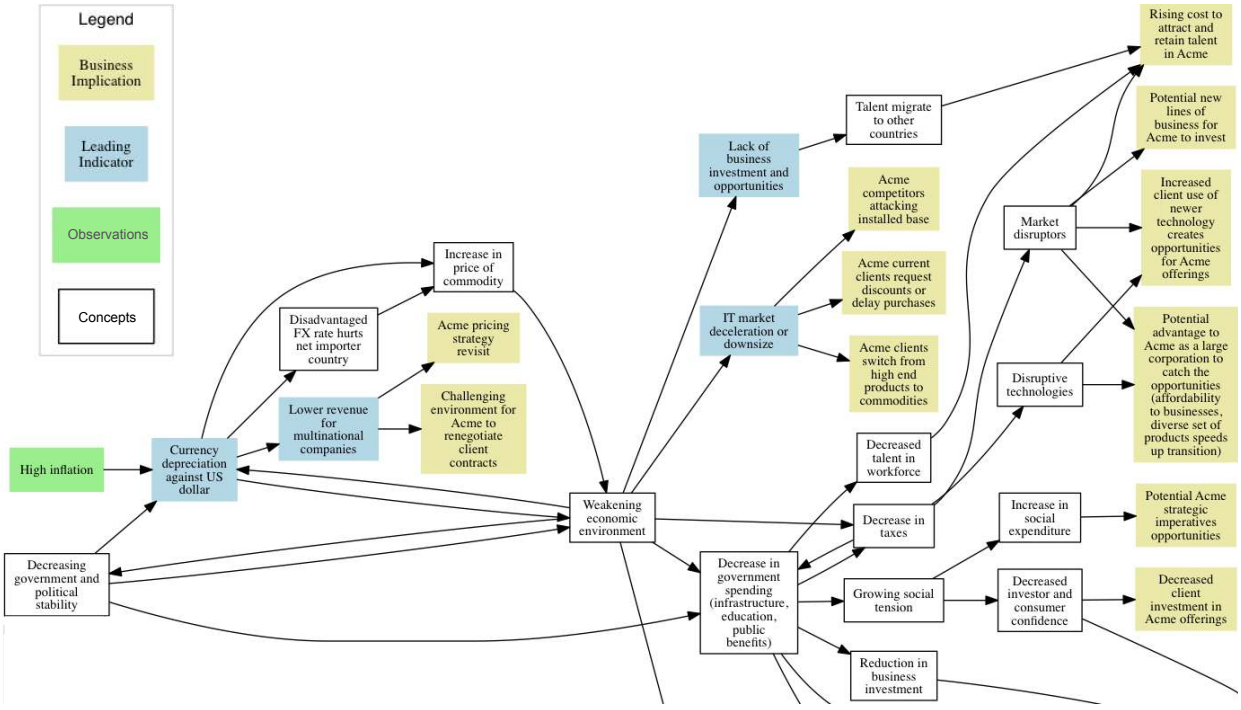


Figure 5: Part of the screenshot of a explanation graph for the scenario shown in Figure 4. Observations are shown in green, leading indicators are shown in blue, and business implications are shown in yellow.

6 Computing Scenarios

To compute the type of scenarios shown in Figure 4, we perform a set of post-processing steps on the computed set of plans. All of the post-processing steps are done automatically. First, we identify the number of plans out of the top-k plans (e.g., 1000) generated by the planner to consider for scenario generation. We argue that this number is problem-dependent rather than being a fixed number for all problems. To calculate the cost cutoff, we calculate the average and the standard deviation of the cost of all plans among the top-k plans. We then consider plans that have a lower cost than the average cost subtracted by the standard deviation. The number of plans considered for scenario generation is shown under the "# of Plans" column in Table 2.

Next, we cluster the resulting plans to create scenarios. Hence, rather than presenting all plans, we group similar plans and only present 3-6 clusters of plans to the end user. We cluster plans according to the predicates present in the last state. Given that the number of ground predicates (i.e, $\mathcal{F}$) is finite, we first represent each plan through a bit array of the same size such that 1 indicates the predicate is in the final state, and 0 indicates that the predicate is not in the final state. To determine the Euclidean distance between two plans, we compute a XOR of the corresponding bit arrays and take the square root of the sum of 1 bits. Normally, we want to avoid plans with opposite predicates (e.g., weakening/strengthening economic environment, increase/decrease in inflation, etc.) ending up in the same cluster. To ensure this, we add a penalty factor to the number of 1 bits we use to compute the distance for every pair of opposite predicates. Given this distance function for each pair of plans, we compute a dendrogram bottom-up using the complete-linkage clustering method (Defays 1977). The user can specify a minimum and maximum consumable number of scenarios. These settings are used to perform a cut through the dendrogram that yields the number of plans in the specified interval with the optimal Dunn index (Dunn 1973), a metric for evaluating clustering algorithms that favors tightly compact sets of clusters that are well separated.

After post-processing is complete, we automatically perform several tasks to prepare the scenarios for presentation. First, we separate the predicates in each cluster (scenario) into business implications and regular predicates. At the same time, we separate probable and possible predicates in each of these categories by determine the proportion of plans where the predicate is present in the last state from all plans in the scenario; predicates that appear in more than 66% of plans are put into the probable category, those that appear between 25% and 66% are placed in the possible category. Second, we identify discriminative predicates, i.e. predicates that appear early on the plans that are part of one scenario but not other scenarios (i.e., they tend to lead to this scenario and not others); these are useful to monitor in order to determine early on whether a scenario is likely to occur. Third, we compute a summary of all plans that are part of the scenario and present this as a graph to the user. Figure 5 shows an example of this graph. This serves as an explanatory tool for the predicates that are presented in each scenario. This graph also shows how the different Mind Maps are connected with each other through concepts that are shared between them.

7 Experimental Evaluations

In this section, we evaluate: (1) the performance of the planner, (2) quality of the clusters measured by the size of the cluster, and (3) how informative each cluster measured by number of predicates and business implications. In the next section, we provide details on the pilot deployment of the Scenario Planning Adviser (SPA) tool, feedback and the lessons learned in interacting with the domain experts as well as the business users. All our experiments were run on a 2.5 GHz Intel Core i7 processor with 16 GB RAM.

We compare the performance of the planner on our two proposed methods to translate the Mind Maps into a planning domain: "ungrounded" and "grounded". The "grounded" method creates 670 actions when considering the full set of Mind Maps. We remove some of these Mind Maps creating 403 actions instead and report on that result under the "ungrounded small" method. To increase the difficulty of the problem, we increase the size of the O. Observations are chosen randomly from the set of possible observations.

Table 2 presents a comparison between "ungrounded small" and "grounded". The objective of this experiment is to show how the planning domain size influence performance and the generated clusters. All numbers shown in each row are averages over 10 runs of the same type of problem, where the same number of observations is considered in both cases. The columns show the planning performance in seconds, total number of business implications, $\mathcal{G}$, number of actions, $\mathcal{A}$, number of observations O, number of discarded observations in the optimal plan, "# of Discards", number of plans considered for scenario generation, "# of Plans", and number of scenarios generated "# of Scenarios". We also show the average, standard deviation, max, and min count on the number of members of each cluster, number of predicates, and number of business implications in each scenario. We used a timeout of 900 seconds. The problems with 30 or more observations did not finish within the time limit.

The results show that the performance of the planner depends on both the number of observations and the size of the domain, as expected. As the number of observations grow the planner's performance worsens but this does not influence the number of plans, the number of scenarios, size of the clusters, or the number of scenario predicates. However, the number of business implications decreases, as expected, as the observation size grows. Looking at the average number of cluster members, the average number of scenarios predicates, and the average number of bossiness implications, the

	Time	$\|\mathcal{G}\|$	$\|\mathcal{A}\|$	$\|O\|$	#of Discards	#of Plans	#of Scenarios	Cluster Members Avg	σ	Max	Min	Scenario Predicates Avg	σ	Max	Min	Bis Implications Avg	σ	Max	Min
Ungrounded Small	0.03	65	403	1	0.0	129.0	3.8	37.0	28.6	76.9	11.2	9.6	2.7	13.5	6.6	4.8	1.7	7.1	2.7
	0.03	65	403	2	0.5	141.7	3.8	39.9	31.7	83.4	6.5	9.8	3.1	13.7	5.4	4.1	1.7	6.1	1.8
	0.05	65	403	4	1.6	120.5	3.6	34.6	27.9	72.1	7.9	10.9	2.7	13.9	6.9	3.7	1.2	5.3	2.1
	0.22	65	403	8	4.4	122.4	3.8	34.8	33.4	82.6	4.3	10.0	2.4	13.0	6.9	2.1	0.9	3.5	1.3
	0.80	65	403	10	5.0	112.6	4.5	25.6	26.0	71.5	5.6	7.6	2.0	10.1	5.4	2.3	0.8	3.8	1.6
	2.33	65	403	12	5.9	100.1	4.2	25.3	20.7	56.2	4.4	9.4	1.4	11.1	7.4	1.7	0.4	2.6	1.2
	9.16	65	403	15	8.8	104.8	3.9	30.2	25.6	68.5	8.8	10.6	1.2	12.4	8.8	1.9	0.4	2.8	1.5
	27.85	65	403	18	9.9	92.8	4.8	20.2	23.5	61.3	3.0	8.5	1.2	10.3	6.7	1.6	0.5	2.4	1.3
	103.71	65	403	20	11.3	117.7	3.9	30.9	26.8	68.0	3.7	9.0	1.4	11.0	7.3	1.8	0.6	2.5	1.0
	179.90	65	403	23	14.9	103.7	4.1	26.3	21.2	58.6	4.4	9.0	1.4	11.1	6.9	1.9	0.6	2.7	1.2
	282.87	65	403	26	16.9	90.6	4.9	20.3	19.0	53.5	5.3	9.5	1.1	11.3	7.8	1.6	0.3	2.0	1.3
Ungrounded	0.03	112	670	1	0.0	91.5	4.4	24.4	16.6	48.6	6.6	7.0	2.5	10.4	4.3	4.5	1.7	6.6	2.2
	0.04	112	670	2	0.4	132.1	4.3	34.4	32.2	80.3	3.7	8.0	3.0	11.7	4.0	3.8	1.8	6.1	1.5
	0.08	112	670	4	1.5	114.1	3.6	32.9	30.7	77.9	4.3	10.2	2.9	13.2	6.1	3.6	1.4	6.0	2.3
	0.35	112	670	8	3.7	109.7	3.6	31.5	24.6	65.9	7.0	9.1	1.9	11.4	6.5	3.8	1.3	5.4	2.3
	1.17	112	670	10	5.1	139.4	4.2	34.6	27.9	73.2	5.9	7.8	2.0	10.1	4.9	2.6	0.8	3.9	1.6
	3.35	112	670	12	5.4	99.5	4.8	22.8	24.8	64.7	3.5	8.6	1.9	11.0	6.3	1.6	0.4	2.8	1.2
	22.01	112	670	15	8.1	92.3	4.1	23.3	22.1	57.9	3.2	9.9	1.8	12.0	7.0	2.5	1.0	4.0	1.5
	85.73	112	670	18	9.4	88.5	4.3	22.2	19.2	51.6	6.7	7.2	1.1	8.7	5.5	2.2	0.3	2.7	1.7
	144.89	112	670	20	10.7	124.3	5.1	26.0	19.4	57.0	5.0	9.0	1.0	10.0	7.2	2.1	0.2	2.3	1.9
	284.73	112	670	23	14.5	106.8	4.8	24.5	23.9	62.5	4.0	8.6	1.6	10.6	6.5	2.9	0.6	3.6	2.0
	511.95	112	670	26	16.8	80.0	4.7	17.2	9.0	30.2	7.8	7.8	1.0	9.5	6.5	1.7	0.7	2.8	1.2

Table 2: Comparison between "ungrounded" and "ungrounded small" as we increase the number of observations: "grounded" considers all of the Mind Maps, "ungrounded small" considers a smaller set of Mind Maps.

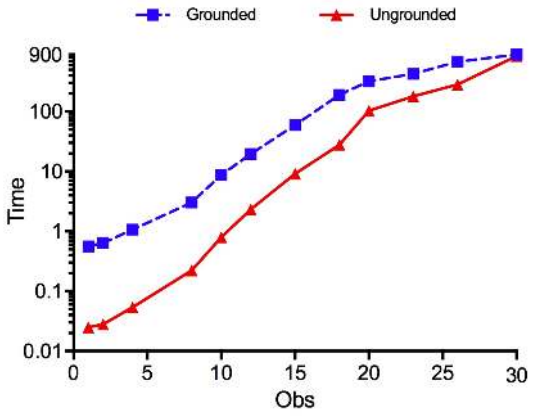


Figure 6: Planning performance comparison between the "grounded" and "ungrounded" methods, as we increase the number of observations. The time is in seconds and is shown in logarithmic scale.

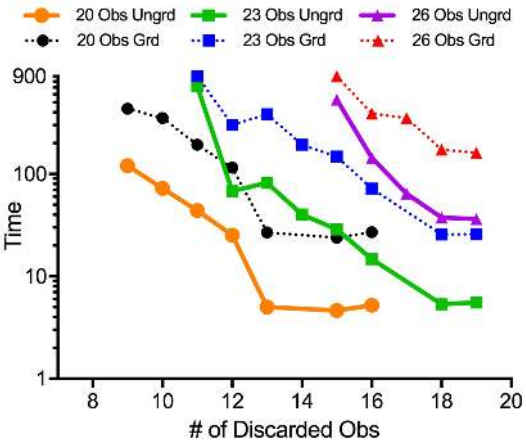


Figure 7: Planning performance comparison between two methods (i.e, "grounded" and "ungrounded") as the number of discarded observations in an optimal plan increases when considering problems with 20, 23, and 26 observations. The time is in seconds and is shown in logarithmic scale.

results show that the clusters in both cases are balanced and informative.

We also compare the planning performance between two methods of translating the Mind Maps. The results in logarithmic scale is shown in Figure 6. Each shown point in the figure is an average over 20 instances. The results show that in our current implementation, as the number of observations increases, planning performance using the "ungrounded" method is significantly better than the "grounded" method.

Considering problems with 20, 23, and 26 observations, we also looked at the number of discarded observations in the optimal plan in each case. This indicates whether or not the observations are explainable in a single path through the Mind Maps. The results in logarithmic scale is shown in Figure 7. The results confirm that the performance of the "ungrounded" method is better than the "grounded" method. It also shows that as the number of discarded observation increases, the planning time decreases. This seems to indicate that the planner identifies the unexplainable observations, through its heuristics, and does not spend time on explaining the unexplainable observations.

Based on these results, we conclude that performance of the planner depends on number of observations, the size of the domain, the method used in the translation of

the Mind Maps, as well as the number of unexplainable observations. Given this result, we deployed the "ungrounded" method and use the full set of Mind Maps.

8 User Experience

The SPA tool was evaluated in a pilot deployment with 7 teams of business users, whose responsibilities included risk management within their business area. For those teams SPA was introduced together with the new scenario planning process; hence, there was no pre-automation baseline available to compare against. In addition the functionality provided by the tool cannot be reproduced manually due to the broad news analysis the tool provides.

The Mind Map were developed over the course of three months by one enterprise risk management expert working with an assistant and in consultation with other experts. While Mind Maps in general can be in any form, we briefly educated the domain expert to provide Mind Maps that have one force (e.g., currency deprecation against US dollar) as their main concept and provide causes and consequences of this force in one Mind Map; the concepts with an edge to the central concept and the concepts with an edge from the main concept and their cascading effects where the last effect is either a business implication or another force with its own separate Mind Map. This ensures that we can automatically translate the Mind Maps into a planning language. We used 23 Mind Maps in the pilot deployment and used the "ungrounded" method to translate the Mind Maps. The resulting planning problem that aggregates the knowledge of all Mind Maps (i.e., the grounding of the actions based on the edges on the Mind Maps) has around 350 predicates and 670 actions.

Additionally, the end users (i.e., the analysts) provided us with a list of possible keywords, such as organizations of interest, key people, key topics, and were able to pick the relevant sequence of observations when we presented them with the summary of relevant news and RSS publications. For RSS publications, around 3,000 news abstracts from 64 publishers, and for Twitter, around 73,000 tweets from around 32,000 users matched our keyword search criteria.

The teams have universally found the tool easy to use and navigate. Although no detailed feedback was collected for each scenario, the teams have reported that approximately 80% of generated scenarios had identified implications directly or indirectly affecting the business. By design, the tool is trying to help the business users to think outside the box and it is expected that some of the scenarios it generates will not be relevant. Judging by the provided comments, the teams whose business is affected by frequent political, regulatory and economic change have found the tool more useful than those operating under relatively stable conditions.

In addition, the teams found the explanation graph, visualization of a set of plans, essential to the adaptation of the tool (Figure 5). They believe that the explanation graph "demystifies" the tool by providing them with an explanation of why they are presented with a particular scenario. This is critical for the business users or policy-makers who would be basing their decisions on the generated scenarios.

The suggestions for improvement focused primarily on the need for further automated assistance in selecting observations based on the news, to ensure that no important context is lost, and on the additional information about the scenarios. Several teams have requested confidence levels or at least ranking information provided with the generated scenarios. We believe this is an interesting future direction and believe more accurate models are required in order to provide that additional information.

In working with the domain experts and users from the start of the pilot deployment, we learned several lessons: (1) The users are interested in being presented with several scenarios rather than one along with the explanation of each scenario. This captures the possible alternatives rather than a precise prediction, analogous to a generation of a multiple plans rather than a single (optimal) plan; (2) The users wanted personalized scenarios specific to their particular use case. To address that we consider the Mind Maps as a template that holds true for all use cases and allow personalization of the scenarios by incorporating different weights of the edges of the Mind Maps. As mentioned previously we automatically generate a serious of questions in order to obtain the impact and likelihoods that are specific to a use case. Hence, computing a set of high-quality plans for different use cases results in different set of plans, which in turn results in different scenarios; (3) The domain experts found themselves continuously updating the Mind Maps after interacting with the tool and we had to enable those continuous updates. In addition to building the automated technique of translating the Mind Maps to planning language, we assigned unique identifiers to each of the concepts in the Mind Maps. This allowed us to develop scripts for supervised detection and propagation of the associated knowledge throughout theses changes.

9 Related Work and Summary

There exist a body of work on the plan recognition problem with several different approaches (e.g., (Zhuo *et al.* 2012)). However, most approaches assume that the observations are perfect, mainly because they do not take as input the raw data and that they do not have to analyze and transform the raw data into observations (Sukthankar *et al.* 2014). Also, most plan recognition approaches assume plan libraries are given as input, whereas we use AI planning (Goldman *et al.* 1999). Furthermore, there is a body of work on learning the domain knowledge (Yang *et al.* 2007; Zhuo *et al.* 2013). Our focus in addressing knowledge engineering challenges was to transform one form of knowledge, expressed in Mind Maps, into another form that is accessible by planners. Learning can be beneficial in domains in which plan traces are available.

In this paper, we applied AI planning techniques for a novel application, scenario planning for enterprise risk management and addressed knowledge engineering challenges of encoding the domain knowledge from domain experts. To this end, we designed Scenario Planning Adviser (SPA), that takes as input the raw data, news and social media posts, and interacts with the business user to obtain key observations. SPA also allows upload of Mind Maps, as one way of expressing the domain knowledge by the domain experts, and obtains additional information based on these Mind Maps by an automatically generated questionnaire. SPA then automatically generates scenarios by first generating large number of plans and then clustering the generated plans into a small set (i.e., 3-6) in order to be consumable by a human user. The SPA system is in pilot deployment with several teams of business users. The feedback we have received so far have been positive and show that our approach seems promising for this application.

10 Acknowledgements

We thank Fang Yuan and Finn McCoole at IBM for providing the domain expertise. We thank Nagui Halim and Edward Shay for their guidance and support. We also thank our LAS collaborators. This material is based upon work supported in whole or in part with funding from the Laboratory for Analytic Sciences (LAS). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the LAS and/or any agency or entity of the United States Government.

References

Husain Aljazzar and Stefan Leue. K*: A heuristic search algorithm for finding the k shortest paths. *Artificial Intelligence*, 175(18):2129–2154, December 2011.

Evgeny Avanesov. Risk management in ISO 9000 series standards. In *International Conference on Risk Assessment and Management*, volume 24, page 25, 2009.

D. Defays. An efficient algorithm for a complete link method. *Computer Journal*, 20(4):364–366, 1977.

J. C. Dunn. A fuzzy relative of the isodata process and its use in detecting compact well-separated clusters. *Journal of Cybernetics*, 3(3):32–57, 1973.

Robert P. Goldman, Christopher W. Geib, and Christopher A. Miller. A new model of plan recognition. In *Proceedings of the 15th Conference in Uncertainty in Artificial Intelligence (UAI)*, pages 245–254, 1999.

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

Christian Muise. Planning.Domains. In *the 26th International Conference on Automated Planning and Scheduling - Demonstrations*, 2016.

Garry D Peterson, Graeme S Cumming, and Stephen R Carpenter. Scenario planning: a tool for conservation in an uncertain world. *Conservation biology*, 17(2):358–366, 2003.

Florian Pommerening and Malte Helmert. Optimal planning for delete-free tasks with incremental LM-Cut. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS)*, 2012.

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

Miquel Ramírez and Hector Geffner. Probabilistic plan recognition using off-the-shelf classical planners. In *Proceedings of the 24th National Conference on Artificial Intelligence (AAAI)*, 2010.

Anton Riabov, Shirin Sohrabi, and Octavian Udrea. New algorithms for the top-k planning problem. In *Proceedings of the Scheduling and Planning Applications woRKshop (SPARK) at the 24th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 10–16, 2014.

Paul JH Schoemaker. Scenario planning: a tool for strategic thinking. *Sloan management review*, 36(2):25, 1995.

Ron M. Simpson, Diane E. Kitchin, and T. L. McCluskey. Planning domain definition using GIPO. *Knowledge Eng. Review*, 22(2):117–134, 2007.

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

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

Shirin Sohrabi, Anton Riabov, and Octavian Udrea. Plan recognition as planning revisited. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*, 2016.

Shirin Sohrabi, Anton Riabov, Octavian Udrea, and Oktie Hassanzadeh. Finding diverse high-quality plans for hypothesis generation. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI)*, 2016.

Shirin Sohrabi, Anton Riabov, and Octavian Udrea. State projection via ai planning. In *Proceedings of the 31st Conference on Artificial Intelligence (AAAI-17)*, 2017.

René M Stulz. Rethinking risk management. *Journal of applied corporate finance*, 9(3):8–25, 1996.

Gita Sukthankar, Christopher Geib, Hung Hai Bui, David V. Pynadath, and Robert P. Goldman. *Plan, Activity, and Intent Recognition*. Morgan Kaufmann, Boston, 2014.

Qiang Yang, Kangheng Wu, and Yunfei Jiang. Learning action models from plan examples using weighted max-sat. *Artificial Intelligence*, 171(2-3):107–143, February 2007.

Hankz Hankui Zhuo, Qiang Yang, and Subbarao Kambhampati. Action-model based multi-agent plan recognition. In *Proceedings of the 26th Annual Conference on Neural Information Processing Systems (NIPS)*, pages 377–385, 2012.

Hankz Hankui Zhuo, Tuan Nguyen, and Subbarao Kambhampati. Refining incomplete planning domain models through plan traces. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2451–2457, 2013.

Attribute Grammars with Set Attributes and Global Constraints as a Unifying Framework for Planning Domain Models

Roman Barták and Adrien Maillard

Charles University, Faculty of Mathematics and Physics

Malostranské nám. 25, Praha 1, Czech Republic

Abstract

The paper presents attribute grammars as a unifying framework for modeling planning domains and problems. The motivation is to exploit techniques from formal languages in domain model verification, plan and goal recognition, domain model acquisition, as well as in planning. Grammar rules are used for action selection while a specific set attribute is used to collect events (preconditions and effects of actions) that are ordered using a global timeline constraint. We show how classical STRIPS, hierarchical task networks, and procedural domain models are transformed to attribute grammars.

Introduction

There is an increasing interest in the connection between automated planning and formal language theory. As pointed in (Geib and Steedman 2007), there are many commonalities between these two fields. Generating valid sentences from a set of syntactic rules can be compared to planning from a set of actions and state variables, that implicitly define acceptable orderings of these actions. In the other way, *parsing*, as the process of assessing whether or not a sentence is grammatically correct with respect to a grammar, is close to *plan recognition* where the objective, given observations, is to recognize the goal of an agent with respect to a, generally hierarchical, model of actions. Links between these fields have lead to some bridging tentatives, for example in (Koller and Petrick 2011), where the problem of sentence generation is tackled with a planning technique. Generally speaking, highlighting similarities between subfields may open new areas of research. It allows to consider progresses made in both fields to be beneficial to each other.

More specifically, it has already been noted that derivation trees of Context-Free (CF) grammars resemble the structure of Hierarchical Task Networks (HTN). This has been used in (Erol, Hendler, and Nau 1996) to show the expressiveness of planning formalisms. Then, there have been some attempts to represent HTNs as CF grammars or equivalent formalisms (Nederhof, Shieber, and Satta 2003) but as demonstrated in (Höller et al. 2014), the languages defined by HTN planning problems (with partial-order, preconditions and effects) lie somewhere between CF and context-sensitive (CS) languages. In (Geib 2016), the author presents an approach with a similar intention with the help of *Combinatory Categorial Grammars* (CCGs), which are part of a category lying between CF and CS grammars, the *mildly context-sensitive grammars*. The author proposes a single model for both plan recognition and planning and he also proposes a planning algorithm based on CCGs. However, it appears that this modeling process is counter-intuitive as it requires a *lexicalization* (the hierarchical structure is contained in the terminal symbols) while decomposition models are more natural in planning. Also, it is not yet sure that this formalism and its planning technique can produce the full range of HTN plans.

Though there is intrinsic similarity between task decomposition in HTNs and symbol rewriting in grammars, there does not exist any approach showing that a full HTN model can be represented using a formal grammar. The major problem is that actions obtained from different high-level tasks may interleave in the plan to satisfy causal links (see the example in Figure 1, where high-level tasks for transferring two containers interleave to share common mover actions). The hierarchical structure of the derivation tree cannot solely describe such interleaving. Therefore, we propose using attribute grammars with the timeline constraint to model HTNs (and STRIPS and procedural domain models), where the grammar describes the hierarchical structure while the timeline constraint ensures the correct ordering of actions satisfying the causal links.

Attribute grammars have been introduced by Knuth (Knuth 1968) to add semantics to CF grammars, making them able to express CS languages. The idea is to add a set of attributes (variables) to each symbol (terminal or nonterminal) and attribute computation rules, modifying the values of attributes depending on the values of other attributes, to each production rule of the grammar. These grammars were originally used for the design of compilers but they are now used for other applications (solving knapsack problems for example (Cleary and O'Neill 2005)). Recently, attribute grammars have been used to model *nested workflows* (Barták 2016), a process-modeling framework that have common points with hierarchical planning. With this approach, it was possible to derive a verification algorithm from the classical *reduction* techniques used in CF grammars (Barták and Dvořák 2017). This algorithm verifies that nested workflows are internally consistent (e.g. every branch of a workflow can be used in a valid process), that is, the workflow has been well modeled. We are not aware of any similar method for planning domain models.

In this paper, we extend attribute grammars with a set attribute and a global timeline constraint and we show that this formalism can represent STRIPS planning, HTN planning (with or without task insertion), and also procedural planning domain models. This is the first time when such a conversion has been done for a full HTN model with action interleaving and hence attribute grammars may serve as a truly unifying framework for various planning domain models. The results are presented for classical sequential planning with propositional state representation, but they can be naturally extended to temporal planning with resources, state variables, and action costs.

Attribute Grammars

An attribute grammar is basically a CF grammar, where the symbols are annotated by sets of attributes. In the original formulation (Knuth 1968), the values of synthesized attributes were calculated from the values of inherited attributes using semantic rules, which were assignment statements. We use a later generalization, where the attributes are connected by constraints – relations between the attributes in the sense of constraint satisfaction problems (CSPs). Briefly speaking, the attribute grammar generates words consisting of terminal symbols, exactly as the context-free grammar does, but additionally, the values are assigned to all attributes such that all the constraints are satisfied.

Formally, an attribute grammar is defined as a tuple $G = (\Sigma, N, \mathcal{P}, S, A, C)$ where Σ is an alphabet — a finite set of terminal symbols, N is a finite set of non-terminal symbols with S as the start symbol, $\mathcal{P}$ is a set of rewriting (production) rules (see below), A associates each grammar symbol $X \in \Sigma \cup N$ with a set of attributes, and $\mathcal{C}$ associates each production $R \in \mathcal{P}$ with a set of constraints over the attributes of symbols used in the rule. $G = (\Sigma, N, \mathcal{P}, S)$ is a classical CF grammar with production rules having the form $X \to w$ where $X \in N$ is a non-terminal symbol and $w \in (\Sigma \cup N)^*$ is a finite sequence of terminal and non-terminal symbols. In the following, we will write the constraints associated with a rule inside brackets $[\,]$. If A are the attributes for symbol X, we will write $X(\vec{A})$. The production rule $X(\vec{A}) \to u[c]$ is used to rewrite a word w with constraints C to word w' with constraints C', denoted as $(w, C) \Rightarrow (w', C')$, if and only if $w = u_1.X(\vec{B}).u_2$, $C' = C \cup \{\vec{A} = \vec{B}\} \cup c$, and C' is a consistent CSP over the variables from attributes of w', where $w' = u_1.u.u_2$. Then the language generated by an attribute grammar G is:

$$L(G) = \{w\sigma \mid (S, \emptyset) \Rightarrow^* (w, C), w \in \Sigma^*, \sigma \text{ solves } C\}$$

where σ is an instantiation of attributes (substitution of values to variables) satisfying the constraints C and $w\sigma$ means applying substitution σ to w, i.e. a word w where the attributes are substituted by values defined in σ. The CSP C is obtained by collecting the constraints from the rules used in the derivation of the word w.

To demonstrate the capabilities of attribute grammars, let us start with a classical context-free grammar for the language $a^i b^j c^k$ (capital letters denote non-terminal symbols, terminals are denoted by lower-case letters):

$$S \to A.B.C \quad (1)$$
$$A \to a|a.A \quad (2)$$
$$B \to b|b.B \quad (3)$$
$$C \to c|c.C \quad (4)$$

After adding attributes and constraints between them, the attribute grammar can describe the well-known context-sensitive language $a^i b^i c^i$:

$$S(n) \to A(n_a).B(n_b).C(n_c) \quad [n = n_a = n_b = n_c] \quad (5)$$
$$A(n) \to a \quad [n = 1] \quad (6)$$
$$A(n) \to a.A(m) \quad [n = m + 1] \quad (7)$$
$$B(n) \to b \quad [n = 1] \quad (8)$$
$$B(n) \to b.B(m) \quad [n = m + 1] \quad (9)$$
$$C(n) \to c \quad [n = 1] \quad (10)$$
$$C(n) \to c.C(m) \quad [n = m + 1] \quad (11)$$

Assume now, that we want to index each terminal symbol (via its attribute) by a unique number between 1 and the total number of symbols in the word and we want the grammar to generate all such words, such as $a_1 b_2 c_3$, $a_1 b_3 c_2$, $a_2 b_1 c_3$ etc. We propose to extend further the attributes by allowing a set as a value of the attribute and by using constraints over these sets. The idea of the grammar is based on collecting the indexes in a set attribute, defining the domain of indexes using a single `dom` constraint, and using the global constraint `allDiff` over this set (Régin 1994). The following grammar generates the requested language (the set attributes are denoted by the capital letters):

$$S(n, I) \to A(n_a, I_a).B(n_b, I_b).C(n_c, I_c) \quad [n = n_a = n_b = n_c, I = I_a \cup I_b \cup I_c, \texttt{dom}(I, 1, 3n), \texttt{allDiff}(I)] \quad (12)$$
$$A(n, I) \to a(i) \quad [n = 1, I = \{i\}] \quad (13)$$
$$A(n, I) \to a(i).A(m, I') \quad [n = m + 1, I = I' \cup \{i\}] \quad (14)$$
$$B(n, I) \to b(i) \quad [n = 1, I = \{i\}] \quad (15)$$
$$B(n, I) \to b(i).B(m, I') \quad [n = m + 1, I = I' \cup \{i\}] \quad (16)$$
$$C(n, I) \to c(i) \quad [n = 1, I = \{i\}] \quad (17)$$
$$C(n, I) \to c(i).C(m, I') \quad [n = m + 1, I = I' \cup \{i\}] \quad (18)$$

Note that when implementing the word generator for such grammars, one may exploit so called open (dynamic) global constraints, where the set of constrained variables can extend (and shrink) during problem solving (Barták 1999). Hence the constraint `allDiff` can be posted to the constraint store immediately, when the rule (12) is applied, rather than waiting until all the indexes I are collected.

Planning and Timeline Constraint

In this paper we will describe formal domain models for classical sequential planning that deals with sequences of actions transferring the world from a given initial state to a state satisfying certain goal condition. World states are modeled as sets of propositions that are true in those states and actions are changing validity of certain propositions.

Formally, let P be a set of propositions, then a state $S \subseteq P$ is a set of propositions that are true in that state (every other proposition is false). Each action a is described by four sets of propositions $(B_a^+, B_a^-, A_a^+, A_a^-)$, where $B_a^+, B_a^-, A_a^+, A_a^- \subseteq P, B_a^+ \cap B_a^- = \emptyset, A_a^+ \cap A_a^- = \emptyset$. Sets B_a^+ and B_a^- describe positive and negative preconditions of action a, that is, propositions that must be true and false right before the action a. Action a is applicable to state S iff $B_a^+ \subseteq S \wedge B_a^- \cap S = \emptyset$. Sets $A_a^+$and A_a^- describe positive and negative effects of action a, that is, propositions that will become true and false in the state right after executing the action a. If an action a is applicable to state S then the state right after the action a will be $\gamma(S, a) = (S \backslash A_a^-) \cup A_a^+$. If an action a is not applicable to state S then $\gamma(S, a)$ is undefined.

The classical planning problem consists of a set of actions A, a set of propositions S_0 called an initial state, and disjoint sets of goal propositions G^+ and G^- describing the propositions required to be true and false in the goal state. A solution to the planning problem is a sequence of actions $a_1, a_2, \ldots, a_n$ such that $S = \gamma(...\gamma(\gamma(S_0, a_1), a_2), ..., a_n)$ and $G^+ \subseteq S \wedge G^- \cap S = \emptyset$. This sequence of actions is called a *plan*.

In our formal models, we will separate generation of actions (using grammar rules) and their sequencing to get valid plans. These valid sequences will be enforced by a *timeline constraint* defined over events imposed by the actions. Assume that action a is an action at position k in the plan. This action imposes the following set of events

$$\begin{aligned}\mathsf{events}(a,k) = \{b^+(k,p) \mid p \in B_a^+\} \cup \{b^-(k,p) \mid p \in B_a^-\} \\ \cup \{a^+(k,p) \mid p \in A_a^+\} \cup \{a^-(k,p) \mid p \in A_a^-\}.\end{aligned}$$

We call b^+ and b^- *before-events* and they correspond to action preconditions, while a^+ and a^- are *after-events* describing action effects. If the position of action in the plan is unknown then k is a variable. The timeline constraint ensures that instantiation of these time variables will order the actions to get a valid plan. We will first describe the semantics of the timeline constraint over events with the same proposition.

Let TL_p be a multiset[1] of events related to proposition p. The constraint $\texttt{Timeline}(TL_p)$ is consistent if and only if it is possible to linearly order the events in TL_p such that they describe a correct evolution of the proposition p. Formally, let $n = |TL_p|$ be the number of events in TL_p. A solution of the timeline constraint is a linear ordering of events, represented by the mapping of events to the set $\{1, \ldots, n\}$, and instantiation of time indexes of events satisfying the following constraints. Let e_i and e_{i+1} be two events such that e_i is right before e_{i+1} in the linear order mentioned above. Then only the following transitions between the events are allowed:

e_i	e_{i+1}	constraints
$a^x(k,p)$	$a^y(l,p)$	$x, y \in \{+,-\}, k < l$
$b^x(k,p)$	$a^y(l,p)$	$x, y \in \{+,-\}, k \leq l$
$a^x(k,p)$	$b^x(l,p)$	$x \in \{+,-\}, k < l$
$b^x(k,p)$	$b^x(l,p)$	$x \in \{+,-\}, k \leq l$

Briefly speaking, the first two rows of the above table say that an after-event can follow any other event (but two after-events cannot happen at the same time as it is not possible to set the value of the proposition two times at the same time). The last two rows say that a before-event must follow events of the same "value", either the requested value of the proposition was set by the previous after-event or it was already verified by the previous before-event so it has been set even earlier. Notice that the above transition constraints require events to be ordered by their time indexes.

If TL is a multiset of events related to propositions from the set P then the constraint $\texttt{Timeline}(TL)$ is consistent iff $\forall p \in P : \texttt{Timeline}(TL \downarrow p)$ is consistent, where $TL \downarrow p$ are all events from TL related to the proposition p.

Assume a multiset of n actions A ($|A| = n$), initial state S_0 and goal conditions G^+ and G^-. We can describe the problem of deciding if the actions in A can be ordered to get a valid plan for this planning problem using a timeline constraint over the following set of events:

$$\begin{aligned}TL(A, S_0, G^+, G^-) = &\mathsf{InitEvents}(S_0) \cup \\ &\mathsf{GoalEvents}(G^+, G^-, |A|) \\ &\cup \bigcup_{a \in A} \mathsf{events}(a, i_a)\} \qquad (19)\end{aligned}$$

$$\begin{aligned}\mathsf{InitEvents}(S_0) = &\{a^+(0,p) \mid p \in S_0\} \cup \\ &\{a^-(0,p) \mid p \notin S_0\} \qquad (20)\end{aligned}$$

$$\begin{aligned}\mathsf{GoalEvents}(G^+, G^-, n) = &\{b^+(n+1,p) \mid p \in G^+\} \cup \\ &\{b^-(n+1,p) \mid p \in G^-\} \qquad (21)\end{aligned}$$

Notice that we use variables i_a to describe the yet unknown position of action a in the plan. The domain of these variables is $\{1, \ldots, n\}$ and we assume that different actions are at different positions, which is enforced by the constraint $\texttt{allDiff}(\{i_a | a \in A\})$.

Proposition 1. *The plan with actions A ($|A| = n$) for the initial state S_0 and goal conditions G^+ and G^- exists if and only if the following CSP has a solution:*

$$\begin{aligned}&\texttt{Timeline}(TL(A, S_0, G^+, G^-)) \wedge \\ &\quad \texttt{dom}(\{i_a | a \in A\}, 1, n) \wedge \\ &\quad \texttt{allDiff}(\{i_a | a \in A\})\end{aligned}$$

Proof. (sketch) If there is a plan then actions in A are linearly ordered and the `dom` and `allDiff` constraints are satisfied. Vice versa, if these constraints are satisfied then they define a linear order of actions.

[1] We allow the same event to appear more times in the set to model events originated from task decomposition in HTN models.

Now, if we have a valid plan, it is easy to verify that events can be linearly ordered to satisfy the timeline constraint for each proposition. The first event will always be the after-event defined by the initial state and then the action events will be ordered increasingly by the time indexes i_a of the actions. If action uses some proposition both in its precondition and effect then the corresponding before-event will be ordered right before the corresponding after-event. As the plan reaches the goal, the before-events defined by the goal conditions can be placed at the ends of corresponding timelines while satisfying the transition constraints in the timelines.

Vice versa, if we have a linear sequence of actions such that their corresponding events are consistently ordered using the timeline constraint, one can verify that action preconditions are satisfied (for each before-event there is some after-event of the same value that is placed earlier in the sequence and no action in between them destroys this causal link). The goal events must be placed at the ends of corresponding timelines due to their time indexes and using the similar argument as for action preconditions, these goal events must be satisfied by some previous after-events of the same value. □

Running example

In the next section, we will show how to translate planning domain models into attribute grammars. We will use a simplified Dock-Worker Robots (DWR) domain (Ghallab, Nau, and Traverso 2004) to give particular examples of grammar rules. The goal in DWR is to transfer containers from their start locations to destinations. For that, the container can be loaded onto a robot that can itself move between adjacent locations. This domain consists of four primitive actions, that are used to build a STRIPS model, and five compound tasks for the HTN model. The $\mathrm{no\text{-}op}$ action is used in the HTN model when the robot is already in the requested location ($\mathrm{Movefake}$). There is one method for each task, except for the $\mathrm{Move\text{-}rob}$ task that has two methods (Figure 1). Notice also that plans for tasks $\mathrm{Tranfer1}$ may interleave when locations are shared between containers.

Planning Domain Modeling using Attribute Grammars

Attribute grammars have already been proposed as a modeling framework for workflows (scheduling problems) (Barták 2016). In this paper we will demonstrate that they can also be used to describe planning domain models. We will present the models for classical propositional representation of planning domains introduced in the previous section.

The core idea of models is that the attribute grammar generates the actions (represented by the terminal symbols) in the plan, though the order of terminals in the word is not relevant, the time indexes will be used to determine the proper action order in the plan. The `Timeline`, `allDiff`, and `dom` constraints expressed over the indexes will model the order of actions and causal relations. This way it is possible to describe feasible plans fully by the means of attribute grammars without the need of any external mechanism (beyond the constraint solver). As far as we know, this is the first time when a pure formal grammar formalism is shown to describe various planning domain modeling formalisms completely.

STRIPS Domain Models

A STRIPS-style planning domain defines a regular language (Erol, Hendler, and Nau 1994), which is recognizable by a finite-state automaton (Kleene's Theorem). Hence a natural representation for STRIPS planning with attribute grammars is using a single non-terminal symbol T that generates actions using a simple rewriting rule $T(S) \rightarrow a.T(S')$ and that disappears when a goal state is reached. The evolution of the state is put into attributes. When generating a sentence, before choosing one of the rules (one of the actions), action preconditions must be valid in the current state S, which can be verified by the rule constraints. Similarly the change of the state from S to S' can be described using a state transition constraint.

In this paper, we do not use this representation because it implies that *forward planning* is the only applicable planning technique. In our framework, we will rather use the attribute grammar to generate actions in the plan and the `Timeline` constraint to sequence them. The planner can then exploit forward, backward, or partial-order planning approaches to generate the plans. Moreover, this model can later be integrated with our HTN model to include task insertion.

Formally, STRIPS planning domains can be represented using an attribute grammar with two non-terminal symbols S and T_{gen}. The top-level rule of the grammar is:

$$\begin{aligned} S(S_0, G^+, G^-) \rightarrow & T_{gen}(I, TL') \\ & [n = |I|, \mathtt{dom}(I, 1, n), \mathtt{allDiff}(I), \\ & TL = TL' \cup \mathsf{InitEvents}(S_0) \\ & \quad \cup \mathsf{GoalEvents}(G^+, G^-, n), \\ & \mathtt{Timeline}(TL)] \end{aligned} \tag{22}$$

For each action $a_k \in A$, we introduce the following rule:

$$\begin{aligned} T_{gen}(I, TL) \rightarrow & a_k(i).T_{gen}(I', TL') \\ & [I = I' \cup \{i\}, TL = TL' \cup \mathsf{events}(a_k, i)] \end{aligned} \tag{23}$$

Finally, the following rule will stop generating actions:

$$T_{gen}(I, TL) \rightarrow \lambda \; [I = \emptyset, TL = \emptyset] \tag{24}$$

The described grammar, let us call it a *STRIPS grammar*, generates words corresponding to valid plans. Note however, that the order of actions in the word does not necessarily correspond to the order of actions in the plan, which is determined by the `Timeline` constraint. The order of actions in the word corresponds to the order in which the planner introduced them. Hence, for a single plan there might be several words generated by the grammar.

Proposition 2. *Let S_0 be an initial state and G^+ and G^- be the goal conditions. Then for each valid plan for the sequential planning problem defined by S_0, G^+, and G^- there exists a word generated by the corresponding STRIPS grammar from the symbol $S(S_0, G^+, G^-)$ such that the word*

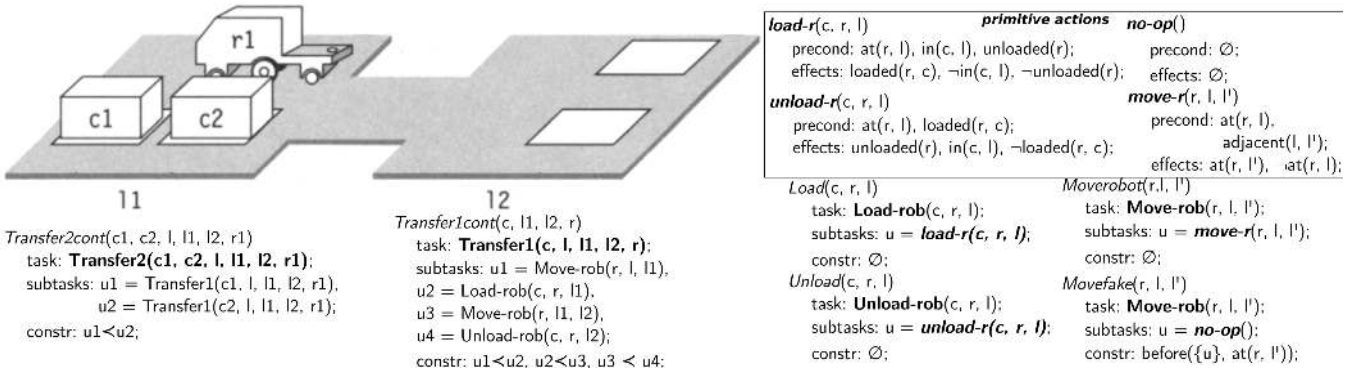


Figure 1: A DWR planning domain (taken from (Ghallab, Nau, and Traverso 2004) and modified); HTN methods (in plain italics) describe how compound tasks (in bold) are decomposed into subtasks until primitive tasks (in bold italics) are obtained.

contains exactly the actions from the plan in the same order. Vice versa, each word generated by the STRIPS grammar from the symbol $S(S_0, G^+, G^-)$ describes a valid plan for the sequential planning problem defined by S_0, G^+, and G^-.

Proof. If there is a valid plan, that is, a sequence of actions, the grammar rules (23)-(24) can generate the same sequence of actions. The grammar rule (22), that initiates generation of the word, adds the events corresponding to the initial state and goal conditions to the timeline and posts the `Timeline`, `dom`, and `allDiff` constraints. According to Proposition 1 these constraints are consistent and hence the grammar generates the described word.

Vice versa, if a terminal word is generated by the STRIPS grammar then the grammar rule (22) is used first so the above constraints are introduced. As they are satisfied by the generated word and according to Proposition 1 the word corresponds to a plan that is obtained by ordering the actions based on their time indexes. □

Example. Here is one example rule for the grounded action $\text{move-r}(r, l, l')$ from the DWR domain. The action is represented as a terminal symbol $\text{move-r}_{r,l,l'}$.

$$T_{gen}(I, TL) \rightarrow \text{move-r}_{r,l,l'}(i).T_{gen}(I', TL') \quad (25)$$
$$[I = I' \cup \{i\},$$
$$TL = TL' \cup \mathsf{events}(\text{move-r}_{r,l,l'}, i)]$$

where

$$\mathsf{events}(\text{move-r}_{r,l,l'}, i) = \{b^+(i, \text{at}(r, l)), \quad (26)$$
$$b^+(i, \text{adjacent}(l, l')),$$
$$a^+(i, \text{at}(r, l')), a^-(i, \text{at}(r, l))\}$$

Hierarchical Task Networks Domain Models

Hierarchical Task Networks (HTN) planning (Erol, Hendler, and Nau 1996) is a formalism in which problems are structured in a hierarchical way. While there are only actions (called also *primitive tasks*) in a classical planning problem, HTN planning introduces abstract *compound* tasks that are referring to hierarchies of other compound tasks and primitive tasks. The compound task decomposes to a *task network*, which is a pair (T, ψ), where T is a finite set of tasks, and ψ is a set of constraints that a plan must satisfy to be valid. The constraints may be of two types:

- ordering constraints between tasks from T,
- *before* and *after* constraints that are generalizations, respectively, of preconditions and effects for tasks; for example $\text{before}(Q, p)$, where $Q \subseteq T$ means that proposition p must be true right before the first task from Q and similarly $\text{after}(Q, p)$ means that p must be true right after the last task from Q.

For a single compound task, there may be several methods to realize it, that is, to decompose it to a task network. Formally, a *method* is a pair (ct, tn), where ct is a compound task, and tn is a task network. One of the decision points in HTN planning consists in choosing which method to apply.

In an *HTN planning problem* we are given an initial task network tn_0, that plays the role of a goal (the tasks to be solved by decomposing them to actions), and an initial state S_0. To solve the problem one needs to decompose all tasks in tn_0 by available methods until primitive tasks (actions) are obtained and these primitive tasks need to be linearly ordered to form a plan that is applicable to the initial state S_0. Also, all the ordering, *before*, and *after* constraints that are introduced during task decompositions must be satisfied.

HTN planning naturally resembles the idea of CF grammars, but because of the other constraints a more powerful mechanism is necessary (Höller et al. 2014). Now we detail how to translate an HTN problem into an attribute grammar with set attributes. We will use one starting non-terminal symbol S, non-terminal symbols TN_i for task networks, and non-terminal symbols T_i for tasks (both compound and primitive). Actions will correspond to terminal symbols.

For the top-level symbol of the grammar, the initial rule generates the initial task network tn_0 and adds the events corresponding to the initial state S_0. This is also where the global constraints including the `Timeline` constraint are in-

troduced:

$$S(S_0) \rightarrow TN_0(I, TL') \quad (27)$$
$$[n = |I|, \texttt{dom}(I, 1, n), \texttt{allDiff}(I),$$
$$TL = TL' \cup \mathsf{InitEvents}(S_0),$$
$$\texttt{Timeline}(TL)]$$

The task network $tn_0 = (\{t_1, \ldots, t_m\}, \psi)$ is described using a single rewriting rule that generates the tasks $\{t_1, \ldots, t_m\}$ and introduces the constraints ψ; the same approach can be used for other task networks in the HTN model:

$$TN_0(I, TL) \rightarrow T_1(I_1, TL_1) \ldots T_m(I_m, TL_m)\ [C] \quad (28)$$

where the constraints C are in the following form:

$$\begin{aligned} C:\ & I = I_1 \cup \ldots \cup I_m, \\ & TL = TL_1 \cup \ldots \cup TL_m \cup \\ & \quad \cup \{b^+(\min \cup_{t_j \in Q} I_j, p) \mid \text{before}(Q, p) \in \psi\} \\ & \quad \cup \{b^-(\min \cup_{t_j \in Q} I_j, p) \mid \text{before}(Q, \neg p) \in \psi\} \\ & \quad \cup \{b^+(1 + \max \cup_{t_j \in Q} I_j, p) \mid \text{after}(Q, p) \in \psi\} \\ & \quad \cup \{b^-(1 + \max \cup_{t_j \in Q} I_j, p) \mid \text{after}(Q, \neg p) \in \psi\}, \\ & \max(I_i) < \min(I_j)\ \ \forall (t_i \prec t_j) \in \psi \end{aligned} \quad (29)$$

Notice how the events, obtained from the before and after constraints, are defined. First, these events are modeled using the before-events because they define which proposition and when should be true in the plan. In particular, differently from action effects, the after event is not making the proposition true (or false), it only says that the proposition should be true (or false) right after all the actions generated from the corresponding tasks Q.

Now, for each method (t_k, tn_k) for the compound task t_k we introduce the following rewriting rule (note that there might be more such rules for the non-terminal symbol T_k, if there are more decomposition methods):

$$T_k(I, TL) \rightarrow TN_k(I, TL)\ [] \quad (30)$$

and for each primitive task T_k that corresponds to action a_k we introduce the following rule:

$$T_k(I, TL) \rightarrow a_k(i)\ [I = \{i\}, TL = \mathsf{events}(a_k, i)] \quad (31)$$

Example. We show how to translate two compound tasks from Figure 1 into rules of the attribute grammar. Rules 32-34 translate methods Movereal and Movefake decomposing the task Move-rob.

$$\text{Move-rob}_{r,l,l'}(I, TL) \rightarrow \text{Movereal}_{r,l,l'}(I, TL) \mid \text{Movefake}_{r,l,l'}(I, TL) \quad (32)$$

$$\text{Movereal}_{r,l,l'}(I, TL) \rightarrow \text{move-r}_{r,l,l'}(i)\ [I = \{i\}, TL = \mathsf{events}(\text{move-r}_{r,l,l'}, i)] \quad (33)$$

$$\text{Movefake}_{r,l,l'}(I, TL) \rightarrow \text{no-op}(i)\ [I = \{i\}, TL = \{b^+(i, at(r, l'))\}] \quad (34)$$

Notice that the no-op action is used to get index i in the plan so the event is placed in the right place of the timeline.

The task Transfer1 is decomposed using a method Transfer1cont which can be expressed using the following rules 35 and 36 :

$$\text{Transfer1}_{c,l,l1,l2,r}(I, TL) \rightarrow \text{Transfer1cont}_{c,l,l1,l2,r}(I, TL) \quad (35)$$

$$\begin{aligned} \text{Transfer1cont}_{\substack{c,l,\\ l1,l2,r}}(I, TL) \rightarrow\ & \text{Move-rob}_{r,l,l1}(I_1, TL_1). \\ & \text{Load-rob}_{c,r,l1}(I_2, TL_2). \\ & \text{Move-rob}_{r,l1,l2}(I_3, TL_3). \\ & \text{Unload-rob}_{c,r,l2}(I_4, TL_4) \\ & [C] \end{aligned} \quad (36)$$

where

$$\begin{aligned} C = \{ & TL = TL_1 \cup TL_2 \cup TL_3 \cup TL_4, \\ & I = I_1 \cup I_2 \cup I_3 \cup I_4,\ \max(I_1) < \min(I_2), \\ & \max(I_2) < \min(I_3),\ \max(I_3) < \min(I_4)\} \end{aligned}$$

Proposition 3. *Let S_0 be an initial state and tn_0 be the initial task network. Then for each valid plan obtained by the decomposition of tasks in tn_0 there exists a word generated by the corresponding HTN grammar from the symbol $S(S_0)$ such that the word contains exactly the actions from the plan and their time indexes correspond to their order in the plan. Vice versa, each word generated by the HTN grammar from the symbol $S(S_0)$ describes a valid plan obtained by the decomposition of tasks in tn_0 and applicable to state S_0*

Proof. (sketch) If there is a valid plan obtained by the decomposition of tn_0 then the rewriting rules (27), (28), (30) (31) can generate the actions in the plan. As the plan is applicable to S_0 then according to Proposition 1 (where $G^+ = G^- = \emptyset$) the `Timeline` constraint over the action events and events for the initial state is consistent. The plan must also satisfy the ordering, before, and after constraints from task networks used in the decompositions and hence the constraints (29) are also consistent. Together, the terminal word obtained by the rewriting rules mimicking the decomposition process is generated by the HTN grammar.

A terminal word generated by the above HTN grammar describes actions that can be obtained by decomposing the initial task network tn_0. Moreover, as the initial state and action preconditions and effects are modeled using the events that satisfy the `Timeline` constraint, the actions, when ordered by their time indexes, form a valid plan applicable to the initial state S_0 (Proposition 1). Finally the ordering, before, and after constraints from task networks used in the decomposition are also satisfied as they were added when the corresponding rewriting rule (28) was applied. □

HTNs with Task Insertion A pure HTN model requires all actions to be generated by task decompositions only. This may be limiting if the description of tasks, in particular methods how to achieve the tasks, is incomplete. For such a problem *HTN with Task Insertion* (TIHTN) was suggested as a method where missing gaps in the task decomposition

can be filled by arbitrary actions (Kambhampati, Mali, and Srivastava 1998). To demonstrate versatility of the proposed modeling framework, let us show how to model TIHTN using attribute grammars. The only necessary modification is adding the rewriting rules (23)-(24) to the HTN grammar and modifying the rule (27) as follows:

$$\begin{aligned} S(S_0) \rightarrow & TN_0(I_1, TL_1).T_{gen}(I_2, TL_2) \\ & [I = I_1 \cup I_2, n = |I|, \\ & TL = TL_1 \cup TL_2 \cup \mathsf{InitEvents}(S_0), \\ & \mathtt{dom}(I, 1, n), \mathtt{allDiff}(I), \\ & \mathtt{Timeline}(TL)] \end{aligned} \tag{37}$$

Procedural Domain Models

Domain Control Knowledge (DCK) allows to express domain-specific constraints over the definition of a valid plan. When it is well-crafted, it can considerably reduce the amount of search needed to get a solution. Procedural DCK (PDCK) (Baier, Fritz, and McIlraith 2007) is closer to a programming language than DCK as it includes constructs from imperative programming languages such as loops and conditionals. PDCK is closely related to HTN as it is action-centric and provides plan templates or skeletons of plans, constraining the appearance and order of actions in a plan. For the domain described on Figure 1, a naive PDCK program could be *"while there are some containers in l1, choose any container and load it onto the robot, then move to l2, and finally unload all the containers"* (Figure 2).

Name of the atomic program	Formal expression
Translation to AG	
Empty program	*nil*
$T_i(\mathrm{j}, \mathrm{j'}, TL, \vec{V}) \rightarrow \lambda\ [\mathrm{j'} = \mathrm{j}, TL = \emptyset]$	
Single action $a \in A$	a
$T_i(\mathrm{j}, \mathrm{j'}, TL, \vec{V}) \rightarrow a\ [\mathrm{j'} = \mathrm{j}+1, TL = \mathsf{events}(a, \mathrm{j})]$	
Choosing any action	any
$T_i(\mathrm{j}, \mathrm{j'}, TL, \vec{V}) \rightarrow a\ [\mathrm{j'} = \mathrm{j}+1, TL = \mathsf{events}(a, \mathrm{j})]$ $\forall a \in A$	
Test action	$\theta?$
$T_i(\mathrm{j}, \mathrm{j'}, TL, \vec{V}) \rightarrow \lambda\ [\mathrm{j'} = \mathrm{j}, TL = \{b^+(\mathrm{j}, p) \vert p \in \theta\} \cup \{b^-(\mathrm{j}, p) \vert \neg p \in \theta\}]$	
Negated test action	$\neg\theta?$
$T_i(\mathrm{j}, \mathrm{j'}, TL, \vec{V}) \rightarrow \lambda\ [\mathrm{j'} = \mathrm{j}, TL = \{b^-(\mathrm{j}, p)\}]\ \forall p \in \theta$ $T_i(\mathrm{j}, \mathrm{j'}, TL, \vec{V}) \rightarrow \lambda\ [\mathrm{j'} = \mathrm{j}, TL = \{b^+(\mathrm{j}, p)\}]\ \forall \neg p \in \theta$	
Non-deterministic choice of variable v_k *of type* t	$\pi(\mathrm{v_k}, \mathrm{t})$
$T_i(\mathrm{j}, \mathrm{j'}, TL, \vec{V}) \rightarrow \lambda\ [\mathrm{j'} = \mathrm{j}, TL = \emptyset, \vec{V_k} = v_k \in t]$	

Table 1: Translation of atomic programs to attribute grammars rules.

Name of the composed program	Formal expression
Translation to AG	
Sequence of programs	$(\sigma_{i_1}; \sigma_{i_2})$
$T_i(\mathrm{j}, \mathrm{j'}, TL, \vec{V}) \rightarrow T_{i_1}(\mathrm{j}, \mathrm{j}_1, TL_1, \vec{V}).T_{i_2}(\mathrm{j}_1, \mathrm{j'}, TL_2, \vec{V})$ $[TL = TL_1 \cup TL_2]$	
Conditional sentence	**if** $\theta_{i_1}?$ **then** σ_{i_2} **else** σ_{i_3}
$T_i(\mathrm{j}, \mathrm{j'}, TL, \vec{V}) \rightarrow T_{i_1}(\mathrm{j}, \mathrm{j}_1, TL_1, \vec{V}).T_{i_2}(\mathrm{j}_1, \mathrm{j'}, TL_2, \vec{V})$ $[TL = TL_1 \cup TL_2]$ $T_i(\mathrm{j}, \mathrm{j'}, TL, \vec{V}) \rightarrow T_{\bar{i_1}}(\mathrm{j}, \mathrm{j}_1, TL_1, \vec{V}).T_{i_3}(\mathrm{j}_1, \mathrm{j'}, TL_2, \vec{V})$ $[TL = TL_1 \cup TL_2]$, where $\theta_{\bar{i_1}} = \neg\theta_{i_1}$	
While-loop	**while** $\theta_{i_1}?$ **do** σ_{i_2}
$T_i(\mathrm{j}, \mathrm{j'}, TL, \vec{V}) \rightarrow T_{i_1}(\mathrm{j}, \mathrm{j}_1, TL_1, \vec{V}).$ $T_{i_2}(\mathrm{j}_1, \mathrm{j}_2, TL_2, \vec{V}).T_i(\mathrm{j}_2, \mathrm{j'}, TL_3, \vec{V})$ $[TL = TL_1 \cup TL_2 \cup TL_3]$ $T_i(\mathrm{j}, \mathrm{j'}, TL, \vec{V}) \rightarrow T_{\bar{i_1}}(\mathrm{j}, \mathrm{j'}, TL, \vec{V})$, where $\theta_{\bar{i_1}} = \neg\theta_{i_1}$	
Non-deterministic choice between two programs	$(\sigma_{i_1} \vert \sigma_{i_2})$
$T_i(\mathrm{j}, \mathrm{j'}, TL, \vec{V}) \rightarrow T_{i_1}(\mathrm{j}, \mathrm{j'}, TL, \vec{V})$ $T_i(\mathrm{j}, \mathrm{j'}, TL, \vec{V}) \rightarrow T_{i_2}(\mathrm{j}, \mathrm{j'}, TL, \vec{V})$	
Non-deterministic iteration	$\sigma^*_{i_1}$
$T_i(\mathrm{j}, \mathrm{j'}, TL, \vec{V}) \rightarrow T_{i_1}(\mathrm{j}, \mathrm{j}_1, TL, \vec{V}).T_i(\mathrm{j}_1, \mathrm{j'}, TL, \vec{V})$ $T_i(\mathrm{j}, \mathrm{j'}, TL, \vec{V}) \rightarrow \lambda\ [\mathrm{j'} = \mathrm{j}]$	

Table 2: Translation of composed programs to attribute grammars rules.

PDCK naturally provides a more informed and directed search. In (Baier, Fritz, and McIlraith 2007), the PDCK language is based on a robot programming language. The authors showed that it is possible to translate that language to a non-deterministic finite automaton with λ-moves (NFA-λ). Hence it is straightforward to model plans accepted by that automaton using attribute grammars. However, we will show that the original program can be directly encoded in an attribute grammar to preserve its compactness. The PDCK program itself can be recognized by a context-free grammar so we take the derivation tree of a specific PDCK program and define an attribute grammar based on it with attributes modeling the evolution of state variables, the indexing of actions, and the global variables in the program.

As there is no interleaving of actions in plans generated by the PDCK program, we may omit the I attribute collecting the action indexes and substitute it by direct indexing of actions. Two integer attributes will be used for that purpose. The first attribute defines the index of the first action generated from the non-terminal symbol and the second attribute defines the index of an action that will be right after the ac-

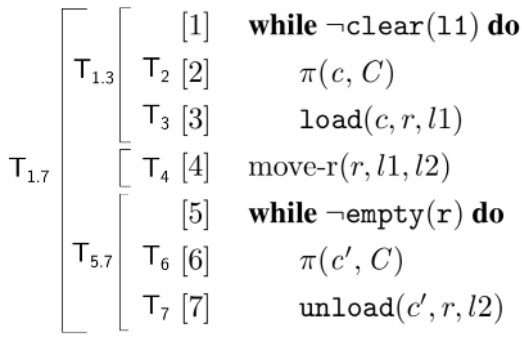


Figure 2: A piece of PDCK program and its derivation tree. The predicate clear(l) is true when there is no container in location l. The predicate empty(r) is true when there is no container on robot r. Each number between brackets refers to one programming construct.

tions generated from a given non-terminal. We preserve the timeline attribute TL, but note that events will only be added to the end of the timeline as the program proceeds.

There are two PDCK programming constructs that need extra explanation. One is a test action $\theta?$, which is logical formula over the state variables. We assume that it is a conjunction of propositions from the current state so it can be represented as a set of propositions (both positive and negative) that are assumed to be true (false) in the current state, similarly to a precondition of an action. Note that a disjunction can be modeled using a non-deterministic choice between two programs. This is useful, if we need to model a negation of the test action $\neg\theta?$, for example, in the compound programs with while-loops and if-then-else constructs. The second PDCK construct is a non-deterministic choice of a variable $\pi(\mathrm{v}, \mathrm{t})$. Basically, it means that the program assigns a value to some global variable that can be used later in the code. To model such variables, we introduce a new attribute $\vec{V}$, which is a vector of these variables. This way, we can easily access these global variables and use them, for example, in constraints.

In summary, each programming construct σ_i is represented as a non-terminal symbol T_i. If this construct is an atomic program then the non-terminal symbol is rewritten to a terminal symbol, for example to an action (Table 1). If the programming construct represents a compound program then the non-terminal symbol is rewritten to other non-terminal symbols representing the sub-programs (Table 2). The top-level rule of the grammar is very close to the STRIPS modeling rule (22):

$$\begin{aligned} S(S_0, G^+, G^-) \rightarrow & T_{\text{Root}}(1, n, TL', \vec{V}) \\ & TL = TL' \cup \mathsf{InitEvents}(S_0) \\ & \quad \cup \mathsf{GoalEvents}(G^+, G^-, n-1), \\ & \mathtt{Timeline}(TL)] \end{aligned} \tag{38}$$

where the only difference is that T_{Root} is a pointer to the translation of the top-level program block.

Example. Table 3 shows an attribute grammar obtained from the PDCK program in Figure 2. In this example, T_{Root} is $T_{1,7}$ and the non-terminal symbol $T_{i,j}$ represents the PDCK code at lines $i - j$. We used some classical simplification of rewriting rules. Namely, the rules

$$O \rightarrow P.Q\ [c_1], P \rightarrow R.S\ [c_2]$$

can be merged to a single rule

$$O \rightarrow R.S.Q\ [c_1 \cup c_2].$$

Also, we used the attribute for global variables only in non-terminal symbols, where that variable is used.

$$\begin{aligned} T_{1,7}(\mathrm{j}, \mathrm{j'}, TL) \rightarrow & T_{1,3}(\mathrm{j}, \mathrm{j}_1, TL_1). \\ & T_4(\mathrm{j}_1, \mathrm{j}_2, TL_2). \\ & T_{5,7}(\mathrm{j}_2, \mathrm{j'}, TL_3) \\ & [TL = TL_1 \cup TL_2 \cup TL_3] \end{aligned} \tag{39}$$

$$\begin{aligned} T_{1,3}(\mathrm{j}, \mathrm{j'}, TL) \rightarrow & T_2(\mathrm{j}, \mathrm{j}_1, TL_1, V). \\ & T_3(\mathrm{j}_1, \mathrm{j}_2, TL_2, V). \\ & T_{1,3}(\mathrm{j}_2, \mathrm{j'}, TL_3) \\ & [TL = TL_1 \cup TL_2 \cup TL_3 \cup \\ & \{b^-(\mathrm{j}, \mathtt{clear(l1)})\}] \end{aligned} \tag{40}$$

$$\begin{aligned} T_{1,3}(\mathrm{j}, \mathrm{j'}, TL) \rightarrow & \lambda \\ & [TL = \{b^+(\mathrm{j}, \mathtt{clear(l1)})\}, \mathrm{j'} = \mathrm{j}] \end{aligned} \tag{41}$$

$$T_2(\mathrm{j}, \mathrm{j'}, TL, V) \rightarrow \lambda\ [V \in C, \mathrm{j'} = \mathrm{j}, TL = \emptyset] \tag{42}$$

$$\begin{aligned} T_3(\mathrm{j}, \mathrm{j'}, TL, V) \rightarrow & \text{load-r}_{V,r,l1} \\ & [TL = \mathsf{events}(\text{load-r}_{V,r,l1}, \mathrm{j}), \mathrm{j'} = \mathrm{j}+1] \end{aligned} \tag{43}$$

$$\begin{aligned} T_4(\mathrm{j}, \mathrm{j'}, TL) \rightarrow & \text{move-r}_{r,l1,l2} \\ & [TL = \mathsf{events}(\text{move-r}_{r,l1,l2}, \mathrm{j}), \\ & \mathrm{j'} = \mathrm{j}+1] \end{aligned} \tag{44}$$

$$\begin{aligned} T_{5,7}(\mathrm{j}, \mathrm{j'}, TL) \rightarrow & T_6(\mathrm{j}, \mathrm{j}_1, TL_1, V). \\ & T_7(\mathrm{j}_1, \mathrm{j}_2, TL_2, V). \\ & T_{5,7}(\mathrm{j}_2, \mathrm{j'}, TL_3) \\ & [TL = TL_1 \cup TL_2 \cup TL_3 \cup \\ & \{b^-(\mathrm{j}, \mathtt{empty(r)})\}] \end{aligned} \tag{45}$$

$$\begin{aligned} T_{5,7}(\mathrm{j}, \mathrm{j'}, TL) \rightarrow & \lambda \\ & [TL = \{b^+(\mathrm{j}, \mathtt{empty(r)})\}, \mathrm{j'} = \mathrm{j}] \end{aligned} \tag{46}$$

$$T_6(\mathrm{j}, \mathrm{j'}, TL, V) \rightarrow \lambda\ [V \in C, \mathrm{j'} = \mathrm{j}, TL = \emptyset] \tag{47}$$

$$\begin{aligned} T_7(\mathrm{j}, \mathrm{j'}, TL, V) \rightarrow & \text{unload-r}_{V,r,l1} \\ & [TL = \mathsf{events}(\text{unload-r}_{V,r,l1}, \mathrm{j}), \mathrm{j'} = \mathrm{j}+1] \end{aligned} \tag{48}$$

Table 3: Grammar for PDCK of Figure 2.

Conclusions

In this paper we suggested attribute grammars as a unifying modeling framework for STRIPS, HTN, and procedural domain models and we showed that any such domain model can be represented fully by an attribute grammar with the

`Timeline` constraint. This is the first time, when such a conversion of planning domain models to formal grammars has been presented. The existing approaches of using formal grammars in planning used intuitive similarity between the concepts but never provided a complete translation of a planning domain model to a grammar (in particular, interleaving of actions in HTN has never been satisfactory presented in existing works).

To simplify notation, we used grounded representation, but the ideas can be naturally extended to lifted domain models with object variables (in addition to time indexes). For the same reasons, we focused on sequential planning with the propositional state model, but the presented concept can be used for temporal planning with state variables too, just the `Timeline` constraint will be modified to handle general temporal relations, as this is for example the case in (Verfaillie, Pralet, and Lemaître 2010).

The presented modeling framework opens new directions of research exploiting relations between automated planning and formal languages. Formal grammars are already used for plan/goal recognition (Geib and Steedman 2007) and there are initial attempts to use them for planning (Geib 2016) and for domain model verification (Barták 2016). The grammar model also simplifies development of algorithms verifying if the plan complies with the hierarchical domain model, which is a problem not yet solved for HTN domain models. Automated acquisition/learning of grammar rules is another promising area for future research. Incremental design of models is supported starting with an initially flat STRIPS-like model (STRIPS-grammar) that can be later extended with high-level tasks modeling typical sub-plans. This gives a complete model as the resulting grammar corresponds to HTN with Task Insertion, where the HTN part of the model can be incomplete.

Acknowledgments

Research is supported by the Czech Science Foundation under the project P103-15-19877S.

References

Baier, J. A.; Fritz, C.; and McIlraith, S. A. 2007. Exploiting Procedural Domain Control Knowledge in State-of-the-Art Planners. In Boddy, M. S.; Fox, M.; and Thiébaux, S., eds., *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007*, 26–33. AAAI.

Barták, R. 1999. Dynamic constraint models for planning and scheduling problems. In *New Trends in Contraints, Joint ERCIM/Compulog Net Workshop, Paphos, Cyprus, October 25-27, 1999, Selected Papers*, 237–255.

Barták, R., and Dvořák, T. 2017. On verification of workflow and planning domain models using attribute grammars. In *Proceedings of the 15th Mexican International Conference on Artificial Intelligence*. Springer.

Barták, R. 2016. Using Attribute Grammars to Model Nested Workflows with Extra Constraints. In Freivalds, R. M.; Engels, G.; and Catania, B., eds., *SOFSEM 2016: Theory and Practice of Computer Science*, number 9587 in Lecture Notes in Computer Science. Springer Berlin Heidelberg. 171–182. DOI: 10.1007/978-3-662-49192-8_14.

Cleary, R., and O'Neill, M. 2005. An Attribute Grammar Decoder for the 01 MultiConstrained Knapsack Problem. In Raidl, G. R., and Gottlieb, J., eds., *Evolutionary Computation in Combinatorial Optimization, 5th European Conference, EvoCOP 2005, Lausanne, Switzerland, March 30 - April 1, 2005, Proceedings*, volume 3448 of *Lecture Notes in Computer Science*, 34–45. Springer.

Erol, K.; Hendler, J.; and Nau, D. S. 1994. Semantics for Hierarchical Task-network Planning. Technical report, University of Maryland at College Park, College Park, MD, USA.

Erol, K.; Hendler, J. A.; and Nau, D. S. 1996. Complexity Results for HTN Planning. *Ann. Math. Artif. Intell.* 18(1):69–93.

Geib, C. W., and Steedman, M. 2007. On Natural Language Processing and Plan Recognition. In Veloso, M. M., ed., *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 1612–1617.

Geib, C. 2016. Lexicalized reasoning about actions. *Advances in Cognitive Systems* 4:187–206.

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

Höller, D.; Behnke, G.; Bercher, P.; and Biundo, S. 2014. Language Classification of Hierarchical Planning Problems. In Schaub, T.; Friedrich, G.; and O'Sullivan, B., eds., *ECAI 2014 - 21st European Conference on Artificial Intelligence, 18-22 August 2014, Prague, Czech Republic - Including Prestigious Applications of Intelligent Systems (PAIS 2014)*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, 447–452. IOS Press.

Kambhampati, S.; Mali, A. D.; and Srivastava, B. 1998. Hybrid planning for partially hierarchical domains. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA.*, 882–888.

Knuth, D. E. 1968. Semantics of Context-Free Languages. *Mathematical Systems Theory* 2(2):127–145.

Koller, A., and Petrick, R. P. A. 2011. Experiences with planning for natural language generation. *Computational Intelligence* 27(1):23–40.

Nederhof, M.-J.; Shieber, S.; and Satta, G. 2003. Partially ordered multiset context-free grammars and ID/LP parsing. *Proceedings of the Eighth International Workshop on Parsing Technologies* 171–182.

Régin, J. 1994. A filtering algorithm for constraints of difference in csps. In *Proceedings of the 12th National Conference on Artificial Intelligence, Seattle, WA, USA, July 31 - August 4, 1994, Volume 1.*, 362–367.

Verfaillie, G.; Pralet, C.; and Lemaître, M. 2010. How to model planning and scheduling problems using constraint networks on timelines. *Knowledge Eng. Review* 25(3):319–336.

Method Composition through Operator Pattern Identification

Maurício Cecílio Magnaguagno, Felipe Meneguzzi
School of Computer Science (FACIN)
Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre - RS, Brazil
mauricio.magnaguagno@acad.pucrs.br
felipe.meneguzzi@pucrs.br

Abstract

Classical planning is a computationally expensive task, especially when tackling real world problems. To overcome such limitations, most realistic applications of planning rely on domain knowledge configured by a domain expert, such as the hierarchy of tasks and methods used by Hierarchical Task Network (HTN) planning. Thus, the efficiency of HTN approaches relies heavily on human-driven domain design. In this paper, we aim to address this limitation by developing an approach to generate useful methods based on classical domains. Our work does not require annotations in the classical planning operators or training examples, and instead, relies solely on operator descriptions to identify task patterns and the sub-problems related to each pattern. We propose the use of methods that solve common sub-problems to obtain HTN methods automatically.

Introduction

Finding a sequence of actions to reach a desired state may be considered a trivial problem for a human, but when faced with many possible actions, the task of finding such sequence become a complex problem. Classical planners have no global view of which actions should be prioritized to efficiently decide strategies to explore the space of possible plans during search. Domain knowledge may be used to provide such a global view in terms of macros (sequences of actions) (Botea et al. 2005) or hierarchical constructions that require decomposition to obtain a plan (Nau et al. 1999). Hierarchical planners use domain knowledge to be able to solve problems orders of magnitude faster than classical planners. However, hierarchical planners rely exclusively on the availability and quality of such domain knowledge to be able to solve problems correctly and fast. Such domain knowledge must be carefully designed by a domain expert so that all valid decompositions may be used to obtain a solution, or risk having a planner that either fails to find a solution, or finds a sub-optimal or even invalid solution. This knowledge requires a domain expert to consider generalized solutions that solve common sub-problems within the domain, often involving recursive decomposition. As a consequence, describing a domain and its possible decompositions is time consuming, especially when a domain expert needs to test the general solution in order to avoid infinite recursions and ensure that the planner eventually returns a solution when one exists for all valid states.

By analyzing a number of existing Hierarchical Task Network (HTN) domains, we notice that similar domains rely on similar methods to solve analogous sub-problems, such as recursively moving until a certain destination is reached. While some descriptions, such as the Action Notation Modeling Language (ANML), allow a debugger to describe such dependency mechanisms (Smith, Frank, and Cushing 2008) to make domain knowledge explicit, we aim to generate domain knowledge using only the information available in a classic domain description. Descriptions of different domains may use different predicates, but share the same plan construction patterns to solve common sub-problems. Generalizing such construction patterns that appear in the domain operators make it possible to automate the process and obtain task descriptions to solve each sub-problem. Without generalizing such patterns one could solve using the method of Erol *et al.* (Erol, Nau, and Subrahmanian 1995), which tries operators using a recursive method, a brute-force solution. Automating the process of task description based on the classical domain description can save development time while taking advantage of hierarchical planners potential speed-up. In this paper, we address the need for domain knowledge for HTN planners using an approach that automatically generates HTN methods from a classical planning domain. The methods generated by our approach not only improves the efficiency of the resulting HTN planner compared to the conversion from PDDL to HTN described by Erol *et al.* (Erol, Nau, and Subrahmanian 1995), but are also readable by a human, allowing manual refinements to further improve the efficiency of the generated HTN domain knowledge. The resulting approach can then be applied to both allow HTN planners to be used efficiently to solve classical planning domains with minimal or no expert-knowledge, or enhance hybrid planners such as GoDel (Shivashankar et al. 2013) and obviate the need for human-designed domain knowledge in order to achieve solution speed ups.

Background

Classical Planning

Automated planning is concerned with finding a set of actions that reaches a goal state from a initial state of the world.

States encode properties of the objects in the world at a particular time. Specifically, the goal is represented by a state formula indicating which properties must be true at the end of a plan. In order to achieve the goal state the operators defined in the domain are used as a transition function that modified states in terms of preconditions and effects. During the planning process the preconditions of actions are used to test which actions are applicable at each state. If applicable, the action effects can be applied, creating a new state. Preconditions are satisfied when a formula (usually a conjunction of predicates) is valid in the state the action is being applied. The effects contain positive and negative sets that add or remove object properties from the state, respectively. Once we reach a state that satisfies the goal, the sequence of actions taken, starting at the initial state, is the plan or solution (Nebel 2000) to a planning problem. This formalism was standardized in the *Planning Domain Definition Language* (PDDL)(McDermott et al. 1998) to allow direct comparisons of efficiency between planning algorithms using a uniform input file format.

Hierarchical planning

Hierarchical planning shifts the focus from goal states to tasks to be solved in order to exploit human knowledge about problem decomposition using an hierarchy of methods and operators as the planning domain. This hierarchy is created by non-primitive tasks, which uses methods with preconditions and sub-tasks to decompose according to context. The sub-tasks are also decomposed until only primitive-tasks mapping to operators remain, which results in the plan itself. The goal is implicitly achieved by the plan obtained from the decomposition process. If no decomposition is possible the task is considered unachievable. Unlike classical planning, hierarchical planning only considers what appears during the decomposition process to solve the problem. With domain knowledge the domain description is more complex than the classical planning description, as recursive loops can be described. Recursive loops occur when a non-primitive task is decomposed by a method that contains itself in the sub-tasks, this may be the desired behavior when we need to apply the same set of operators several times until a stop condition is met, for example, to walk until a destination is reached.

Each task is represented by a name, a set of parameters, a set of preconditions, and a set of sub-tasks. Once a non-primitive task is decomposed, the sub-tasks replace the current task in the task network. Some tasks may have an empty set of sub-tasks, representing no further decomposition. Backtracking is required for flexibility, as branches may fail during decomposition. Backtracking is costly, but in some cases can be avoided by look-ahead preconditions that check an entire branch of the domain. In some domains it is possible to guide the search directly to a solution or failure. This planning formalism is capable of describing the same domains as STRIPS with a built-in heuristic function tailored to the domain and expert preferences (Lekavỳ and Návrat 2007), with all the methods required beforehand, which consumes project time to consider every single case.

SHOP (Nau et al. 1999) is one of the best known implementations of HTN planning algorithms. The successors of SHOP, SHOP2 and JSHOP2 (Ilghami and Nau 2003), share most of their algorithm using a more complex decision about which task to decompose at any step in order to support interleaved tasks. Since no standard description exists for HTN, we opted to use the same description used by SHOP2 and JSHOP2. SHOP2 (Nau et al. 2001), for example, supports unordered task decomposition, a feature that separates this planner from its predecessor, SHOP (Nau et al. 1999). JSHOP2 description follows a simplified version of the LISP style adopted by PDDL, without labels for every part of the operator or method. The operator represents the same as the classical operator, an action that can take place in this domain. The operators have a name, a set of parameters and three sets. The first set represents the preconditions, the second set the negative-effects with what is going to be false in the next state, while the final set represents the positive-effects with what is going to be true at the next state. The methods have a name, a set of parameters, a set of preconditions, and instead of effects they have a set of sub-tasks to be performed. Methods can also be decomposed in different ways and have an optional label for each case. The problem contains two sets, the first represents the initial state and the second a list of tasks to be performed. Instead of just interpreting the domain and problem, the description can be compiled to achieve better results with static structures.

Identifying operator patterns

To generate HTN methods based on classical planning descriptions, one must first identify common patterns of operator usage in order to obtain generic methods that could be used in planning domains. Such common patterns are based on how predicates are used by operators. The use of predicates as source of information has already been explored by Pattison and Long (2010) in goal recognition. Here, predicates were partitioned into groups to help differentiate which predicates are more likely to be a goal. Instead of the partitions defined by Pattison and Long, we partition predicates based on their mutability, as shown in Algorithm 1. Predicates that appear only in the initial state (but not in any action) are considered *irrelevant*, they make no difference in the action application. However predicates that appear in any action precondition but never as an effect define *constant* relations of the domain. Predicates that appear in the effects of any action represent what is possible to change, the *mutable* relations of the domain. Knowing which predicates are constant helps to prune impossible values for the variables at any state, while mutable predicates can indicate which actions can take place once (adding or removing a feature from the current state) or several times in the same plan. Based on the previous observation of how actions with certain predicate types are used we defined a set operator patterns that once matched against an action can relate to a method that solves its related sub-problem. The following subsections explore such operator patterns.

Swap pattern

Some planning instances require the application of an action repeatedly, the only difference being the values of the

Algorithm 1 Classification of predicates into irrelevant, constant or mutable

```
1: function CLASSIFY_PREDICATES(predicates, operators)
2:     ptypes ← Table
3:     pre ← PRECONDITIONS(operators)
4:     eff ← EFFECTS(operators)
5:     for each p ∈ predicates do
6:         if p ∈ eff
7:             ptypes[p] ← mutable
8:         else if p ∈ pre
9:             ptypes[p] ← constant
10:        else
11:            ptypes[p] ← irrelevant
12:    return ptypes
```

parameters. Such actions usually revolve around swapping the truth value of two instances of the same predicate, such as moving from one place to another affects the predicate *at* in the example action from Listing 1. Once the swaps achieve the predicate required by another action precondition or goal predicate the process can stop. This pattern commonly appears in discretized scenarios where an agent swaps its current position among adjacent and free coordinates in an N-dimensional space, where N is the arity of the position predicate. The position is the predicate that is going to be swapped, while the adjacency is a constraint that implies this operator may be executed several times in order to traverse a discretized space.

This operator pattern is related to the path-finding subproblem and was already identified and exploited by other planners to speed up search. Hybrid STAN (Fox and Long 2001) is one such planner; it uses a path planner and a resource manager to solve sub-problems with specialized solvers. We classify operators involved in the swap pattern using Algorithm 2. Swap operators contain a constraint in the preconditions, otherwise the swap would have no restrictions requiring only one operator to solve the sub-problem, and a predicate that is modified from the preconditions to the effects using the same parameters as in the constraint. Since several operators may include the swap pattern over the same predicate, they can be merged into a single method with different constraints, such as a climb operator that changes the agent position like a move operator, but only

```
(:action move :parameters (
    ?bot - robot
    ?source ?destination - hallway)
  :precondition (and
    (at ?bot ?source)
    (not (at ?bot ?destination))
    (connected ?source ?destination) )
  :effect (and
    (not (at ?bot ?source))
    (at ?bot ?destination) ) )
```

Listing 1: Move operator with swap pattern in PDDL.

```
(:method (swap_predicate ?object ?goal)
  base
  ( (predicate ?object ?goal) )
  ()
  using_operator
  (
    (constraint ?current ?intermed)
    (swap_predicate ?object ?current)
    (not (predicate ?object ?goal))
    (not (visited_predicate ?object
        ?intermed))
  )
  (
    (!operator ?object ?current ?intermed)
    (!!visit_predicate ?object ?current)
    (swap_predicate ?object ?goal)
    (!!unvisit_predicate ?object ?current)
  ) )
```

Listing 2: Methods for *swap* operator pattern using JSHOP description.

if there is a wall nearby the current position. Swap identification can also be used to infer that an agent or object will never be at two different configurations in the same state, proving that no plan exists for such goal state. Listing 1 shows the *move* operator with the swap pattern in PDDL, in which a *robot* moves from *source* to *destination* when *source* and *destination* are *connected*. Listing 2 shows two decompositions for the generic *swap_predicate*. The first decomposition acts as the base of the recursion, with the predicate with goal values. The second decomposition applies one more step, marks the current position as visited to avoid loops, recursively decomposes *swap_predicate* and unvisits the previously visited positions to be able to reuse such positions later if needed. Visit and unvisit are internal operations done by bookkeeping operators, prefixed by *!!* in JSHOP.

Algorithm 2 Classification of swap operators

```
1: function CLASSIFY_SWAP(operators, ptypes)
2:     swaps ← Table
3:     for each op ∈ operators do
4:         constraints ← CONST_POS_PRECOND(op, ptypes)
5:         pre+ ← MUTABLE_POS_PRECOND(op, ptypes)
6:         pre− ← MUTABLE_NEG_PRECOND(op, ptypes)
7:         eff+ ← ADD_EFFECTS(op)
8:         eff− ← DEL_EFFECTS(op)
9:         for each pre ∈ (pre+ ∩ eff−) do
10:            pre2 ← NAME(pre) ∈ eff+
11:            if pre2
12:                cparam ← PARAM(pre) △ PARAM(pre2)
13:                for c ∈ constraints do
14:                    if c ⊆ cparam
15:                        swaps[op] ← ⟨pre, constraint⟩
16:                        break
17:    return swaps
```

```
(:action report :parameters (
    ?bot - robot
    ?location - hallway
    ?beacon - beacon)
  :precondition (and
    (at ?bot ?location)
    (in ?beacon ?location)
    (not (reported ?bot ?beacon)) )
  :effect (reported ?bot ?beacon) )
```

Listing 3: Report operator with dependency pattern with move in PDDL.

Dependency pattern

In the same way some planning instances require the effects of an action to make another action applicable, fulfilling the preconditions. Such precondition turns the first action effects into a dependency for the second action preconditions to be satisfied and the action applied. The operators are classified as dependency using Algorithm 3. Each pair of operators is compared to find a match between effects and preconditions of operators that have not already been classified as swap operators. Listing 3 shows the *report* operator with the dependency pattern in PDDL, which requires a *robot* to be *at* the same *location* of a *beacon* to report its status. To achieve the *at* precondition there is a dependency with the *move* operator.

Algorithm 3 Classification of dependency operators

```
1: function CLASSIFY_DEPENDENCY(operators, ptypes, swaps)
2:   dependencies ← Table
3:   for each op ∈ operators do
4:     pre+ ← POS_PRECOND(op, ptypes, mutable)
5:     pre− ← NEG_PRECOND(op, ptypes, mutable)
6:     eff+ ← ADD_EFFECTS(op)
7:     eff− ← DEL_EFFECTS(op)
8:     for each op2 ∈ operators do
9:       swap_op ← swaps[op]
10:      swap_op2 ← swaps[op2]
11:      if swap_op ≠ ∅ and swap_op2 ≠ ∅ and
         NAME(swap_op) = NAME(swap_op2)
12:        continue
13:      pre2+ ← POS_PRECOND(op2)
14:      pre2− ← NEG_PRECOND(op2)
15:      if op = op2 or (pre2+ ⊆ eff+ and pre2− ⊆ eff−)
16:        continue
17:      eff2+ ← ADD_EFFECTS(op2)
18:      for each pre ∈ pre+ do
19:        if not NAME(pre) ∈ eff2+
20:          continue
21:        if dependencies[op] = ∅
22:          dependencies[op] ← Set
23:        APPEND(dependencies[op], ⟨op2, pre⟩)
24:  return dependencies
```

```
(:method (dependency_first_before_second
    ?param)
  goal_satisfied
  ( (goal_predicate) )
  () )
(:method (dependency_first_before_second
    ?param)
  satisfied
  ( (predicate ?param) )
  ( (!second ?param) ) )
(:method (dependency_first_before_second
    ?param)
  unsatisfied
  ( (not (predicate ?param)) )
  ( (!first ?param) (!second ?param) ) )
```

Listing 4: Methods for *dependency* operator pattern using JSHOP description.

There are three possible cases we need to handle with this pattern. In the first case the goal predicate is already satisfied and no action takes place. In the second case the preconditions of the second action are already fulfilled and the action can be applied immediately. In the third case the precondition of the second action require the first action applied before the second action. This operator pattern is common in many planning domains (e.g. able to pickup item after achieving a certain position with move), as several distinct actions may be required to fulfill a sequence of preconditions to achieve a goal predicate. Actions that already matched the swap pattern are not tested against the dependency pattern, otherwise such actions would be classified with a dependency of themselves. We consider the dependency pattern a specialization of the swap pattern between different actions. Listing 4 shows the three cases defined for the operator dependency pattern using a JSHOP-style description.

Free-variable pattern

Some goal state predicates may leave some variables open while mapping to a task. For example a position that must be occupied requires an, as yet unspecified, agent, but no agent is bound to this variable while other similar tasks require an agent at a place, which maps to the same task with a specific agent. Methods can propagate bound variables to be used by operators or other method as tasks are decomposed. Before propagated, free-variables must be bound. In order to unify variables such as the agent we create a new method with the single purpose of unification according to the constant preconditions of the related operator, acting as typed parameters of PDDL. Therefore we add a new level to the hierarchy with a method that simply unifies and propagates to the next level with all variables bound. Listing 5 shows a possible scenario where *op1* requires 3 terms to be applied. Terms *t1* and *t2* are known based on information from goal state, while *t3* is left to be decided at run-time. It is possible to apply directly the original method in cases where the three terms are known. We do not merge both methods in one method to explicitly say that we are looking for the

```
(:method (three_terms ?t1 ?t2 ?t3)
  apply_op_with_three_terms
  ( (precond1 ?t1 ?t2) (precond2 ?t2 ?t3) )
  ( (!op1 ?t1 ?t2 ?t3) ) )
(:method (unify_three_terms ?t1 ?t2)
  unify_term_t3
  ( (precond2 ?t2 ?t3) )
  ( (three_terms ?t1 ?t2 ?t3) ) )
```

Listing 5: Methods for *free-variable* operator pattern using JSHOP description.

value of *t3*.

With the predicates classified and each operator related to the operator patterns previously defined, we need to relate the goal state with a set of tasks that achieve each part of the goal using the methods generated for each pattern.

Composing methods and tasks

With the methods obtained based on the patterns previously described we need to relate such methods with the original goal predicates to create tasks that reach a valid goal state. In order to determine which method to apply we select the operators that are more closely related to the goal, the ones that contain a goal predicate in their effect list. We can use such goal operators to identify which sub-problems we are trying to solve based on the patterns available. If the goal operator matches the dependency pattern, we use methods for each case: a goal that is already satisfied decomposes to an empty set of subtasks; a satisfied precondition decomposes to the operator that achieves the goal predicate; otherwise it decomposes to the dependency operator that achieves a precondition required by another operator that achieves the goal predicate. If the goal operator matches the swap pattern, we create a specific swap method for the predicate being swapped containing all operators that match the swap pattern with this predicate.

Some methods may depend on other methods, to solve this we also apply a dependency check between methods to decompose the required methods first. This happens when the first operator of a dependency also requires a dependency or an operator used in a dependency is also classified as a swap. A new level in the hierarchy is created as such operators are replaced by their respective methods. With the methods selected, we only need to add tasks with the corresponding objects based on goal predicates. If such information is not available in the goal predicates some variables remain free to be unified at run-time. To avoid repeating costly unifications, we use the free-variable pattern to unify variables as high in the hierarchy as possible. At the end of the process we have the original set of operators incremented with some bookkeeping operators, the set of generated methods, and the set of tasks replacing the goal predicates. Algorithm 4 shows such steps to convert a classical description to an HTN description using our approach. The operator patterns identified generate methods that are currently added independently of usage, as some methods may hint the domain expert about the relation among operators even when certain methods are not connected to the rest of the hierarchy.

Algorithm 4 Convert goals to tasks

```
 1: function GOALS_TO_TASKS(domain, problem)
 2:   op ← OPERATORS(domain)
 3:   pred ← PREDICATES(domain)
 4:   goals ← GOALS(problem)
 5:   tasks ← Set
 6:   met ← Set
 7:   ptypes ← CLASSIFY_PREDICATES(op, pred)
 8:   swaps ← CLASSIFY_SWAP(op, ptypes)
 9:   dependencies ← CLASSIFY_DEPENDENCY(op,
      ptypes, swaps)
10:   goal_op ← Array
11:   for each o ∈ op do
12:     for each goal ∈ goals do
13:       if goal ∈ EFFECTS(o)
14:         APPEND(goal_op, ⟨goal, o⟩)
15:   ADD_SWAP_METHODS(swaps, op, met, ptypes)
16:   ADD_DEPEND_METHODS(swaps, dependencies, op,
      met, ptypes)
17:   goal_tasks ← Array
18:   for each m ∈ met do
19:     for each d ∈ DECOMPOSITION(m) do
20:       for each ⟨goal, o⟩ ∈ goal_op do
21:         if o ∈ SUBTASKS(d)
22:           met2 ← UNIFY_VARIABLES(m, o)
23:           APPEND(goal_tasks, ⟨goal, met2⟩)
24:   INJECT_METHOD_DEPENDENCIES(swaps, met)
25:   for each ⟨goal, met2⟩ ∈ goal_tasks do
26:     if free-variable ∈ met2
27:       APPEND(tasks, UNIFY_METHOD(met2))
28:     else
29:       APPEND(tasks, met2)
30:   return ⟨op, met, tasks⟩
```

Use case: Rescue Robot domain

In order to illustrate how our operator patterns can be applied to a concrete domain, we use the rescue robot domain [1] as a use case, as several patterns are identified in the operator set. This domain has a small operator set and can be represented by a 2D map, which means we can explore it deeply without complex constructions. The map contains rooms and hallways as locations where the rescue robot and beacons may be located. The robot must be in the same hallway or room of a beacon to report the status. The set of operators include:

- **Enter** a room connected to the current hallway.
- **Exit** the current room to a connected hallway.
- **Move** from the current hallway to a connect hallway.
- **Report** status of beacon in the current room or hallway.

[1] The rescue robot domain was created by Kartik Talamadupula and Subbarao Kambhampati.

We use our operator patterns to infer how such operators are related to the problem. The operators **Enter**, **Exit** and **Move** swap the predicate *at*. They all require source and destination to be connected locations, which matches our constraint requirement. **Move** creates a dependency for **Enter**, as **Enter** creates a dependency for **Exit**, but since they are already considered *swap* operators we can prioritize *swap* over *dependency* patterns. Swap *at* method may be needed zero or more times to match the destination *at*, which behaves as shown in Figure 1, without the invisible visit/unvisit operators to control which source and intermediate positions where visited, and the equivalent JSHOP output in Listing 6.

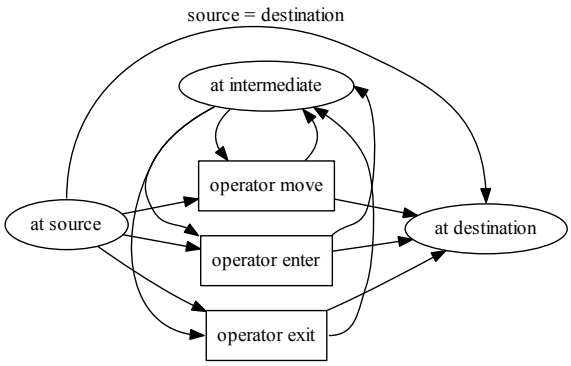


Figure 1: Source, intermediate and destinations are reachable locations the robot may visit using move, enter or exit operations.

Only one operator remains unclassified in this domain, **Report**, which has a precondition *at*. Instead of creating a dependency for each swap operator we can inject the dependency between such methods and make clear that **Report** have a dependency with the swap *at* method previously created, generating a new method. Now this higher level task can be used to report each beacon in the problem, the possible branches the dependency method may take are shown in Figure 2 and the equivalent JSHOP output in Listing 7.

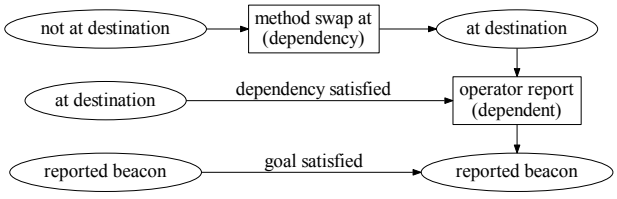


Figure 2: The destination must be reached by the swap at method before any non-reported beacon is reported.

```
(:method (swap_at_until_at ?bot ?source)
  base
  ( (at ?bot ?source) )
  () )
(:method (swap_at_until_at ?bot ?source)
  using_enter
  (
    (at ?bot ?current)
    (connected ?current ?intermediate)
    (not (at ?bot ?source))
    (not (visited_at ?bot ?intermediate))
  )
  (
    (!enter ?bot ?current ?intermediate)
    (!!visit_at ?bot ?current)
    (swap_at_until_at ?bot ?source)
    (!!unvisit_at ?bot ?current) ) )
(:method (swap_at_until_at ?bot ?source)
  using_exit
  (...)
  (
    (!exit ?bot ?current ?intermediate)
    (!!visit_at ?bot ?current)
    (swap_at_until_at ?bot ?source)
    (!!unvisit_at ?bot ?current) ) )
(:method (swap_at_until_at ?bot ?source)
  using_move
  (...)
  (
    (!move ?bot ?current ?intermediate)
    (!!visit_at ?bot ?current)
    (swap_at_until_at ?bot ?source)
    (!!unvisit_at ?bot ?current) ) )
```

Listing 6: Methods generated to solve the identified swap at pattern, one base method that decompose to an empty set of tasks when the goal location is reached by a robot and other three recursive methods that move a robot to new locations.

Implementation and Experiments

We implemented a SHOP-like (Nau et al. 1999) HTN planner in Ruby much like JSHOP2 (Ilghami and Nau 2003) that translates PDDL domains into an HTN structure in Ruby whose execution is equivalent to the HTN forward decomposition algorithm [2]. Thus, a PDDL domain is translated into an intermediate representation in Ruby upon which our algorithm operates, therefore maintaining the system independent of language and style choices, as special features from the language are downgraded to commonly supported features. This is the case for PDDL type support, typed objects are added as propositions to the initial state and typed parameters are compiled into preconditions.

Since many domains require predicates that do not match any of the templates described in this paper, we add methods to achieve such predicates using the PDDL to HTN conversion process from Erol *et al.* (Erol, Nau, and Subrahmanian 1995) as fallback, essentially "brute-forcing" a search through HTN methods to achieve a particular predicate. To

[2] Available at https://github.com/Maumagnaguagno/HyperTensioN

```
(:method
 (dependency_swap_at_until_at_before_report
     ?bot ?source ?beacon)
  goal-satisfied
  ( (reported ?bot ?beacon) )
  () )
(:method
 (dependency_swap_at_until_at_before_report
     ?bot ?source ?beacon)
  satisfied
  (
    (robot ?bot)
    (location ?source)
    (beacon ?beacon)
    (in ?beacon ?source)
    (at ?bot ?source)
  )
  ( (!report ?bot ?source ?beacon) ) )
(:method
 (dependency_swap_at_until_at_before_report
     ?bot ?source ?beacon)
  unsatisfied
  (
    ...
    (not (at ?bot ?source)) )
  (
    (swap_at_until_at ?bot ?source)
    (!report ?bot ?source ?beacon) ) )
```

Listing 7: Methods generated to solve the identified dependency between the action report and the predicate at.

avoid infinite loops in the brute-force mechanism we mark visited actions as they are applied during the recursion, and unmark them as the recursion backtracks. Since our current approach does not check for task interference and order we permute the generated tasks until the original goal state is satisfied. Such behavior can be emulated by expressive enough HTN planners using ordering constraints.

We have tested our approach with multiple domains to discover variations of the operator patterns identified. Our approach took 0.1s or less to generate HTN domain knowledge using the patterns in this paper, and thus it is very efficient in terms of the overall time it can save during search. We performed our experiments using an Intel E5500 2.8GHz CPU with 2GB of RAM running Windows. The classical planner (CP) used in the comparison was also developed in Ruby and is doing Breadth-first search with a binary-vector state representation. In a small set of problems from the Rescue Robot domain, Table 1, we can see that the patterns found were enough to greatly speed up plan search when compared with the pure Brute-Force (BF) approach. Consider the Goldminers domain at Table 2, in which two problems with grids of size 10x10 cells contain agents that must move to pick and deposit gold at certain positions. State-space planners suffer with more positions, gold and agents available, while an HTN can focus its search and solve such problems much faster. The sequences of movement actions is where HTN can focus its search in the experiments. More complex domains, such as the ones from ICAPS, still require human intervention to either complete or correct the domain with knowledge that was not inferred from the provided PDDL. Such as the Floortile domain in which an agent can move to the four cardinal directions and paint either to its north or south position, our approach fails to see that both actions are required to color the top and bottom rows of the grid, which returns failures for solvable problems. Other domains such as the Grid, requires an agent to collect keys to open doors in a labyrinth scenario to reach a goal position, which requires several journeys to move towards key, door and goal. Our solution only generates methods to fulfill a single journey, making problems with several doors unsolvable.

Table 1: Rescue Robot tested with multiple planners. Time-outs occur at 100 seconds.

Problem	CP	HTN BF	HTN Patterns + BF
pb1	0.001	0.044	0.067
pb2	0.002	11.190	0.255
pb3	0.009	Time-out	0.072
pb4	0.004	20.353	0.197
pb5	0.001	96.979	0.218
pb6	0.001	Time-out	0.132

Table 2: Goldminers tested with diverse planners using 100s as time-out.

Problem	CP	HTN BF	HTN Patterns + BF
pb1	Time-out	Time-out	6.270
pb2	Time-out	Time-out	3.668

Conclusions and Future Work

In this paper we have developed an approach to automatically generate HTN domain knowledge using a PDDL specification. Our approach relies exclusively on a number of patterns of state changes we identified as common in most planning domains, therefore dispensing with example plans.

Existing work has investigated techniques to bridge the gap between classical planning and HTN in multiple ways, however, most such work either require a dataset comprised of a number of solution plans or generated methods that are not competitive with a fast classical planner. The first approach comparable to ours is the brute-force conversion (Erol, Nau, and Subrahmanian 1995). Although this approach translates any PDDL problem into an HTN one that generates equivalent plans, the resulting domain knowledge is not competitive with current classical planners. Indeed, the translation proposed by Erol *et al.* (with some modifications to avoid being stuck doing and undoing by applying the same set of actions) is our fallback approach when neither of the patterns we found apply. The approach from Lotinac and Jonsson (Lotinac and Jonsson 2016) is the most comparable to ours and generates HTNs from invariance analysis (Lotinac and Jonsson 2016). Finally, the GoDeL (Shivashankar et al. 2013) planner is an hybrid approach that

uses methods with sub-goals and landmarks to guide search. Here, instead of trying to decompose a task, methods generate plans to achieve parts of a state-based goal, and uses a classical planner as a fallback option when methods fail or are insufficient. GoDeL's approach is able to perform better if a domain expert supplies more domain knowledge while performing as a classical planner if only classical operators are supplied.

Empirical evaluation has shown that our approach is capable of not only generating valid HTN methods for domains that relate with our operator patterns, it also generates efficient HTN method libraries that can greatly speed-up search. Nevertheless, the HTN knowledge generated for many domains does not allow an HTN planner using blind search to surpass a fast classical planner. Domains in which most of our patterns apply tended to result in better performance, whereas domains that relied on a brute force translation of a PDDL task into HTN methods did worst. Our initial approach focused on obtaining plans faster than what a classical planner or a brute-force translation could achieve, no work has been done regarding plan quality. We expected all planners to eventually return a plan when one is possible, unless some limitation forces the planner to stop, such as time or memory available. Thus, as future work, we aim to investigate mechanisms to further improve the efficiency and quality of the resulting HTN domain knowledge and its interaction with more advanced HTN planners. First, we aim to search for new patterns that could be applicable to the remaining domains, which would make it possible to use more domains to better benchmark our implementation against more planners. Second, we will evaluate the performance of the methods we generate with HTN planners that selectively choose methods for decompositions rather than performing blind search.

Acknowledgments

We acknowledge the support given by CAPES/Pro-Alertas (88887.115590/2015-01) and CNPQ within process number 305969/2016-1 under the PQ fellowship.

References

[Botea et al. 2005] Botea, A.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-FF: Improving AI planning with automatically learned macro-operators. *Journal of Artificial Intelligence Research* 24:581–621.

[Erol, Nau, and Subrahmanian 1995] Erol, K.; Nau, D. S.; and Subrahmanian, V. S. 1995. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence* 76(1-2):75–88.

[Fox and Long 2001] Fox, M., and Long, D. 2001. Hybrid STAN: Identifying and managing combinatorial optimisation sub-problems in planning. In *Proceedings of International Joint Conference on Artificial Intelligence*, 445–452. Morgan Kaufmann.

[Ilghami and Nau 2003] Ilghami, O., and Nau, D. S. 2003. A general approach to synthesize problem-specific planners. Technical report, DTIC Document.

[Lekavỳ and Návrat 2007] Lekavỳ, M., and Návrat, P. 2007. Expressivity of strips-like and htn-like planning. In *Agent and Multi-Agent Systems: Technologies and Applications*. Springer. 121–130.

[Lotinac and Jonsson 2016] Lotinac, D., and Jonsson, A. 2016. Constructing hierarchical task models using invariance analysis. In *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI-16)*.

[McDermott et al. 1998] McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL − The Planning Domain Definition Language. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems 1998 (AIPS'98)*.

[Nau et al. 1999] Nau, D.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, 968–973. Morgan Kaufmann Publishers Inc.

[Nau et al. 2001] Nau, D.; Munoz-Avila, H.; Cao, Y.; Lotem, A.; and Mitchell, S. 2001. Total-order planning with partially ordered subtasks. In *Proceedings of International Joint Conference on Artificial Intelligence*, volume 1, 425–430.

[Nebel 2000] Nebel, B. 2000. On the compilability and expressive power of propositional planning formalisms. *Journal of Artificial Intelligence Research* 12:271–315.

[Shivashankar et al. 2013] Shivashankar, V.; Alford, R.; Kuter, U.; and Nau, D. 2013. The GoDeL planning system: a more perfect union of domain-independent and hierarchical planning. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, 2380–2386. AAAI Press.

[Smith, Frank, and Cushing 2008] Smith, D. E.; Frank, J.; and Cushing, W. 2008. The ANML language. In *The ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling*.

Extracting Incomplete Planning Action Models from Unstructured Social Media Data to Support Decision Making

Lydia Manikonda[1], **Shirin Sohrabi**[2], **Kartik Talamadupula**[2], **Biplav Srivastava**[2], **Subbarao Kambhampati**[1]
[1]School of Computing, Informatics, and Decision Systems Engineering, Arizona State University
[2]IBM T. J. Watson Research Center
{lmanikonda, rao}@asu.edu, {ssohrab, krtalamad, biplavs}@us.ibm.com

Abstract

Despite increasing interest in leveraging the wealth of online social media data to support data-based decision making, much work in this direction has focused on tasks with straightforward "labeling" decisions. A much richer class of tasks can benefit from the power of sequential decision making. However, supporting such tasks requires learning some form of action or decision models from unstructured data – a problem that had not received much attention. This paper leverages and extends machine learning techniques to learn decision models (incomplete action models) for planning from unstructured social media data. We provide evaluations showing the potential of unstructured data to build incomplete planning action models, which can further be extended to build PDDL-style action models for many real-world domains. Our models can be used to support novel quantitative analysis of online behaviors that can indirectly explain the offline behaviors of social media users.

1 Introduction

There is a growing interest in exploiting the burgeoning amount of user-generated data on the Internet – especially on social media platforms – to provide data-based decision support. While the initial wave of work in this direction was limited to support single *labeling* decisions (e.g. recommendations), there is an increasing interest in supporting more complex scenarios that require planning and other forms of sequential decision making. A prime example is the category of tasks that are classified as "self-help", and which involve a number of steps and often complicated sequences of actions. Examples here include quitting smoking, losing weight, or traveling the world. A number of online groups contain a plethora of crowd-generated wisdom about appropriate courses of action that have worked for a variety of different individuals. The main problem we consider in this paper is the extraction of such information so that it can be applied towards an automated way of helping new users with similar goals. Such automated approaches need not be restricted to plan synthesis alone; they can also include a number of other sequential decision making problems including plan critiquing, plan ranking, and even merely the extraction of plan traces that can be used as input to existing model-learning methods.

Although there exists a large body of literature on planning and decision making, almost all of it assumes that the action model has been specified *a priori*. This has turned into a very pressing bottleneck for the AI planning community as a whole, where planning techniques depend very heavily on the availability of *complete and correct* models (Kambhampati 2007). One of the challenges that must be overcome to tide over this problem is to *extract* usable causal relationships from unstructured natural language data on social media. This can be a very daunting problem, since social media posts are made in unrestricted natural language, and meant for human consumption. The text from these posts can be highly nuanced and extremely arbitrary, making automated extraction of causal relationships and action models an AI-complete task.

While parsing individual posts can be arbitrarily hard, our aim is to investigate if the massive scale and redundancy of the posts might nevertheless help extract reasonable approximations of causal and action models. We hypothesize that the feasibility of this endeavor might increase if we further focus on the so-called *shallow* models (c.f. (Kambhampati 2007; Tian, Zhuo, and Kambhampati 2016)). To this end, we propose and experiment with a six-phase pipeline that leverages shallow natural language processing (NLP) techniques to extract incomplete causal relationships. We envision that these relationships can be extended to generate complete PDDL-style domain models, in the spirit of (Srivastava and Kambhampati 2005) – however, this specifically is not the main focus of this paper.

As mentioned earlier, such approximate causal models can be utilized to automatically explain (c.f. *plan explanation, plan critiquing*) the experiences shared by users on online social media towards achieving their personal goals. In order to achieve this, our proposed pipeline addresses five main tasks: (1) *extract* actions from users' posts; (2) *process* the extracted actions to reduce redundancy; (3) *build* plan traces from the extracted actions; (4) *construct* an action precedence graph from these traces; and (5) *plan* using these precedence graphs. While we focus on an end-to-end solution, mid-stream output from our pipeline—e.g. plan traces—can also be fed to existing approaches for learning action models from complete, partial, or noisy plan traces (c.f. (Yang, Wu, and Jiang 2007; Tian, Zhuo, and Kambhampati 2016)). We evaluate the plans – which are represented as shallow workflows – to demonstrate the utility of

subreddit	Main goal
(/r/stopsmoking)	How to *quit* smoking ?
(/r/C25K)	With no experience of running, how to *run* a 5K ?
(/r/weddingplanning)	How to *plan* for a wedding ?

Table 1: Subreddits and their main goals

our novel six-phase pipeline. Within the current context, we define (shallow) *workflows* as a sequence of actions where the final action in the sequence is/achieves the user's goal.

In the rest of this paper, we will describe the details of our proposed pipeline; we first focus on explaining the data in Section 2. Details of the six-phase pipeline that we implemented, including the metrics for evaluating the action models we extracted to support sequential decision problems, are presented in Section 3. Section 4 describes the evaluation methodology and the results obtained through quantitative and qualitative analyses. Section 5 presents the related work focusing on how the existing literature and the proposed solution through the pipeline are different. Section 6 concludes the paper with a discussion on future work.

2 Data

In this paper, we utilize social media data to identify the important actions which are described by the users trying to achieve a goal. Towards this goal, we consider posts from the popular social news website called "Reddit" (https://www.reddit.com/) where the registered users submit content in different forms like web urls or text posts. Along with sharing content, users can comment and vote on a given post (up or down votes) that determines the popularity or rank of a post in a given thread. The content entries on this platform are designed in a tree format where each branch of a tree represents a sub-community referred as "Subreddit". Each subreddit is categorized to a particular domain that ranges from being very general to sometimes very personal.

We used the Python Wrapper for Reddit API[1] to crawl posts and their metadata from three different subreddits shown in Table 1. For the ease of reading, we represent the subreddit '/r/stopsmoking' as *Quit Smoking*; '/r/C25K' as *Couch to 5K*; '/r/weddingplanning' as *Wed. Planning*. Note that the entire pipeline is automated and there is no manual intervention in any of the processes. Table 2 provides the relevant statistics about the raw dataset and the actions extracted by the pipeline to build an action model.

3 Pipeline

We utilize the automated planning and NLP techniques to build a six-phase pipeline (as shown in Figure 1). This pipeline utilizes the raw unstructured social media data to extract structured shallow workflows. The main contributions or the challenges addressed by this pipeline are: 1) extract the plan traces from the raw unstructured data, 2) utilize the plan traces for building an incomplete action model that are capable of generating workflows that are near optimal. Our main contribution lies in considering the unstructured social media data and building shallow models which are capable of generating plans that are near optimal.

[1]https://praw.readthedocs.io/en/latest/index.html

	Domain Name		
	Quit Smoking	**Couch to 5K**	**Wed. Planning**
# Users	787	604	969
Tot. # of traces	1598	1131	3442
Avg. trace len.	17.97	16.7	21.29
# Unique Actions (orig)	1712	1299	2666
# Unique Actions (model)	234	194	355
# Pre-actions	1499	1060	2795
	(117,6.4,16.9)	(84,5.5,13.8)	(170,7.9,22.5)
# Post-actions	1398	982	2619
	(31,6,6.5)	(29,5.1,5.6)	(37,7.4,8.2)

Table 2: Statistics of users, plan traces and actions. Numbers in bracket are max, avg and std. dev.; min=1; Unique Actions (orig) are the set of actions that are extracted from the crawled raw data; Unique Actions (model) are the set of actions obtained after generalization.

Achieving the first goal is an important contribution of this paper, as most of the existing work (e.g., (Gregory and Lindsay 2016; Tian, Zhuo, and Kambhampati 2016; Yang, Wu, and Jiang 2007; Yoon 2007)) for domain model acquisition, assume that the plan traces required are readily available. Hence, these systems may not be functional when the traces are not available. To address the first challenge we mentioned earlier, the pipeline utilizes raw unstructured social media data that is processed to remove redundancies and repetitions to extract plan traces in lifted representations. To address the second challenge, these plan traces are utilized to compute the probabilities which determine the causal relationships between actions to finally build an incomplete action model. For each of the three domains we described in Section 2, we automatically extract the important actions to build a shallow model that is used to generate workflows.

The pipeline consists of six different components which are executed sequentially. The six different components of this pipeline shown in Figure 1 consists of: (1) fragment extractor - filtering the available data or posts to find the relevant posts, given a particular goal; (2) action extractor - identifying the candidate list of action names and their parameters as well as the initial plan fragment; (3) generalizer - grouping similar action names into the same cluster; (4) trace builder - converting the posts into plan traces; (5) sequential probability learner: learning the ordering among actions; (6) model validator: validating the extracted model. More details about each of the component are explained in detail below along with a running example.

3.1 Phase-1: Fragment Extractor

The main goal of this component is to extract the fragments from the corresponding subreddit. We define fragment as the relevant post that contains information about achieving the given goal of the subreddit. An individual fragment may contain more than one action that helps achieve the goal. To do this, we first crawl the individuals who are actively participating on a subreddit associated with a given goal. We

Figure 1: Six-phase pipeline

crawl the timelines of these individuals that we assume are the sequence of actions or a workflow that helps these individuals to achieve the given goal. We define *timeline* as the set of goal-related posts shared by the same user chronologically.

Running example [relevant posts]: I spent few weeks drinking and partying. In a similar situation in the past, I take a cigarette and used to smoke pretty much non-stop. But this season I was assaulted by the triggers. Smoking in restaurants, communal areas. Many times I thought I can get a cigarette now. But those thoughts were always chased by reason and the power of conviction I have to quit smoking.

The running example is taken from the Quit Smoking domain. This example is an excerpt of a post shared by a user on Reddit whose main goal is to quit smoking. In this study, we consider each post made by a user as a plan trace. Posts made by all users on this subreddit are aggregated to build the model in latter steps.

3.2 Phase-2: Action Extractor

Each post may have more than one sentence, where each sentence may have more than one verb. For each sentence, we extract the verbs and their corresponding nouns using the Stanford part of speech tagger (Toutanova et al. 2003), a state of the art tagger with reported 97.32% accuracy. The extracted verbs are the candidate list of action names. We assume that the order of sentences in a post is indicative of the order of actions we extract from them. In the plan trace, the extracted action names from the first sentence will appear before the extracted action names from the second sentence.

Along with the action names (verbs), we also extract the action parameters (nouns) using the similar strategy and attach the most frequently co-occurring action parameter (noun) with a given action name (verb). For this pipeline, we assume that each action (verb) will have only one action parameter (noun) and two action words can have the same action parameter. For example, assume that there is an action a_i in our dataset which occurs in multiple plan traces and co-occurs with nouns n_a, n_b and n_c. Noun with the largest co-occurrence frequency with a_i is chosen to be the action parameter for a_i. In the examples provided in this paper, action parameters are attached to an action as *<action_name>_<action_parameter_name>* (or sometimes we use *<action_name>* (*<action_parameter_name>*) interchangeably). Since certain English words can be classified as multiple parts of speech tags, we make similar assumptions.

Running example: [action names]: spent_smoke drink_beer party_hard take_day smoke_day assault_trigger smoke_day thought_smoke chase_life quit_smoke

From the post made by the user obtained in phase-1, we extract all the verbs and their associated nouns. We assume that the sequentiality among actions is pre-established in the original post made by the user. This assumption sets a constraint that all the verbs extracted are ordered in the same way they occur in the post made by the Reddit user. Since the word 'smoke' can be either a noun or verb, we see the similar pattern in this extracted set of actions and their corresponding parameters.

3.3 Phase-3: Generalizer

Since we are handling unrestricted natural language text, it is normal that a same action is used to represent this action's synonyms. Across the aggregated set of posts, there might be verbs that can summarize or subsume a given verb. This motivated us to utilize hierarchical agglomerative clustering approach where low level actions are expressed in high level format. Performing this operation helps reduce the redundancy of actions.

To remove redundant actions, we utilize the agglomerative clustering approach to group semantically similar actions. When clustering the actions, only the action names are considered and their parameters are ignored. This approach (Han, Kamber, and Pei 2011) utilizes Leacock Chodorow similarity metric (*lch* for short)[2] to measure the distance between any two given actions (W_i and W_j – action words). This is one of the popular metrics utilized to compute the semantic similarity between pairs of words. The *l*ch similarity is computed as follows:

$$Sim(W_i, W_j) = Max[log2D - logDist(c_i, c_j)] \quad (1)$$

where *Dist(c_i, c_j)* is the shortest distance between concepts c_i and c_j (a concept is the general notion or abstract idea) and *D* is the maximum depth of a taxonomy.

We consider a threshold metric (or closeness metric) α to verify the quality and stop the process of agglomeration. The agglomerative clustering algorithm terminates when the closeness metric is greater than the linkage metric at any given point of time. In hierarchical clustering, there are three different types of linkage metrics – *single*, *complete* and *average*. In this paper, we utilize the *complete* linkage metric as the Clustering Quality (we refer to as *cq*) measured is higher (*cq*=8.33) compared to the other linkage metrics (single (*cq*=5.23) and average (*cq*=7.17)). The formal equation to compute the complete linkage metric is $max\{d(a,b) : a \in A, b \in B\}$ where, $d(a,b)$ is the distance metric, *A* and *B* are two separate clusters. When the algorithm terminates, semantically similar actions will be grouped into the same cluster.

Each cluster may have more than a single action that requires us to find a unique cluster representative. To do

[2] http://www.nltk.org/howto/wordnet.html

this, we utilize a popular statistic from Information Retrieval community – *Term Frequency–Inverse Document Frequency* to rank all the actions present in a cluster based on this metric. For each cluster, we choose the top-ranked action with the highest *tfidf* value to be the representative of the respective cluster. The original parameter associated with this action word is continued to be the action parameter after this action word is chosen to be the cluster representative. The statistic can be computed as shown in equation 4 that uses the *TF* and *IDF* equations in 2 and 3 respectively.

$$tf(t,d) = \frac{f_{t,d}}{max f_{t',d} : t' \in d} \tag{2}$$

$$idf(t,D) = log \frac{N}{|d \in D : t \in d|} \tag{3}$$

$$tfidf(t,d,D) = tf(t,d) * idf(t,D) \tag{4}$$

where t is the given action; d is the set of raw posts shared by a given user; D is the super set of all sets of raw posts made by all the users in our raw dataset (where $|D|$ will be equal to the number of unique users in our dataset). Each cluster will be represented by a unique top-ranked action word.

Running example: [clustering]: We map the action names to the cluster representatives of their corresponding cluster. spent → spend, drink → party, take → taken

3.4 Phase-4: Trace Builder

The initial plan fragments are converted into plan traces by replacing the action names with their corresponding cluster representatives[3]. This process is repeated on all the posts to build the traces.

Running example: [rebuilding plan traces]: initial plan fragment: [spent_smoke, drink_beer, party_hard, take_day, smoke_day, assault_trigger, smoke_day, thought_smoke, chase_life, quit_smoke], plan trace: [spend_time, party_hard, taken_hold, smoke_day, assault_trigger, smoke_day, thought_smoke, chase_life, quit_smoke]

The actions in the running example such as *spent*, *drink*, *take* are represented in their corresponding high level mapped format. Since *drink* is mapped to *party*, *drink_beer* is represented as *party_hard*. In the plan fragment, [*spend_time*, *party_hard*, *party_hard*, *taken_hold*, . . .], two '*party*' actions are occurring sequentially. Hence, we remove repetitions and include only one such instance. However, in general a plan may include repeated actions and we acknowledge that our system may miss out on those plans with repeated actions.

3.5 Phase-5: Sequence Probability Learner

After extracting the plan traces, we then extract the pre-actions and post-actions for any given action. Due to co-occurrence in the plan traces, actions are inter-related to other actions with a probability ($p(a_i, a_j)$) describing the chance of action a_j following action a_i. This probability is computed purely in a data-driven fashion. This procedure considers a constraint metric θ that decides whether a co-occurring relation should be included in the model. We compute the conditional probability $p(a_i \mid a_j)$ using the following equation:

$$p(a_i \mid a_j) = \frac{p(a_i \cap^* a_j)}{p(a_j)} \tag{5}$$

Let a_1 and a_2 are two actions where a_1 is an effect of a_2 which indicates that unless a_2 is executed, a_1 cannot be executed. The support-based probability then is computed where a_1 will be the postcondition of a_2 if $p(a_1 \mid a_2) > \theta$ and a_2 will be the precondition of a_1. The $\cap^*$ in Equation 5 represents an ordered conjunction that considers the sequentiality of a_1 and a_2 while computing the frequency of their occurrence together.

Once we extract the pre-actions and post-actions for every action in the data set, we represent the relationships in the form of an Action Graph (*AG*) as our incomplete action model (*M*) to generate shallow workflows. *AG* consists of actions as *nodes* and each *edge* is a transition between two actions a_i and a_j. The edge weight between any two nodes a_i and a_j is the support-based probability $p(a_i, a_j)$.

Running example: [data-driven probabilities]: In the plan trace dataset, we compute the sequential probabilities for any given action pairs. This results in generating a precedence graph shown in Figure 2 for the "Quit Smoking" domain, where the sink node is the action quit with no subsequent effect.

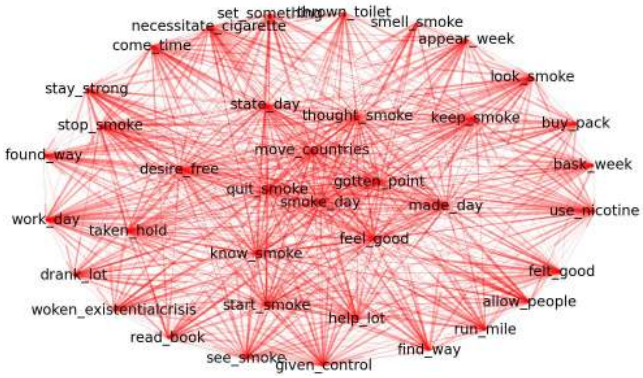


Figure 2: Part of the directed graph G for the $QuitSmoking$ domain with *quit_smoke* as the sink node.

3.6 Phase-6: Model Validator

We divide the set of plan traces *D* into training data, D_{tr}, and testing data, D_{te}. By this step, we have the set of plan traces represented in the lifted action format. This common lifted representation ensures that a given action uses the same name in both D_{tr} and D_{te}. We use D_{tr} to build the model *M*. Let *T* be the set of transitions present in *M* and

[3]Note, it is possible to replace two sequential action names by the same cluster representative. In that case we remove the repeated action name; hence, reducing the length of the plan trace. For example, if the representation of a plan trace is: $[a_1, a_2, a_2, a_5, a_8]$, after post-processing it will be converted to $[a_1, a_2, a_5, a_8]$.

$T^{'}$ be the set of transitions in test dataset. Since *M* is used to generate workflows, the goodness of this model should be measured to trust the quality of these plans. To determine goodness of *M*, we define a new metric called *explainability* that can be computed as shown in Equation 6.

$$T^{''} = T \cap T^{'}$$
$$Explainability = \frac{|T^{''}|}{|T^{'}|} \quad (6)$$

4 Evaluation Methodology

We evaluate the pipeline from two perspectives: 1) data and approach employed to construct the incomplete action model in terms of explainability 2) workflows generated by the incomplete action model in terms of soundness and completeness. We evaluate the data utilized by the pipeline followed by the extracted plans represented as shallow workflows. Although we have the incomplete action model in the *pre-action* $\longrightarrow$ *action* $\longrightarrow$ *post-action* format (a sample of these models extracted for the three domains is show in Table 3), we are still in the process of attempting to convert and refine this incomplete model to a PDDL-style model. This attempt could be a valuable contribution to the automated planning community (Srivastava and Kambhampati 2005). Planning community can no longer depend on a fixed set of domains for the International Planning Competition (IPC) challenges but instead expand the domains to any real-world scenarios.

Quit Smoking
(:action change(ability) [:pre-action eat(gross) crave(succeed) dealt(reality)] [:post-action set(goal) run(mile) quit(smoke)]) Possible explanation: Someone is craving for success and is dealing with the reality of eating gross food who wants to change his abilities that led that person to set some goals, run miles and quit smoking.
Couch to 5K
(:action sign(race) [:pre-action recommend(c25k) push(run) refer(program)] [:post-action begin(week) run(minute) cover(mile) know(battle) kept(pace)]) Possible explanation: A person was recommended the couch to 5K reddit forum and was being pushed to run. So, he refers to a program and signs up for the race. After this, he begins from the next week to run few minutes and cover few miles. The person knows the battle but he kept the pace.
Wedding Planning
(:action hate(dress) [:pre-action pick(dress) saw(dress) blame(problem) cost(much) prove(difficult)] [:post-action kill(wed) find(dress) move(wed)]) Possible explanation: The person sees and picks her dress. It may cost a lot but starts blaming someone for the problem and now hates the dress. The next steps could be to kill the wedding at the moment, find a new dress and move the wedding date.

Table 3: Sample actions from the incomplete models extracted for the 3 domains **automatically** by this pipeline and their possible explanations provided by the human subjects.

4.1 Evaluation-1 – Explainability

Prior to analyzing the pipeline, it is important to examine whether the data we are utilizing to construct the incomplete action models is consistent across all the experiential posts shared online by the users. To evaluate this, we measure the explainability of the incomplete action model by varying the α value (clustering threshold). α decides on the amount of redundancy to be removed from the posts. The smaller the value of α, the larger the redundancy present in the data considered. We fix the size of the training data (D_{tr}) to 80% of the entire set of plan traces and the remaining as the test data set (D_{te}) and conduct experiments on all the three domains separately. The dataset from each domain consists of a set of plans that are aimed at achieving the primary goal of the corresponding domain. The pipeline first utilizes D_{tr} to build the incomplete action model *M* and then use the test dataset D_{te} to evaluate the explainability of *M*.

α	**Quit Smoking**	**Couch to 5K**	**Wed. Planning**
2.50	65.66%	64.5%	73.39%
2.25	65.66%	64.59%	73.39%
2.0	68.41%	69.78%	77.7%
1.75	69.33%	70.67%	78.39%
1.50	80.58%	82.06%	84.68%
1.25	90.42%	89.43%	91.6%
1.0	89.31%	89.91%	91.04%

Table 4: Average explainability measured by Eq. 6 as we vary α through 10-fold cross-validation

As shown in Table 4, the maximum explainability value was reached at α = 1.25. It is expected that if the data and the approach are correct, the explainability value should be directly proportional to the value of α. This trend is clearly visible in the results shown in Table 4. This trend also positions more confidence in building the best incomplete model used to generate shallow workflows. Also, we focus on how well can these incomplete domain models explain the newly seen data to decide the consistency of goal-oriented experiences shared by users. The results obtained through 10-fold cross-validation show that *M* has the potential to obtain 90% accuracy. The results display the strength of unstructured data from social media platforms like Reddit could be employed to build incomplete models.

4.2 Evaluation-2 – Soundness & Completeness

Next, we examine the "goodness" of the incomplete action models by evaluating the generated shallow workflows. Each workflow is generated by representing the incomplete model as a graph and is the shortest path in this graph from a given source node to the goal node. For example, in *Quit Smoking* domain, the source node can be *start(smoke)* and the goal node is *quit(smoke)*. To identify the best path, we utilized the weight-based Djikstra's shortest path algorithm from the *NetworkX* (`https://networkx.github.io/`) Python library. We rate each plan on a binary-scale evaluating it's soundness and completeness metrics.

Domain	Soundness	Completeness
Quit Smoking	42%	38%
Couch to 5K	66%	45%
Wedding Planning	36%	43%

Table 5: Soundness and Completeness as evaluated by the human subjects. Note that higher the percentage values, the better the workflows that are generated.

Soundness: is defined as whether a given shallow workflow is meaningful and can help achieve the goal.

Completeness: is defined as if a given shallow workflow is missing any important actions to achieve the goal.

We recruited 10 human test subjects who evaluated the top-5 workflows generated by *M*. We provide instructions to the test subjects and ask them to rate the soundness and completeness of each workflow. Each subject evaluates all the top-5 workflows from the three domains and the combined statistics are shown in Table 5. Each percentage value in this table is the average value of all the votes gathered by the plans in a given domain.

The best plan among these 15 plans (combined all top-5 plans from the 3 domains considered) is from the *Couch to 5K* domain – *inhale(nose)* $\rightarrow$ *exhale(mouth)* $\rightarrow$ *aid(loss)* $\rightarrow$ *transform(life)* $\rightarrow$ *outpaced(brain)* $\rightarrow$ *slow(pace)* $\rightarrow$ *run(minutes)*. This shallow workflow was described by the human subjects as "If you inhale through nose and exhale from mouth (a powerful breathing pattern[4]) that will help you relax and transforms by keeping your slow pace to run the 5K in minutes." Notice that these workflows are not partially meaningful. However, the evaluation results showed that they make sense to humans as shown by the results presented in Table 5. The table shows that the *Couch to 5K* domain has highest soundness and completeness values which might be due to the fact that the number of original number of actions are relatively lower that led to a model with less redundancy. Another reason could be the workflows generated from this domain are more meaningful to the human test subjects. With regard to completeness, test subjects expressed the difficulty of not being completely aware of the domains and so by default assumed that there should be a missing action in the plan.

5 Related Work

The work reported in this paper brings together work from three communities that are quite far apart – traditional (classical) planning, social computing and Natural Language Processing (NLP).

Automated Planning & AI: Classical planning techniques have sought to mostly ignore domain acquisition and maintenance issues in favor of search efficiency and plan synthesis. Towards this goal, multiple works have focused on learning the action models through inductive logic programming, from sets of successful plan traces (Yang, Wu, and Jiang 2007; Zhuo et al. 2010), improving partial models (Oates and Cohen 1996; Gil 1994), etc. Recent work on model-lite planning (Kambhampati 2007; Yoon 2007; Zhuo, Nguyen, and Kambhampati 2013) acknowledges that learned models may be forever plagued by incompleteness and laden with uncertainty, and plan synthesis techniques themselves may have to change in order to accommodate this reality. Other existing works (Addis and Borrajo 2011; Lindsay et al. 2017; Tenorth, Nyga, and Beetz 2010; Waibel et al. 2011) that includes literature from the field of robotics attempts at learning action models from the web where they consider plans recommended by websites like wikihow.com for a given task. This work focuses on carefully constructing well-curated complete domain models where as the work proposed in this paper emphasizes on building incomplete models that are efficient enough to perform automated planning tasks that include plan recognition and obtaining meaningful plans.

Social Computing: On the other hand, work in the social media field puts a premium on the analysis of data, but very little on complex decision-making that can build on the knowledge embedded within that data (Kiciman and Richardson 2015). Given the high level human engagement with these platforms, researchers have sought to utilize the data generated on them for various analyses that can help understand and predict users' behaviors (Golder and Macy 2011; Sarker et al. 2015; Paul and Dredze 2011). Building on this theme of using human-generated data on social media, the crowd sourcing community realized that in addition to using the inadvertent by-product of user participation on social media, it could also directly utilize the "crowd" to prepare plans for goal-oriented tasks (Law and Zhang 2011; Manikonda et al. 2014). This work has gained traction in recent years in part as a response to the unavailability of good planning models for many real-world, everyday planning and scheduling tasks. Such "hybrid" intelligence systems utilize domain knowledge that is split between humans and machines, with each party possessing complementary information; unfortunately, these systems are still far from being scalable and cost-effective.

NLP: There is another set of work (Harabagiu and Maiorano 2002; Collier 1998) from the natural language processing (NLP) community which focus on extracting domain templates. The templates extracted from this literature capture most important information of a particular domain and they can be used across multiple instances of that domain especially in the field of information extraction. For example, the GISTexter summarization system considers semantic relations from WordNet along with summary statistics over an arbitrary document collection. This type of summarization could be at a disadvantage if there is only one instance of the domain as input as addressed by Filatova et. al 2006. This direction of work is later extended to identify event schema using count-bases statistics and by building formal generative models (Chambers and Jurafsky 2008; Chambers 2013). The main distinction of this line of research from our work is two fold: (1) the model: our main aim is to build shallow models where as, the existing literature aims at constructing full models; (2) the unstructured natural language data on online social media platforms: our proposed pipeline handles natural language with different

[4]https://goo.gl/BiKvGG

styles of language, where as the existing literature considers fairly structured text available online.

6 Conclusions and Future Work

To support sequential decision making, action models extracted from the unstructured data are very valuable. However, extracting these models from unstructured data is difficult. Towards exploring these challenges and to measure the feasibility of building usable action models, this paper proposes a novel six-phase pipeline. This pipeline takes as input the unstructured web data and automatically generates the incomplete action model. Through evaluations, we show the capability of utilizing shallow NLP techniques to overcome the challenges posed by various entities and successfully generate incomplete action models. We acknowledge that the workflows generated by the incomplete action models are shallow. However, the evaluations displayed the power of experiential statuses shared by the users on online social media platforms can be used to generate incomplete action model. Also, the capability of these models to generate plans as workflows are tagged by human subjects as sound to a certain extent.

As a future work, these incomplete action models can be translated to PDDL-style models. In addition, hierarchical representation of actions can be extracted in order to enhance the extracted models. We hope that this work inspires the research community to utilize the potential of incomplete action models to perform automated planning tasks. Also, considering the wealth of information present on online social media platforms especially the goal-oriented posts shared publicly, we envision that further action models are constructed towards supporting sequential decision making.

References

Addis, A., and Borrajo, D. 2011. From unstructured web knowledge to plan descriptions. *Information Retrieval and Mining in Distributed Environments* 41–59.

Chambers, N., and Jurafsky, D. 2008. Unsupervised learning of narrative event chains. In *Proc. ACL*.

Chambers, N. 2013. Event schema induction with a probabilistic entity-driven model. In *Proc. EMNLP*.

Collier, R. 1998. Automatic template creation for information extraction. In *Ph.D. thesis, Univerisyt of Sheffield*.

Filatova, E.; Hatzivassiloglou, V.; and Mckeown, K. 2006. Automatic creation of domain templates. In *Proc. COLING/ACL*.

Gil, Y. 1994. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proc. ICML*.

Golder, S. A., and Macy, M. W. 2011. Diurnal and seasonal mood vary with work, sleep, and daylength across diverse cultures. *Science* 333(6051).

Gregory, P., and Lindsay, A. 2016. Domain model acquisition in domains with action costs. In *Proc. ICAPS*.

Han, J.; Kamber, M.; and Pei, J. 2011. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 3rd edition.

Harabagiu, A. M., and Maiorano, S. J. 2002. Multi-document summarization with gistexter. In *Proc. LREC*.

Kambhampati, S. 2007. Model-lite planning for the web age masses: The challenges of planning with incomplete and evolving domain models. In *Proc. AAAI*.

Kiciman, E., and Richardson, M. 2015. Towards decision support and goal achievement: Identifying action-outcome relationships from social media. In *Proc. KDD*.

Law, E., and Zhang, H. 2011. Towards large-scale collaborative planning: Answering high-level search queries using human computation. In *Proc. AAAI*.

Lindsay, A. A.; Read, J. J.; Ferreira, J. F. J.; Hayton, T. T.; Porteous, J. J.; and Gregory, P. J. P. 2017. Framer: Planning models from natural language action descriptions. In *Proc. ICAPS*.

Manikonda, L.; Chakraborti, T.; De, S.; Talamadupula, K.; and Kambhampati, S. 2014. AI-MIX: Using automated planning to steer human workers towards better crowdsourced plans. In *Proc. IAAI*.

Oates, T., and Cohen, P. R. 1996. Searching for planning operators with context-dependent and probabilistic effects. In *Proc. AAAI*.

Paul, M., and Dredze, M. 2011. You are what you tweet: Analyzing twitter for public health. In *Proc. ICWSM*.

Sarker, A.; Ginn, R.; Nikfarjam, A.; OConnor, K.; Smith, K.; Jayaraman, S.; Upadhaya, T.; and Gonzalez, G. 2015. Utilizing social media data for pharmacovigilance: A review. *Journal of Biomedical Informatics* 54.

Srivastava, B., and Kambhampati, S. 2005. The case for automated planning in autonomic computing. In *Proc. ICAC*.

Tenorth, M.; Nyga, D.; and Beetz, M. 2010. Understanding and executing instructions for everyday manipulation tasks from the world wide web. In *Proc. ICRA*.

Tian, X.; Zhuo, H. H.; and Kambhampati, S. 2016. Discovering underlying plans based on distributed representations of actions. In *Proc. AAAMAS*.

Toutanova, K.; Klein, D.; Manning, C. D.; and Singer, Y. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proc. NAACL*.

Waibel, M.; Beetz, M.; Civera, J.; dAndrea, R.; Elfring, J.; Galvez-Lopez, D.; Haussermann, K.; Janssen, R.; Montiel, J.; Perzylo, A.; et al. 2011. A world wide web for robots. *IEEE Robotics & Automation Magazine* 18(2):69–82.

Yang, Q.; Wu, K.; and Jiang, Y. 2007. Learning action models from plan examples using weighted max-sat. *Artif. Intell.* 171(2-3).

Yoon, S. 2007. Towards model-lite planning: A proposal for learning and planning with incomplete domain models. In *Proc. Workshop on AI Planning and Learning, ICAPS*.

Zhuo, H. H.; Yang, Q.; Hu, D. H.; and Li, L. 2010. Learning complex action models with quantifiers and logical implications. *Artif. Intell.* 174(18).

Zhuo, H. H.; Nguyen, T.; and Kambhampati, S. 2013. Refining incomplete planning domain models through plan traces. In *Proc. IJCAI*.

Domain Model Acquisition with Missing Information and Noisy Data

Peter Gregory
Schlumberger Gould Research,
Madingley Road,
Cambridge, UK
pgregory@slb.com

Alan Lindsay and **Julie Porteous**
Digital Futures Institute,
School of Computing,
Teesside University, UK
firstinitial.lastname@tees.ac.uk

Abstract

In this work, we address the problem of learning planning domain models from example action traces that contain missing and noisy data. In many situations, the action traces that form the input to domain model acquisition systems are sourced from observations or even natural language descriptions of plans. It is often the case that these observations are noisy and incomplete. Therefore, making domain model acquisition systems that are robust to such data is crucial. Previous approaches to this problem have relied upon having access to the underlying state in the input plans. We lift this assumption and provide a system that does not require any state information. We build upon the *LOCM* family of algorithms, which also lift this assumption in the deterministic version of the domain model acquisition problem, to provide a domain model acquisition system that learns domain models from noisy plans with missing information.

Introduction

When faced with the task of creating a planning domain model that accurately models a real-world problem that needs to be solved, in most current situations an AI Planning expert must also learn to become a domain expert in the problem area to be modelled. Domain model acquisition is an area of research trying to reduce the gap between domain expert and modelling expert. Domain model acquisition is the problem of automatically generating planning models from input data of some form. This input data can vary in many ways, but typically at least contain collections of plan traces in some form. Other information that may be available are intermediate states, solution metadata (such as plan costs, or whether plans are goal-directed or optimal), etc.

In this work, we study the problem of domain model acquisition when the input plans have noisy data and missing information. This problem has previously been studied (Mourao et al. 2012) with the assumption that intermediate state information is present. We relax this assumption, and provide an algorithm that does not rely on intermediate state information being present. There are important situations in which intermediate state information cannot be accessed. For example, when translating plans created for people to follow (e.g. the machine tool calibration plans in (Parkinson et al. 2012)), the plans only mention the actions,

```
a) (new-move p1-0 p1-1 p1-2) b) (new-move p0-0 p1-1 p1-2)
   (continue p1-2 p2-2 p3-2)    (continue p1-0 p2-2 p3-2) *
   (end-move p3-2)              (end-move p3-2)
   (new-move p3-1 p2-1 p1-1)    (new-move p3-1 p2-1 p1-1)
   (end-move p1-1)              (end-move ____) **
```

Figure 1: The first plan (a) shown is an example plan from the English Peg Solitaire domain. The second plan (b) shows the same plan with noise (*) and missing information (**).

and there is no state description. Plans created by people for other people can also be prone to mistakes and oversights, or, in the language of this paper, noise and missing information. Another place in which missing and noisy data is a problem, and in fact a motivating reason for developing a domain model acquisition system of this type, is the Framer system (Lindsay et al. 2017): a domain model acquisition system that has natural language descriptions of plans as its input. Although the exact details are not important here, it should be clear enough that natural language descriptions of plans are prone to noise and missing information, and do not provide information about intermediate states.

We call our system *LC_M*, as it is a version of the *LOCM* system with noisy and incomplete data (the *C* is misplaced and the underscore represents missing data). The *LOCM* family of algorithms (Cresswell, Mccluskey, and West 2009; Cresswell and Gregory 2011; Gregory and Cresswell 2015; Lindsay et al. 2017) are domain model acquisition systems, all sharing the assumption that plan traces with no intermediate state form the input to the system. It has proven possible to correctly learn domain models with rich structure, including the vast majority of the IPC domains, whilst still adhering to this very strong assumption about the input data. In this work, we assume that the input plans are generated through some process of observation, whether human or machine. We assume that each action is observed, but that the observer may either perceive the wrong action type (i.e. the action name), and / or the wrong action parameters. Missing data can be seen if the observer fails to perceive an action or action parameter with any degree of certainty. We also have the assumption that there is an underlying deterministic planning model that would be learnt by *LOCM* if no noise and missing information were present.

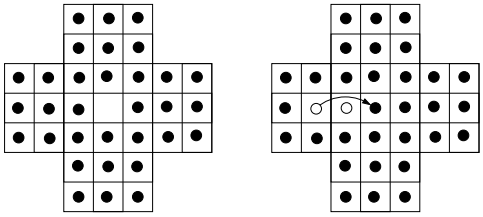


Figure 2: An English Peg Solitaire board, and an example of a move. Pegs must jump over other pegs, both removing the peg jumped over whilst leaving their current position clear.

Background

The *LOCM* system (Cresswell, Mccluskey, and West 2009; Cresswell, McCluskey, and West 2013) forms the basis for the work in this paper, therefore we provide an introduction to the most relevant parts of the *LOCM* system. To do this we use a running example of the one-player board game English Peg Solitaire. The goal is to clear the board of pegs, leaving a single peg in the middle. Pegs are cleared when an adjacent peg jumps over it, into another adjacent empty position, and this jump must be in a straight line. If the same peg performs more than one consecutive jump, then this counts as a single move in the optimisation criteria. In the planning domain, this is modelled as three different operators:

```
(new-move pos-from pos-over pos-end)
(continue pos-from pos-over pos-end)
(end-move pos)
```

The *LOCM* domain model acquisition system works by building finite state machines for each type of object in a planning domain, asserting that the behaviour of each object can therefore be defined as a finite state machine. It operates with the assumption that each action parameter asserts a transition in this state machine. Each object in a plan can be seen as going through a sequence of transitions, where a transition is defined by an action name and a parameter index. The transitions for the peg solitaire domain are shown in Table 1. For the plan in Figure 1 a) for example, the object `p1-1` has the transition sequence *new-move.2, new-move.3, end-move.1*. In Table 1, the imaginary zeroth parameters of the actions are also listed as transitions. This is important, as the structure of the plans can carry extra object-independent

Transition	Meaning
end-move.1	The position that a move ends on.
new-move.1	The position of the peg to move.
new-move.2	The position of the middle peg to be removed.
new-move.3	The empty position that the peg will land on.
continue.1	The position of the peg to move.
continue.2	The position of the middle peg to be removed.
continue.3	The empty position that the peg will land on.
end-move.0	The imaginary zeroth parameter of end-move.
new-move.0	The imaginary zeroth parameter of new-move.
continue.0	The imaginary zeroth parameter of continue.

Table 1: Table of transition meanings in peg solitaire domain.

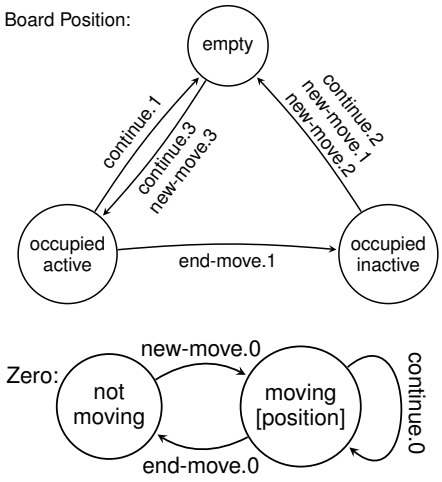


Figure 3: The finite state machines derived by *LOCM* for the Peg Solitaire domain. There are two state machines: one for the 'board position' type, and another for the zero machine, which models global dynamics of the system.

information about the domain structure. The zero transition sequence for the plan in Figure 1 a) then, is *new-move.0, continue.0, end-move.0, new-move.0, end-move.0*

The domain model that *LOCM* learns for this domain can be described by the state machines in Figure 3, where there are two different state machine types: one for board positions and a zero state machine. The zero machine is the machine generated by assuming that each operator has a hidden zeroth parameter, and this represents zero place predicates in the domain. A crucial assumption in the *LOCM* system is that each transition appears at most once in each state machine. In order to construct the state machines for each type, *LOCM* performs an incremental unification of states, based on the transition sequences seen in the input. The consequence of the rule that each transition appears at most once, is that for a transition sequence pair A,B the end state of the A transition is the start state of the B transition. For the zero machine in Figure 3, the machine is constructed using the steps defined in Figure 4. Note, importantly, that a transition pair can change the structure of the generated state machine significantly. This is important in the context of this work, because even a small amount of noise can lead to incorrect state machines being learnt, and hence provides strong motivation to find ways of dealing with noise.

The final aspect of the *LOCM* system to discuss is the learning of state parameters. State parameters define temporary relationships that exist between different object state

Transition	In Parameter	Out Parameter
end-move.0		end-move.1
new-move.0	new-move.3	
continue.0	continue.3	continue.1

Table 2: Table of the position state parameter transitions for the moving state of the zero machine in Figure 3.

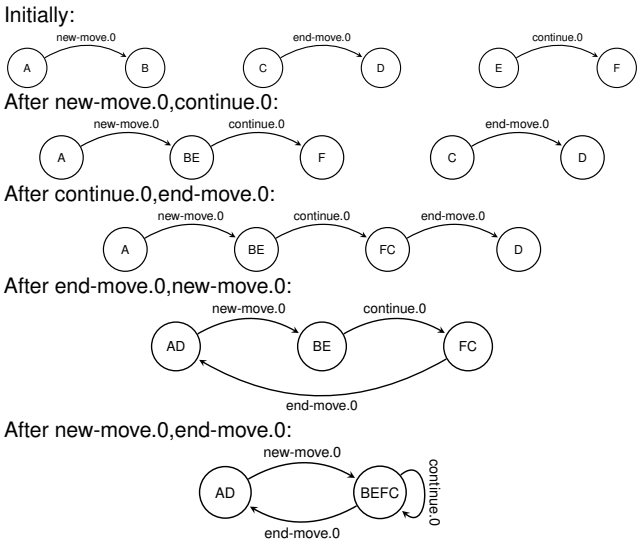


Figure 4: The progression of state unifications for the zero machine on the input plan in Figure 1 by *LOCM*. Initially, there is an assumption of independence between the different transitions. After considering the transitions in the example, the machine specified in Figure 3 is produced.

machines, typical examples include the location of a truck in a logistics domain. In the zero machine of Figure 3, for example, the moving state has the state parameter '[position]' which records the board position that is currently active during a move. In general, if a state in a *LOCM* state machine has a parameter, this means that for each pair of consecutive transitions in and out of the state there is a transition index in each that always has the same value. They effectively provide constraints between the parameters of the actions that affect the state machine with the parameter. The transition positions for the moving state in the zero machine are shown in Table 2.

Another convenient way of representing the input transitions is to look at the transition matrix. This idea was important in the *LOCM2* system (Cresswell and Gregory 2011) for deriving state machines for objects with multiple behaviours. We will, however, use the matrix for a different purpose here. The transition matrix for the board position type in the English Peg Solitaire domain is shown in Figure 5. The matrix has a row and a column for each transition that a board position can go through. A cell in the matrix is crossed if the input data sees two transitions in sequence row label and then column label.

Missing Values

In this section, we discuss how we deal with missing information in the absence of noise. It is important to discuss this first, as we use the techniques here as a sub-procedure when reasoning about plans with noise. To generate the state machines, we first split the input plans around the missing information, so that there are now a greater number of shorter input plans, with no missing information. Since we assume no noise, then we can generate *LOCM* state machines, based on

	e-m.1	n-m.1	n-m.2	n-m.3	c.1	c.2	c.3
end-move.1		X	X			X	
new-move.1				X			X
new-move.2				X			X
new-move.3	X				X		
continue.1				X			X
continue.2				X			X
continue.3	X				X		

Figure 5: The transition matrix for the English Peg Solitaire domain. The names of the transitions are abbreviated on the column labels in order to preserve space.

the parts of the input plans with no missing information. We then employ the standard *LOCM* algorithm discussed above to learn the state machines, along with their state parameters.

After this process, we then use these *LOCM* machines to fill in the missing information. In order to deal with missing information, we rely on a constraint encoding of the input plans and the state machines generated by the *LOCM* system. This constraint model is a matrix model of each plan, in which the rows correspond to time-stamped variables for each time t: the action label, the action parameters, the object *LOCM* states and the values of the state parameters. The constraints are conceptually similar to those used in the constraint-based planners SeP (Barták and Salido 2011) and Constance (Gregory, Long, and Fox 2010).

If an object is in the argument of an action, then it must transition in the correct way that its *LOCM* state machine defines. Constraints are posted to ensure that state parameters appear in the correct arguments of the actions. If an object is an appropriate filler for a missing value then a conditional constraint updates its state if is selected to fill the missing value. If an object is inappropriate or not selected then its state is unchanged between timesteps. Similarly, for missing action names, conditional constraints are posted for each of the possible action fillers. Because much of the plan is known, the vast majority of the timeline can be filled in immediately, leaving only the states in which there are gaps in the operator name and arguments. Figure 8 shows an example of a timeline, for the plan in Figure 1 a) with the missing parameter from 1 b). The missing parameter was in the *end-move.1* position of the final action, and is highlighted in bold. The only consistent value that can occupy this position is the *p1-1* object: this can be seen by the fact that the object performing the *end-move.1* transition starts in the *occupied active* state, and only *p1-1* meets this condition.

Using this type of approach to finding missing data leads to two risks. Firstly, that once the plans are split into smaller plans, there will not be enough data to correctly learn the *LOCM* state machines. However, *LOCM* typically only requires a small amount of data to learn correct domain structures this is unlikely to be an issue. The other risk to this kind of approach is that there are multiple consistent objects which could take the place of the missing argument, which is an unfortunately an unavoidable problem. An algorithmic

▷ The Missing Value Model Acquisition System
function missingValueModel(Π : a collection of plans)
  $\Pi' \leftarrow$ split plans on missing information
  $M \leftarrow LOCM(\Pi')$
  $M_c \leftarrow$ constructConstraintModel(M, Π)
  if solve(M_c) is consistent **then**
    return solve(M_c)
  else
    return inconsistent
  end if
end function

Figure 6: The missing value model algorithm. The function constructConstraintModel returns the constraint model described above.

description of how to perform domain model acquisition in the presence of missing information is shown in Figure 6. We now change focus to discuss how we use these results to help deal with problems involving noisy data.

Noisy Data

Consider the plan fragment in Figure 7, with a single mistake in the parameters (where p2-1 in action 5 should be p2-2). By simply changing a single object parameter, the transition sequence of two objects are corrupted: the swapped in object and the swapped out object (in this case, the objects p2-1 and p2-2). Because of the error, the transition sequence for p2-1 is now *new-move.3, end-move.1, new-move.3, new-move.1* and the transition sequence for p2-2 is *new-move.2, end-move.1, new-move.2*. Each transition pair in those sequences are invalid in the true domain. Figure 9 shows the occurrence matrix (a version of the transition matrix in which the number of times each transition pair occurs is shown in each cell) for a small number of plans, including a plan which included the error from the previous example. The values corresponding to the error are highlighted. Our technique for domain model acquisition when plans may contain noise rely on forming hypotheses about which cells in the occurrence matrix may contain errors, and trying to find replacement values which still support the data.

In this section we consider the implications of noisy data on the generated *LOCM* model. There are two ways an error may add connections to the transition matrix: where obj$^{\times}$ (an error action symbol) adds connections in the matrix, because of the new transition pairs added, or where obj$^{\checkmark}$ (the correct action symbol) adds a new spanning connection, because of the missing transition. We consider each

```
1: (new-move p1-2 p2-2 p2-3)
2: (end-move p2-3)
3: (new-move p4-1 p3-1 p2-1)
4: (end-move p2-1)
5: (new-move p2-4 p2-3 p2-1)*
6: (end-move p2-2)
7: (new-move p2-1 p2-2 p2-3)
```

Figure 7: Plan fragment with single error.

time	0	1	2	3	4	5
action		n-m	c	e-m	n-m	e-m
arg1		p1-0	p1-2	p3-2	p3-1	**p1-1**
arg2		p1-1	p2-2		p2-1	
arg3		p1-2	p3-2		p1-1	
zero	NM	M	M	NM	M	NM
M-prm		p1-2	p3-2		p1-1	
p1-0	OI	E	E	E	E	E
p1-1	OI	E	E	E	OA	**OI**
p1-2	E	OA	E	E	E	E
p2-2	OI	OI	E	E	E	E
p3-2	E	E	OA	OI	OI	OI
p3-1	OI	OI	OI	OI	E	E
p2-1	OI	OI	OI	OI	E	E

Figure 8: Timeline visualisation of the plan from Figure 1 a) with the missing parameter from 1 b). Abbreviations used in the table are n-m (new-move), c (continue), e-m (end-move), M-prm (the state parameter of the moving state of the zero machine, NM (not moving), M (moving), OI (occupied inactive), E (empty) and OA (occupied active). The correct value of the missing parameter and its state value is shown in bold.

of these cases below.

Added transitions The modified action becomes a new observation, t, in the transition sequence of obj$^{\times}$. If the obj$^{\times}$ and obj$^{\checkmark}$ are of the same type then the added transition may induce new connections due to new otherwise unseen orderings of transitions (e.g., an end-move.1 transition added, but with no preceding jump-new-move.3 or continue-3). For example, if we consider the plan fragment:

```
(end-move p3-6)
(new-move p4-0 p4-1 p4-2)
(end-move p3-6)
```

In this example, the final end-move action argument, pos-4-2, has been replaced by pos-3-6. As a result the transition sequence for pos-3-6 includes: end-move.1;end-move.1. This will contrast with any correct sequence, which will have either jump-new-move.3;end-move.1 or continue-3;end-move.1.

If they are of different sorts then, from the definition of sort in *LOCM*, t will not be a correct transition of obj$^{\times}$. However, this erroneous observation will collapse the sorts together, losing any distinction between them. Also, new connections will be made from the previous transition of obj$^{\times}$ to t and then from t to the next transition of obj$^{\times}$.

Missing transitions There is not always enough information to detect an error directly. In these cases we might detect the error in the transition sequence of the correct argument. For example, consider the peg-solitaire sequence below, where the last parameter of a jump-new-move action has been swapped:

```
(new-move p5-2 p4-2 p3-2)
...
(new-move p4-0 p4-1 p3-6)
(move p4-2)
```

In this example, pos-3-6 does not appear in further transitions; therefore there is no direct clue that this object is incorrect. However, if we consider the correct argument: pos-

	e-m.1	n-m.1	n-m.2	n-m.3	c.1	c.2	c.3
end-move.1		61	76	**1***		12	
new-move.1				39			5
new-move.2	**1***			47			9
new-move.3	148	**1***			37		
continue.1				8			0
continue.2				12			0
continue.3	37				15		

Figure 9: The occurrence matrix for the English Peg Solitaire domain. Instead of showing just which pairs of transitions occur, as in the transition matrix, we show the number of times that the transition pairs happen (in this instance for a set of random walks). A single error is introduced to the data, and the faulty transitions are labelled with asterisks.

4-2 then this error will add the transition pair: jump-new-move.2;end-move.1, which will not be observed in a correct trace. This indicates the possibility of a missing transition, M, which extends the current transition pair, X;Y, to X;M;Y, for some M such that connected[X][M] and connected[M][Y].

Action labels We assume that the number of arguments of each action header in the plan trace is consistent with the correct action. *LOCM* requires consistent arities for the same action headers. We therefore select the most frequent arity for each action symbol and replace any others with the missing value symbol. The implication of an incorrect action symbol is that each argument will generate potentially unexpected transition sequences, essentially acting similarly to an added transition. In the case of structural redundancy there will be no clues (e.g., fly and zoom in Zeno-travel).

Building Error Hypotheses

We define an error hypothesis as a set of symbols (either action names or action parameters) in the input plans that we have supporting evidence to believe are incorrect. In order to detect noise in the plan traces, we have formulated two distinct ways of hypothesising errors in the plan symbols. These are from the occurrence matrix and from the potential state parameters. Recall from Figure 9 that noise in the data often translates into cells in the occurrence matrix that have low values. Even a single error can impact on the structure generated by *LOCM*. Firstly, it can lead to badly-formed state machines. In Figure 4, for example, observing two new-move.0 transitions incorrectly would lead to states AD and BEFC being unified. An error can lead to state parameters not being discovered. The state parameter in the 'moving' state of the zero machine in Figure 3, for example, is supported by action sequences having co-occurring parameters. Take the action sequences:

```
a) (new-move p1-0 p1-1 p1-2)  b) (new-move p1-0 p1-1 p1-2)
   (continue p1-2 p2-2 p3-2)     (continue p1-2 p2-2 p4-2)
   (end-move p3-2)               (end-move p3-2)
```

Where the continue.3 transition is an error in sequence b).

	e-m.1	c.1	c.2	c.3
continue.1	0.01	0.00	0.00	0.00
continue.2	0.01	0.00	0.00	0.02
continue.3	1.00	1.00	0.00	0.01
new-move.1	0.01	0.00	0.01	0.00
new-move.2	0.00	0.01	0.00	0.00
new-move.3	0.99	1.00	0.00	0.01

Figure 10: A state parameter ratio matrix for the moving state in the zero state machine. Each element in the grid represents the proportion of in-out transition pairs from the moving state that accompany a transition and have the same object in their respective arguments.

LOCM will reject the state parameter in the moving state of the zero machine because the object in the continue.3 argument does not match the one in the following end-move.1 argument. In an analogous way to which we build the occurrence matrix, we build a state parameter ratio matrix in order to see how frequently objects co-occur in the plans. Figure 10 shows an example of this for the moving state of the zero state machine in Figure 3. Taking the top-left corner as an example, each time an end-move follows a continue action, the object in the first argument of the end-move is the same as the first argument in the continue action 1% of the time. Strictly, in order to support a state parameter, then all in-out actions should have a pair of transitions which always coincide. However, we note that pairs with high ratios may in fact *always* coincide without the presence of noise.

Our approach for forming hypotheses about the noise in plan traces starts by examining each element of the structures generated by *LOCM* and considering to what extent it is supported in the data. There are two main outputs of the *LOCM* analysis: the transition matrix and the state parameters. In each case we can count the number of examples in the data that either support (connections in the transition matrix) or refute (state parameters) a structural element. We interpret weak support for model structures as an indication of erroneous input and use these as starting points for fixing the data. Our approach is to focus on specific weakly supported elements and then attempt to remove them. The goal here is to find small changes to the plan traces that result in structures that are well supported in the data.

We start by attempting to remove connections (pairs: $t1; t2$) from the transition matrix. We form only hypotheses on values that fall underneath a threshold value t_{Occ}. We focus on each of the plan step pairs that are represented by $t1; t2$ and attempt to find fixes that can be explained by the model without the sequence: $t1; t2$. There should always be an alternative explanation in the case of errors. This is because we assume that correct structure will be well supported in the surrounding plan traces. For state parameters, we look for high values in the state parameter ratio matrix. Again, we apply a threshold, t_{SPR} to the ratios between sequential transitions, and therefore identify the most likely relationships obscured by noise.

```
▷ The LC_M Domain Model Acquisition Algorithm
function LC_M(Π : a collection of plans)
    Occ ← the occurrence matrix
    SPR ← the state parameter ratio matrices
    OccTrs ← the transition pairs in Occ < t_Occ
    SPRTrs ← the transition pairs in SPR > t_SPR
    M ← missingValueModel(Π)
    for all h ∈ hypotheses(OccTrs, SPRTrs) do
        Π′ ← replace hypothesised noise(Π, h)
        if missingValueModel(Π′) is consistent then
            M ← missingValueModel(Π′)
        else
            return M
        end if
    end for
    return M
end function
```

Figure 11: The *LC_M* algorithm. The function hypotheses returns the hypothesised structural faults in the transitions, ordered by how much support there is in the data for the fault. The algorithm either completes when all the faults have been shown to be faults or when one hypothesis leads to an infeasible model.

The *LC_M* Algorithm

So far we have described solutions to two separate problems. Firstly, how to fill in missing data from plan traces, and secondly how to identify which structural elements seem badly supported and may be artifacts of noise in the plan traces. We now show a simple, but powerful, way in which these ideas can be combined to provide an algorithm that generates domain models in the presence of both noise and missing information. The key to the algorithm is that once structural elements are hypothesised as incorrect, all of the objects that lead to the hypothesised faulty structure can be transformed into missing information. At this point, the constraint model for discovering the most likely missing objects can be employed as firstly a test of consistency over the data and (providing the changes are consistent) will provide suitable object replacements for the noise values. The complete *LC_M* algorithm is given in Figure 11.

Consider the plan in Figure 7. Suppose that the occurrence matrix in Figure 9 and the state parameter ratio matrix in Figure 10 represent the plans from which the plan is taken. The transition pair new-move.3,end-move.1 in the matrix has a ratio of 0.99, meaning that 99% of the time, these arguments were equal in sequential actions. It seems likely that this is really 100% and the 1% remaining is an artifact of the noise. We hypothesise that this transition pair is part of a state parameter and remove any argument value that does not support this hypothesis. One part of the plan in Figure 7 that does not support the hypothesis is:

```
(new-move p2-4 p2-3 p2-1)
(end-move p2-2)
```

The hypothesis leads to the removal of the new-move.3 and end-move.1 argument values.

Domain	ER	AE	TME	SPE	TME'	SPE'
Grid	0.001	3	3	0	1	0
	0.005	24	12	1	1	0
	0.010	52	20	3	7	1
	0.050	224	36	5	15	3
	0.100	497	47	5	28	4
Gripper	0.001	3	5	3	0	0
	0.005	38	27	4	4	3
	0.010	77	36	4	12	2
	0.050	355	49	4	25	5
	0.100	784	50	4	20	4
Logistics	0.001	3	6	6	8	11
	0.005	30	37	11	9	11
	0.010	66	84	13	16	12
	0.050	301	187	13	56	14
	0.100	649	214	13	63	12
Parking	0.001	3	4	3	1	0
	0.005	38	23	5	4	3
	0.010	78	36	5	10	4
	0.050	360	50	5	18	4
	0.100	785	59	5	16	5
Peg Sol.	0.001	1	1	0	0	0
	0.005	15	12	2	6	3
	0.010	34	14	2	1	1
	0.050	134	31	2	9	5
	0.100	296	34	4	7	4
Storage	0.001	3	2	3	0	9
	0.005	29	31	12	4	3
	0.010	63	65	12	16	6
	0.050	269	175	12	76	12
	0.100	583	231	12	94	12
TyreWorld	0.001	2	4	1	1	1
	0.005	23	29	2	4	1
	0.010	49	52	2	12	1
	0.050	200	106	3	21	2
	0.100	449	142	3	47	2

Table 3: The results of the empirical evaluation on noisy data for the *LC_M* system. The abbreviations in the headings are ER (Error Rate), AE (Atomic Errors, TME (Transition Matrix Errors), SPE (State Parameter Errors), TME' (Transition Matrix Errors in corrected model), SPE' (State Parameter Errors in corrected model)

```
(new-move p2-4 p2-3 _____)
(end-move _____)
```

The constraint model confirms that the data has a consistent domain model, with the state parameter, and assigns a consistent object to the missing arguments. For Figure 7, there is only one value that can occupy these parameters: p2-2.

Empirical Analysis

In this section we present an evaluation of the system. The aim is to establish how robust our approach is to missing information and noise. In order to do this we first generate a model using plans with no errors and use this as the correct model for comparison. The noisy training data sets

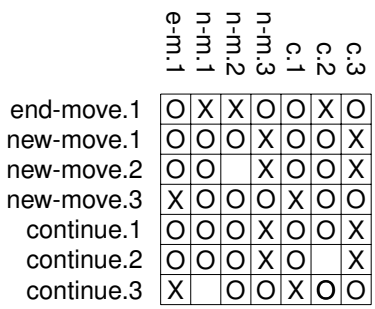


Figure 12: The transition matrix for the English Peg Solitaire domain with 5% noise. The 'O' entries represent the errors in the matrix.

are each derived from these plans, which means that we can determine how well our approach to correcting the errors performs. A collection of standard benchmark planning domains is used for the evaluation.

All of our experiments are run on Mac OSX version 10.11.6 using an Intel 2 GHz i7 CPU with 8 GB system memory. *LC_M* is implemented in Java (version 1.8.65) using the Choco constraint library (Prud'homme, Fages, and Lorca 2014) version 3.3.1.

We have simulated noisy data by first generating a set of action sequences for each domain and then adding noise. In each of the domains, except Grid, we have used the first 10 problems from the standard benchmark sets and generated 5 action sequences for each problem. In Grid the 5 problems were used and 10 action sequences were generated per problem. In most domains the action sequences are random walks of a length randomly selected between 1 and 100. However, for Grid and Peg-solitaire, where random walks provide poor coverage of the transitions, a goal-directed plan is used as one of the action sequences for each problem.

Each plan is passed through a channel that simulates the introduction of certain types of noise. The generated noise is governed by the probability for recording an incorrect value and can function with or without action symbol replacement. Replacement objects are drawn from the set of objects observed in the plan and the replacement action symbols are randomly selected from the observed actions.

We first evaluate our system on action sequences with increasing numbers of missing information. We then test the system on noisy plans.

Missing Information

We first test how robust the system is at filling in plans that contain only missing information. This forms an important subsystem of our complete approach and therefore provides an indication of how robust our system will be to noise in each of the tested domains. As detailed above, the plans are split around actions that contain missing information, resulting in a smaller set of plans with no noise. These plans are then used as input to *LOCM*, which induces a transition matrix and set of state parameters. A CP model is then constructed using the original plans with variables for each missing value. This model enforces the transition and state parameter constraints. If the split plans provide sufficient information to construct a correct model then solutions for the CP should be valid action sequences.

Error rate	0.005		0.01		0.05		0.1	
#Errors	$\#E_{start}$	$\#E_{end}$	$\#E_{start}$	$\#E_{end}$	$\#E_{start}$	$\#E_{end}$	$\#E_{start}$	$\#E_{end}$
Grid	29	2	68	5	337	51	711	240
Gripper	47	0	105	0	524	5	1076	13
Logistics	36	1	86	0	408	22	715	74
Parking	46	0	113	0	499	42	1098	218
Pegsol	21	3	36	10	207	67	408	136
Storage	39	0	69	3	366	17	717	121
Tyreworld	33	7	70	32	303	122	653	446

Table 4: The number of errors before ($\#E_{start}$) and after ($\#E_{end}$) parameter filling for several domains and error rates.

There are several ways that the CP model can have different output from the original plan sequences. The planning model will quite often allow alternative values for the same missing value. For example, consider a truck with 2 packages in it. If a put down action is missing the package and no further references are made to these packages then there is no information to distinguish them. As symbols are removed from the plan more symbols will appear equivalently appropriate for filling a specific action name or object. This is particularly relevant as we are only considering the dynamics of the model. As there are typically more than one argument for an action it is less likely, although possible that action symbols can be substituted in a similar manner.

As well as making equivalent replacements, the CP can also fail to produce a solution. This happens when the missing information in the context of a specific object results in either the type of the object becoming undetectable (only appears where the action symbol is missing), or the object becomes completely unobserved. In the former case, the original plans are used to guess its type as the most commonly observed type (this approach is helpful in the case of noise). Therefore the CP may fail because the correctly typed object may not be available.

Table 4 presents the number of errors in the plans (here a missing symbol is counted as an error) before and after our system has been used to fill the parameters. The results show that the system is often able to complete the majority of the missing information. It should be noted that we are reporting symbol errors as opposed to the number of actions with errors in them. For example, in the Parking domain at the 0.1 error rate, 36% of actions had at least one error.

As the error rate increases the performance degrades. This is expected because as the number of missing values increases, the number of alternative plans increases. Also, the increased missing information will increase the number of objects that become obscured, resulting in fewer of the CP models being solvable. In Gripper, Logistics, Parking, Peg-solitaire and Storage the number of failed models at each rate was 10 or fewer, leaving at least 40 completed plans that can be used to learn a model. In Tyreworld and Grid there were 21 and 14 failed models at the 0.1 rate.

Another factor is that as the amount of missing information is increased and the plans are broken up into smaller pieces the number of observations of each transition pair reduces. This can lead to erroneous state parameters being in-

duced, or, in the case of no observations, missed transition pairs. As a result, in Tyreworld and Grid, *LOCM* induced tighter models than the original domain models at the 0.1 and 0.05 rates and also for Grid at the 0.01 rate. These extra constraints can prevent the correct objects from being filled back into the plans.

Noisy Data

In this section we evaluate the complete system on plans with noisy data. The results for missing information indicate that the underlying missing information filling system is fairly robust in most of the tested domains. In this part of the evaluation the system first identifies weakly supported structural elements and then attempts to modify the input plans so that the structure is no longer supported by the data. In this section we therefore examine whether *LC_M* is able to isolate the erroneous structural elements without breaking those parts of the structure that were not effected by the errors.

We have used the same baseline training data and simulated noise using symbol replacement rates of: 0.001, 0.005, 0.01, 0.05 and 0.1. In this evaluation we have considered noise in the arguments of the actions and not in the action name. The system starts by using the noisy plans to generate the transition pairs and state parameter structures. The set of hypotheses for weakly supported structure elements are created from the set of all transition pairs and partial state parameters. These are ordered using a notion of how well they are supported in the data. For state parameters this uses the number of matched parameters against not matching parameters; and for transitions pairs this uses the number of occurrences of the transition pair against other pairs. A threshold value of 0.5 was used to prune the weakest of these hypotheses. E.g., a partial state parameter hypothesis is pruned if it is supported by less than half of the occurrences in the data. We test each hypothesis by first of all identifying the actions and the specific arguments in each plan that are inconsistent with the hypothesis. These arguments are replaced by the missing information symbol. The missing information filling system is then used to determine if there exist completions of the plans that support the hypothesis. If there are then the hypothesis is accepted, otherwise the hypothesis is rejected. The next hypothesis is then tested.

Table 3 presents the number of transition matrix and partial state parameter differences, as well as the underlying symbol differences. It shows that for low levels of noise, *LC_M* typically corrects the majority of structural errors introduced. As the noise level increases to 10%, fewer structural errors are corrected. However, their number is still typically reduced by half. State parameters are less well supported than transition matrix errors, in some cases introducing errors, even. However, several domains do see reductions in state parameter errors. Note that it is the transition matrix that determines the state machines in *LOCM* and so it is in some ways more important to reduce the errors in the transition matrix.

It may be informative to consider the effect of 5% noise, in order to understand why it becomes so hard to correct errors at this level. Figure 12 shows the transition matrix from Figure 5 overlayed with the transitions induced by this level of noise. As can be seen, there are only three transition pairs that are correctly identified as missing. Despite this, 22 of these erroneous transition pairs were detected and removed by *LC_M*. Clearly the fact that nine invalid transition pairs remain is problematic, and future work will determine whether this number can be reduced still further. The interaction between different hypotheses provides one of the most problematic difficulties: the decisions made in supporting an early hypothesis can make it impossible to support a later one, for example. One solution to this issue could be to consider the entire current hypothesis set *simultaneously*.

Related Work

In addition to the *LOCM* family of algorithms, there is a great amount of work in planning domain model acquisition without noise and missing information, from the TRAIL system (Benson 1996) to Opmaker (McCluskey et al. 2009; Richardson 2008), ARMS (Wu, Yang, and Jiang 2007), LAMP (Zhuo et al. 2010) and ASCOL (Jilani et al. 2015). To our knowledge, there is no other domain model acquisition system targeting noisy and incomplete input, except for (Mourao et al. 2012), which depends on intermediate state information. In addition to the planning community, there is wide and active interest in automatic model acquisition in many of the sub-fields of combinatorial search and beyond, for example in constraint satisfaction (O'Sullivan 2010; Bessiere et al. 2014), general game playing (Björnsson 2012; Gregory, Björnsson, and Schiffel 2015), and software engineering (Reger, Barringer, and Rydeheard 2015).

Conclusions

Modelling planning domains is a difficult and time consuming activity in general. Tools that can assist a domain expert in formulating a representation of their planning problem can save time and widen the usage of planning technologies. Domain model acquisition systems allow models to be learnt from existing plans. However, the process that observes the input plans may be prone to errors itself, making the input plans an unreliable source due to noise and missing information. In this work, we have presented a technique for performing domain model acquisition in the presence of noise and missing information.

LC_M returns the most likely underlying deterministic domain based on the frequency of support for various domain structures, such as the occurrence matrix cells and *LOCM* state parameters provided in the input data. The result of this is a system that learns planning domain models in a larger number of real-world situations. Although the performance of *LC_M* is already promising, structural flaws will remain for input data generated using a high-noise process, though reduced. An important future line of research will focus on the limits of such an endeavour: whether more errors can be removed based on better hypothesis schemes, or whether there is a technical limit beyond which errors cannot be distinguished from valid structure.

Acknowledgements

This work is supported by EPSRC Grant EP/N017447/1.

References

Barták, R., and Salido, M. A. 2011. Constraint satisfaction for planning and scheduling problems. *Constraints* 16(3):223–227.

Benson, S. S. 1996. *Learning Action Models for Reactive Autonomous Agents*. Ph.D. Dissertation.

Bessiere, C.; Coletta, R.; Daoudi, A.; Lazaar, N.; Mechqrane, Y.; and Bouyakhf, E. H. 2014. Boosting Constraint Acquisition via Generalization Queries. In *European Conference on Artificial Intelligence*.

Björnsson, Y. 2012. Learning Rules of Simplified Boardgames by Observing. In *European Conference on Artificial Intelligence*, 175–180.

Cresswell, S., and Gregory, P. 2011. Generalised Domain Model Acquisition from Action Traces. In *International Conference on Automated Planning and Scheduling*, 42 – 49.

Cresswell, S. N.; Mccluskey, T. L.; and West, M. M. 2009. Acquisition of Object-Centred Domain Models from Planning Examples. In *ICAPS*, 338 – 341.

Cresswell, S.; McCluskey, T.; and West, M. 2013. Acquiring planning domain models using LOCM. *The Knowledge Engineering Review* 28(2):195 – 213.

Gregory, P., and Cresswell, S. 2015. Domain Model Acquisition in the Presence of Static Relations in the LOP System. In *International Conference on Automated Planning and Scheduling*, 97–105.

Gregory, P.; Björnsson, Y.; and Schiffel, S. 2015. The GRL System : Learning Board Game Rules With Piece-Move Interactions. In *GIGA*.

Gregory, P.; Long, D.; and Fox, M. 2010. Constraint Based Planning with Composable Substate Graphs. In *European Conference on Artificial Intelligence*, 453–458.

Jilani, R.; Crampton, A.; Kitchin, D. E.; and Vallati, M. 2015. Ascol: A tool for improving automatic planning domain model acquisition. In *AI*IA 2015, Advances in Artificial Intelligence - XIVth International Conference of the Italian Association for Artificial Intelligence, Ferrara, Italy, September 23-25, 2015, Proceedings*, 438–451.

Lindsay, A.; Read, J.; Ferreira, J.; Hayton, T.; Porteous, J.; and Gregory, P. 2017. Framer: Planning models from natural language action descriptions. In *ICAPS*.

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

Mourao, K.; Zettlemoyer, L.; Petrick, R. P. A.; and Steedman, M. 2012. Learning STRIPS Operators from Noisy and Incomplete Observations. In *Uncertainty in Artifical Intelligence*, 614 – 623.

O'Sullivan, B. 2010. Automated modelling and solving in constraint programming. In *AAAI*.

Parkinson, S.; Longstaff, A.; Crampton, A.; and Gregory, P. 2012. The Application of Automated Planning to Machine Tool Calibration. In *International Conference on Automated Planning and Scheduling*.

Prud'homme, C.; Fages, J.-G.; and Lorca, X. 2014. *Choco3 Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S.

Reger, G.; Barringer, H.; and Rydeheard, D. 2015. Automata-based Pattern Mining from Imperfect Traces. *ACM SIGSOFT Software Engineering Notes* 40(1):1–8.

Richardson, N. E. 2008. *An Operator Induction Tool Supporting Knowledge Engineering in Planning*. Ph.D. Dissertation, School of Computing and Engineering, University of Huddersfield, UK.

Wu, K.; Yang, Q.; and Jiang, Y. 2007. ARMS: An automatic knowledge engineering tool for learning action models for AI planning. *The Knowledge Engineering Review* 22(2):135–152.

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