



# KEPS 2011

Proceedings of the Workshop on Knowledge  
Engineering for Planning and Scheduling

Freiburg, Germany  
June 12, 2011

**Edited by**  
**Roman Barták, Simone Fratini,**  
**Lee McCluskey, Tiago Stegun Vaquero**

## Organization

**Roman Barták**, Charles University, Czech Republic  
contact email: bartak@ktiml.mff.cuni.cz

**Simone Fratini**, ISTC-CNR, Italy  
contact email: simone.fratini@istc.cnr.it

**Lee McCluskey**, University of Huddersfield, UK  
contact email: lee@hud.ac.uk

**Tiago Stegun Vaquero**, University of Sao Paulo, Brazil  
contact email: tiago.vaquero@usp.br

## Program Committee

**Mark Boddy**, Adventium Labs, U.S.A.

**Adi Botea**, NICTA/ANU, Australia

**Luis Castillo**, IActive Intelligent Solutions, Spain

**Amedeo Cesta**, ISTC-CNR, Italy

**Stefan Edelkamp**, Universität Dortmund, Germany

**Susana Fernández**, Universidad Carlos III de Madrid, Spain

**Jeremy Frank**, NASA Ames, USA

**Antonio Garrido**, Universidad Politecnica de Valencia, Spain

**Arturo González-Ferrer**, University of Granada, Spain

**Rania Hatzi**, Harokopio University of Athens, Greece

**Peter A. Jarvis**, NASA, U.S.A.

**Karen Myers**, SRI International, USA

**John Levine**, University of Strathclyde, UK

**José Reinaldo Silva**, University of Sao Paulo, Brazil

**David E. Smith**, NASA, USA

**Dimitris Vrakas**, Aristotle University of Thessaloniki, Greece

## Foreword

Knowledge engineering for AI planning and scheduling deals with the acquisition, design, validation and maintenance of domain models, and the selection and optimization of appropriate machinery to work on them. These processes impact directly on the success of real planning and scheduling applications. The importance of knowledge engineering techniques is clearly demonstrated by a performance gap between domain-independent planners and planners exploiting domain dependent knowledge. Despite the progress in automated planning and scheduling systems, these systems still need to be fed by careful problem description and they need to be fine tuned for particular domains or problems.

This, the third KEPS workshop in the series at ICAPS (following on from 2008 and 2010), promises to be the most successful yet in terms of participation. This is evidenced by the 17 papers in this volume, written by authors from 13 different countries representing 4 continents, showing a growing interest in KEPS and a global participation in the subject area. In the one-day workshop authors will be able to present their papers by talks, demonstrations and posters, depending on the nature of the content of their paper and its likely appeal to workshop participants. The workshop will also continue to debate the perennial theme of the Knowledge Engineering Competition (ICKEPS) during a plenary session, in preparation for the planned fourth run of ICKEPS at ICAPS 2012. Topics of the papers this year include methods and tools for knowledge acquisition and domain modelling, plan visualisation, knowledge representation languages, and knowledge extraction.

The largest subset of papers is concerned with general or specific knowledge engineering tools. Vaquero et al's review of tools and methods concentrates on reviewing support tools for the process of designing a planning domain model. Clements et al's work is concerned with the process of building models of real applications, and they describe a development environment for helping build such models. A paper by Grzes et al introduces a novel form of knowledge engineering for MDP planning, based on a relational database scheme, aimed specifically at applications in assistive technology. Two papers concentrate on translation tools - Seipp and Helmert show how to enact useful reformulation of planning problems by translating them to a multi-valued variable representation and then conducting a "fluent merging" technique, whereas Porco et al investigate the novel idea of translating problems of high computational complexity into STRIPS in order to utilise standard planners in their solution.

Several papers address the issue of extending existing domain model languages or finding new ones: Jonas's paper proposes a new domain modelling language based on Java, while Estler and Wehrheim's paper describes how to represent planning problems as graphs and perform planning using graph transformations. Two papers address HTN planning extension: Off and Zhang extend HTN planning to deal with worlds that require an open-world assumption, whereas Castillo et al extend HTN planning to make it more compatible with the world of IT, utilising graphical notations such as BPM and UML.

Plan visualisation is utilised by several authors: Vaquero et al in one paper describe an overall framework for encompassing post-design analysis in planning systems utilising visualisation; in another paper they show how during a post-design phase, plan rationals can be derived to drive model refinement. Glinsky

and Barták describe a tool being developed to visualise a plan for verification purposes, while Gerevini and Saetti's paper concentrates on using plan visualisation to drive a mixed-initiative planning process built around an existing domain independent planner.

A major theme in knowledge engineering concerns techniques which extract knowledge from existing domain models. Bernardini and Smith introduce a method for extracting useful invariants in temporal domains, while Wickler's paper defines domain model features which can be used to characterise the domain formulation, having potential for use in domain validation. Schmidt et al's paper describe a method to improve HTN planning utilising extracted domain knowledge, while Fujimura details a method using information extraction to aid production planning and scheduling.

We would like to thank to all contributors to this workshop and special thanks go to the members of program committee and other reviewers for their valuable comments.

Roman Barták, Simone Fratini, Lee McCluskey, Tiago Stegun Vaquero  
KEPS 2011 Organizers  
June 2011

## Contents

### Oral Presentations

<b>A Brief Review of Tools and Methods for Knowledge Engineering for Planning &amp; Scheduling .....</b>	<b>7</b>
Tiago Stegun Vaquero, José Reinaldo Silva, J. Christopher Beck	
<b>Acquisition and Re-use of Plan Evaluation Rationales on Post-Design.....</b>	<b>15</b>
Tiago Stegun Vaquero, José Reinaldo Silva, J. Christopher Beck	
<b>The Challenge of Grounding Planning in Simulation in an Interactive Model Development Environment .....</b>	<b>23</b>
Bradley J. Clement, Jeremy D. Frank, John M. Chachere, Tristan B. Smith, Keith Swanson	
<b>Finding Mutual Exclusion Invariants in Temporal Planning Domains.....</b>	<b>31</b>
Sara Bernardini, David E. Smith	
<b>Using Planning Domain Features to Facilitate Knowledge Engineering.....</b>	<b>39</b>
Gerhard Wickler	
<b>Fluent Merging for Classical Planning Problems.....</b>	<b>47</b>
Jendrik Seipp, Malte Helmert	
<b>Heuristic Search-Based Planning for Graph Transformation Systems .....</b>	<b>54</b>
H.-Christian Estler, Heike Wehrheim	

### Poster Presentations

<b>JPDL: A fresh approach to planning domain modelling .....</b>	<b>63</b>
Michael Jonas	
<b>Cooperated Integration Framework of Production Planning and Scheduling based on Order Life-cycle Management .....</b>	<b>71</b>
Shigeru Fujimura	
<b>Relational Approach to Knowledge Engineering for POMDP-based Assistance Systems with Encoding of a Psychological Model .....</b>	<b>77</b>
Marek Grześ, Jesse Hoey, Shehroz Khan, Alex Mihailidis, Stephen Czarnuch, Dan Jackson, Andrew Monk	
<b>Open-Ended Domain Model for Continual Forward Search HTN Planning .....</b>	<b>85</b>
Dominik Off, Jianwei Zhang	

<b>Taking Advantage of Domain Knowledge in Optimal Hierarchical Deepening Search Planning .....</b>	<b>93</b>
Pascal Schmidt and Florent Teichteil-Königsbuch, Patrick Fabiani	

<b>Automatic Polytime Reductions of NP Problems into a Fragment of STRIPS.....</b>	<b>101</b>
Aldo Porco, Alejandro Machado, Blai Bonet	

<b>A Conceptual Framework for Post-Design Analysis in AI Planning Applications.....</b>	<b>109</b>
Tiago Stegun Vaquero, José Reinaldo Silva, J. Christopher Beck	

## **System Demonstrations**

<b>An Interactive Tool for Plan Visualization, Inspection and Generation .....</b>	<b>118</b>
Alfonso E. Gerevini, Alessandro Saetti	

<b>An extended HTN knowledge representation based on a graphical notation .....</b>	<b>126</b>
Francisco Palao, Juan Fdez-Olivares, Luis Castillo and Oscar García	

<b>VisPlan – Interactive Visualisation and Verification of Plans .....</b>	<b>134</b>
Radoslav Glinský, Roman Barták	

# **Oral Presentations**

# A Brief Review of Tools and Methods for Knowledge Engineering for Planning & Scheduling

**Tiago Stegun Vaquero<sup>1</sup> and José Reinaldo Silva<sup>1</sup> and J. Christopher Beck<sup>2</sup>**

<sup>1</sup>Department of Mechatronics Engineering, University of São Paulo, Brazil

<sup>2</sup>Department of Mechanical & Industrial Engineering, University of Toronto, Canada  
 tiago.vaquero@usp.br, reinaldo@usp.br, jcb@mie.utoronto.ca

## Abstract

In this paper we present a brief overview of the Knowledge Engineering for Planning and Scheduling (KEPS) area in the light of a prospective design process of planning application models. The main discussion is based on the fact that KE is better introduced in the planning world through the design process, more than through the planning techniques. Thus, we examine the fundamental steps in the design process of AI planning domain models considering techniques and methods that have appeared in the research literature. We analyze design phases that have not been received much attention in practical planning literature.

## Introduction

The 25th. anniversary special issue of The Knowledge Engineering Review (KER) brings some reflection on the development of KE during this years and what would be the alternatives for the future (McBurney and Parsons 2011). We think that it would be interesting to briefly analyze in this paper the relationship between KE and planning and scheduling, specially, in the capability of the combined area (KE and P&S) of treating complex real problems. First of all it would be interesting to ask “to which point KE would be directed to better contribute to solve a planning problem?”. Clearly we can identify two main points where knowledge is involved in planning and scheduling: i) in the support of the planning algorithm, that is, in the underlying knowledge system used as problem solver; ii) in the modeling of the surrounding domain, and the explicit enunciation of the planning problem. Besides, expert knowledge about the domain can be used to provide better answers.

While much of the mainstream of AI planning has focused on developing and improving planning techniques, for almost twenty years there existed some research on design processes for planning that considers the special characteristics of this class of problem in which the knowledge management is particularly important (Allen, Hendler, and Tate 1990). The integration of the planning algorithms with the

design processes is clearly a vital, strategic goal for planning research. Both lines of work are essential, specially if real-life application is the final objective.

Knowledge engineering for planning has not yet reached the maturity of other traditional engineering areas (e.g., Software Engineering (Sommerville 2004)) to define a common sense design process for planning applications. Nevertheless, research in the planning literature has shown some discussion about the needs and singularities of such design process and life cycle (McCluskey et al. 2003; Simpson 2007; Vaquero et al. 2007). Some of these initiatives introduce techniques, methods and tools to support designers during the design life cycle.

In this paper we present an overview of the Knowledge Engineering for Planning and Scheduling (KEPS) area in the light of a hypothetical design process of planning application models. We present a review of tools and methods that address the challenges encountered in each phase of a design process. While examining reviewing the literature about the design cycle, we pinpoint those phases and aspects that have not been received much attention in practical planning literature. Our goal is to provide some inputs for new upcoming research on KEPS in order to address the challenges that have receiving least attention in the AI planning community. They will be important to the development of real planning and scheduling applications.

## Design Process of Planning Domain Models

Design process principles have become important to the success of the development and maintenance of real world planning applications. A well-structured design process increases the chances of building an appropriate planning application while reducing possible costs of fixing errors in the future. In this section we examine existing research on knowledge engineering for planning in the light of an prospective design process which derive some features from Software Engineering and Design Engineering fields and expert knowledge from the experience from real planning domain modeling (Vaquero et al. 2007). Such process follows



a partially ordered sequence of phases. The baseline phases are as follows:

1. **Requirements Specification:** the elicitation, analysis, and validation of requirements, potentially using a semi-formal approach and viewpoint analysis (Sommerville and Sawyer 1997).
2. **Knowledge Modeling:** the abstraction, modeling and re-use of the domain definition and the basic relationships within the planning problem.
3. **Model Analysis:** verification and validation of the domain model and the planning problem, as well as model enhancement.
4. **Deploying Model to Planner:** translation of the problem specification into a communication language understood by automated planners.
5. **Plan Synthesis:** interaction with one or more automated planning systems to create potential solutions to the planning problem.
6. **Plan Analysis and Post-Design:** analysis of the generated plans according to some metrics. New insights may be generated and added to the requirements as part of the overall, iterative design process.

Designing a real planning application following a pure theoretical approach can be often impractical and therefore research on real planning application has followed a more practical approach: developing tools to support the design process. Most of the work on KE for planning refers to tools that cover some of the above phases of the design (specific tools) while few of them try to cover the whole process (general tools). In what follows we will explain and analyze how the available tools approach each of the design phases.

In such analysis, we emphasize the characteristics of two of the most important general tools for KE for planning: GIPO (Simpson 2007) and itSIMPLE (Vaquero et al. 2007). GIPO is the pioneer tool for KE for planning that explicitly focus on the challenges of building a planning domain model and itSIMPLE has focused on a disciplined design process of real planning applications. itSIMPLE integrates a set of languages and tools to support the cyclic design process of a domain model, from a informal representation to a formal model. The KE tool itSIMPLE is the winner of the 3rd International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS) and has been designed by the author of this thesis. We also include other tools and methods that have been active on the planning community, especially in KE research. For example, we include tools such as ModPlan (Edelkamp and Mehler 2005) for modeling and analysis with PDDL, MrSPOCK (Cesta et al. 2008) verification of temporal and causal constraints, TIM (Fox and Long 1998; Cresswell, Fox, and Long 2002) and DISCOPLAN (Gerevini and Schubert 1998) for model analysis, VAL (Howey, Long, and Fox 2004) for plan validation, and a number of application-specific tools such as JABBAH (González-Ferrer, Fernández-Olivares, and Castillo 2009) for modeling and planning in business process.

## Requirements Specification

It is well-known in the software and systems world that the lack of a requirements phase can be a primary cause of difficulties in a project ranging from budget or schedule overruns to outright failure (Kotonya and Somerville 1996; Sommerville and Sawyer 1997). In real-life projects a clear identification and analysis of requirements is a key issue to the success of the project.

Real-life projects also have distinctive classes of users, stakeholders whose viewpoints (Sommerville and Sawyer 1997) must be combined and made consistent with the goals of the developer or designer. Therefore, a phase of requirement elicitation and analysis must not be missed. In the context of complex systems, the specifications are very unlikely to be provided at once, as components of the new system and even some basic requirements may not be known in the initial phase. Therefore, a requirements specification phase is generally divided in two steps:

1. knowledge acquisition or requirements elicitation.
2. analysis of the requirements to spot conflicts, omissions or misconceptions about the interaction between the planning systems and its surrounding environment.

The two pioneering works on knowledge acquisition in planning were O-Plan (Tate, Drabble, and Dalton 1996) and SIPE (Myers and Wilkins 1997). Both projects have developed tools that help in the knowledge engineering process. Designed for some specific applications, O-Plan introduced the “Common Process Method” while SIPE introduced the Act Editor.

One of the first domain-independent tools in literature for supporting knowledge acquisition is the system GIPO (Simpson 2007). Knowledge acquisition is performed in GIPO with visual interface that has been designed to assist the user by taking care of simple syntax details in order to avoid syntactically ill-formed model specifications (Simpson 2007). GIPO treats this phase as a large knowledge acquisition process and does not distinguish non-functional requirements from functional or operational aspects.

The system itSIMPLE (Vaquero et al. 2007; 2009) focuses not only on the initial phases of a design process but also on the design life cycle of planning applications. The tool provides an integrated environment that supports knowledge acquisition in its early stages. The work was one of the first to introduce the principles of requirements engineering to AI planning. itSIMPLE is more pragmatic with the use of a visual interface for *Elicitation* and *Analysis of Requirements*; it goes directly to the operational and functional aspects, as well as non-functional requirements, using initially a semi-formal representation language, the Unified Modeling Language (UML) (OMG 2005). UML is the most commonly used language for requirements representation in software engineering. Many engineers, working in different application areas, are familiar with this representation.

Because of its pragmatic approach, itSIMPLE is currently closer to users and stakeholders while GIPO embodies a more designer-oriented approach. The former environment goes directly to requirements analysis and validation while the latter encompasses all that in the knowledge analysis

and representation in Object Centered Language (OCL), a language created as part of the GIPO project to better capture the semantics inherent in planning applications (Simpson et al. 2000). Therefore, no external analysis is considered in GIPO while such analysis is an important feature in itSIMPLE via Petri Nets techniques (Silva and Santos 2003). Viewpoint analysis and requirements engineering approaches are lacking in GIPO whereas a sound manipulation for knowledge that are strong in GIPO are missing in itSIMPLE. Certainly both are necessary.

Regarding domain-dependent tools, the work from Bonasso and Boddy (2010) describes an ongoing project for eliciting planning information from the domain experts in order to support NASA operations personnel in planning and executing activities on the International Space Station (ISS). Aiming at the initial phase of the design, this work introduces a tool for gathering procedural requirements, including a) time, for both task duration and for temporal constraints among procedures, b) resources that are required, produced or consumed by a procedure, c) preconditions, post-conditions and other constraints for both a given procedure and among concurrently executing procedures, and d) the decomposition of large procedures into the fundamental actions used to build up a mission plan (Bonasso and Boddy 2010). With such planning information acquired using template forms, the goal is to generate actions specifications in a standard planning languages that automated planners can use. Even though the tool has a particular application, its method can be generalized to other domains.

## Knowledge Modeling

No real-life system is truly isolated. Any planning system is embedded in a real domain: a myriad of “non-system” tools, objects, people, and processes with which it must interact. It is necessary to have a model for this environment – and it is important that this model be developed independently of the planning system (McDermott 1981). The term model implies that we have a representation or a formalism that mirrors behaviors in the real domain. Such model representation must provide semantics, implying that named elements within it correspond directly to named elements in the real domain (McCluskey 2002). Moreover, since several planning problems could be related to the same or similar domains, reusability is a key issue in real-life planning systems and is a fundamental part of the modeling process. Reuse can speed up the design process by using well-tested model structures and elements from other successful applications.

GIPO utilizes the Object Centered Language (OCL) to model domain ontology, objects and actions. GIPO’s GUI allows the use of diagrams to support users defining such domain elements. For the domain ontology representation, GIPO (version IV) provides an editor to create “Concept Diagrams” in the style of UML class diagrams to define the kinds of objects and concepts, as well as their relationships. These diagrams also provide the opportunity to define properties that are common to all object instances of the various concepts (Simpson 2007).

The Life History editor from GIPO allows the user to

draw state machines that describe the dynamics of an object class. It can be used to name the states that object instances can occupy and to show the possible transitions between the states. In addition to the manual input of the action representation, users can be assisted by induction techniques to aid the acquisition of detailed operator descriptions. The operator description assistant, called *opmaker*, requires as input an initial structural description of the domain along with a training instance and a valid plan for that instance (Simpson 2007).

GIPO is the only tool that currently provides support for knowledge re-use in which the development of the action representation may involve the use of common design patterns of planning domain structures (called “Generic Types”). Of course, domain knowledge re-use and storage raise a number of issues such as when to create new reusable knowledge to avoid storing too many similar ones, or what is the impact on planner performance of a reusable component. At the moment a base of cases does not exist and these issues have consequently not yet been demonstrated in practice.

itSIMPLE provides a tool-set and methods to support designers during domain model creation through an object-oriented approach (Vaquero et al. 2007). The system uses the diagrammatic language UML (OMG 2005) for *Knowledge Acquisition* and *Modeling* processes. Modeling is made through UML diagrams, from a high level of abstraction (such as use case diagrams) to lower levels (e.g., class diagrams or state machine diagrams). The visual components provided by UML can make the planning domain modeling process friendly and can facilitate communication and analysis of requirements belonging to different viewpoints (e.g., stakeholders, planning domain experts, users).

Classes, properties, relationships, and constraints are defined by using class diagrams which represent most of static characteristics of a domain. Operators’ parameters and durations are also specified in the class diagram. The dynamics of operators (actions) are modeled by using UML state machine diagrams. These diagrams represent the states that a class object can enter during its life. One diagram is built for each class that has dynamic features. An action’s pre- and post-conditions are defined by using a formal constraint language of UML, called Object Constraint Language (OCL – a different “OCL” than used in GIPO) (OMG 2003), as part of operator representation. Planning problems are created by using object diagrams which represent snapshots of a planning domain, most commonly the initial state and the goal state. Users can also create preferences and constraints with object diagrams, for instance on the plan trajectory, which can capture either desirable or undesirable situations (Vaquero et al. 2007).

ModPlan (Edelkamp and Mehler 2005) is a planning workbook that provides some knowledge acquisition and modeling functionalities. The tool uses PDDL (version 2.2) as the base representation of the knowledge and it is aimed more at planning experts than designers with domain knowledge.

Bouillet et al. (2007) describe an ongoing knowledge engineering and planning framework that supports designers during construction of planning domains and problems

based on OWL ontologies (McGuinness and van Harmelen 2004). The state of the world is represented as a set of OWL facts, using a Resource Description Framework (RDF) graph, while actions are described as RDF graph transformations. Planning goals are described as RDF graph patterns. The framework allows the creation of planning domain through OWL ontologies extension in a collaborative manner. The framework (2007) provides a certain re-use capability by the general concept of ontology, but not as explicitly as GIPO. As mentioned by Bouillet et al. (2007), the use of OWL ontologies as a basis for modeling domains allows the re-use of existing knowledge in the Semantic Web. While the framework has been applied towards composing workflows in stream processing systems, it can be seen as a general tool and could, therefore, be applied in other planning domains (Bouillet et al. 2007).

Inspired by GIPO and itSIMPLE, Vodrazka and Chrpá (2010) created a compact modeling tool for planning as an attempt to simplify the model construction process to non-planning experts. Unlike GIPO and itSIMPLE, the tool called VIZ uses straightforward approach to model simple STRIPS-like domains. The tool provides a graphical interface that uses uniquely simple (non-standard) diagrams to capture action specifications, objects and their relations.

Besides the general purpose tools, as those considered above, there are also research on specific knowledge applications. For example, JABBAH (González-Ferrer, Fernández-Olivares, and Castillo 2009) is a KE tool dedicated to Business Process Modeling (BPM). This tool is able to support modeling and representation of business process models in order to use planners to obtain action plans for task management.

## Model Analysis

The definition of a suitable planning domain is related to the possibility of analyzing the model during the design phases. Contrasting features of the model being built and the acquired requirements becomes very important in non-trivial planning applications. Model analysis encompasses verification, validation, knowledge enhancement and refinement of the entire model (McCluskey 2002). Generally, the analysis, performed manually, automatically or system-assisted, focuses on two main aspects: the static and dynamic properties. Finding errors, inconsistencies and incoherences in such properties can save time and resources in posterior phases.

Static analysis essentially investigates whether the model is self-consistent. Such analysis can range from simple syntax checkers and debuggers to cross-validation of different parts of the model, particularly for those models containing a set of diagrams or representation schemes. For example, static analysis can be applied to verify the definition of types of objects, constraints, state definition, and other static model elements.

Dynamic analysis entails validating whether the behavior of modeled actions is consistent to the requirements and to what is expected by humans. That involves the examination of how actions interact with each other and how they are executed. Both static and dynamic analysis can be made independently of the planner.

Most real-life planning problems require the investigation and enhancement of specific knowledge, acquired during analysis, in order to achieve reliable planner performance and high plan quality. Some of this specific knowledge may take the form of heuristics or domain control knowledge that can be used to guide planners in finding an efficient plan. Moreover, knowledge enhancement may be concerned with the inclusion of design decisions, reasons, and justifications (i.e. rationales) in the specification process and documentation, which supports the maintenance of complex projects. Klein (1993) explains how rationale are important to engineering design projects for airplane parts.

Few methods and tools are available in the planning literature that can deal with domain analysis. As described by McCluskey (2002), because AI planning has been largely in the realm of research, many researchers in the past used nothing more than basic syntax checkers in support of their model analysis process. However, this approach is neither sufficient nor efficient in large models. Inspired by these large applications, recent research has introduced more elaborated domain analysis techniques.

GIPO (Simpson 2007) checks local and global model consistency such as object class hierarchy consistency, object state descriptions invariants satisfaction, mutual consistency of predicate structures, and others. This static validation can uncover potential errors within a domain specification. In addition to static analysis, GIPO provides a visual representation of dynamic behavior for analysis, a combination of state-machine-like diagrams to show how objects of two or more concept types coordinate their dynamic movements (Simpson 2007). GIPO allows designers to check the model against a set of problems by using a *stepper*. The *stepper* provides the manual selection of actions state-by-state to verify their applicability and to validate the dynamic part of domain model (Simpson 2007). Knowledge discovery and enhancement during domain analysis is not provided by GIPO.

itSIMPLE (Vaquero et al. 2007) provides a rich graphical interface where different viewpoints can be used to validate or criticize a model. Users are supported while creating coherent diagrams to avoid modeling mistakes. For example, snapshots are created based on class diagrams and all constraints defined on them. The tool can check each snapshot for coherence in order to avoid inconsistent states. For dynamic analysis, the environment uses Petri Nets (Murata 1989), a formal representation for dynamic domain validation deploying visual and animated information of the entire system based on the UML state machine diagrams. However, the approach with Petri Net is not fully implemented and tested. itSIMPLE has no support for knowledge enhancement, trusting the requirement validation process to provide insight into improving the problem description and also, based on the direct intervention of the designer, to provide heuristics to guide the planner.

In the literature there is a large variety of techniques for knowledge extraction during domain analysis, but most of them are dependent of the planning system (McCluskey 2002). The extraction of properties such as types, invariants, strategies, heuristics, or subproblems can be a way to en-

hance models with essential information to be used during the planning process. Systems such as TIM (Fox and Long 1998; Cresswell, Fox, and Long 2002) and DISCOPLAN (Gerevini and Schubert 1998) find types and state invariants while RSA (Scholz 1999) and RedOp (Haslum and Jonsson 2000) find different types of constraints on which action sequences are necessary or relevant for solving a given problem. Moreover, the work described in (Fox and Long 1999; Crawford et al. 1996) introduces detection of symmetry as additional knowledge to improve planners' performance.

### Deploying Model to Planner

Standard planners cannot directly parse arbitrary specification languages. Enabling them to do so would require a large amount of effort which is, at best, peripheral to the interests of the researchers, mainly directed to the implementation of AI planning algorithms. Therefore, it is reasonable to develop a unified communication language embodying a convergence of goals and computational effort.

At present, PDDL (Fox and Long 2003) works as such a language even though it was not explicitly designed with that purpose in mind. Thus, existing tools and integrated environments should be able to translate specifications to a communication language without any (or minimum) loss in the problem specification. The communication language must, therefore, have the same expressive power as the specification language. Since expressiveness issues are not in the scope of this thesis, we do not go into that discussion.

Considering general purpose tools, GIPO, itSIMPLE and VIZ have sound and efficient mechanisms to translate their respective front-end languages to PDDL. GIPO translates its OCL domain model to PDDL 2.2 (Simpson 2007) while itSIMPLE transfers the knowledge in UML to a solver-ready PDDL model up to version 3.1 (the latest version of PDDL) (Vaquero, Silva, and Beck 2010). VIZ translates simple diagrams into a STRIPS-like PDDL model (Vodrazka and Chrapa 2010).

Regarding specific tools, JABBAH translates a business process model into a solver-ready representation, in this case the Hierarchical Task Network (HTN) (González-Ferrer, Fernández-Olivares, and Castillo 2009). Fernández et al. (2009) describe an approach to represent data-mining processes, using Predictive Model Markup Language (PMML), in terms of automated planning and translate the data mining tasks into PDDL. The tool PORSCHE II (Hatzel et al. 2009), while focusing on semantic description of web services, provides a translation process from OWL-S to PDDL to make the domain available for planners.

### Plan Development

In this phase, plans are produced by planning algorithms based on knowledge specified and modeled in the domain model. This phase is where most research work on AI planning are focused. However, instead of focusing on planning techniques (see (Ghallab, Nau, and Traverso 2004) for details on techniques) we look at the research on KE tools that give support and facilitate the use of planning algorithms, by an integrated environment that support different planners

- using distinct AI planning approaches - and by including features to communicate and visualize the resulting plans.

The most significant work on this phase is itSIMPLE (Vaquero et al. 2009; Vaquero, Silva, and Beck 2010). As an integrated environment, itSIMPLE uses PDDL to communicate automatically with solvers, including Metric-FF, FF, SGPlan, MIPS-xxl, LPG-TD, LPG, hsp, SATPlan, Plan-A, Blackbox, MaxPlan, LPRPG, and Marvin. In fact, the tool allows new planners to be easily added. This feature gives to itSIMPLE the flexibility to exploit recent advances in solver technology. Designers can test different planning approaches on their model and identify the most promising one. Other tools like GIPO, ModPlan, JABBAH and VIZ do not have such extensive connection with planners.

### Plan Analysis and Post-Design

Because of the characteristics of models, it is likely that some problem instances, domains, and models will be better suited to the one planning algorithm rather than another. Furthermore, for complex problems, the lack of knowledge or ill-defined requirements and metrics could propagate to specifications and from there to the problem submitted to the planner. Either of these scenarios (and others) may lead to the generation of poor quality plans. Regardless, bad plans must be spotted and fixed.

A last fundamental step in the design cycle is the analysis of generated plans with respect to the requirements and quality metrics. Plan analysis naturally leads to feedback and the discovery of hidden requirements for refining the model, giving consequently the capacity of improving the quality of generated plans. We call '*post-design analysis*' the process performed after plan generation, in which we have a base model and a set of planners to investigate the solutions generated by them. Such a post-design process requires approaches that range from simple plan validation and visualization to a more sophisticated treatment based on metrics. Such a treatment should be able to evaluate the plan and to relate defects to a set of requirements or even to a lack of such requirements. What may be produced is a new insight and a need to change requirements which can be used in the next design iteration.

AI planning research on plan analysis has developed tools and techniques mostly for plan validation, plan visualization (e.g., diagrams and Gantt charts), animation, plan querying and summarization. In (Smith and Holzmann 2005), formal verification is used in order to check the existence of undesirable plans with respect to the domain model. The work from Howey, Long, and Fox (2004) describes the system VAL, a plan validation tool for PDDL. Given an input plan in PDDL syntax and its respective domain model and problem, VAL validates action execution while recognizing if the plan does not reach the specified goal or if it contains invalid sequence of actions. VAL has recently been extended to capture most of PDDL features (up to version 3.0) such as continuous effects, exogenous events and process handling.

GIPO provides a simple plan visualizer (*animator*) to allow a graphical view of successful plans. Recent publications (McCluskey and Simpson 2006) describes such visualization tool. In contrast with GIPO, itSIMPLE starts from

basic visualization and simulation of plans to a more sophisticated simulation interaction and analysis of domain variable (Vaquero et al. 2007). With a flexible interface to a large set of planners, it is possible to analyze plans produced by different planning techniques. The tool supports plan evaluation through a functionality called “Variable Tracking”, which allows analysis based on variable observation or quality metrics displayed on charts. The functionality called “Movie Maker” provides a simulation and a visualization of plans through a sequence of UML object diagrams, snapshot-by-snapshot. A minimal interaction with the simulator is allowed where new actions can be added or removed to check different situations. The plan analysis tools provided by itSIMPLE aim to help designer adjust models by observing plans being executed in diagrammatic form. However, the use of these diagrams prevent the proper analysis of large-scale problems.

ModPlan integrates VAL for plan validation (Edelkamp and Mehler 2005) and, for plan visualization, it includes the animation system Vega (Hipke 2000) allowing a magnification to an arbitrary part of the plan. Gantt charts are plotted for temporal plans, in which a horizontal open oblong is drawn against each activity indicating estimated duration. Plan animation is assisted by users and is provided for some benchmark domains. JABBAH also shows output plans for business processes using Gantt diagram (González-Ferrer, Fernández-Olivares, and Castillo 2009).

The work from Haas and Havens (2008) introduces a specific dynamic plan simulator for the Canadian CoastWatch project. CoastWatch is an oversubscribed dynamic multi-mode problem with unit resources and lies in the Search & Rescue domain. CoastWatch datasets simulate a typical day for the Canadian Coast Guard, where officers assign resources (planes, helicopters, ships) to execute several different kinds of missions (rescue, patrol, transport). The dynamic simulator includes a visualization tool which creates an animation of the planning and scheduling problem on GoogleEarth™. The animation steps through the scheduling horizon and visualizes the different entities in action. Such application-dependent simulator creates a good communication channel between project participants.

Aiming at plans with rich sophistication and complexity, Myers (2006) describe a domain-independent framework for plan summarization and comparison that can help a human user to understand both key strategic elements of an individual plan and important differences among plans. The goal of such summarization and comparison is to analyze the relative merits of various plan candidates before deciding on a final option. The approach is grounded in a domain metatheory, which specifies important semantic properties of tasks, actions, planning variables, and plans. This work defines three capabilities grounded in the metatheoretic approach: (1) summarization of an individual plan, (2) comparison of pairs of plans, and (3) analysis of a collection of plans. The approach has the benefit of framing summaries and comparisons in terms of high-level semantic concepts, rather than

low-level syntactic details of plan structures and derivation processes. As reported by Myers (2006), application of these capabilities within a rich application domain facilitates user understandability of complex plans.

Giuliano and Johnston (2010) proposes a visualization tool for multi-objective problems in space telescope control systems that helps users while selecting schedules to be executed. The tool supports schedule analysis by keeping user objectives separated instead of combined to make trade-offs between competing objectives. The analysis is done through charts and graphs to explore the different aspects of distinct schedules. In a similar direction, Cesta et al. (2008) describe the MrSPOCK system able also to validate schedules and illustrate trade-offs of space mission plans. These two works emphasize how important and difficult it is to work with different criteria coming from distinct groups in real problems.

The main focus of works like Myers (2006), Cesta et al. (2008) and Giuliano and Johnston (2010) is on helping users to (1) better understand the underlying properties of generated plans and schedules and (2) to select the most appropriated solutions to be executed. In fact, re-modeling, or the refinement cycle, seems not be the main target for most of plan analysis tools in AI planning literature. In addition, acquiring valuable information during analysis process itself is not the main goal either.

## Discussion

As defined by Fox (2011), knowledge engineering is “a discipline that involves integrating knowledge into computer systems in order to solve complex problems, normally requiring a high level of human expertise”. Therefore, KE is connected with solving real problems. In what follows we look at the tools and its underlying knowledge systems and methods in this perspective, that is, by the contribution they might have to the integration of knowledge into computer systems that solve real problems.

As we mentioned before, the contribution of these tools could appear: i) in the knowledge system underlying algorithms encapsulated in planners; ii) in the knowledge representation methods applied to the domain environment; iii) in the knowledge about the design process. Here, we focus the discussion on the last two kinds of contribution (since the present analysis does not include planners).

Concerning the knowledge representation methods applied to the domain we should point the following:

1. none of the tools treat differently the knowledge encapsulated in the planning problem and in the surrounding domain. Such a distinction could be valuable since in several cases it is required to solve different instances of the planning problem. All of those instances have the same surrounding domain and such knowledge could be reused.
2. the general problem of reusing knowledge is not explored besides some tools as GIPO where an attempt were made by using design patterns.
3. there is no attempt in any tool to investigate the relationship between knowledge domain and the underlying knowledge system that supports planners. That implies

that there is no way to find a “best planner” to a given planning domain. Some tools (like itSIMPLE) address at least a pool of planners where the result could be compared, but that is far from solving the matching issue.

Regarding the design process, the challenging points are:

1. there is not a referential design process for planning and scheduling applications. At most, general phases are adapted from a General Design Theory (Tomiya 1994) or more recent developments of that. Therefore, it is not clear if formal methods such as Process Algebra, Petri Nets, and others would help or improve the process.
2. in all tools, few attention was given to requirement analysis which is important to any real problem. Certainly it is not possible to assume that the planning problem could be synthesized ad hoc, without a requirement gathering and analysis.
3. once the planning problem is defined, it is necessary to provide a model of the overall domain that could be formally verified. In that point it is important to notice that to cover scheduling a formal time verification is required. Unfortunately there is not a long list of time formalisms that could be applied for that.
4. all previous topics are concerned with what could be called the design phases, that is, all phases that culminate with a design model for the domain and planning problem. However the design process could be accelerated if a post-design process is added to feed the refinement cycle, just when the top down refinement no longer improves the model significantly.

Besides all these points another important issue is that there is not enough effort to support scheduling for several reasons, where the most significant is the lack of a time formalism that could be more practical to be inserted in software tools. In addition, there has been not enough effort to support resource reasoning.

## Conclusion

In this paper, we presented a brief overview of tools and methods on knowledge engineering for planning considering a design process of planning domain models. We reviewed existing tools that support each phase of such design process, especially the plan analysis and post-design which drive re-modeling and refinement. The intention of this paper is to raise a discussion about where we are on the use of knowledge engineering for planning and scheduling and to give some inputs about where we want to go as research community in this area.

## References

- Allen, J. F.; Hendler, J.; and Tate, A. 1990. *Readings on Planning*. San Mateo, Ca., USA: Morgan-Kaufman.
- Bonasso, P., and Boddy, M. 2010. Eliciting Planning Information from Subject Matter Experts. In *Proceedings of ICAPS 2010 Workshop on Scheduling and Knowledge Engineering for Planning and Scheduling (KEPS)*, 5–12.
- Bouillet, E.; Feblowitz, M.; Liu, Z.; Ranganathan, A.; and Riabov, A. 2007. A Knowledge Engineering and Planning Framework based on OWL Ontologies. In *Proceedings of the Second International Competition on Knowledge Engineering*.
- Cesta, A.; Finzi, A.; Fratini, S.; Orlandini, A.; and Tronci, E. 2008. Validation and verification issues in a timeline-based planning system. In *Proceedings of the ICAPS 2008 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.
- Crawford, J.; Ginsberg, M.; Luks, E.; and Roy, A. 1996. Symmetry-breaking predicates for search problems. In *Fifth International Conference on Principles of Knowledge Representation and Reasoning*, 148–159. Cambridge, Massachusetts: Morgan Kaufmann.
- Cresswell, S.; Fox, M.; and Long, D. 2002. Extending tim domain analysis to handle adl constructs. In *Knowledge Engineering Tools and Techniques for AI Planning: AIPS'02 workshop*.
- Edelkamp, S., and Mehler, T. 2005. Knowledge acquisition and knowledge engineering in the modplan workbench. In *Proceedings of the First International Competition on Knowledge Engineering for AI Planning*.
- Fernández, S.; Fernández, F.; Sánchez, A.; de la Rosa, T.; Ortiz, J.; Borrajo, D.; and Manzano, D. 2009. On Compiling Data Mining Tasks to PDDL. In *Proceedings of the Third International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS)*, ICAPS 2009, 8–17.
- Fox, M., and Long, D. 1998. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research* 9:367–421.
- Fox, M., and Long, D. 1999. The detection and exploitation of symmetry in planning problems. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*, 956–961. Stockholm, Sweden: Morgan Kaufmann.
- Fox, M., and Long, D. 2003. PDDL2.1: An extension of PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)* 20:61–124.
- Fox, J. 2011. Formalizing knowledge and expertise: where have we been and where are we going? *The Knowledge Engineering Review* 26(1):5–10.
- Gerevini, A., and Schubert, L. 1998. Inferring state constraints for domain-independent planning. In *Proceedings of 15th National Conference on Artificial Intelligence*, 905–912. Madison, USA: AAAI Press/MIT Press.
- Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. San Francisco, CA, USA: Morgan Kaufman.
- Giuliano, M. E., and Johnston, M. D. 2010. Visualization Tools for Multi-Objective Scheduling Algorithms. In *Proceedings of ICAPS 2010 System Demonstration*, 11–14.
- González-Ferrer, A.; Fernández-Olivares, J.; and Castillo, L. 2009. JABBAH: A Java Application Framework for the Translation Between Business Process Models and HTN. In *Proceedings of the Third International Competition on*

*Knowledge Engineering for Planning and Scheduling (ICK-EPS)*, ICAPS 2009, 28–37.

Haas, W., and Havens, W. S. 2008. Generating Random Dynamic Resource Scheduling Problems. In *ICAPS 2008 Workshop on Knowledge Engineering for Planning and Scheduling*.

Haslum, P., and Jonsson, P. 2000. Planning with reduced operator sets. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling Systems (AIPS)*, 150–158. Breckenridge, CO: AAAI Press.

Hatzi, O.; Meditskos, G.; Vrakas, D.; Bassiliades, N.; Anagnostopoulos, D.; and Vlahavas, I. 2009. PORSCE II: Using Planning for Semantic Web Service Composition. In *Proceedings of the Third International Competition on Knowledge Engineering for Planning and Scheduling*, 38–45.

Hipke, C. A. 2000. *Distributed Visualization of Geometric Algorithms*. Phd thesis, University of Freiburg.

Howey, R.; Long, D.; and Fox, M. 2004. Val: Automatic plan validation, continuous effects and mixed initiative planning using pddl. In *ICTAI'04: Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence*, 294–301. Washington, DC, USA: IEEE Computer Society.

Klein, M. 1993. Capturing design rationale in concurrent engineering teams. *IEEE Computer* 26(1):39–47.

Kotonya, G., and Somerville, I. 1996. Requirements engineering with viewpoints.

McBurney, P., and Parsons, S. 2011. *The Knowledge Engineering Review*, volume 26 - Special Issue 01 (25th Anniversary Issue). Cambridge University Press.

McCluskey, T. L., and Simpson, R. M. 2006. Tool support for planning and plan analysis within domains embodying continuous change. In *Proceedings of the ICAPS 2006 Workshop on Plan Analysis and Management*.

McCluskey, T.; Aler, R.; Borrajo, D.; Haslum, P.; Jarvis, P.; Refanidis, I.; and SCHOLZ. 2003. Knowledge engineering for planning roadmap.

McCluskey, T. L. 2002. Knowledge Engineering: Issues for the AI Planning Community. *Proceedings of the AIPS-2002 Workshop on Knowledge Engineering Tools and Techniques for AI Planning*. Toulouse, France 1–4.

McDermott, J. 1981. Domain knowledge and the design process. In *Proceedings of the 18th Conference on Design Automation*, 580–588. Piscataway, NJ, USA: IEEE Press.

McGuinness, D. L., and van Harmelen, F. 2004. OWL Web Ontology Language Overview. W3C recommendation.

Murata, T. 1989. Petri nets: Properties, analysis and applications. In *Proceedings of the IEEE*, volume 77, 541–580.

Myers, K. L., and Wilkins, D. 1997. The Act-Editor User's Guide: A Manual for Version 2.2.

Myers, K. L. 2006. Metatheoretic Plan Summarization and Comparison. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS-06)*. Cumbria, UK: AAAI Press.

OMG. 2003. *OMG Unified Modeling Language Specification - Object Constraint Language, Version 2.0*.

OMG. 2005. *OMG Unified Modeling Language Specification, Version 2.0*.

Scholz, U. 1999. Action constraints for planning. In *Biundo & Fox*, 148–160. Berlin, Heidelberg: Springer Verlag.

Silva, J. R., and Santos, E. A. 2003. Viewpoint requirements validation based on petri nets. In *Proceedings of the 17th Int. Conf. of Mechanical Engineering, Brazilian Mechanical Eng. Society*.

Simpson, R. M.; McCluskey, T. L.; Liu, D.; and Kitchin, D. 2000. Knowledge Representation in Planning: A PDDL to OCLh Translation. In *In Proceedings of the 12th International Symposium on Methodologies for Intelligent Systems*. Charlotte, North Carolina, USA: Springer.

Simpson, R. M. 2007. Structural Domain Definition using GIPO IV. In *Proceedings of the Second International Competition on Knowledge Engineering for Planning and Scheduling*.

Smith, M. H., and Holzmann, G. J. 2005. Model Checking Autonomous Planners: Even the best laid plans must be verified. In *Aerospace, 2005 IEEE Conference*, 1–11. IEEE Computer Society.

Sommerville, I., and Sawyer, P. 1997. Viewpoints: Principles, Problems and a Practical Approach to Requirements Engineering. *Annals of Software Engineering* 3:101–130.

Sommerville, I. 2004. *Software Engineering (7th Edition)*. Pearson Addison Wesley.

Tate, A.; Drabble, B.; and Dalton, J. 1996. O-Plan: a Knowledge-Based Planner and its Application to Logistics. In *Advanced Planning Technology ARPI*, 259–266. AAAI Press.

Tomiyama, T. 1994. From general design theory to knowledge-intensive engineering. *Artificial Intelligence for Engineering, Design, Analysis and Manufacturing* 8:319–333.

Vaquero, T. S.; Romero, V.; Tonidandel, F.; and Silva, J. R. 2007. itSIMPLE2.0: An integrated Tool for Designing Planning Environments. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS 2007)*. Providence, Rhode Island, USA: AAAI Press.

Vaquero, T. S.; Silva, J. R.; Ferreira, M.; Tonidandel, F.; and Beck, J. C. 2009. From Requirements and Analysis to PDDL in itSIMPLE3.0. In *Proceedings of the Third International Competition on Knowledge Engineering for Planning and Scheduling, ICAPS 2009*, 54–61.

Vaquero, T. S.; Silva, J. R.; and Beck, J. C. 2010. Improving Planning Performance Through Post-Design Analysis. In *Proceedings of ICAPS 2010 workshop on Scheduling and Knowledge Engineering for Planning and Scheduling (KEPS)*, 45–52.

Vodrazka, J., and Chrpá, L. 2010. Visual design of planning domains. In *Proceedings of ICAPS 2010 workshop on Scheduling and Knowledge Engineering for Planning and Scheduling (KEPS)*, 68–69.

# Acquisition and Re-use of Plan Evaluation Rationales on Post-Design

Tiago Stegun Vaquero<sup>1</sup> and José Reinaldo Silva<sup>1</sup> and J. Christopher Beck<sup>2</sup>

<sup>1</sup>Department of Mechatronics Engineering, University of São Paulo, Brazil

<sup>2</sup>Department of Mechanical & Industrial Engineering, University of Toronto, Canada  
 tiago.vaquero@usp.br, reinaldo@usp.br, jcb@mie.utoronto.ca

## Abstract

In this article we investigate how knowledge acquired during a plan analysis phase can be represented, stored, and re-used to support the identification and evaluation of potential adjustments to a domain model. We describe a post-design framework that combines a knowledge engineering tool and an ontology-based reasoning system for the acquisition and re-use of human-centric rationales for plan evaluations. We aim at rationales that represent information about (1) why a given plan is classified as good or bad, (2) what are the properties of the plan that directly impact its quality, and (3) how these properties affect the plan quality, positively or negatively. This paper shows a case study, based on a benchmark problem, which illustrates the process of development and acquisition of rationales.

## Introduction

Decisions about knowledge modeling and planning algorithm development drastically affect the quality of plans. From a planning technology perspective, in a *ceteris paribus* scenario, factors such as the improper choice of planning techniques and heuristics may lead to the generation of poor solutions. From a knowledge engineering perspective, lack of knowledge, ill-defined requirements and inappropriate definition of metrics, constraints and preferences can contribute directly to malformed models and, consequently, to unsatisfactory plans, independent of the planning algorithm. Traditionally, much of planning research has focused on a perspective in which new algorithms are developed and tuned to obtain high performance and better plans. Not much investigation has been done from the knowledge engineering (KE) perspective, especially re-modeling and refining the planning problem based on observations and information that emerge during the design process itself.

In plan analysis, hidden knowledge and requirements captured from human feedback raise the need for a continuous re-modeling process. The capture, representation and use of such human-centered feedback is still an unexplored area in the knowledge engineering for AI planning. Moreover, the impact of such feedback and re-modeling on the planning performance is unknown.

In (Vaquero, Silva, and Beck 2010), we propose of a post-design tool for AI planning that combines the open-source

KE tool itSIMPLE (Vaquero et al. 2007) and a virtual prototyping environment called Blender to support the short-term identification of inconsistencies and hidden requirements. In that work, there is a first attempt to manually interpret and insert rationales to support design decisions in a model adaptation approach. However, the proposed tool did not provide a process to capture, store and evaluate rationales. Meanwhile, the discussion about the use of rationales has grown very fast specially in the software engineering community, guiding the process of documentation and reuse (Daughtry et al. 2009).

In this paper, we present a post-design tool for AI planning that addresses the acquisition and re-use of human-centered rationales for plan evaluations. We aim at rationales that describe the reasons behind the plan classification (e.g., bad or good quality) given by designers or users. Such rationales describe what are the properties of the plan that impact its quality and how these properties affect the plan quality: positively or negatively. We study an approach that combines the open-source KE tool itSIMPLE (Vaquero et al. 2007) and an ontology-based reasoning system to support the capture, representation and re-use of rationales during plan evaluation. The aim of rationale re-use is to present to the human planner the plan properties and elements that are likely to impact its quality. This information becomes a starting point for the evaluation of a newly generated plan. In addition, the tool contributes to the continuous discovery process of hidden requirements (e.g., constraints and preferences) that were not identified during the virtual prototyping phase but can be available from user evaluation and justification.

This paper is organized as follows. First, we discuss concepts in knowledge engineering for planning and their role in plan analysis and model adaptation – processes that are generally performed in a post-design phase. We then focus on the contribution of rationales to plan design and life cycle, that is, the capture, analysis and re-use of rationales. Next, we present a case study based on the benchmark planning problem called Gold Miner. Following there is a discussion of the results and some concluding remarks.

## Knowledge Engineering and Post-Design

Requirements engineering (RE) and knowledge engineering (KE) principles have become important to the suc-



cess of the design and maintenance of real world planning applications (McCluskey 2002). While pure AI planning research focuses on developing reliable planners, KE for planning research focuses on the design process for creating reliable models of real domains (McCluskey 2002; Vaquero et al. 2007). A well-structured life cycle to guide design increases the chances of building an appropriate planning application while reducing possible costs of fixing errors in the future. A simple design life cycle is feasible for the development of small prototype systems, but fails to produce large, knowledge-intense applications that are reliable and maintainable (Studer, Benjamins, and Fensel 1998).

Research on KE for planning and scheduling has created tools and techniques to support the design process of planning domain models (Vaquero et al. 2009; Simpson 2007). However, given the natural incompleteness of the knowledge, practical experience in real applications such as space exploration (Jónsson 2009) has shown that, even with a disciplined process of design, requirements from different viewpoints (e.g. stakeholders, experts, users) still emerge after plan generation, analysis, revision and execution (Hatzi et al. 2010). For example, the identification of unsatisfactory solutions and unbalanced trade-offs among different quality metrics and criteria (Jónsson 2009; Rabideau, Engelhardt, and Chien 2000; Cesta et al. 2008) indicates a lack of understanding of requirements and preferences in the model. These hidden requirements raise the need for iterative re-modeling and tuning process. In some applications, finding an agreement or a pattern among emerging requirements is an arduous task (Jónsson 2009), making re-modeling a non-trivial process.

A fundamental step in the modeling cycle is the analysis of generated plans with respect to the requirements and quality metrics. Plan analysis naturally leads to feedback and the discovery of hidden requirements for refining the model. We call ‘*post-design analysis*’ the process performed after plan generation, in which we have a base model and a set of planners that provide the solutions to be evaluated. In fact, literature on plan analysis has shown interesting tools and techniques for plan animation (McCluskey and Simpson 2006; Vaquero et al. 2007), visualization (e.g. Gantt charts), virtual prototyping (Vaquero, Silva, and Beck 2010), and plan querying and summarization (Myers 2006).

Unfortunately, visualization and simulation approaches, such as the virtual prototyping used in our previous work (Vaquero, Silva, and Beck 2010), can not assure that all missing knowledge will emerge, specially in real planning problems. In many real cases, user feedback and rationales are hard to understand and compile; they are captured in pieces over time, making patterns hard to be identified. Analyzing plans individually in such real cases will probably not correctly emphasize the hidden requirements; they must be captured, pieced together and recognized. The accumulation of data from plan evaluation and their respective rationales can serve as a foundation for the identification of domain knowledge that cannot clearly or easily be detected during the first plan analysis interactions with visualization techniques. In this work, we want to go a step further on such investigation of post-design. We focus on studying the

capture, representation of human-centric feedback from plan evaluation, in the form of rationales, and the reuse of such rationales for further evaluations.

## Rationales in Planning

In software engineering, a design rationale is essentially the explicit recording of the issues, alternatives, tradeoffs, decisions and justifications that were relevant to the elements in the design. Rationales can be used in a number of ways in the design of an artifact:

- to explore and evaluate the various design alternatives discussed during the design process.
- to determine the changes that are necessary to modify a design.
- to facilitate better communication among people who are involved in the design process.
- to assist in making decisions during the design process.
- in design verification, to check if the artifact/product reflects what the designers and the users actually wanted.
- to re-use past experiences and to avoid the same mistakes made in the previous design.

Requirements engineering research has already reported the importance of rationale-based approaches; they have provided improvements in quality and reduction in costly errors that outweigh the costs of capturing rationales (Ramamash and Dhar 1994).

In planning literature, rationale has been generally referred to the “why a plan is the way it is”, and to “the reason as to why the planning decisions were taken” (Polyank and Austin 1998). These rationales, usually called *plan rationales*, have been recognized as an important type of information (Wickler, Potter, and Tate 2006; Polyank and Austin 1998) that can influence not only the plan synthesis process but the whole life cycle of a plan. In such life cycle, plan rationales can be acquired and used in the plan synthesis process itself, or in plan analysis, evaluation, explanation, plan indexing and retrieval, failure recovery, and plan communication.

Most of the work on plan rationales focuses on capturing and using them to improve the plan generation. The existing approaches of capturing plan rationales are related to the identification of planning decisions made by the planners (e.g., rationales in the form of causality, dependency) that stem from the planning process itself (planning trace). As an example of plan rationale related to a planner decision might be “action *A* is chosen at the state *S* because it achieves goal *g*” or “because *A*’s effects match an open condition of partial plan *p*”. These planning decisions are usually analyzed and re-used for making further similar decisions. For example, the usefulness of storing plan rationales to help future planning has been demonstrated by several types of case-based planners (Upal and Elio 1999). The case-based approach proposes that each planning decision within a plan be annotated with a rationale for making that decision. In this case, the planners remember past planning solutions and failures so they can be re-used or avoided in the future. The

idea behind storing these types of rationales is that a previously made and retrieved planning decision will only be applied in the context of the current planning problem if the rationales for it also holds in the current problem (Upal and Elio 1999). The work of Wickler, Potter, and Tate (2006) describes a recording process of rationales into a plan ontology in which a planner can record the justifications for including components into the plan represented in the <I-N-C-A> ontology within the framework called I-X. In addition, research on plan rationales has also focused on learning and using such information to produce control knowledge or plan-refinement strategies, which would, as a result, improve the plan quality. A good review of plan rationales is provided in (Polyank and Austin 1998).

Design and decision rationales coming from people have received the least amount of attention in the planning literature. Differently from existing work on rationales in planning, we focus on acquiring human-centric rationales that emerge from user feedback, observations and justifications during plan evaluation. Based on general and individual criteria, interests, feelings and expectations, the rationales from plan evaluation generate explanations and justifications as to why a plan was classified into a specific quality level. Therefore, we extend here the concept of plan rationales with rationales that encompass “why a certain plan element does or does not satisfy a criterion” or “why a certain plan does or does not satisfy a preference”. Moreover, these rationales could explain “why a certain metric does or does not satisfy a given criterion” and “what is the effect of a given plan characteristic or element in the plan quality” (e.g., it decreases or increases the quality). As an example of plan rationale, one might say that “the plan has a decreased quality because the robot left a block too close to the edge of the table” or “the plan has a high quality because the robot avoided repeatedly passing through the two most crowded areas of the building while cleaning it”. We call these explanations *plan evaluation rationales*.

In this paper we focus on the implementation of the processes related to plan evaluation and the acquisition and re-use of rationales. We have designed a framework called *Post-Design Application Manager* (postDAM) that integrates itSIMPLE and a reasoning system called tuProlog<sup>1</sup> (a Java implementation of the Prolog engine). The implemented framework supports users on the following processes: classification of metrics and plans (plan evaluation), and the acquisition and re-use of rationales. In this paper we focus on the acquisition and re-use of rationales processes to support plan analysis.

### Acquiring Rationales for Plan Evaluations

One of the main goals of this work is to capture the knowledge behind the classifications made upon the metric values and the plans. Rationales for plan evaluation may refer to elements and properties of the plan, including the plan structure itself. Therefore, it is necessary in the first place to consider a formal foundation of terms, concepts, relations and axioms to provide the base vocabulary of plan elements that

can be used to specify a rationale. In this work, such a formal foundation is the Plan Ontology. Before introducing how rationales are captured and represented, we first describe the plan ontology utilized in the postDAM.

### Representation of Evaluated Plans

As mentioned by Tate (Tate 1996), a richer plan representation could provide the following: a common basis for human and system communication about plans; a shared model of what constitutes a plan; mechanisms for automatic manipulation and analysis of plans; a target representation for reliable acquisition of plan information and feedback; formal reasoning about plans and re-use mechanisms. Such a representation is often based on an ontology - a plan ontology - which explicitly specifies the intended meanings of the terms being used, such as processes, activities, the constraints over their occurrences, or the meaning of the planning problem itself (Grüninger and Kopena 2005).

Among different ontologies for representing plans, we have chosen the Process Specification Language (PSL) (Schlenoff, Knutilla, and Ray 1996; Grüninger and Menzel 2003; Grüninger and Kopena 2005). PSL is an expressive ontological representation language of processes (plans), including activities and the constraints on their occurrences. PSL has been designed as a neutral interchange ontology to facilitate correct and complete exchange of process information among manufacturing systems such as scheduling, process modeling, process planning, production planning, simulation, project management, workflow, and business process applications (Grüninger and Kopena 2005). An interesting aspect of the PSL architecture is that it supports a set of extensions. A designer can extend PSL precisely to fit their expressive needs. We use PSL as the base for the plan ontology utilized in the postDAM framework.

Two of the most important terms in the core of PSL Ontology are the *activity* and its *occurrence*. As described in (Grüninger and Menzel 2003), an activity is a repeatable pattern of behavior, while an activity occurrence corresponds to a concrete instantiation of this pattern. For example, the term *pickup(r,x)* can denote the class of activities for picking up some object *x* with robot *r*, and the term *move(r,x,y)* can refer to the class of activities for moving robot *r* from location *x* to location *y*. The ground terms such as *pickup(Robot1,BlockA)* and *move(Robot1,LocA,LocB)* are instances of these classes of activities, and each instance can have different occurrences (e.g., two different occurrences of *move(Robot1,LocA,LocB)* can appear in plan). In fact, activities may have several or no occurrences. The relationship between activities and activity occurrences is represented by the *occurrence\_of(o,a)* relation. Any activity occurrence corresponds to a unique activity. *Object* is also a term in PSL-core ontology. An *object* might be an argument of activities or fluents. Fluents are used to describe facts. For example, *at(Robot1,LocA)* is a fluent in PSL while *Robot1* and *LocA* are objects. Moreover, the term *State* is also used in PSL ontology, in its extensions. *States* may refer to a particular situation of the domain or to a fluent. Relationships such as *prior(f,o)* and *holds(f,o)* denote, respectively, a fluent (or state) *f* that holds prior to the activity occurrence *o* and af-

<sup>1</sup>tuProlog: see <http://alice.unibo.it/xwiki/bin/view/Tuprolog/>.

ter such occurrence. The complete lexicon of the language is detailed in the PSL website.<sup>2</sup>

As opposed to most applications of the PSL ontology, in this work we focus on the representation of plans generated by automated planners. Therefore, additional semantics and vocabularies must be considered. We have extended the PSL lexicon to include the terms, characteristics and vocabulary of AI planning, including *domain*, *problem*, *operators*, *pre- and post-conditions*, *objects*, *fluents that define states*, *plan*, *metrics*, and *plan quality*. Table 1 summarizes such additional terms and vocabulary used in the postDAM framework to represent a plan and its quality.

<i>domain(d)</i>	<i>d</i> is a domain
<i>problem(i)</i>	<i>i</i> is a problem instance
<i>problem_of(i,d)</i>	<i>i</i> is a problem instance of <i>d</i>
<i>problem_solving_acti- vity_of(a,i)</i>	<i>a</i> the problem-solving activity of problem instance of <i>i</i>
<i>plan(p)</i>	<i>p</i> is a plan
<i>solution_of(p,d,i)</i>	<i>p</i> is a solution to the problem <i>i</i> of domain <i>d</i>
<i>fluent_of(f,s)</i>	<i>f</i> is a fluent of the state <i>s</i> . A set of fluents defines a state
<i>numeric_fluent_of(f,v,s)</i>	<i>f</i> is a numeric fluent with value <i>v</i> in the state <i>s</i>
<i>positive_precondition(a,f)</i>	<i>f</i> is a precondition of <i>a</i>
<i>negative_precondition(a,f)</i>	<i>f</i> is a negative precondition of <i>a</i>
<i>effect(a,f)</i>	<i>f</i> is an effect of <i>a</i>
<i>negative_effect(a,f)</i>	<i>f</i> is a negative effect of <i>a</i>
<i>metric(m)</i>	<i>m</i> is a metric
<i>metric_value(p,m,v)</i>	<i>m</i> has value <i>v</i> in plan <i>p</i>
<i>metric_quality(p,m,q)</i>	<i>m</i> has quality value <i>q</i> in plan <i>p</i>
<i>quality(p,q)</i>	<i>p</i> has quality value <i>q</i>

Table 1: Addition terms and relations to PSL Ontology used in postDAM

As an example of the ontological representation of plans in postDAM, let us suppose a simple planning problem from the classical blocks world domain in which block B must be unstacked from block A and put on the table. A plan with two actions solves the problem. In such example, the plan structure is represented along with the basic information about domain and problem, as well as the initial state. Figure 1 illustrates the two blocks example, along with basic concepts and elements of the proposed PSL extension (some of the terms are omitted in the figure to provide a clear view of ontology structure). The top area of Figure 1 illustrates the terms of the domain of application and the problem, whereas the bottom illustrates the plan structure.

Reasoning about a given plan would require the proper encoding of the PSL ontology, including the plan representation, action specifications and propagation rules. Such reasoning could be used to infer or check plan properties and characteristics (e.g. to infer in which state a given goal or fluent is achieved). We use Prolog for encoding the ontology in the postDAM framework.

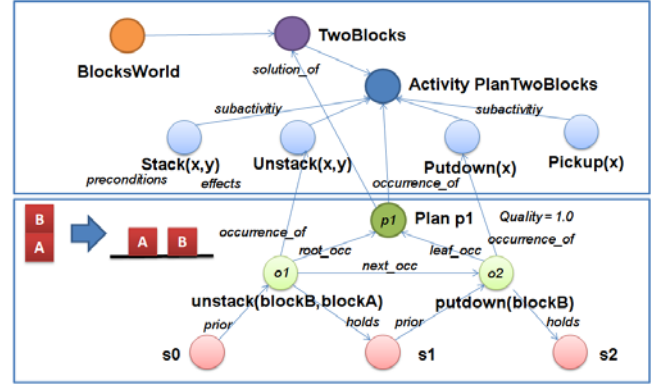


Figure 1: An illustration of the plan structure in the plan ontology used in postDAM

### Representation of Rationales for Plan Evaluations

We aim at rationales that explain what affects the quality of plans (positively and negatively) and consequently why a plan has a given classification (including factual reasons or preferences). Conceptually, *plan evaluation rationales* in postDAM have the following straightforward format:

$$if < condition > then < effect on plan quality > \quad (1)$$

The *condition* of the rationale can be any logical sentence involving the properties of a plan, while the *effects* are pre-defined terms that specify whether the condition increases or decreases the quality of the plan. A rationale may refer to the effect on plan quality or on a particular metric quality. For example, one may have the following evaluation rationale: “if (*action1* occurs after *action2* AND *action3* is the last action in the plan) then (plan quality decreases)”.

In order to represent rationales, we use the plan ontology described above along with a new vocabulary to capture the concepts of plan evaluation rationale. The vocabulary includes an important term called *rationale(r)* to represent an evaluation rationale. The relation *quality\_rationale\_of(r,p,j)* denotes the relationship between a rationale *r* and the plan *p* along with a user justification *j*. Depending on the ontology encoding, the justification might be a string object such as in the following example: *quality\_rationale\_of(r1,p1, “when action1 occurs after action2 and action3 is the last action in the plan the quality is decreased”)*. The effects on plan quality are represented using the relations *affect\_plan\_quality(r,p,e)* and *affect\_metric\_quality(r,p,m,e)* where *e* can be either *increase* or *decrease*. The former refers to the direct effect on the quality of a plan, while the latter refers to the effect on a particular metric *m*. The following sentences represent the conceptual format of the rationale (sentence 1) using the above extended vocabulary:

$$\begin{aligned} < condition > \rightarrow quality\_rationale\_of(r,p,j). \\ quality\_rationale\_of(r,p,j) \rightarrow \\ affect\_plan\_quality(r,p,e). \end{aligned}$$

<sup>2</sup>PSL is available at <http://www.mel.nist.gov/psl/ontology.html>.

In the first sentence, if *condition* is satisfied then relation *quality\_rationale\_of* becomes true, and consequently relation *affect\_plan\_quality* also becomes true according to the second sentence. The rationale's condition is represented using the vocabulary defined in the plan ontology.

We also consider different abstraction levels of rationales in the extended ontology. Rationales can be classified as *problem-dependent*, *domain-dependent* or *domain-independent*. Problem-dependent rationales are those that are only applied in the context of a particular problem instance (e.g., “if a robot moves to location 7 the plan has a low quality”). Domain-dependent rationales are those that are applicable in all problem instances of a particular domain (e.g., “if any robot performs a loop in its path the resulting plan quality is low”). Finally, domain-independent rationales are those applied to any planning domain or set of domains (e.g., “if a vehicle performs a loop of movements then the quality of the plan will be decreased”). The relation used to represent these concepts is the *abstraction\_level(r,l)*, where *l* refers to one of the three levels of abstraction. The level of abstraction of rationales assists their re-use as will be discussed in the next section. Table 2 summarizes the terms and relations for representing rationales in postDAM.

<i>rationale(r)</i>	<i>r</i> is a rationale
<i>quality_rationale_of(r,p,j)</i>	<i>r</i> is a rationale of plan <i>p</i> with justification <i>j</i>
<i>affect_plan_quality(r,p,e)</i>	<i>r</i> describes the effect <i>e</i> on quality of <i>p</i>
<i>affect_metric_quality(r,p,m,e)</i>	<i>r</i> describes the effect <i>e</i> on quality of metric <i>m</i> of <i>p</i>
<i>abstraction_level(r,l)</i>	<i>l</i> is the abstraction level of <i>r</i>

Table 2: Rationales terms and relations in postDAM

The following examples represent two rationales using the vocabulary in Table 2.

*rationale(r1).*  
*abstraction\_level(r1, problem – dependent).*  
 $\forall p, o, sg \cdot leaf\_occ(o, p) \wedge holds(sg, o) \wedge$   
 $fluent\_of(at(robot1, loc1), sg) \rightarrow$   
 $quality\_rationale\_of(r1, p, j1).$   
 $\forall p \cdot quality\_rationale\_of(r1, p, j1) \rightarrow$   
 $affect\_plan\_quality(r1, p, increase).$

*rationale(r2).*  
*abstraction\_level(r2, problem – dependent).*  
 $\forall p, o, sg, v \cdot leaf\_occ(o, p) \wedge holds(sg, o) \wedge$   
 $numeric\_fluent\_of(traveleddistance(robot1, v), sg) \wedge$   
 $v > 50 \rightarrow quality\_rationale\_of(r2, p, j2).$   
 $\forall p \cdot quality\_rationale\_of(r2, p, j2) \rightarrow$   
 $affect\_metric\_quality(r2, p, fuelused, decrease).$

The above example represents two rationales, *r1* and *r2*. The first rationale can be interpreted as follows: “The quality of a plan increases whenever the *robot1* is at *loc1* in the state generated by the last activity occurrence in the plan (*leaf\_occ(o,p)*), and the goal state is still *sg*”. The second rationale has the following meaning: “The quality of the metric *fuelused* decreases whenever the traveled distance of the *robot1* (variable *v*) is greater than 50 to the goal state”. These rationales are manually inserted and maintained by users using itSIMPLE’s interface.

## Re-using Rationales for Plan Evaluation

During the plan evaluation, stored rationales can be re-used to support classification and justification. When a new plan is created, we can use past evaluations to identify the good and bad characteristics of the plan. In postDAM, the re-use process involves the tool itSIMPLE, the reasoning interface tuProlog and the *Plan Analysis Database*. The information from the abstraction level of rationales is essential to the re-use process. Supposing a new plan *pn* is generated to solve a problem instance *i* in the domain *d*, the process of re-using existing rationales is as follows:

1. itSIMPLE first translates the domain operators, the objects, the problem instance, and the new plan *pn* to the PSL plan ontology.
2. itSIMPLE accesses the plan analysis database to select the following rationales: (1) problem-dependent rationales that are applied specifically to the problem *i*; (2) domain-dependent rationales that are applied to any problem instance in domain *d*; and (3) domain-independent rationales that are applied to any planning domain.
3. The translated plan and selected rationales are put together with the PSL-Core and core theories forming a knowledge base to the plan.
4. The knowledge base is then read by the tuProlog for inference requests.
5. itSIMPLE accesses tuProlog to check which rationales are applied to plan *pn*. itSIMPLE requests the inference of *quality\_rationale\_of(R,p1,J)*, where *R* and *J* are variables to be instantiated by the Prolog engine.
6. The list of applied rationales (instantiated values of variable *R*), along with their respective justification (instantiation values of variable *J*), is read by itSIMPLE and attached to the new plan *pn* (in the XML format).

The re-used rationales can be seen in the evaluation summary provided by itSIMPLE.

## Case Study

In this section, we present a case study using a benchmark domain from the *International Planning Competitions* (IPC) to evaluate the acquisition process of plan evaluation rationales in a post-design domain adaptation: the Gold Miner domain. This domain was chosen from a recent KE competition, because it is intuitive and has a clear correspondence between objects in the real and virtual world.

The procedure used for the case study is as follows:

1. We gather all rationales found and observations made during a virtual prototyping phase of plans generated by eight state-of-the-art planners: SGPlan5, MIPS-xxl 2006, LPG-td, MIPS-xxl 2008, SGPlan6, Metric-FF, LPG 1.2, and hspsp. For more details about the virtual prototyping phase see (Vaquero, Silva, and Beck 2010)
2. We represent the rationales in itSIMPLE using the proposed extension of the PSL ontology and the acquisition process described above.
3. We then analyze the rationales regarding their applicability, re-use and generality.

## The Gold Miner Domain

The Gold Miner is a benchmark domain from the learning track of IPC-6 (2008). In this domain, a robot is in a mine and has the objective of reaching a location that contains gold. The mine is represented as a grid in which each cell contains either hard or soft rock. There is a special location where the robot can either pickup an unlimited supply of bombs or pickup a single laser cannon. The laser cannon can be used to destroy both hard and soft rock, whereas the bomb can only penetrate soft rock. If the laser is used to destroy a rock that is covering the gold, the gold will also be destroyed. However, a bomb will not destroy the gold, just the rock. This particular domain has a simple optimal strategy<sup>3</sup> in which the robot must (1) get the laser, (2) shoot through the rocks (either soft or hard) until it reaches a cell neighboring the gold, (3) go back to get a bomb, (4) explode the rock at the gold location, and (5) pickup the gold. In this case study we used the propositional typed PDDL model from the testing phase of IPC-6.

During the virtual prototyping and model refinement cycles performed with the Gold Miner domain described in (Vaquero, Silva, and Beck 2010), a set of observations were made. We summarize these observations as follows:

- One planner generated invalid solutions in which the robot used the laser at the gold location, destroying the gold (Vaquero, Silva, and Beck 2010).
- Some planners provided (valid) plans in which the laser cannon was fired at an already clear location.
- The laser cannon was left in a different position from the initial one. It would be better if the robot could leave the laser only at the same spot as the bomb source.
- Unnecessary move actions were present in some plans.

In this case study, we address each one of the above rationales in the postDAM framework.

The undesirable firing behavior of the laser cannon naturally decreases the quality of the plan, and specifically to the *laserusage* metric. The following rationale *rGM1* denotes the issue of firing to an already clear position (the element *jGM1* represents the user's justification of rationale *rGM1*):

$$\begin{aligned} & \text{rationale}(rGM1). \\ & \forall p, o, t, x, y, s \cdot \text{subactivity\_occurrence}(o, p) \wedge \\ & \quad \text{occurrence\_of}(o, \text{firelaser}(t, x, y)) \wedge \\ & \quad \text{prior}(s, o) \wedge \text{fluent\_of}(\text{clear}(y), s) \rightarrow \\ & \quad \text{quality\_rationale\_of}(rGM1, p, jGM1). \\ & \forall p \cdot \text{quality\_rationale\_of}(rGM1, p, jGM1) \rightarrow \\ & \quad \text{affect\_plan\_quality}(rGM1, p, \text{decrease}). \end{aligned}$$

The justification *jGM1* could be encoded as a string like *jGM1* = 'Laser fired to nowhere (clear position), at ' + *y*. Such a justification will be instantiated with every possibility of firing the laser cannon to a clear position *y*. itSIMPLE captures and records these instantiations in the XML representation of the plan as well as in the database. The evaluations summary provided by the tools shows such justifications to the users.

<sup>3</sup>IPC-6 2008. <http://eecs.oregonstate.edu/ipc-learn/>

The above rationale is applicable to plans of any problem instances in the Gold Miner domain and, therefore, can be considered in this work as a domain-dependent rationale (*abstraction\_level*(*rGM1*, domain-dependent)). However, one might consider it as a domain-independent rationale, applicable to a class of domains in which action *firelaser*(*t, x, y*) exists and fluent *clear* is used as one of the effects.

A specific undesirable firing behavior was also detected in some of the plans generated for the problem instance *gold-miner-target-5x5-01* from IPC. In this case, two laser cannon shots were made at rock locations that did not belong to a reasonable path to the gold position, consequently decreasing plan quality. The following rationale *rGM2* represents such situation (the objects *node2i1* and *node3i1* represent the specific locations where the laser was fired):

$$\begin{aligned} & \text{rationale}(rGM2). \\ & \forall p, o1, o2, t \cdot \text{occurrence\_of}(p, \text{plangoldminertarget-5x501}) \wedge \text{subactivity\_occurrence}(o1, p) \wedge \\ & \quad \text{subactivity\_occurrence}(o2, p) \wedge \\ & \quad \text{occurrence\_of}(o1, \text{firelaser}(t, \text{node2i0}, \text{node2i1})) \wedge \\ & \quad \text{occurrence\_of}(o2, \text{firelaser}(t, \text{node3i0}, \text{node3i1})) \rightarrow \\ & \quad \text{quality\_rationale\_of}(rGM2, p, jGM2). \\ & \forall p \cdot \text{quality\_rationale\_of}(rGM2, p, jGM2) \rightarrow \\ & \quad \text{affect\_plan\_quality}(rGM2, p, \text{decrease}). \end{aligned}$$

The condition of the rationale *rGM2* checks the existence of two specific firing occurrences. Note that rationale *rGM2* is restricted to solutions of the problem *gold-miner-target-5x5-01* by the condition *occurrence\_of*(*p*, *plangoldminertarget5x501*) (where *plangoldminertarget5x501* is the activity of solving problem *gold-miner-target-5x5-01*). Therefore, this is a problem-dependent rationale (*abstraction\_level*(*rGM2*, problem-dependent)).

During the virtual prototyping phase, we raised the issue of the position in which the laser cannon was left at the goal state. Leaving the cannon at the same position as the bomb source was preferred. The following rationale *rGM3* denotes such a preference and expectation:

$$\begin{aligned} & \text{rationale}(rGM3). \\ & \forall p, o, sg, x \cdot \text{leaf\_occ}(o, p) \wedge \text{holds}(sg, o) \wedge \\ & \quad \text{fluent\_of}(\text{bombat}(x), sg) \wedge \\ & \quad \text{fluent\_of}(\text{laserat}(x), sg) \rightarrow \\ & \quad \text{quality\_rationale\_of}(rGM3, p, jGM3). \\ & \forall p \cdot \text{quality\_rationale\_of}(rGM3, p, jGM3) \rightarrow \\ & \quad \text{affect\_plan\_quality}(rGM3, p, \text{increase}). \end{aligned}$$

The condition of *rGM3* checks the existence of both fluents *bombat*(*x*) and *laserat*(*x*) (where *x* is a location) at the goal state *sg*. In this case study, the rationale *rGM3* is considered domain-dependent since it is applicable for all synthesized solutions for the Gold Miner domain.

In all refined models resulted from the virtual prototyping phase, unnecessary move actions appear in their respective plans. During the experiment, the rationales for detecting and explaining such undesirable characteristic evolve from a specific approach to a more general one (reaching a reusable representation). Due to limited space, we will focus on an example of rationale that referred to some loops in the

robot's path. This rationale captured during the experiment has the following representation:

*rationale(rGM4).*  
 $\forall p, o1, o2, a, t, x1, x2 \cdot \text{subactivity\_occurrence}(o1, p) \wedge$   
 $\text{subactivity\_occurrence}(o2, p) \wedge$   
 $\text{next\_subocc}(o1, o2, a) \wedge$   
 $\text{occurrence\_of}(o1, \text{move}(t, x1, x2)) \wedge$   
 $\text{occurrence\_of}(o2, \text{move}(t, x2, x1)) \rightarrow$   
 $\text{quality\_rationale\_of}(rGM4, p, jGM4).$   
 $\forall p \cdot \text{quality\_rationale\_of}(rGM4, p, jGM4) \rightarrow$   
 $\text{affect\_plan\_quality}(rGM4, p, \text{decrease}).$

Rationale *rGM4* refers to all sequences of move actions that take a robot from location *x1* to *x2* and then back to *x1* immediately after that. As the re-modeling process progressed, more sophisticated explanations can be constructed manually (Vaquero 2011). For example, a rationale could be defined to detect any complex loop (*x1, x2, ..., xn, x1*) performed by a robot. The following rationale representation captures any outer loop in a single robot's path:

*rationale(rGM5).*  
 $\forall p, o1, o2, x, y, z, a \cdot \text{subactivity\_occurrence}(o1, p) \wedge$   
 $\text{subactivity\_occurrence}(o2, p) \wedge$   
 $\text{occurrence\_of}(o1, \text{move}(t, x, y)) \wedge$   
 $\text{occurrence\_of}(o2, \text{move}(t, y, z)) \wedge$   
 $\text{next\_subocc}(o1, o2, a) \rightarrow$   
 $\text{consecutive\_move}(o1, o2, t, p).$   
 $\forall p, o1, o2, o3, x, y, z, w, h, a \cdot$   
 $\text{subactivity\_occurrence}(o1, p) \wedge$   
 $\text{subactivity\_occurrence}(o2, p) \wedge$   
 $\text{subactivity\_occurrence}(o3, p) \wedge$   
 $\text{occurrence\_of}(o1, \text{move}(t, x, y)) \wedge$   
 $\text{occurrence\_of}(o2, \text{move}(t, z, w)) \wedge$   
 $\text{occurrence\_of}(o3, \text{move}(t, h, z)) \wedge$   
 $\text{min\_precedes}(o1, o2, a) \wedge \text{next\_subocc}(o3, o2, a) \wedge$   
 $\text{consecutive\_move}(o1, o3, t, p) \rightarrow$   
 $\text{consecutive\_move}(o1, o2, t, p).$   
 $\forall p, o1, o2, x, y, z, a \cdot \text{subactivity\_occurrence}(o1, p) \wedge$   
 $\text{subactivity\_occurrence}(o2, p) \wedge$   
 $\text{occurrence\_of}(o1, \text{move}(t, x, y)) \wedge$   
 $\text{occurrence\_of}(o2, \text{move}(t, z, x)) \wedge$   
 $\text{min\_precedes}(o1, o2, a) \wedge$   
 $\text{consecutive\_move}(o1, o2, t, p) \rightarrow$   
 $\text{loop\_of\_move}(o1, o2, t, p).$   
 $\forall p, o1, o11, o2, o22, x, y, l, h, a \cdot$   
 $\text{occurrence\_of}(o1, \text{move}(t, x, l)) \wedge$   
 $\text{occurrence\_of}(o2, \text{move}(t, h, x)) \wedge$   
 $\text{min\_precedes}(o1, o2, a) \wedge \text{next\_subocc}(o11, o1, a) \wedge$   
 $\text{next\_subocc}(o2, o22, a) \wedge$   
 $\text{occurrence\_of}(o11, \text{move}(t, y, x)) \wedge$   
 $\text{occurrence\_of}(o22, \text{move}(t, x, y)) \rightarrow$   
 $\text{has\_previous\_loop\_move}(o1, o2, t).$   
 $\forall p, o1, o2, t, x, y, z \cdot \text{subactivity\_occurrence}(o1, p) \wedge$   
 $\text{subactivity\_occurrence}(o2, p) \wedge$   
 $\text{occurrence\_of}(o1, \text{move}(t, x, y)) \wedge$   
 $\text{occurrence\_of}(o2, \text{move}(t, z, x)) \wedge$   
 $\neg \text{has\_previous\_loop\_move}(o1, o2, t) \wedge$   
 $\text{loop\_of\_move}(o1, o2, t, p) \rightarrow$   
 $\text{quality\_rationale\_of}(rGM5, p, jGM5).$

$\forall p \cdot \text{quality\_rationale\_of}(rGM5, p, jGM5) \rightarrow$   
 $\text{affect\_plan\_quality}(rGM5, p, \text{decrease}).$

In the above rationale, we define new relations to assist the identification of outer loops (for this context only) such as *consecutive\_move(o1,o2,t,p)*, *loop\_of\_move(o1,o2,t,p)* and *has\_previous\_loop\_move(o1,o2,3)*. The relation *consecutive\_move(o1,o2,t,p)* and *loop\_of\_move(o1,o2,t,p)* together capture any consecutive sequence of move occurrences that constitutes a loop, whereas relation *has\_previous\_loop\_move(o1,o2,t)* filters the outer loops. It is indeed possible to capture the existing inner loops performed by the robot, but we see them as redundant information when we capture the outer ones. The interesting point here is that we can design and create any supporting concept, relations and axiom for defining a rationale condition.

The rationales represented in this case study can be reused and applied in a new plan evaluation process. Using the integration between itSIMPLE and the reasoning system (tuProlog) we can automatically generate some justifications to the initial plan evaluation made by itSIMPLE. All justifications are based on the rationales acquired and inserted in the *Plan analysis Database* in this case study. For example, if a plan has a loop in the robot's path, the framework will be able to detect such a loop and pinpoint where it is happening to the user, as well as how it is affecting the plan quality (in this case negatively).

## Discussion

The proposed acquisition process of human-centered rationales has shown to be feasible using an ontological approach. The case study provides an indication that these rationales can be an important source of essential knowledge such as user preferences. The reuse of past evaluation experience gives an important starting point in any analysis of a newly generated plan. Such re-use not only supports the knowledge acquisition process but also the decision process on already built applications. In the latter case, rationales can be applied to identify the trade-offs of plans selected for execution, knowing beforehand their advantages and disadvantages.

The case study has also shown that rationales can evolve during design and over time. As designer becomes familiar with the observations and users' intention, the representation of the collected rationales becomes more accurate, encompassing the exact envisaged situations. Therefore, rationales have their own maturity process in the design cycle.

Although we have not implemented the cross-project reuse of rationales in this work, it is possible to infer from the case study some examples about what could be enhanced in planning design patterns. In the Gold Miner domain experiment, the rationale related to the undesirable movement loops is a potential candidate for being attached to design patterns such as the *transportation* described in (Long and Fox 2000). Moreover, metrics utilized in the experiments could enhance the design patterns; some of them can be used in a number of planning applications such as the travel distance (a common quality measure in transportation and navigation benchmark domains).

We believe that the matching process between past evaluations and design patterns can speed up the design process itself, improve quality, reduce cost, and decrease problem fixing issues in real planning applications.

## Conclusion

We have described a post-design framework that integrates a number of tools to assist the discovery of missing requirements, to support evaluation rationale acquisition and re-use, and to guide the model refinement cycle. In previous work we demonstrated that following a careful post-design analysis, we can improve not only plan quality but also solvability and planner speed (Vaquero, Silva, and Beck 2010). In this paper we demonstrated how evaluation rationales can be captured, represented and re-used. We discuss that this type of human-centric feedback can be useful and reusable in further plan evaluations and in other planning domains. In a real planning application, the analysis process that follows design becomes essential to have the necessary knowledge represented in the model.

## References

- Cesta, A.; Finzi, A.; Fratini, S.; Orlandini, A.; and Tronci, E. 2008. Validation and verification issues in a timeline-based planning system. In *Proceedings of the ICAPS 2008 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.
- Daughtry, J. M.; Burge, J. E.; Carroll, J. M.; and Potts, C. 2009. Creativity and rationale in software design. *ACM SIGSOFT Software Engineering Notes* 34:27–29.
- Grüninger, M., and Kopena, J. B. 2005. Planning and the Process Specification Language. In *Proceedings of the ICAPS 2005 Workshop on the Role of Ontologies in Planning and Scheduling*, 22–29.
- Grüninger, M., and Menzel, C. 2003. The Process Specification Language (PSL) theory and applications. *AI Magazine* 24(3):63–74.
- Hatzi, O.; Vrakas, D.; Bassiliades, N.; Anagnostopoulos, D.; and Vlahavas, I. P. 2010. A visual programming system for automated problem solving. *Expert Systems With Applications* 37:4611–4625.
- Jónsson, A. K. 2009. Practical Planning. In *ICAPS 2009 Practical Planning & Scheduling Tutorial*.
- Long, D., and Fox, M. 2000. Automatic Synthesis and use of Generic Types in Planning. In *Artificial Intelligence Planning and Scheduling AIPS-00*, 196–205. Breckenridge, CO: AAAI Press.
- McCluskey, T. L., and Simpson, R. M. 2006. Tool support for planning and plan analysis within domains embodying continuous change. In *Proceedings of ICAPS 2006 Workshop on Plan Analysis and Management*.
- McCluskey, T. L. 2002. Knowledge Engineering: Issues for the AI Planning Community. *Proceedings of the AIPS-2002 Workshop on Knowledge Engineering Tools and Techniques for AI Planning*. Toulouse, France 1–4.
- Myers, K. L. 2006. Metatheoretic Plan Summarization and Comparison. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS-06)*. Cumbria, UK: AAAI Press.
- Polyank, S., and Austin, T. 1998. Rationale in Planning: Causality, Dependencies and Decisions. *Knowledge Engineering Review* 13(3):247–262.
- Rabideau, G.; Engelhardt, B.; and Chien, S. 2000. Using generic preferences to incrementally improve plan quality. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*. Breckenridge, CO: AAAI Press.
- Ramaesh, B., and Dhar, V. 1994. Representing and maintaining process knowledge for large-scale systems development. *IEEE Expert: Intelligent Systems and Their Applications* 9(2):54–59.
- Schlenoff, C.; Knutilla, A.; and Ray, S. 1996. Unified process specification language: Functional requirements for modeling processes. In *National Institute of Standards and Technology*.
- Simpson, R. M. 2007. Structural Domain Definition using GIPO IV. In *Proceedings of the Second International Competition on Knowledge Engineering for Planning and Scheduling*.
- Studer, R.; Benjamins, V. R.; and Fensel, D. 1998. Knowledge Engineering: Principles and Methods. *Data and Knowledge Engineering* 25(1-2):161–197.
- Tate, A. 1996. Representing plans as a set of constraints - the I-N-OVA model. In *Proceedings Third International Conference on AI Planning Systems (AIPS-96)*. Edinburgh: AAAI Press.
- Upal, M. A., and Elio, R. 1999. Learning rationales to generate high quality plans. In *Proceedings of the Twelfth International FLAIRS Conference*, 371–377. Menlo Park, CA, USA: AAAI Press.
- Vaquero, T. S.; Romero, V.; Tonidandel, F.; and Silva, J. R. 2007. itSIMPLE2.0: An integrated Tool for Designing Planning Environments. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS 2007)*. Providence, Rhode Island, USA: AAAI Press.
- Vaquero, T. S.; Silva, J. R.; Ferreira, M.; Tonidandel, F.; and Beck, J. C. 2009. From Requirements and Analysis to PDDL in itSIMPLE3.0. In *Proceedings of the Third International Competition on Knowledge Engineering for Planning and Scheduling, ICAPS 2009*, 54–61.
- Vaquero, T. S.; Silva, J. R.; and Beck, J. C. 2010. Improving Planning Performance Through Post-Design Analysis. In *Proceedings of the ICAPS 2010 Workshop on Scheduling and Knowledge Engineering for Planning and Scheduling (KEPS)*, 45–52.
- Vaquero, T. S. 2011. *Post-Design Analysis for AI Planning Applications*. Ph.D. Dissertation, Polytechnic School of the University of São Paulo, Brazil.
- Wickler, G.; Potter, S.; and Tate, A. 2006. Recording Rationale in <I-N-C-A> for Plan Analysis. In *Proceedings of the ICAPS 2006 Workshop on Plan Analysis and Management*.



# The Challenge of Grounding Planning in Simulation with an Interactive Model Development Environment

Bradley J. Clement<sup>\*</sup>, Jeremy D. Frank<sup>\*\*</sup>,  
John M. Chachere<sup>\*\*\*</sup>, Tristan B. Smith<sup>+</sup> and Keith J. Swanson<sup>\*\*</sup>

<sup>\*</sup>Jet Propulsion Laboratory, California Institute of Technology <sup>\*\*\*</sup>SGT Inc, <sup>+</sup>MCT Inc., <sup>\*\*</sup>NASA Ames Research Center  
{FirstName.MiddleInitial.LastName}@nasa.gov

## Abstract

A principal obstacle to fielding automated planning systems is the difficulty of modeling. Physical systems are modeled conventionally based on specification documents and the modeler's understanding of the system. Thus, the model is developed in a way that is disconnected from the system's actual behavior and is vulnerable to manual error. Another obstacle to fielding planners is testing and validation. For a space mission, generated plans must be validated often by translating them into command sequences that are run in a simulation testbed. Testing in this way is complex and onerous because of the large number of possible plans and states of the spacecraft. Though, if used as a source of domain knowledge, the simulator can ease validation. This paper poses a challenge: to ground planning models in the system physics represented by simulation. A proposed, interactive model development environment illustrates the integration of planning and simulation to meet the challenge. This integration reveals research paths for automated model construction and validation.

## Introduction

There are several applications that have benefited from using model-based planners. But, there are long-standing, fundamental problems in applying automated planning to physical systems: models are abstractions that are disconnected from the physical system (reducing accuracy) and limited in representation (increasing complexity). Brooks remarked, "Explicit representations and models of the world simply get in the way. It turns out to be better to use the world as its own model" (1991). The successes of model-based applications dull this point, but developing a model that is sufficient for a real application can be a painful struggle, especially if the modeling language is missing basic features like numeric state variables, in which case the language can seem to "get in the way." While languages have become more expressive, algorithms that parse them do not scale well, and detailed modeling may require too much effort. Abstraction has its advantages! So, can existing planning systems somehow use "the world" as their model?

That is the challenge: to ground an automated planner in system physics ("the world") and thus simplify model

development, verification, and validation. Space missions often develop simulation testbeds that serve as ground truth for the system and can be automated to evaluate test cases in batches. So, one approach to meeting the challenge is to help automate model development and testing by integrating the planner and simulator.

We first describe a sample activity to show the complexity of translating domain knowledge into different elements of a declarative domain model. We then explain how prior work in verification and validation does not address this problem. We propose an Interactive Model Development Environment (IMDE) to simplify the construction, validation, and maintenance of automated planning systems with help from a simulator. The majority of the paper describes IMDE functions and architecture. We discuss both current and near-term technologies that can be used to build such an IMDE and mention progress on a proof-of-concept implementation. We conclude with research goals that could help produce valuable mission planning technologies.

## Model Development Challenges

As discussed above, fielding model-based planning applications is challenging because typical modeling processes are complex and error-prone and because the combinations of possible test scenarios can seem as overwhelming as testing the entire system being modeled.

Originally planning languages and algorithms used only Boolean state variables. Such variables are generally impractical for representing time, location, and other numerical states. Planning languages are more expressive now (Howey et al. 2004, Fox and Long, 2003) but their limitations still force inelegant workarounds that make system models complex. Modeling choices can strongly influence the performance of automated planning. So, performance requirements can spur model revisions that increase complexity further. This complexity compounded by human error and lack of information about the modeled system's behavior can produce inconsistencies with the



model and the modeled system. Identifying and fixing these inconsistencies can require significant work.

To make the discussion concrete, consider the difficulty of representing an activity for changing a spacecraft's attitude (its 3-dimensional orientation).

### Example activity model of spacecraft slewing

```
(:durative-action slew
:parameters (?from - attitude
             ?to - attitude)
:duration (= ?duration 5)
:condition
  (and
    (at start (pointing ?from))
    (at start (cpu-on))
    (over all (cpu-on))
    (at start (>= (sunangle) 20.0))
    (over all (>= (sunangle) 20.0))
    (at start (communicating))
    (over all (communicating))
    (at start (>= (batterycharge) 2.0)))
:effect
  (and
    (at start (decrease (batterycharge)2.0))
    (at start (not (pointing ?from)))
    (at end (pointing ?to))))
```

The PDDL above specifies a spacecraft attitude change activity. The activity model is more abstract than a typical simulator's, which would use the spacecraft's command set, lighting conditions (a function of the orbit), dynamics of slewing the spacecraft, communication asset locations, spacecraft power utilization, and battery performance. Typically, extracting knowledge from the simulator to configure the planning system is manual, inefficient and error-prone. The modeler may have a lot of questions:

- How do planner model attitudes relate to real spacecraft operations' continuous attitudes? For example, does it suffice to represent a deep-space craft with camera directional sensors using a discrete valued attitude variable with values such as to-Earth (for deep-space), Earth-nadir (for Earth orbits), Sun-pointing (for solar power generation), and others for sets of navigation guide stars? How does data from the inertial measurement unit map to these discrete directions?
- How does the planner model battery discharge? How can the model conservatively estimate the battery energy consumed by subsystems for different possible system states? For example, does temperature affect power usage? How is a cap on battery capacity modeled to avoid overfilling? How is solar recharging modeled?
- What drives slew duration? Is it proportional with angular slew distance? Will a slew always follow the shortest rotation? Must it avoid pointing instruments at the sun? What determines the choice of control system (reaction wheels, thrusters, or torque rods)?
- How is reaction wheel momentum dumped?
- Along what axes can the spacecraft slew while communicating? conducting science measurements? recharging the battery? changing trajectory?

- What are the communication coverage requirements? What information is needed about the spacecraft orbit, availability of ground communication assets, and the spacecraft antenna type and configuration? When do ground stations require communications to monitor trajectory changes or other related activities?
- How does the abstract slew correspond to one or more sequences of spacecraft commands? Are there setup and teardown activities? Is the slew for each axis performed separately to avoid risk of concurrent interactions?

Before flight, the orbit, attitude, engineering subsystem specification, and simulations can change frequently. These changes require efficiently reconfiguring the activity planner. For example:

- New targets or navigation aids require updating the set of discrete attitudes.
- Changes in sequences can cause a change in attitude control system performance, leading to activity changes.
- Any power-using subsystem that changes performance (e.g., attitude control system or communication) will change power consumption. If planning determines mission objectives are infeasible, a need to slew faster could also increase power consumption
- Changing orbit, communication coverage plan, or antenna configuration may change the activity.
- Changing flight software (or the uses of major spacecraft operating modes) might require changing the commands that affect attitude.

### Verification, validation, and model checking

Validating the planning model (not just a plan) is a central challenge to automating model development. The planning system is partly a plan *verification* system because it checks constraints on system states that the plan's activities affect. However, *validating* the plan additionally requires validating that the effects are realized as expected. Validating the planning *model* requires validating all plans that the planner generates or accepts as feasible.

Model checking may detect violations of formal properties by the planning model or individual plans "in isolation" (e.g., Howey et al., 2004, Brat et al., 2008, Long et al., 2009, Raimondi et al., 2009, Cesta et al., 2010), but our goal is to validate the model against the simulator. Simulation of activities in the planning model can directly indicate problems, for example, an unrealized effect of an activity or a system fault. Model checking cannot substitute for this functionality without using a complete model of the simulator. Current model checking systems have the same representation and scaling problems as planning, so a detailed model (which rarely exists anyway) would likely be unusable.

## Interactive Model Development Environment

We now describe an approach that integrates planning and simulation in order to help automate model development and validation in the context of a hypothetical IMDE. We make assumptions that simplify the discussion, describe IMDE design features and architecture, and outline a concept of operation for modeling with the IMDE.

### Assumptions

The following assumptions simplify stating the challenge in the IMDE context and also indicate additional challenges addressed toward the end of the paper.

- The simulator input includes a list of time-tagged commands.
- The simulator runs deterministically.
- The simulator reports any errors (undesirable behavior).
- The system (spacecraft) and simulator are defect-free.
- The simulator is a black box (the user can neither change nor inspect its code and models)
- The simulator outputs time-tagged value samples of system state variables.
- Formal flight rules define mission constraints that are verifiable with the simulator output.
- Every plan that the planner sends to the simulator is consistent with the planner's model.
- An action in the plan corresponds to a list of time-tagged commands.
- The planner is sound but not necessarily complete.

### IMDE design features

The hypothetical IMDE could share many features of a traditional programming language Integrated Development Environment (IDE); An IMDE's model corresponds to an IDE's code, plans correspond to test cases, and the simulator corresponds to the computer. One distinctive IMDE function is the generation of test cases to aid model validation. Another is the generation of suggestions on how to fix modeling errors. In the traditional IDE, this is similar to suggesting code fixes for program run failures. Following sections discuss validation and model fixes.

Figure 1 shows the system architecture of the proposed IMDE. The Model Editor provides traditional IDE functions. The Simulation API Browser provides model creators access to the simulation API. With the Abstraction Editor a user documents how plan model building blocks (objects, states, timelines, actions, constraints) relate to data and commands in the simulation API, thus providing traceability for detecting model problems. These abstractions are the semantic glue connecting the planner to the “the world”/simulator. The Abstraction and Refinement Engines integrate the planner and simulator. The Refinement Engine transforms a plan into simulator

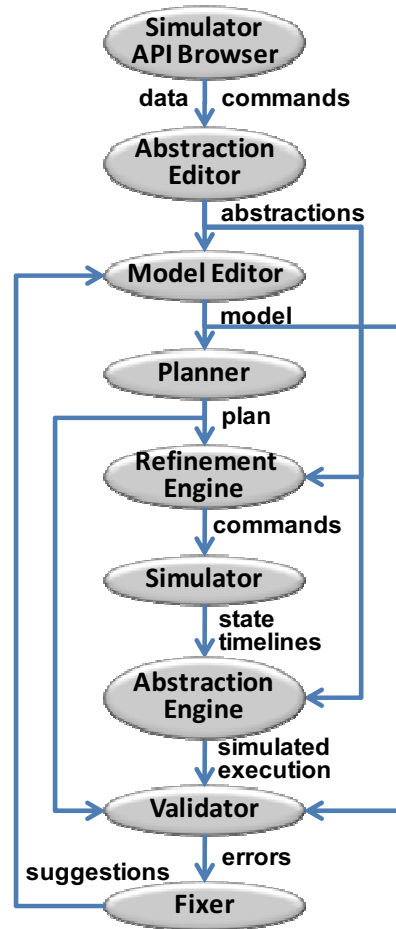


Figure 1: Hypothetical IMDE architecture.

command input. The Abstraction Engine transforms simulator output into an actual/simulated execution for comparison with the expected/planned execution. These executions are time-tagged actions and state variable values in the language of the planner. The Validator identifies discrepancies between the two executions, errors reported by the simulator, and any planning model constraint violations, some of which the simulator may not check (e.g., flight rules). A Plan Viewer (not shown) comparatively displays the simulated and planned executions (e.g., in a Gantt chart). The Plan Viewer (and/or an Error Viewer) visualizes discrepancies between the executions and highlights those that indicate modeling errors. Finally, the Fixer suggests model changes that may eliminate one or more errors seen in the current and past simulations of different plans. We explain how to detect errors and make suggestions after describing the IMDE concept of operation.

## Concept of operation

An IMDE user may start with an empty or existing planning domain model. Depending on the modeling language, an edit to the model may add, change, or remove actions, state variables, constraints, and effects (or their associated object types and sets). The user may create and edit abstractions to ground the model in simulator elements. A simulation interface exposes these elements, including commands and system state variables. These edits initiate the following basic workflow.

1. The user edits the model, or
2. the user edits abstraction by either
  - a. copying variables from the simulator interface to the model (e.g. `sunangle`),
  - b. abstracting variables in the model (e.g. `(pointing Earth)` is true in the planner if the simulator's `xyz` attitude is for each axis within 1 degree of the attitude to point directly at Earth),
  - c. copying a simulator command to the model as an action (e.g. `turn on CPU`), or
  - d. abstracting a command sequence to model an action (e.g. `command sequence to slew spacecraft`)
3. The IMDE generates possible initial states and plans and tests each by
  - a. translating the initial state and plan into simulator commands,
  - b. running the simulator with those commands,
  - c. translating simulator output to an execution,
  - d. checking the execution for violations of constraints in the planning domain model, and
  - e. checking for discrepancies between planned and simulated executions.
4. The IMDE analyzes test results to suggest changes to the planning model that could fix discrepancies.
5. The user assesses planned and simulated executions, their constraint violations, their discrepancies, and suggested fixes.
6. Repeat.

The idea is that when the user edits the model, in the background the IMDE generates and simulates different plans to search for discrepancies indicating modeling errors. The user can be made aware of these errors even while editing (much like syntax errors in an IDE), and when the user is ready to see what is in error, the IMDE may already have suggested fixes for the user to select.

## Translating plan information as abstraction and refinement

The previous workflow uses abstractions heavily. For example, in step 3a the Refinement Engine may translate one `slew(?from,?to)` action in the plan into three

ordered subsequences of simulator commands to rotate the spacecraft around each of its three axes. In step 2c and 2d, the user specifies this abstraction as an action decomposition, similar to hierarchical plan decomposition (Clement et al., 2007).

Another abstraction type for data specifies how state variables in the planning model relate to those in the simulator output. For example, an abstraction could map the simulator `xyz` spacecraft attitude to a discrete `(pointing ?target)` planner predicate, with `?target` either `Earth`, `Sun`, or `SomewhereElse`. An abstraction function could specify that `(pointing Earth)` is true if the simulator `xyz` attitude for each axis is within 1 degree of pointing the transceiver to the Earth's center. In general, an abstraction could be any function of a set of time-varying variables that calculates the time-varying values of some variable.

When the Refinement Engine translates initial state and plan information into simulator commands using these abstractions in steps 2a and b, some data abstractions may need to be reversed. For example, translating a plan's `slew(Sun,Earth)` action into simulator commands would translate the `Sun` and `Earth` symbols to the corresponding `xyz` attitudes for pointing to the targets.

The Abstraction Engine checks for discrepancies with the planned execution and helps identify modeling errors by translating simulation results into execution information in the planner language using the abstractions in Step 3c. The abstractions provide the time-varying planner state values, but another step is needed to construct the execution that explains these values. Our assumptions make this relatively simple, but in general it can be a difficult state estimation optimization problem.

## Identifying modeling errors

Modeling errors are indicated by errors explicitly reported by the simulator and by plan constraint violations on the simulated execution that do not occur in the planned execution (a discrepancy in constraint violations). For example, in testing the `slew(Sun, Earth)` action, the simulator might report an error from the fault management system because the computer had not yet been booted when commands were sent to the reaction wheels (a flight rule violation). This is an error in the planning model because the `slew` action lacked a necessary precondition that the computer be booted. As another example, the plan test case might include a goal (or constraint) `(pointing Earth)` to check that the effect of a `slew` is achieved. The simulator output could be error-free and translate back to an execution where `(pointing Earth)` was never achieved, failing the goal. This could be the result of the plan containing another overlapping `slew` command that

commanded the spacecraft to retarget the slew. In this case the modeling error was in allowing overlapping slews.

This simple specification for identifying modeling errors applies generally to different kinds of errors. For example, how would an error in the timing of a slew be detected? If the model specified a fixed duration for `slew`, the test plan still only needs a constraint that `(pointing ?to)` be true at the end of the `slew` activity. If the slew takes longer than expected, then the constraint will be violated in the simulated execution.

Discrepancies between the planner and simulator need not be modeling problems. Defining the planning states as abstractions of the simulator's states could naturally lose information. For example, the planning model could represent battery depletion as instantaneous while the simulation represents depletion as gradual. Discrepancies will probably manifest between the planned and simulated battery levels. But, planning the battery levels conservatively could avert simulation failures. The user may choose to omit specific discrepancies from reporting (as with waiving constraint violations in mixed initiative planning systems, Aghveli et al. 2007). However, the discrepancy might indicate an efficiency improvement opportunity: a more detailed battery depletion model could enable scheduling more activities.

These discrepancies of inefficiency could be detected by running plans that have constraint violations through the simulator and seeing if the same violations occur in the simulated execution.

### Generating plans to validate the model

The reason for generating different plans to test (step 3) is to validate that the model will work for all situations. Validating the model requires validating all possible plans that can be constructed from the model. In general, there may be an infinite number of possible plans, but there may be a manageable number that is enough to validate a single part of the model.

For example, if the user wants to ensure that the `(pointing ?to)` effect is always satisfied at the end of `slew(?from, ?to)`, then a complete space of plans to test would combine all possible initial attitudes, slews for all target attitudes (slew from each attitude to each other attitude), all possible additional actions (slews from each target to each other target), and the different temporal orderings of those other actions with respect to `slew(?from, ?to)`.

It is possible to generate all of these plans with special purpose code, but the planner itself may be leveraged to accomplish this. Instead of generating all combinations, incorporate this parameterization into a planning problem: what initial state and ordering of instantiations of `slew(?from, ?to)` will achieve `(pointing ?to)` at

the end? The set of valid solutions to this planning problem is the test suite.

Now, it is expected that multiple simulations could map to a single plan. For example, there are an infinite number of `xyz` attitudes that translate to `(pointing Earth)`. So, why not test all possible simulations instead of all possible plans? If plans are meant to be the only mechanism for generating command sequences for the spacecraft, the other simulations will never occur because a plan only translates to one set of commands resulting in one deterministic simulation. On the other hand, the initial state is not dependent on actions in the plan, so the complete space of test cases would include the infinite number of attitudes that translate to `(pointing Earth)`. In this case, conventional test coverage techniques may still be necessary.

Another reason to generate simulations instead of plans may be that the model has just been started, and many actions have yet to be modeled, so the necessary plan-based test cases to validate the first modeled action would be insufficient. Thus, generating simulations based on the simulator interface specification (using simulator commands instead of planner actions) would be useful and more robust to model changes. It may also be better to generate simulator-based test cases when there are many actions in the planning model. If activities are defined for many combinations and orderings of simulator commands, then the space of plans necessary to validate an action could be greater than the space of simulations due to repetition of simulator commands in a combination of actions.

Again, it may be possible to cleverly scope the validation to reduce the number of sequences tested. For example, test cases including two slews following the slew to be validated should find the same errors as those test cases with only a single following slew. Thus, a tractable number of test cases may be identified for validating an action in a model. This test coverage problem is known to be quite difficult and, thus, part of the challenge.

The tractability of validating the entire model depends on that of individual actions. Validating each action in isolation is enough to validate the entire model since the soundness of the planner guarantees combinations of actions.

### Suggesting changes to the model

When the IMDE runs a batch of plans through the simulator, some may result in simulator errors and some may result in planning constraint errors. These indicate that there are modeling errors, but the modeler may not be able to deduce the actual mistake by looking at any one execution. For example, suppose the slew was never executed because the CPU was never turned on, resulting

in a simulation error flag. There would be a violation of (`pointing Earth`) in the simulated execution, but no information in the output ties these errors with the state of the computer. So, the modeler would have to know the spacecraft (and simulator) very well to guess the problem after seeing it in a single run.

By finding relationships between plan/state attributes and simulator/discrepancy errors, the IMDE can generate plausible suggestions for fixing the model. For example, if a complete set of test cases showed that the slew failed every time that the computer was not booted, a machine learning classifier or data mining algorithm could identify the pattern. Then, the IMDE could suggest abstracting the *computerMode* variable in the simulator interface to a *cpu-on* predicate in the planning model and add the predicate as a precondition to *slew*. Other suggestions include adding a constraint that a *turnOnCpu* action always precedes *slew* or adding a simulator command to the slew abstraction/decomposition to *bootCpu*. These suggestions from the IMDE Fixer component (see Figure 1) could include changing constraints on an action, adding state variables, or creating new actions.

The challenge of generating suggestions may be in framing the learning problem. Plans have variable numbers of actions, so there is not an obvious feature set over which to learn. In addition, the modeler may want suggestions in terms of complex functional relationships of multiple variables. For example, the desired fix may be to avoid exhausting memory storage by adding a constraint that the sum of durations of all communications activities in a day must be greater than the sum of data collected multiplied by a particular constant. The number of functional relationships that may be part of a feature set of a learning algorithm could easily be intractable. On a positive note, the modeler may be able to deduce the needed fix with the help of overly-specific suggestions learned from a limited set of features.

## Technology Foundations

While the ultimate vision of the IMDE has yet to be achieved, many component technologies have been built. This section describes some of these technologies as well as research activities that enable the goal.

The itSimple tool (Vaquero et al., 2007) is a plan domain modeling environment very similar to the proposed IMDE. Users of itSimple can build static models of objects, actors, and relationships between them in a specialization of UML and dynamic models of how states of the objects are allowed to change using Petri Nets (an encoding of state charts); the Petri Net model acts as a simulation. The resulting models are automatically translated by itSimple to PDDL, after which the users can

continue refining the resulting models. A distinct difference from the IMDE approach is the assumed access to the simulator model (white-box simulation).

The Procedure Integrated Development Environment (PRIDE) (Izygon et al., 2008) is a procedure authoring technology prototype that can be used to create procedures for execution by flight controllers and crew. PRIDE presents procedure authors with a command and telemetry database; users can drag commands and telemetry references into a plan directly from the command and telemetry database GUI. PRIDE provides access to either state-chart simulations or high-fidelity simulations that the procedure writer can use to manually check procedures for correctness. Procedures can also be automatically verified by means of translation to Java and the use of model checking software (Brat et al., 2008). The use cases for creating procedures are quite similar to the assumptions made here. However, there is no abstraction mapping, and procedures lack formalisms needed for planning.

The Data Abstraction Architecture (DAA) (Bell et al., 2010) is designed to address the problem of transforming spacecraft or space system telemetry into useful information for operators (be they flight controllers or crew). The system allows system operators to write common data transformations using a GUI; the transformations are then executed by an engine that accepts telemetry as input, and produces more intuitive information as output. The DAA framework is well suited to editing data abstractions for the IMDE, but it would need to be extended to capture transformations of plan actions into simulator commands.

VAL takes steps toward the Fixer IMDE element by validating that a specific plan is indeed a solution to a planning problem that may be specified with continuous effects, including limited forms of time-dependent change on numerical state variables (Howey, et al., 2004). VAL can also advise modelers how to fix a plan. The goal explored here is how to validate that all plans execute as intended and suggest fixes to the model, not just the plan. Furthermore, the approach in VAL would have to apply to simulated executions.

The LOCM system (Cresswell et al. 2009) learns planning domain models from sets of example plans. Its distinguishing feature is that the domain models are learned without any observation of the states in the plan or about predicates used to describe them. This works because the objects are grouped into sorts, and the behavior available to objects of any given sort is described by a single parameterized state machine. LOCM is the latest in a number of plan domain learning systems that could be employed to abstract black-box simulations into domain models as part of the Fixer in our proposed IMDE. However, doing so may require learning abstractions from

simulated command sequences, which plan domain learning systems presently do not do.

Techniques for ordering test cases to expose errors more quickly can also be leveraged. Instead of generating test plans by systematically trying each permutation of plan features, test cases may be chosen that are believed to more likely discover a flaw based on results of past cases. The Nemesis test system has had success with this by using a genetic algorithm to smartly choose test cases (Barltrop et al., 2010). A complementary strategy is to use coverage techniques to quickly sweep across the landscape of test cases and learn combinations of features to more quickly converge on a formula describing the conditions under which a flaw appears (Barrett, 2009). This can be used to converge quickly on suggestions to fix modeling errors.

### Challenges in relaxing the assumptions

The usefulness of the described IMDE may still be insufficient because of limiting assumptions. We describe those we deem important and their associated challenges.

It is possible that an action may correspond to multiple commands, a loop, or any arbitrary function generating commands. As long as this function is a legitimate simulator input, then this is not a difficult problem.

Many systems have uncertain behavior, for example, stemming from attitude and temperature control. If the simulation testbed can be invoked in a way that explores different outcomes, then a single plan now corresponds to multiple (possibly infinite) test cases for which the model should be validated. This presents an additional difficulty in determining a tractable number of test cases sufficient for validating the model. It also presents a problem of how to model the activity correctly; if the action duration varies between 30 and 40 seconds, what is the best duration value to use? Moreover, constructing the simulated execution from state values may not be obvious and, in general, can be a difficult state estimation problem!

In addition, the spacecraft may be able to execute sequences conditioned on the perceived system state. This requires simulations that incorporate all possible perceived states that could influence the plan outcomes.

We have discussed some basic examples of modeling errors on preconditions and effects. For expressive language elements such as activity decomposition (as opposed to the mapping of plan actions to simulator command sequences) and parameter dependency functions, how can errors and fixes be automatically identified? Does this similarly extend to errors in abstraction specifications?

Relaxing other assumptions may not pose difficult research challenges but can change the nature of the system capability. For example, if the simulator or system (e.g. spacecraft) does have defects, then discrepancies that are inconsistencies between planned and executed behavior

may now be (in addition to modeling errors) indications of those system defects. So the IMDE now can identify simulator and system defects and validate them against the planner. Thus, the IMDE may more generally be designed for validating multiple systems against each other. This validation is especially important for interactions between autonomous spacecraft subsystems (such as an onboard planner or a guidance, navigation, and control system).

Another assumption was that the simulator is a black box. One option is to treat the effects of an action as properties that are input to a model checker, which is used to directly analyze the simulation model. The System-Level Autonomy Trust Enabler (SLATE) validates a complete model of the system and its operation, incorporating device, control, execution, and planning models (Boddy et al., 2008). The conventional approach of building a model only at an abstract level requires extensive testing of different scenarios and could only be guaranteed to work if all possible scenarios are tested. SLATE only requires testing of individual behaviors whose performance envelopes are incorporated into the model. Since the model of the system is complete, SLATE can prove system-level properties as model checking does.

Another strategy for validating plan abstractions (in particular, those of hierarchical plans) is to summarize the potential constraints and effects of the potential decompositions of each abstract action in the model (Clement et al., 2007). A planner can use this summary information to create a plan whose actions are detailed to different levels necessary to conclude that all further refinements of the plan are either valid or invalid. Like SLATE, summary information validates higher level actions composed of more detailed validated actions. Summary information differs in that abstract actions retain choices of refinement for flexibility of execution, while abstract actions in SLATE are robust to uncertain system behavior. Instead of validation through testing like SLATE and the IMDE, summary information relies on an accurate, detailed model and, thus, applies only to white box simulation, similar to model checking approaches.

A more aggressive approach is to automatically abstract the simulation model to create the planner model, i.e. augment the approach of itSimple (Vaquero et al. 2007) to translate more expressive models to declarative planning languages. Automating such translations requires a deep understanding of the semantics of the simulation language and may not be feasible for all simulation approaches.

### Preliminary Proof of Concept

We have implemented a simple simulator and planner to explore the challenges of building an IMDE. The system provides a two-dimensional slew example and a simple

surface explorer (e.g., rover) example. The system manages a simulator (implemented with ASPEN, Chien et al., 2000) and a planner (EUROPA, Frank and Jónsson, 2003) using an enhanced Eclipse IDE. Simple file formats are used for initial state, simulator commands, output, and executions. A Java library translates these files, supports abstraction specifications, and fulfills the Refinement and Abstraction Engine roles. Currently, the system only detects model errors from single plan simulations.

## Conclusion

The maturation of model-based planning provides an opportunity to improve the state of the art in planning applications. But, the improvement requires spacecraft engineers to build and validate planning models that represent complex constraints derived from diverse information sources. This paper hypothesizes that an Interactive Model Development Environment could overcome many of the associated challenges, providing features to prevent, catch, and repair model errors. While the technologies described above support the described IMDE features, there remain significant research challenges to achieve the overall vision:

- How can a complete but tractable space of test cases be identified for activity model validation?
- Can a single test case contribute to the validation of multiple model elements?
- How can errors in different modeling language features, command refinement, and data abstraction be clearly identified based on simulation output of these tests?
- What are the features of a learning problem for classifying an error?
- How can suggested fixes be discovered for these errors?
- How can test cases be chosen strategically to converge more quickly on modeling error and fix hypotheses?

## Acknowledgements

We gratefully acknowledge the assistance and comments of members of the MER, LCROSS and LADEE mission operations and flight software teams, as well as members of the Johnson Space Center Mission Operations Directorate, in formulating this work. Some of the research described in this paper was performed by the Jet Propulsion Laboratory, California Institute of Technology. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government, or the Jet Propulsion Laboratory, California Institute of Technology. © 2011 California Institute of Technology. Government sponsorship acknowledged.

## References

- Aghevoli, A., Bachmann, A., Bresina, J.L., Greene, J., Kanefsky, R., Kurien, J., McCurdy, M., Morris, P.H., Pyrzak, G., Ratterman, C., Vera, A., Wragg, S., Planning Applications for Three Mars Missions. *Proceedings of the International Workshop on Planning and Scheduling for Space*. Baltimore, MD, 2007.
- Barltrop, K., Clement, B., Horvath, G., and Lee, C. Automated Test Case Selection for Flight Systems using Genetic Algorithms. *Proceedings of the AIAA Infotech@Aerospace Conference*, 2010.
- Barrett, A. and Dvorak, D. A Combinatorial Test Suite Generator for Gray-Box Testing, IEEE SMC-IT 2009.
- Scott Bell, David Kortenkamp, Jack Zaiantz. A Data Abstraction Architecture for Mission Operations. In *Proc. of the International Symposium on AI, Robotics, and Automation in Space*, 2010.
- Boddy, M., Carpenter, T., Shackleton, H., Nelson, K. System-Level Autonomy Trust Enabler (SLATE), In *Proc. of the U.S. Air Force T&E Days*, AIAA, Los Angeles, CA, Feb, 2008.
- Brat, G., Gheorghiu, M., Giannakopoulou, D., “Verification of Plans and Procedures,” In *Proc. of IEEE Aerospace Conf.*, 2008.
- Brooks, R. A. Intelligence without representation. *Artificial Intelligence*. 47, pp. 139–159, 1991.
- Cesta, A., Finzi, A., Fratini, S., Orlandini, A., Tronci, E. Validation and Verification Issues in a Timeline-Based Planning System. *Knowledge Engineering Review*, 25(3): 299-318, 2010.
- Chien, S., Rabideau, G., Knight, R., Sherwood, R., Engelhardt, B., Mutz, D., Estlin, T., Smith, B., Fisher, F., Barrett, T., Stebbins, G., & Tran, D. ASPEN - Automating space mission operations using automated planning and scheduling. In *Proc. SpaceOps*, 2000.
- Clement, B., Durfee, E., Barrett, A. Abstract Reasoning for Planning and Coordination. *Journal of Artificial Intelligence Research*, vol. 28, 453-515, 2007.
- Cresswell, S.; McCluskey, T. L.; and West, M. M. 2012. Acquiring planning domain models using LOCM. *Knowledge Engineering Review*, to appear.
- Fox, M. & Long, D. (2003), PDDL2.1: An extension of PDDL for expressing temporal planning domains, *Journal of Artificial Intelligence Research* 20, 61–124.
- Frank, J. and Jónsson, A. Constraint-Based Interval and Attribute Planning. *Journal of Constraints, Special Issue on Constraints and Planning*, 2003.
- Howey, R. and Long, D. and Fox, M. VAL: automatic plan validation, continuous effects and mixed initiative planning using PDDL. In: *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, 15-17, Nov 2004.
- Izygon, M., Kortenkamp, D., Molin, A., “A Procedure Integrated Development Environment for Future Spacecraft and Habitats,” *Space Technology and Applications International Forum*, 2008.
- Long, D., Fox, M., and Howey, R. Planning Domains and Plans: Validation, Verification and Analysis. In *Proc. Workshop on V&V of Planning and Scheduling Systems*, 2009.
- Raimondi, F., Pecheur, C., and Brat, G. PDVer, a Tool to Verify PDDL Planning Domains. In *Proc. Workshop on Verification and Validation of Planning and Scheduling Systems*, ICAPS, 2009.
- Vaquero, T., Romero, V., Sette, F., Tonidandel, F., Reinaldo Silva, J. ItSimple 2.0: An Integrated Tool for Designing Planning Domains. *Proceedings of the Workshop on Knowledge Engineering for Planning and Scheduling*, 2007.

# Finding Mutual Exclusion Invariants in Temporal Planning Domains

**Sara Bernardini**

London Knowledge Lab  
23-29 Emerald Street, London WC1N 3QS  
s.bernardini@lkl.ac.uk

**David E. Smith**

NASA Ames Research Center  
Moffet Field, CA 94035-1000  
david.smith@nasa.gov

## Abstract

We present a technique for automatically extracting temporal mutual exclusion invariants from PDDL2.2 planning instances. Our technique builds on other approaches to invariant synthesis presented in the literature, but departs from their limited focus on instantaneous discrete actions by addressing temporal and numeric domains. To deal with time, we formulate invariance conditions that account for both the entire structure of the operators (including the conditions, rather than just the effects) and the possible interactions between operators.

## Introduction

A number of planning domain specification languages designed and used to describe complex real-world planning problems adopt a constraint-based representation centered on multi-valued state variables. Examples of large temporal systems based on such languages are: EUROPA2 (Frank and Jónsson, 2003), ASPEN (Chien et al., 2000), IxTeT (Ghallab and Laruelle, 1994), HSTS (Muscettola, 1994) and OMPS (Fratini, Pecora, and Cesta, 2008).

In contrast, the majority of the benchmark domains currently used by the planning community were developed for the International Planning Competitions (IPCs) and are therefore encoded in the PDDL language, which is propositional in nature. Tools designed for translating propositional representations into variable/value representations would facilitate the testing of application-oriented planners on these benchmarks. Designing such tools is primarily concerned with the generation of multi-valued state variables from propositions and operators, which does not depend on the target language of the translation.

This paper presents a technique for generating temporal multi-valued state variables from a PDDL2.2 instance. More specifically, we describe a technique for identifying *temporal mutual exclusion invariants*, which state that certain atoms can never be true at the same time, as a preliminary step to synthesizing state variables. In fact, each identified group of mutually exclusive atoms constitutes the domain of a single state variable.

Our technique builds on the invariant synthesis presented in Helmert (2009) which is used to translate a subset of PDDL2.2 into FDR (Finite Domain Representation),

a multi-valued planning task formalism used within the planner Fast Downward (Helmert, 2006). Helmert’s invariant synthesis is limited to non-temporal and non-numeric PDDL2.2 domains (the so called, PDDL “Level 1”). In contrast, our technique addresses temporal and numeric domains (PDDL – “Level 3”). Developing invariants for such tasks is more complex than handling tasks with instantaneous discrete actions, because interference between concurrent operators complicates the identification of state variables. For this reason, a simple generalization of Helmert’s approach does not work in temporal settings. In extending the theory to capture the temporal case, we have had to formulate invariance conditions that take into account the entire structure of the operators (including the conditions, as opposed to the effects only) as well as the possible interactions between them. As a result, we have constructed a significantly more comprehensive technique that is able to find not only invariants for temporal domains, but also a broader set of invariants for non-temporal domains.

This paper is organized as follows. We first identify a set of initial invariant candidates by inspecting the domain. We then check these candidates against a set of properties that assure invariance. If a candidate turns out not to be an invariant, we show that in some cases it is possible to refine it so as to make it a real invariant. An experimental evaluation of our approach and a presentation of conclusions and future work close the paper.

## Invariant Candidates

An *invariant* is a property of world states such that when it is satisfied by a state  $s$ , it is satisfied by all states that are reachable from  $s$ . Usually, we are interested in invariants that are satisfied in the initial state. If an invariant holds in the initial state, it holds in all the reachable states. Here, we focus on *mutual exclusion* invariants, which state that certain atoms can never be true at the same time. For example, if we take the *Logistics* domain, a mutual exclusion invariant for this domain states that two atoms indicating the position of a truck  $\text{trk0}$ , such as  $\text{at}(\text{trk0}, \text{loc0})$  and  $\text{at}(\text{trk0}, \text{loc1})$ , can never be true at the same time. Intuitively, this means that the truck cannot be at two different positions simultaneously.

More formally, let  $I = (\mathcal{D}, \mathcal{P})$  be a PDDL instance, where  $\mathcal{D}$  is a planning domain and  $\mathcal{P}$  a planning problem, an *in-*



**variant candidate** is a tuple  $\mathcal{C} = \langle \Phi, F, V \rangle$ , where  $\Phi$  is a non-empty subset of the atoms in the domain  $\mathcal{D}$ , and  $F$  and  $V$  are two disjoint sets of variables. The atoms in  $\Phi$  are called the candidate's *components*, while the two sets  $F$  and  $V$  are respectively called *fixed* and *counted* variables. They are both subsets of  $\text{Var}[\Phi]$ , which collects the variables in  $\Phi$ . For example, if we take the *Logistics* domain and the predicate  $\text{at}(\text{truck}, \text{loc})$ , the following is a candidate:  $\mathcal{C}_{at} = \langle \{\text{at}(\text{truck}, \text{loc})\}, \{\text{truck}\}, \{\text{loc}\} \rangle$ , where  $\text{at}(\text{truck}, \text{loc})$  is the only component of this candidate,  $\text{truck}$  is the fixed variable and  $\text{loc}$  the counted variable.

An **instance**  $\gamma$  of the candidate  $\mathcal{C}$  is a function that maps the fixed variables in  $F$  to objects of the problem  $\mathcal{P}$ . Assuming we have a problem with two trucks  $\text{trk1}$  and  $\text{trk2}$ , we have two possible instances of  $\mathcal{C}_{at}$ :  $\gamma_{\text{trk1}}: \text{truck} \rightarrow \text{trk1}$  and  $\gamma_{\text{trk2}}: \text{truck} \rightarrow \text{trk2}$ .

The **weight** of an instance  $\gamma$  in a state  $s$  is the number of ground instantiations of the variables in  $V$  that make some  $\phi \in \Phi$  true under  $\gamma$  in  $s$ .<sup>1</sup> Thus, considering the *Logistics* domain and the instance  $\gamma_{\text{trk1}}$ , if we have a state  $s$  where the atom  $\text{at}(\text{trk1}, \text{loc1})$  holds, then the weight of  $\mathcal{C}_{at}$  is one. Intuitively, the weight of  $\gamma$  in a state  $s$  is the number of the candidate's components that are true in  $s$  when the fixed variables have been instantiated according to  $\gamma$ .

Given a **cardinality set**  $S = \{x \mid x \in \mathbb{N}\}$ , the semantics of a candidate  $\mathcal{C}$  is: for all the possible instances  $\gamma$  of  $\mathcal{C}$ , if the weight of  $\gamma$  is within  $S$  in a state  $s$ , then it is within  $S$  in any successor state  $s'$  of  $s$ . Thus, if we prove that the candidate  $\mathcal{C}$  holds (i.e.  $\mathcal{C}$  is an invariant) and is satisfied in the initial state, we have that at most  $k = \max(S)$  atoms in  $\Phi$  are true in any reachable state. Since we focus on finding mutually exclusive sets of propositions, we are interested in cases in which at most one atom in  $\Phi$  is true in any reachable state. Considering the *Logistics* domain again, the candidate  $\mathcal{C}_{at}$  means that, for each truck  $\text{trk}$  in the domain, if the number of locations  $\text{loc}$  where  $\text{at}(\text{trk}, \text{loc})$  is true is at most one in a state  $s$ , then it is at most one in any successor state  $s'$  of  $s$ . If we prove that what is stated by the candidate is true and each truck is at a maximum of one location in the initial state, then each truck cannot be at multiple locations at the same time in any reachable state. Hence, for each truck, we can create a state variable that corresponds to the predicate  $\text{at}$  and represents the position of the truck. The values of this variable represent the presence of the truck in the various locations that it can occupy.

In Helmert's work, he considers only the cardinality set  $S = \{1\}$ . However, we consider the set  $S = \{0, 1\}$  because, with durative actions, it is common for a proposition to be deleted at the beginning of an action (e.g. the location of an object being moved), and replaced by a new proposition at the end of the action (e.g. the new location of the object). This corresponds to a decrease in the weight of  $\gamma$  to zero at the beginning of the action, and an increase back to one at the end. Allowing  $S = \{0, 1\}$  could be useful in non-temporal domains as well, since it allows operators bringing the weight from zero to one to be classified as safe for invari-

ance conditions. This approach therefore allows us to find more invariants than the techniques using only  $S = \{1\}$ . Although we focus here on  $S = \{0, 1\}$ , our technique for finding invariants can be generalized to larger cardinality sets.

## Invariance Conditions

In order to show that a candidate  $\mathcal{C}$  is an actual invariant, we need to guarantee that, for any instance  $\gamma$  of  $\mathcal{C}$ , the weight of  $\gamma$  is within the cardinality set  $S = \{0, 1\}$  in the initial state and all the operators in the domain  $\mathcal{D}$  keep the weight within this set. When an operator satisfies this condition, we say that it is *safe* and so it does not *threaten* the candidate  $\mathcal{C}$ .

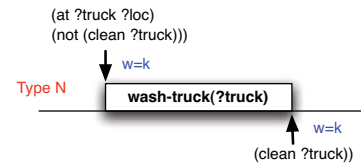
More formally, given an instance  $\gamma$  of a candidate  $\mathcal{C}$ , an operator  $op$  is *safe* if, for any situation where: i) the weight of  $\gamma$  is less than or equal to one prior to executing  $op$  and ii) it is legal to execute  $op$ , the weight of  $\gamma$  is guaranteed to remain less than or equal to one through the execution of  $op$  and immediately following  $op$ . A domain  $\mathcal{D}$  is safe for  $\mathcal{C}$  if and only if all operators in  $\mathcal{D}$  are safe for any instance  $\gamma$  of  $\mathcal{C}$ .

A *sufficient condition* for  $\mathcal{C}$  to be an actual invariant is that the domain is safe for  $\mathcal{C}$ .

Given a candidate  $\mathcal{C}$  and an instance  $\gamma$ , when can we ensure that an operator  $op$  is safe, i.e. maintains the weight of  $\gamma$  within the cardinality set  $S = \{0, 1\}$ ? Clearly, if the operator does not change the weight of  $\gamma$ , then it is safe. On the other hand, if an operator increases the weight of  $\gamma$  by two or more at any time-point, it is definitely not safe. If the operator increases the weight of  $\gamma$  by one, there might be circumstances in which it is safe, depending on the structure of the conditions and the effects of the operator itself and on its interactions with other operators.

Given an instance  $\gamma$  of a candidate  $\mathcal{C}$ , an operator  $op$  is safe if and only if it falls in one of the following six categories:

1. **Type N - Inert.** The operator  $op$  does not affect the weight of  $\gamma$ . Clearly, an inert operator is safe because it preserves the weight of  $\gamma$ . Considering a simple *Logistics* domain, the figure below shows an example of such an operator with respect to the candidate  $\mathcal{C} = \langle \{\text{at}(\text{truck}, \text{loc})\}, \{\text{truck}\}, \{\text{loc}\} \rangle$ .



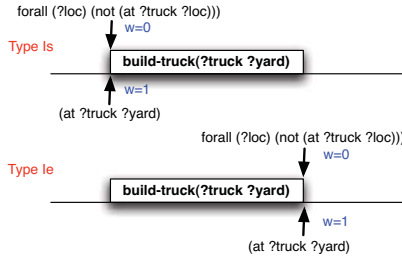
2. **Type D: Decreasing.** The operator  $op$  decreases the weight of  $\gamma$  at some time-point, and does not increase it at any time point. A decreasing operator may or may not have a condition on  $\gamma$ , and the decrease may even be universally quantified. Like an inert operator, a decreasing operator is safe because it does not cause an increase in the weight at any time-point, and therefore maintains the weight within the cardinality set  $S = \{0, 1\}$ . The figure below shows one of several possible decreasing operators with respect to the candidate  $\mathcal{C} = \langle \{\text{at}(\text{truck}, \text{loc})\}, \{\text{truck}\}, \{\text{loc}\} \rangle$ .

<sup>1</sup>The weight of  $\gamma$  is equal to the cardinality of the set of all ground atoms that unify with some  $\phi \in \Phi$  under  $\gamma$  in  $s$ .

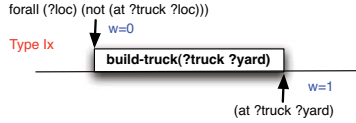


3. **Type I: Increasing.** The operator  $op$  increases the weight of  $\gamma$  from zero to one. We identify three possible sub-cases:

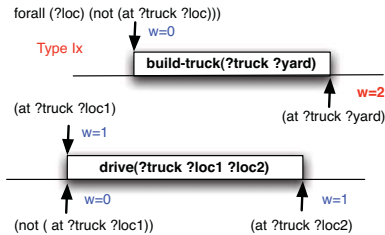
- **Types Is and Ie:** The operator  $op$  increases the weight of  $\gamma$  by one at some time-point (start/end) and its conditions require that the weight of  $\gamma$  be zero at the same time-point (start/end). Increasing operators of type Is and Ie are safe because they bring the weight from zero to one at just one time-point. The figure below shows an increasing operator at start and an increasing operator at end with respect to the candidate  $\mathcal{C} = \langle \{at(truck, loc)\}, \{truck\}, \{loc\} \rangle$ .



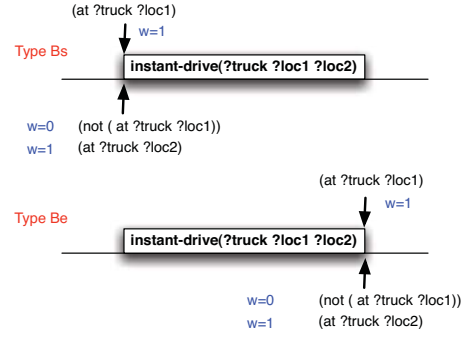
- **Type Ix:** a condition at start guarantees that the weight of  $\gamma$  is zero and an add effect at end increases the weight by one. The figure below shows an example of such an operator with respect to the candidate  $\mathcal{C} = \langle \{at(truck, loc)\}, \{truck\}, \{loc\} \rangle$ .



An operator of type Ix is safe if it is mutex with all those operators that may increase the weight of  $\gamma$  over its duration. The following picture shows a simple example of when this might happen.



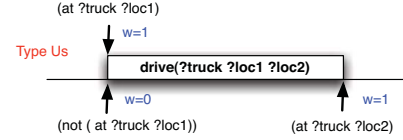
4. **Type B: Balanced.** The operator  $op$  preserves the weight of  $\gamma$  by checking that the weight is one at some time-point (start/end), decreasing the weight by one at that time-point and then bringing back the weight to one at that same time-point. Balanced operators are always safe because they act at only one time-point (start/end) and do not change the overall weight of  $\gamma$ . The figure below shows a balanced operator at start (**Type Bs**) and a balanced operator at end (**Type Be**) with respect to the candidate  $\mathcal{C} = \langle \{at(truck, loc)\}, \{truck\}, \{loc\} \rangle$ .



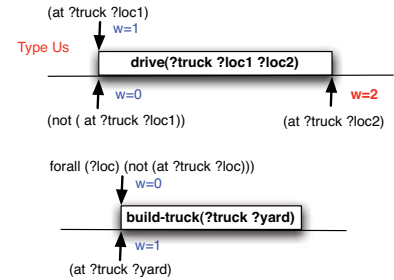
5. **Type U: Temporarily Unbalanced.** The operator  $op$  ensures that the weight of  $\gamma$  is one at start, brings the weight from one to zero at start or at end and then restores the weight to one at end.

We have two different configurations for a temporarily unbalanced operator:

- **Type Us:** a condition at start guarantees that the weight is one, a delete effect at start decreases the weight from one to zero, and an add effect at end restores the weight to one. The figure below shows an example of such an operator with respect to the candidate  $\mathcal{C} = \langle \{at(truck, loc)\}, \{truck\}, \{loc\} \rangle$ .

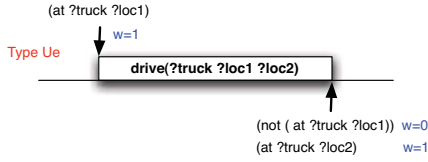


An unbalanced operator of type Us is safe if it is mutex with all those operators that may increase the weight of  $\gamma$  over its duration. The following picture shows a simple example of when this might happen.

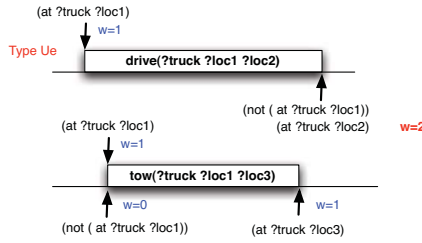


Unbalanced operators of type Us are particularly common because they model the usage of renewable resources. A renewable resource is needed during the execution of the action, so the weight goes from one to zero at start, but it is not consumed by the action, so the weight returns to one at end.

- **Type Ue:** a condition at start guarantees that the weight is one and a delete and an add effect at end bring the weight from one to zero and then back to one. The figure below shows an example of such an operator with respect to the candidate  $\mathcal{C} = \langle \{at(truck, loc)\}, \{truck\}, \{loc\} \rangle$ .



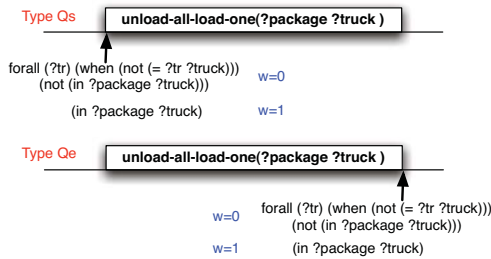
An unbalanced operator of type Ue is safe if it is mutex with all operators that may alter the weight during its execution. Although this operator does not cause an overall change in the weight of  $\gamma$  when executed in isolation, it might give rise to problematic situations when another operator  $op_i$  capable of changing the weight is allowed to take place over its duration. This is because the application of  $op_i$  may have the side effect of making the delete effect of  $op$  no longer applicable, which would in turn provoke an overall increase of the weight by two instead of one. The figure below exemplifies this situation.



One could argue that unbalanced operators of type Ue originate from a faulty description of renewable resources and so they should in reality be operators of type Us. We have found a few examples of operators of type Ue in the domains of previous IPCs, but none resulted in provable invariants.

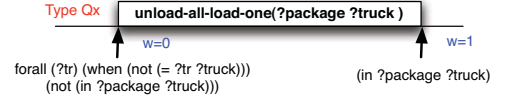
6. **Type Q: Quantified Delete.** The operator  $op$  sets the weight of  $\gamma$  to zero at some time-point (start/end) through a universally quantified delete effect and then brings back the weight to one at the same time-point (start/end) or after that. We distinguish three possible sub-cases:

- **Types Qs and Qe:** a universally quantified effect sets the weight to zero at some time-point (start/end) and an add effect increases the weight by one at the same time-point (start/end). Operators of type Qs and Qe are safe because they ensure that only the single add effect will be true. The figure below shows an example of such operators with respect to the candidate  $C = \langle \{in(package, truck)\}, \{package\}, \{truck\} \rangle$ .

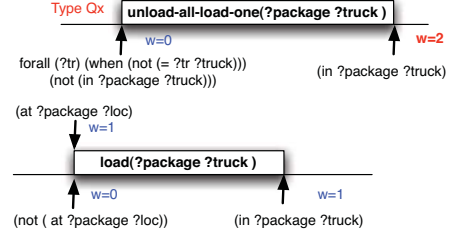


- **Type Qx:** a universally quantified effect at start sets the weight to zero and an add effect at end increases

the weight by one. The figure below shows an example of such an operator with respect to the candidate  $C = \langle \{in(package, truck)\}, \{package\}, \{truck\} \rangle$ .



An unbalanced operator of type Qx is safe if it is mutex with all those operators that may alter the weight during its execution. The following picture shows an example of when this might happen.



Inert and balanced safe operators represent the temporal generalization of the non-threatening operators used in Helmert's invariant synthesis (Helmert, 2009). The criteria for identifying increasing, decreasing and quantified delete operators can be readapted for use in non-temporal planning domains. They correspond to the use of the cardinality set  $S = \{0, 1\}$  instead of  $S = \{1\}$ , which allows us to capture a broader set of invariants than Helmert's approach. In contrast, unbalanced operators are specific to temporal planning and correspond to cases where the effects of an action are not fully realized until the end. Such operators can still be safe, as long as no other operator can disrupt the candidate during the execution of the operator.

## Temporal Mutex Conditions

We now clarify the exact nature of the temporal mutex conditions that must hold in order to ensure the safeness of unbalanced operators and operators whose effects are split over time, such as operators of type Dx, Ix, and Qx.

In order to assess if an operator  $op$  is safe, we first need to establish what kinds of operators may disrupt the weight during the execution of  $op$  and then specify the exact mutex relationships that must hold between  $op$  and the possibly disrupting operators.

Let us consider the second issue first. In general, how can we establish whether two durative PDDL operators are mutex or not? Since in PDDL2.2, effects can only happen at the start and end of the operators, and conditions can only be specified at the start, end, and over all, there are nine types of mutex. We refer the reader to (Smith and Jónsson, 2002) for a discussion of mutex between actions with general conditions and effects.

Given two durative operators  $op_1$  and  $op_2$ , these nine types of *mutex operators* are the following:

1. **Start-Start:**  $op_1$  and  $op_2$  cannot start at the same time if:
 
$$\exists p \in (\text{Cond}_{start}(op_1) \cup \text{Cond}_{all}(op_1) \cup \text{Eff}_{start}(op_1)) : \neg p \in (\text{Cond}_{start}(op_2) \cup \text{Cond}_{all}(op_2) \cup \text{Eff}_{start}(op_2))$$

2. **End-End:**  $op_1$  and  $op_2$  cannot end at the same time if:  
 $\exists p \in (\text{Cond}_{\text{end}}(op_1) \cup \text{Cond}_{\text{all}}(op_1) \cup \text{Eff}_{\text{end}}(op_1)) :$   
 $\neg p \in (\text{Cond}_{\text{end}}(op_2) \cup \text{Cond}_{\text{all}}(op_2) \cup \text{Eff}_{\text{end}}(op_2))$
3. **Start-End:**  $op_1$  cannot start at the time that  $op_2$  ends if:  
 $\exists p \in (\text{Cond}_{\text{start}}(op_1) \cup \text{Cond}_{\text{all}}(op_1) \cup \text{Eff}_{\text{start}}(op_1)) :$   
 $\neg p \in (\text{Cond}_{\text{end}}(op_2) \cup \text{Eff}_{\text{end}}(op_2))$
4. **Invariant-Start:**  $op_2$  cannot start during  $op_1$  if:  
 $\exists p \in \text{Cond}_{\text{all}}(op_1) :$   
 $\neg p \in (\text{Cond}_{\text{start}}(op_2) \cup \text{Cond}_{\text{all}}(op_2) \cup \text{Eff}_{\text{start}}(op_2))$
5. **Invariant-End:**  $op_2$  cannot end during  $op_1$  if:  
 $\exists p \in \text{Cond}_{\text{all}}(op_1) :$   
 $\neg p \in (\text{Cond}_{\text{end}}(op_2) \cup \text{Cond}_{\text{all}}(op_2) \cup \text{Eff}_{\text{end}}(op_2))$
6. **Invariant-Invariant:**  $op_1$  and  $op_2$  cannot overlap if:  
 $\exists p \in \text{Cond}_{\text{all}}(op_1) : \neg p \in \text{Cond}_{\text{all}}(op_2)$

In addition, we have: 7. mutex **End-Start** (dual to case 3), 8. mutex **Start-Invariant** (dual to case 4) and 9. mutex **End-Invariant** (dual to case 5). For brevity, we refer to such mutex as *mutex-SS*, *mutex-EE*, and so on.

As for identifying possibly disrupting operators, we need to reason about the operators in the domain according to two criteria: i) what type of legal weight change they produce (from zero to one, from one to zero or from one to zero to one); and ii) at what time-points the changes happen.

Following this reasoning, for each type of unbalanced operator  $op$ , we identify a set of **mutex constraints** that involve  $op$  and those operators that can possibly disrupt its weight. If these constraints are satisfied, then  $op$  is safe.

- An increasing operator of type  $Ix$  is safe if it is:
  1. mutex  $IS$  with any operator of type  $(I,Q)s$
  2. mutex  $IE$  with any operator of type  $Us, (I,Q)x$ , and  $(I,Q)e$
- An unbalanced operator of type  $Us$  is safe if it is:
  1. mutex  $IS$  with any operator of type  $(I,Q)s$
  2. mutex  $IE$  with any operator of type  $Us, (I,Q)x, (I,Q)e$
- An unbalanced operator of type  $Ue$  is safe if it is:
  1. mutex  $IS$  with any operator of type  $(I,Q,B)s$
  2. mutex  $IE$  with any operator of type  $Us, (I,Q)x$ , and  $(I,Q,B,U)e$
- A quantified delete operator of type  $Qx$  is safe if it is:
  1. mutex  $IS$  with any operator of type  $(I,Q)s$
  2. mutex  $IE$  with any operator of type  $Us, (I,Q)x$ , and  $(I,Q,U)e$

## Decision Tree

In Figure 1, we show a binary **decision tree**  $\mathcal{T}$  that can be used to determine whether an operator  $op$  is safe w.r.t an instance  $\gamma$  of a candidate  $\mathcal{C}$  or not. The internal nodes of the tree test the structure of the conditions and effects of the operator. The abbreviations stand for: Add-s  $\rightarrow$  add effect at start, Del-s  $\rightarrow$  delete effect at start, W=0-s  $\rightarrow$  weight is zero at start, UQ del-s  $\rightarrow$  universally quantified delete effect at start. Abbreviations for conditions and effects at end are

analogous. On the basis of the configuration of the conditions and effects of the operator  $op$ , the leaf nodes assign a classification: either  $op$  is safe or it is unsafe (respectively, “OK” and “X” in the tree). The leaves of the tree marked with “OK” represent all the possible cases in which we accept an operator as safe. Green labels in the figure link these cases with the five categories of safe operators described above. Close to the corresponding branches of the tree, we also give a graphical representation of the configuration of the operator’s conditions and effects. It is worth noting that a few of the operators in the tree are quite bizarre and unlikely to appear in practice. For example, operators of type 1 (such as *Isle*) could not even be executed without required concurrency – some other operator would have to reduce the weight back to zero in the middle. Nevertheless, we have included these operators in the tree for completeness.

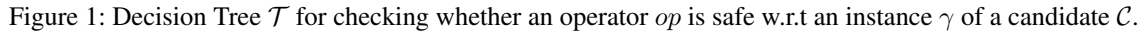
## Guess, Check and Repair Algorithm

As with other related techniques (Gerevini and Schubert, 2000; Helmert, 2009), our algorithm for finding invariants implements a *guess, check and repair* approach. We start from a simple set of initial candidates and use the decision tree in Figure 1 to evaluate if each candidate  $\mathcal{C}$  is an invariant. If we reach a failure leaf for any operator  $op$  in the domain, before discarding  $\mathcal{C}$ , we identify what features of  $op$  threaten  $\mathcal{C}$  and exploit this knowledge for creating new candidates that are guaranteed not to be threatened by the same operator  $op$ . These new candidates need to be checked against the invariance conditions and might fail due to different threatening operators. The tree in Figure 1 associates, whenever possible, a set of fixes to dead leaves.

When we create the set of initial candidates, we ignore constant predicates, i.e. predicates whose atoms have the same truth value in all the states (for example, type predicates). In fact, they are trivially invariants and so are typically not interesting. Among the modifiable atoms, we use initial predicates with the following characteristics: the set  $\Phi$  contains only one atom  $\phi$  and the set  $V$  contains only one counted variable. The candidate  $\mathcal{C}_{\text{at}} = \{\text{at}(\text{truck}, \text{loc}), \{\text{truck}\}, \{\text{loc}\}\}$  is an example of an initial candidate. This choice comes from experience and is the same as for other related techniques (Helmert, 2009).

Given an initial candidate, we test the safety of each operator in the domain by traversing the decision tree in Figure 1. The main difficulty associated with traversing the tree is that we can check the mutex constraints associated with some branches of the tree only when we know the type of each operator. The simplest way to handle this is to make two iterations: the first to classify operators according to types and the second to check the operators. However, we follow a more efficient approach by checking most of the operators during the first iteration, and just returning to do the mutex checks for those operators that require them, after all of the operators have been classified. We apply the following procedure:

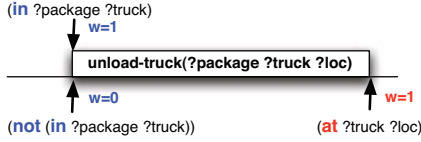
1. Select a candidate invariant  $\mathcal{C}$  and traverse the decision tree  $\mathcal{T}$  for each operator in the domain  $\mathcal{D}$ .



- As an example, consider the *Logistics* domain with the operator `unload-truck`, shown in the figure below. For the candidate  $\mathcal{C}_{at} = \langle \{\text{at}(\text{package}, \text{loc})\}, \{\text{package}\}, \{\text{loc}\} \rangle$ , we see that operator `unload-truck` threatens  $\mathcal{C}_{at}$  because it increases the weight at end without decreasing

1. **Fix SS:** the atom  $\phi$  unifies with a positive condition at start and a delete effect at start of  $op$ .
2. **Fix EE:** the atom  $\phi$  unifies with a positive condition at end (or over all) and a delete effect at end of  $op$ .





it or checking that the weight is zero. If we traverse the tree in Figure 1 guided by the conditions and effects of the operator `unload-truck`, we reach leaf 22. Although this is a failure leaf, it indicates that, before discarding  $\mathcal{C}_{at}$ , we can try to apply fixes SS, EE and SE. Fix SS can be used in this case because the atom  $\phi = \text{in}(\text{?package } \text{?truck})$  appears both in the positive conditions at start and in the delete effects at start. Therefore, we add the candidate  $\mathcal{C}_{at/in} = \{\{\text{at}(\text{package}, \text{loc}), \text{in}(\text{package}, \text{truck})\}, \{\text{package}\}, \{\text{loc}, \text{truck}\}\}$  to the list of candidates to check. By evaluating the new candidate  $\mathcal{C}_{at/in}$  against the invariance conditions, we will conclude that  $\mathcal{C}_{at/in}$  is in fact an invariant.

## Experimental Results

In this section, we present some experimental results for the invariant synthesis technique developed above. The current version of the algorithm is implemented in the Python language. The experiments were conducted by using a 2.53 GHz Intel Core 2 Duo processor with a memory of 4 GB.

Below, we present the invariants that the algorithm finds for some temporal domains of the IPC-2008. Each invariant is enclosed in braces where the predicate names indicate the components of the invariant, numbers not enclosed in square brackets indicate the position of the fixed variables in the list of arguments of the corresponding predicate and numbers enclosed in square brackets indicate the counted variables. For example, considering our running example,  $\{\text{at } 0 [1], \text{in } 0 [1]\}$  indicates the invariant having  $\{\text{at}(\text{package}, \text{location}) \text{ in}(\text{package}, \text{vehicle})\}$  as components, `package` as a fixed variable, and  $\{\text{location}, \text{vehicle}\}$  as counted variables.

- *Elevators-strips*:  

```
{passengers 0 [1]}
{lift-at 0 [1]}
{passenger-at 0 [1], boarded 0 [1]}
```
- *Sokoban-strips*:  

```
{at 0 [1]}
{at 1 [0], clear 0}
{clear [0]}
```
- *Openstacks-adl*:  

```
{stacks-avail [0]}
{waiting [0], started [0], shipped [0]}
{waiting 0, started 0, shipped 0}
{started [0], waiting [0]}
{started 0, waiting 0}
{waiting [0]}
{waiting 0}
```

Table 1 compares the number of invariants (# INV) found by the Temporal Invariant Synthesis (TIS) just discussed with those found by a Simple version of the Invariant Synthesis (SIS) for the temporal domains of the IPC-6, IPC-5,

IPC-4 and IPC-3. The SIS represents a simple generalization of Helmert's invariant synthesis (Helmert, 2009) to the temporal case.

Table 1 also reports the number of invariants obtained by applying fixes (# FIX) and run time (RT) for generating invariants for the temporal domains. The computational time is negligible; there is no significant delay associated with either checking a broad set of configurations in the operators' conditions and effects or performing the mutex checks.

For the invariants found by our algorithm, the most common operators are of type 23, which means that the operator does not even potentially threaten the invariant because it is inert or decreasing, and type 15, which corresponds to the usage of a renewable resource. We also found operators of types 6c, 11, 15, and 23. Additional operators of types 8, 12, 16, 17, 18, and 22 were found while examining invariant candidates that were ultimately rejected. Based on our discussions with Cushing (Cushing et al., 2007), these operators' types appear to be consistent with his analysis.

Table 2 shows a comparison between the number of state variables obtained by instantiating invariants for domains of the IPC-6 coming from a Naive Invariant Synthesis (NIS), which basically produces a state variable with two truth values (true and false) for each atom in the domain, the Simple Invariant Synthesis (SIS), and our Temporal Invariant Synthesis (TIS). In many domains the TIS yields a significant reduction in the number of state variables in comparison with the other two techniques. In six instances of Elevators-str, Sokoban-str, and Transport-Num the reduction is greater than an order of magnitude.

## Conclusions and Future Work

In this paper, we presented a technique for automatically synthesizing invariants starting from temporal planning domains expressed in PDDL2.2. Our technique builds on Helmert's invariant synthesis (Helmert, 2009), but extends it to apply to temporal domains and also identifies a broader set of invariants. This is achieved by considering the cardinality set  $S = \{0, 1\}$  instead of  $S = \{1\}$  and by analyzing the entire structure of an operator to assess its safety with respect to an invariant. Finding a wider set of invariants allows us to synthesize a smaller number of state variables to represent a domain. All the temporal planners that use state variables to represent the world greatly benefit from dealing with a relatively small number of state variables.

Our technique can be incorporated in any translation from PDDL2.2 to a language based on multi-valued state variables. In particular, we have used the temporal invariant synthesis described here in our translator from PDDL2.2 to NDDL, EUROPA2's domain specification language (Bernardini and Smith, 2008). The use of this translator, which includes the temporal invariant synthesis described here as one of its core steps, has facilitated the testing of EUROPA2 against domains of the IPCs originally expressed in PDDL2.2.

In the future, we intend to use information about *types*, which are available in PDDL2.2 domains, for identifying a more comprehensive set of invariants. As an example, let us consider a domain in which we have a predicate

Domains	# INV SIS	# INV TIS	# FIX TIS	RT TIS
Crew Planning-IPC-6	0	3	0	0.0054
Elevators-Num-IPC-6	0	2	1	0.0025
Elevators-Str-IPC-6	0	3	1	0.0037
Modeltrain-Num-IPC-6	3	7	1	0.0089
Openstacks-Adl-IPC-6	2	7	4	0.0043
Openstacks-Num-IPC-6	4	10	6	0.0054
Openstacks-Num-Adl-IPC-6	2	6	4	0.0030
Openstacks-Str-IPC-6	4	11	6	0.0073
Parcprinter-Str-IPC-6	5	7	2	0.0126
Pegsol-Str-IPC-6	0	2	1	0.0008
Sokoban-Str-IPC-6	0	3	1	0.0033
Transport-Num-IPC-6	0	3	1	0.0030
Woodworking-Num-IPC-6	2	7	3	0.0167
Openstacks-IPC-5	2	7	4	0.0048
Pathways-IPC-5	0	0	0	0.0003
Pipesworld-IPC-5	0	8	7	0.0266
Rovers-IPC-5	4	9	0	0.0142
Storage-IPC-5	0	3	2	0.0071
TPP-IPC-5	0	1	0	0.0006
Trucks-IPC-5	0	2	2	0.0055
Airport-IPC-4	2	2	0	0.0399
Pipesworld-NT-IPC-4	0	4	4	0.0162
Pipesworld-T-IPC-4	0	8	7	0.0270
Satellite-IPC-4	0	2	1	0.0027
UMTS-4	0	0	0	0.0079
Depots-IPC-3	0	6	5	0.0113
DriverLog-IPC-3	0	2	2	0.0051
ZenoTravel-IPC-3	0	1	1	0.0031
Rovers-IPC-3	4	9	0	0.0137
Satellite-IPC-3	0	2	1	0.0027

Table 1: Number of invariants (# INV), number of invariants coming from fixes (# FIX) and run time (RT) for generating invariants for the temporal domains of the IPCs by using the Temporal Invariant Synthesis (TIS) and the Simple Invariant Synthesis (SIS).

(pred ?arg1 - supertype ?arg2 - type)  
and the types subtype1 and subtype2 are both of type supertype. Given an invariant candidate  $C = \langle \{pred(arg1, arg2)\}, \{arg1 - supertype\}, \{arg2 - type\} \rangle$ , suppose that no operator threatens  $C$  when  $arg1$  is bound to an object of type subtype1, but an operator  $op$  threatens  $C$  when  $arg1$  is bound to an object of type subtype2. In this case, our algorithm rejects the candidate and, if no fix involving  $pred$  can be applied, the algorithm encodes  $pred$  with binary state variables. However, if we enrich the algorithm with the ability to use information about types, it will consider two more specific candidates  $C_1 = \langle \{pred(arg1, arg2)\}, \{arg1 - subtype1\}, \{arg2 - type\} \rangle$  and  $C_2 = \langle \{pred(arg1, arg2)\}, \{arg1 - subtype2\}, \{arg2 - type\} \rangle$ . Now, the algorithm will accept  $C_1$  as an invariant since it is not threatened by any operator, while it will fail  $C_2$  since  $op$  threatens it.

## Acknowledgments

We thank Malte Helmert and Gabriele Röger for making their code for translating PDDL into FDR available and William Cushing for helpful discussions about the configu-

Domains	# SV		
	NIS	SIS	TIS
Crew Planning - p10	112	112	106
Crew Planning - p20	305	305	261
Crew Planning - p30	510	510	498
Elevators-Str - p10	203	203	21
Elevators-Str - p20	592	592	34
Elevators-Str - p30	1240	1240	49
Openstacks-Num - p10	71	71	29
Openstacks-Num - p20	121	121	49
Openstacks-Num - p30	171	171	69
Modeltrain-Num - p10	397	205	191
Modeltrain-Num - p20	396	204	188
Modeltrain-Num - p30	910	418	390
Parcprinter-Str - p10	641	641	431
Parcprinter-Str - p20	1273	1273	673
Parcprinter-Str - p30	669	669	439
Pegsol-Str - p10	66	66	33
Pegsol-Str - p20	66	66	33
Pegsol-Str - p30	66	66	33
Sokoban-Str - p10	490	490	72
Sokoban-Str - p20	127	127	37
Sokoban-Str - p30	1131	1131	75
Transport-Num - p10	1292	1292	36
Transport-Num - p20	1292	1292	36
Transport-Num - p30	1772	1772	64
Woodworking-Num - p10	143	143	95
Woodworking-Num - p20	239	239	151
Woodworking-Num - p30	251	251	158

Table 2: Number of state variables (# SV) for temporal domains of the IPC-6 that are obtained by instantiating invariants coming from: (1) a Naive Invariant Synthesis (NIS); (2) a Simple Invariant Synthesis (SIS); and (3) our Temporal Invariant Synthesis (TIS).

rations of temporal operators. We are grateful to the anonymous reviewers for their suggestions on earlier drafts of the paper. This work has been supported by the London Knowledge Lab and the NASA Exploration Systems Program.

## References

- Bernardini, S., and Smith, D. E. 2008. Translating pddl2.2. into a constraint-based variable/value language. In *Proc. of the Workshop on Knowledge Engineering for Planning and Scheduling, 18th International Conference on Automated Planning and Scheduling (ICAPS'08)*.
- Chien, S.; Rabideau, G.; Knight, R.; Sherwood, R.; Engelhardt, B.; Mutz, D.; Estlin, T.; B. Smith; Fisher, F.; Barret, T.; Stebbins, G.; and Tran, D. 2000. ASPEN - Automated planning and scheduling for space missions operations. In *6th International Conference on Space Operations*.
- Cushing, W.; Weld, D.; Kambhampati, S.; Mausam; and Talamadupula, K. 2007. Evaluating temporal planning domains. In *Proc. of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS-07)*, 105–112.
- Frank, J., and Jönsson, A. 2003. Constraint based attribute and interval planning. *Journal of Constraints* 8(4):339–364. Special Issue on Planning.
- Fratini, S.; Pecora, F.; and Cesta, A. 2008. Unifying Planning and Scheduling as Timelines in a Component-Based Perspective. *Archives of Control Sciences* 18(2):5–45.
- Gerevini, A., and Schubert, L. 2000. Discovering state constraints in discoplan: Some new results. In *In Proc. of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, 761–767.
- Ghallab, M., and Laruelle, H. 1994. Representation and control in IxTeT, a temporal planner. In *Proc. of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, 61–67. AAAI Press.
- Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.
- Helmert, M. 2009. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence* 3(17):503–535.
- Muscettola, N. 1994. HSTS: Integrating planning and scheduling. In Zweben, M., and Fox, M., eds., *Intelligent Scheduling*. Morgan Kaufmann, 451–469.
- Smith, D., and Jönsson, A. 2002. The logic of reachability. In *Proc. of the Sixth International Conference on AI Planning and Scheduling (AIPS-02)*, 379–387.

# Using Planning Domain Features to Facilitate Knowledge Engineering\*

**Gerhard Wickler**

Artificial Intelligence Applications Institute  
University of Edinburgh  
Edinburgh, Scotland

## Abstract

This paper defines a number of features that can be used to characterize planning domains, namely *domain types*, *relation fluency*, *inconsistent effects* and *reversible actions*. These features can be used to provide additional information about the operators defined in a STRIPS-like planning domain. Furthermore, the values of these features may be extracted automatically; efficient algorithms for this are described in this paper. Alternatively, where these values are specified explicitly by the domain author, the extracted values can be used to validate the consistency of the domain, thus supporting the knowledge engineering process. This approach has been evaluated using a number of planning domains, mostly drawn from the international planning competition. The results show that the features provide useful information, and can highlight problems with the manual formalization of planning domains.

## Introduction

Specifying a planning domain and a planning problem in a formal description language defines a search space that can be traversed by a state-space planner to find a solution plan. It is well known that this specification process, also known as *problem formulation* (Russell and Norvig 2003), is essential for enabling efficient problem-solving through search (Amarel 1968).

The Planning Domain Definition Language (PDDL) (Fox and Long 2003) has become a de-facto standard for specifying STRIPS-like planning domains and problems with various extensions. PDDL allows for the specification of some auxiliary information about a domain, such as types, but this information is optional.

---

\*This work has been sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-09-1-3090. The University of Edinburgh and research sponsors are authorized to reproduce and distribute reprints and on-line copies for their purposes notwithstanding any copyright annotation hereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

## Domain Features

In this paper we will formally define four domain features that can be used to assist knowledge engineers during the problem formulation process, i.e. the authoring of a planning domain which defines the state space. These features may also be exploited by a planning algorithm to speed up the search, but this possibility depends on the actual planning algorithm used and will not be evaluated in this paper. The features defined here are: *domain types*, *relation fluency*, *inconsistent effects* and *reversible actions*. These features are not new, at least at an informal level. Their specification is either already part of PDDL or could easily be added to the language.

The values these features take for a given domain can also be computed independent of their explicit specification. A comparison of the computed features to the ones specified in the formal domain definition can then be used to validate the formalization, thus supporting the domain author in producing a consistent domain. Applying this approach to various planning domains shows that the features defined here can be used to identify certain representational problems.

## Related Work

Amongst the features mentioned above, domain types have been discussed most in the planning literature. A rigorous method for problem formulation in the case of planning domains was presented in (McCluskey and Porteous 1997). In the second step of their methodology types are extracted from an informal description of a planning domain. Types have been used as a basic domain feature in TIM (Fox and Long 1998). Their approach exploits functional equivalence of objects to derive a hierarchical type structure. The difference between this approach and our algorithm will be explained in the relevant section below. This work has later been extended to infer generic types such as mobiles and resources that can be exploited to optimize plan search (Coles and Smith 2006).

The distinction between rigid and fluent relations (Ghalab *et al.* 2004) is common in AI planning and will be discussed only briefly. Inconsistent effects of different actions are exploited in the GraphPlan algorithm (Blum and Furst 1995) to define the mutex relation. However, this is applied to pairs of actions (i.e. fully ground instances of operators)



rather than operators. Reversible actions, as a domain feature, are not related to regression of goals, meaning this feature is unrelated to the direction of search (forward from the initial state or regressing backwards from the goal). The reversibility of actions (or operators) does not appear to feature much in the AI planning literature. However, in generic search problems they are a common technique used to prune search trees (Russell and Norvig 2003).

Preprocessing of planning domains is a technique that has been used to speed up the planning process (Dawson and Siklossy 1977). Perhaps the most common preprocessing step is the translation of the STRIPS (function-free, first-order) representation into a propositional representation. An informal algorithm for this is described in (Ghallab *et al.* 2004, section 2.6). A conceptual flaw in this algorithm (highlighted by the analysis of inconsistent effects) will be briefly discussed in the conclusions of this paper.

### Type Information

Many planning domains include explicit type information. In PDDL the `:typing` requirement allows the specification of typed variables in predicate and operator declarations. In problem specifications, it allows the assignment of constants or objects to types. If nothing else, typing tends to greatly increase the readability of a planning domain. However, it is not necessary for most planning algorithms to work.

In this section we will show how type information can be inferred from the operator descriptions in the planning domain definition. If the planning domain includes explicit type information the inferred types can be used to perform a consistency check, thus functioning as a knowledge engineering tool. In any case, type information can be used to simplify parts of the planning process. For example, if the planner needs to propositionalize the planning domain, type information can be used to limit the number of possible values for variables, or a ground backward searcher may use this information to similar effect.

The formalism that follows is necessary to show that the derived type system is **maximally specific** given the knowledge provided by the operators, that is, any type system that further subdivides a derived type must necessarily lead to a search space that contains type inconsistent states.

### Type Consistency

The simplest kind of type system often used in planning is one in which the set of all constants  $C$  used in the planning domain and problem is divided into disjoint types  $T$ . That is, each type corresponds to a subset of all constants and each constant belongs to exactly one type. This is the kind of type system we will look at here.

**Definition 1 (type partition)** A *type partition*  $\mathcal{P}$  is a tuple  $\langle C, T, \tau \rangle$  where:

- $C$  is a finite set of  $n(C) \geq 1$  constant symbols  $C = \{c_1, \dots, c_{n(C)}\}$ ,
- $T$  is a set of  $n(T) \leq n(C)$  types  $T = \{t_1, \dots, t_{n(T)}\}$ , and
- $\tau : C \rightarrow T$  is a function defining the type of a given constant.

A type partition divides the set of all constants that may occur in a planning problem into a set of equivalence classes. The availability of a type partition can be used to limit the space of world states that may be searched by a planner. In general, a world state in a planning domain can be any subset of the powerset of the set of ground atoms over predicates  $P$  with arguments from  $C$ .

**Definition 2 (type function)** Let  $P = \{P_1, \dots, P_{n(P)}\}$  be a set of  $n(P)$  predicate symbols with associated arities  $a(P_i)$  and let  $T = \{t_1, \dots, t_{n(T)}\}$  be a set of types. A **type function** for predicates is a function  $arg_P : P \times \mathbb{N} \rightarrow T$

which, for a given predicate symbol  $P_i$  and argument number  $1 \leq k \leq a(P_i)$  gives the type  $arg_P(P_i, k) \in T$  of that argument position.

This is the kind of type specification we find in PDDL domain definitions as part of the definition of predicates used in the domain, provided that the typing extension of PDDL is used. The type function is defined by enumerating the types for all the arguments of each predicate.

**Definition 3 (type consistency)** Let  $\langle C, T, \tau \rangle$  be a type partition. Let  $P_i \in P$  be a predicate symbol and let  $c_1, \dots, c_{a(P_i)} \in C$  be constant symbols. The ground first-order atom  $P_i(c_1, \dots, c_{a(P_i)})$  is **type consistent** iff  $\tau(c_k) = arg_P(P_i, k)$ . A world state is **type consistent** iff all its members are type consistent.

Thus, for a given predicate  $P_i$  there are  $|C|^{a(P_i)}$  possible ground instances that may occur in world states. Clearly, the set of type consistent world states is a subset of the set of all world states. The availability of a set of types can also be used to limit the actions considered by a planner.

**Definition 4 (type function)** Let  $O = \{O_1, \dots, O_{n(O)}\}$  be a set of  $n(O)$  operator names with associated arities  $a(O_i)$  and let  $T = \{t_1, \dots, t_{n(T)}\}$  be a set of types. A **type function** for operators is a function  $arg_O : O \times \mathbb{N} \rightarrow T$

which, for a given operator symbol  $O_i$  and argument number  $1 \leq k \leq a(O_i)$  gives the type  $arg_O(O_i, k) \in T$  of that argument position.

Again, this is exactly the kind of type specification that may be provided in PDDL where the function is defined by enumeration of all the arguments with their types for each operator definition.

**Definition 5 (type consistency)** Let  $\langle C, T, \tau \rangle$  be a type partition. Let  $O_i(v_1, \dots, v_{a(O_i)})$  be a STRIPS operator defined over variables  $v_1, \dots, v_{a(O_i)}$  with preconditions  $prec(O_i)$  and effects  $effects(O_i)$ , where each precondition/effect has the form  $P_j(v_{P_j,1}, \dots, v_{P_j,a(P_j)})$  or  $\neg P_j(v_{P_j,1}, \dots, v_{P_j,a(P_j)})$  for some predicate  $P_j \in P$ . The operator  $O_i$  is **type consistent** iff:

- all the operator variables  $v_1, \dots, v_{a(O_i)}$  are mentioned in the positive preconditions of the operator, and
- if  $v_k = v_{P_j,l}$ , i.e. the  $k$ th argument variable of the operator is the same as the  $l$ th argument variable of a precondition or effect, then the types must also be the same:  $arg_O(O_i, k) = arg_P(P_j, l)$ .

The first condition is often required only implicitly (see (Ghallab *et al.* 2004, chapter 4)) to avoid the complication of “lifted” search in forward search. We will use this condition shortly to show that a type consistent system is closed.

Given a type partition  $\langle C, T, \tau \rangle$  and type functions  $arg_P$  and  $arg_O$ , we can define a most general state-transition system over all type consistent states as follows:

**Definition 6 (state-transition system  $\Sigma^*$ )** Let  $\langle C, T, \tau \rangle$  be a type partition. Let  $P = \{P_1, \dots, P_{n(P)}\}$  be a set of predicate symbols with associated type function  $arg_P$  and let  $O = \{O_1, \dots, O_{n(O)}\}$  be a set of type consistent operators. Then  $\Sigma^* = (S^*, A^*, \gamma)$  is a (restricted) state-transition system, where:

- $S^*$  is the powerset of the set of all type consistent ground atoms with predicates from  $P$  and arguments from  $C$ ,
- $A^*$  is the set of all (type consistent) ground instances of operators from  $O$ , and
- $\gamma$  is the usual state transition function for STRIPS actions:  $\gamma(s, a) = (s - effects^-(a)) \cup effects^+(a)$  iff action  $a$  is applicable in state  $s$ <sup>1</sup>.

This state-transition system forms a super-system to a state-transition system defined by a planning problem containing a type consistent initial state, and a set of type consistent operator definitions, in the sense that the states of that system (the reachable states from the initial states) must be a subset of  $S^*$  and the actions must be a subset  $A^*$ . It is therefore interesting to observe that  $\Sigma^*$  is closed:

**Proposition 1 (closed  $\Sigma^*$ )** Let  $s \in S^*$  be a type consistent state, i.e. a type consistent set of ground atoms. Let  $a \in A^*$  be a type consistent action that is applicable in  $s$ . Then the successor state  $\gamma(s, a)$  is a type consistent state in  $S^*$ .

To show that the above is true, we need to show that every atom in  $\gamma(s, a)$  is type consistent. Each atom in  $\gamma(s, a)$  was either in the previous state,  $s$ , in which case it was type consistent by definition, or it was added as a positive effect. Since the action is an applicable instance of a type consistent operator  $O_i$  there must be a substitution  $\sigma$  such that  $\sigma(precs^+(O_i)) \subseteq s$ . Furthermore, this substitution grounds every operator variable because type consistency requires all of them to occur in the positive preconditions. Given the type consistency of  $s$ , all arguments in  $\sigma(precs^+(O_i))$  must agree with  $arg_P$ . Given the type consistency of  $O_i$ , all arguments of  $a$  must agree with  $arg_O$ , and therefore so must the effects  $\sigma(effects(O_i))$ . Hence, all positive effects are type consistent, meaning every element of  $\gamma(s, a)$  must be type consistent. ■

## Derived Types

The above definitions assume that there is an underlying type system that has been used to define the planning domain and problems in a consistent fashion. We shall continue to assume that such a type system exists, but it may not have been explicitly specified in the PDDL definition of

the domain. We shall now define a type system that is derived from the operator descriptions in the planning domain.

**Definition 7 (type name)** Let  $O = \{O_1, \dots, O_{n(O)}\}$  be a set of STRIPS operators. Let  $P$  be the set of all the predicate symbols used in all the operators. A **type name** is a pair  $\langle N, k \rangle \in (P \cup O) \times \mathbb{N}$ .

A type name can be used to refer to a type in a derived type system. There usually are multiple names to refer to the same type. The basic idea behind the derived types is to partition the set of all type names into equivalence classes, and then assign constants used in a planning problem to different equivalence classes, thus treating each equivalence class as a type.

**Definition 8 (O-type)** Let  $O = \{O_1, \dots, O_{n(O)}\}$  be a set of STRIPS operators over operator variables  $v_1, \dots, v_{a(O_i)}$  with  $conds(O_i) = precs(O_i) \cup effects(O_i)$  and all operator variables mentioned in the positive preconditions. Let  $P$  be the set of all the predicate symbols used in all the operators. An **O-type** is a set of type names. Two type names  $\langle N_1, i_1 \rangle$  and  $\langle N_2, i_2 \rangle$  are in the same O-type, denoted  $\langle N_1, i_1 \rangle \equiv_O \langle N_2, i_2 \rangle$ , iff one of the following holds:

- $N_1(v_{1,1}, \dots, v_{1,a(N_1)})$  is an operator with precondition or effect  $N_2(v_{2,1}, \dots, v_{2,a(N_2)}) \in conds(N_1)$  which share a specific variable:  $v_{1,i_1} = v_{2,i_2}$ ,
- $N_2(v_{2,1}, \dots, v_{2,a(N_2)})$  is an operator with precondition or effect  $N_1(v_{1,1}, \dots, v_{1,a(N_1)}) \in conds(N_2)$  which share a specific variable:  $v_{1,i_1} = v_{2,i_2}$ , or
- there is a type name  $\langle N, j \rangle$  such that  $\langle N, j \rangle \equiv_O \langle N_1, i_1 \rangle$  and  $\langle N, j \rangle \equiv_O \langle N_2, i_2 \rangle$ .

**Definition 9 (O-type partition)** Let  $(s_i, g, O)$  be a STRIPS planning problem. Let  $C$  be the set of all constants used in  $s_i$ . Let  $T = \{t_1, \dots, t_{n(T)}\}$  be the set of O-types derived from the operators in  $O$ . Then we can define the function  $\tau : C \rightarrow T$  as follows:

$$\tau(c) = t_i : \forall R(c_1, \dots, c_{a(R)}) \in s_i : (c_j = c) \Rightarrow \langle R, j \rangle \in t_i$$

Note that  $\tau(c)$  is not necessarily well-defined for every constant mentioned in the initial state, e.g. if a constant is used in two relations that would indicate different derived types (which rely only on the operator descriptions). In this case the O-type partition cannot be used as defined above. However, if appropriate unions of O-types are taken then this results in a new type partition for which  $\tau(c)$  is defined. In the worst case this will lead to a type partition consisting of a single type. Given that this approach is always possible, we shall now assume that  $\tau(c)$  is always defined.

**Definition 10** Let  $T = \{t_1, \dots, t_{n(T)}\}$  be the set of O-types for a given set of operators  $O$  and let  $P = \{P_1, \dots, P_{n(P)}\}$  be the predicates that occur on operators from  $O$ . We can easily define type functions  $arg_P$  and  $arg_O$  as follows:

$$arg_P(P_i, k) = t_i : \langle P_i, k \rangle \in t_i \text{ and} \\ arg_O(O_i, k) = t_i : \langle O_i, k \rangle \in t_i$$

**Proposition 2** Let  $(s_i, g, O)$  be a STRIPS planning problem and let  $\langle C, T, \tau \rangle$  be the O-type partition derived from this problem. Then every state that is reachable from the initial state  $s_i$  is type consistent.

<sup>1</sup>See the definition of a STRIPS operator in (Ghallab *et al.* 2004, page 28) and the discussion of inconsistent effects below.

To show this we first show that the initial state is type consistent. Since the definition of  $\tau$  is based on the argument positions in which they occur in the initial state, this follows trivially.

Next we need to show that every action that is an instance of an operator in  $O$  is type consistent. All operator variables must be mentioned in the positive preconditions according to the definition of an  $O$ -type. Furthermore, if a precondition or effect share a variable with the operator, these must have the same type since  $\equiv_O$  puts them into the same equivalence class.

Finally we can show that, if action  $a$  is applicable in a type consistent state  $s$ , the resulting state  $\gamma(s, a)$  must also be type consistent. Every atom must come either from  $s$  in which case it must be type consistent, or it comes from a positive effect, which, given the type consistency of  $a$  means it must also be type consistent. ■

This shows that the type system derived from the operator definitions is indeed useful as it creates a state space of type consistent states. However, the question that remains is whether it is the best or even only type system. Clearly, there may be other type systems that give us type consistent state space. The system that consists just of a single type is a trivial example. A better type system would divide the set of constants into more types though, as this reduces the size of a type consistent state space. We will now show that the above type system is maximally specific given the knowledge provided by the operators.

**Theorem 1** *Let  $(s_i, g, O)$  be a STRIPS planning problem and let  $\langle C, T, \tau \rangle$  be the  $O$ -type partition derived from this problem. If two constants  $c_1$  and  $c_2$  have the same type  $\tau(c_1) = \tau(c_2)$  then they must have the same type in every type partition that creates a type consistent search space.*

The first step towards showing that the above holds is the insight that operators can be used to constrain types in both directions, forward and backward. If an operator variable  $v_i$  appears in a precondition and an effect, then the type of the position of the predicate in the effect must be subset of the type of the position in the precondition or the application of the operator may lead to a state that is not type consistent. Since types are defined by an equivalence relation, however, the two types must actually be the same type. Hence the type in the effect also constrains the type in the precondition.

Now, for two type names to be in the same  $O$ -type, there must be a connecting chain  $\langle R_0 O_1 R_1 \dots O_n R_n \rangle$  of alternating first order literals and operators such that  $R_{i-1}$  and  $R_i$  are conditions of  $O_i$  which share an operator variable as the  $j_{i-1}$ th and  $j_i$ th argument respectively. The variable that is shared may vary along the chain. For each step along the chain, if a constant may occur in the  $j_{i-1}$ th position in  $R_{i-1}$  it may also occur in the  $j_i$ th position in  $R_i$ . Thus, there may be two type consistent states that are connected by  $O_i$  and which contain instances of  $R_{i-1}$  and  $R_i$ . Since both states are type consistent, both instances must be type consistent, too.

Now let us assume that  $c_1$  appears as  $j_0$ th argument in  $R_0$  and let  $c_2$  appears as  $j_n$ th argument in  $R_n$ . Furthermore, let us assume there exists a type partition that assigns  $c_1$  and  $c_2$

to different types. Since  $c_1$  is the  $j_0$ th argument in  $R_0$  there may be another state in which  $c_1$  appears as  $j_n$ th argument in  $R_n$ . Thus it appears in the same position of the same predicate as  $c_2$ , which means it must have the same type to be type consistent. ■

## An Efficient Algorithm

The algorithm to derive domain types  $t_d$  treats types as sets of predicate and argument-number pairs. That is  $t_d \subseteq 2^{P \times \mathbb{N}}$ . Each domain type  $t_d$  corresponds to exactly one type  $t \in T$ . The only argument taken by the algorithm is the set of operator definitions  $O$ .

```

function extract-types( $O$ )
   $pTypes \leftarrow \emptyset$ 
   $vTypes \leftarrow \emptyset$ 
  for every  $op \in O$  do
    extract-types( $op, pTypes, vTypes$ )
  return  $pTypes$ 

```

The variable  $pTypes$  contains the  $O$ -types that have been discovered so far. Initially there are no  $O$ -types and the set is empty.  $vTypes$  is a set of pairs of variables (used in operator definitions) and  $O$ -types, best implemented as a map and also initially empty. The procedure then analyzes each operator in the given set, thereby building up the type system incrementally.

```

function extract-types( $op, pTypes, vTypes$ )
  for every  $p \in pre(op) \cup eff(op)$  do
    for  $i = 1$  to  $a(p)$  do
       $t_{pi} \leftarrow t_d \in pTypes : \langle rel(p), i \rangle \in t_d$ 
       $\langle v, t_v \rangle \leftarrow v_t \in vTypes : \exists t_d : v_t = \langle arg(i, p), t_d \rangle$ 
      if  $undef(\langle v, t_v \rangle)$  do
        if  $undef(t_{pi})$  do
           $t_{pi} \leftarrow \{ \langle rel(p), i \rangle \}$ 
           $pTypes \leftarrow pTypes \cup t_{pi}$ 
           $vTypes \leftarrow vTypes \cup \langle arg(i, p), t_{pi} \rangle$ 
        else
          if  $undef(t_{pi})$  do
             $t_v \leftarrow t_v \cup \{ \langle rel(p), i \rangle \}$ 
          else
            merge-types( $t_v, t_{pi}, pTypes, vTypes$ )

```

The analysis of a given operator goes through every precondition and effect of the operator, looking at every argument position in turn. The next steps of the algorithm depend on whether the predicate-position combination has been used before (in which case it will appear in the  $pTypes$ ) and whether the variable at that position has been used before (in which case it will be a key in the  $vTypes$ ). If only one or neither have been used, the algorithm simply adds the relevant elements to the  $pTypes$  and the  $vTypes$ . If both have been used it may be necessary to merge the respective  $O$ -types.

```

function merge-types( $t_1, t_2, pTypes, vTypes$ )
  if  $t_1 = t_2$  do
    return
   $pTypes \leftarrow pTypes - \{t_1, t_2\}$ 
   $t_{new} \leftarrow t_1 \cup t_2$ 
   $pTypes \leftarrow pTypes \cup \{t_{new}\}$ 
  for every  $\langle v, t_v \rangle \in vTypes$  do
    if  $(t_v = t_1) \vee (t_v = t_2)$  do
       $vTypes \leftarrow vTypes - \langle v, t_v \rangle$ 
       $vTypes \leftarrow vTypes + \langle v, t_{new} \rangle$ 

```

Of course, no action is required if the type of the variable and the type for the predicate-position combination is the same. Otherwise we replace the two sets representing the (previously different) types in  $pTypes$  with a new type that is the union of the two sets. Also we need to update the pairs in  $vTypes$  to ensure that keys that previously had one of the now removed types as value will now get the new type as their new value.

It is easy to see that the algorithm runs in polynomial time. Furthermore, the analysis performed by the algorithm uses only the operator descriptions, and thus its run time does not depend on the problem size.

This algorithm shares the input with TIM (Fox and Long 1998), namely the operator specifications. Both algorithms use the argument positions in which parameters occur in preconditions and effects as the basis for their analysis. TIM uses this information to construct a set of finite state machines to model transitions of objects, whereas our algorithm builds the equivalence classes directly. The result produced by TIM is a hierarchical type system that is used to derive state invariants. In contrast, the type system derived by our algorithm is flat, meaning it may be less discriminating than the structure derived by TIM. However, we could show that the types derived by our algorithm are maximally specific for given operator descriptions. In addition, a flat type system can be used to enrich the operator definitions explicitly by simply adding unary predicates as type preconditions.

## Evaluation

To evaluate the algorithm we have applied it to a small number of planning domains. To avoid any bias we used only planning domains that were available from third parties, mostly from the international planning competition. Since the algorithm works on domains and the results have to be interpreted manually only a limited number of experiments was possible. Random domains are not suitable as they cannot be expected to encode an implicit type system. The algorithm has been used on random domains, but this did not result in any useful insights.

A planning domain on which the algorithm has been used is the DWR domain (Ghallab *et al.* 2004). In this domain types are defined explicitly, so it was possible to verify consistency with the given types. The algorithm produced the following, listing the argument positions in predicates where they are used (the  $pTypes$ ):

```

type: [loaded-0, unloaded-0, at-0]
type: [attached-0, top-1, in-1]
type: [occupied-0, attached-1, belong-1,
       adjacent-1, adjacent-0, at-1]
type: [belong-0, holding-0, empty-0]
type: [loaded-1, holding-1, on-1, on-0,
       in-0, top-0]

```

The first type states that it is used as the first argument in the loaded, unloaded and at predicate. This corresponds exactly to the robot type in the PDDL specification of the domain. Similarly, the other types correspond to pile, location, crane and container, in this order. The main difference is that the derived types do not have intelligible names.

The other domains that were used for testing did not come with type information specified in the same way as the DWR domain. However, they all use unary predicates to add type information to the preconditions (but not every unary predicate is a type). The domains used are the following STRIPS domains from the international planning competition: movie, gripper, logistics, mystery, mprime and grid. The algorithm derives between 3 and 5 types for each of these domains which appears consistent with what the domain authors had in mind. The only domain that stands out is the first, in which each predicate has its own type. However this appears to be appropriate for this very simple domain.

## Static and Fluent Relations

Another domain feature that is useful for the analysis of planning domains concerns the relations that are used in the definition of the operators. The set of predicates used here can be divided into static (or rigid) relations and fluent (or dynamic) relations, depending on whether atoms using this predicate can change their truth value from state to state.

**Definition 11 (static/fluent relation)** Let  $O = \{O_1, \dots, O_{n(O)}\}$  be a set of operators and let  $P = \{P_1, \dots, P_{n(P)}\}$  be a set of all the predicate symbols that occur in these operators. A predicate  $P_i \in P$  is **fluent** iff there is an operator  $O_j \in O$  that has an effect that uses the predicate  $P_i$ . Otherwise the predicate is **static**.

The algorithm for computing the sets of fluent and static predicate symbols is trivial and hence, we will not list it here.

There are at least two ways in which this information can be used in the validation of planning problems. Firstly, if the domain definition language allowed the domain author to specify whether a relation is static or fluent then this could be verified when the domain is parsed. This might highlight problems with the domain. Secondly, in a planning problem that uses additional relations these could be highlighted or simply removed from the initial state.

The computation of static and fluent relations has been tested on the same domains as the derived types. As is to be expected, nothing interesting can be learned from this experiment.

## Inconsistent Effects

In a STRIPS-style operator definition the effects are specified as and add- and delete-lists consisting of a set of (function-free) first-order atoms, or a set of first-order literals where positive elements correspond to the add-list and negative elements correspond to the delete-list. Normally, the definition of an operator permits potentially inconsistent effects, i.e. a positive and a negative effect may be complementary.

### Operators

**Definition 12 (potential inconsistency)** Let  $O$  be a planning operator with positive effects  $e_1^p, \dots, e_{n(e^p)}^p$  and negative effects  $e_1^n, \dots, e_{n(e^n)}^n$ , where each positive/negative effect is a first-order atom.  $O$  has **potentially inconsistent effects** iff  $O$  has a positive effect  $e_i^p$  and a negative effect  $e_j^n$  for which there exists a substitution  $\sigma$  such that  $\sigma(e_i^p) = \sigma(e_j^n)$ .

It is fairly common for planning domains to define operators with potentially inconsistent effects. For example, the move operator in the DWR domain is defined as follows:

```
(:action move
:parameters (?r ?fr ?to)
:precondition (and (adjacent ?fr ?to)
  (at ?r ?fr) (not (occupied ?to)))
:effect (and (at ?r ?to) (occupied ?to)
  (not (occupied ?fr)) (not (at ?r ?fr))))
```

This operator has a positive effect  $(\text{at } ?r \text{ ?to})$  and a negative effect  $(\text{at } ?r \text{ ?fr})$ . These two effects are unifiable and represent a potential inconsistency. Since this is a common feature in planning domains there is no need to raise this to the domain author. Effects that are necessarily inconsistent may be more critical.

**Definition 13 (necessary inconsistency)** Let  $O$  be a planning operator with positive effects  $E^p = \{e_1^p, \dots, e_{n(e^p)}^p\}$  and negative effects  $E^n = \{e_1^n, \dots, e_{n(e^n)}^n\}$ , where each positive/negative effect is a first-order atom.  $O$  has **necessarily inconsistent effects** iff  $O$  has a positive effect  $e_i^p$  and a negative effect  $e_j^n$  such that  $e_i^p = e_j^n$ .

None of the domains used in the experiments above specified an operator with necessarily inconsistent effects. Given the definition of the state-transition function for STRIPS operators (Ghallab *et al.* 2004) as

$$\gamma(s, a) = (s - E^n) \cup E^p$$

it should be clear that the negative effect  $e_j^n$  can be omitted from the operator description without changing the set of reachable states. If  $e_j^n \notin s$  then its removal from  $s$  will not change  $s$ , and the addition of  $e_i^p$  ensures that  $e_j^n \in \gamma(s, a)$  because  $e_i^p = e_j^n$ . If  $e_j^n \in s$  it will be removed in  $\gamma(s, a)$ , but it will subsequently be re-added. Thus, the presence of the negative effect does not change the range of the state-transition function.

From a knowledge engineering perspective this means that an operator with necessarily inconsistent effects indicates a problem and should be raised to the domain author. However, this is only true for simple STRIPS operators where actions are instantaneous and thus, all effects happen simultaneously. If effects are permitted at different time points

then only those that are necessarily inconsistent at the same time point must be considered a problem.

### Actions

Since actions are ground instances of operators, there is no need to distinguish between necessarily and potentially inconsistent effects. All effects must be ground for actions and therefore inconsistent effects are always necessarily inconsistent. Even if necessarily inconsistent operators are not permitted in a domain, actions with inconsistent effects may still occur as instances of operators with potentially inconsistent effects.

Whether it is desirable for the planner to consider such actions depends on the other effects of the action. For example, in the DWR domain no action with inconsistent effects needs to be considered. However, if an action has side effects then it may make sense to permit such actions. For example, circling an aircraft in a holding pattern does not change the location of the aircraft, but it does reduce the fuel level. If such side effects are important actions with inconsistent effects may need to be permitted. And, of course, every action has the side effect of taking up a step in a plan.

If actions with inconsistent effects are considered by the planner, this may lead to further complications. This is because the definition of the state-transition function first subtracts negative effects from a state and then adds positive effects. For actions that have no inconsistent effects this order is irrelevant. However, if actions with inconsistent effects are permitted the result may be surprising. For example, returning to the move operator in the DWR domain, this has been defined with a positive effect  $(\text{occupied ?to})$  and a negative effect  $(\text{occupied ?fr})$ . Thus, the action  $(\text{move } r \text{ loc loc})$  will result in a state in which  $(\text{occupied loc})$  holds. Now suppose the domain had been defined using the predicate  $\text{free}$  instead of  $\text{occupied}$ . In this case the result of  $(\text{move } r \text{ loc loc})$  would result in a state in which  $(\text{free loc})$  holds. This problem occurs only with inconsistent effects.

None of the domains used in the tests above require actions with inconsistent effects and thus, they can be ignored by the planner. The following algorithm can be used to find the applicable actions (without inconsistent effects) in a given state.

```
function addApplicables( $A, o, p, \sigma, s$ )
  if not empty( $p^+$ ) then
    let  $p_{next} \in p$ 
    for every  $s_p \in s$  do
       $\sigma' \leftarrow \text{unify}(\sigma(p_{next}), s_p)$ 
      if valid( $\sigma'$ ) then
        addApplicables( $A, o, p - p_{next}, \sigma', s$ )
  else
    for every  $p_{next} \in p^-$  do
      if falsifies( $s, \sigma(p_{next})$ ) then return
    for every  $e_p \in \text{effects}^+(o)$  do
      for every  $e_n \in \text{effects}^-(o)$  do
        if  $e_p = e_n$  then return
   $A \leftarrow A + \sigma(o)$ 
```

The algorithm adds all instances of operator  $o$  that are applicable in state  $s$  to the set of actions  $A$ . The parameter  $p$  represents the remaining preconditions (initially empty) and a substitution  $\sigma$  (also initially empty) will be built up by the algorithm. It first deals with the remaining positive preconditions and uses those to construct the substitution for all the parameters of the operators. Note that we require an operator to mention all its parameters in the positive preconditions. When the positive preconditions have been tested, the algorithm checks the negative preconditions under  $\sigma$  which must now be fully ground. Finally, the algorithm tests for inconsistent effects by doing a pairwise comparison between positive and negative effects. This algorithm can also be used to generate the actions for the next action layer in a planning graph. A goal regression version is slightly different as it is no longer guaranteed that all the operator parameters will be bound after the unification with a goal (and possibly static preconditions).

### Reversible Actions

A common feature in many planning domains (and in many classic search problems) is that they contain actions that can be reversed by applying another action. There is usually no need to consider such actions during the search process.

### Reversible Operators

The idea here is to apply the concept of reversibility to operators: an operator may be reversed by another operator (or the same operator), possibly after a suitable substitution of variables occurring as parameters in the operator definition. Note that this definition is somewhat narrow as it demands this pattern to be consistent across all instances of the two operators, i.e. it excludes the possibility of an operator sometimes being reversed by one operator, and sometimes by another, depending on the values of the parameters.

**Definition 14 (reversing operators)** *An action  $a$  that is applicable in a state  $s$  is **reversed by an action**  $a'$  if the state that results from applying the sequence  $\langle aa' \rangle$  in  $s$  results in  $s$ , i.e. the state remains unchanged. An operator  $O$  is **reversed by an operator**  $O'$  under substitution  $\sigma'$  iff for every action  $a = \sigma(O)$  that is an instance of  $O$ :*

- if  $a$  is applicable in a state  $s$  then  $a' = \sigma'(\sigma'(O'))$  is applicable in  $\gamma(s, a)$  and
- $\gamma(\gamma(s, a), a') = s$ .

For example, consider the `(move ?r ?l1 ?l2)` operator from the DWR domain. This can be reversed by another move operation with different parameters, as defined by the substitution  $\sigma' = \{?l1 \leftarrow ?l2, ?l2 \leftarrow ?l1\}$ , i.e. `(move ?r ?l1 ?l2)` is reversed by  $\sigma'((move ?r ?l1 ?l2)) = (move ?r ?l2 ?l1)$ .

While this definition captures the idea of a reversing operator, it is not very useful from a computational point of view. Another way to avoid exploring states that are the result of the application of an action followed by its reverse action is to store all states in a hash table and test whether the new state has been encountered before, an approach that is far

more general than just testing for reversing actions. Computationally, it is roughly as expensive as the test suggested by the above definition. The key here is that both are state specific. A definition of reversibility that does not depend on the state in which an action is applied would be better.

From a domain author's perspective, it is often possible to specify which operators can be used to reverse another operator, as we have shown in the DWR move example above. If this information is available during search then there is no need to apply the reverse action, generate the state, and compare it to the previous state. Instead a relatively simple substitution test would suffice:  $a' = \sigma'(\sigma'(O'))$ .

**Proposition 3** *Let  $O_1$  be an operator with positive effects  $E_1^p$  and negative effects  $E_1^n$  that is reversed by  $O_2$  with positive effects  $E_2^p$  and negative effects  $E_2^n$  under substitution  $\sigma'$ . Then the two sets of positive/negative effects must cancel each other:*

$$E_1^p = \sigma'(E_2^n) \text{ and } E_1^n = \sigma'(E_2^p)$$

Suppose there is a positive effect in  $E_1^p$  that is not in  $\sigma'(E_2^n)$ . Now suppose an instance of  $O$  was applied in a state in which the effect in question does not already hold. The effect would then be added by the instance of  $O$  but it would not be deleted by the reversing action, and thus the original state and the state resulting from the two actions in sequence would not be the same. A similar argument holds for an effect in  $E_1^n$  that is not in  $\sigma'(E_2^p)$ . ■

This means we can let the domain author specify reversing operators and then use the above necessary criterion for validation. Or we could treat the above criterion as sufficient and thus exclude a portion of the search space. This may lead to an incompleteness in the search, but the domains we have used for our evaluation do not show this problem.

### Unique Reversibility

In fact we have made an even stronger assumption to carry out some experiments with the domains mentioned above: we have assumed that there is at most one operator that reverses a given operator. We have then, for each domain, done a pairwise test on all the operators defined in the domain to see whether the necessary criterion holds. This resulted in discovering that the move operator can be reversed by itself with a substitution automatically derived from the operator definition, and similarly it discovered the reversibility between the take and put operators and the load and unload operators in the DWR domain.

Perhaps surprisingly, the unique reversibility was not given for all domains. The `logistics` domain contains load and unload operators for trucks and airplanes. These are specified as four distinct operators. However, in terms of their effects the two load operators and the two unload operators cannot be distinguished. The only difference lies in the preconditions where the `?truck` parameter is required to be a truck and the `?airplane` parameter is required to be an airplane.

This result can be interpreted in two ways: one could argue that the necessary condition may not be used as sufficient in this domain. Or one could argue that this domain contains redundancy that can be removed by merging the

two load and unload operators, which would not change the set of reachable states in this example but means the planner has fewer actions to consider. Either way, testing for the necessary reversibility condition has highlighted this domain feature.

## Conclusions

This paper has defined four planning domain features that can be used by knowledge engineers to provide information about the domain they are encoding. The formal definition of the features was used to design algorithms that can extract the actual feature values from the domain description. The algorithms are based on the domain description only, i.e. they do not require a planning problem as input. The extracted features can then be compared to the feature values specified by the domain author to validate the domain description. This approach has been evaluated using domains taken mostly from the international planning competition. The result shows that features were consistent with those available in the domains, where explicitly specified. Those features that were not specified were extracted and manually verified, to ensure they are consistent with the given set of operators.

The first feature, the type system, is a rather simple, flat division into equivalence classes. This may not be suitable for very complex planning domains, but the domains we have analyzed do not exhibit much hierarchical structure. The advantage of such a type system is that it can be easily added to the operator descriptions in the form of unary preconditions. Furthermore, we showed that the type system derived by our algorithm is the most specific type system of its kind based solely on the operator descriptions. An open question is whether this is identical to the least general generalization (Plotkin 1969) used in machine learning. The algorithm could be refined to derive a hierarchical type system if one takes into account the directionality of the operators, but for a type system consisting of equivalence classes this is irrelevant. Also, the algorithm described in this paper should also be applicable to hierarchical task network domains, but this has not yet been implemented.

Actions with inconsistent effects are another feature we have defined. For most domains, such actions are probably not desirable. In fact, the admission of such actions leads to a different planning problem as the state spaces with or without such actions may be different for the same planning domain and problem. Also, planners that translate a STRIPS planning problem (with negative preconditions) into a propositional problem (without negative preconditions) need to be more careful if actions with inconsistent effects are permitted. The translation method described in (Ghallab *et al.* 2004, section 2.6) does not work in this case as it introduces independent predicates for a predicate and its negations, which can become true in the same state if an action with inconsistent effects is applied. This would render the planner potentially unsound.

The final feature which defines reversible actions is somewhat different as it can only be usefully used as a necessary criterion to test whether one operator is the reverse of another. The more strict, sufficient definition does not pro-

vide any computational advantage. The difference is simply that the necessary criterion can be computed on the basis of the operator descriptions, whereas the sufficient test requires knowledge of the state in which an action is applied. The difference is quite subtle though, and may not matter in practice. The necessary criterion requires the positive and negative effects to cancel each other. However, if a state contains an atom that is also added by the first action, but then deleted by the second action, then the state will be changed. If an operator listed all the relevant atoms also as preconditions, this exception would not hold.

Implementations of the algorithms described in this paper (in Java) exist. They are currently being ported to PHP where they can be used as part of an extension to MediaWiki that allows the semi-formal specification of planning knowledge to support distributed development and sharing of procedural knowledge.

## References

- Saul Amarel. On representations of problems of reasoning about actions. In Donald Michie, editor, *Machine Intelligence 3*, pages 131–171. Elsevier/North-Holland, 1968.
- Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. In *Proc. 14th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1636–1642. Morgan Kaufmann, 1995.
- Andrew Coles and Amanda Smith. Generic types and their use in improving the quality of search heuristics. In *Proc. 25th Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2006)*, 2006.
- Clive Dawson and Laurent Siklossy. The role of preprocessing in problem-solving systems. In *Proc. 5th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 465–471. Morgan Kaufmann, 1977.
- Maria Fox and Derek Long. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367–421, 1998.
- Maria Fox and Derek Long. PDDL2.1 : An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.
- Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning*. Morgan Kaufmann, 2004.
- T.L. McCluskey and J.M. Porteous. Engineering and compiling planning domain models to promote validity and efficiency. *Artificial Intelligence*, 95:1–65, 1997.
- Gordon Plotkin. A note on inductive generalization. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 5*, pages 153–164. Edinburgh University Press, 1969.
- Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.

# Fluent Merging for Classical Planning Problems

Jendrik Seipp and Malte Helmert

Albert-Ludwigs-Universität Freiburg

Institut für Informatik

Georges-Köhler-Allee 52

79110 Freiburg, Germany

{seipp, helmert}@informatik.uni-freiburg.de

## Abstract

Fluent merging is a reformulation technique for classical planning problems that can be applied automatically or semi-automatically. The reformulation strives to transform a planning task into a representation that allows a planning algorithm to find solutions more efficiently or to find solutions of better quality. This work introduces different approaches for fluent merging and evaluates them within a state-of-the-art planning system.

## Introduction

In classical planning we try to find plans in a fully observable world. While searching for a plan we move from one state to another. Each state is a function that assigns values to a number of variables. The set of values a variable can have is called its domain.

In the original definition of planning problems in PDDL notation (McDermott et al. 1998) there are only *Boolean variables*. Recent research has shown that combining several such variables into more general *finite-domain variables*, a process that has been called *fluent merging* (van den Briel, Kambhampati, and Vossen 2007), can make plan search more efficient. Helmert (2009) discusses a substantial number of planning approaches that benefit from such a conversion. Success stories include a SAT-based planner (Chen, Zhao, and Zhang 2007), a planner based on integer programming (van den Briel, Vossen, and Kambhampati 2005), symbolic planning with BDDs (Edelkamp and Helmert 2001) and heuristic search planners using pattern databases (Edelkamp 2001; Haslum et al. 2007), merge-and-shrink abstractions (Helmert, Haslum, and Hoffmann 2007), the causal graph heuristic (Helmert 2004) or the context-enhanced additive heuristic (Helmert and Geffner 2008). In all these cases, finite-domain fluents are derived by combining groups of Boolean variables that cannot be true simultaneously (i. e., which are *mutex*).

In this paper, we examine additional ways of merging fluents in order to facilitate planning, using a representation where mutex propositions have already been combined as a starting point. Our work is inspired by an article by van den Briel, Kambhampati, and Vossen (2007), in which the authors describe the possible benefits of general fluent merging for planning. They claim that only combining mutex

propositions is too conservative and propose the combination of finite-domain variables that have “strong dependencies”. They mention two possible methods for discovering such dependencies. The first method merges variables with the property that *all* operators that change one of the variables also mention the other variable in a precondition or effect. The second method merges variables with the property that at least one operator has a precondition but no effect on the first variable and an effect on the second variable. (However, this is a very general criterion and is often satisfied by thousands of variable pairs in a planning task, not all of which can be merged within practical resource limits.)

Van den Briel et al. present examples that indicate that their ideas could lead to improved planner performance, but do not report experimental results or provide a precise algorithm. They suggest that further research on the topic is necessary.

## Formal Semantics

We formalise finite-domain planning tasks using the  $SAS^+$  formalism (Bäckström and Nebel 1995), largely following the notation of Helmert, Haslum, and Hoffmann (2007). (We briefly remark that our implementation has been extended to cover finite-domain representations allowing conditional effects, but we limit ourselves to the easier  $SAS^+$  case here for simplicity of presentation.)

**Definition 1** ( $SAS^+$  planning task)

An  $SAS^+$  **planning task** or  $SAS^+$  **task** for short is a 4-tuple  $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$  with the following components:

- $\mathcal{V} = \{v_1, \dots, v_n\}$  is a set of **state variables** or **fluents**, each with an associated finite domain  $\mathcal{D}_v$ . If  $d \in \mathcal{D}_v$  we call the pair  $v = d$  an **atom**.  
A **partial variable assignment** over  $\mathcal{V}$  is a function  $s$  on some subset of  $\mathcal{V}$  such that  $s(v) \in \mathcal{D}_v$  wherever  $s(v)$  is defined. If  $s(v)$  is defined for all  $v \in \mathcal{V}$ ,  $s$  is called a **state**.
- $\mathcal{O}$  is a set of **operators**, where an operator is a triple  $\langle \text{name}, \text{pre}, \text{eff} \rangle$  where **name**, the **name** of the operator is a unique symbol that distinguishes this operator from others, and **pre** and **eff** are partial variable assignments called **preconditions** and **effects**, respectively.
- $s_0$  is a state called the **initial state**, and  $s_\star$  is a partial variable assignment called the **goal**.



We assume that the reader is familiar with the semantics of planning tasks (operator application, plans, etc.) and refer to the literature for details (e. g., Helmert, Haslum, and Hoffmann 2007). To clarify notation, we only recall one important definition, that of *transition systems*.

**Definition 2** (transition systems)

A **transition system** is a 5-tuple  $\mathcal{T} = \langle S, L, T, s_0, S_\star \rangle$  such that

- $S$  is a finite set called the set of **states** of  $\mathcal{T}$ ,
- $L$  is a finite set called the set of **transition labels** or **labels** of  $\mathcal{T}$ ,
- $T \subseteq S \times L \times S$  is the set of **transitions** of  $\mathcal{T}$ ,
- $s_0 \in S$  is the **initial state** of  $\mathcal{T}$ , and
- $S_\star \subseteq S$  is the set of **goal states** of  $\mathcal{T}$ .

We write  $(s \xrightarrow{l} s') \in T$  or simply  $s \xrightarrow{l} t$  when  $T$  is clear from context to denote that  $\mathcal{T}$  has a transition from  $s$  to  $s'$  with label  $l$ , i. e., to denote  $\langle s, l, s' \rangle \in T$ .

Planning semantics are defined in terms of transition systems. Put briefly, each planning task  $\Pi$  defines a transition system whose states are the states of  $\Pi$  (i. e., the complete assignments to the fluents of  $\Pi$ ), whose initial and goal states match the initial and goal states of  $\Pi$ , and whose transitions are defined by the semantics of operator application for the operators of  $\Pi$ , with transition labels corresponding to operator names.

In addition to the transition system of the complete planning task  $\Pi$ , for the purposes of fluent merging we are also interested in more localised views that only capture the semantics of planning tasks with respect to a *particular fluent*. This can be achieved by considering transition systems *induced by atomic abstractions*. Again, we only provide the definition for the limited case that is important for this work and refer to the literature for more general definitions of abstractions and induced transition systems (Helmert, Haslum, and Hoffmann 2007).

**Definition 3** (atomic abstractions)

Let  $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$  be an  $\text{SAS}^+$  task, and let  $v \in \mathcal{V}$  be one of its state variables. The **transition system induced by the atomic abstraction to  $v$** , or more succinctly the **transition system for  $v$** , is the transition system  $\mathcal{T}_v = \langle S^v, L^v, T^v, s_0^v, S_\star^v \rangle$  such that:

- $S_v = \mathcal{D}_v$  (i. e., the domain of  $v$  forms the states of  $\mathcal{T}_v$ ),
- $L^v$  is the set of operator names in  $\mathcal{O}$ ,
- there is a transition  $d \xrightarrow{l} d'$  whenever the transition system defined by  $\Pi$  has a transition  $s \xrightarrow{l} s'$  with states  $s$  and  $s'$  such that  $s(v) = d$  and  $s'(v) = d'$ ,
- $s_0^v = s_0(v)$  (i. e., the initial state in  $\mathcal{T}_v$  is the value of  $v$  in the initial state of  $\Pi$ ), and
- $S_\star^v$  consists of all possible values  $d \in \mathcal{D}_v$  that can occur in goal states of  $\Pi$ .

Even though they are defined *semantically*, based on the exponentially large transition systems of planning tasks, transition systems induced by atomic abstractions in  $\text{SAS}^+$

tasks can be computed *syntactically*, i. e., using efficient operations directly on the compact task representation  $\Pi$ . In particular, an operator  $\langle \text{name}, \text{pre}, \text{eff} \rangle$  induces a transition transition  $d \xrightarrow{\text{name}} d'$  in the transition system for  $v$  iff

- $d$  is compatible with  $\text{pre}(v)$ , i. e.,  $\text{pre}(v)$  is undefined or  $\text{pre}(v) = d$ , and
- $d'$  is compatible with  $\text{eff}(v)$ , i. e.,  $\text{eff}(v)$  is undefined or  $\text{eff}(v) = d'$ .

Moreover, it is not necessary to iterate over all (possibly exponentially many) goal states in order to determine  $S_\star^v$ : rather, the set of abstract goal states is simply the complete domain  $\mathcal{D}_v$  if  $s_\star(v)$  is undefined, and  $\{s_\star(v)\}$  otherwise.

A transition system for a variable  $v$  can be viewed as a labeled directed graph, and it shares many similarities with *domain transition graphs* (DTGs), introduced by Jonsson and Bäckström (1998). Van den Briel et al. use DTGs as the basis for defining fluent merging. However, there are some semantic differences between the two kinds of graphs, which make definitions of fluent merging based on DTGs slightly more complicated. In particular, atomic abstractions for  $v$  represent the behaviour of *all* operators with respect to  $v$ , while DTGs only consider operators that change the value of the represented variable.

Given only the DTGs of a planning task, it is not possible to reconstruct which operators have preconditions on variables that they do not modify. Given only the transition systems for individual variables, however, it is possible to reconstruct the complete transition system of an  $\text{SAS}^+$  task (Helmert, Haslum, and Hoffmann 2007, Theorem 8 and following discussion). This is done through the operation of computing *synchronised products* (Helmert, Haslum, and Hoffmann 2007), which also provide the formal underpinnings of fluent merging.

**Definition 4** (synchronised product)

Let  $\mathcal{T}^1 = \langle S^1, L, T^1, s_0^1, S_\star^1 \rangle$  and  $\mathcal{T}^2 = \langle S^2, L, T^2, s_0^2, S_\star^2 \rangle$  be transition systems with the same labels.

The **synchronised product** of  $\mathcal{T}^1$  and  $\mathcal{T}^2$  is defined as  $\mathcal{T}^1 \otimes \mathcal{T}^2 = \langle S, L, T, s_0, S_\star \rangle$ , where

- $S = S^1 \times S^2$ ,
- $(\langle s^1, s^2 \rangle \xrightarrow{l} \langle t^1, t^2 \rangle) \in T$  iff  $(s^1 \xrightarrow{l} t^1) \in T^1$  and  $(s^2 \xrightarrow{l} t^2) \in T^2$ ,
- $s_0 = \langle s_0^1, s_0^2 \rangle$ , and
- $S_\star = S_\star^1 \times S_\star^2$ .

Synchronised products play an important role in the computation of so-called *merge-and-shrink abstractions*; they correspond to the *merge* steps in the abstraction computation (Helmert, Haslum, and Hoffmann 2007). They also provide a clean and direct semantics for fluent merging in  $\text{SAS}^+$  tasks. As discussed above,  $\text{SAS}^+$  tasks can be equivalently represented through the set of all atomic transition systems,  $\{\mathcal{T}_v \mid v \in \mathcal{V}\}$ . *Fluent merging* then means choosing two fluents  $u, v \in \mathcal{V}$ , removing their transition systems  $\mathcal{T}_u$  and  $\mathcal{T}_v$  from the set, and replacing them with their synchronised product,  $\mathcal{T}_u \otimes \mathcal{T}_v$ . If we interpret this at the task level, this can be seen as replacing fluents  $u$  and  $v$  with a *product fluent*  $u \otimes v$ , a view which we will take on in the following.

There is one small issue that makes this symmetry between state variables and transition systems imperfect: in cases where a goal value is defined for one of  $u$  and  $v$  but not the other, there is no clean way of defining a goal value for the product fluent  $u \otimes v$ . However, this can easily be addressed by standard compilation techniques to compile away disjunctive goals (Gazen and Knoblock 1997).

### Fluent Merging in Fast Downward

Based on the theoretical definition of Fluent Merging we briefly discuss how we implemented the technique by integrating it into the Fast Downward planning framework (Helmert 2006). First we briefly explain the framework itself. In order to find a plan, Fast Downward proceeds in three main stages (Helmert 2006):

- In the *translation* stage, a PDDL (McDermott et al. 1998) problem is parsed, normalised, grounded and translated into an SAS<sup>+</sup> planning task.
- In the *knowledge compilation* stage a relevance analysis is performed and some data structures are prepared for the last stage.
- In the last stage the actual *search* is executed. For this purpose many different heuristics and search methods are available.

We integrate fluent merging between the first and second stage. The fluent merging algorithm reads the SAS<sup>+</sup> planning task that was written by the translator and outputs a new SAS<sup>+</sup> planning task with merged variables. The new component can be integrated easily since none of the original components have to be altered.

### Fluent Merging Algorithm

The fluent merging algorithm is composed of two steps. In the first step, we select groups of variables that are then merged in the second step. We tested many different *selection* methods and will discuss them later. This section explains the second step, the generic *merging* procedure.

The merging procedure follows the definition of synchronised products that provides the formal semantics of fluent merging, but unlike that definition works directly on the task description level.

Let  $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_* \rangle$  be an SAS<sup>+</sup> planning task and  $a, b \in \mathcal{V}$  the variables that we want to replace with a merged variable  $a \otimes b$ .

**Merged Variable** The new variable is assigned the domain  $\mathcal{D}_a \times \mathcal{D}_b$ .

**Initial State** We set  $s_0(a \otimes b) = \langle s_0(a), s_0(b) \rangle$ .

**Operators** Every operator that mentions  $a$  or  $b$  in its precondition or effect needs to be updated. In some cases, a single operator needs to be replaced by multiple new operators. All new operators are assigned the same name as the original operator, so that plans for the modified problem (described as sequences of operators represented by their names) can be used verbatim as plans for the original problem. The algorithm for adapting an operator is shown in Figure 1. The

**procedure** adapt-operator(ops  $\mathcal{O}$ , op  $o$ , var  $a$ , var  $b$ )

1.  $\langle name, pre, eff \rangle := o$
2. **if**  $o$  mentions neither  $a$  nor  $b$  **then**
3.   **return**
4. **if**  $eff(a)$  and  $eff(b)$  are defined **and**  $pre(a)$  and  $pre(b)$  are undefined **then**
5.    $eff(a \otimes b) := \langle eff(a), eff(b) \rangle$
6.   **return**
7.  $poss_a := \{pre(a)\}$  if defined, else  $\mathcal{D}_a$
8.  $poss_b := \{pre(b)\}$  if defined, else  $\mathcal{D}_b$
9. **foreach**  $a_{pre} \in poss_a$  **do**
10.   **foreach**  $b_{pre} \in poss_b$  **do**
11.      $a_{eff} := eff(a)$  if defined, else  $a_{pre}$
12.      $b_{eff} := eff(b)$  if defined, else  $b_{pre}$
13.      $pre_{new} := pre$
14.      $eff_{new} := eff$
15.      $pre_{new}(a \otimes b) := \langle a_{pre}, b_{pre} \rangle$
16.      $eff_{new}(a \otimes b) := \langle a_{eff}, b_{eff} \rangle$
17.      $o_{new} := \langle pre_{new}, eff_{new} \rangle$
18.      $\mathcal{O} := \mathcal{O} \cup \{ \langle name, pre_{new}, eff_{new} \rangle \}$
19.  $\mathcal{O} := \mathcal{O} \setminus \{o\}$

Figure 1: Algorithm that adapts an operator  $o \in \mathcal{O}$  during the merge of variables  $a$  and  $b$ .

special cases in lines 2–6 are not strictly necessary, but speed up the computation and lead to a more compact result in common cases.

**Goal** We have to distinguish three cases:

- $s_*(a)$  and  $s_*(b)$  undefined:  
Nothing to do.
- $s_*(a)$  and  $s_*(b)$  both defined:  
Set  $s_*(a \otimes b) = \langle s_*(a), s_*(b) \rangle$ .
- Exactly one of  $s_*(a)$  and  $s_*(b)$  is defined:  
Without loss of generality, we assume that  $s_*(a)$  is defined. Then any of the values in the set  $\{ \langle s_*(a), d \rangle \mid d \in \mathcal{D}_b \}$  should be treated as a possible goal value for  $a \otimes b$ . Since SAS<sup>+</sup> cannot represent goals of this form directly, we use a standard compilation technique for first compiling the actual goal into an operator (Gazen and Knoblock 1997) and then adapt this operator as described previously.

After these steps, all references to the old variables  $a$  and  $b$  can be removed from the task description.

While the algorithm as described only merges two variables at a time, it can be invoked repeatedly to merge “groups” or “clusters” of more than two variables. For example, to merge variables  $a$ ,  $b$  and  $c$  into a single group, we would first merge  $a$  and  $b$  into  $a \otimes b$  and then  $a \otimes b$  and  $c$

into  $(a \otimes b) \otimes c$ . Synchronised product operations are associative and commutative modulo isomorphism of transition systems, so the precise merge order does not matter.

### Variable Selection

There are many possible criteria for finding variables to merge. In this section we present the methods that we have implemented in the course of this work. A *selection method* is an algorithm that has the following input:

- An SAS<sup>+</sup> planning task  $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_* \rangle$
- $m$ , the number of variables that should be merged into a single “group” or “cluster”
- $n$ , the maximum number of such groups to form

It returns a set of variable groups to be merged by the combination algorithm. Usually it will produce  $m$  groups of size  $n$  each, unless this is not possible because  $m \cdot n > |\mathcal{V}|$ . A selection method may also opt to produce smaller or fewer groups if it cannot find a sufficient number of suitably large promising candidate groups to merge, but this is not the case for the simpler methods described in this section.

The first set of experiments was conducted with the selection methods below. In these methods, each group is assigned a score that represents its suitability to be merged. This assignment is done by an evaluation function  $e : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}$  that assigns a numerical value to all pairs of variables in the planning task. If  $m \geq 3$ , i. e., we seek to merge groups of three or more variables, variable groups  $G$  of size  $m$  are scored by computing the sum of all pairwise scores for pairs of variables in  $G$ . In the last step of the selection algorithm all groups are sorted in descending order of scores and we repeatedly pick the first group in the list until the maximum number of merges  $n$  has been reached. Groups to be merged are not allowed to overlap: once a variable has formed part of a merge, all groups that contain it are eliminated from further consideration.

We experimented with the following evaluation functions:

- Random variables (rand)  
Randomly select variable groups.  
$$e(a, b) = \text{random}()$$
- Mutex variables (mutex)  
Prefer groups with variables whose domains are maximally mutex, i. e., contain as many value pairs as possible that cannot be simultaneously true according to the mutex information generated by Fast Downward’s translation algorithm.  
$$e(a, b) = |\{(d_a, d_b) \in \mathcal{D}_a \times \mathcal{D}_b \mid d_a \text{ and } d_b \text{ mutex}\}|$$
- Number of atoms (size)  
Choose variables whose merging minimises the total number of atoms of the planning task.  
$$e(a, b) = -(|\mathcal{D}_{a \otimes b}| - (|\mathcal{D}_a| + |\mathcal{D}_b|))$$
- Connected variables (conn)  
Prefer variable groups that are heavily connected in the causal graph.  
$$\text{conn}(a, b) = \text{cg\_weight}(a, b) + \text{cg\_weight}(b, a)$$

Domain	no-merge	rand	mutex	size	conn	cycles	goals	ops
blocks (35)	35	35	35	35	31	31	31	35
driverlog (20)	20	17	13	16	19	14	18	15
grid (5)	5	1	1	5	0	1	1	2
gripper (20)	20	20	15	20	20	20	20	20
logistics00 (28)	28	28	28	28	28	28	28	28
logistics98 (35)	35	28	35	35	20	20	21	11
miconic (150)	150	150	150	150	150	150	150	150
mprime (35)	35	30	35	35	34	29	35	20
psr-small (50)	50	49	48	48	47	48	47	49
zenotravel (20)	20	20	16	16	20	20	19	15
depot (22)	17	11	14	12	<b>15</b>	<b>15</b>	13	14
freecell (80)	78	75	<b>77</b>	76	72	72	57	37
pathways (30)	15	14	<b>16</b>	<b>17</b>	14	14	13	15
pipes-nt (50)	38	5	8	<b>16</b>	14	14	9	<b>16</b>
pipes-t (50)	24	9	3	<b>17</b>	11	8	9	15
rovers (40)	34	31	34	<b>35</b>	34	34	34	24
schedule (150)	60	58	59	59	54	52	39	<b>60</b>
tpp (30)	28	20	<b>24</b>	<b>24</b>	22	<b>24</b>	23	16
trucks (30)	17	15	14	<b>16</b>	14	14	<b>16</b>	6
Total (880)	709	616	625	<b>660</b>	619	608	583	548

Table 1: Comparison of solved tasks for different fluent merging methods with the  $h^{\text{cea}}$  heuristic,  $n = 5$  and  $m = 2$ . Number of tasks in each domain is shown in parentheses. In the domains above the separator line *no-merge* already solves all instances, so no improvement is possible. Below the separator the best results among the different selection methods are highlighted in bold and cases where the performance of the *no-merge* method is exceeded are underlined.

$$e(a, b) = \text{conn}(a, b)$$

Here, the *cg\_weight* of a causal graph edge is the number of operators that induce it (Helmert 2006).

- Two-cycle pairs (cycles)  
Prefer variables that form a two-cycle in the causal graph.

$$e(a, b) = \begin{cases} \text{conn}(a, b) & \text{if } \langle a, b \rangle \in E \\ & \text{and } \langle b, a \rangle \in E \\ \text{conn}(a, b) - 10^9 & \text{else} \end{cases}$$

Here  $E$  is the set of directed edges of the causal graph.

- Goal variables (goals)  
Prefer variables that appear in the goal description.

$$e(a, b) = \begin{cases} \text{conn}(a, b) & \text{if } s_*(a) \text{ or } s_*(b) \text{ defined} \\ \text{conn}(a, b) - 10^9 & \text{else} \end{cases}$$

- Variables minimizing number of operators (ops)  
Prefer variable groups whose merging minimise the number of new operators.

$$e(a, b) = |\mathcal{O}| - |\mathcal{O}_{\text{new}}|$$

Here  $\mathcal{O}_{\text{new}}$  is the set of operators that would result from merging  $a$  and  $b$ .

Table 1 shows the number of tasks solved by a greedy best-first search with deferred evaluation (Richter and Helmert 2009) and the  $h^{\text{cea}}$  heuristic in a number of IPC domains after applying fluent merging with the selection methods mentioned above. For the experiments we allowed the

search component of the planner to run for at most 30 minutes and use 2 GB of memory. The column *no-merge* reports results without performing fluent merging. The maximum group size  $m$  is set to 2 in these experiments, while the maximum number of groups  $n$  is set to 5. Other experiments ( $n \in \{10, 20, \dots, 100, 125, 150, 175, 200, 250, 300\}$ ,  $m \in \{3, 4, 5, 6, 7\}$ ) all led to worse performance, i. e., fewer plans found in more time.

Although some of the entries in the table show improvements by performing fluent merging, we were not satisfied with the overall results. While looking for a smarter selection method we found the *same object method*, which performed significantly better in the Schedule and Pathways domains. (Unfortunately, other domains were not tested at the time.) Between the initial tests with the above mentioned methods and the experiments with the same object method, the implementation of the  $h^{cea}$  heuristic in Fast Downward was improved, making a direct comparison to the other methods' results difficult. We are currently rerunning all experiments with the improved  $h^{cea}$  version.

### Same Object Method

The “same object method” exploits the fact that planning tasks are typically not given directly in a grounded representation like  $SAS^+$ , but instead use a first-order PDDL representation based on logical predicates and objects. As a first approximation, it is not entirely unreasonable to assume that  $SAS^+$  fluents that stem from PDDL propositions that talk about the same object are more closely related than ones which do not. For example, even without looking at any operator definitions, a human problem solver might suspect that the two grounded propositions (*painted chair1*) and (*polished chair1*) are more closely related to each other than the two grounded propositions (*painted chair1*) and (*polished table3*) because they speak of the same object, *chair1*.

The same object method starts by associating exactly one object from the input PDDL representation with each  $SAS^+$  fluent. Recall that each  $SAS^+$  fluent  $v$  (before we apply our fluent merging algorithm) is formed from a group  $A_v$  of mutually exclusive PDDL atoms. The object associated with  $v$  is simply the object that occurs most frequently as a term in the atoms  $A_v$ . (Tie-breaking rules are applied when there is no unique such atom.) For example, variable  $v$  might be derived from mutex group  $A_v = \{(at\ c2\ loc1), (at\ c2\ loc2), (at\ c2\ loc3)\}$ , which mentions the objects *c2*, *loc1*, *loc2* and *loc3*. Of these objects, *c2* is mentioned most frequently and hence becomes the object associated with  $v$ .

We only allow merges of fluents that are associated with the same object. However, since this can still lead to merged fluents with very large domains, we again limit the number of variables to be merged into a single group and the number of groups to merge, as in the previous algorithms. After some experimentation and following the comparatively good performance of the “size” method in the previous experiment, we decided to use the combined domain size of a group, i. e., the product of the domain sizes of the involved variables, as the quality measure for a merge, preferring

groups whose combined domain size is as low as possible.

## Experiments

In our initial experiments, the same object method significantly outperformed the other variable selection methods we tried, so all our subsequent experiments were based on this approach. After discouraging initial results with the merge-and-shrink heuristic (Helmert, Haslum, and Hoffmann 2007) and the landmark-cut heuristic (Helmert and Domshlak 2009), we concentrated our further experiments on satisficing configurations of Fast Downward. We again used greedy best-first search with deferred evaluation and tested three different heuristics:

- $h^{cea}$ : the context-enhanced additive heuristic (Helmert and Geffner 2008)
- $h^{FF}$ : the FF/additive heuristic (Hoffmann and Nebel 2001; Keyder and Geffner 2008)
- $h^{CG}$ : the causal graph heuristic (Helmert 2004)

In all our experiments, forming groups of only two fluents ( $m = 2$ ) produced better results than using larger clusters, so we only present results for this case. For the second fluent merging parameter, the number  $n$  of groups to form, the picture is more varied and differs significantly from domain to domain. Therefore, we report results for different values of this parameter, taken from the set  $\{0$  (no merges),  $2, 5, 10, 15, 20, 30\}$ . For all experiments we used a 30 minute timeout for the search component of the planner.

The  $h^{cea}$  heuristic could not be significantly improved by fluent merging. Here only three more planning tasks could be solved by combining variables, and the overall coverage never improved.

The other two heuristics however showed some stronger potential benefits from fluent merging. As Table 2 shows, a total of 6 problem instances for  $h^{FF}$  and 12 instances for  $h^{CG}$  could be solved by some variation of fluent merging that eluded the same algorithm in the original problem representation. We should note that those two heuristics are already highly competitive planning methods. For  $h^{FF}$ , some parameter settings also achieve better *overall* performance in the tested domains. Table 3 provides detailed results for a particularly positive case, the challenging Sokoban domain, in which fluent merging increases the coverage of  $h^{FF}$  from 24 to 29 (out of 30) solved instances.

## Conclusions and Future Work

We have provided the first general implementation and experimental evaluation of fluent merging for classical planning. Our results show that the approach holds promise: in some domains and with some combination methods, simply reformulating a problem instance by merging certain pairs of fluents improved problem solving performance.

However, our results also make it obvious that fluent merging does not improve heuristic accuracy across the board, and that further research is needed to find out which and how many fluents to merge, and whether fluent merging is useful for a given planning task at all.

Domain	Merges $h^{FF}$							Merges $h^{CG}$						
	0	2	5	10	15	20	30	0	2	5	10	15	20	30
airport (50)	25	25	25	25	25	25	25	<b>21</b>	<b>21</b>	<b>21</b>	<b>21</b>	<b>21</b>	20	20
assembly (30)	30	30	30	30	30	30	30	6	<b>7</b>	<b>7</b>	<b>7</b>	<b>7</b>	<b>7</b>	<b>7</b>
depot (22)	19	18	19	<b>20</b>	<b>20</b>	<b>20</b>	<b>20</b>	12	12	12	12	<b>13</b>	<b>13</b>	<b>13</b>
driverlog (20)	20	20	20	20	20	20	20	<b>20</b>	<b>20</b>	18	18	18	18	18
freecell (80)	76	<b>80</b>	78	77	79	78	75	72	70	70	<b>75</b>	74	74	72
miconic (150)	<b>150</b>	<b>150</b>	<b>150</b>	<b>150</b>	<b>150</b>	80	80	<b>150</b>	<b>150</b>	<b>150</b>	<b>150</b>	<b>150</b>	80	80
pprinter (30)	<b>23</b>	22	22	22	22	22	22	<b>24</b>	23	23	22	22	22	22
pipes-nt (50)	<b>43</b>	41	42	42	<b>43</b>	42	42	24	23	24	25	25	25	<b>26</b>
pipes-t (50)	38	<b>39</b>	38	37	<b>39</b>	37	37	17	<b>18</b>	17	15	16	15	15
psr-small (50)	50	50	50	50	50	50	50	50	50	50	50	50	50	50
rovers (40)	<b>40</b>	<b>40</b>	<b>40</b>	<b>40</b>	<b>40</b>	<b>40</b>	37	<b>32</b>	31	<b>32</b>	31	31	<b>32</b>	<b>32</b>
satellite (36)	34	34	34	34	34	34	34	34	34	34	34	34	34	34
schedule (150)	<b>150</b>	149	149	149	149	149	148	149	149	149	149	149	149	149
sokoban-sat (30)	24	28	<b>29</b>	28	28	28	28	<b>27</b>	26	24	25	25	25	25
storage (30)	<b>20</b>	<b>20</b>	<b>20</b>	<b>20</b>	19	19	19	20	20	20	20	20	20	20
tpp (30)	30	30	30	30	30	30	30	<b>27</b>	<b>27</b>	<b>27</b>	<b>27</b>	<b>27</b>	<b>27</b>	26
trucks (30)	<b>19</b>	17	17	18	18	18	18	10	11	11	11	11	<b>12</b>	11
wood-sat (30)	<b>29</b>	<b>29</b>	28	28	28	28	<b>29</b>	11	11	11	11	11	<b>14</b>	12
<b>Total (908)</b>	<b>820</b>	<b>822</b>	<b>821</b>	<b>820</b>	<b>824</b>	750	744	<b>706</b>	703	700	703	704	637	632

Table 2: Comparison of solved tasks for different maximum numbers of merges using the heuristics  $h^{FF}$  and  $h^{CG}$ . The maximum group size is set to 2 in this experiment. Number of tasks in each domain is shown in parentheses. Best results for each heuristic are highlighted in bold.

Inst.	$h^{FF}$ 0 Merges			$h^{FF}$ 2 Merges			$h^{FF}$ 5 Merges		
	Cost	Exp.	Time	Cost	Exp.	Time	Cost	Exp.	Time
Sokoban									
#16	345	29682	4.18	<b>329</b>	<b>21743</b>	<b>3.97</b>	371	24656	5.39
#17	<b>114</b>	<b>8543</b>	<b>1.27</b>	215	29104	5.31	247	38733	8.51
#18	497	2421586	314.03	<b>275</b>	1033770	172.36	301	<b>752976</b>	<b>152.67</b>
#19				93	<b>351433</b>	<b>182.57</b>	<b>52</b>	776873	476.85
#20									
#21	256	75853	11.23	<b>224</b>	<b>41600</b>	<b>8.62</b>	236	81226	19.9
#22				<b>389</b>	4024596	<b>767.93</b>	409	<b>3824691</b>	833.27
#23	343	<b>24924</b>	<b>2.98</b>	311	31248	5.46	<b>299</b>	27809	5.68
#24	<b>137</b>	173933	<b>26.48</b>	165	219517	40.26	<b>137</b>	<b>139241</b>	30.57
#25	221	<b>71862</b>	<b>12.14</b>	<b>185</b>	74991	14.67	245	89785	20.78
#26	402	<b>577660</b>	<b>110.98</b>	<b>320</b>	794300	200.1	434	1404490	420.94
#27				113	<b>529446</b>	<b>197.51</b>	<b>97</b>	923336	412.6
#28				536	<b>1983688</b>	<b>557.24</b>	<b>486</b>	3858328	1287.57
#29							<b>779</b>	<b>4673208</b>	<b>914.51</b>
#30	502	886401	134.08	494	719545	<b>116.84</b>	<b>470</b>	<b>677857</b>	127.8

Table 3: Detailed results for the sokoban-sat08-strips domain (15 smallest tasks omitted), using greedy best-first search with deferred evaluation and  $h^{FF}$ . The maximum group size was set to 2. For each number of merges we report the plan cost, number of expansions and search time in seconds. Best results are highlighted in bold.

Additionally, it can be expected that the parameters for the maximum number of merges and the maximum group size were not optimally set in our experiments. With current planning systems obtaining more and more knobs to tweak, the automatic parameter tuning methods like the ones found in the ParamILS framework appear worth investigating (Hutter et al. 2009).

Finally, we remark that in this work, we used the finite-domain representations generated by Fast Downward's translation component as a starting point. As van den Briel, Kambhampati, and Vossen (2007) observe, the reformulation performed by this translator is already a form of fluent merging (based on mutexes of the planning instance at hand), and it is far from clear whether the particular merging choices performed by the translation algorithm are ideal. Hence, another interesting question is whether we can derive a general fluent merging algorithm that starts from a regular Boolean encoding of a planning task and leads to a better representation than the one found by Fast Downward's default algorithm.

## References

- Bäckström, C., and Nebel, B. 1995. Complexity results for SAS<sup>+</sup> planning. *Computational Intelligence* 11(4):625–655.
- Chen, Y.; Zhao, X.; and Zhang, W. 2007. Long-distance mutual exclusion for propositional planning. In Veloso, M. M., ed., *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, 1840–1845.
- Edelkamp, S., and Helmert, M. 2001. The model checking integrated planning system (MIPS). *AI Magazine* 22(3):67–71.
- Edelkamp, S. 2001. Planning with pattern databases. In Cesta, A., and Borrajo, D., eds., *Pre-proceedings of the Sixth European Conference on Planning (ECP 2001)*, 13–24.
- Gazen, B. C., and Knoblock, C. A. 1997. Combining the expressivity of UCPOP with the efficiency of Graphplan. In Steel, S., and Alami, R., eds., *Recent Advances in AI Planning. 4th European Conference on Planning (ECP 1997)*, volume 1348 of *Lecture Notes in Artificial Intelligence*, 221–233. Springer-Verlag.
- Haslum, P.; Botea, A.; Helmert, M.; Bonet, B.; and Koenig, S. 2007. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, 1007–1012. AAAI Press.
- Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 162–169. AAAI Press.
- Helmert, M., and Geffner, H. 2008. Unifying the causal graph and additive heuristics. In Rintanen, J.; Nebel, B.; Beck, J. C.; and Hansen, E., eds., *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, 140–147. AAAI Press.
- Helmert, M.; Haslum, P.; and Hoffmann, J. 2007. Flexible abstraction heuristics for optimal sequential planning. In Boddy, M.; Fox, M.; and Thiébaux, S., eds., *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, 176–183. AAAI Press.
- Helmert, M. 2004. A planning heuristic based on causal graph analysis. In Zilberstein, S.; Koehler, J.; and Koenig, S., eds., *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, 161–170. AAAI Press.
- Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.
- Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173:503–535.
- Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.
- Hutter, F.; Hoos, H. H.; Leyton-Brown, K.; and Stützle, T. 2009. ParamILS: an automatic algorithm configuration framework. *Journal of Artificial Intelligence Research* 36:267–306.
- Jonsson, P., and Bäckström, C. 1998. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence* 100(1–2):125–176.
- Keyder, E., and Geffner, H. 2008. Heuristics for planning with action costs revisited. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*, 588–592.
- 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.
- Richter, S., and Helmert, M. 2009. Preferred operators and deferred evaluation in satisficing planning. In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, 273–280. AAAI Press.
- van den Briel, M.; Kambhampati, S.; and Vossen, T. 2007. Fluent merging: A general technique to improve reachability heuristics and factored planning. In *ICAPS 2007 Workshop on Heuristics for Domain-Independent Planning: Progress, Ideas, Limitations, Challenges*.
- van den Briel, M.; Vossen, T.; and Kambhampati, S. 2005. Reviving integer programming approaches for AI planning: A branch-and-cut framework. In Biundo, S.; Myers, K.; and Rajan, K., eds., *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005)*, 310–319. AAAI Press.

# Heuristic Search-Based Planning for Graph Transformation Systems

H.-Christian Estler<sup>1</sup> and Heike Wehrheim<sup>2</sup>

<sup>1</sup>ETH Zurich, christian.estler@inf.ethz.ch

<sup>2</sup>University of Paderborn, wehrheim@uni-paderborn.de

## Abstract

Graph notations have proven effective in modeling complex systems. In recent years, it has therefore been proposed that graphs could also be used for modeling planning problems, *i.e.* using graphs to express states and actions. Translating these graphs into PDDL would be desirable since today's Graph Transformation tools are not as efficient as modern planning tools. Unfortunately, such a translation is not always possible as the formalisms have different expressiveness. In this work, we present an extension of a Graph Transformation tool with classic planning algorithms. Using two case studies, we show that this extension makes planning with graphs more feasible – without the need of translating into PDDL. Furthermore, we demonstrate how typical modeling artifacts, like meta-models, can be leveraged to semi-automatically develop heuristics for the planner.

## Introduction

Graphical notations are widely used in computer science to model complex systems. Their depiction often allows for an easier and faster understanding of the structure of a system and they can be more accessible to non-experts compared to purely text-based notations. A well known example of such a notation is the *Unified Modelling Language* (UML) (OMG 2010) which has widespread use in industry and academia.

To leverage the advantages of graphical notations in the area of planning, researchers are investigating how such notations could be used to model planning problems. Tools like ITSIMPLE (Vaquero, Tonidandel, and Silva 2005) allow the user to model a planning problem using different types of UML diagrams and subsequently generate planning problems in PDDL (Ghallab et al. 1998) which can be solved using off-the-shelf planning tools.

In this paper, we explore another approach towards solving planning problems which are modeled using UML diagrams. Rather than transforming the diagrams – which represent states and actions – into another language, we use them *as-is* when searching for

a plan. The diagrams themselves are (directed labeled) graphs and we can thus use a graph transformation tool for applying actions to states, thereby generating the search space. The advantage of this approach is that we can use the full expressiveness of the graph formalism which, in our case, is different from the expressiveness of PDDL. However, as graph transformations are in general computationally expensive, we are facing the fundamental problem of planning tools: the need to minimize the number of states in the search space (*i.e.* the number graph transformations).

We have built a planning framework that uses heuristic search algorithms (currently *A\** or *Best First*) to direct the search in a state space where new states are generated using a graph transformation tool. To the best of our knowledge, this is the first tool to experiment with such a combination. While it does not come as a surprise that a heuristic-driven approach typically uses less transformations than a non-heuristic approach, the evaluation of our framework yields insight of how efficient such a tool can perform in practice. Using our planning framework and two case studies, we will demonstrate i) that planning with graph transformation tools becomes more feasible when using heuristic search strategies instead of non-heuristic approaches, ii) how users can be enabled to write domain-specific heuristics for graph based planning problems and iii) how modeling artifacts, such as a meta-model, can be used to semi-automatically learn domain-specific heuristics.

## Case Studies

The first case study we present in this paper is the *n*-puzzle problem. *n* numbered tiles are positioned on a square board. The objective is to place the tiles in order, using only *slide* moves, *i.e.* only a tile adjacent to the empty field can slide onto that field.

The second case study has more of a “real-world problem” character. It is based on the research project *Neue Bahntechnik Paderborn* (NBP) at the University of Paderborn. NBP aims at the development of a future railway system where small, driverless vehicles act completely autonomously with respect to individual goals. The vehicles are called *RailCabs*, referring to the idea that the transport of passengers or goods is demand

driven, as it is with regular cabs.

A typical planning problem for the NBP case study is the following (see Fig. 2): passengers and cargo needs to be transported from Paderborn to Berlin. Furthermore, passengers need to be transported from Paderborn to Leipzig. The RailCabs (RC1, RC2, RC3) which are needed to satisfy these request share part a part of their route. Whenever possible, the RailCabs should build a *convoy* by driving close together, therewith minimizing energy consumption. Furthermore, every RailCab is required to be in contact with a *Base Station* (BS1, BS2, ...) to enable communication. Safety requirements such as “a RailCab with dangerous cargo is not allowed in a convoy” have to be met.

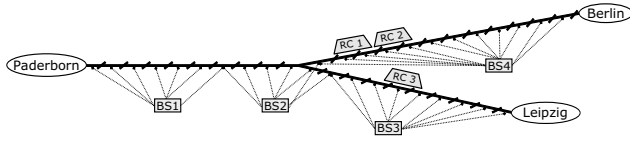


Fig. 2: Coordinating RailCabs constitutes a planning problem.

In total, the NBP planning problems consists of 15 possible actions (*e.g.* *move RailCab*, *create convoy*, *change Base Station*) and five rules which define safety requirements.

## Graphs and Graph Transformations

The graphs we use to model planning problems are called *story patterns* (Fischer et al. 2000). They were developed as part of an extension of UML activity diagrams.

In its most simple form, a story pattern equals an UML object diagram. We use this simple form to model the start state of a planning problem. Fig. 1 shows a start state of a NBP planning problem. Nodes and edges of the graph are labeled, where a label consists of two strings which are separated by a colon. The

front-string is the object name, whereas the rear-string defines the type of the object.

Story patterns are also used to define the actions of a planning problem. They describes how a graph can be transformed into another graph and are therefore also call it a *graph transformation rule*. An example for the “move” action of a RailCab is shown in Fig. 3.

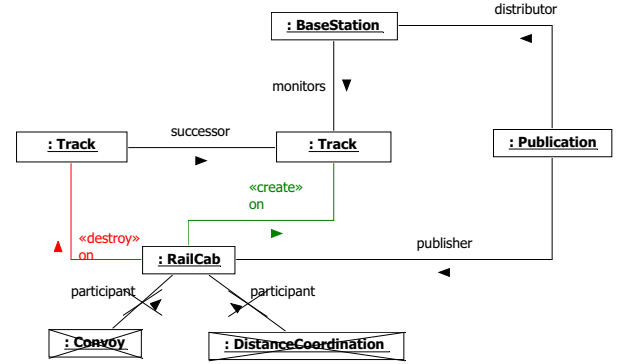


Fig. 3: Story pattern for the action *move*.

In addition to the nodes and edges of a regular object diagram, a graph transformation rule can use the special annotations *<<create>>* and *<<destroy>>* for nodes and edges. Furthermore, nodes and edges can be crossed out to define that the rule is only applicable to graphs which do not contain certain nodes or edges (called *Negative Application Conditions (NACs)*). In contrast to the start state graph from Fig. 1, the node labels omit a string in front of the colon. This omission defines that only the type of a node is of interest, not its specific object-name.

The execution of a transformation rule is performed in two steps: First, the graph to be changed (*e.g.* the one from Fig. 1) is searched for a subgraph which equals the graph of the transformation rule except for crossed out elements or those which are annotated with

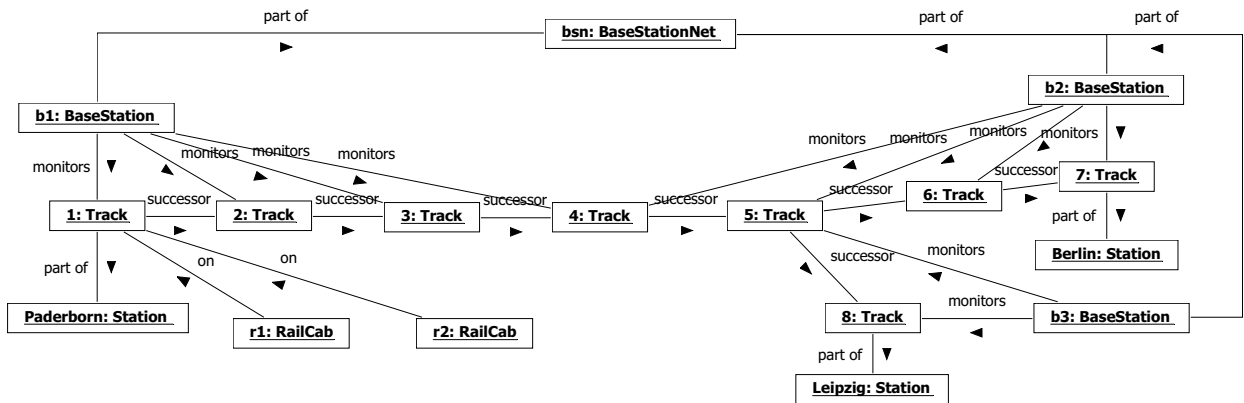


Fig. 1: Story pattern modeling a start state of a NBP planning problem.



<<create>>. If such a subgraph exists, then we have found a *matching*. Secondly, the subgraph will be modified by creating and deleting those elements which are annotated accordingly in the transformation rule.

Modeling goal states or states which are forbidden (e.g. due to safety requirements) is similar to modeling actions. The only difference is that we use story patterns without <<create>> or <<delete>> annotations. NACs, however, are allowed. The idea is to state only the properties that are of interest for a state, in order to be considered a goal state or a forbidden state. Fig. 4 shows an example of a goal state for the NBP problem.

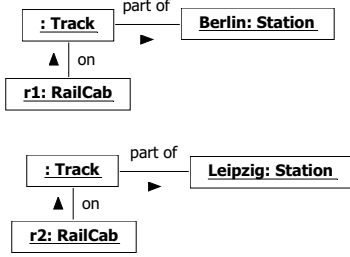


Fig. 4: Story patterns are used to define goal states.

The tuple  $(G, R)$ , where  $G$  is a set of graphs and  $R$  is a set of graph transformation rules is called a *Graph Transformation System* (GTS).

It is worthwhile to mention the complexity of graph transformations: Establishing a matching between a transformation rule and a graph implies that one has to find graph homomorphisms. Deciding if a homomorphism exists is  $\mathcal{NP}$ -complete. Furthermore, we need to check if a newly generated state is already present in the search space, i.e. for every new graph, we check if there exists a graph isomorphism to a graph already in the search space. Deciding graph isomorphism is known to be in  $\mathcal{NP}$ .

Though story patterns alone are sufficient to model a GTS planning problem, our framework additionally requires the user to model a UML class diagram. This class diagram defines the types (classes), labeled edges (associations) and possible node connections (multiplicity constraints) in a planning problem. The class diagram needs to be provided before modeling the story patterns. It serves as a meta-model, i.e. only nodes and edges which are declared in the class diagram can be used in a story pattern, ensuring consistency between different story patterns. An example of a class diagram for the NBP case study is shown in Fig. 5.

## The Planning Framework

Our planning framework utilizes two other tools. The first one is FUJABA<sup>1</sup> (From UML to Java and back again), an open-source tool for model-based software

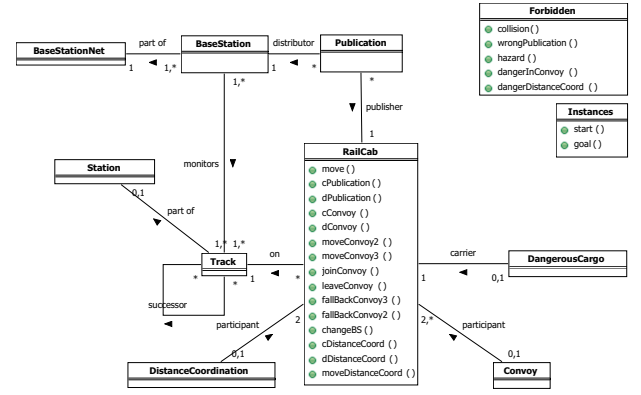


Fig. 5: UML class diagram for the NBP case study.

engineering. We use it as a front-end for the user input, i.e. a user models the story patterns and the class diagram within FUJABA.

The second tool is GROOVE<sup>2</sup> (Rensink 2004). The main feature of GROOVE is its *simulator* which allows for generating and analyzing graph transition systems. While GROOVE also incorporates an editor to define graphs and transformation rules, we use it as a back-end only, i.e. we rely on it for graph transformations and isomorphism checks but we apply our own algorithms to control the generation of the state space.

The graph formalisms used by FUJABA and GROOVE are very similar but not identical. Therefore, we use a translation procedure presented by Röhs *et al.* in (Röhs 2009; Röhs and Wehrheim 2010). In principal, our framework can work without FUJABA in case a user prefers to model a GTS directly in GROOVE. Furthermore, we designed the planning framework to be as independent of a specific graph transformation tool as possible. For example, none of the original GROOVE source code has been modified. While it is not possible to simply exchange GROOVE for another graph transformation tool (some parts of the framework's implementation are GROOVE specific), large parts of the framework, could be reused as-is in case GROOVE should be replaced.

## Writing Heuristics

The heuristic search algorithms of our framework –  $A^*$  and *Best First* – rely upon heuristics in order to perform efficiently. From a knowledge engineering standpoint it is desirable that users can define heuristics with the same notation they use to define the planning problem. As a first step, however, we have to identify the properties and functionalities which such a notation should provide. Therefore, we currently only provide an API with about 30 methods that ease the development of heuristic functions for GTS.

<sup>1</sup><http://www.fujaba.de>

<sup>2</sup><http://groove.cs.utwente.nl>

The design of the API was driven by the following observations:

- We think about graph nodes in terms of their types and their object names. For example, a node that represents a particular RailCab has the type “RailCab” and the object name “r1”.
- The object name and the type are both labels of a node. Furthermore, a node has incoming and outgoing edges. These edges can have labels themselves. Source and target of an edge are nodes again.
- A node should allow us to easily access and analyze its close-by neighbors, *i.e.* the nodes which are at the source of an incoming edge or at the target of an outgoing edge.
- Checking reachability between nodes is crucial when developing heuristic functions for graphs. For example, we need to be able to check, if a RailCab node can reach a node that represents a station. The check should return the distance between the nodes or the list of nodes along the path. Furthermore, a reachability check should (optionally) take into account that edges are directed. It is also important that we can restrict the check in a way that only certain nodes are taken into account. For example, if we want to check reachability between a RailCab and a station, this check should use tracks but not Base Stations.

The API hides the internals of GROOVE’s graph data structures. For example, nodes and edges are assigned unique identifiers internally but such identifiers are of no use to us as we do not know their meaning. We must also not forget that a user models a planning problem within FUJABA and thus might not even know anything about GROOVE and its internal graph representations.

In this paper, we evaluate four different heuristics which have been implemented using the API. Two well-known heuristics are for the 8-puzzle:

- $h_{Puz}^1$  = the number of misplaced tiles, *i.e.* the number of tiles which are not in their goal position.
- $h_{Puz}^2$  = the sum of the *Manhattan distances* of every misplaced tile.

Furthermore, two heuristics for the NBP problem:

- $h_{NBP}^1$  = the sum of the shortest distance from the current position to the goal position for every RailCab; That is, we measure for each RailCab the minimal number of tracks between its current position and its goal position. Then, we add up all those values.
- $h_{NBP}^2 = \infty$  if any RailCab can no longer reach its goal position from its current position. Otherwise return 0.

To give the reader an idea of how the API is used in practice, we provide an example for  $h_{NBP}^1$  in listing 1. Though the source code may not be completely self-explanatory, we do not explain the details in this paper. Rather, we provide the key observation from our

experiments with the API: many properties of heuristic functions for GTS planning problems can be expressed using reachability tests between nodes in the graph.

We experimented with more sophisticated heuristics which, for example, take into account the possibility of building convoys and having different costs for different sorts of move actions (regular move, move in a 2-convoy, move in 3-convoy). We found that it quickly becomes quite cumbersome to write such heuristics by hand. Therefore, we developed a semi-automatic approach to writing heuristics that free the user from this burden.

## Learning Heuristics

Using our API for writing heuristics, we can easily implement methods that extract feature values, *e.g.* the number of RailCabs or the number of tracks, from a given graph. Not having to decide how such feature values relate to the costs of solving a planning problem simplifies the development of a heuristic. For our framework, we developed an experience-based learning approach, *i.e.* we solve many problem instances and learn a heuristic estimate from experience. For instance, we define a set of features and store the values of such features in a so called *feature vector*. By providing many feature vectors together with the cost value of solving the corresponding problem, a learning algorithm can derive a function (a *regression function* to be precise) that predicts the costs based on a feature vector only. The approach we use for learning a regression function is called *Support Vector Machines* (SVM) (Boser, Guyon, and Vapnik 1992).

Instead of embedding a specific SVM implementation directly within our planning framework, we utilize a machine learning framework called WEKA<sup>3</sup> (Hall et al. 2009), which not only provides different SVMs but also other learning techniques. This provides the flexibility to experiment with different SVMs without the need to modify any code and also allows for future experiments with other learning approaches.

A learning algorithm can only yield meaningful results if it is trained on a sufficiently large data set. Asking the user to provide hundreds or thousands of different problem instances is undesirable and impractical. Our framework accounts for this by providing a *problem instance generator* which can generate many unique problem instances based on a single problem specification.

The language we use to write such a problem specification is ALLOY (Jackson 2002; 2006). It is based on first-order relational logic which facilitates an automatic analysis. ALLOY models can be executed and analyzed with the *Alloy Analyzer*<sup>4</sup>. The Alloy Analyzer translates an ALLOY model into a boolean formula and

<sup>3</sup><http://www.cs.waikato.ac.nz/ml/weka/>

<sup>4</sup><http://alloy.mit.edu>

```

1  public double getHeuristicWeight(Graph stateGraph, Set<HSRule> goalRules, Set<HSRule> actionRules) {
2      Map<String, String> railCabStationMap = new HashMap<String, String>();
3      double factor = 0;
4      int distance = 0;
5
6      // find out the costs for the "move" rule
7      for(HSRule r: actionRules) {
8          if(r.getRuleName().equals("move")) {
9              factor = r.getWeight("Cost"); } }
10
11      // get the goal graph
12      Graph goalGraph = goalRules.iterator().next().getGrooveRule().lhs();
13
14      // access the goal graph and find out where each RailCab shall go
15      HSGraphAccess goalAcc = new HSGraphAccess(goalGraph);
16      Set<HSNode> goalRailCabs = goalAcc.getAllNodesWithEdge("RailCab");
17      for(HSNode goalRC: goalRailCabs) {
18          HSNode goalTrack = goalRC.getSingleTargetOfEdgeLabel("on");
19          String stationID = goalTrack.getSingleTargetOfEdgeLabel("part_of").getNodeID();
20          railCabStationMap.put(goalRC.getNodeID(), stationID); }
21
22      // determine the distance between the current RailCab positions and their goals
23      HSGraphAccess acc = new HSGraphAccess(stateGraph);
24      Set<HSNode> railCabs = acc.getAllNodesWithEdge("RailCab");
25      Set<HSNode> tracks = acc.getAllNodesWithEdge("Track");
26
27      for(HSNode rc: railCabs){
28          String goalStation = railCabStationMap.get(rc.getNodeID());
29          HSNode goalTrack = acc.getNode(goalStation).getSingleSourceOfEdgeLabel("part_of");
30          int i = rc.getSingleTargetOfEdgeLabel("on").canReach(goalTrack, tracks, true);
31          if( i > 0) // if i > 0 then goal is reachable
32              distance += i;
33      }
34      return distance * factor;
35  }

```

Listing 1: Implementation of the heuristic  $h_{NBP}^1$ , using the heuristics API of our planning framework.

uses an “off-the-shelf” SAT solver to find satisfying assignments for such a formula. By enumerating over different satisfying assignments, we receive different instances of an ALLOY model. These instances can be used as training problems for the SVM.

We use the UML class diagram, which describes the general structure of a planning problem, to automatically generate a *skeleton* of an ALLOY specification. This skeleton needs to be manually extended with constraints such that an ALLOY instance represents a meaningful planning problem. Examples for such constraints would be: “Each RailCab can reach its goal station using tracks” or “If a track is monitored by more than one Base Station, than its successor and ancestor tracks have different Base Stations”.

After generating a sufficient amount of ALLOY instances, the planning framework automatically translates each instance into a GTS planning problem. Then, for each problem, a feature vector is created, based on the features specified by the user. After solving the problems optimally (*e.g.* by using  $A^*$  with an admissible heuristic), each resulting cost value is stored together with its corresponding feature vector in a WEKA input file. Based on this input file, we finally learn the regression function and encode it – using the API – as a heuristic for the planning framework.

An example of two different feature sets which we used to learn heuristic functions are:

- $f_{Weka}^1$ : number of RailCabs; number of Tracks; number of stations in the goal rule; average distance to the goal station

- $f_{Weka}^2$ : number of RailCabs not at their goal position; average distance to the goal station; average branching factor

Trained on 140 different planning problems, the SVM learned the following heuristic functions:

$$h_{Weka}^1 = 5.4226*j_1 + 0.0769*j_2 - 0.8092*j_3 + 1.6148*j_4 - 2.5722$$

where  $j_1$  = number of RailCabs;  $j_2$  = number of tracks;  $j_3$  = number of stations in the goal rule; and  $j_4$  = average distance to the goal station.

$$h_{Weka}^2 = 6.3639 * k_1 + 1.9876 * k_2 - 1.2123 * k_3 - 4.2353$$

where  $k_1$  = number of RailCabs not at their goal position;  $k_2$  = average distance to the goal station; and  $k_3$  = average branching factor.

The entire process of generating the training data and learning the heuristic functions took about 35 minutes.

## Related Work

Edelkamp and Rensink (Edelkamp and Rensink 2007) described the similarities and differences between planning tools and graph transformation tools. They found that “graph transformation systems provide a flexible, intuitive input specification for systems of change with a sound mathematical basis”. A performance comparison of the graph transformation tool GROOVE (Rensink 2004) and the heuristic search planner FF (Hoffmann and Nebel 2001) demonstrated that planners can vastly outperform the graph transformation tool. However, the paper also describes why a GTS planning problem might not be suited for translation into PDDL: PDDL

does not support the creation/deletion of objects nor untyped domain objects.

Another paper by Edelkamp (Edelkamp, Jabbar, and Lafuente 2006) proposes several heuristic functions which can be used when performing heuristic search on GTS. While these heuristics – which can be encoded using our API – have the advantage of not being domain specific, they rely on the availability of a complete graph which defines the goal state. For the planning problem we are interested in, the goal states are typically defined using incomplete graphs which only state the properties of interest.

Röhs and Wehrheim (Röhs and Wehrheim 2010) have used GROOVE to solve planning problems using its built-in model checker. Their approach consists of the following steps: 1) modeling a planning problem using FUJABA, 2) translating the FUJABA graphs into GROOVE graphs, 3) using the GROOVE simulator to build a complete graph transition system, 4) finding a valid path from the start state to a goal state (valid means a path without forbidden states), 5) reporting the action names used along the path back to the user.

GROOVE’s model checker (MC) is used to search for a counterexample to the following statement: “*there exists no path to a goal node without any forbidden states along the way*”. If a counterexample can be found, it is a valid plan for the planning problem. The problem with the model checking approach is obvious: the generation of the entire state space (step 3) is very expensive and not necessary to solve a planning problem. We compare our heuristic search algorithms with the MC approach in the next section. Note that GROOVE allows to disable the exploration forbidden states using rule *priority*. We show the results of the MC approach with and without priorities.

## Evaluation

We evaluated the implementation of our planning framework on different planning problems. As a point of reference, we used the model checking based planner developed by Röhs (Röhs and Wehrheim 2010). Both planners use GROOVE to perform the graph transformations and build the graph transition system. Considering the fact that our current implementation requires additional bookkeeping due to implementation details, we focus on the number of states and transitions rather than the runtime. Our experiments were carried out on a quadcore machine with an “Intel Core i7 Q820” processor (3.06GHz core speed), 8 GB RAM, running Windows 7 Professional 64 bit, Java 1.6.0\_22 (32 bit) with a JVM heap size of 1.2 GB.

### N-Puzzle Problems

The first problem we used for the evaluation was the n-puzzle. Remember that *Best First* (BF) returns the first solution it finds, whereas *A\** returns an optimal solution (given an admissible heuristic).

One of the 8-puzzle problems, 8puzzle-06, is specifically modeled to be easily solvable. Only two slide

actions are necessary to reach the goal state. The purpose of this problem instance is to demonstrate the disadvantage of checking for a goal state only after the entire state space has been generated, as it is done with the model checking planner (MC). The results of the experiments are shown in table 1

We did not try to solve the 15-puzzle using the MC planner or our planner with the  $h_{Puz}^1$  heuristic. With a state space of 1.3 trillion states, the problem is currently not feasible. For the 8-puzzle, both our algorithms find solutions for all problem instances. We observed that the model checking planner was able to generate the entire state space (approx. 180.000 states) but ran out of memory while performing the search for a counterexample.

### NBP Problems

To evaluate the NBP case study, we used two different problems (NBP-B and NBP-M) and created twelve different problem instances. The instances vary in the number of tracks and the number of RailCabs used. The model checking planner was used a first time without any rule priorities and a second time with a priority of 1 for all rules which describe forbidden states. The heuristic  $h_{Empty}$  simply returns the value 0, *i.e.* the results show how the algorithms perform without any heuristic estimate. The heuristic  $h_{NBP}^1$  was modified to measure the distance-to-goal only (rather than costs) when used in combination with *Best-First*.

**NBP-B Problems** The first NBP problem models the situation that RailCabs have to move from Paderborn station to the stations Berlin and Leipzig, respectively. A single problem instance has the name “NBP-B-X-Y”, where *X* represents the number of RailCabs and *Y* represents the number of tracks. The results of the experiments are shown in table 2.

**NBP-M Problems** In an NBP-M problem, RailCabs have to move from Paderborn station to Berlin station and Munich station, respectively. The track network allows to travel to Berlin directly or by taking a detour over Munich. It is, however, not possible to travel to Munich over the Berlin route. One RailCab is carrying dangerous cargo, which prevents it from joining a convoy.

Our findings for these problem instances are quite similar to ones from the NBP-B problems. The BF and *A\** algorithms outperformed both model checking approaches. As we can see in table 2, the heuristic  $h_{NBP}^2$  performed very good for this particular problem, as it is likely for a RailCab to move along a route from which the goal station is unreachable.

**Learned Heuristics** Finally, we evaluated the learned heuristics  $h_{Weka}^1$  and  $h_{Weka}^2$  and compared them with the empty heuristic  $h_{Empty}$  and the manually written heuristics  $h_{NBP}^1$  and  $h_{NBP}^2$ . As the learned heuristics were used to estimate the costs, we only compared the results for the *A\** algorithm. Table 3 shows

	MC			BF with $h^1_{Puz}$			BF with $h^2_{Puz}$			A* with $h^1_{Puz}$			A* with $h^2_{Puz}$		
	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)
8puzzle-01	*	*	*	1453	2432	2	1688	2778	3	10207	17727	89	<b>662</b>	1031	<1
8puzzle-02	*	*	*	1389	2333	2	<b>251</b>	404	<1	82159	164840	6462	903	1505	25
8puzzle-03	*	*	*	<b>627</b>	1040	<1	955	1550	1	5926	10140	32	1032	1723	1
8puzzle-04	*	*	*	1086	1810	1	<b>117</b>	187	<1	27457	50070	649	2153	3631	4
8puzzle-05	*	*	*	1409	2344	1	<b>250</b>	401	<1	5311	9053	23	583	965	<1
8puzzle-06	*	*	*	<b>6</b>	8	<1	<b>6</b>	8	<1	<b>6</b>	8	<1	<b>6</b>	8	<1
15puzzle-01	~	~	~	~	~	~	<b>25967</b>	41225	336	~	~	~	%	%	%
15puzzle-02	~	~	~	~	~	~	<b>4997</b>	7717	22	~	~	~	%	%	%
15puzzle-03	~	~	~	~	~	~	<b>3561</b>	5468	13	~	~	~	%	%	%

\* out of memory exception

~ not evaluated

% premature termination after 4 hours

Table 1: Results for the n-puzzle problem. For each search strategy the number of states, transitions and solving time is shown. The minimal number of explored states is highlighted in bold for each problem instance.

	MC without Prio			MC with Prio			BF with $h_{Empty}$			BF with $h^1_{NBP}$			BF with $h^2_{NBP}$			A* with $h_{Empty}$			A* with $h^1_{NBP}$			A* with $h^2_{NBP}$		
	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)	#states	#trans	time (s)
NBP-B-2-08	293	837	1	253	665	<1	110	170	<1	<b>51</b>	74	<1	89	159	<1	268	642	<1	<b>88</b>	176	<1	204	457	<1
NBP-B-2-10	401	1011	1	315	797	<1	<b>77</b>	108	<1	126	237	<1	177	312	<1	358	852	<1	<b>117</b>	217	<1	233	517	<1
NBP-B-2-15	776	1966	2	515	1247	1	200	377	<1	<b>123</b>	216	<1	138	193	<1	564	1326	<1	<b>199</b>	392	<1	293	647	<1
NBP-B-2-20	1301	3131	3	765	1797	2	202	320	<1	<b>131</b>	218	<1	135	196	<1	817	1892	1	<b>78</b>	132	<1	352	777	<1
NBP-B-3-08	7615	34182	59	4647	15773	21	291	610	<1	550	1473	1	<b>271</b>	432	<1	5782	19009	29	<b>233</b>	514	<1	3502	10136	13
NBP-B-3-10	12426	53300	132	6381	21029	29	908	2437	2	<b>206</b>	446	<1	535	855	<1	8249	25780	50	<b>356</b>	864	<1	3891	11020	15
NBP-B-3-15	34956	134530	234	13713	45017	50	<b>234</b>	244	<1	1689	5137	5	1587	3167	5	16706	53554	201	<b>2344</b>	6935	43	5031	14520	24
NBP-B-3-20	78052	277431	837	24795	80813	104	4428	13228	46	1232	3584	5	<b>346</b>	485	<1	29068	94063	638	<b>693</b>	1751	5	6171	18020	36
NBP-M-3-08	9332	54647	61	5329	23670	24	685	2945	2	3594	17526	30	<b>263</b>	443	<1	5281	22346	26	2252	7691	8	<b>1788</b>	6840	6
NBP-M-3-10	14008	78031	71	7570	32439	26	2409	10365	16	4557	21681	42	<b>737</b>	1655	1	8041	36001	57	<b>2418</b>	9826	11	2541	9558	10
NBP-M-3-15	39584	194855	*	17667	65815	281	6810	24655	107	5595	20744	52	<b>751</b>	1277	1	19877	75207	274	8943	31727	151	<b>5087</b>	15969	27
NBP-M-3-20	48587	227263	*	23477	83978	95	15662	55782	500	10220	36913	106	<b>2048</b>	4149	8	27191	99387	485	10253	33918	140	<b>5678</b>	17674	35

\* Exception while executing Dijkstra's algorithm.

 Table 2: Results for the NBP-B and NBP-M problem instances. For each search strategy the number of states, transitions and solving time is shown. The minimal number of explored states is highlighted in bold for *Best-First* and A\*, respectively.

the number of states, transitions and the cost-value of the resulting plan.

We observe that  $h^2_{Weka}$  performed above the average. For 6 out of 12 problems it used the fewest number of states to find a solution. Also, for 6 out of 12 problems it used the fewest number of states while returning an optimal solution. It worked particularly well on problems of category NBP-M and was otherwise only outperformed by  $h^1_{NBP}$ , which – on average – had higher cost values.

It may seem surprising that  $h^1_{Weka}$  performed very similar to  $h^2_{NBP}$ . We explain this with the fact that the “average distance” calculation within  $h^1_{Weka}$  also returns a very high value in case a RailCab can no longer reach its goal. This equals the implementation of  $h^1_{NBP}$ . Taking this detail into account, we can conclude that  $h^1_{Weka}$  only performs as a cut-off heuristic and is otherwise as ineffective in estimating the costs as  $h^2_{NBP}$  (which estimates 0).

The findings of the experiments demonstrates that a learned heuristic should cover dynamic aspects of the problem, as it is done with  $h^2_{Weka}$ . If we learn functions based on static feature like “number of RailCabs”, the resulting function is of little use. Trained with the right features, however, a learned heuristic can be effective.

## Conclusion

In this paper, we presented a framework for heuristic search-based planning for graph transformation systems. The planning framework uses the GROOVE graph transformation tool to perform the necessary graph transformations. Planning problems are modeled in FUJABA, using a graphical notation called story patterns. We developed an API that enables users to easily write heuristic functions for GTS planning problems. Furthermore, we presented an approach to semi-automatically learn heuristic functions using Support Vector Machines and constraint-based problem instance generation. The efficiency and effectiveness of the heuristic search-based planner was compared to a previously proposed model checking based planner. The results indicate that our planner is superior with respect to the number of states that need to be explored in order to find a solution.

The observation that a heuristic search algorithm expands less states than a model checker, in order to find a plan, does not come as a surprise. At the beginning of our research, however, we were confronted with the situation of not being able to encode heuristic information that would improve the solving of a GTS planning problem. Having overcome the technical issues, we were surprised by the actual performance improvements us-

	$A^*$ with $h_{Empty}$			$A^*$ with $h^1_{NBP}$			$A^*$ with $h^2_{NBP}$			$A^*$ with $h^1_{WEKA}$			$A^*$ with $h^2_{WEKA}$		
	#states	#trans	Costs	#states	#trans	Costs	#states	#trans	Costs	#states	#trans	Costs	#states	#trans	Costs
<i>NBP-B-2-08</i>	268	642	17	88	176	21	204	457	17	185	404	17	<b>74</b>	114	21
<i>NBP-B-2-10</i>	358	852	23	117	217	23	233	517	23	226	488	23	<u><b>92</b></u>	135	23
<i>NBP-B-2-15</i>	564	1326	38	<u><b>199</b></u>	392	38	293	647	38	288	628	38	255	544	38
<i>NBP-B-2-20</i>	817	1892	53	<u><b>78</b></u>	132	53	352	777	53	348	758	53	315	674	53
<i>NBP-B-3-08</i>	5782	19009	26	<b>233</b>	514	31	3502	10136	26	3500	10210	26	604	1210	27
<i>NBP-B-3-10</i>	8249	25780	32	<b>356</b>	864	33	<b>3891</b>	11020	32	<b>3891</b>	11020	32	2660	6762	33
<i>NBP-B-3-15</i>	16706	53554	47	<b>2344</b>	6935	48	5031	14520	47	5032	14520	47	<u><b>4462</b></u>	12308	47
<i>NBP-B-3-20</i>	29068	94063	62	<b>693</b>	1751	63	6171	18020	62	6171	18020	62	<u><b>5602</b></u>	15808	62
<i>NBP-M-3-08</i>	5281	22346	19	2252	7691	21	1788	6840	19	1749	6584	19	<b>965</b>	3469	21
<i>NBP-M-3-10</i>	8041	36001	27	2418	9826	27	2541	9558	27	2583	9520	27	<u><b>1776</b></u>	6004	27
<i>NBP-M-3-15</i>	19877	75207	40	8943	31727	45	5087	15969	40	4978	15532	40	<u><b>4044</b></u>	12274	40
<i>NBP-M-3-20</i>	27191	99387	51	10253	33918	51	5678	17674	51	5672	17660	51	<u><b>5443</b></u>	16836	51

Table 3: Results for the learned heuristics. The third column of each search strategy shows the costs of the solution.  $A^*$  with  $h_{Empty}$  is optimal. The fewest number of states is highlighted in bold. The fewest number of states while being optimal is highlighted using underlining.

ing even simple heuristics as the ones described above. Furthermore, we found that the encoding of a planning problem using a GTS not only gives the advantage of having a graphical notation. It also aids the development of heuristics as one mostly thinks about the planning problem in terms simple properties such as nodes, neighbor-nodes or reachability between nodes.

As future work, we plan to analyze additional planning problems to further test our heuristics API. Once the API has reached a stable state, one can investigate how to write heuristics using the same graphical notations that are used to model the planning problem. Furthermore, we would like to explore potential performance benefits of using other graph transformation tools and changing our framework to account for multi-core architectures.

## References

- Boser, B. E.; Guyon, I. M.; and Vapnik, V. N. 1992. A Training Algorithm for Optimal Margin Classifiers. In *COLT '92: Proceedings of the fifth annual workshop on Computational learning theory*, 144–152. New York, NY, USA: ACM.
- Edelkamp, S., and Rensink, A. 2007. Graph Transformation and AI Planning. In Edelkamp, S., and Frank, J., eds., *Knowledge Engineering Competition (ICKEPS)*. Canberra, Australia: Australian National University.
- Edelkamp, S.; Jabbar, S.; and Lafuente, A. 2006. Heuristic Search for the Analysis of Graph Transition Systems. In Corradini, A.; Ehrig, H.; Montanari, U.; Ribeiro, L.; and Rozenberg, G., eds., *Graph Transformations*, volume 4178 of *Lecture Notes in Computer Science*, 414–429. Springer.
- Fischer, T.; Niere, J.; Torunski, L.; and Zündorf, A. 2000. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, 296–309. London, UK: Springer.
- Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilins, D. 1998. PDDL - the Planning Domain Definition Language, version 1.2. Cvc tr-98-003/dcs tr-1165, Yale Center for Computational Vision and Control.
- Hall, M.; Frank, E.; Holmes, G.; Pfahringer, B.; Reutemann, P.; and Witten, I. H. 2009. The WEKA data mining software: an update. *SIGKDD Explorations* 11(1):10–18.
- Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.
- Jackson, D. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Transactions on Software Engineering and Methodology* 11(2):256–290.
- Jackson, D. 2006. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.
- OMG. 2010. OMG Unified Modeling Language (OMG UML) Infrastructure Version 2.3. Technical Report formal/2010-05-03.
- Rensink, A. 2004. The GROOVE Simulator: A Tool for State Space Generation. In Pfalz, J.; Nagl, M.; and Böhlen, B., eds., *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, 479–485. Springer.
- Röhs, M. 2009. Sichere Konfigurationsplanung adaptiver Systeme durch Model Checking. Master's thesis, Universität Paderborn.
- Röhs, M., and Wehrheim, H. 2010. Sichere Konfigurationsplanung selbst-adaptierender Systeme durch Model Checking. In Gausemeier, J.; Rammig, F.; Schäfer, W.; and Trächtler, A., eds., *Entwurf mechatronischer Systeme*, volume 272, 253–265.
- Vaquero, T.; Tonidandel, F.; and Silva, J. 2005. The itSIMPLE tool for Modeling Planning Domains. In *Proceedings of the First International Competition on Knowledge Engineering for AI Planning and Scheduling (ICKEPS)*.

# **Poster Presentations**

## JPDL: A fresh approach to planning domain modeling

**Michael Jonas**

*The IMPACT Laboratory*  
Arizona State University  
Tempe, AZ 85287, USA

### Abstract

In our recent work developing autonomous agents we attempted to apply off the shelf technologies such as PDDL and a variety of planners from the planning competitions to our domain to serve as a baseline for evaluation of our domain specific solutions. Unexpectedly, our efforts were met with resistance by the team we were integrating this work with once they saw the format of academic solutions. Furthermore, as the project evolved we became increasingly frustrated by the infeasibility of developing sufficiently large domains. In response to the feedback of our partner developers outside of the planning communities we developed a new approach to domain modeling that is compatible with the trends of the software development world. This paper introduces this approach, JPDL (Java Planning Description Language), which takes into account these lessons and the practical requirements of the developers of real autonomous systems.

### Introduction

Standardized domain representation languages have done great things for knowledge representation (KR) and the planning and scheduling (PS) communities hereafter referred to as the community. Languages like PDDL have driven the community forward by giving a standard semantic meaning to problems and have allowed planners to compete solely on their own merit by removing individual programmer skill from planner evaluation (McDermott et al. 1998; Fox and Long 2003). Over time the complexity of these languages has grown, their expressiveness has increased and they have on occasion diverged to create new languages such as SHOP and GoLog (Nau et al. 1999; Levesque et al. 1997). Each new language has focused on some new aspect of planning such as HTN. What we feel is lacking from the community is a domain modeling language with a strong focus on usability that would be attractive to developers.

In response to this we have developed a new domain representation language and created a proof of concept implementation of this language in Java alongside a domain specific planner that uses it. This language, JPDL (Java Planning Description Language), uses a subset of the existing Java syntax and is therefore Java compilable. The semantic meaning is precisely defined and checkable via the JVM.

Copyright © 2011, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Because JPDL is Java compilable, it is also Java executable, and the JVM can be used to do plan validation in a precise unambiguous way. We feel this provides a modeling language that benefits from advances in language usability in recent decades.

Our motivation for JPDL came from our personal experience developing cognitive architectures for embodied systems. Our experience has shown us how impractical it can be to convince a company to train its developers on an entirely new language, especially once they see the form of languages such as PDDL. The software development world has made great, valuable strides in language usability in recent decades. Trends have arisen and certain languages have gained popularity for their usability. We find that many of these trends have received little to no focus in the family of domain modeling languages and we find this to be to the detriment of both academia and industry.

The first such trend is the rise of development environments such as Eclipse and Visual Studio. Many developers prefer IDE's because of the benefits they provide such as real time feedback in code correctness. Languages that are so entirely different from what IDE's today support lose access to these benefits. Additionally, developers lose access to the rich feedback compilers provide for well supported languages for syntactic errors. From our experience using planners from the planning competition like FF, LAMA, and SGPlan we have found that the feedback provided by existing planners on code errors to be more sparse and less informative than those provided by Visual Studio or Javac (Hoffmann 2001; Richter and Westphal 2008; Hsu and Wah 2008). Losing access to these benefits hinders the usability of academic domain representation languages.

Another trend is the decline of Lisp-like syntax. Languages like Java and C++ have been the mainstay for long enough that we should consider providing at least one language of similar form to developers who prefer these families. One reason for this change of popularity may be that due to the syntax differences, a compiler is fundamentally more able to diagnose certain syntactic errors in Java or C++ than Lisp. Forcing developers to develop using Lisp-like syntaxes hinders the usability of academic domain representation languages.

One last trend is the advancement of newer programming paradigms such as object-oriented code. We feel this indi-



cates at the least a preference of many programmers. Ignoring such a trend further alienates those programmers, yet again hindering the usability of academic domain representation languages.

The repetition above is pointed in that reinforces our motivation. The main thrust of this paper is that the usability of academic domain representation languages is hindered by their current form which has ignored beneficial trends of the software development world and as a result has made the solutions provided unnecessarily unattractive to those it was intended to benefit. Our experiences attempting to apply academic solutions to real large scale systems has proved to us how unattractive these can be to those outside our community and how completely infeasible representing certain domains can be if even possible at all. JPD L is our response to these frustrations. It is our attempt at merging the gap between what is desired by developers and what is provided by academia.

The rest of this paper focuses on the JPD L language specifics. Section 2 formally introduces JPD L terms. Section 3 shows an example Blocksworld domain and a straightforward representation in JPD L. Section 4 discusses the limitations of this work and future work potential.

symbol	meaning
$S_L$	State List
$E_L$	pending Effect list
$C_L$	constraint list
$s$	State
$V$	State Variable set
$P$	primitive set
$t$	time
$v$	State Variable
$i$	identity of a State Variable
$I_v$	Identity set on a State Variable
$P_v$	Primitive set on a State Variable
$f$	function
$E$	Effect set

## Preliminaries

First we provide a brief summary of the components of JPD L. Then we expand upon these definitions in each following subsections in the order they are introduced.

The primary function of JPD L is to describe a predictive model of the world that can be used for planning and meta-planning. The core structure of JPD L is the **State List** which holds time continuous description of all recognized facts. The State List encodes time dependent knowledge about the world into **States**. A domain programmer creates a domain specific State definition called a **World Model** which contains all of the **State Variables** specific to that domain. A plan is represented by a time stamped list of **Effects**, which describe instantaneous changes to the State List. To perform plan validation all one must do is build an initial State List, apply the pending Effect list in time order, and check whether the resulting timeline is consistent and meets the goal criteria. To the domain programmer all of the above structures form a library that is as precise and is

easier to learn and debug than other existing world modeling languages.

## State List Structure

Informally, a State List is a timeline of features of the world. A State List consists of a time ordered list of States that are strung along continuously in time from a starting time to an eventual time point.

Formally these features of the world are encoded into a set of State Variables  $V$  which individually consist of an unchanging identity  $i$ , an ordered set of identifies of other State Variables  $I_v$ , and an ordered set of primitive values  $P_v$ . A State  $s$  consists of an ordered set of State Variables  $V$  an ordered set of primitives  $P$ , and two time bounds  $t_l$  and  $t_r$  which define the period of time for which this mapping of  $I_v$  and  $P_v$  on  $V$  holds. A State List consists of a time ordered set of States  $S$  such that time is defined continuously from some initial time  $t_i$ . When the values of  $I_v$  and  $P_v$  on a State are subject to no further changes that State is said to be finalized. After the finalized States is the current State, a single State at the tail of the State List whose values are still being defined.

## Effects, Actions, and Constraints

Effects are the mechanism by which the State List is updated. A primitive Effect describes some instantaneous change to a State List. When these changes are made the Effect is said to have been applied. Formally, a primitive Effect is an operator  $S'_L = E(f_e, I_e, P_e, t, S_L, E_L, C_L)$ . The function  $f_e$  uses the State List  $S_L$  with States finalized up to immediately before time  $t$  to create a new State List  $S'_L$  by changing  $I_v$  and  $P_v$  for the State at time  $t$ . This may require creating a new State so that no values of  $I_v$  and  $P_v$  are changed at any time before  $t$ . Using the identities from  $I$  the function  $f_e$  can usefully access  $V$  on all states from  $t_i$  to  $t$  thereby allowing non-Markovian descriptions that utilize State Variable values on previous States.

In contrast to primitive Effects a complex Effect instantiates other Effects and adds them to the pending Effect list,  $E_L$ . When the Effect being instantiated is for any time after  $t$  the action is said to be temporal. This immediately facilitates HTN planning by allowing each Effect to form a tree with child Effect nodes. An action is a more abstract term referring to all Effects included in a tree. A plan can be described minimally by the set of root nodes of actions.

Effects can in principle be conflicting if written improperly. This is the case whenever the State List  $S'_L$  is different depending on the order of application of Effects. Well written Effects will either self govern their results depending on the order of application, or they will use **constraints**. A constraint  $C$  is an operator  $T/F = C(f_c, I, t_l, t_r, S_L)$  that determines if a State List is consistent as states are finalized between states  $t_l$  and  $t_r$ . A State List is found to be consistent with respect to a constraint if the values  $I_v$  and  $P_v$  on State Variables with identifiers  $I$  on all States from time  $t_l$  to  $t_r$  match some pattern determined by  $f_c$ . If a State List is consistent with respect to every constraint up to some time the State List is consistent up to that time. Effects that can conflict should as part of their application add constraints to

the constraint list to validate that the results of their application were in fact maintained on the finalized State List. If a constraint returns false the State List is said to be inconsistent starting with the first State that a constraint fails to match this pattern to.

In contrast to Effects, constraints are executed on finalized states. This implies that they should make no changes to the State List. Because of this their order of execution should be irrelevant. This characteristic allows them to resolve any Effect conflicts mentioned above. However, being unable to modify States does not prevent constraints from adding more constraints to the constraint list as part of their execution. This allows constraints to be a fairly powerful and robust means of consistency checking over continuous spans of time, at intervals through time, and even with intervals and patterns dependent on the values of State Variables encountered specific to a plans result. Effects and constraints are used both for domain description and problem descriptions as covered next.

## Problem Description

Describing a planning problem requires three things:

1. A description of the world in its initial State.
2. A description how the world will change over time dependent or independent of agent choices.
3. A description of whether a solution is satisfactory.

Each of these criteria is satisfied by an initialization operator responsible for creating  $S_L$ ,  $E_L$ , and  $C_L$ . Formally, a problem description consists of a State List initialization operator  $S_L = I_s(f_i)$ , a pending Effect list initialization operator  $E_L = I_e(f_e)$ , and a constraint list initialization operator  $C_L = I_c(f_c)$ . These together with the State and Effect descriptions form a complete planning domain and problem.

## Example Domain

Above we provided a syntax independent functional description of JPDL. We have created a library implementation of JPDL in Java and in this section we show specific syntactic examples of a well established AI domain converted to JPDL syntax as well as provide an algorithm for plan validation. As was motivated in the introduction, JPDL uses a subset of Java syntax and is therefore compilable and the JVM can be used to check correct semantic interpretation.

Inferring from above, to provide a complete JPDL description this section needs to contain a structure for State Variables important to this domain, an organization of those State Variables onto a State, the set of known Effects that can modify the State, and initialization functions to setup the State List, pending Effect list, and constraint list.

## State Variables and States

For demonstration purposes we translate the Blocksworld domain popular in the planning competition and ever since STRIPS (Fikes and Nilsson 1971). Because of its age and persistence there has been much scrutiny of this domain and at this point many different representations exist (Slaney and

Thiebaux 2001). For brevity and clarity we are using a representation with 1 action which requires conditional effects. In PDDL the predicates consist of:

```
(:predicates
  (on ?x ?y)
  (clear ?x)
  (block ?b)
)
```

To represent the equivalent in JPDL we need to write a Block class.

```
public class Block extends StateVariable
{
    Block on;
    boolean clear;
}
```

Note that the Block class must extend the StateVariable class for it to be properly utilized. This State Variable class has an id field that serves to provide a State Variable with a persistent identity across States as well as a constructor to initialize a unique value for this field. Unless an identifier is manually specified in the constructor our implementation assigns a value for the id field that is the class name of the State Variable followed by a static number that counts the total number of State Variables instantiated this way so far. This gives us reasonably intuitive id's for debugging such as Block1 and the manually specified id Table.

Now that our State Variables are defined we must organize them onto a State to make them accessible to both an initialization function and non-Markovian actions as described in the previous section. The name of this class is arbitrary but customarily we named this class WorldModel. The only requirement of this class is that it has at least one StateVariable on it and that it extends the State class.

```
public class WorldModel extends State
{
    Block[] blocks;
}
```

To demonstrate how everything so far is used we introduce the State List initialization function of the problem. There are three initialization functions which are responsible for initializing the State List, Pending Effect List, and Constraint List. The initialization function below is the first of these on the problem class being defined. Fig 1 shows a pictorial representation of the results of this initialization.

```
public class Example extends Problem
{
    public static void initStateList()
    {
        State initState = new WorldModel();
        stateList.add(initState);

        initState.blocks = new Block[4];

        Block table = new Block("table");
        initState.blocks[0] = table;
        Block block1 = new Block();
```

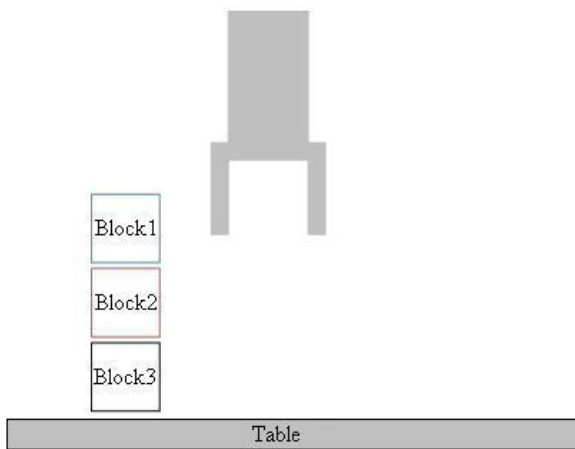


Figure 1: A pictorial representation of the results of our BlocksWorld initialization function.

```

initState.blocks[1] = block1;
Block block2 = new Block();
initState.blocks[2] = block2;
Block block3 = new Block();
initState.blocks[3] = block3;

block3.on=table;
block2.on=block3;
block1.on=block2;
block1.clear=true;
}
}

```

Note that problem definition must extend the Problem class and overload its init methods. This class contains fields for stateList, pendingEffectList, and constraintList and its constructor instantiates each. The add method associated with these lists is a reserved word associated with all three fields used primarily for initialization. The add function is equivalent to Java's List interface semantically.

All State Variables that can appear as member variables in Effects should also be in a shallow list on each State. This is important for State Variables on Effects to be properly updated prior to application to reflect State Variables on the current State.

### Effects and the Pending Effect list

As mentioned above the Blocksworld sample domain only has one action. This action moves a block from being on top of one block to the table or another block. There are three parameters in its header. In order they are the block to be moved, the block it was moving to (or the table), and the block it is moving from (or the table).

```

(:action puton
:parameters (?X ?Y ?Z)
:precondition (and (on ?X ?Z)
                  (clear ?X)
                  (clear ?Y)
                  (not (= ?Y ?Z)))

```

```

                  (not (= ?X ?Z))
                  (not (= ?X ?Y))
                  (not (= ?X Table)))
:effect (and (on ?X ?Y) (not (on ?X ?Z))
            (when (not (= ?Z Table))
                  (clear ?Z))
            (when (not (= ?Y Table))
                  (not (clear ?Y)))))

```

What follows is a fairly straightforward translation.

```

public class PutOn extends Effect
{
    Block x,y,z;

    public void apply(StateList sl)
    {
        if(x.on==z&&x.clear&&y.clear&&
            y!=z&&x!=z&&x!=y&&x.id!="table")
        {
            x.on=y;
            if(z.id!="table")
                z.clear=true;
            if(y.id!="table")
                y.clear=false;
        }
    }
}

```

When the apply method is called the State List is assumed to be finalized up to just before the Effect time. This implies that the last State in the State List is the State currently being defined, a.k.a. the current State. Prior to this method call any State Variables of the Effect will be *synchronized* to the current State using their id field. This is accomplished via a pre-apply method that is called in plan validation prior to each Effect. This method uses the id field of each State Variable on the Effect to change the pointer of that State Variable to the State Variable with the same id on the current State. This removes the need for programmers to search the current State and fetch those variables manually and more generally this allows us to have State Variable fields rather than String id fields on the Effect.

In order to do plan validation the desired plan must be built as Effects and added to the pending Effect list. Our libraries contain a parser that reads a file containing the plan to be validated. Rather than get into the parsing and file format details we simply build the plan directly in the init method here for brevity. What this plan will do is to unstack the blocks so they are all on the table.

```

public class Example extends Problem
{
    ...
    public static void initPendingEffectList()
    {
        Puton unstack1 = new Puton(0);
        unstack1.x = new Block("Block1");
        unstack1.y = new Block("Table");
        unstack1.z = new Block("Block2");
        pendingEffectList.add(unstack1);

        Puton unstack2 = new Puton(1);
        unstack2.x = new Block("Block2");
    }
}

```

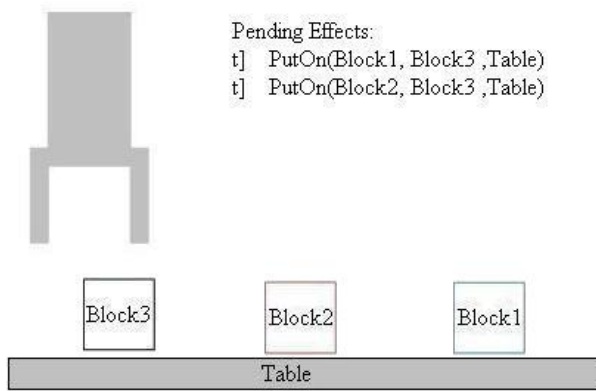


Figure 2: A setup for an inconsistent world. Whichever Effect is applied first would prevail, resulting in the block it is moving being stacked on Block3. Since these are supposed to be simultaneous this results in an inconsistent State List. This should be detected by properly written Effects and constraints.

```

unstack2.y = new Block("Table");
unstack2.z = new Block("Block3");
pendingEffectList.add(unstack2);
}
}

```

The single parameter in the Effect constructor is the time of the Effect. This can be set using the *t* field during constructors as well. After initialization the Effect list is sorted in ascending order by time.

### Goal Constraints

All that remains is to describe patterns that need to match to check whether a State List is consistent and whether the goals have been met. Even in this simple domain inconsistency is possible. Consider the case shown in Figure 2 where every block is on the table and there are two simultaneous pending PutOn Effects as indicated. The resulting State List is dependant on the application order of the pending Effects. What should the result be? Our answer is that neither result should dominate and that in fact the plan produces an inconsistent world and so either the Effect descriptions or the plan needs reworking.

This problem can be fixed in a number of ways using constraints. We will show the most general solution here. We can detect if any condition of applying the Effect was not met when it was applied and set a catch-all error variable on the State when this happens. In this case we will need a constraint to detect this variable and if it is true declare the State List to be inconsistent.

```

From class PutOn:
...
public void apply(StateList sl)
{
    if(...)
        ...
}

```

```

else sl.getCurrentState().error=true;
}

public class ErrorCheck extends Constraint
{
    public boolean check(StateList sl)
    {
        return !sl.getCurrentState().error;
    }
}

```

What makes this solution so general and attractive is its scalability to additional Effects in the domain description. Also, this provides precision to the domain programmer to describe when the world is consistent but the action has done nothing versus when the action has created an inconsistent world by its inapplicability to a State List.

The last missing piece is the constraint list initialization method. This is very similar to the initialization method for Effects.

```

public class Example extends Problem
{
    ...
    public static void initConstraintList()
    {
        FlagCheck c1 = new FlagCheck(ti, te);
        constraintList.add(c1);
        Goal c2 = new Goal(te, te);
        constraintList.add(c2);
    }
}

```

The code above introduced two new things. First there are reserved symbols on the problem class to hold values for the initial and eventual time, *ti* and *te* respectively. Default values for these variables are 0 for the initial time and *Double.MAX\_VALUE* for eventual time. These variables are used in the ErrorCheck constructor as the left and right hand closed bounds of the constraint. During plan validation the check method of the ErrorCheck constraint we have instantiated will be executed on the State List each time a new State is finalized. The second constraint we have instantiated is a typical goal. When the left bound of a constraint is the eventual time it is checked once the pending Effect list is empty. In practice in our implementation they are also checked after the State List has been finalized to a reasonable time to handle a domain where the pending Effect list is perpetuating itself. The goal constraint we instantiated is shown below.

```

public class Goal extends Constraint
{
    public boolean check(StateList sl)
    {
        State current = sl.getCurrentState();
        for(Block block : current.blocks)
        {
            if(block.on.id!="Table")
                return false;
        }
        return true;
    }
}

```

```

Init( $S_L, E_L, C_L$ )
 $t_c \leftarrow t_i$ 
 $s_c \leftarrow S_L(t_i)$ 
while  $E_L.hasNext()$  &  $t_c < e.t$  do
   $e \leftarrow E_L.getNext()$ 
  if  $t_c \neq e.t$  then
    finalize( $s_c$ )
    if !checkConstraints( $s_c$ ) then
      return false
    end if
     $S_L.split(t_c)$ 
     $t_c \leftarrow e.t$ 
     $s_c \leftarrow S_L(t_c)$ 
  end if
   $e.preApply(s_c)$ 
   $e.apply(S_L)$ 
end while
if checkConstraints( $S_L(t_e)$ ) then
  return true
else
  return false
end if

```

Figure 3: Plan validation algorithm pseudocode.

This continues our ongoing problem description that coincides with the plan created on the pending Effect list. For each loop are the approximate equivalent to the for-each and for-all operators in PDDL. The above constraint returns true if and only if all blocks that are not the table itself are on the table. Because the plan we added to the pending Effect list unstacks the blocks for the world we constructed on the initial State List, this constraint should be satisfied if plan validation is working properly. The next section shows how this is accomplished.

### Plan Validation

In order to demonstrate a consistent semantic meaning to JPDL we now walkthrough the plan validation steps of our ongoing Blocksworld Example. The JPDL libraries we provide are written in Java which allows us to use the JVM to enforce semantic meaning, but the definitions above could be interpreted successfully by a planner in any language so long as the same syntax is used and semantic meaning is enforced.

The plan validation method accepts as a parameter the name of a problem class. The algorithm returns true if all constraints are met on each finalized state or false otherwise. This class as well as the State and Effect descriptions should be in the classpath so they can be found using reflectance. The pseudocode for our plan validation algorithm is shown in Figure 3.

The algorithm in Figure 3 introduces several unique symbols.  $t_c$  refers to the current time,  $s_c$  the current State,  $e.t$  the time of effect  $e$ , and a handful of methods referred to in our example domain section. Two methods we wish to specifically address are the split and pre-apply methods. In the former case a time is specified and the State at that time is

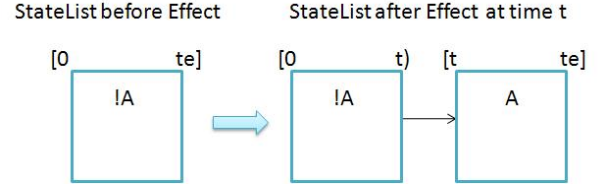


Figure 4: A primitive State List and the result of applying an Effect that requires State splitting.

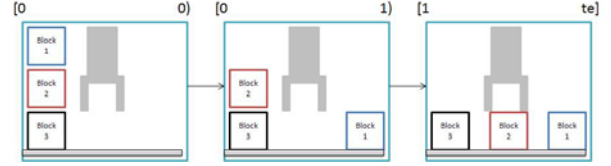


Figure 5: The State List of our ongoing Blocksworld example at the end of plan validation.

deep cloned. Following this the time bounds on both copies are set such that there is a continuous timeline with a new State starting at  $t$ .

Figure 4 demonstrates this concept by introducing a visualization of a State List and an example Effect modifying it. In this figure and Figure 5 each rectangle represents a single State. The span of time for which each State is valid is in the upper left and upper right corner just outside the rectangle with time progressing from left to right. A [ or ] indicates an inclusive bound, a ( or ) indicates an exclusive bound. Inside each rectangle is information about the State Variables we wish to show, either in list form or a pictorial representation. As shown in Figure 4, when an Effect is executed, if there is not already a State with starting time ( $e.t$  the State List is split before the Effect is applied.

Once the State to which an Effect will be applied is in hand pre-apply is called to synchronize that Effect with that State. The pre-apply method maps each State Variable field of the Effect to point to the State Variable on the State with a lexicographically matching id value if such a State Variable exists. This significantly reduces programmer burden by eliminating the need to match these variables manually inside each apply method. If State Variables are desired from previous states they can be fetched using the synchronize method. This method accepts a State and a State Variable with an id value to search for. It returns true if a match was found and updates the State Variable as a side effect. The rest of the pseudocode should be simple to interpret.

Continuing with our Blocksworld example, after initialization the State List consists of a single State previously pictured in Figure 1. Additionally the pending Effect list has been initialized such that the blocks should be unstacked when all pending Effects are resolved and the constraint list has been initialized to check that the blocks are unstacked at the eventual time and that no inconsistent Effects were applied. Figure 5 shows the resulting State List when the plan validation algorithm terminates.

## Limitations and Future Work

JPDL was developed due to necessity when existing approaches failed in our work. As the start of a new domain modeling language we feel we have presented a solid foundation that is satisfactory for many domains. This foundation was sufficient to overcome the problems we encountered and we felt that the lessons learned could be of benefit to the community. However it is hardly without its limitations and we recognize there is plenty of room for expansion even to catch up to all the capabilities of existing languages. We felt it prudent to recognize these limitations in the applicability of this work at present and close by noting areas that we may expand on in future work.

First, JPDL does not support partially defined states. All primitives must have a value, and missing State Variables are assumed not to exist in a typical closed-world assumption. This makes it unsuitable for conformant planning. For similar reasons there is no support for stochastic or non-deterministic actions in the library at present. These are all features supported by languages such as PDDL and are the subject of entire planning competition tracks (Younes and Littman 2004).

Effects in their present form only describe instantaneous changes. Many domains have within them continuously changing values. Language extensions such as PDDL+ were created to support these domains (Fox and Long). We have only created prototype solutions for continuous changes and more work is required before this is added to the JPDL specification.

Existing research has shown the value of concepts such as derived predicates (Thiebaux, Hoffmann, and Nebel 2005). Entire architectures have been focused on using hierarchical concept and breakdowns for planning (Langley et al. 1991; Albus and Shoemaker 2002). We would like to add some form of derived predicate to the JPDL specification. Doing so has several benefits including adding scalability to the language by reducing repetitive code in Effects, removing constraints otherwise necessary to ensure consistency, and increasing the functional independence of developers by increasing code modularity.

The translation example shown in this paper is so straightforward one is naturally led to wonder whether it is possible to translate all of JPDL to an existing format such as PDDL. If such a translation was possible domain writers could use JPDL to decrease domain programming burden and then translate it to PDDL and use a variety of planners developed for the planning competitions. There are several very detail oriented problems with such a translation. Among them are how to support order-based notions such as sequencing deletes and over-all constraints before adds from arbitrary Java style code, supporting PDDL mutex locks on Java primitive types, and translating numeric function calls such as taking a square root. Nevertheless, our preliminary work has shown that a subset of JPDL is weakly translatable. This leads to interesting conclusions as to what parts of JPDL are infeasible to translate and why; even more so when encountering fundamentally untranslatable JPDL elements. Analysis needs to be done into which of these elements are fundamentally problematic for general planners to use and

therefore should be excluded from JPDL or alternatively if they are not fundamentally problematic what additional expressiveness and usability supporting them adds to a domain modeling language.

It is worth noting that translating from PDDL to JPDL has a different host of problems as well. Some of these problems are shared such as mutex locks, and some of them are new such as representing multi-key multi-value associations accurately. The latter would most likely require an external library such as guava's Multimap. There is a lot more to say about computational costs of PDDL syntax and data structures as well as the strengths and weaknesses of using a declarative versus procedural language. Our preliminary work on the topic is encouraging.

As was mentioned in the introduction we developed a domain specific planner alongside JPDL. This planner implemented an A\* search approach with backtracking and eventual brute force search for completeness. While not particularly sophisticated, it is worth noting that within 2 hours of effort our built from scratch planner was yielding satisfactory solutions. It is our opinion that most project developers would be willing to spend a similar effort coding domain specific heuristics if it would lead to a performance increase over domain independent planners.

Our planner did not parse and understand JPDL syntax so much as execute it and reason about our specific domain. In the future we would like to create a domain independent planner that understands JPDL syntax and incorporates existing heuristics. Ideally the heuristic methods used should be overloadable by domain programmers allowing for the rapid development of other domain specific planners with a fallback of the domain independent planner.

Finally, we intend to make the Java JPDL library implementation and the plan validation code available under the GNU public license.

## Acknowledgements

We'd like to acknowledge J. Benton for his support and feedback in the formation of this paper and our industry partners without whom this work would not have happened.

## References

- Albus, J., and Shoemaker, C. 2002. 4d/rcs: A reference model architecture for unmanned vehicle systems version 2.0. <http://www.nist.gov/el/isd/rcs.cfm>.
- Fikes, R., and Nilsson, N. 1971. Strips: A new approach to the application of theorem proving to problem solving. In *IJCAI-71*.
- Fox, M., and Long, D. Pddl+: Modelling continuous time-dependent effects.
- Fox, M., and Long, D. 2003. Pddl 2.1 : An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20.
- Hoffmann, J. 2001. Ff: The fast-forward planning system. In *AAAI*.
- Hsu, C.-W., and Wah, B. 2008. The sgplan planning system in ipc-6. In *Proceedings of IPC*.

- Langley, P.; McKusick, K.; Allen, J.; Iba, W.; and Thompson, K. 1991. The icarus cognitive architecture. *ACM Sigart Bulletin*.
- Levesque, H. J.; Reiter, R.; Lesperance, Y.; Lin, F.; and Scherl, R. B. 1997. Golog: A logic programming language for dynamic domains. *Elsevier*.
- 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 *AIPS1998*.
- Nau, D.; Cao, Y.; Lotem, A.; and Munoz-Avila, H. 1999. Shop: Simple hierarchical ordered planner. In *IJCAI-99*.
- Richter, S., and Westphal, M. 2008. The lama planner using landmark counting in heuristic search. In *Proceedings of IPC*.
- Slaney, J., and Thiebaux, S. 2001. Blocks world revisited. *Elsevier*.
- Thiebaux, S.; Hoffmann, J.; and Nebel, B. 2005. In defense of pddl axioms. *Elsevier*.
- Younes, and Littman. 2004. Ppddl 1.0: The language for the probabilistic part of ipc-4. In *International Planning Competition*.

# Cooperated Integration Framework of Production Planning and Scheduling based on Order Life-cycle Management

**Shigeru Fujimura**

WASEDA University

2-7 Hibikino, Wakamatsu-Ku, Kitakyushu, Fukuoka, 808-0135, Japan

[fujimura@waseda.jp](mailto:fujimura@waseda.jp)

<http://www.fujimura-lab.org/>

## Abstract

Order Life-cycle Management (OLM) is to manage information changing in the life cycle of many kinds of orders utilized in manufacturing companies. It improves the smart and prompt decision-making in dynamically changing situations. In addition, customer satisfaction can be obtained with an exact information presentation to customers. In this paper, the novel approach integrating many kinds of production planning and scheduling modules with an OLM system to realize the high performance Supply Chain Management (SCM) is proposed. The database for an OLM system according to Order Transition Model utilizes past records of order information with influence relations between orders. Many kinds of information are extracted from it and transferred to a production planning and scheduling module as a processing module in a multi-stage scheduling system. In this paper, a design of the cooperated integration framework of production planning and scheduling based on OLM is described.

## Introduction

An Order Life-Cycle Management (OLM) system manages several states of orders from creation to termination in the order life cycle through many kinds of changes in phases as receiving customer orders, production, inspection, delivery and so on. The purpose to use an OLM system is to visualize the information to promote quick decision making depending on dynamically changing situations. Many vendors of Enterprise Resource Planning (ERP) systems realize OLM features as one of original parts in their own systems. However, OLM features provided by these vendors are a sort of mechanism that is oriented towards state changes of orders after receiving from customers (hereinafter referred to as narrow sense OLM: nOLM). Management of order information that is obtained before receiving from customers is very important and effective for production planning and scheduling, however it is not applicable for these systems.

On the other hand, the purpose of Supply Chain Management (SCM) systems is to minimize inventory

fluctuation at each element in a supply chain by achieving the smooth information flow between elements. Using it, the infrastructure for information communication is covered widely; however, it cannot improve the performance of a supply chain, because it also uses no forecasted or intangible tentative order before receiving exact orders from customers. It contains information, such as some uncertainty of amount, detailed specification, and/or other data. By using tentative orders, decision making of worker capability, order control, high precise available-to-promise, and lead time reduction by early procurement of raw materials will be enabled.

To realize such SCM, it is very important not only using tentative orders but also integrating production planning and scheduling. In production planning for the long-term strategic and middle-term tactics levels, order information is generally utilized as aggregated information, and in production scheduling for the short-term operational level, the aggregated information is mapped to the separated detailed information and it is utilized. Furthermore, in production scheduling, there are many cases in which the partial aggregation is necessary and plural production planning and scheduling mechanisms should be combined each other to solve a problem efficiently. It is called multi-stage production planning and scheduling. SCM vendors offer many kinds of production scheduling mechanisms, but do not support this feature. It is important to utilize both aggregated and detailed information, because the processing flow utilizing them is similar to the thinking flow of human brain and the calculation performance will be accelerated. Therefore, the framework of multi-stage production planning and scheduling is required.

Research about OLM and multi-stage production planning and scheduling is conducted from the following viewpoints:

- (1) Evaluation of influence for a supply chain construction architecture by a multi-agent system for nOLM (Roy 2004, Chatfield 2007)



- (2) Research of the adoptive holonic production system featuring a multi-stage decision making mechanism in each holon (Leitao 2006)
- (3) Suggestion of modeling technique using multi-echelon decision making mechanism (Vorst 2000)

However, in these papers related to this study, it is argued about methods to use information treated in just only nOLM. On the basis of the above-mentioned present research, in this paper OLM is expanded to deal with detailed widespread orders that are also handled before receiving actual orders from customers (hereinafter referred to as wide sense OLM (or just OLM)). In addition, several processing modules, which are provided for production planning or production scheduling, cooperate each other combined with orders managed by OLM to realize more effective SCM (see Figure 1).

By realizing such a cooperated integrated framework, acceleration of decision making of work surrounding production environment and smooth information flow in a supply chain of the inside and outside of a company are enabled, and this actualized supply chain can contribute to a company profit increase. In this paper, a design of the cooperated integration framework of production planning and scheduling based on OLM is described.

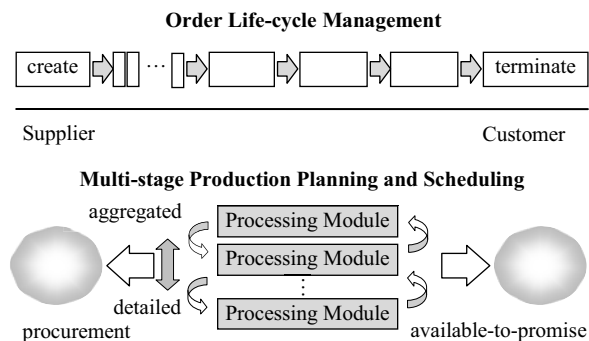


Figure 1: Integration Framework of Production Planning and Scheduling based on OLM

## Order Life-cycle Management

### Order Transition Model

An order is changed by various factors. An order treated as a production order must be changed in synchronization with changes of information from customers and changes of present stock quantity. In addition, according to decision of a human operator in charge of scheduling, it might be changed. When detailed scheduling changes a detailed order, an aggregated order of it should be also changed. OLM is used to follow such changes of orders and to make production planning and scheduling modules cooperate.

In OLM, the following three changes of orders are handled and influence relations between orders are managed.

- (1) Create: It is a change to generate a new order. A generated order is managed until it is terminated in (3).
- (2) Update: It is a change to update information of an order.
- (3) Terminate: It is a change to terminate an order of which management becomes needless.

Through these changes, order information managed by OLM is updated to the latest state. Between two orders, an influence relation can be defined such as A affects B. The following options are offered for these orders connected by an influence relation.

- If A changes, A notifies B upon the changed information.
- B can refer to information of A.
- If an influence relation is removed, B is notified upon it.

For orders managed in this way, the following accessing methods are offered: a method to select the latest orders, a method to extract the chain of order transition, and a method to search orders which an order affects.

### Processing Modules and Views

Orders managed by OLM can maintain various kinds of information and OLM supports the framework in which order information can be referred and changed through the several sides of views. Various processing modules of production planning and scheduling, use orders extracted by one side of views. There are many types of processing modules: a production planning module to fix aggregated orders balancing product mix, an outline scheduling module to set the starting time of operations for the bottle necked process stage, a detailed scheduling module to fix a detailed time schedule and so on. According to the purpose of a processing module, applied orders are different. For the production planning module, only total production amount of aggregated order is used. For the outline scheduling module, aggregated orders fixed by the production planning module are needed. Based on the above information, the detailed scheduling module decides the starting time of all processes using detailed orders. In addition, in accordance with a result of the detailed scheduling module, the amount of aggregated order might be changed and the production planning module might be activated again, furthermore adjustment of the result of the detailed scheduling module might be necessary. This kind of information consistency between different views in the same order and between different orders should be maintained. For the former, the function that updates a view with updating of the other view (view consistency maintenance function) is necessary, and for the latter, the updating function which uses influence relations between

orders in OLM (relation consistency maintenance function) is necessary.

Figure 2 shows the conception of order transition of OLM. Each polyhedron (in this diagram it is cube) shows information of an order, and each order is identified by ID. Type expresses a type of a change of each order in Order Transition Model. An arrow expresses a change of order state, and a bolt arrow shows an influence relation (e.g. from ID:9 to ID:10). In addition, figure 2 shows access of information from two views, and visible views are drawn with gray color. For each order, visible views are defined individually. An order that polyhedron frame is drawn with dotted lines, is already updated at least once (e.g. ID:1-7). An order that polyhedron frame is drawn with solid lines, will be used as the latest information by processing modules (e.g. ID:8-10). An order that polyhedron is drawn transparently is terminated (e.g. ID:11).

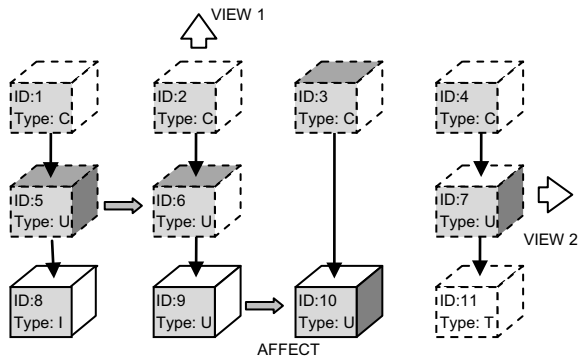


Figure 2: Order Transition Model of OLM

### Visible Information from View

Processing modules of production planning and scheduling refer or change information of orders, which are visible from the designated view. Such order information includes not only actual order information but also many kinds of other information used in the processing module; for example, for a detailed scheduling module, there are recipe information for scheduling, which is used when it makes a detailed schedule for each order, and detailed allocated time information after it is scheduled. Cooperation with various kinds of processing modules is enabled by managing these kinds of information above mentioned as information of order.

### Application Example

In this chapter, an application example for resource allocation is shown to explain how to cooperate OLM with processing modules.

### Applied Process Outline

An example applied in this chapter is scheduling of a single operation, which is important because individual operation schedule affects the production capacity of a whole process. In this example, a process of paper production is assumed; it involves the operations of pulping, papermaking, cutting, finishing, and packing. In particular, the target of this example is focused on the scheduling of papermaking which can use multiple machines. Paper products can be shipped as rolled paper web, or after cutting, finishing and packing. In the latter case, papermaking must be scheduled with allowance for leveling of the downstream operations so as to avoid increases in inter-stage inventories or overload.

The scheduling horizon is set as one week (7 days). The total production volume for every item is given for this period. The goal is to allocate resources for the total production volume with regard to the processing capacity of every resource, shipping schedule, and other conditions. There are three product groups: group A (newspaper A1, A2), group B (copying paper (PPC) B1, B2, B3, B4, B5, B6, B7), and group C (tissue paper C1, C2). Every product group is manufactured by different machines. The total volume of every scheduled product is given in Table 1.

Table 1: Production Orders

Product Group	Quantity(t)	Product	Quantity(t)
A: Newspaper	1,730	A1	1,260
		A2	470
B: PPC Paper	5,500	B1	1,200
		B2	800
		B3	700
		B4	1,500
		B5	300
		B6	500
		B7	500
C: Tissue Paper	1,450	C1	750
		C2	700

Table 2: Production Capacity of Machines

Product Group		Product	Capacity(t/day)		
			(- : not used)		
Group	Subgroup		Machine 1	Machine 2	Machine 3
A	A1	A1	250	450	-
	A2	A2	160	220	-
B	BG1	B1	350	700	-
		B2	350	700	-
		B3	350	700	-
	BG2	B4	-	700	700
		B5	-	700	700
		B6	-	700	700
	B7	B7	-	700	700
C	CG1	C1	-	-	400
		C2	-	-	400

In addition the following constraints and preferences apply to the process.

- ✓ In continuous processes (pulp, papermaking), the production volume is restricted by the capacity of the paper machines. The available machines for every product and their capacities are given in Table 2.
- ✓ Pulp for group A (A1, A2) should be produced as uninterruptedly as possible (without switching to pulp for other products).
- ✓ Since product B4, B5 and B6 involve complex processing, parallel processing should be avoided. That is, these products should not be manufactured by multiple machines simultaneously. In addition they should be manufactured independently of the other products in the same group.
- ✓ Products of group C can be produced at any time.

### Scheduling Procedure by a Human Operator

For this example, scheduling procedure organized by a human operator in charge of scheduling is shown.

	1	2	3	4	5	6	7	8
Machine 1	A1					A2		
Machine 2	BG1					BG2		B7
Machine 3	CG1							

(a) Aggregated Scheduling

	1	2	3	4	5	6	7	8
Machine 1	A1					A2		
Machine 2	BG1					A2	BG2	B7
Machine 3	CG1							

(b) Operation Splitting Adjustment (1)

	1	2	3	4	5	6	7	8
Machine 1	A1					A2		
Machine 2	BG1					A2	BG2	
Machine 3	CG1					BG2	B7	

(c) Operation Splitting Adjustment (2)

	1	2	3	4	5	6	7	8
Machine 1	A1					A2		
Machine 2	B1	B2	B3		A2	B4		
Machine 3	C1			B7	B5	B6	B4	C2

(d) Detailed Scheduling

Figure 3: Schedules of Application Example

#### (1) Aggregated scheduling for each product subgroup (see Figure 3 (a))

Prior to detailed scheduling for every product, the products are organized into several groups, and a simplified schedule is generated for each product group. Operations on a product group are assigned to one of the available machines. After that, a check is made to ascertain out whether all the products can be manufactured in the

required volumes during the scheduling period. In this application example, products B1/B2/B3, B4/B5/B6, and C1/C2 are similar in processing capacity and available machines, and hence these are treated as product subgroup BG1, BG2 and CG1 respectively. Products A1 and A2 can be processed by Machine 1 or 2; here they are preferentially assigned to Machine 1 with higher capacity, then check the result. As regards subgroup BG1, subgroup BG2, and product B7, parallel processing by multiple machines should be avoided as much as possible. Thus, they are assigned only to Machine 2, and then check the result. Subgroup CG1 can only be processed by Machine 3, and therefore they are assigned to Machine 3 and check the available processing time of Machine 3. Since Machine 1 is heavily loaded, part of the products must be shifted to Machine 2. In turn, Machine 2 cannot process both the initially assigned products and those shifted from Machine 1, and therefore, part of the products must be shifted to Machine 3.

#### (2) Operation splitting adjustment (1) (see Figure 3 (b))

For each simplified schedule of product groups, the work is split and the operations are reassigned so as to finish production within the scheduling period. First consider the partial shifting of A1 or A2 to Machine 2. In this example, the production of A2 is split, and part of it is shifted to Machine 2. Then the processing time is adjusted according to the processing capacity. According to the preference that pulp for A1 and A2 should be produced as uninterruptedly as possible, processing by Machine 2 is placed immediately before processing of A2 by Machine 1.

#### (3) Operation splitting adjustment (2) (see Figure 3 (c))

The work is split again, and the operations are re-assigned so as to resolve conflicts caused by correction (2). In this application example, shifting A2 to Machine2 results in competition with product subgroup BG2. Thus the production of subgroup BG2 is split, and part of it is shifted to Machine 3. Since parallel processing of BG2 should be avoided, modification is performed as shown in Figure 3(c). In addition, product B7 is also shifted to Machine 3. The parallel processing of B7 and BG2 is unavoidable in terms of total capacity.

#### (4) Detailed Scheduling for individual products (see Figure 3 (d))

(see Figure 3 (d))

When the scheduling for product groups is completed, operation splitting and time adjustment are performed for each product. In the case of Machine 2, the product groups are divided into products, and the processing order is determined simply. In the case of Machine 3, C1 and C2 are separated, and the processing order is determined. The same applies to product subgroup BG2. Considering the preference restricting parallel processing of B7 with the subgroup BG2, this product is swapped with C2 in the processing sequence. As a result, B7 and B3 are processed in parallel. After that, a final schedule can be obtained.

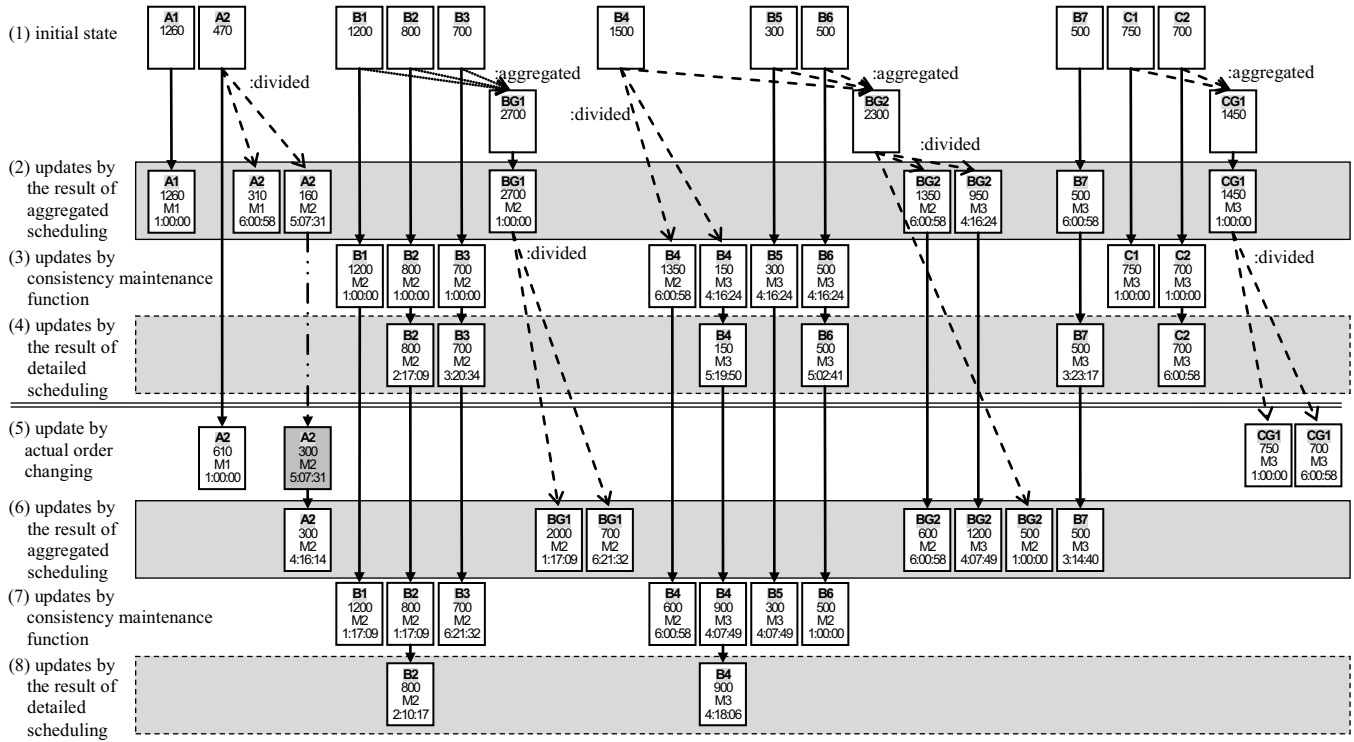


Figure 4: Order Transition of Application Example

### Order Transition and Cooperation with Processing Modules

In this section, it is described how orders are changed in OLM and processing modules are cooperated with OLM, when the proposed framework is applied to this application example. Figure 4 shows the order transactions appearing in this example. Each Box expresses a state of order. Information of order is described on the inside of a box: a product name or subgroup name in the first line, required quantity in the second line, a machine ID in the third line and starting time (date:hours:minutes) in the fourth line. A downward arrow shows transition of order state and the other dotted arrow shows influence relations.

Processing flow is almost same as the aforementioned scheduling procedure by a human operator. The process from (1) to (3) in the former section corresponds to aggregated scheduling, and the process of (4) corresponds to detailed scheduling.

- (1) Initial state: Initial orders are created and connected to related orders with an influence relation such as "divided" or "aggregated".
- (2) Execute the aggregated scheduling module: Before execution, orders referred by this module are designated. Using these order information, aggregated scheduling is executed. After execution, using changed information, orders are updated. The scheduling result is the same as Figure3 (c).

- (3) Update orders by consistency maintenance function: According to the result of (2) and influence relations, some other orders are updated for consistency.
- (4) Execute the detailed scheduling module: Before execution, orders referred by this module are designated. Using information of these order, detailed scheduling is executed. At first, starting time of each order is already set and inconsistency might be occurred, however in this module all of it should be solved. After execution, using changed information, orders are updated. The scheduling result is the same as Figure3 (d).

Furthermore, to explain the interaction of processing modules when an order is changed, one scenario is prepared which is a case when the quantity of an order is changed after building a certain schedule.

- (5) Update by actual order changing: Changed order is drawn with gray color in Figure 4. In this scenario, quantity of the order is changed from 160 to 300.
- (6) Execute the aggregated scheduling module: using already assigned information for orders, the aggregating scheduling is executed again. Only affected part is modified. The scheduling result is shown in Figure 5 (a).
- (7) Update orders by consistency maintenance function
- (8) Execute the detailed scheduling module: The scheduling result is shown in Figure 5 (b).

	1	2	3	4	5	6	7	8
Machine1	A1					A2		
Machine2	BG2	BG1			A2	BG2	BG1	
Machine3	CG1			B7	BG2	CG1		

(a) Aggregated Scheduling

	1	2	3	4	5	6	7	8
Machine1	A1					A2		
Machine2	B6	B1	B2		A2	B4	B3	
Machine3	C1			B7	B5	B4	C2	

(b) Detailed Scheduling

Figure 5: Changes by Re-Scheduling

It is shown that the proposed integration framework of processing modules based on OLM is applicable to the thinking process of human brain and effective for the re-scheduling caused by some information changes.

### Required Functions for Processing Modules

For the multi-stage production planning and scheduling system, processing modules to perform various kinds of decision making must be prepared. Moreover, the framework based on the centered OLM system that an individual scheduling result affects mutually, is necessary. The author has developed Self-construction Scheduling System called *ScheMe* as a system which learns master information and the scheduling technique that are necessary for scheduling through operation of a user in scheduling duties and uses the information acquired by the former learning mechanism as a user support function (Fujimura 2010). This system can be applied in various kinds of process and it offers the structure, which can apply the different scheduling approaches according to its situation for the same target process. To realize multi-stage production planning and scheduling system which can make a decision one by one based on orders extracted from OLM through its own view, the following functional extension is required.

- (1) **Lightweight Recipe:** Recipe is information referred to make a product from order information; for a scheduling system it is used to build a schedule. Generally, recipe information is prepared before using this system. However, in this proposed framework, order information is dynamically changed and recipe should be adopted flexibly. For this requirement, the functions to make it dynamically and add its information to order information are required.
- (2) **Multi-View Gantt Chart:** Generally, purpose of production scheduling is to assign a used resource or machine and set the starting time of each operation. However, in this proposed framework, according the

level of abstraction, instead of these it might be important to set only sequence of processing operations. Therefore, Multi-View Gantt Chart which can be used to watch such kinds of information from various kinds of viewpoints is necessary.

- (3) **Repair Function for Changed Orders:** As shown in the former chapter, for re-scheduling, most parts of already built schedule should be kept intact and this feature makes a decision more smoothly. These kinds of information can be stored in OLM system and maintained with consistency. So processing modules should refer and reuse these as useful information.

### Conclusion

In this paper, a design of the cooperated integration framework of production planning and scheduling based on the centered database of OLM is proposed. OLM provides the order transition model, and enables the management of changes of order information with consistency and cooperation with processing modules for production planning and scheduling.

By realizing such a cooperated integrated framework, acceleration of decision making of work surrounding production environment and smooth information flow in a supply chain of the inside and outside of a company are enabled, and this actualized supply chain can contribute to a company profit increase.

### Acknowledgments

This study was assisted by a MEXT 2008-10 Grant-in-Aid for Scientific Research [Fundamental Research (C) No. 20560388], and we take this occasion to express our deep gratitude.

### References

- Roy, D.; Anciaux, D.; Monteiro, T.; and Ouzizi, L. 2004. Multi-agent architecture for supply chain management. *Journal of Manufacturing Technology Management*, Vol.15, No.8: 745-755
- Chatfield, D.C.; Hayya, J.C.; and Harrison, T.P. 2007. A multi-formalism architecture for agent-based, order-centric supply chain simulation. *Simulation Modeling Practice and Theory*, Vol.15: 153-174
- Leitao, P.; and Restivo, F. 2006. ADACOR: A holonic architecture for agile and adaptive manufacturing control. *Computers in Industry*, Vol.57: 121-130
- Vorst, J.G.A.J.van der; Beulens, A.J.M.; and P.van Beek 2000. Modeling and Simulating multi-echelon food systems. *European Journal of Operational Research*, Vol.122: 354-366
- Xue, H.; Zhang, X.; Shimizu, Y.; and Fujimura, S., 2010. Conception of self-construction production scheduling system. *Electronics and Communications in Japan*, Vol.93, Issue 1: 19-29

# Relational Approach to Knowledge Engineering for POMDP-based Assistance Systems with Encoding of a Psychological Model

**Marek Grześ, Jesse Hoey & Shehroz Khan   Alex Mihailidis & Stephen Czarnuch**

School of Computer Science  
University of Waterloo, Canada

Dept. of Occupational Science and Occupational Therapy  
University of Toronto, Canada

**Dan Jackson**

School of Computing Science  
Newcastle University, UK

**Andrew Monk**

Department of Psychology  
University of York, UK

## Abstract

Partially observable Markov decision process (POMDP) models have been used successfully to assist people with dementia when carrying out small multi-step tasks such as hand washing. POMDP models are a powerful, yet flexible framework for modeling assistance that can deal with uncertainty and utility. Unfortunately, POMDPs usually require a very labor intensive, manual setup procedure. Our previous work has described a knowledge driven method for automatically generating POMDP activity recognition and context sensitive prompting systems for complex tasks. We call the resulting POMDP a SNAP (SyNdetic Assistance Process). In this paper, we formalise this method using a relational database. The database encodes the goals, action preconditions, environment states, cognitive model, user and system actions, as well as relevant sensor models, and automatically generates a valid POMDP model of the assistance task. The strength of the database is that it allows constraints to be specified, such that we can verify the POMDP model is, indeed, valid for the task. To the best of our knowledge, this is the first time the MDP planning problem is formalised using a relational database. We demonstrate the method on three assistance tasks: handwashing, and toothbrushing for elderly persons with dementia, and on a factory assembly task for persons with a cognitive disability.

## 1 Introduction

Quality of life (QOL) of persons with a cognitive disability (e.g. dementia, developmental disabilities) is increased significantly if they can engage in “normal” routines in their own homes, workplaces, and communities. However, they generally require some assistance in order to do so. For example, difficulties performing activities of daily living at home, such as preparing food, washing, or cleaning, or in the workplace, such as factory assembly, may trigger the need for personal assistance or relocation to residential care settings (Gill and Kurland 2003). Moreover, it is associated with diminished QOL, poor self-esteem, anxiety, and social isolation for the person and their caregiver (Burns and Rabins 2000).

Technology to support people in their need to live independently is currently available in the form of personal and social alarms and environmental adaptations and aids. Looking to the future, we can imagine intelligent, pervasive computing technologies using sensors and effectors that help with more difficult cognitive problems in planning, sequencing and attention. In the example of assisting people with dementia, the smart environment would prompt whenever the residents get stuck in their activities of daily living.

The technical challenge of developing useful prompts and a sensing and modelling system that allows them to be delivered only at the appropriate time is difficult, due to issues such as the system needing to be able to determine the type of prompt to provide, the need for the system to recognize changes in the abilities of the person and adapt the prompt accordingly, and the need to give different prompts for different sequences within the same task. However, such a system has been shown to be achievable through the use of advanced planning and decision making approaches. One of the more sophisticated of these types of systems is the COACH (Hoey et al. 2010). COACH uses computer vision to monitor the progress of a person with dementia washing their hands and prompts only when necessary. COACH uses a partially observable Markov decision process (POMDP), a temporal probabilistic model that represents a decision making process based on environmental observations. The COACH model is flexible in that it can be applied to other tasks (Hoey et al. 2005). However, each new task requires substantial re-engineering and re-design to produce a working assistance system, which currently requires massive expert knowledge for generalization and broader applicability to different tasks. An automatic generation of such prompting systems would substantially reduce the manual efforts necessary for creating assistance systems, which are tailored to specific situations and tasks, and environments. In general, the use of *a-priori* knowledge in the design of assistance systems is a key unsolved research question. Researchers have looked at specifying and using ontologies (Chen et al. 2008), information from the Internet (Pentney, Philipose, and Bilmes 2008), logical knowledge bases (Chen et al. 2008; Mastrogiovanni, Sgorbissa, and Zaccaria 2008), and programming interfaces for context aware human-computer interaction (Salber, Dey, and Abowd 1999).

In our previous work, we have developed a knowledge driven method for automatically generating POMDP activity recognition and context sensitive prompting systems (Hoey et al. 2011). The approach starts with a description of a task and the environment in which it is to be carried out that is relatively easy to generate. Interaction Unit (IU) analysis (Ryu and Monk 2009), a psychologically motivated method for transcoding interactions relevant for fulfilling a certain task, is used for obtaining a formalized, i.e., machine interpretable task description. This is then combined with a specification of the available sensors and effectors to build a work-

ing model that is capable of analyzing ongoing activities and prompting someone. We call the resulting model a SyNdedic Assistance Process or SNAP. However, the current system uses an ad-hoc method for transcoding the IU analysis into the POMDP model. While each of the factors are well defined, fairly detailed and manual specification is required to enable the translation.

The long-term goal of the approach presented in this paper is to allow end-users, such as health professionals, caregivers, and family members, to specify and develop their own context sensitive prompting systems for their needs as they arise. This paper describes a step in this direction by defining a relational database that serves to mediate the translation between the IU analysis and the POMDP specification. The database encodes the constraints required by the POMDP in such a way that, once specified, the database can be used to generate a POMDP specification automatically that is guaranteed to be valid (according to the SNAP model). According to the best of our knowledge, this is the first time the MDP planning problem is formalised using a relational database. This novel approach helps coping with a number of issues, such as validation, maintenance, structure, tool support, association with a workflow method etc., which were identified to be critical for tools and methodologies which could support knowledge engineering in planning (McCluskey 2000). This paper gives the details of this relational database, and shows how it solves these various issues. It then demonstrates the application of this method to specify a POMDP in three examples: two are for building systems to assist persons with dementia during activities of daily living, and one is to assist persons with Down's syndrome during a factory assembly task. We show how the method requires little prior knowledge of POMDPs, and how it makes specification of relatively complex tasks a matter of a few hours of work for a single coder.

The remainder of this paper is structured as follows. First, we give an overview of the basic building blocks: Knowledge engineering requirements, POMDPs, and IU analysis. Then, Section 3 describes the relational database we use, frames the method as a statistical relational model, and shows how the database can be leveraged in the translation of IU analysis to POMDP planning system. Section 4 shows how the method can be applied to three tasks, and then the paper concludes.

## 2 Overview of the method

### 2.1 Requirements from Knowledge Engineering

The IU analysis and the sensor specification need to be translated into a POMDP model, and then the policy of action can be generated. The relational database provides a natural link between these two elements of the prompting system, and the use of the database represents additionally a novel approach to knowledge engineering (KE) for planning. For an extensive review of challenges which KE for planning faces, the reader is referred to (McCluskey 2000). This area is essentially investigating the problem of how planning domain models can be specified by technology designers who are not necessarily familiar with the AI planning technology. In (McCluskey 2000), authors collected a number of requirements which such a methodology should satisfy. Some of most important ones are: (1) acquisition, (2) validation, (3) maintenance, and additionally the representation language should

be: (4) structured, (5) associated with a workflow method, (6) easy to assess with regard to the complexity of the model, (7) tool supported, (8) expressive and customizable, and (9) with a clear syntax and semantics. In our work on the SNAP process, we found that these requirements can be to a great extent supported when one applies the relational database formalism to store and to process the domain model. The acquisition step (1) does not have its full coverage in our case since, e.g., the types of planning actions are known, as well as the structure of the IU analysis. This allows specifying the structure of the relational database and designing SQL-tables beforehand and reusing one database model (see Section 3) in all deployments of the system. The database technology is a standard method of storing data, and checking validation (2) of the data is highly supported. This includes both simple checks of data types, as well as arbitrarily complex integrity checks with the use of database triggers. Once the database of a particular instance is populated, the designer can automatically generate a SNAP for a particular user/task/environment combination taking input for the sensors through the ubiquitous sensing technician's interface, and the POMDP can be fed into the planner, and then simulated. Since, the overall process is straightforward for the designer, this allows for a traditional dynamic testing of the model, where the designer can adjust the domain model easily via the database interface, generate a new POMDP file, and then simulate it and assess its prompting decisions. This shows that also maintenance (3) is well supported in our architecture. The SQL relational language is also flexible in representing structured (4) objects. In our work, it is used in conjunction with a workflow method (5), where the technology designer follows specific steps which require populating specific tables in the database. The relational database technology is one of the most popular ways of storing data, and it is vastly supported by tools and those tools are nowadays becoming familiar even to a standard computer user. In our implementation, a PHP-based web interface is used, which from the user's point of view does not differ from standard database-based systems.

### 2.2 Partially observable Markov decision processes

A POMDP is a probabilistic temporal model of a system interacting with its environment (Åström 1965), and is described by (1) a finite set of state variables, the cross product of which gives the state space,  $S$ ; (2) a set of observation variables,  $O$  (the outputs of some sensors); (3) a set of system actions,  $A$ ; (4) a reward function,  $R(s, a, s')$ , giving the relative utility of transitioning from state  $s$  to  $s'$  under action  $a$ ; (5) a stochastic transition model  $Pr : S \times A \rightarrow \Delta S$  (a mapping from states and actions to distributions over states), with  $Pr(s'|s, a)$  denoting the probability of moving from state  $s$  to  $s'$  when action  $a$  is taken; and (6) a stochastic observation model with  $Pr(o|s)$  denoting the probability of making observation  $o$  while the system is in state  $s$ . Figure 1(a) shows a POMDP as a Dynamic Bayesian network (DBN) with actions and rewards, where arrows are interpretable as causal links between variables.

### 2.3 Specifying the task: Interaction Unit Analysis

Task analysis has a long history in Human Factors (Kirwan and Ainsworth 1992) where this approach is typically used to help define and break-down 'activities of daily living'

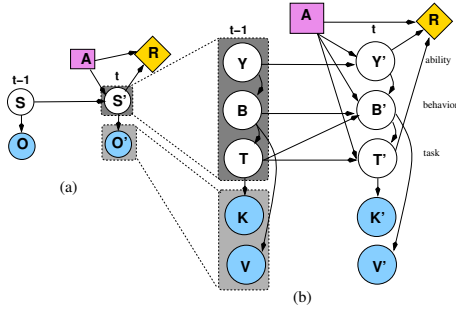


Figure 1: Two time slices of (a) a general POMDP; (b) a factored POMDP for interactions with assistive technology.

(ADL)— i.e. activities that include self-care tasks, household duties, and personal management such as paying bills. The emphasis in task analysis is on describing the actions taken by a user and the intentions (goals and sub-goals) that give rise to those actions. There has been less emphasis on how actions are driven by the current state or changes in the environment. Syndetic modeling (Duke et al. 1998) remedies this omission by describing the conjunction of cognitive and environmental precursors for each action. Modeling both cognitive and environmental mechanisms at the level of individual actions turns out to be much more efficient than building separate cognitive and environmental models (Ryu and Monk 2009).

The task analysis technique (Wherton and Monk 2009), breaks a task down into a set of goals, states, abilities and behaviours, and defines a hierarchy of tasks that can be mapped to a POMDP, a policy for which will be a situated prompting system for a particular task (Hoey et al. 2011). The technique involves an experimenter video-taping a person being assisted during the task, and then transcribing and analysing the video. The end-result is an Interaction Unit (IU) analysis that uncovers the states and goals of the task, the client’s cognitive abilities, and the client’s actions. A simplified example for the first step in tea-making (getting out the cup and putting in a tea-bag) is shown in Table 1. The rows in the table show a sequence of steps, with the client’s current goals, the current state of the environment, the abilities that are necessary to complete the necessary step, and the behaviour that is called for. The abilities are broken down into ability to *recall* what they are doing, to *recognise* necessary objects like the kettle, and to *perceive affordances* of the environment.

A second stage of analysis involves proposing a set of sensors and actuators that can be retrofitted to the user’s environment for the particular task, and providing a specification of the sensors that consists of three elements: (1) a name for each sensor and the values it can take on (e.g. on/off); (2) a mapping from sensors to the states and behaviours in the IU analysis showing the evidentiary relationships, and (3) measurements of each sensor’s reliability at detecting the states/behaviours it is related to in the mapping.

The IU analysis (e.g. Table 1) can be converted to a POMDP model by factoring the state space as shown in Figure 1(b). The method is described in detail in (Hoey et al. 2011), here we give a brief overview. The *task* variables are a characterisation of the domain in terms of a set of high-level variables, and correspond to the entries in the state column in Table 1. For example, in the first step of tea making, these

include the box condition (open, closed) and the cup contents (empty or with teabag). The task states are changed by the client’s *behavior*,  $B$ , a single variable with values for each behaviour in Table 1. For the first IU group in tea making, these include opening/closing the box, moving the teabag to the cup, and doing nothing or something unrelated (these last two behaviours are always present). The client’s *abilities* are their cognitive state, and model the ability of the client to recall (Rl), recognise (Rn) and remember affordances (Af). For the first IU group, these include the ability to recognise the tea box and the ability to perceive the affordance of moving the teabag to the cup.

The system actions are prompts that can be given to help the client regain a lost ability. We define one system action for each necessary ability in the task. The actions correspond to a prompt or signal that will help the client with this particular ability, if missing. *Task* and *behavior* variables generate observations,  $O$ . For example, in a kitchen environment there may be sensors in the counter-tops to detect if a cup is placed on them, sensors in the teabags to detect if they are placed in the cup, and sensors in the kettle to detect “pouring” motions. The sensor noise is measured independently (as a miss/false positive rate for each state/sensor combination) (Pham and Olivier 2009; Hoey et al. 2011).

### 3 Relational Database

The technology designer should be able to define the prompting system (planning problem specification) easily and with minimal technical knowledge of planning. The approach we are proposing in this paper is to provide a relational database which can be populated by the designer using standard database tools such as forms or web interface, and then translating the database into the POMDP specification using a *generator* software, which implements parts of the overall relational model not stored in the database. This is in accordance with the main goal of the way relational databases should be used. The database in itself explicitly stores the minimum amount of information which is sufficient to represent the concept. Those relations which are not represented explicitly are then extracted on demand using SQL queries. We adhere to this standard in our design and our generator contains such implicit parts of the model which are not stored in the database. Below, we show how our methodology of specifying planning tasks is motivated and justified by locating this work in the context of relevant AI research on planning, and probabilistic and relational modelling.

In the application areas which are considered in this paper, planning problems are POMDPs. POMDPs can be seen as Dynamic Decision Networks (DDNs). In POMDP planners, DDNs have propositional representation, where the domain has a number of attributes, and attributes can take values from their corresponding domains. The problem with designing methodologies for such propositional techniques is that the reuse of the model in new instances is not straightforward, and a relational approach becomes useful. In the case of modelling POMDPs, Statistical Relational Learning (Getoor and Taskar 2007) is the way to make relational specification of DDNs possible.



IU	Goals	Task States	Abilities	Behaviours
1	Final	cup empty on tray, box closed	Rn cup on tray, Rl step	No Action
2	Final, cup TB	cup empty on tray, box closed	Af cup on tray WS	Move cup tray→WS
3	Final, cup TB	cup empty on WS, box closed	Rl box contains TB, Af box closed	Alter box to open
4	Final, cup TB	cup empty on WS, box open	Af TB in box cup	Move TB box→cup
5	Final	cup tb on WS, box open	Af box open	Alter box to closed
	Final	cup tb on WS, box closed		

Table 1: IU analysis of the first step in tea making. Rn=recognition, Rl=Recall, Af=Affordance, tb=teabag, ws=work surface.

### 3.1 Statistical Relational Learning

Probabilistic Relational Models (PRM) define a template for a probability distribution (Getoor and Taskar 2007), that specifies a concrete distribution when ground with specific data. The family of distributions that can be obtained from a specific PRM is what we seek in the problem of specifying POMDPs for prompting systems. Our goal is to have a template which would be flexible and general enough to represent POMDPs for different tasks, but also specific enough so that one relational model would be sufficient. Figure 2 shows the main components of the probabilistic model specified using the PRM.

The first element is the relational schemata which can be formalised as a specification of types of objects, their attributes, and relations between objects of specific types. The two additional components are: for each attribute the set of

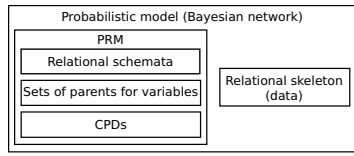


Figure 2: The probabilistic model and its components when specified as a PRM.

parents this attribute probabilistically depends on, and the corresponding conditional probability distributions. These elements together with the relational structure is exactly what is shared between different prompting systems which we build for cognitively disabled people, and this part of the model can be designed beforehand using the outcomes of years of research in this area which ranges from artificial intelligence to cognitive psychology. We argue in this paper, that the relational database is a good way of representing and specifying these kinds of models which define POMDP planning problems. The PRM part of the model is exactly what can be designed beforehand by POMDP planning experts and cognitive scientists, and every particular deployment of the system will be reduced to populating the database by the technology designer who is not required to have POMDP specific knowledge.

The relational schemata of the PRM can be represented directly as a standard relational database where tables and implicit tables determined by SQL queries define objects, and columns in tables define attributes. Relationships are modelled as primary/foreign key constraints.

The two remaining elements of the PRM are also partially incorporated in the relational database, defined as SQL queries to the database, or explicitly encoded in the software which reads the database and produces the final input file for the POMDP planner. The PRM model contains everything which is required to obtain the probabilistic model

for the specific case, except for the data – objects, values of their attributes, values of probabilities, and specifications of some dependencies which are represented relationally in the model – and this complementary element is named a relational skeleton in (Getoor and Taskar 2007). This skeleton contains objects which adhere to relational constraints defined in the relational model (database tables, SQL queries, the generator implementation). Once this skeleton is provided, the PRM can be translated into a ground Bayesian network in the original case, and into a ground DDN in our implementation which has to model time (two slice Bayesian network) and decisions.

### 3.2 Relational Schemata

The above discussion showed how our methodology originates from the state-of-the-art methods for relational probabilistic modelling. In this paragraph, we show technical details of how the database was designed. Figure 3 shows the structure of the entire database. All tables which have their names starting with `t_iu_` represent the IU table, and the user interface shows the view of the full IU table to the user (not individual tables separately). The core table is `t_env_variables_values` which stores domain attributes and their possible values. The sensor model, `t_sensor_model`, associates environment variables with sensors (`t_observations_values`). There is also a sensor model for behaviours in the corresponding table. There is a table for possible behaviours of the client in the modelled domain (`t_behaviours`) and dynamics of the client's actions are defined in associated tables which store effects and preconditions of behaviours. Essentially, the database encoding of effects and preconditions of client's actions contains information which is equivalent to STRIPS operators in classical planning. One behaviour can have different effects depending on the precondition. Additionally, our probabilistic model can make use of the states in which a specific behaviour is impossible (`t_when_is_behaviour_impossible`). Table `t_abilities` stores client's abilities which are relevant to the task. Finally, rewards are defined in `t_rewards` and the associated table allows specifying sets of states which yield a particular value of the reward.

### 3.3 Relational Probabilistic Model

We present only one piece of the relational specification of probabilistic dependencies: the client behaviour dynamics model.

Let us assume that  $I$  is a set of rows in the IU table.  $T$ ,  $T'$ ,  $B$ ,  $B'$ ,  $Y$ , and  $Y'$  are as specified in Figure 1b.  $\rho$  is a random behaviour constant and is set to 0.01 in the current implementation. We define the following functions:

1.  $row\_rel : I \times T \rightarrow \{0, 1\}$  is 1 for task states relevant in

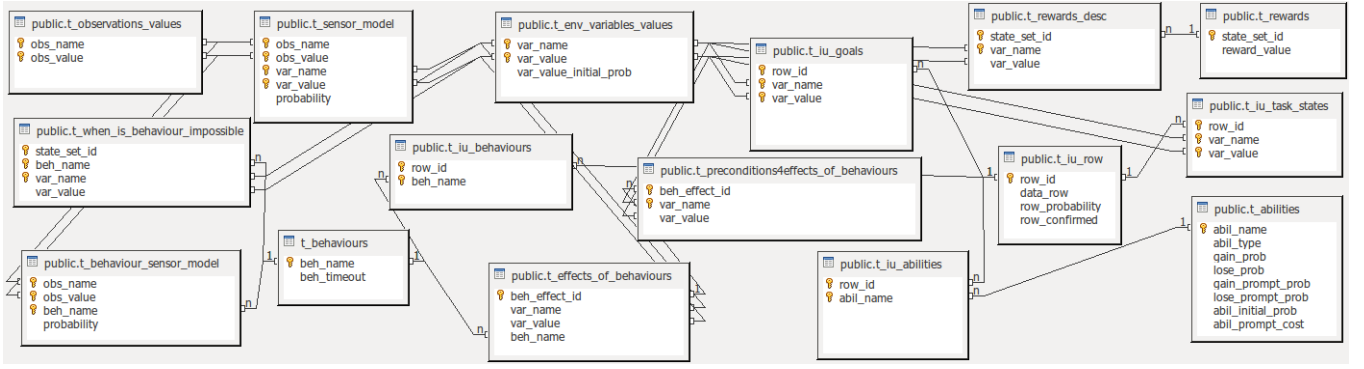


Figure 3: Complete SNAP database diagram.

- row,  $i$ , and 0 otherwise. We write this as of the row,  $i$ , only:  $row\_rel(i)$  leaving the remaining variables implicit. The same shorthand is applied to the other functions.
2.  $row\_rel\_b : I \times T \times B' \rightarrow \{0, 1\}$  is defined as  $row\_rel\_b(i, b') = row\_rel(i) \wedge behaviour(i, b')$  where  $behaviour(i, b') = 1$  when  $b'$  is the behaviour of row  $i$ .
  3.  $row\_abil\_rel : I \times Y' \rightarrow \{0, 1\}$  is 1 when all abilities of row  $i$  are satisfied by  $y'$ , and 0 otherwise.
  4.  $goal : T \rightarrow \{0, 1\}$  is 1 when  $t$  is a goal state, 0 otherwise.
  5.  $bn : B' \rightarrow \{0, 1\}$  is 1 when  $b' = nothing$ , 0 otherwise.
  6.  $same : B \times B' \rightarrow \{0, 1\}$  is 1 when  $b = b'$ . This is a bias which indicates that behaviours are likely to stay the same.
  7.  $impossible\_beh : T \rightarrow \{0, 1\}$  is 0 for states  $t$  when there is no behaviour which is possible in  $t$ , and 1 when there is at least one behaviour which is possible in  $t$ .
  8. For all functions defined above,  $\neg f(x)$  is defined as  $\neg f(x) = 1 - f(x)$  which defines a negation of  $f(x)$  when 0 and 1, the domain of  $f(x)$ , are treated as boolean values.

The above functions are used in the definition of the dynamics of behaviours  $B'$ ,  $beh\_dyn : B' \times Y' \times T \times B \rightarrow [0, 2]$ .

$$beh\_dyn =$$

$$\sum_{i \in I} [row\_abil\_rel(i) \wedge row\_rel\_b(i) \wedge p(i) \vee$$

$$\neg row\_abil\_rel(i) \wedge row\_rel(i) \wedge bn] \vee$$

$$\prod_{i \in I} [\neg row\_rel(i)] \wedge bn \vee$$

$$goal \wedge bn \vee$$

After normalisation,  $beh\_dyn$  defines probability  $P(b'|b, y', t)$  of  $b'$  when the previous behaviour was  $b$ , the person will have abilities  $y'$ , and the system is in state  $t$ . It is important to recall that non-invasive prompts are assumed here which influence abilities  $Y'$  and not behaviours  $B'$  directly. The first term (1) of this equation is for rows which have their abilities and state relevance satisfied. (2) defines behaviour 'nothing' when state is relevant in the row but abilities are not present. (3) sets behaviour 'nothing' in all states which are not relevant in any row. (4) reflects the fact that only behaviour 'nothing' happens when the goal state has been reached. It is important to note here, that the IU analysis in original SNAP (Hoey et al. 2011) has to have task states in all rows disjunctive, which means that each state can be relevant in one row only. This is of course not

always the case in practice, and we add an extension here which specifies probability  $p(i)$  of a row, used in (2), when there are other rows which have the same states relevant.

All the functions necessary to specify the POMDP are represented as algebraic decision diagrams (ADDs) in SPUDD notation (Hoey et al. 1999). These functions are computed with queries on the database. Each such query extracts some subset of ADDs from the relational skeleton. The ADDs are then combined using multiplication and addition to yield the final conditional probability tables (CPTs). Some relations are explicitly represented in the database whereas others need to be extracted using more complex SQL queries. For example, data for  $row\_rel(i)$ ,  $row\_rel\_b(i)$ , and  $row\_abil\_rel(i)$  is read from the IU table in the database. The example subset of the relational skeleton for the IU analysis from Table 1, and diagrams for selected functions are in Figure 4. SQL queries extract compound algebraic decision diagrams from the relational skeleton, and the generator software multiplies those diagrams in order to obtain final functions, such as  $P(b'|b, y', t) = row\_rel\_b(i, b')$ . The following more complex SQL query example for  $goal$  returns sets of states with the highest reward:

```
SELECT var_name, var_value, reward_value,
       t_rewards.state_set_id
FROM t_rewards_desc INNER JOIN t_rewards
ON t_rewards_desc.state_set_id=t_rewards.state_set_id
WHERE reward_value=(SELECT MAX(reward_value)
                    FROM t_rewards)
ORDER BY 4
```

A schematic is shown in Figure 4, where the CPT for client behaviour dynamics is gathered from the relevant tables in the relational skeleton. The original table schema in the PRM for the relations in Figure 4 can be seen in Figure 3.

### 3.4 Advantages of Database Engines

The advantage of the relational database is that it allows for easy implementation of the constraints required by the model. The simplest example are constraints on attribute values. For example, probabilities have to be in the range  $[0, 1]$ , or literals should follow specific naming patterns (according to the requirements of the POMDP planner). These simple constraints are easily implemented in the definition of SQL tables. Some more complex constraints which involve more than one attribute are also required. For instance in the planner which we use, sensors and domain attributes are in the

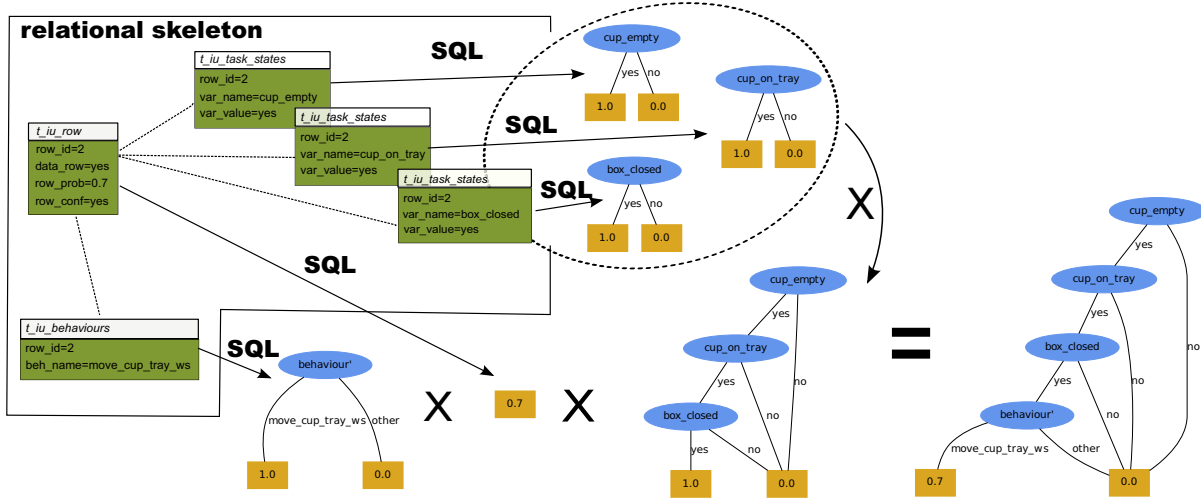


Figure 4: Subset of the relational skeleton for  $P(b|b, y, t')$  showing the generation of  $\text{behaviour}(i, b') \wedge p(i) \wedge \text{row\_rel}(i) = \text{row\_rel\_b}(i, b')$  for row  $i = 2$  in Table 1. The algebraic decision diagrams (ADDs) representing the relations are multiplied to give the final CPT.

same name space, which means that their names have to be different. Such things can be easily implemented using database triggers, and the user will be prompted at the input time and informed about the constraint.

#### 4 Demonstrative Examples

We demonstrate the method on three assistance tasks: handwashing, and toothbrushing with older adults with dementia, and on a factory assembly task for persons with a developmental disability. We show that once our relational system is designed (i.e. the database and the generator which reads the database and outputs the POMDP file), the system is generic and allows the designer to deploy the system in different tasks simply by populating the database for the new task. The IU analysis for handwashing was performed by a professional IU analyser, and handwashing was used as the testbed for our method. The analysis for the other two tasks were performed by two different biomedical engineers, with limited experience with POMDPs or planning in AI. As an example of the power of our method, the factory task was coded in about six hours by the engineer. This can be compared to a manual coding of the system for handwashing (a smaller task), that took at least three months of work resulting in the system described in (Boger et al. 2006).

The IU analysis breaks an ADL down into a number of sub-tasks, or sub-goals. For the factory task, there are six sub-goals. The decomposition arises according to the elements of recall noted in the IU analysis videos. The six sub-goals are partially ordered, and the partial ordering can be specified as a list of pre-requisites for each sub-goal giving those sub-goals that must be completed prior to the sub-goal in question. Since each sub-goal is implemented as a separate POMDP controller, a mechanism is required to provide hi-level control to switch between sub-goals. We have implemented two such control mechanisms. A deterministic controller is described in (Hoey et al. 2011), and a probabilistic and distributed method in (Hoey and Grzes 2011). All con-

trollers described in the last section are implemented in Java, and run as separate processes and can be easily distributed across several PCs ensuring scalability.

##### 4.1 COACH and prompting

Examples of automatic generation of task policies using IU analyses and a relational database were implemented for the task of handwashing and toothbrushing. For these examples, people with mild to moderate dementia living in a long term care facility were asked to wash their hands and brush their teeth in two separate trials.

**Handwashing** The washroom used for this task had a sink, pump-style soap dispenser and towel. Participants were led to the sink by a professional caregiver, and were encouraged to independently wash their own hands. The IU analysis was performed on videos captured from a camera mounted above the sink. The task was broken into 5 steps, each comprised of multiple substeps: 1) wet hands; 2) get soap; 3) wash hands; 4) turn off water; and 5) dry hands. Steps 1 and 2 can be completed in any order, followed by step 3. After completion of step 3, steps 4 and 5 can be completed in any order.

**Toothbrushing** The videos used for the analysis captured participants in a washroom that had a sink, toothbrush, tube of toothpaste and cup, as they tried to independently brush their own teeth. A formal caregiver was present to provide coaching and assistance if required. The IU analysis was completed based on videos of several different people and included multiple different methods of completing the task. The task was divided into 6 main steps, each containing multiple sub-steps: 1) wet brush; 2) apply toothpaste; 3) brush teeth; 4) clean mouth; 5) clean brush; and 6) tidy up. Steps 1 and 2 can be completed in any order, followed by step 3. Steps 4 and 5 can also be completed in any order after step 3, and step 6 is the final step following the first 5 steps.

A policy was generated for each sub-step entered into the relational database. Simulations were run to test the hand-

washing and toothbrushing policies with a user assumed to have mild-to-moderate dementia. To simulate a person with mild-to-moderate dementia the user was forced to forget steps of the task throughout the simulation (i.e., do nothing) but respond to prompting if provided. Tables 2 (handwashing) and 3 (toothbrushing) show a sample of the belief state of the POMDP, the system's suggested action and the actual sensor states for several timesteps of two simulations. Probabilities of the belief state are represented as the height of bars in corresponding columns of each time step. In the handwashing example, the user is prompted to take the towel ( $t=1$ ). Detecting that the towel was taken ( $t=2$ ), the system prompts the user to dry his/her hands. The sensors indicate the towel was returned to the surface without drying the user's hands ( $t=3$ ) so the system again prompts the user to take the towel. As a second example with the toothbrushing simulation ( $t=1$ ), the system prompts the user to turn on the tap. The user does nothing, so the system tries to prompt the user to take the brush from the cup (in this case either turning the tap on or taking the toothbrush can happen first). The sensors indicate the brush was taken ( $t=3$ ), so the system returns to prompting the user to turn on the tap.

Time step, t	Observations		Task		Behaviour				Ability			System Action (Prompt)			
	hands_wet	towel_position	hands_dry	hands_wet	towel_in_hand	towel_on_surface	other	nothing	take_towel	dry_hands	put_down_towel		Af_dry	Af_put_towel_down	Af_take_towel
0	wet	on_surface													Af_take_towel
1	wet	on_surface													Af_take_towel
2	wet	in_hand													Af_dry
3	wet	on_surface													Af_take_towel
4	wet	in_hand													Af_dry
5	dry	in_hand													Af_put_down_towel
6	dry	on_surface													donothing

Table 2: Example simulation in the handwashing task. The main goal shown in the subtask is to dry the hands after taking the towel from the surface, while the secondary goal is to return the towel to the surface.

Time step, $t$	Observations		Task		Behaviour				Ability			System Action (prompt)						
	brush_wet	tap	brush_position	brush_wet	brush_in_hand	brush_in_cup	brush_on_surface	tap_on	other	nothing	alter_tap_to_on		take_brush_from_cup	wet_brush	take_brush_from_surface	Rn_brush_in_cup	Rn_brush_on_surface	Af_tap
0	dry	off	in_cup															Af_tap
1	dry	off	in_cup															Af_tap
2	dry	off	in_cup															Rn_brush_cup
3	dry	off	in_hand															Af_tap
4	dry	on	in_hand															Af_water
5	dry	on	in_hand															Af_water
6	wet	on	in_hand															donothing

Table 3: Example simulation in the toothbrushing task. The goal in the shown sub-task is to turn on the tap, take the toothbrush from either the surface or the cup, and wet the brush.

## 4.2 Factory Assembly Task

In this example, workers with a variety of intellectual and developmental disabilities are required to complete an assembly task of a 'Chocolate First Aid Kit'. This task is completed at a

workstation that consists of five input slots, an assembly area, and a completed area. The input slot contain all of the items necessary to perform kit assembly-specifically the main first aid kit container (white bin), and four different candy containers that need to be placed into specific locations within the kit container. The IU analysis was completed based on videotapes of a specific adult worker who has Down's Syndrome completing this assembly task. The worker was assessed with a moderate-to-mild cognitive impairment and was able to follow simple instructions from a job coach. The IU analysis broke this task into 6 required steps: 1) prepare white bin; 2) place in bin chocolate bottle 1; 3) place in bin chocolate box 1; 4) place in bin chocolate box 2; 5) place in bin chocolate bottle 2; and 6) finish output bin and place in completed area. Steps 2, 3, 4, and 5 can be completed in any order. Through a hierarchical task analysis (Stammers and Sheppard 1991) each of these steps were further broken down into sub-steps.

Policies were generated for each of the required assembly steps and were simulated by the authors for three different types of clients: mild, moderate, and severe cognitive impairment. Figure 5 is the output of sample timestamps for step 2 for a user with severe cognitive impairment. Again, probabilities of the belief state are represented as the height of bars in corresponding columns of each time step. In this specific example, the system is more active in its prompting based on the fact that the user is assumed to have diminished abilities with respect to the different aspects that needs to be completed. For example ( $t=1$ ), the worker has deteriorating ability to recognize that the slot that holds the required chocolate bottle is empty. As such, the system correctly prompts the worker to recognize that the slot is empty and needs to be filled. In another example ( $t=5$ ), the system recognizes that the worker has not placed the bottle in its correct location in the white bin, and provides a prompt for the person to recall that the bottle needs to be in that position in order to reach the final goal state. When the worker does not respond to this prompt, the system decides ( $t=6$ ) to play a different, more detailed, prompt (a prompt related to the affordance ability).

## 5 Conclusions and discussion

POMDP models have proven to be powerful for modelling intelligent human assistance (Hoey et al. 2010). Unfortunately, POMDPs, being propositional models, usually require a very labour intensive, manual setup procedure. A standard approach which can make AI models portable and configurable is to introduce the notion of objects and relations between them and this is found in relational methods such as Statistical Relational Learning. In this paper, we derive a methodology for specifying POMDPs for intelligent human assistance which is motivated by relational modelling in frame-based SRL methods (Getoor and Taskar 2007). The core of our approach is the relational database which the user populates in order to prepare the deployment of the system for a specific task. The database and the POMDP generator have a structure that is designed based on experience in the domain of assistance. Content provided by the designer of a particular implementation then encodes the goals, action preconditions, environment states, cognitive model, user and system actions, as well as relevant sensor models, and automatically generates a valid POMDP model of the assistance task being modelled. The strength of the database is

Time step, t	Observations		Task		Behaviour		Ability		System Action (Prompt)
	slot_orange_sensor	bottle1_position_sensor	bottle1_position.in.hand	bottle1_position.in.whitebin	bottle1_position.in.whitebin.pos1	bottle1_position.other	bottle1_position.in.slot_orange	slot_orange.empty	
0	-	-	-	-	-	-	-	-	Rn bottle1 in slot
1	empty	other	-	-	-	-	-	-	Rn slot empty
2	empty	other	-	-	-	-	-	-	Rn slot empty
3	full	in_slot_orange	-	-	-	-	-	-	Rn bottle1 in slot
4	full	in_hand	-	-	-	-	-	-	Rl bottle1 in hand
5	full	in.whitebin	-	-	-	-	-	-	Af bottle1 to pos1
6	full	in.whitebin	-	-	-	-	-	-	Rl bottle1 in bin
7	full	in.whitebin	-	-	-	-	-	-	Af bottle1 to pos1
8	full	in.whitebin.pos1	-	-	-	-	-	-	do nothing

Figure 5: Example simulation in the factory assembly task. The goal in the shown sub-task is to take the bottle, named bottle 1, from the orange slot and to place the bottle in the white bin in pos1.

that it allows constraints to be specified, such that we can verify the POMDP model is, indeed, valid for the task. We demonstrate the method on three assistance tasks: handwashing and toothbrushing for elderly persons with dementia, and on a factory assembly task for persons with a cognitive disability. This demonstration shows that the system, once designed using the relational approach, can be instantiated to create a POMDP controller for an arbitrary intelligent human assistance task. The use of the relational database makes the process of specifying POMDP planning tasks straightforward and accessible to standard computer users.

## 6 Acknowledgements

This research was sponsored by American Alzheimers Association grant numbers ETAC-08-89008 and ETAC-07-58793.

## References

- Åström, K. J. 1965. Optimal control of Markov decision processes with incomplete state estimation. *Journal of Mathematical Analysis and Applications* 10:174–205.
- Boger, J.; Hoey, J.; Poupart, P.; Boutilier, C.; Fernie, G.; and Mihailidis, A. 2006. A planning system based on Markov decision processes to guide people with dementia through activities of daily living. *IEEE Transactions on Information Technology in Biomedicine* 10(2):323–333.
- Burns, A., and Rabins, P. 2000. Carer burden in dementia. *International Journal of Geriatric Psychiatry* 15(S1):S9–S13.
- Chen, L.; Nugent, C. D.; Mulvenna, M.; Finlay, D.; Hong, X.; and Poland, M. 2008. A logical framework for behaviour reasoning and assistance in a smart home. *International Journal of Assistive Robotics and Mechatronics* 9(4):20–34.
- Duke, D.; Barnard, P.; Duce, D.; and May, J. 1998. Syndetic modelling. *Human-Computer Interaction* 13(4):337.
- Getoor, L., and Taskar, B., eds. 2007. *Statistical Relational Learning*. MIT Press.
- Gill, T., and Kurland, B. 2003. The burden and patterns of disability in activities of daily living among community-living older persons. *Journal of Gerontology Series A: Biological Sciences and Medical Sciences* 58A(1):M70–M75.
- Hoey, J., and Grześ, M. 2011. Distributed control of situated assistance in large domains with many tasks. In *Proc. of ICAPS*.
- Hoey, J.; St-Aubin, R.; Hu, A.; and Boutilier, C. 1999. SPUD: Stochastic planning using decision diagrams. In *Proceedings of Uncertainty in Artificial Intelligence*, 279–288.
- Hoey, J.; Poupart, P.; Boutilier, C.; and Mihailidis, A. 2005. POMDP models for assistive technology. In *Proc. AAAI Fall Symposium on Caring Machines: AI in Eldercare*.
- Hoey, J.; Poupart, P.; von Bertoldi, A.; Craig, T.; Boutilier, C.; and Mihailidis, A. 2010. Automated handwashing assistance for persons with dementia using video and a partially observable markov decision process. *Computer Vision and Image Understanding* 114(5):503–519.
- Hoey, J.; Plötz, T.; Jackson, D.; Monk, A.; Pham, C.; and Olivier, P. 2011. Rapid specification and automated generation of prompting systems to assist people with dementia. *To Appear in Pervasive and Mobile Computing*. doi:10.1016/j.pmcj.2010.11.007.
- Kirwan, B., and Ainsworth, L. 1992. *The task analysis guide*. London: Taylor and Francis.
- Mastrogiovanni, F.; Sgorbissa, A.; and Zaccaria, R. 2008. An integrated approach to context specification and recognition in smart homes. In *Smart Homes and Health Telematics*, 26–33. Springer.
- McCluskey, L. 2000. Knowledge engineering for planning roadmap. Working Paper.
- Pentney, W.; Philipose, M.; and Bilmes, J. 2008. Structure learning on large scale common sense statistical models of human state. In *Proc. AAAI*.
- Pham, C., and Olivier, P. 2009. Slice&dice: Recognizing food preparation activities using embedded accelerometers. In *European Conference on Ambient Intelligence*, 34–43. Berlin: Salzburg, Austria: Springer-Verlag.
- Ryu, H., and Monk, A. F. 2009. Interaction unit analysis: A new interaction design framework. *Human-Computer Interaction* 24(4):367–407.
- Salber, D.; Dey, A.; and Abowd, G. 1999. The context toolkit: Aiding the development of context-enabled applications. In *Proc. of the Conference on Human Factors in Computing Systems (CHI)*, 434–441.
- Stammers, R., and Sheppard, A. 1991. *Evaluation of Human Work*. Taylor & Francis, 2nd edition. chapter Chapter 6: Task Analysis.
- Wherton, J. P., and Monk, A. F. 2009. Problems people with dementia have with kitchen tasks: the challenge for pervasive computing. *Interacting with Computers* 22(4):253–266.

# Open-Ended Domain Model for Continual Forward Search HTN Planning

**Dominik Off and Jianwei Zhang**

TAMS, Department of Informatics, University of Hamburg  
Vogt-Kölln-Strasse 30, 22527 Hamburg, Germany  
{off,zhang}@informatik.uni-hamburg.de

## Abstract

Domain models for automated planning systems often rely on the closed world assumption. Unfortunately, the closed world assumption is unreasonable in many real world planning domains. We propose an open-ended domain model based on definite clauses that can be flexibly extended and is able to automatically determine relevant but unknown information. The determination of relevant but unknown information is intended to be the starting point of an active information gathering process which might result in the enablement of additional planning alternatives. This is particularly relevant for situations in which it would otherwise be impossible to find any plan at all. Moreover, we present several knowledge representation constructs that help to deal with the special challenges of open-ended domains. The proposed domain model is mainly intended for hierarchical task network planning systems that generate plans in open-ended domains by means of interleaving planning and knowledge acquisition.

## Introduction

Planning systems have been developed that in principle are efficient enough to solve realistic planning problems in real time. However, “classical” planning approaches fail to generate plans when necessary information is not available at planning time, because they rely on having a complete representation of the current state of the world. (Nau 2007) nicely summarized this problem as follows:

In most automated-planning research, the information available is assumed to be static, and the planner starts with all of the information it needs. In real-world planning, planners may need to acquire information from an information source such as a web service, during planning and execution. This raises questions such as What information to look for? Where to get it? How to deal with lag time and information volatility? What if the query for information causes changes in the world? If the planner does not have enough information to infer all of the possible outcomes of the planned actions, or if the plans must be generated in real time, then it may not be feasible to generate the entire plan in advance. Instead, it may be necessary to interleave planning and plan execution.

We propose a domain model that is able to answer some of the questions raised in the above quotation. More precisely,

the main contributions of this work are:

1. We propose an open-ended domain model—called *Domain Model for Artificial Cognitive Systems (ACogDM)*—based on definite clauses that is able to answer the questions: ‘What to look for?’ and ‘Where to get it?’.
2. We demonstrate how the proposed language of the domain model can be easily extended by additional constructs.
3. We propose extensions of our basic domain model that help to deal with the special requirements (e.g., computational complexity) for open-ended domains.

The proposed domain model is particularly intended for forward search (i.e., forward decomposition) *Hierarchical Task Network (HTN)* (Ghallab, Nau, and Traverso 2004) planning approaches. However, it might also be useful for other approaches.

A domain model for planning is usually composed of information about the state of the domain and information about the possible activities of an agent. We call the former part of a domain model the *state model* and the later part the *activities model*.

## Extendable State Model

Several non-classical planning systems use axiomatic inference techniques to reason about the state of the world (Ghallab, Nau, and Traverso 2004). Often the well investigated definite-clause inference techniques are used. Usually axiomatic inference is supported by calling a theorem prover as a subroutine of the overall planning process. The exploited knowledge representation and theorem proving systems (e.g., PDDL axioms (Thiébaux, Hoffmann, and Nebel 2005)) often rely on the *closed world assumption (CWA)*. However, if we want to enable a planner to reason about unknown information in a partially known domain, then we need a state model and theorem proving system that are not based on the CWA. Particularly, we need an appropriate handling of negation.

As an alternative to implicitly representing negative information (e.g., by using the negation-as-failure semantics (Clark 1987))—as often done by definite-clause theorem provers—it is possible to extend the syntax of definite clauses for the purpose of supporting the explicit representation of negative information. It has been stated in literature

that this approach is often practically infeasible, because of the sheer magnitude of negative facts that would have to be stated (Subrahmanian 1999). We agree with this argumentation, but only under the assumptions that (1) a complete state model should be represented and (2) it is not possible to define a complete representation for local parts of the overall model. However, with respect to the context and objectives of this work neither of these two assumptions is fulfilled, since it is intended to develop an adequate domain model for incompletely known domains which—as introduced later—permits the explicit representation of complete parts at the level of predicates. Thus, we believe that it is reasonable to directly represent negative information in the context of this work.

We use *definite clauses* as the representational basement for state models. The definition and notation of *definite clauses*, *definite goals*, *definite programs* and *substitutions* is borrowed from (Nilsson and Maluszynski 1995). In short, a definite clause is notated as  $A_0 \leftarrow A_1, \dots, A_n$  whereas  $n \geq 0$ . Furthermore,  $\top$  denotes an atomic formula that is true in every interpretation. A definite clause for which  $n = 0$  is notated as  $A_0 \leftarrow \top$ . Moreover, we use ' $\neg$ ' in definite clauses and definite goals as the *negation as (final) failure* operator as introduced by (Clark 1987) and implemented in several prolog systems.

Additionally, we introduce two special kinds of atomic formulas: *literals* and *statements*. If  $f$  is an atomic formula, then we call  $f$  and  $\neg f$  a literal. Furthermore, we call  $st$  a *statement* iff it can be constructed by the following rules:

- $st$  is a literal
- $st = (\neg st')$  and  $st$  is a statement
- $st = (st' \wedge st'')$  and  $st'$  as well as  $st''$  are statements
- $st = (st' \vee st'')$  and  $st'$  as well as  $st''$  are statements

Literals and statements are syntactically defined as atomic formulas for the purpose of reasoning about them in the language of definite clauses. Conceptually a literal essentially is what is known as a literal in first order logic. Similarly, a statement essentially is what is known as a first order logic sentence. Statements—including literals—are always implicitly quantified. Statements in a definite clause are (implicitly) universally quantified and statements that constitute a definite goal are (implicitly) existentially quantified.

Similar to PDDL with PDDL Axioms our state model enables domain experts to express factual (e.g., Bob's mug is in the kitchen) and axiomatic (e.g., Bob's mug is in room  $X_1$  if Bob's mug is on table  $X_2$  and  $X_2$  is in room  $X_1$ ) knowledge. Due to the objective to deal with open-ended domains we additionally support the explicit representation of negative information. Moreover, we support the flexible extension of the representation language of a state model by additional constructs. These additional constructs are intended to constitute higher level (conceptual) knowledge and are called *concepts*. In principle, our state model can be extended to support any conceptual knowledge as long as we can compile this information to the underlying knowledge representation formalism, namely, a set of definite clauses. We exploit this feature in the following part of the paper by

successively adding support for additional concepts that are intended to deal with the special requirements of open-ended domains. For example, we are going to support the explicit representation of subsumption-relations (e.g., 'A mug is an object').

A *state model* is formally defined as follows:

**Definition 1** (state model). A *state model* is a quadruple  $s_M = (F, C, R_D, R_G)$ .  $F$  is a set of literals and  $C$  is a set of atomic formulas such that  $F \cap C = \emptyset$ .  $R_D$  is a set of definite clauses  $l \leftarrow s$  such that  $l$  is a literal and  $s$  is a statement.  $R_G$  is a set of definite clauses.  $s_M^{dp} = \{f \leftarrow \top \mid f \in F \cup C\} \cup R_D \cup R_G$  is the definite program constituted by the state model.

A state model  $s_M$  is represented by the four sets  $F$ ,  $C$ ,  $R_D$ ,  $R_G$ .  $F$  represents a set of facts about the state of a domain.  $C$  contains additional conceptual knowledge.  $R_D$  represents domain-specific rules (i.e., domain-specific axiomatic knowledge). In contrast,  $R_G$  represents generic (i.e., domain-independent) rules (e.g.,  $(A \wedge B)$  holds if  $A$  and  $B$  hold).  $F$ ,  $C$  and  $R_D$  are intended to be specified by a domain expert in order to model the state of a certain domain.  $R_G$ , however, represents generic rules that are defined together with the supported state model language constructs in order to be able to map these constructs to the level of definite clauses.

The fact that a state model constitutes a definite program has the advantage that the semantics of a state model is based on the well-known semantics of a definite program. From a more practical perspective, we can additionally benefit from the actuality that several highly optimized prolog implementations are available that can automatically determine whether a definite goal can be proved or not.

Based on the semantics of a definite program we can define the derivability of an atomic formula as follows:

**Definition 2** (derivable). An atomic formula  $f$  is *derivable* with respect to a state model  $s_M$  and a grounding substitution  $\sigma$  (denoted as  $s_M \vdash_\sigma f$ ) iff  $f\sigma$  is a logical consequence<sup>1</sup> of  $s_M^{dp}$ .

In order to specify the semantics of statements we add the following generic rules to the set  $R_G$  of a state model  $s_M = (F, C, R_D, R_G)$ :

$$(st \wedge st') \leftarrow st, st' \quad (\text{GR1})$$

$$(st \vee st') \leftarrow st \quad (\text{GR2})$$

$$(st \vee st') \leftarrow st' \quad (\text{GR3})$$

$$\neg \neg st \leftarrow st \quad (\text{GR4})$$

$$\neg (st \wedge st') \leftarrow (\neg st \vee \neg st') \quad (\text{GR5})$$

$$\neg (st \vee st') \leftarrow (\neg st \wedge \neg st') \quad (\text{GR6})$$

These rules determine the semantics of statements. In particular, the handling of the introduced negation operator ' $\neg$ ' is specified. Furthermore, please note that it directly

<sup>1</sup>More precisely, this means that  $f$  is a member of the least Herbrand model  $M_{s_M^{dp}}$  (Nilsson and Maluszynski 1995, Theorem 2.16).



follows from Definition 2 that a statement  $st$  is derivable with respect to a state model  $s_M$  and a substitution  $\sigma$  iff  $st\sigma$  is a logical consequence of  $s_M^{dp}$ . For the purpose of avoiding misunderstandings we would like to emphasize again that statements are syntactically treated as atomic formulas, but semantically constitute a first order logic sentence.

As already pointed out, a domain modeller has the opportunity to define domain-specific axioms. Axioms are known to be an important feature of domain languages (Thiébaux, Hoffmann, and Nebel 2005). Two example axioms are defined as follows:

$$in\_room(O, R) \leftarrow on(O, T), in\_room(T, R) \quad (DR1)$$

$$\begin{aligned} neg\ in\_room(O, R) &\leftarrow in\_room(O, R2), \\ &R2 \neq R \end{aligned} \quad (DR2)$$

DR1 represents the fact that an object is in a room  $R$  if it is lying on a table which is in room  $R$ . DR2 is an example for the explicit representation of negative information. It represents the fact that an object can only be in one room at a given point in time.

Due to the fact that we support the explicit representation of negative information it is in principle possible to construct a *syntactically inconsistent*<sup>2</sup> state model. This means that it is possible to construct a state model where  $st$  and  $neg\ st$  are derivable. Indeed, it is desirable to support domain modellers with software tools in order to prevent the creation of inconsistent state models. However, semantic inconsistencies may also occur in CWA based representations. For example, one can create a CWA based state model such that  $open(door1)$  and  $closed(door1)$  is derivable. Thus, one also has to deal with inconsistency in CWA based models. Furthermore, note that the explicit representation of negation by means of the definite clause  $neg\ open(door1) \leftarrow closed(door1)$  has the advantage that it would make it possible to detect the semantic inconsistency via syntactic techniques. However, we will not further address that problem here, since dealing with consistency is not in the focus of this work. The fact that a state model  $s_M$  is consistent is denoted as  $c(s_M)$ . In the following part of this paper it is implicitly always assumed that a state model is consistent.

### Conceptualizing Open-Endedness

CWA based knowledge representation and reasoning systems (e.g., prolog) can in principle also be used in open-ended domains. Nevertheless, in open-ended domains one has to consider that it is possible that true instances of a statement “exist” but cannot be derived due to a lack of knowledge. CWA based approaches are—by definition—unable to reason about unknown (i.e. non-derivable) but possibly true information. More precisely, it is unfeasible for CWA based systems to distinguish between instances of statements that cannot be derived because the existence is impossible and instances of a statement that might be derivable if additional information about the state of the domain were available.

<sup>2</sup>See (Nguyen 2008) for more details about syntactic and semantic inconsistencies.

*Example 1* For example, let us assume that the only literals that can be derived from the state model  $s_M$  of an agent are  $mug(bobs\_mug)$  and  $in\_room(bobs\_mug, kitchen)$ . If one would try to derive whether a true instance of  $mug(X) \wedge color(X, red)$  exists with respect to  $s_M$ , then the only information a CWA based reasoner can provide is that such an instance cannot be derived. Nevertheless, in principle there are two possible situations in which an instance of this statement exist. It might be (1) possible that Bob’s mug is red or (2) it might be possible that there is an additional (i.e., non-derivable) mug that is red. For the purpose of also exemplifying the case where the existence of an instance is impossible, let us take a look at the statement  $in\_room(bobs\_mug, office)$ . Once again, the only thing a CWA based reasoner can tell us about the literal is the fact that it is not derivable. However, in this case the existence of a true instance is impossible if one makes the reasonable assumption that Bob’s mug cannot be in two different rooms at same point in time as specified by DR2.

Summing up, the CWA leads to a strong limitation that makes it hard to reason about unknown information. The objective of the proposed open-ended domain model is to enable the distinction between situations in which the existence of a non-derivable instance of a statement is impossible and situations in which additional information might make non-derivable instances derivable. Moreover, in the latter case the domain model should make it possible to derive all situations in which the existence of an additional instance is possible. If we want to enable such a reasoning, then we need an open-ended domain model. We propose an open-ended domain model that is based on the following three concepts: a *F-extension*; an *open-ended literal*; and a *possibly-derivable statement*.

For the purpose of reasoning about open-ended domains we have to reason about possible extensions of a state model. Here we only consider extensions that are constituted by adding factual knowledge (i.e., a set of literals) to a state model. These extensions are called *F-extensions* and are formally conceptualized as follows:

**Definition 3** (F-extension). A state model  $s'_M = (F', C, R_D, R_G)$  is called an *F-extension* of  $s_M = (F, C, R_D, R_G)$  (denoted as  $s_M \sqsubseteq_F s'_M$ ) iff  $F \subseteq F'$  and  $c(s_M) \Rightarrow c(s'_M)$ .

In other words, one can create an F-extension of a state model by adding literals such that a consistent world model stays consistent. We denote the set of all instances of a statement  $st$  that are derivable with respect to a state model  $s_M$  as  $\tilde{\vdash}(s_M, st)$  respectively as  $\tilde{\vdash}(st)$  if the respective state model is apparent. Furthermore, we call literals for which the existence of non-derivable instances is possible *open-ended*:

**Definition 4** (open-ended literal). A literal  $l$  is called *open-ended* w.r.t. a state model  $s_M$  (denoted as  $l^\infty$ ) iff it is possible that there is an instance  $l\sigma$  of  $l$  and a state model  $s'_M$  such that  $s_M \sqsubseteq_F s'_M$ ,  $l\sigma \notin \tilde{\vdash}(s_M, l)$  and  $l\sigma \in \tilde{\vdash}(s'_M, l)$ .

Please note that for a ground literal the following holds:

**Remark 1.** If  $l$  is ground, then  $l$  is open-ended iff  $\not\vdash l$  and  $\not\vdash neg\ l$  holds.



Let us recall the situation of Example 1 in order to exemplify the concept of an open-ended literal.  $mug(X)$  and  $color(red, X)$  are examples of open-ended literals, because the existence of non-derivable mugs and red things is possible. In contrast,  $mug(bobs\_mug)$  is not open-ended, since the only possible instance is already derivable.

Let  $ground(l)$  be a meta-predicate that holds iff  $l$  is ground and  $non-ground(l)$  be a meta-predicate that holds iff  $l$  is non-ground. The following two clauses constitute a first attempt to specify an open-ended literal by means of a set of definite-clauses:

$$l \models \leftarrow non-ground(l) \quad (GR7)$$

$$l \models \leftarrow ground(l), \not\models l, \not\models (neg\ l) \quad (GR8)$$

In other words, a literal  $l$  is open-ended if it is non-ground (GR7); or if it is ground and neither  $l$  nor  $neg\ l$  can be derived (GR8).

Based on the definition of an open-ended literal, a *possibly-derivable statement* is defined as follows:

**Definition 5** (possibly-derivable statement). A statement  $st$  is possibly-derivable w.r.t. to a state model  $s_M$  and a set of open-ended literals  $L_x$  (denoted as  $\Diamond(st, L_x)$ ) iff the existence of a new instance  $l\sigma$  for each  $l \in L_x$  implies the existence of a new instance  $st\sigma$  of  $st$ .

A possibly-derivable statement constitutes the partition of a logical statement into a derivable and an open-ended part (i.e., a set of open-ended literals). This partition determines what additional information is necessary in order to derive an additional (i.e., non-derivable w.r.t. the state model at hand) instance of a given statement. Note that there may be more than one way to partition a statement into a derivable and an open-ended part.

Let us assume that we have the same state model  $s_M$  as introduced in Example 1 and would like to know whether the statement  $st = mug(X) \wedge color(X, red)$  is possibly-derivable (i.e., we are looking for a red mug). In this example there are two different situations in which  $st$  is possibly-derivable. In the first situation,  $X$  is substituted with  $bobs\_mug$  and  $st$  is possibly-derivable with respect to  $s_M$  and the resulting set of open-ended literals  $\{color(bobs\_mug, red)\}$ . In the second situation, we exploit the fact that there might exist an unknown red mug and  $st$  is possibly-derivable with respect to  $s_M$  and the resulting set of open-ended literals  $\{mug(X), color(X, red)\}$ .

Let  $literal(l)$  be a meta-predicate that holds iff  $l$  is a literal. In order to be able to derive possibly-derivable statements we introduce the following generic rules:

$$\Diamond(st, L_x) \leftarrow \Diamond(st, \emptyset, L_x) \quad (GR9)$$

$$\Diamond(st, L_x, L_x) \leftarrow literal(st), st, \forall l \in L_x : l \models \quad (GR10)$$

$$\Diamond(st, L_x, L_x \cup \{st\}) \leftarrow literal(st), st \models \quad (GR11)$$

$$\begin{aligned} \Diamond((st \wedge st'), L_x, L_x') &\leftarrow \Diamond(st, L_x, L_x''), \\ &\quad \Diamond(st', L_x'', L_x') \end{aligned} \quad (GR12)$$

$$\Diamond((st \vee st'), L_x, L_x') \leftarrow \Diamond(st, L_x, L_x') \quad (GR13)$$

$$\Diamond((st \vee st'), L_x, L_x') \leftarrow \Diamond(st', L_x, L_x') \quad (GR14)$$

$$\begin{aligned} \Diamond(neg(st \wedge st'), L_x, L_x') &\leftarrow \\ \Diamond((neg\ st \vee neg\ st'), L_x, L_x') &\quad (GR15) \end{aligned}$$

$$\begin{aligned} \Diamond(neg(st \vee st'), L_x, L_x') &\leftarrow \\ \Diamond((neg\ st \wedge neg\ st'), L_x, L_x') &\quad (GR16) \end{aligned}$$

GR10 and GR11 specify under what conditions a literal is possibly-derivable. The general idea is that a literal is possibly-derivable if it is derivable or open-ended. Thus, every open-ended literal is possibly-derivable, because for every open-ended literal it is possible that there is a consistent extension of the current domain model so that it is derivable w.r.t. this extension. Note that a (non-ground) literal can be both derivable and open-ended.  $L_x$  denotes the set of open-ended literals of the previous part of a statement and initially is empty (see GR9). Including  $L_x$  into the recursive definition is necessary in order to consider the possible dependencies between different parts of a statement. To be more precise, it has to be ensured that all literals that have been "chosen" to be in the open-ended part of a statement stay open-ended after additional substitutions. This is exactly what is done in GR10 by means of ensuring that possible substitutions that are necessary in order to derive an instance of  $st$  do not affect the open-endedness of the literals in  $L_x$ . Besides the correct handling of the set of open-ended literals, GR13 - GR16 essentially describe well-known rules of first order logic.

## Continual Planning in Open-Ended Domains

As already mentioned, ACogDM is developed for forward decomposition HTN planners (e.g., SHOP (Nau et al. 1999) or SHOP2 (Nau et al. 2003)). In this section we briefly motivate and explain how the proposed domain model can be combined with such a planner so that the combination constitutes a continual planning system.

Forward decomposition HTN planners choose between a set of *relevant* (Ghallab, Nau, and Traverso 2004) methods or planning operators (i.e., actions) that can be in principle applied to the current *task network*. In the context of this work preconditions of action or methods are represented by definite goals of the form ' $\leftarrow st$ ' such that  $st$  is a statement (e.g.,  $\leftarrow (mug(X) \wedge neg\ in\_room(X, kitchen))$ ). If it does not lead to ambiguity, then we will omit the leading ' $\leftarrow$ ' of a definite goal. A relevant method or planning operator can actually be applied if and only if its precondition  $p$  holds (i.e., an instance  $p\sigma$  is derivable) with respect to the given domain model. Therefore, we define the set of *relevant preconditions* with respect to a given *planning context* (i.e., a domain model and a task network) to be the set of all preconditions of relevant methods or planning operators. A HTN planner cannot continue the planning process in situations where no relevant precondition is derivable with respect to the domain model at hand. The notation of a relevant precondition is a first step to determine relevant

extensions of a domain model, since only domain model extensions that make the derivation of an additional instance of a relevant precondition possible constitute an additional way to continue the planning process. All other possible extensions are irrelevant, because they do not imply additional planning alternatives.

The general idea is to adapt a forward decomposition HTN planner such that the behaviour is not changed as long as sufficient information is available in order to generate a plan. However, if necessary information is missing, then the planning process is stopped and a partial plan prefix and a set of open-ended literals of a relevant and possibly-derivable precondition is returned. If the planner stops the planning process due to a lack of knowledge, then the set of open-ended literals constitute a relevant extension of the domain model that would make it possible to continue the planning process. Hence, a planner can answer the question “What to look for?” as follows: Look for non-derivable instances of the open-ended part (i.e., a set of open-ended literals) of possibly-derivable and relevant preconditions. For example, if want a planner to perform the task “Deliver Bob’s mug into the kitchen”, but the fact whether the kitchen door is open or closed cannot be derived from the domain model, then a planner returns a partial plan (e.g.,  $[pick\_up(bobs\_mug), \dots]$ ) and a set of open-ended literals (e.g.,  $\{open(kitchen\_door)\}$ ). Based on that, a planner can try to generate and execute a plan that acquires a non-derivable instance for each open-ended literal (e.g., try to acquire whether the kitchen door is open). Subsequently, a planner can continue the planning process based on the updated domain model. By this means a planner can automatically switch between planning and acting such that missing information can be acquired by means of active information gathering.

### Additional State Model Constructs

Supporting the representation on a conceptual meta-level—in contrast to representing knowledge on the level of definite clauses—has the advantage that it eases the knowledge engineering process, since domain experts can represent knowledge on a higher abstraction level that is often closer to the way they think about the domain.

In this section, we are going to extend the state model by additional concepts that make it possible to reduce the open-endedness of the state model. The general idea is that one can reduce the open-endedness by means of exploiting additional domain knowledge such that the number of open-ended literals can be reduced. For example, according to Remark 1 one could deduce that an open-ended and ground literal  $l$  is not open-ended if additional domain knowledge would make it possible to derive  $l$  or  $\neg l$ .

With the current state model (i.e., the state model constituted by GR1 - GR16) every non-ground literal is open-ended (see GR7). To put it another way, we assume that we never know all instances of a non-ground literal. However, this might not always be the case. On the conceptual—or semantical—level domain constraints can limit the number of possible instances of a statement. For example, let us assume that the literal

$in\_room(bobs\_mug, office)$  is derivable. In this case the non-ground literal  $in\_room(bobs\_mug, X)$  is not open-ended (i.e., no additional instance is possible) if we assume that an object can only be in one room at a given point in time. In order to be able to express these kinds of constraints we extend the language of the state model by constructs of the form  $i_{max}(l, n, c)$  such that  $l$  is a literal,  $n \in \mathbb{N} \cup \{\infty\}$  and  $c$  is a statement.  $i_{max}(l, n, c)$  specifies that the literal  $l$  can maximally have  $n$  ground instances if  $c$  holds. In order to “ground” this additional construct to the level of definite clauses we have to add the following rules:

$$i_{max}(l, n) \leftarrow i_{max}(l, n, c), c \quad (GR17)$$

$$i_{max}(l, \infty) \leftarrow non\_ground(l), \neg i_{max}(l, n, X_1) \quad (GR18)$$

$$i_{max}(l, 1) \leftarrow ground(l) \quad (GR19)$$

Now we can formulate an advanced version of (GR7) as follows:

$$l \models \leftarrow non\_ground(l), i_{max}(l, n), n < |\tilde{\vdash}(l)| \quad (GR20)$$

In other words, a literal is open-ended if the number of derivable instances is less than the number of maximum instances.

A less flexible, but easier way to define the maximum number of instances for a subset of non-ground literals is based on the *instantiation scheme* of a literal.

A literal or a term is called *duplicate-variable-free* iff it does not contain two identical variables. For example,  $p(X, Y)$  is duplicate-variable-free and  $p(X, X)$  is not duplicate-variable-free. For duplicate-variable-free terms and literals we define a corresponding *instantiation scheme* as follows:

**Definition 6** (instantiation scheme). Let  $g$  be a duplicate-variable-free term or literal. The *instantiation scheme*  $g^\rho$  of  $g$  is defined as follows:

$$g^\rho := \begin{cases} ground & \text{if } g \text{ is ground} \\ var & \text{if } g \text{ is a variable} \\ f(u_1^\rho, \dots, u_m^\rho) & \text{else if } g = f(u_1, \dots, u_m) \end{cases}$$

An instantiation scheme abstracts from the concrete arguments of a literal by replacing variables with the constant *var* and ground terms with the constant *ground*. We restrict instantiation schemes here to duplicate-variable-free terms and literals, because the multiple occurrence of the same variable imposes additional constraints that otherwise would be unintentionally abstracted away. Moreover, from the knowledge engineering perspective we wanted to keep the definition of an instantiation scheme simple, since instantiation schemes are intended to be specified by a human domain expert. Explicitly representing possible constraints that result from duplicate variables in a literal would make the representation significantly more difficult while only being necessary for the minority of literals. Additionally, please note that Definition 6 can also be applied to negative literals, since the negation operator is technically a “normal” predicate. Let  $\mathcal{L}^\rho := \{l^\rho | l \in \mathcal{L}\}$ . The maximum number of possible instances with respect to an instantiation scheme is defined by the function  $i_{max-\rho} : \mathcal{L}^\rho \rightarrow \mathbb{N} \cup \infty$ . In

ACogDM, we can define the possible number of instances with respect to a representation scheme with atomic formulas of the form  $i_{max-p}(scheme, n)$  such that  $scheme$  is an instantiation scheme and  $n \in \mathbb{N} \cup \{\infty\}$ . In order to support these constructs we add the following generic rule to the state model:

$$i_{max}(l, n) \leftarrow i_{max-p}(l^p, n) \quad (GR21)$$

For example, the fact that an object can only be in one room at a given point in time can be easily represented by the atomic formula  $i_{max-p}(in\_room(ground, var), 1)$ . However, now we have a semantically redundant representation because the conceptually same actuality is already specified by the domain specific rule DR2. Note that both representations have been introduced for different technical reasons. DR2 solely makes it possible to derive that all statements of the form  $neg\ in\_room(obj, r)$  are true if it is known that  $in\_room(obj, r')$  and  $r' \neq r$  hold. In contrast,  $i_{max-p}(in\_room(ground, var), 1)$  solely makes it possible to deduce that all statements with the instantiation scheme  $in\_room(ground, var)$  can only have one instance.

We can omit redundancies introduced by  $i_{max-p}$  via adding generic rules. For the purpose of achieving this we first introduce the *sub-scheme*-relation as follows:

**Definition 7** (sub-scheme). An instantiation scheme  $s$  is called a *sub-scheme* of an instantiation scheme  $s'$  (denoted as  $s \leq s'$ ) iff one of the following holds:

- $s' = var$  ;
- $s = ground \wedge (s' = ground \vee s' = var)$  ;
- or  $s = g(\alpha_1, \dots, \alpha_n)$  and  $s' = g(\beta_1, \dots, \beta_n)$  and for all  $1 \leq i \leq n$  it holds that  $\alpha_i \leq \beta_i$ .

The *sub-scheme*-relation constitutes an ordering on instantiation schemes. We are interested in this ordering, since it is related to  $i_{max-p}$  as stated by the following proposition:

**Proposition 1.** If  $l$  and  $l'$  are duplicate-variable-free literals, then the following holds:  $l^p \leq l'^p \Rightarrow i_{max-p}(l) \leq i_{max-p}(l')$ .

We define the *lift* of a duplicate-variable-free literal or term with respect to a compatible instantiation scheme as follows:

**Definition 8** (lift). Let  $g$  be a duplicate-variable-free literal or term,  $g'^p$  be an instantiation scheme such that  $g^p \leq g'^p$  and  $X^*$  denote a new (i.e., unused) variable. The *lift* of  $g$  w.r.t.  $g'^p$  is defined by the function  $\rho_{\uparrow}$  as follows:

- $\rho_{\uparrow}(g, g'^p) := g$ ; if  $g^p = g'^p$
- $\rho_{\uparrow}(g, g'^p) := X^*$ ; if  $g'^p = var$  and  $g$  is not a variable
- $\rho_{\uparrow}(g, g'^p) := f(\rho_{\uparrow}(u_1, u_1'^p), \dots, \rho_{\uparrow}(u_m, u_m'^p))$ ; if  $g = f(u_1, \dots, u_m)$  and  $g'^p = f(u_1'^p, \dots, u_m'^p)$

Lifting a literal or a term with respect to an instantiation scheme  $g^p$  essentially means to replace ground terms by new variables such that the instantiation scheme of the resulting literal is  $g^p$ . For example, lifting  $in\_room(bobs\_mug, office)$  with respect to the instantiation scheme  $in\_room(ground, var)$  results in  $in\_room(bobs\_mug, X)$ . Now we can propose the following:

**Proposition 2.** For each duplicate-variable-free literal  $l$ ,  $neg\ l$  is derivable w.r.t. a state model  $s_M$  if the following holds:

1.  $\neg \exists_{\sigma} : s_M \vdash_{\sigma} l$ ; ( $l$  is not derivable)
2.  $\exists_{l'^p \in \mathcal{L}^p} : l^p \leq l'^p \wedge |\tilde{\vdash}(s_M, \rho_{\uparrow}(l, l'^p))| = i_{max-p}(l'^p)$

This means that we can derive  $neg\ l$  if  $l$  is not derivable and it exists an instantiation scheme that is more general than the instantiation scheme of  $l$  for which all possible instances are already derivable. Proposition 2 constitutes a rule that enables us to now derive, based on the definition of  $i_{max-p}$ , that something cannot hold. We can now represent Proposition 2 as the following rule:

$$neg\ l \leftarrow \neg \vdash l, i_{max-p}(l'^p, n), l^p \leq l'^p, |\tilde{\vdash}(\rho_{\uparrow}(l, l'^p))| = n \quad (GR22)$$

For example, we can now derive  $neg\ in\_room(bobs\_mug, office)$  if  $in\_room(bobs\_mug, kitchen)$  and  $i_{max-p}(in\_room(ground, var), 1)$  are derivable. Thus, we can now omit the domain specific rule DR2 in order to remove the redundancy without losing derivable information.

We proposed an open-ended state model where all statements are by default interpreted based on the *open world assumption* (OWA). Nevertheless, in order to combine the best of both worlds it is possible to define on the predicate level if a literal should be interpreted based on the CWA or the OWA. This property of a predicate is called the *interpretation model* of a predicate and can either be OWA or CWA. For example, imagine a predicate `connection(R1, D, R2)` which describes that room R1 is connected via door D with room R2. The relation that is represented by this predicate is rather static, thus even in dynamic unstructured environments it is possible to equip an artificial agent a priori with all true ground instances of this relation. In this situation it would be reasonable to define the interpretation model of the `connection` predicate as CWA. This definition implies  $neg\ connection(R1, D, R2)$  holds iff. `connection(R1, D, R2)` cannot be derived—which in fact is the negation-as-failure semantics as introduced by (Clark 1987). Predicate based CWAs reduces the lack of knowledge and can significantly improve the performance of the plan generation and knowledge acquisition process.

A predicate is symbolically represented as  $[name/n]$  where *name* is the name of the predicate and *n* denotes the arity. The predicate of a literal  $l$  is denoted as  $l^e$ . The fact that a predicate is interpreted with respect to the CWA is represented by atomic formulas of the form  $cwa([name/n])$ . Thus, all predicates that are not defined as being interpreted with respect to the CWA are—by default—interpreted based on the OWA. In order to support CWAs at the level of predicates we only have to add the following rule:

$$neg\ l \leftarrow cwa(l^e), \neg l \quad (GR23)$$

Another featured knowledge representation construct is the explicit definition of *subsumption*-relations between literals. More precisely, subsumption is a relation between

concepts which are constituted by literals. The subsumption relation can only be defined for literals that have the same arity. Let  $X_i$  ( $1 \leq i \leq n$ ) be variables and  $p(X_1, \dots, X_n)$  and  $p'(X_1, \dots, X_n)$  be literals. The fact that a literal  $p(X_1, \dots, X_n)$  is conceptually subsumed by a literal  $p'(X_1, \dots, X_n)$  is denoted as  $p(X_1, \dots, X_n) \sqsubseteq p'(X_1, \dots, X_n)$ . Information about subsumption relations can now be exploited as follows:

$$l \leftarrow l' \sqsubseteq l' \quad (\text{GR24})$$

In other words, a literal is derivable if there is a subconcept that is derivable. Moreover, it can be easily shown that the following holds:

$$\text{neg } l' \sqsubseteq \text{neg } l \leftarrow l \sqsubseteq l' \quad (\text{GR25})$$

Similarly, the fact that the literals  $p(X_1, \dots, X_n)$  and  $p'(X_1, \dots, X_n)$  are disjunct is denoted as  $p(X_1, \dots, X_n) \sqcap p'(X_1, \dots, X_n)$ . Knowledge about the disjointness is exploited by the following inference rule:

$$\text{neg } l \leftarrow l \sqcap l', l' \quad (\text{GR26})$$

## Knowledge Acquisition

We already proposed an answer to the question: “What to look for?”. In this section we briefly survey our answer to the second initial question: “Where to get it?”.

The central concept to answer this question is an *external knowledge source*. Anything that is able to provide additional information about the world (e.g., perception, human-computer interaction, low-level reasoning and planning) might serve as an external knowledge source. Of course, an external knowledge source has to conform to a corresponding interface in order to enable the planner to submit queries to various external sources in an uniform manner.

Artificial agents—especially robots—can usually acquire information from a multitude of sources. Sources may differ strongly from each other in terms of the type of information they can provide and other non-functional characteristics (e.g., acquisition cost, reliability, degree of necessary human interaction, world altering effects). Here we restrict ourselves to the following two major properties of an external knowledge source: the type of information it in principle can provide, and how expensive it is to answer a certain question. Let  $ks$  be the symbolic representation of a knowledge source and  $l$  be a literal. We further extend the representation language of our state model by constructs of the form *applicable<sub>ks</sub>*( $ks, l$ ) in order to denote that  $ks$  is in principle able to provide new instances of  $l$ .

Now we can answer the question “Where to get it?” with: “You can get the desired information from an applicable knowledge source”.

How expensive it is to acquire new information from external sources strongly depends on: the information one is looking for, the chosen knowledge source, and the current situation. The expense that takes these three issues into account is called the *acquisition cost*. The fact that the cost to

acquire a new instance of a literal  $l$  from a knowledge source  $ks$  is  $c$  is specified by constructs of the form *ac*( $ks, l, c$ ).

Based on the applicability of external knowledge sources and the expected acquisition cost, a planner can decide (i.e., plan) how to acquire relevant information (i.e., the open-ended part of a relevant precondition) from external knowledge sources.

## Activities Model

The activities model of ACogDM contains knowledge about *planning steps* and *tasks*. The term *planning step* is used as an abstraction of *planning operators* (i.e., actions), *HTN methods* and *High-level actions (HLAs)*. Planning operators and HTN methods are mainly defined as in (Ghallab, Nau, and Traverso 2004) and HLAs are mainly defined as in (Russell and Norvig 2010). Additionally, it is possible to specify a cost for each planning step. Please note that specifying the cost of an action or an HTN method is not a new idea and, for example, also supported by SHOP2 (Nau et al. 2003).

In many domains there are tasks respectively goals for which a lot of possible solutions exist. However, in the light of additional domain knowledge one can often significantly reduce the number of possible plans and thereby reduce the computational effort. Continual planning approaches benefit to a special degree from the reduction of alternative solutions, because less alternatives usually also means less unnecessary execution of planning operators. And execution is often (e.g., in robotics) a time intensive process.

For forward-search HTN planning (e.g., SHOP (Nau et al. 1999)) we propose to support additional domain knowledge which makes it possible to reduce the number of alternative plans for a given task.

*Example 2* For example, let us assume that we instruct a robot to pick up Bob’s mug from the kitchen table (*pick\_up(bobs\_mug, kitchen\_table)*) and there is exactly one HTN methods that always decomposes this task into the subtasks [*goto(kitchen\_table)*, *grasp(bobs\_mug)*]. Moreover, let us assume that there are in principle several different ways to go into the kitchen. Nevertheless, how the robot actually performs the task of going into the kitchen does not affect the task of grasping Bob’s mug. This information can be exploited in a situation where a planner successfully generated a plan for the purpose of getting into the kitchen and then realizes that it is impossible to grasp Bob’s mug (e.g., because the mug is in another room). In this situation it obviously does not make sense to backtrack and try to find an alternative plan for the task *goto(kitchen\_table)*. A planner that knows that these tasks can be solved independently can first generate a sufficiently good plan for *goto(kitchen)*, cut alternative decompositions for *goto(kitchen)* and then plan to grasp Bob’s mug.

In ACogDM it is possible to express the interdependency of subtasks by task lists of the form [ $\{t_1, \dots, t_m\}, \dots, \{t_{m+k}, \dots, t_{m+k+j}\}$ ] such that one can plan individually for each set of tasks embraced by ‘{}’. For Example 2 one can represent the subtasks of *pick\_up(bobs\_mug, kitchen)* as [*goto(kitchen\_table)*], [*grasp(bobs\_mug)*]].

## Related Work

Most existing automatic theorem proving or knowledge representation and reasoning systems, including planning domain models, do not systematically analyze failed inferences or queries. The only known exception is the “WhyNot” tool of PowerLoom (Chalupsky and Russ 2002) which tries to generate a set of plausible partial proofs for failed queries. Nevertheless, “WhyNot” is rather a debugging tool that tries to generate human readable explanations that describe why the overall reasoning process failed. Therefore, this approach is not adequate for the objectives of this work.

Exploiting local closed world assumptions is also featured by PowerLoom (Chalupsky, MacGregor, and Russ 2010) and has also been proposed by (Etzioni, Golden, and Weld 1997).

The approach of (Dornhege et al. 2009) also makes it possible to integrate external components into the planning process. However, integration is not done autonomously (i.e., by reasoning on the need to acquire information from external sources), but predefined in the domain description.

Converting knowledge from one representation scheme to another in general and particularly converting an ontology (e.g., a description logic based representation) to a definite program is not a new idea. The integration of description logic and logic programming is currently an active research topic (Motik and Rosati 2007). How an OWL based ontology can be converted to prolog programs is described in (Samuel et al. 2008). Furthermore, it is possible to express a subset of OWL directly as a logic program, namely, a *description logic program* (Hitzler, Studer, and Sure 2005). Description logic has been used in many different aspects in planning systems (Gil 2005). An approach that combines HTN planning and description logic reasoning is described by (Hartanto and Hertzberg 2008).

## Discussion and Conclusion

We have presented an open-ended domain model based on definite clauses that can be extended by additional constructs. The proposed conceptualization of open-endedness allows us to automatically determine relevant but unknown information which makes additional planning alternatives possible. In particular, it often makes it possible to find any plan at all if insufficient information is a priori available.

We observe definite clauses to be a solid representational basement that makes it relatively easy to extend the state model language by additional constructs. Furthermore, we define several additional state model constructs that help to deal with the special challenges of open-ended domains as well as exemplify how a basic state model can be successively extended. The additional state model constructs so to speak reduce the “open-endedness” of a state model by enabling it to rule out possible extensions of a state model.

## Acknowledgements

This work is funded by the DFG German Research Foundation (grant #1247) – International Research Training Group CINACS (Cross-modal Interactions in Natural and Artificial Cognitive Systems)

## References

- Chalupsky, H., and Russ, T. A. 2002. Whynot: Debugging failed queries in large knowledge bases. In *AAAI/IAAI*, 870–877.
- Chalupsky, H.; MacGregor, R. M.; and Russ, T. 2010. *PowerLoom Manual (Version 1.48)*. University of Southern California, Information Sciences Institute.
- Clark, K. L. 1987. Negation as failure. *Logic and databases* 293–322.
- Dornhege, C.; Eyerich, P.; Keller, T.; Trüg, S.; Brenner, M.; and Nebel, B. 2009. Semantic attachments for domain-independent planning systems. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, 114–121. AAAI Press.
- Etzioni, O.; Golden, K.; and Weld, D. S. 1997. Sound and efficient closed-world reasoning for planning. *Artif. Intell.* 89(1-2):113–148.
- Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning Theory and Practice*. Elsevier Science.
- Gil, Y. 2005. Description logics and planning. *AI Magazine* 26(2):73–84.
- Hartanto, R., and Hertzberg, J. 2008. Fusing dl reasoning with htn planning. In *KI*, 62–69.
- Hitzler, P.; Studer, R.; and Sure, Y. 2005. Description logic programs: A practical choice for the modelling of ontologies. In *1st Workshop on Formal Ontologies Meet Industry, FOMI’05, Verona, Italy, June 2005*.
- Motik, B., and Rosati, R. 2007. A faithful integration of description logics with logic programming. In *IJCAI*, 477–482.
- Nau, D. S.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. Shop: Simple hierarchical ordered planner. In *IJCAI*, 968–975.
- Nau, D.; Au, T. C.; Ilghami, O.; Kuter, U.; Murdock, W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *Journal on Artificial Intelligence Research* 20.
- Nau, D. S. 2007. Current trends in automated planning. *AI Magazine* 28(4):43.
- Nguyen, N. T. 2008. *Advanced Methods for Inconsistent Knowledge Management*. Springer.
- Nilsson, U., and Maluszynski, J. 1995. *Logic, Programming, and PROLOG*. New York, NY, USA: John Wiley & Sons, Inc.
- Russell, S. J., and Norvig, P. 2010. *Artificial Intelligence: A Modern Approach*. Prentice Hall.
- Samuel, K.; Obrst, L.; Stoutenburg, S.; Fox, K.; Franklin, P.; Johnson, A.; Laskey, K. J.; Nichols, D.; Lopez, S.; and Peterson, J. 2008. Translating owl and semantic web rules into prolog: Moving toward description logic programs. In *Theory and Practice of Logic Programming*, volume 8, 301–322.
- Subrahmanian, V. S. 1999. Nonmonotonic logic programming. *IEEE Trans. Knowl. Data Eng.* 11(1):143–152.
- Thiébaux, S.; Hoffmann, J.; and Nebel, B. 2005. In defense of pddl axioms. *Artificial Intelligence* 168(1-2):38–69.

# Taking Advantage of Domain Knowledge in Optimal Hierarchical Deepening Search Planning

Pascal Schmidt<sup>†‡</sup>, Florent Teichtel-Königsbuch<sup>†</sup> and Patrick Fabiani<sup>†</sup>

<sup>†</sup>ONERA - The French Aerospace Lab

F-31055, Toulouse, France

firstname.lastname@onera.fr

<sup>‡</sup>Université de Toulouse

F-31000, Toulouse, France

## Abstract

In this paper, we propose a new algorithm, named HDS for Hierarchical Deepening Search, to solve large structured classical planning problems using the divide and conquer motto. A large majority of planning problems can be easily and recursively decomposed in many easier subproblems, what is efficiently exploited for instance by domain-independent approaches such as landmark techniques or domain-knowledge formalisms like Hierarchical Task Networks (HTN). We propose to exploit domain knowledge in the form of HTNs to guide the generation of multiple levels of subgoals during the search. Compared with traditional HTN approaches, we rely on task effects and task-level heuristics to recursively optimize the plan level-by-level, instead of depth-first non-optimal planning in the network. Higher level plan solutions are decomposed into subproblems and refined into finer level plans, which are in turn decomposed and refined. Backtracks between levels occur when costs of refined plans exceed the expected costs of higher-level plans, thus ensuring to produce optimal plans at each level of the hierarchy. We demonstrate the relevance of our approach on several well-known domains compared with state-of-the-art domain-knowledge planners.

## INTRODUCTION

Automated planning is a field of Artificial Intelligence which aims at automatically computing a sequence of actions that lead to some goals from a given initial state. Many subareas have been explored, some assuming that effects of actions are deterministic (Ghallab, Nau, and Traverso 2004). Even in this case, solving realistic problems is challenging because finding a solution path may require to explore an exponential number of states with regard to the number of state variables. To cope with this combinatorial explosion, efficient algorithms use heuristics, which guide the search towards optimistic or approximate solutions. Remarkably, hierarchical methods iteratively decompose the planning problem into smaller and much simpler ones.

In a vast majority of problems, the planner must deal with constraints, such as multiple predefined phases or protocols. Such constraints generally help solving the planning problem, because they prune lots of search paths where these constraints do not hold. They can be given by an expert of

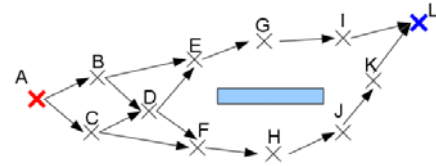


Figure 1: Example of path planning graph

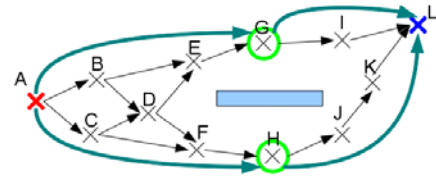


Figure 2: Path planning graph with high level choice

the problem to solve — which is often the case in many realistic applications such as military missions — or beforehand automatically deduced from the model. In this paper, we assume that these constraints are known and given to the planner. We thus propose a new method to model and solve a deterministic planning problem, based on a hierarchical and heuristic approach and taking advantage of these constraints.

## Intuition on a simple example

We illustrate our idea on a simple navigation problem, but our approach pre-eminently targets complex structured problems formalized in a kind of hierarchical STRIPS semantics (Ghallab, Nau, and Traverso 2004). In the graph of Figure 1, the robot must go from A to L. A human user who see this graph can immediately say that there is an important choice to do: go around the wall by the north through G or by the south through H, as shown in Figure 2. Therefore, it seems to us interesting to solve this problem at coarse grain, using this information to decide where we should pass before exploring at fine grain this solution, avoiding to explore the non-chosen branch. Refinement of the chosen path into elementary steps may question the previous choice, revealing an unseen difficulty. For instance, there may be a hole in E discovered when exploring the path via G in details, forcing the agent to reappraise the choice of this path and

changing its higher level decision to the path via  $H$ . We then replan at coarse grain using this new information, until the solution converges.

Intuitively, this approach consists in making jumps in the state graph, then refining these jumps by recursively doing shorter ones until we apply only elementary steps.

## Related work

The idea of adding domain-dependent control knowledge to help finding a plan is wide spread. We can cite TLPlan (Bacchus and Kabanza 2000) in which the authors use temporal logic (LTL) to give properties defining “good” plans (i.e. cheap plans that lead to the goal) over a sequence of actions or states (not only the current state). This allows for very precise guidance of the search either by checking if the current partial plan is correct, or if it may lead to a complete plan that satisfies the formulas.

Other approaches use what is called *procedural knowledge*: an user, who writes a planning problem, knows by experience some techniques, some groups of actions (and recursively) that achieve a subgoal and knows how to break down each goal and subgoal into finer subgoals. Several works are done in this field. In Hierarchical Task Networks (HTNs) (Erol, Hendler, and Nau 1994), the global mission is recursively broken down into a combination of subtasks, until the planner applies only elementary actions. The High-level Actions (HLA) framework (Marthi, Russel, and Wolfe 2008) differs from HTNs on the fact that no recipe is given for the whole mission: the planner has to build the first high level plan then refine it the same way than for HTNs. Planning algorithms are also associated with the BDI formalism (de Silva, Sardina, and Padgham 2009). Our main difference with these formalisms and associated planning techniques is that we plan one hierarchical level at a time and keep a coherence in the abstraction level of the different tasks in each hierarchical plan. Thus, we allow the planner to foresee shortcuts and difficulties *at each level* of the hierarchy, avoiding to plan an elementary step without knowing the long-term effect of this step at coarse grain.

Other works aim at automatically learning some kind of procedural knowledge. For instance, Landmarks Planning techniques as used in Lama (Richter, Helmert, and Westphal 2008), where the planner deduces a set of subgoals from the problem, Macro-FF (Botea et al. 2005), where the planner tries to make groups of actions that have interesting effects, or HTN-MAKER (Hogg, Muñoz-Avila, and Kuter 2008) where the algorithm tries to generalize tasks by analyzing admissible plans. While these works are interesting, they assume that knowledge is learned rather given by human experts, what definitely targets applications with different design and operational constraints.

We now present how we extended the HTN formalism to implement our contribution, and the algorithm we developed to solve problems expressed in this formalism. In a last part, we compare the performances of our planner with SHOP2, dynDFS and TLPlan on several planning benchmarks.

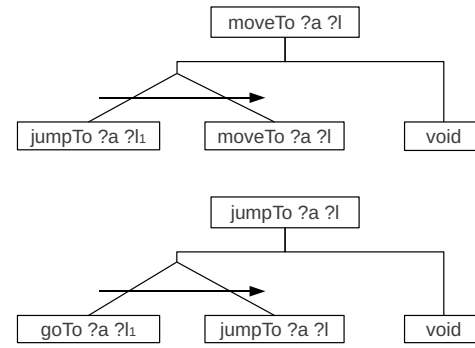


Figure 3: Example of HTN

## FORMALISM

**PDDL planning** The goal of “classical” planning is to compute a strategy called *plan* to reach a *goal* with the exact knowledge of the applicable *actions* and their *effects* in a completely known world. A problem of classical planning is a  $P = (s_0, g, A)$  where  $s_0$  is the initial state of the world,  $g$  the goal to reach, defined as a set of states, and  $A$  a set of actions. The initial state of the world and all the other *states* of the world are represented by a set of literals  $L$  describing the world.  $A$  is the set of actions  $a$ . The goal is defined by a set of literals either true or false. If all literals (and their value) are given in the goal description, the goal state is unique, otherwise it defines a set of states. Each action  $a$  is a triplet  $a = (\text{name}(a), \text{precond}(a), \text{effects}(a))$ , where  $\text{name}(a)$  is the name of the action,  $\text{precond}(a)$  are the preconditions required on the current state to apply  $a$ , and  $\text{effects}(a)$  are the modifications done on the current state by the application of  $a$ . A plan  $\pi$  is a sequence of actions.  $\pi$  is a solution of the problem if by the application of all its actions from  $s_0$  it leads to  $g$ .

In order to describe planning problems, the PDDL language (presented on (Fox and Long 2003)) and its various extensions are widely used. It is based on the Strips formalism, and breaks the problem into two parts, the *domain* that contains the set of actions  $A$ , and the *problem* that contains the initial state of the world  $s_0$  and the goal definition  $g$ .

**Expressing hierarchy with HTNs** A Hierarchical Task Network (HTN) (Erol, Hendler, and Nau 1994) is an extension to classical planning that consists in modeling *tasks*, that is, abstract actions with different *methods* to break them down. A HTN problem is a tuple  $(s_0, g, A, T)$ , where  $s_0$  is the initial state,  $g$  is the goal,  $A$  the set of elementary actions (as above), and  $T$  the set of tasks. A task  $t \in T$  is a set of preconditions and a set of methods:  $t = (\text{precond}(t), M(t))$  where  $\text{precond}(t)$  is a literal formula that represents the set of states where the task can be performed, and  $M(t)$  is the set of methods  $m(t)$ . Each method  $m(t)$  defines a possible decomposition of the task into subtasks or elementary actions. There are two ways of breaking down a task, *parallel* and *sequential*. A parallel decomposition gives the subtasks the possibility to be executed in parallel whereas a sequential decomposition forces the planner to put the subtasks one

after the other in the given ordering. In most applications, the set of tasks is given by a human expert of the domain, and have a significant influence on the performance of the planner.

A graphical representation of a HTN is shown on Figure 3. Tasks and elementary actions are represented in boxes, a horizontal line shows the different choices of methods for that task, and slanted bars show the decompositions of methods. Sequential decompositions are represented by arrows. We can see here a model to solve a path planning problem, where `moveTo ?a ?l` represents the highest level task (the mission) consisting for an agent `?a` to reach the location `?l`, `jumpTo` a high level move and `goTo` an elementary step. The `void` task is the termination case, necessary to stop the recursion.

**Meta-effects to link tasks** In the standard HTN formalism as defined by (Erol, Hendler, and Nau 1994), each task represents a group of methods to achieve a sub-goal, but the planner does not have knowledge of the accomplished sub-task. Therefore, it is impossible to tie up a task after another one without exploring it in details to know its effects. In other terms, the standard formalism does not allow for helpful coarse grain exploration of the problem. In order to use HTN tasks directly as macro-operators at any level, the first extension we need to add to the HTN formalism must give the planner the ability to know the effect of a task.

In order to do so, we introduce meta-effects for HTNs. These meta-effects are attached to tasks like effects are attached to elementary actions. This allows the planner to get a knowledge of the main effects of a task and to assemble high-level tasks to make a high level plan. With that knowledge, it will be able to check the pre-conditions of the next task and compute a high-level heuristic.

Our task  $t$  is now a set of preconditions, a set of methods and a set of effects:  $t = (\text{precond}(t), M(t), \text{effects}(t))$ .

Here is the BNF of the meta-effects in PDDL-like syntax:

```
<metaEffect> ::= ":metaEffect" <effect>

<effect> ::= <pEffect>
| "(not" <effect> ")"
| "(forall" "(" <typedVarLst> ")" <effect> ")"
| "(when" <cond> <effect> ")"
| "(and" <effect>+ ")"
<pEffect> ::= <atomicTermFormula>
| "(assign" <fHead> <fExp> ")"
| "(increase" <fHead> <fExp> ")"
| "(decrease" <fHead> <fExp> ")"
```

where `<fHead>` is a function, `<fExp>` an expression, `<atomicTermFormula>` a boolean expression, `<cond>` a boolean condition and `<typedVarLst>` a list of typed variables.

An example is shown on Figure 4. We give meta-effects to high-level tasks `moveTo` and `jumpTo` that give the result of the task, i.e. the position of the robot at the end of the task. These meta-effects are written in a rounded box on the graphic representations of HTN. Meta-effects can be more or less precise depending on which points are considered as relevant by the expert. Here, an estimated cost of a move or a jump, computed with euclidean distance from the start-

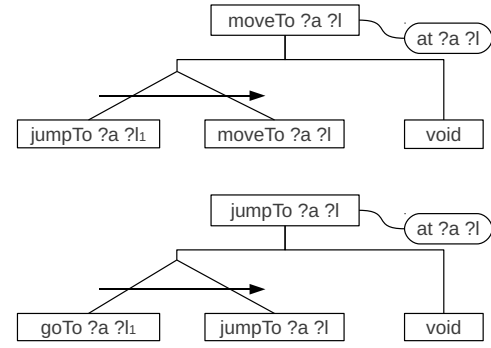


Figure 4: Meta-effects in HTN

ing point of the task to its destination, can be associated to the meta-effect if the underlying planner can deal with it. In PDDL, this example is written as:

```
:metaEffect
(
  and
    (increase (cost) (dist (at ?a) ?l))
    (assign (at ?a) ?l)
)
```

The level of precision of the meta-effects has an important influence on the planning process: if they are exhaustive with respect to the effects of the underlying actions, then the effects of the task are totally predictable, and the choice of a given task will not need to be reconsidered. Contrary to the works by (Marthi, Russel, and Wolfe 2008), who also define meta-effects, our effects are generally not complete, i.e. some numerical estimations of the final state are not well evaluated and some predicate changes are not present. This simplification allows us to use meta-effects the same way as normal effects in any forward planning algorithm.

Inspired by admissible heuristics in classical planning, we define *optimistic meta-effects* such that the long term cost of a meta-effect is lower than the real one.

**Macro-tasks to avoid recursion** Another weakness of standard HTNs concerns the modeling of methods that must be decomposed into an unknown number of subtasks (determined at planning time). For instance, consider our navigation graph of Figure 3. To break down the `jumpTo` task, we need to recursively write that `jumpTo` is a sequence of one `goTo` to a given point next to the starting point, followed by a `jumpTo` from there to the goal.

This may cause several problems. For modelers and readers who do not have expert programming skills, it is not very intuitive to break this task down using recursion. One must deal with termination cases, or ask oneself if he would rather use right or left recursion. Most importantly, task recursion is incoherent with our idea of doing jumps in the state graph. At high level of hierarchy, the planner tries to plan a `moveTo` by refining it into a `jumpTo` and a `moveTo`. At the next level, it will be a `goTo` then a `jumpTo`, then another `jumpTo` and a `moveTo`. That is, the computed plan does not have any consistence in terms of hierarchy.

Thus, we introduce *macro-tasks* as another extension in the spirit of regular expressions. The aim is to break



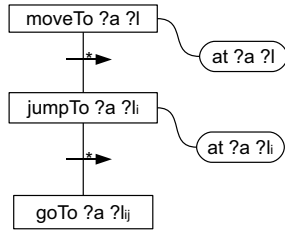


Figure 5: Macro-task example

down a task into an unknown number of subtasks that are all at the same level in the hierarchy. A method  $m$  is now defined as a precondition and a macro-task :  $m = (\text{precond}(m), \text{macroTask}(m))$ . A macro-task is recursively defined by several alternatives that express how to group subtasks together, the terminal case being a single subtask:

- ordered: subtasks are executed in sequence;
- multseq: a subtask is executed an unknown number of times until a final condition is met;
- optional: a subtask is executed only if needed;
- pickOne: a subtask is executed, where the value of a variable satisfying a given constraint is set.

Macro-tasks are lazily refined by the parser, so that the planner's algorithm described in the next section needs to only assume that macro-tasks are sequences of subtasks.

The definition of the grammar is the following:

```
<macroTask> ::= <naryTask>
| ordered <macroTask>*
| multseq <macroTask> until <cnd>
| optional <macroTask>
| pickOne <varLst> <cnd> <macroTask>
```

where  $\langle \text{varLst} \rangle$  is a list of variables,  $\langle \text{naryTask} \rangle$  a subtask with its parameters and  $\langle \text{cnd} \rangle$  a boolean test.

An example of this extension is presented on Figure 5. Compared with Figure 4, the model is simpler and more understandable, and above all, all different occurrences of a same task are all at the same level of the hierarchy.

The PDDL decomposition of `jumpTo` is written as:

```
:subtasks
(:multseq
  (:pickOne (?l1 - loc) (isElem (at ?a) ?l1)
    (goTo ?a ?l1)
  )
until (= (at ?a) ?l)
)
```

where `loc` is a type representing a location and `isElem` is a boolean function that tests if the path between its two arguments is elementary and possible or not. The keyword `multseq` represents a set of subtasks with an unknown arity, and `pickOne` defines the point of choice of a given variable.

## ALGORITHM DESCRIPTION

In this section, we present an algorithm that is able to solve any problem expressed in the previous formalism. The main idea of this algorithm, named HDS for Hierarchical Deepening Search, consists in computing first a plan with a low

level of precision, then using this plan as a guide to compute a more precise plan, until we obtain a detailed plan that contains only elementary actions. At each step, the algorithm backtracks to a previous higher level plan if the cost of the current plan is higher than the expected quality of the lower-level plans.

### Using a lower precision plan as a guide

Let  $n$  be a given level of the hierarchy. We assume first that complete plans have been constructed for all upper levels including  $n$ . The by-level planner (see Algorithm 1) uses the macro-tasks at level  $n + 1$  and the plan  $\mathcal{P}_n$  at level  $n$  to compute a higher precision plan  $\mathcal{P}_{n+1}$  that solves level  $n + 1$ . We can use any forward planning algorithm, for instance  $A^*$ , slightly modified to handle constraints from  $\mathcal{P}_n$  in its exploration.

The first idea is that the by-level planner uses all tasks (actions and macro-tasks) as elementary actions, using the meta-effects of macro-tasks as normal effects. The second idea consists in keeping track, for each state, of its position in the HTN by means of an extended state  $\sigma := (\sigma.s, \sigma.p)$ , composed of the state  $\sigma.s$  and the position  $\sigma.p$  in the HTN. Using this extended state, we can significantly restrict the branching factor: in each state, we pick-up actions that can be applied according to the position  $\sigma.p$  in the HTN among the ones whose preconditions are satisfied in state  $\sigma.s$ . This is quite similar to works by (Kuter and Nau 2005) and (Marthi, Russel, and Wolfe 2008), except that we run this algorithm at each level of the hierarchy, not only the finest one.

We initialize the forward planner with a root node containing the initial state of the problem (lines 2 to 5). In each state that is explored by the forward planning algorithm, we look at the position in the upper plan and the possible solutions proposed by the method decomposition of the upper task. Among all of these solutions, we keep only the applicable ones according to the current state and the preconditions.

The possible successors are defined by the upper plan and the methods of the meta-actions. We keep track of the current task of the upper plan and the current position in the methods decomposition of the task. According to the position in the higher plan, we have different branching possibilities:

- if just entered a primitive action: apply it in the new plan and go to the next task,
- if just entered a meta-action: successors are the different acceptable methods according to their preconditions,
- if ordered/parallel: apply in sequence/parallel the different sub-tasks, without choices,
- if optional: develop two successors, one with the optional subtask inserted, one without,
- if multseq/multpar: develop in sequence/parallel the subtask until the end condition is true
- if pickone: develop all successors with all combination of variables accepted by the condition,
- if face a subtask: check the preconditions, if true apply the effects, otherwise declare the branch as dead-end.

Using an algorithm inspired from A\* for this forward planner, we have the algorithm 2. As in A\*, we maintain a planning tree for each level of hierarchy. This tree contains at each node:

- a state (node. $\sigma$ )
- the lowest cost to reach it (node.cost),
- an estimation of the cost to reach the goal (computed by an heuristic) (node.estim)
- and the successors of this node (node.succs), that is, the reachable states according to the different applicable and acceptable actions.

The algorithm rides recursively through this tree, choosing at each node its most promising successor, that is, the one with the lowest sum of its cost and its estimated cost (line 24). Once a tip node is reached, that is, a node without successor, the algorithm applies all the applicable and acceptable actions from this state (line 8), and affects the resulting states as successors for the current node (line 15).

The algorithm stops when the goal set has been reached or when it is established that the problem has no solution, that is, when no more node can be developed. It then extracts the plan from the A\* tree ( $\emptyset$  if the problem has no solution) (line 5).

---

**Algorithm 1: By-level planner:  $\mathcal{P}_{n+1} = \text{by-level}(\mathcal{P}_n)$**

---

```

1 begin
  //  $\mathcal{P}_n$  is the higher level plan
2   root. $\sigma \leftarrow \text{initState}$ ;
3   root.cost  $\leftarrow 0$ ;
4   root.estim  $\leftarrow \text{heuristic}(\text{root}.\sigma)$ ;
5   root.succs  $\leftarrow \emptyset$ ;
6    $\mathcal{P}_{n+1} = \text{runPlanner}(\text{root}, \mathcal{P}_n)$ ;
7   return  $\mathcal{P}_{n+1}$ ;

```

---

### Links between levels

We present now (see Algorithm 3) how we construct the complete hierarchical plan (defined at all levels of the hierarchy) by refining or backtracking between plans iteratively constructed by the by-level planner.

We initialize the planner with the `init_task` of the problem (line 2), that is used by the by-level algorithm as a guide to compute the first plan. Then we keep an instance of the by-level planner for each level. The by-level planner is launched using the plan extracted from the upper level (line 5).

Then, once the currently lowest level (lets call it  $n$ ) by-level planner ends its work, we call a plan updater (line 10) on the higher level plans. This updater reports the actual best estimated cost to the final node of the upper by-level plan (at level  $k, k < n$ ). By propagating this cost to the whole best branch of the planning structure, the updater will be able to determine if the best plan is still the same or not (line 10). This can happen if the meta-effects of a task were too optimistic and the development of this task at a finer level has shown an extra cost. In this case, another branch can pretend

---

**Algorithm 2: Astar by-level planner:**

---

```

1 begin runAStar(root,  $\mathcal{P}_n$ )
2   goalReached  $\leftarrow \text{false}$ ;
3   while root.cost  $< \infty \wedge \neg \text{goalReached}$  do
4     | goalReached  $\leftarrow \text{aStarRecPlanner}(\text{node}, \mathcal{P}_n)$ ;
5   return extractPlan(root);
6 begin aStarRecPlanner(node,  $\mathcal{P}_n$ )
7   if node.succs =  $\emptyset$  then
8     |  $\mathcal{A}_{st} \leftarrow \text{next}(\text{node}.\sigma.p) \cap \text{acceptable}(\text{node}.\sigma.s)$ ;
9     | forall the  $a \in \mathcal{A}_{st}$  do
10      | node'. $\sigma.s \leftarrow \text{apply}(a.\text{effects}, \text{node}.\sigma.s)$ ;
11      | node'. $\sigma.p \leftarrow \text{track}(\text{node}.\sigma.p, a, \mathcal{P}_{n+1})$ ;
12      | node'.cost  $\leftarrow \text{node.cost}$ 
13      |   + cost(node. $\sigma.s, a, \text{node}'.\sigma.s)$ ;
14      | node'.estim  $\leftarrow \text{heuristic}(\text{node}'.\sigma)$ ;
15      | node.succs  $\leftarrow \text{node.succs} \cup \{\text{node}'\}$ ;
16   if node.succs =  $\emptyset$  then node.estim =  $\infty$ ;
17   goalReached  $\leftarrow \text{false}$ ;
18   else
19     | if satisfies(node. $\sigma.s, \text{goal}$ ) then
20       | goalReached  $\leftarrow \text{true}$ ;
21     | else
22       | node'  $\leftarrow$ 
23       |   argmin $_{n' \in \text{node.succs}} (n'.\text{cost} + n'.\text{estim})$ ;
24       | goalReached  $\leftarrow \text{aStarRecPlanner}(\text{node}')$ ;
25       |  $c \leftarrow \min_{n' \in \text{node.succs}} (n'.\text{cost} + n'.\text{estim})$ ;
26       | node.estim  $\leftarrow c - \text{node.cost}$ ;
27   return goalReached;

```

---

to have better estimated costs to the goal and invalidate the plan.

This updater is called on each plan, from level  $n - 1$  to level 0 (lines 8 to 13). At each step, the current evaluation of the best possible plan is reevaluated and used for the directly upper plan. The planning sequence starts again at the coarser level where the best plan has changed.

We continue propagating the new cost estimation towards the coarsest plan. For each level, we note if the plan has been questioned or not. We then restart the computation at the coarsest level which has been questioned.

This algorithm terminates if it finds a plan containing only elementary tasks and which is not invalidated by the upper by-level planners, that is when the final solution is found, or when the estimated cost for the highest level by-level planner reaches infinity, that is, when it is estimated that no plan can be computed to reach the goal with the given decomposition.

### HDS Properties

HDS properties first rely on the HTN and its meta-effects. If the solution (resp. optimal solution) is not reachable through the HTN, HDS will not be able to find any solution (resp. the optimal solution) of the problem. Assuming the HTN is well written, i.e. the optimal solution is reachable, the plan-

**Algorithm 3: Hierarchical planner**


---

```

1 begin
2    $\mathcal{P}_0 \leftarrow \text{init\_task};$ 
3    $n \leftarrow 0;$ 
4   while ( $\text{cost}(\mathcal{P}_0) < \infty$ )  $\wedge$  ( $\neg \text{is\_elem}(\mathcal{P}_n)$ ) do
5      $\mathcal{P}_{n+1} \leftarrow \text{by-level}(\mathcal{P}_n);$ 
6      $\text{max\_cost} \leftarrow \text{cost}(\mathcal{P}_{n+1});$ 
7      $\text{last\_change} \leftarrow n+1;$ 
8     for  $i \leftarrow n$  to 1 step -1 do
9        $\text{old\_}\mathcal{P}_i \leftarrow \mathcal{P}_i;$ 
10       $\text{reeval}(\mathcal{P}_i, \text{max\_cost});$ 
11      if  $\mathcal{P}_i \neq \text{old\_}\mathcal{P}_i$  then
12         $\text{last\_change} \leftarrow i;$ 
13         $\text{max\_cost} \leftarrow \text{cost}(\mathcal{P}_i);$ 
14       $n \leftarrow \text{last\_change};$ 
15   if  $\text{cost}(\mathcal{P}_0) = \infty$  then
16     return  $\emptyset;$ 
17   else
18     return  $[\mathcal{P}_0 : \mathcal{P}_n];$ 

```

---

ner may still consider an intermediate solution that does not allow the planner to reach the optimal solution if the meta-effects are not optimistic (i.e. their long-term cost are higher than the real cost); the backtrack process will not be able to detect it.

Second, the properties of our algorithm depends on the properties of the by-level planner used:

**Correctness.** If the by-level planner is correct, complete and optimal (i.e. it finds the optimal solution iff there exists a solution at a given level), HDS is still: (1) correct, because the finest plan is computed only with the elementary actions and by the by-level planner; (2) complete and optimal, since, given optimistic meta-effects, the cost of any plan is always lower to the best one, that is no plans with a higher cost than optimal ones can be proposed as a solution and the planner cannot conclude that no plan can reach the solution (that would be a plan with infinite cost).

**Termination.** If the by-level planner terminates, as the minimum estimated costs strictly increases at each hierarchical backtracks and there is a finite number of hierarchical levels, HDS will converge to the solution or conclude to the fact that no solution exists in a finite time.

## EXPERIMENTAL STUDY

### Implementation

We implemented our planner in the OCaml language, which is a functional language, like Lisp, therefore suitable for our highly recursive algorithm. We tuned the garbage collector of the language, forcing it to optimize the RAM occupation. This optimization made us gain approximatively 40% in speed, which is however not significant compared to other improvements obtained over some state-of-the-art domain-knowledge planners as shown in the following.

To implement our by-level planner, we chose the Dijk-

stra algorithm, modified to use macro-tasks and information from upper plan. Even if there exist far more efficient algorithms in the literature, we chose to implement a very simple and quite naive by-level planner in order to highlight the relevance of our global Hierarchical Deepening Search approach (efficiency does not come from the by-level planner but from our general framework). Along the same line, we do not use generic heuristics, such as Hmax or Hadd (Haslum and Geffner 2000). Without these heuristics, we have much less constraints on the formalism, and our planner accepts object functions, i.e. functions that return an object instead of just a number. We can also use non linear functions or effects.

### Comparisons with other planners

Our planner HDS is optimal given an HTN decomposition on the problem. We compared HDS with TLPlan (Bacchus and Kabanza 2000), a non optimal domain-dependent planner based on LTL temporal logic; with dynDFS (Pralet and Verfaillie 2008), a domain-dependent optimal temporal planner based on the Timelines formalism; and with SHOP2 configured to find the optimal solution, which is a successful HTN planner (without meta-effects). The first three tracks presented in the next are from the IPC3 planning competition (ICAPS 2002). All planners were allowed 2Gb of RAM and 10 minutes to plan each problem on a 3GHz Intel processor.

**Satellite.** The Satellite STRIPS domain comes from IPC3 where a fleet of satellites have to take pictures of various events with various instruments. In the STRIPS version, each action has a time cost of one. The aim is to minimize the total time of the mission. Parallelism between satellites is authorized. In this domain, we compared HDS with dynDFS and with TLPlan. SHOP2 is not presented here as we did not have HTNs for SHOP2 on this track.

Figure 6 presents planning times and costs for the different planners. Since parallelism of tasks is not yet available on HDS, our HTN decomposition is quite weak, including only tasks to initialize a sensor (turn towards an acceptable ground station then switch on the sensor and calibrate it) and to take a picture (turn to event then take picture), and cannot really take advantage of the hierarchical framework.

HDS performances are similar to dynDFS, which is specialized in parallelism, but can solve fewer problems. In particular, HDS finds optimal plans for the problem that could be solved. As a reference, we report also the costs found by the domain-independent planner Lama (Richter, Helmert, and Westphal 2008) which, as TLPlan, cannot take advantages of parallelism in term of costs. Lama was set in the optimizing mode, and even if it does not always get the optimal cost (in a non parallelized plan), it can often find a much better solution than the one found by TLPlan, showing that TLPlan solutions are far from the optimal ones (TLPlan did not take advantage of parallelism and get high costs as soon as the problem has more than one satellite).

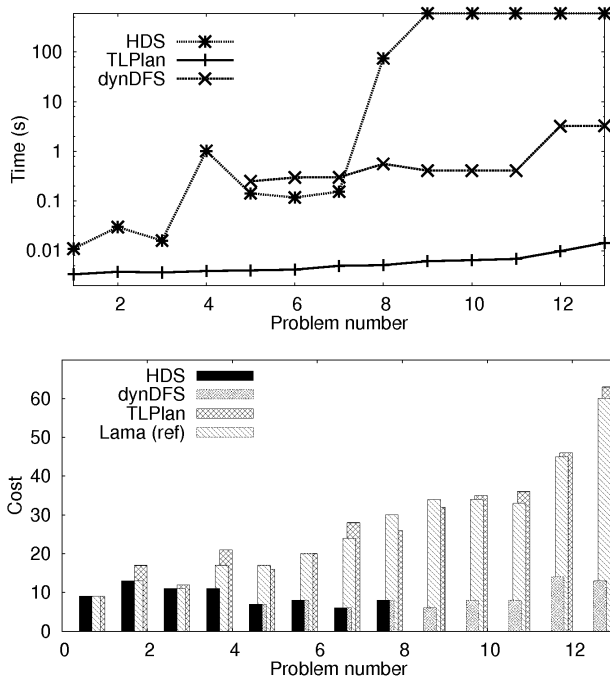


Figure 6: Satellite

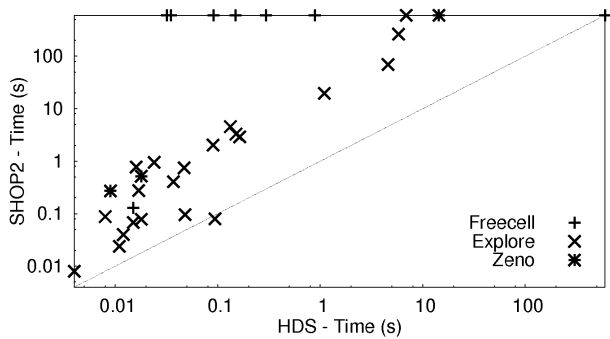


Figure 7: HDS vs SHOP2

**Freecell.** This domain comes also from IPC3 and is inspired from by the famous Microsoft Windows game. We compared ourselves with SHOP2, giving SHOP2 exactly the same HTNs as the ones given to HDS, except meta-effects and macro-tasks that cannot be handled by SHOP2. We configured these HTNs to find the optimal solution at the finest level of the hierarchy. We forced the planners to send to the home location all unneeded cards, as automatically done in the Windows game. We provided another method to move a block of cards of the same column if enough free cells are available. Figure 7 presents planning times for both planners. Costs are not plotted since HDS is optimal and SHOP2 is configured here to be optimal. SHOP2 is able to solve only the first problem, whereas HDS can solve the seven first ones. Additionally to meta-effects and macro-tasks, this difference can be due to several other factors: Lisp is far less efficient than OCaml and HDS can deal

with more abstract functions and denser problem descriptions than SHOP2, leading to more efficient computation.

**Zeno Traveler.** In this domain also extracted from IPC3, the planner has to make people reach their destination by plane. The planes have two speed modes: slow and zoom. Slow consumes far less fuel than zoom, but it is far slower. In the numeric version, as each move only takes one time step, zoom is never the best solution. Like Satellite, the lack of parallelism between tasks in our model restricts the capacities of HDS. We only gave as knowledge the information that the plane can only go to destinations where someone is waiting or where someone needs to go, and that someone already at his destination is not allowed to board the plane.

We compared HDS with SHOP2 in its optimal mode, using its IPC3 HTN decomposition. We can see on Figure 7 the advantage of HDS: in the first two problems, with just one plane, HDS computation time is around 20 milliseconds, whereas SHOP2 computation time is around 200 milliseconds. For the other problems, the planner must choose among multiple planes, and even if parallelism is not really taken into account, SHOP2 cannot solve any of these problems, whereas HDS can solve the third problem in 14 seconds.

**Explore and Guide.** This domain particularly puts in evidence the advantages of our approach. The goal is, for a helicopter, to drive back intruders to the border, having explored their known exit path in order to ensure that no trap is present. The zone is described as a discrete grid, the helicopter knows the position and the exit path of each vehicle and must perform a "explore" action on each cell of the exit path to prepare the drive back procedure of each vehicle. The aim is to minimize the number of explored cells and the mileage of the helicopter.

In this problem, non concurrent high-level tasks are easy to identify and their effects and costs are well approximated. Once the highest-level plan is computed, it is very helpful for the computation of the exploration strategy, that is split into sub-zones by an expert.

We gave to SHOP2 exactly the same HTNs as to HDS, except meta-effects and macro-tasks. The main algorithmic difference is that HDS first explores at low precision and then refines this plan (with backtracks), whereas SHOP2 directly explores at the finest precision level. Both planners return the same optimal solution for each problem. The results are presented in Figure 7, where we can see that HDS is still between one and two orders of magnitude quicker than SHOP2.

## CONCLUSION AND FUTURE WORKS

In this paper, we proposed to use both macro-operator techniques and procedural control knowledge within the same informed planning framework. We introduced the *meta-effects* and *macro-tasks* extensions to the HTN formalism, allowing us to jump forward in the state graph. We also proposed an algorithm named HDS that explores, level by level,

such a structure, thus detecting traps and optimizing an abstract plan before refining it into a precise executable plan, backtracking to another high-level solution if necessary. We furthermore proved that HDS, thanks to the proposed extensions to the HTN formalism, is very efficient and optimal given the decomposition on structured problems. This contribution provides an assistance to write large planning problems using domain expertise, and to reduce the complexity of the underlying planning algorithm. The required domain expertise can be also automatically extracted from the model, and used in our approach.

In a close future, we plan to implement real parallelism, not only in our model but also in our planner. We expect gains by introducing more human expertise in the domains and better performances on some problems. Since our algorithmic approach is quite generic, especially concerning the by-level planner, we plan to extend our contribution to other planning schemes, such as probabilistic planning, using a forward MDP by-level planner.

## References

- Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*.
- 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.
- de Silva, L.; Sardina, S.; and Padgham, L. 2009. First principles planning in bdi systems. In *Autonomous Agents and Multiagent Systems (AAMAS-09)*.
- Erol, K.; Hendler, J.; and Nau, D. S. 1994. HTN planning: complexity and expressivity. In *AAAI'94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 2)*, 1123–1128.
- Fox, M., and Long, D. 2003. Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:2003.
- Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning*. San Francisco, CA, USA: Morgan Kaufmann.
- Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. 140–149. AAAI Press.
- Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2008. Htn-maker: Learning htms with minimal additional knowledge engineering required. In *Association for the Advancement of Artificial Intelligence (AAAI-08)*.
- ICAPS. 2002. Planning competition. <http://ipc.icaps-conference.org/>.
- Kuter, U., and Nau, D. S. 2005. Using domain-configurable search control for probabilistic planning. In *AAAI*.
- Marthi, B.; Russel, S.; and Wolfe, J. 2008. Angelic hierarchical planning: Optimal and online algorithms. In *International Conference on Automated Planning and Scheduling (ICAPS-08)*.
- Pralet, C., and Verfaillie, G. 2008. Using constraint networks on timelines to model and solve planning and scheduling problems. In *Proc. ICAPS*.
- Richter, S.; Helmert, M.; and Westphal, M. 2008. Landmarks revisited. In *23rd AAAI Conference on Artificial Intelligence (AAAI-08)*.

# Automatic Polytime Reductions of NP Problems into a Fragment of STRIPS

**Aldo Porco**

Departamento de Computación  
Universidad Simón Bolívar  
Caracas, Venezuela  
aldo@gia.usb.ve

**Alejandro Machado**

Departamento de Computación  
Universidad Simón Bolívar  
Caracas, Venezuela  
alejandromachado@gia.usb.ve

**Blai Bonet**

Departamento de Computación  
Universidad Simón Bolívar  
Caracas, Venezuela  
bonet@ldc.usb.ve

## Abstract

We present a software tool that is able to automatically translate an NP problem into a STRIPS problem such that the former problem has a solution iff the latter has one, a solution for the latter can be transformed into a solution for the former, and all this can be done efficiently. Moreover, the tool is built such that it only produces problems that belong to a fragment of STRIPS that is solvable in non-deterministic polynomial time, a fact that guarantees that the whole approach is not an overkill. This tool has interesting applications. For example, with the advancement of planning technology, it can be used as an off-the-shelf method to solve general NP problems with the help of planners and to automatically generate benchmark problems of known complexity in a systematic and controlled manner. Another interesting contribution is related to the area of Knowledge Engineering in which one of the goals is to devise automatic methods for using the available planning technology to solve real-life problems.

## Introduction

Deciding plan existence for STRIPS is PSPACE-complete (Bylander 1994). This means that any decision problem that can be solved by a deterministic algorithm that uses polynomial space can be reduced in polynomial time to deciding plan existence of a STRIPS problem. However, although such reductions exist, there is no known algorithm that automatically generates a STRIPS problem from an arbitrary PSPACE problem (up to our knowledge).

Such an algorithm would be a valuable tool for a number of reasons. First and most importantly, the algorithm would provide the basis for developing a General Problem Solver able to tackle PSPACE problems, suitably described in a high-level declarative language, by translating them into STRIPS and then applying one of the many available planners. Second, given the recent (and future) advancements in the area, the algorithm would be of practical interest too because it would offer an easy way to solve concrete instances of difficult problems and, in some cases, competitively with other specialized algorithms. Finally, by generating STRIPS problems in a controlled manner, one could design benchmark problems in order to evaluate the heuristics or algorithms used in planning. All these applications assume in

one way or another that the input is a declarative description of a PSPACE problem, that a solution for the input problem can be efficiently computed from a solution to the planning problem, and that solving the latter problem is no more difficult than solving the former problem.

In this paper we take a first step in constructing such a tool by considering the class NP instead of PSPACE. Nonetheless, the result is an interesting tool that is able to translate a given NP problem, expressed as a decision problem, into a STRIPS problem that satisfies above properties. We present the formal translation and its properties, and evaluate it on classical NP-complete problems such as satisfiability and the computation of Hamiltonian paths on digraphs.

The input problem is specified using the existential fragment of second-order logic that is known to capture NP. This is a fundamental result in the area of Descriptive Complexity Theory (DCT) on which the tool is firmly grounded. On the other hand, the guarantee that the planning problem is no more difficult than the input (in the worst case) is achieved by targeting a class of STRIPS problems for which plan existence can be decided in NP.

Part of the contribution is related to the area of Knowledge Engineering for Planning and Scheduling (KEPS) that focuses on the practical deployment of planning resources for solving real-life problems. The tool presented in this work produces a planning problem that can be fed into a planner from a declarative description of a NP problem. Thus, the tool can be thought as a KEPS tool that permits the use of planning technology for solving real-life problems that are not directly specified in PDDL. However, differently from other tools, our framework permits to formally characterize the scope, soundness and completeness of the tool, and to obtain worst-case guarantees on the complexity of solving the generated problems.

Our tool is not the first such tool. Cadoli and Schaerf (2005) develop one that translates a DATALOG-like specification of an NP problem, called NP-SPEC, into a SAT problem that is then fed into a solver to find a solution to the input problem. Hence, our proposal is quite similar to NP-SPEC, yet we target STRIPS instead of SAT. Furthermore, unrestricted STRIPS as well as specifications based on second-order logic go well beyond NP, and thus we have a clear direction for extending the tool in the future.

Another related work is that of Mitchell and Ternovska

(2005) that proposes a general framework for describing problems, in NP and beyond, that is based on the Model Extension (MX) problem. A simple parameterization of MX renders the decision problem in NP, while fully unrestricted MX is in NEXPTIME. Mitchell and Ternovska propose a language for expressing general problems, but do not present a practical solver based on their ideas.

In the following, we revise the relevant concepts from DCT, describe the tool, give the formal translation from NP problems into STRIPS, and study its properties. At the end, we present experiments and conclude with a discussion.

## Descriptive Complexity Theory

It is a field of research, lying in the intersection of mathematics and computer science, that studies complexity theory from a pure mathematical viewpoint without committing to a model of computation such as Turing machines. DCT began with the seminal work of Fagin (1974) on NP. Today, the major complexity classes had been characterized and important results had been obtained (Immerman 1998).

In DCT, a decision problem like SAT<sup>1</sup> is denoted as a collection of finite (first-order) structures that satisfy a second-order existential sentence. In this section, we review the fundamental concepts from logic and the most important results from DCT that are relevant to this work.

## Languages

Every logical language is constructed from a set of symbols or vocabulary. The symbols are typically partitioned into pure logical symbols such as ‘ $\wedge$ ’, ‘ $\exists$ ’, etc., punctuation symbols such as ‘(’ and ‘)’, and relational, functional and constant symbols. The logical and punctuation symbols belong to every language while the relations, functions and constants change from one language to another. Hence, it is convenient to define the *signature* of the language as the finite set of relations, functions and constants that are allowed in the formulae. Signatures are denoted by tuples like  $\sigma = \langle P^1, Q^2, f^1, A, B \rangle$  that contains two relational symbols  $P$  and  $Q$  of arities 1 and 2 respectively, one functional symbol  $f$  of arity 1, and two constants  $A$  and  $B$ . In DCT, functional symbols can be avoided altogether and thus will not be considered in the rest of the paper. We denote with  $\text{FOL}(\sigma)$  and  $\text{SOL}(\sigma)$  the sets of all first-order and second-order formulae built from  $\sigma$ . In general, if  $\mathcal{L}$  denotes a logic or fragment of a logic,  $\mathcal{L}(\sigma)$  denotes the set of all formulae belonging to  $\mathcal{L}$  built from  $\sigma$ .

SOL differs from FOL in that quantification on relational symbols is permitted. SOL formulae are constructed as

- any  $\text{FOL}(\tau)$  formula,  $\tau \supseteq \sigma$ , is a  $\text{SOL}(\sigma)$  formula,
- if  $\Phi$  and  $\Psi$  are  $\text{SOL}(\sigma)$  formulae, then so are ‘ $(\Phi)$ ’, ‘ $\neg\Phi$ ’, ‘ $\Phi \wedge \Psi$ ’, ‘ $\Phi \vee \Psi$ ’, etc., and
- if  $R^a \notin \sigma$  and  $\Phi$  is a  $\text{SOL}(\sigma)$  formula, then ‘ $(\exists R)\Phi$ ’ and ‘ $(\forall R)\Phi$ ’ are  $\text{SOL}(\sigma)$  formulae.

We typically denote FOL formulae with lowercase Greek letters, and other formulae with capital letters. For the rest

<sup>1</sup>In this paper, SAT is the subset of satisfiable CNF formulae.

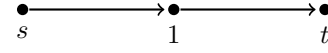


Figure 1: the digraph that corresponds to the structure  $\mathcal{A} = \langle |\mathcal{A}|, E^{\mathcal{A}}, s^{\mathcal{A}}, t^{\mathcal{A}} \rangle$  where  $|\mathcal{A}| = \{0, 1, 2\}$ ,  $E^{\mathcal{A}} = \{(0, 1), (1, 2)\}$ ,  $s^{\mathcal{A}} = 0$  and  $t^{\mathcal{A}} = 2$ .

of the paper, we only consider (first- and second-order) formulae without free variables, also called sentences, and for second-order formulae, those that comply with the form

$$\Phi = (\mathcal{Q}_1 R_1^{a_1})(\mathcal{Q}_2 R_2^{a_2}) \cdots (\mathcal{Q}_n R_n^{a_n})\psi$$

where each  $\mathcal{Q}_i$  is a quantifier in  $\{\exists, \forall\}$ , and  $\psi$  is a first-order sentence over  $\sigma \cup \{R_1^{a_1}, \dots, R_n^{a_n}\}$ . This form is universal because for every second-order sentence there is an equivalent of this form. If all  $\mathcal{Q}_i$ s are existential quantifiers, then we say that  $\Phi$  is a second-order existential sentence. The class of all second-order existential sentences, also called the existential fragment of SOL, is denoted by  $\text{SO}\exists$ ; similarly, one defines  $\text{SO}\forall$ . If there is  $1 < k < n$ , such that  $\mathcal{Q}_i = \exists$  for all  $i < k$  and  $\mathcal{Q}_i = \forall$  for all  $i \geq k$ , then the sentence belongs to the fragment  $\text{SO}\exists\forall$ , and so on.

## First-Order Structures

Logical formulae are interpreted with respect to first-order structures. A first-order structure over signature  $\sigma = \langle R_1^{a_1}, \dots, R_s^{a_s}, c_1, \dots, c_t \rangle$ , where  $R_i$  is an  $a_i$ -ary relational symbol and  $c_j$  is a constant, is a tuple  $\mathcal{A} = \langle |\mathcal{A}|, R_1^{\mathcal{A}}, \dots, R_s^{\mathcal{A}}, c_1^{\mathcal{A}}, \dots, c_t^{\mathcal{A}} \rangle$  with non-empty universe  $|\mathcal{A}|$  where each  $R_i^{\mathcal{A}} \subseteq |\mathcal{A}|^{a_i}$  is a subset of  $a_i$ -tuples from  $|\mathcal{A}|$  (called the interpretation of  $R_i$ ) and each  $c_j^{\mathcal{A}} \in |\mathcal{A}|$  is an element of  $|\mathcal{A}|$  (called the interpretation of  $c_j$ ). Without loss of generality, we assume that the universe is always of the form  $|\mathcal{A}| = \{0, 1, \dots, n-1\}$ . DCT is only interested in finite structures (i.e. structures of finite universe); the class of all finite structures for signature  $\sigma$  is denoted by  $\text{STRUC}[\sigma]$ .

We do not have enough space to formally present the semantic interpretation of formulae with respect to structures. We only say that a formula  $\phi$  in  $\text{FOL}(\sigma)$  (or  $\text{SOL}(\sigma)$ ) holds (or is satisfied) in structure  $\mathcal{A} \in \text{STRUC}[\sigma]$  iff the formula holds when each relation  $R_i$  and constant  $c_j$  is interpreted by  $R_i^{\mathcal{A}}$  and  $c_j^{\mathcal{A}}$  respectively. If  $\phi$  holds in  $\mathcal{A}$ , we write  $\mathcal{A} \models \phi$ . For example, consider  $\sigma = \langle E^2, s, t \rangle$  and  $\mathcal{A} = \langle |\mathcal{A}|, E^{\mathcal{A}}, s^{\mathcal{A}}, t^{\mathcal{A}} \rangle$  where  $|\mathcal{A}| = \{0, 1, 2\}$ ,  $E^{\mathcal{A}} = \{(0, 1), (1, 2)\}$ ,  $s^{\mathcal{A}} = 0$  and  $t^{\mathcal{A}} = 2$ . Then,  $\mathcal{A} \models (\exists x)(E(s, x) \wedge E(x, t))$  and  $\mathcal{A} \not\models (\forall x)(\exists y)E(x, y)$ . Notice that  $\sigma$  can describe digraphs with designated vertices  $s$  and  $t$ ; e.g.,  $\mathcal{A}$  corresponds to the graph shown in Fig. 1.

Second-order formulae are also interpreted with respect to first-order structures. For example, the sentence

$$\Phi_{2\text{COL}} = (\exists R^1)(\forall xy)[E(x, y) \rightarrow \neg(R(x) \leftrightarrow R(y))]$$

holds in  $\mathcal{A}$  since the unary relation  $R = \{1\}$  makes it true. Indeed, this formula holds in a structure  $\mathcal{A} \in \text{STRUC}[\sigma]$  iff the digraph encoded by  $\mathcal{A}$  is 2-colorable.

The class of finite structures in  $\text{STRUC}[\sigma]$  that satisfy a sentence  $\Phi \in \mathcal{L}(\sigma)$  is denoted by  $\text{MOD}[\Phi]$ ; e.g.,  $\text{MOD}[\Phi_{2\text{COL}}]$  is the class of all finite 2-colorable digraphs.

DCT requires some basic numeric relations and constants that have fixed interpretation at each structure; the relations are  $<^2$ ,  $SUC^2$ ,  $BIT^2$ ,  $PLUS^3$  and  $TIMES^3$  while the constants are 0 and max. For the interpretations,  $x < y$  iff  $x$  is less than  $y$ ,  $SUC(x, y)$  iff  $y = x + 1$ ,  $BIT(x, y)$  iff the  $y$ th-bit in the number  $x$  is 1,  $PLUS(x, y, z)$  iff  $z = x + y$ , and  $TIMES(x, y, z)$  iff  $z = x \times y$ . The constants 0 and max are mapped into 0 and  $\|\mathcal{A}\| - 1$  respectively.

### Complexity Classes

The last example shows that a sentence can describe a collection of finite discrete structures (such as digraphs) that satisfy a certain property (such as 2-colorability).

In DCT, a decision problem corresponds to a class of first-order structures that satisfy a sentence. The seminal work of Fagin (1974) established that every decision problem in NP can be characterized by the class of structures that satisfy a second-order existential sentence; in symbols,  $NP = SO\exists$ . Consider SAT, for example. An instance of SAT is a CNF with  $m$  clauses over  $n$  propositional variables, where a clause is a subset of positive and negative literals. Two relational symbols  $\sigma_{SAT} = \langle N^2, P^2 \rangle$  suffice to describe the positive and negative occurrences of propositions in clauses:  $N(x, y)$  (resp.  $P(x, y)$ ) expresses that the proposition  $x$  appears negatively (resp. positively) in clause  $y$ ; e.g.,  $(p \vee \neg q \vee r) \wedge (\neg p \vee \neg r) \wedge (\neg p \vee q)$  is encoded with  $\mathcal{A} = \langle |\mathcal{A}|, N^{\mathcal{A}}, P^{\mathcal{A}} \rangle$  where  $|\mathcal{A}| = \{0, 1, 2\}$ ,  $N^{\mathcal{A}} = \{(1, 0), (0, 1), (2, 1), (0, 2)\}$  and  $P^{\mathcal{A}} = \{(0, 0), (2, 0), (1, 2)\}$ . On the other hand, a truth-assignment is a subset of true propositions and the existence of a truth assignment (satisfiability) can be expressed with the  $SO\exists$  sentence  $\Phi_{SAT}$ :<sup>2</sup>

$$(\exists T^1)(\forall y)(\exists x)[(P(x, y) \wedge T(x)) \vee (N(x, y) \wedge \neg T(x))]$$

The following list shows the major results of DCT on the characterization of complexity classes (Immerman 1998):

- non-deterministic log-space (NL) equals FOL extended with a transitive-closure operator (FO(TC)),
- P equals second-order Horn sentences (SO-Horn),
- NP equals  $SO\exists$  and coNP equals  $SO\forall$ ,
- the polynomial-time hierarchy (PH) equals SO, and
- PSPACE equals SOL extended with a transitive-closure operator (SO(TC)).

A transitive-closure operator is a *syntactic construct* whose interpretation coincides with the transitive closure of a relation. Thus, it is not surprising that NL equals FO(TC) because checking the existence of a path from node  $s$  to node  $t$  in a digraph with designated vertices  $s$  and  $t$  is NL-complete (Sipser 2005), and this property holds whenever  $s$  is related to  $t$  in the transitive closure of the edge relation.

### Syntactic Abbreviations

Quite often one needs to quantify over a  $k$ -ary function  $f^k$  instead of a relation. This can be accomplished by quantifying over a  $(k + 1)$ -ary relation  $F^{k+1}$  and adding first-order

<sup>2</sup>This sentence assumes that  $m \geq n$ . If not, add tautological clauses to the CNF.

formulae that guarantees that  $F$  represents  $f$ . For example, a unary function  $f$  can be represented by the binary relation  $F$  and the formula

$$\psi_{fun} = (\forall xy y')(F(x, y) \wedge F(x, y') \rightarrow y = y').$$

Likewise, in need of an injective function, one quantifies over a relation  $F$  and adds above formula plus

$$\psi_{inj} = (\forall xx' y)(F(x, y) \wedge F(x', y) \rightarrow x = x').$$

Finally, if the function needs to be total, then one should use  $\psi_{tot} = (\forall x)(\exists y)F(x, y)$ . We use the abbreviations  $(\exists F \in Fun)\phi$  and  $(\exists F \in Inj)\phi$  to denote  $(\exists F^2)(\phi \wedge \psi_{fun} \wedge \psi_{tot})$  and  $(\exists F^2)(\phi \wedge \psi_{fun} \wedge \psi_{inj} \wedge \psi_{tot})$  respectively, and ‘PFun’ instead of ‘Fun’ and ‘PInj’ instead of ‘Inj’ if the function does not need to be total. For example, the following sentence defines digraphs with directed Hamiltonian paths

$$\Phi_{DHP} = (\exists F \in Inj)(\forall x)[x < \max \rightarrow (\exists x' yz)(E(y, z) \wedge F(x, y) \wedge SUC(x, x') \wedge F(x', z))].$$

To see how it works, observe that a directed Hamiltonian path over the vertices  $|\mathcal{A}| = \{0, \dots, n-1\}$  can be thought of as a permutation  $f$  on the vertices such that  $E(f(x), f(x+1))$  for each  $0 \leq x < n-1$ .

### The Tool

The input to the tool is a signature  $\sigma$ , a  $SO\exists$  sentence  $\Phi$  describing a property of interest, and a first-order structure  $\mathcal{A}$  describing an object on which to test the property  $\Phi$ . The output is a PDDL domain and instance that have a valid plan if and only if  $\mathcal{A}$  satisfies  $\Phi$ . Moreover, a *certificate* for the satisfaction of the property, in the form of values for the second-order variables in  $\Phi$ , can be recovered in linear time from the plan.

Hence, we can think of the tool as a generator of reductions among problems. Recall that a reduction from problem  $A$  into problem  $B$  is a computable function  $f$  such that for each instance  $\omega$ ,  $\omega \in A$  iff  $f(\omega) \in B$ .

In our case, the reduction decomposes in two functions

$$\mathcal{D} : \text{Signatures} \times SO\exists \rightarrow \text{PDDL Domains},$$

$$\mathcal{I} : \text{Signatures} \times SO\exists \times \text{STRUC} \rightarrow \text{PDDL Instances}$$

such that  $\text{dom} = \mathcal{D}(\sigma, \Phi)$  is a PDDL domain and  $\text{ins} = \mathcal{I}(\sigma, \Phi, \mathcal{A})$  is a PDDL instance.

In order to get something of theoretical and practical interest, the reduction should run in polynomial time (so that it would not be able to check whether  $\mathcal{A}$  satisfies  $\Phi$  and then produce a trivial planning problem) and its output should be solvable in NP (so that complexity of solving the output problem is no bigger than the complexity of solving the input problem). However, the plan-existence decision problem for PDDL is not in NP because 1) the number of grounded fluents and actions may be exponential in the input size, and 2) a minimum-length plan may be of exponential size in the number of grounded fluents and actions. Thus, not every reduction works for our purposes and we must be careful with its design. In the following two sections, we present an acceptable reduction and study its formal properties.



## Reductions

A (grounded) STRIPS planning problem is a tuple  $P = \langle F, I, G, O \rangle$  where  $F$  is a collection of fluents (propositions),  $I \subseteq F$  denotes the initial state,  $G \subseteq F$  denotes the goal states and  $O$  is a collection of actions. Each action  $a \in O$  is defined by three subsets of fluents  $pre(a)$ ,  $add(a)$  and  $del(a)$  that stand for the precondition, and the positive and negative effects of the action. As usual, action  $a$  is applicable at state  $s$  iff  $pre(a) \subseteq s$ , and the result of applying  $a$  is  $res(s, a) = (s \setminus del(a)) \cup add(a)$ . A plan for state  $s$  is a sequence of actions applicable from the initial state  $I$  that achieves the goal condition.

A PDDL planning problem is a pair  $\langle \text{dom}, \text{ins} \rangle$  made of a domain and instance description in the PDDL language (McDermott et al. 1998). In this paper, we only consider the simplest fragment of PDDL which is equivalent to ungrounded STRIPS, and hence the *grounding* of  $\langle \text{dom}, \text{ins} \rangle$  results in an STRIPS problem  $P$ . The size of the grounding is polynomial for *fixed* domain, but exponential for unrestricted domain and instance.

Getting an acceptable reduction is not obvious at the beginning, but once you get one, others become apparent. For lack of space, we present only one reduction that is aimed at SAT-based planners. We are aware of other reductions, including one designed for optimal sequential planners that produce a delete-free problem together with a length bound.

The first step in the translation is to transform the formula by eliminating the implications and bi-implications, and moving the negations to the literal level. Further, constants other than 0 and max are removed by introducing unary relational symbols with interpretations consisting only of the unique element that interprets each constant. After the formula is preprocessed in this manner, the domain and problem are generated as follows.

### Domain

$\mathcal{D}(\sigma, \Phi)$  produces a domain for signature  $\sigma$  and sentence  $\Phi \in \text{SOL}(\sigma)$  of the form  $(\exists R_1^{a_1}) \cdots (\exists R_n^{a_n})\psi$ . The actions in the domain are divided in three groups: actions for setting the truth-value of the second-order variables, actions for proving the sentence  $\psi$  and two other actions.

**Actions for variables** For each second-order variable  $R_i$  or arity  $a_i$ , there is an action `set_Ri_true` with  $a_i$  parameters that sets the fluent `Ri` and removes `not-Ri` which is *initially true*; these fluents are used to denote the truth value of  $R_i$ . For example, the action for the relation  $T^1$  in SAT is

```
(:action set_T_true
:parameters (?x)
:precondition (and (guess) (not-T ?x))
:effect (and (T ?x) (not (not-T ?x))))
```

The fluent `guess` is used to create two phases within plans: a ‘guess’ phase for setting the value of quantified relations, and a ‘proof’ phase for showing the validity of  $\psi$ .

**Actions for formulae** These actions are designed following the structure of  $\psi$  and make use of fluents that denote the validity of the subformulae in  $\psi$ .

For each subformula  $\theta$ , there is a fluent  $\mathfrak{F}[\theta]$  that denotes its validity in the structure and actions to add it. The fluent  $\mathfrak{F}[\theta]$  has parameters that match the free variables in  $\theta$ . The function  $\mathfrak{F}[\cdot]$  is called the fluent translation and it is closely related to the Tseitin translation (Tseitin 1968).

The actions are generated by recursing over the subformulae  $\theta$  of  $\psi$  in a depth-first manner as follows:<sup>3</sup>

- if  $\theta(\bar{x}) = \bigwedge_{i=1}^n \theta_i(\bar{x}_i)$  with  $\bar{x} = \bigcup_{i=1}^n \bar{x}_i$ , then generate the action `prove $[\theta]$`  with parameters  $\bar{x}$ , precondition  $\bigwedge_{i=1}^n \mathfrak{F}[\theta_i](\bar{x}_i)$  and unique add effect  $\mathfrak{F}[\theta](\bar{x})$ ,
- if  $\theta(\bar{x}) = \bigvee_{i=1}^n \theta_i(\bar{x}_i)$  with  $\bar{x} = \bigcup_{i=1}^n \bar{x}_i$ , then generate  $n$  actions of the form `prove $[\theta]_i$`  with precondition  $\mathfrak{F}[\theta_i](\bar{x}_i)$  and unique add effect  $\mathfrak{F}[\theta](\bar{x})$ ,
- if  $\theta(\bar{x}) = (\exists y)\theta'(\bar{x}, y)$ , then generate `prove $[\theta]$`  with precondition  $\mathfrak{F}[\theta'](\bar{x}, y)$  and unique add effect  $\mathfrak{F}[\theta](\bar{x})$ ,
- if  $\theta(\bar{x}) = (\forall y)\theta'(\bar{x}, y)$  then generate two actions. The idea is to prove  $\theta(\bar{x})$  by varying  $y$  over all objects.

The first action `prove $[\theta]_0$`  shows  $\theta'(\bar{x}, 0)$ . The action has parameters  $\bar{x}$ , precondition  $\mathfrak{F}[\theta'](\bar{x}, 0)$  and unique effect  $\mathfrak{F}[(\forall y \leq z)\theta'(\bar{x}, y)](\bar{x}, 0)$ . (Observe that the fluent translation is applied to a different formula in which the quantification is bounded by  $z$ .)

The second action `prove $[\theta]_1$`  shows  $\theta'(\bar{x}, z')$  inductively proves  $(\forall y \leq z)\theta'(\bar{x}, y)$  once  $(\forall y < z)\theta'(\bar{x}, y)$  holds. The action has parameters  $\bar{x}, z', z''$ , precondition  $\mathfrak{F}[(\forall y \leq z)\theta'(\bar{x}, y)](\bar{x}, z') \wedge \mathfrak{F}[\theta'](\bar{x}, z'') \wedge \text{SUC}(z', z'')$  and unique add effect  $\mathfrak{F}[(\forall y \leq z)\theta'(\bar{x}, y)](\bar{x}, z'')$ .

All these actions have as additional precondition the fluent `proof`. Also, notice that there are no actions for literals as such are taken care by the fluent translation as follows:

- $\mathfrak{F}[Q(\bar{x})](\bar{x}) = Q(\bar{x})$ ,
- $\mathfrak{F}[\neg Q(\bar{x})](\bar{x}) = \text{not-}Q(\bar{x})$ ,
- $\mathfrak{F}[(\forall y)\theta'(\bar{x}, y)](\bar{x}) = \mathfrak{F}[(\forall y \leq z)\theta'(\bar{x}, y)](\bar{x}, \text{max})$ ,
- in all other cases,  $\mathfrak{F}[\theta](\bar{x}) = \text{holds\_}<\text{id}>(\bar{x})$  where  $<\text{id}>$  is a unique identifier for  $\theta$ .

**Other actions** Two other actions are required. One for switching the phase from ‘guess’ to ‘proof’ called `begin-proof` that has precondition `guess`, adds `proof` and removes `guess`, and another action called `prove-goal` that has precondition  $\mathfrak{F}[\psi]$  and unique add effect `holds_goal`. Figure 2 shows the domain for  $\Phi_{\text{SAT}}$ .

**Abbreviations** In the presence of abbreviations, the operators for the second-order variables are extended in order to make the translations more efficient. For  $(\exists F \in \text{Fun})$ , the precondition and delete of `set_F_true` are extended with the fluent `(free_F_dom ?x)` so that there can be at most one fluent  $F(x, y)$  true for each  $x$  and thus there is no need to include the subformula  $\psi_{\text{fun}}$ . Similarly, for  $(\exists F \in \text{Inj})$  the precondition and delete are further extended with the fluent `(free_F_ran ?y)`.

<sup>3</sup> $\theta(\bar{x})$  means that the free variables in  $\theta$  are among those in  $\bar{x}$ .

```

(define (domain SAT)
  (:constants zero max)
  (:predicates
    (holds_and_2 ?x ?y) (holds_and_6 ?x0 ?x1)
    (holds_exists_8 ?x0) (holds_forall_9 ?x0)
    (holds_or_7 ?x0 ?x1) (holds_goal)
    (N ?x ?y) (P ?x ?y) (T ?x) (not-T ?x)
    (suc ?x ?y) (guess) (proof)
  )
  (:action set_T_true
    :parameters (?x)
    :precondition (and (guess) (not-T ?x))
    :effect (and (T ?x) (not (not-T ?x))))

  (:action prove_forall_9_1
    :precondition (and (proof)
      (holds_exists_8 zero))
    :effect (holds_forall_9 zero))

  (:action prove_forall_9_2
    :parameters (?y1 ?y2)
    :precondition (and (proof)
      (suc ?y1 ?y2)
      (holds_forall_9 ?y1)
      (holds_exists_8 ?y2))
    :effect (holds_forall_9 ?y2))

  (:action prove_exists_8
    :parameters (?y ?x)
    :precondition (and (proof)
      (holds_or_7 ?y ?x))
    :effect (holds_exists_8 ?y))

  (:action prove_or_7_0
    :parameters (?y ?x)
    :precondition (and (proof)
      (holds_and_2 ?y ?x))
    :effect (holds_or_7 ?y ?x))

  (:action prove_or_7_1
    :parameters (?y ?x)
    :precondition (and (proof)
      (holds_and_6 ?y ?x))
    :effect (holds_or_7 ?y ?x))

  (:action prove_and_2
    :parameters (?y ?x)
    :precondition (and (proof)
      (P ?x ?y) (T ?x))
    :effect (holds_and_2 ?y ?x))

  (:action prove_and_6
    :parameters (?y ?x)
    :precondition (and (proof)
      (N ?x ?y) (not-T ?x))
    :effect (holds_and_6 ?y ?x))

  (:action prove-goal
    :precondition (holds_forall_9 max)
    :effect (holds_goal))

  (:action begin-proof
    :precondition (guess)
    :effect (and (proof) (not (guess)))) )

```

Figure 2: Full domain translation for  $\Phi_{\text{sat}} = (\exists T^1)(\forall y)(\exists x)[(P(x, y) \wedge T(x)) \vee (N(x, y) \wedge \neg T(x))]$ .

## Problem

The PDDL problem is generated by the call  $\mathcal{J}(\sigma, \Phi, \mathcal{A})$ . The objects in the problem correspond to the elements in the universe  $|\mathcal{A}| = \{0, \dots, n-1\}$ : 0 is mapped to the object zero,  $n-1$  to the object max, and the other elements  $0 < i < n-1$  to objects obj.i. The goal is to achieve holds\_goal, and the initial situation consists of fluents describing the truth-value of all the relations in  $\mathcal{A}$  and the predefined relations such as  $<$ , SUC, etc. that are mentioned in  $\Phi$ . Also, for each second-order variable  $R$ , the initial situation has fluents to denote false values for  $R$ , and in cases where  $R$  is a function, the initial situation has fluents of the type free\_R\_dom and/or free\_R\_ran.

## Formal Properties

The most important properties to care about are soundness, completeness and the complexity guarantees. Soundness and completeness mean that the translation function actually implements a reduction between decision problems, while the complexity guarantees refer to the time to compute the reduction and the complexity of deciding plan existence on the generated problem. In this section we show that the tool is a polytime reduction from the NP problem  $\text{MOD}[\Phi]$  into a fragment of STRIPS that is decidable in NP.

It is well known that checking plan existence for STRIPS problems without deletes is in NP (Bylander 1994). The proof relies on the fact that an optimal plan does not repeat actions and thus is of linear size. A similar complexity result for STRIPS can be obtained if each action with non-empty delete list can be applied at most once.

**Definition 1.** A STRIPS problem  $P = \langle F, I, G, O \rangle$  is at-most-once iff the operators can be partitioned into  $O = O_0 \cup O_1$  such that all operators in  $O_0$  are delete-free, and for each  $a \in O_1$ , there is a fluent  $p \in \text{pre}(a) \cap \text{del}(a)$  that is added by no action; i.e.,  $p \notin \text{add}(a)$  for all  $a \in O$ . The class of all at-most-once problems is denoted by STRIPS-1.

Consider now the grounding function  $\mathcal{G}$  that maps a pair  $\langle \text{dom}, \text{ins} \rangle$  of PDDL domain and instance into a STRIPS problem  $P = \mathcal{G}(\text{dom}, \text{ins})$ . For fixed dom, the function  $\text{ins} \rightsquigarrow \mathcal{G}(\text{dom}, \text{ins})$  runs in polytime  $\mathcal{O}(\|\text{ins}\|^k)$  for some  $k$  that only depends on dom. Likewise, the translation function  $\mathcal{J}$  runs in polytime in the size of the structure  $\mathcal{A}$ , but exponential in the largest arity of a second-order existential quantifier in  $\Phi$ . Therefore, the function  $f_{\sigma, \Phi} : \text{STRUC}[\sigma] \rightarrow \text{STRIPS}$  defined by

$$f_{\sigma, \Phi}(\mathcal{A}) = \mathcal{G}(\mathcal{D}(\sigma, \Phi), \mathcal{J}(\sigma, \Phi, \mathcal{A}))$$

is a polytime function that maps  $\sigma$ -structures into grounded STRIPS problems. This function is a reduction.

**Theorem 2.** The function  $f_{\sigma, \Phi}$  is a polytime reduction from the decision problem  $\text{MOD}[\Phi]$  into STRIPS-1.

*Proof.* (Sketch.) The proof is by structural induction on the subformulae  $\theta$  of  $\psi$ , starting from literals and building up towards more complex subformulae. The statement to show is that  $\langle \mathcal{A}, R_1^\pi, \dots, R_n^\pi \rangle \models \theta(\bar{x})$  iff there is a plan  $\pi$  that achieves  $\mathcal{F}[\theta](\bar{x})$  and defines interpretations  $R_i^\pi$  for the second-order variables  $R_i$ . Here,  $\langle \mathcal{A}, R_1^\pi, \dots, R_n^\pi \rangle$  denotes

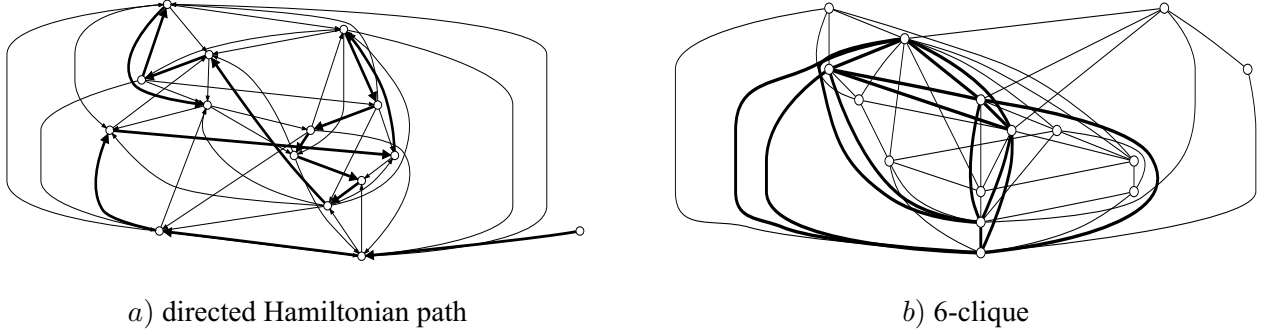


Figure 3: Two examples of NP-Complete problems reduced to STRIPS and the solutions computed by an off-the-shelf planner. The left panel shows a digraph of 15 vertices with a Hamiltonian path and the right panel a graph of 15 vertices with a 6-clique.

the extension of  $\mathcal{A}$  with the relations  $R_i^\pi$  that interpret the symbols  $R_i$ .

For  $\mathcal{A} \in \text{STRUC}[\Phi]$ ,  $f_{\sigma, \Phi}(\mathcal{A})$  is a STRIPS-1 problem because all operators are delete-free except `begin-proof` and the `set-true` operators, but each one of these deletes a precondition that is added by no operator.  $\square$

**Corollary 3.** *The plan-existence decision problem for STRIPS-1 is NP-complete.*

*Proof.* For inclusion, note that every action  $a \in O_1$  can be applied at most once. Because each such action deletes fluents, then some (or all) actions in  $O_2$  may be required before applying another action from  $O_1$ . Thus, the size of a plan is at most quadratic in the total number of actions. For hardness,  $f_{\sigma_{\text{SAT}}, \Phi_{\text{SAT}}}$  reduces SAT into STRIPS-1 in polytime.  $\square$

Therefore, our tool translates any given problem in NP, encoded by a  $\text{SO}\exists$  sentence, into PDDL/STRIPS. Such a sentence specifies the properties of a solution in a declarative manner. The solution of the problem is obtained from the valuation of the second-order variables that make the sentence true and that is contained in any valid plan for the planning problem. For example, the assignment that satisfies the CNF encoded by a structure  $\mathcal{A}$  corresponds to the values of the second-order variable  $T$ .

It is important to say that although  $\text{SO}\exists$  captures the whole class NP, there are problems that are easier to encode as sentences than others. For instance, there are succinct and clear sentences for SAT, Hamiltonian path,  $k$ -colorability, vertex cover and other problems, yet we do not have at this moment sentences for most of the benchmark problems used in planning. On the other hand, toy problems such as Blocksworld are not interesting and the exercise of abstracting relevant aspects of a real-world task into  $\text{SO}\exists$  sentences may reveal the core difficulties involved in a task.

In the rest of this section, we derive tight bounds on the length of parallel plans. These bounds are used with SAT-based planners to show that a given problem has no solution and also to improve performance.

### Horizon Windows

A horizon window for a STRIPS problem  $P$  is an interval of the form  $[s, f]$  such that  $P$  has a plan iff it has a plan of length  $\ell \in [s, f]$ . A window is a parallel-horizon window if  $\ell$  refers to the makespan of a parallel plan. Horizon windows can be effectively used to prune the search space.

The recursive structure of the generated problem permits the calculation of non-trivial horizon windows and of tight parallel-horizon windows. Indeed, since all set operators can be applied concurrently, a parallel plan needs at most one time step to execute them. The plan also requires the operators `begin-proof` and `prove-goal`. Thus, the parallel-horizon window is  $[2, 3]$  (the lower bound 2 applies when there is a plan that uses no set operators) plus the parallel-horizon window  $mkspw(\psi)$  of the sentence  $\psi$ . The parallel-horizon window is inductively defined as

- $mkspw(\theta) \doteq [0, 0]$  if  $\theta$  is a literal,
- $mkspw(\bigwedge_{i=1}^n \theta_i) \doteq 1 + \bigvee_{i=1}^n mkspw(\theta_i)$ ,
- $mkspw(\bigvee_{i=1}^n \theta_i) \doteq 1 + \bigwedge_{i=1}^n mkspw(\theta_i)$ ,
- $mkspw((\exists y)\theta(\bar{x}, y)) \doteq 1 + mkspw(\theta)$ , and
- $mkspw((\forall y)\theta(\bar{x}, y)) \doteq \|\mathcal{A}\| + mkspw(\theta)$ ,

where  $\mathcal{A}$  is the structure associated to the problem, and the operations between windows and scalars are  $[a, b] \vee [a', b'] \doteq [\max\{a, a'\}, \max\{b, b'\}]$ ,  $[a, b] \wedge [a', b'] \doteq [\min\{a, a'\}, \max\{b, b'\}]$  and  $c + [a, b] \doteq [c + a, c + b]$ . SAT, for example, has the window  $[\|\mathcal{A}\| + 5, \|\mathcal{A}\| + 6]$  which means that the CNF encoded by a structure  $\mathcal{A}$  is satisfiable iff there is a parallel plan of makespan  $\ell$  such that  $\|\mathcal{A}\| + 5 \leq \ell \leq \|\mathcal{A}\| + 6$ .

By bounding the upper limit of parallel horizon windows, we obtain the following.

**Theorem 4.** *Consider a signature  $\sigma$ ,  $\Phi \in \text{SO}\exists(\sigma)$  and  $\mathcal{A} \in \text{STRUC}[\sigma]$ . Then, to decide  $\mathcal{A} \models \Phi$ , it is enough to consider parallel plans of makespan linear on  $\|\mathcal{A}\|$  for fixed  $\Phi$  but independently of the arities in  $\sigma$  and  $\Phi$ . More precisely, it is enough to consider plans of makespan  $\max_b q_b(\|\mathcal{A}\| - 1) + h_b + 3$  where  $q_b$  is the number of universal quantifiers along branch  $b$  in the parse tree of  $\psi$ ,  $h_b$  is the height of branch  $b$ , and  $\psi$  is the FOL part of  $\Phi$ .*

*Proof.* Let  $n = \|\mathcal{A}\|$  and  $T$  the parse tree for  $\psi$ . For a maximal branch  $b \in T$ , let  $q_b$  be the number of universal quantifiers in  $b$ ,  $h_b$  its height, and  $u(b)$  the upper limit of the parallel horizon window along  $b$ . The upper limit  $u(\psi)$  of  $mkspw(\psi)$  is  $\max_{b \in T} u(b)$ , and  $u(b) = q_b n + h_b - q_b = q_b(n-1) + h_b$ . End by adding 3 to the upper limit  $u(\psi)$ .  $\square$

This bound is tight for SAT. This result is surprising because one would expect the need to consider parallel plans of makespan  $\mathcal{O}(\|\mathcal{A}\|^k)$  for some  $k$ . However, note that a linear makespan does not mean a linear number of operators.

Finally, observe that by composing the translation  $f_{\sigma, \Phi}$  with any translation from STRIPS to SAT, and using the upper bound of the window, one obtains a polytime reduction from the problem expressed by  $\Phi$  into SAT.

## Experiments

We performed experiments on the NP-complete problems SAT, CLIQUE, DIRECTEDHAMILTONIANPATH, 3-DIMENSIONALMATCHING, 3-COLORABILITY and  $k$ -COLORABILITY. Also, we computed the chromatic number of random graphs using the tool as an oracle.

The instances for SAT were taken from the SATLIB repository,<sup>4</sup> the instances for graph problems were randomly generated according to the  $G(n, p)$  model (Bollobás 2001), and the instances for matching were generated by randomly choosing triplets from  $\{0, \dots, n-1\}^3$  with probability  $p$ .

The experiments were performed on an Intel Xeon processor running at 1.86 GHz with 2 GB of RAM. The planners were run for 30 minutes with a limit of 1 GB of memory. The planners that solved more instances are M and Mp that support lower and upper bounds for time horizons (Rintanen 2010b; 2010a), and a num2sat (Hoffmann et al. 2007) modified to handle upper bounds on time horizons. Among these, M was the one that solved more instances.

Figure 3 shows two examples with solutions: the left panel shows a random digraph of 15 vertices that has a directed Hamiltonian path, and the right shows a random graph of 15 vertices with a 6-clique. These structures were discovered by M on the problems obtained from the sentences and the structures encoding the graphs.

Table 2 shows a summary of results for M. In total, we ran the planner on 1,920 instances from which 1,614 were solved: 706 on the positive side meaning that the input structure satisfies the property, and the rest 908 instances on the negative side. The problems of type uf20, uf50 and uf75 are random satisfiable 3CNF instances from the phase transition region with 20, 50 and 75 propositional variables respectively, while the problems uuf50 and uuf75 are random unsatisfiable instances. The instances of type  $n-k$  in CLIQUE refer to graphs with  $n$  vertices on which to look for cliques of size  $k$ , those  $n-k$  in  $k$ -COLORABILITY refer to graphs with  $n$  vertices on which to test  $k$ -colorability, and those of type  $n$  in other problems refer to graphs with  $n$  vertices.

The table contains information about the total number of instances of each type ( $N$ ), the number of instances solved by M ( $N^*$ ), the number of instances solved in the positive

instance	$\chi$	$k$ -colorability						
		1	2	3	4	5	6	7
10-0.75-1	5	2	2	6	101	3		
10-0.75-2	5	1	2	2	6	4		
10-0.85	7	2	2	3	6	4	1,265	4
15-0.25	2	27	62					
15-0.60	5	27	29	54	118	72		
15-0.70	6	28	28	33	47	329	67	
20-0.10	3	214	350	705				
20-0.25	4	211	272	1,261	837			

Table 1: Results for M on the computation of chromatic numbers on random graphs. For each instance, the table shows the chromatic number  $\chi$ , and the time (in seconds) to prove/disprove  $k$ -colorability for increasing values of  $k$ . The last value for  $k$ , for each instance, is the chromatic number.

and negative, and the average time that M took per instance. As it can be seen, we tried to generate a balanced set of problems with positive and negative instances. Overall, we think that M behaves very good as it solves 84.06% of the benchmark which is made of NP-complete problems of varying size and difficulty.

## Chromatic Numbers

The chromatic number of a graph  $G = (V, E)$  is the least  $k$  such that  $G$  is  $k$ -colorable. It is NP-hard to compute the chromatic number of a graph, but we can do it by testing for  $k$ -colorability for increasing values of  $k = 1, \dots, |V|$ .<sup>5</sup> Table 1 shows results for the computation of chromatic numbers on random graphs. For each instance, it shows the chromatic number  $\chi$  and the time to prove/disprove the existence of a  $k$ -coloring for increasing values of  $k$ .

## Discussion

We presented a “black box” that given as input a signature  $\sigma$ , a second-order existential sentence  $\Phi$  and a structure  $\mathcal{A} \in \text{STRUC}[\sigma]$ , outputs a STRIPS problem  $P$  that is solvable in non-deterministic polynomial time and has a plan iff  $\mathcal{A} \models \Phi$ . The black box is fully automated and runs in polynomial time in the size  $\|\mathcal{A}\|$ , and thus can be thought of as an efficient method to generate polytime reductions from NP into STRIPS.

The choice of  $\text{SO}\exists$  as the input language is arbitrary. However,  $\text{SO}\exists$  is a widely accepted formalism for expressing problems because it is *declarative* and not *tied* to any particular problem. In theory, one could choose any NP-Complete problem, such as SAT or Hamiltonian Path, as the input language for representing NP problems. This would make the translation much easier, but then the user would have to express his problems as instances of them, rendering the approach uninteresting.

We have not compared our approach with direct translations of problems into SAT, tools such as NP-SPEC from other areas, or specialized solvers. We expect to perform some of these comparisons in the near future. Specially,

<sup>4</sup><http://www.satlib.org>

<sup>5</sup>One can do better by performing a binary search on  $k$ .

with tools that generate SAT instances from problem descriptions, because our tool combined with M can be thought as a tool that reduces  $SO\exists$  to SAT.

More ambitiously, we would like to consider complexity classes beyond NP by exploiting known results in DCT. For example, PSPACE equals  $SO(TC)$  and thus any  $SO(TC)$  formula can be mapped into STRIPS. Unfortunately, the reduction is not as easy as the one for NP. The general reductions that we know consists of going from the formula to a polyspace TM that decides the validity of the formula in the input structure and then to simulate the TM with STRIPS. Instead, we would like more meaningful and practical reductions.

**Acknowledgments** Thanks to M. Helmert, P. Haslum and J. Hoffmann for interesting discussions, to J. Rintanen for helping us with M and Mp, and to the anonymous reviewers for their comments and references to related work.

## References

- Bollobás, B. 2001. *Random Graphs*. Cambridge University Press, second edition.
- Bylander, T. 1994. The computational complexity of propositional STRIPS planning. *Artificial Intelligence* 69:165–204.
- Cadoli, M., and Schaerf, A. 2005. Compiling problem specifications into SAT. *Artificial Intelligence* 162:89–120.
- Fagin, R. 1974. Generalized first-order spectra and polynomial-time recognizable sets. *American Mathematical Society* 7:27–41.
- Hoffmann, J.; Gomes, C.; Selman, B.; and Kautz, H. A. 2007. SAT encodings of state-space reachability problems in numeric domains. *Proc. 20th Int. Conf. on Automated Planning and Scheduling*, 1918–1923.
- Immerman, N. 1998. *Descriptive Complexity*. Springer.
- McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, New Haven, CT.
- Mitchell, D. G., and Ternovska, E. 2005. A framework for representing and solving NP search problems. *Proc. 20th National Conf. on Artificial Intelligence*, 430–435.
- Rintanen, J. 2010a. Heuristic planning with SAT: Beyond strict depth-first search. *Proc. 23rd Australasian Joint Conf. on Artificial Intelligence*, 415–424.
- Rintanen, J. 2010b. Heuristics for planning with SAT. *Proc. 16th Int. Conf. on Principles and Practice of Constraint Programming*, 414–428.
- Sipser, M. 2005. *Introduction to Theory of Computation, 2nd Edition*. Boston, MA: Thomson Course Technology.
- Tseitin, G. S. 1968. On the complexity of derivation in propositional calculus. In Slisenko, A. O., ed., *Studies in Constructive Mathematics and Mathematical Logic, Part 2*. Springer. 115–125.

	$N^*/N$	#pos	#neg	avg. time
SAT: $mkspw = [n + 5, n + 6]$				
uf20	40/40	40	0	1.7
uf50	40/40	40	0	146.7
uf75	15/40	15	0	362.1
uuf50	40/40	0	40	548.5
uuf75	1/40	0	1	1,746.4
CLIQUE: $mkspw = [2n + 4, 3n + 7]$				
10-3	40/40	22	18	1.2
10-4	40/40	12	28	2.2
10-5	40/40	1	39	32.3
15-3	40/40	22	18	10.5
15-4	40/40	11	29	36.6
15-5	39/40	4	35	74.3
15-6	37/40	1	36	79.4
20-3	40/40	25	15	40.2
20-4	40/40	17	23	72.6
20-5	39/40	10	29	159.6
20-6	34/40	4	30	185.2
25-3	40/40	30	10	111.9
25-4	40/40	18	22	231.0
25-5	39/40	10	29	387.5
25-6	36/40	8	28	394.1
DIRECTEDHAMILTONIANPATH: $mkspw = [n + 3, n + 10]$				
10	40/40	15	25	1.1
15	39/40	18	21	63.7
20	31/40	20	11	70.0
25	29/40	20	9	202.1
30	22/40	20	2	629.1
3-DIMENSIONALMATCHING: $mkspw = [3n + 4, 3n + 6]$				
10	40/40	36	4	9.6
15	40/40	40	0	251.5
20	13/40	13	0	1,191.0
25	0/40	0	0	—
3-COLORABILITY: $mkspw = [2n + 4, 2n + 7]$				
10	40/40	18	22	0.1
15	40/40	24	16	0.9
20	40/40	12	28	3.0
25	40/40	30	10	8.9
30	40/40	9	31	20.9
40	40/40	4	36	75.1
50	40/40	1	39	196.7
k-COLORABILITY: $mkspw = [2n + 4, 3n + 6]$				
10-2	40/40	9	31	1.9
10-3	40/40	18	22	2.8
10-4	40/40	27	13	11.0
15-2	40/40	7	33	33.5
15-3	40/40	16	24	46.5
15-4	40/40	24	16	91.7
20-2	40/40	3	37	254.9
20-3	40/40	12	28	395.9
20-4	40/40	20	20	497.3
25-2	0/40	0	0	—
25-3	0/40	0	0	—
25-4	0/40	0	0	—
Total	1,614/1,920	706	908	180.9

Table 2: Results for M. For each problem type, the table shows number of solved instances ( $N^*$ ), total number of instances ( $N$ ), number of solved instances that satisfy the property (#pos), number of solved instances that do not satisfy the property (#neg), and the average time in seconds.

# A Conceptual Framework for Post-Design Analysis in AI Planning Applications

Tiago Stegun Vaquero<sup>1</sup> and José Reinaldo Silva<sup>1</sup> and J. Christopher Beck<sup>2</sup>

<sup>1</sup>Department of Mechatronics Engineering, University of São Paulo, Brazil

<sup>2</sup>Department of Mechanical & Industrial Engineering, University of Toronto, Canada  
 tiago.vaquero@usp.br, reinaldo@usp.br, jcb@mie.utoronto.ca

## Abstract

A disciplined design process does not guarantee a complete and flawless model of a planning problem, even with the existing methods and tools available in the AI community. A post-design project phase can identify essential hidden knowledge and directly impact the final planning system. The central theme of this paper is the design of a domain-independent framework to structure the post-design process of planning applications for a better understanding and capture of missing requirements. The framework feeds the re-modeling process, closing the loop of the design cycle. We describe the conceptual model of such post-design framework, named *Post-Design Application Manager* (postDAM).

## Introduction

Since the end of the 1990s there has been an increasing interest in the application of AI planning techniques to solve real-life problems. In addition to characteristics of academic problems, such as the need to reason about actions, real-life problems require detailed knowledge elicitation, engineering, and management. It is necessary a systematic design process in which Knowledge and Requirements Engineering techniques and tools play a fundamental role. A well-structured life cycle to guide design increases the chances of building an appropriate planning application while reducing possible costs of fixing errors in the future. However, given the natural incompleteness of the knowledge, practical experience in real applications such as space exploration (Jónsson 2009) has shown that, even with a disciplined process of design, requirements from different viewpoints (e.g. stakeholders, experts, users) still emerge after plan generation, analysis and execution.

Conceptually, a design process does not guarantee a complete and flawless model of a planning problem, even with the existing methods and tools available in the AI community (McCluskey 2002; Vaquero et al. 2009). The identification of unsatisfactory solutions and unbalanced trade-offs among different quality metrics and criteria (Jónsson 2009; Rabideau, Engelhardt, and Chien 2000; Cesta et al. 2008) indicates a lack of understanding of requirements and preferences in the model. These hidden requirements and rationales raise the need for iterative model analysis, re-modeling and tuning process. However, the gathering and interpreta-

tion of the hidden requirements must be made carefully. The lack of a structured process for acquiring emerging knowledge can lead re-modeling to wrong directions and prevent a proper judgment of resulting plans, as well as their respective quality and tradeoffs.

The design process of planning domain models has a saturation point in which further model adjustment make no significant impact on the final plan. Therefore, the analysis and evaluation of generated plans - with respect to the requirements and quality metrics - becomes a fundamental step in the modeling cycle. Such a post-design process naturally leads to feedback and the discovery of hidden requirements for refining the model. While some hidden requirements are detected in a single plan analysis cycle, others are only discovered in after several analysis cycles; this aspect indicates that plan analysis have different stages of hidden requirements identification. In fact, the literature on knowledge engineering for planning has shown interesting tools and techniques for plan analysis, including plan animation (McCluskey and Simpson 2006; Vaquero et al. 2007), visualization (e.g. Gantt charts), virtual prototyping (Vaquero, Silva, and Beck 2010), and plan querying and summarization (Myers 2006). However, all that work did not consider different identification stages of hidden requirements and did not explore the effects of the missing knowledge in the re-modeling loop.

In this paper, we present a conceptual model of a post-design framework for AI planning applications, called *Post-Design Application Manager* (postDAM), that addresses the different stages of the plan analysis process, including (1) a short-term analysis stage with simulation and visualization of plans, (2) a long-term analysis stage with acquisition and re-use of human-centered rationales for plan evaluations, and (3) a stage for cross-project analysis that would provide generalization of knowledge from different domains to be applied in other planning application designs. The framework aims at structuring the post-design process for the continuous discovery and identification of hidden requirements (e.g., constraints and preferences) that would potentially improve the model and, consequently, the quality and performance of planners. postDAM was designed as a complement of the KE tool itSIMPLE (Vaquero et al. 2009) in order to support designer while dealing with post-design challenges.

This paper is organized as follows. First, we briefly in-

introduce the concepts of the postDAM framework, including its layered structure for handling the different stages of the plan analysis process. Then, we describe the layers of the framework and their role in discovering missing and hidden requirements, as well in the re-use of domain knowledge. We provide a short discussion about the components of the framework that have been implemented in itSIMPLE and the ones that should be addressed next. Finally, we conclude and provide some final remarks.

## The postDAM Framework

Due to the challenge of capturing requirements from different sources during design and the natural incompleteness of knowledge in complex planning applications, domain models might not reflect exactly the behavior expected by developers and stakeholders. Besides the initial expectation reflected in the requirements, design participants tend to learn and recognize their real needs during the development of the artifact. Incompleteness can mean that most of the recognition process must be part of the post design process, where a better and practical view of the system is available. The *Post-Design Application Manager* framework, postDAM, aims to support such an essential phase of plan development.

The primary goal of postDAM is to structure the post-design process in order to organize the plan analysis and support the capture of emerging requirements. The post-design process was designed as a domain-independent integrated framework with three layers of plan analysis, each requiring a different level of abstraction. As illustrated in Figure 1, the three layers are: *Short-term Plan Analysis*, *Long-term Plan Analysis*, the *Cross-project Re-use Analysis*. The abstraction level in these layers ranges from single plan evaluation (for acquiring metrics and rationales and to identify missing requirements) to the analysis of several cumulative plan evaluations (to discover hidden knowledge and to allow a cross-domain re-use of rationales). Figure 1 illustrates the layered analysis and how the layers interact and feed the re-modeling process in order to close the design loop.

We do not aim to replace the analysis done during the initial design process, but to complement the knowledge acquisition process, enhancing the feedback provided by post-design. This complementary process is as important as those performed during model design.

In what follows, we describe the main goal and role of each proposed layer of the framework.

### First Layer: Short-term Plan Analysis

The first layer in the framework aims to analyze plans directly during the design and development phase. The goal is to detect malformed solutions and identify inconsistent behavior, missing requirements and misinterpreted features that can be clearly spotted by design actors. Each plan is analyzed individually in an attempt to verify the overall coherence of the model and of the resulting plan. Figure 2 illustrates the short-term analysis process.

As the first step in this layer, plans must be validated to guarantee that they are in fact solutions to the specified planning problem and that they do not violate any constraints or

preferences defined in the domain model. Performing analysis without first checking plan validity is a waste of time and resources. If a plan is not a valid solution, it is necessary to revise the model of the domain, the plan itself, and the planner that generated it.

If a valid plan satisfies a problem specification, it does not mean that it can be directly applied and executed in real life. There are two main reasons why a plan cannot be directly executed. First, the plan might be composed of high level actions that can not be directly executed. This situation generally requires a post-processing to translate the high level plan into a low level sequence of actions or procedures that can be executed by controllers. Second, the specification of the planning problem and domain of application might be incomplete which would lead to unexpected results during the execution. For example, the lack of a safety constraint in the model to prevent robots from navigating too close to each other might result in crashes during execution. In this case, the execution of the plan must be investigated seeking for possible incoherence and unexpected characteristics. However, testing plan execution in real word can be very costly, risky, and sometimes impossible. Thus, a more elaborated plan analysis must take place. The postDAM framework uses a Virtual Prototyping (Cecil and Kanchanapiboon 2007) approach for such analysis.

A virtual prototyping stage is included in the first layer as a mechanism for plan simulation, which can reveal incompleteness and omissions in the problem specification or in the model of the domain of application. The virtual prototyping approach provides several advantages in system design (Cecil and Kanchanapiboon 2007). The approach has been widely used in industry for reasons such as: it provides cross-functional evaluation at a lower cost; it enables engineers to consider costly mistakes and downstream issues earlier in the product design cycle; and it facilitates better communication of product design issues among engineers of varying backgrounds (Cecil and Kanchanapiboon 2007). Connecting AI planning with the literature on virtual prototyping would benefit from these advantages.

The purpose of this visual technique is twofold. Firstly, it serves as a mimic of the real world in which the plan will be executed. The execution can consider physical properties that illustrate the applicability of plans in real scenarios. Properties such as collisions, gravity, mass, inertia and others can be verified (most of the available development environments for virtual prototyping have these properties already implemented in embedded engines - they can be included or not in the virtual model). Second, the visual and sound effects provide an excellent means of communication among design actors, such as domain experts, planning experts, stakeholders and users. These effects can give experts and non-experts a clear view of the domain model as well as an underlying planning strategy, as opposed to looking at plan traces. Flaws and missing constraints, not detected during design (or specification), can be spotted and discussed by visual inspection (Cecil and Kanchanapiboon 2007). Hypothetical examples of model inconsistency are: a robot trespassing solid bodies or sharp corner areas; a robot's battery reaching undesirable or unfeasible power levels; or a (pro-

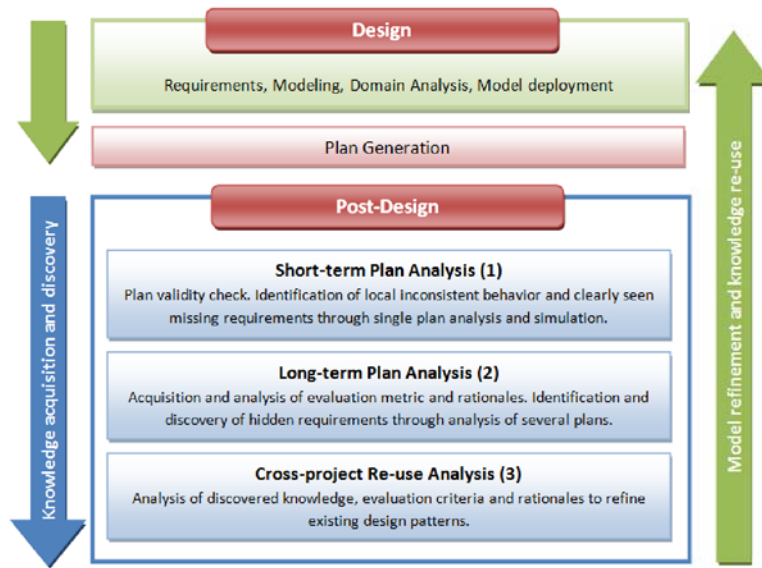


Figure 1: The conceptual model of the postDAM framework

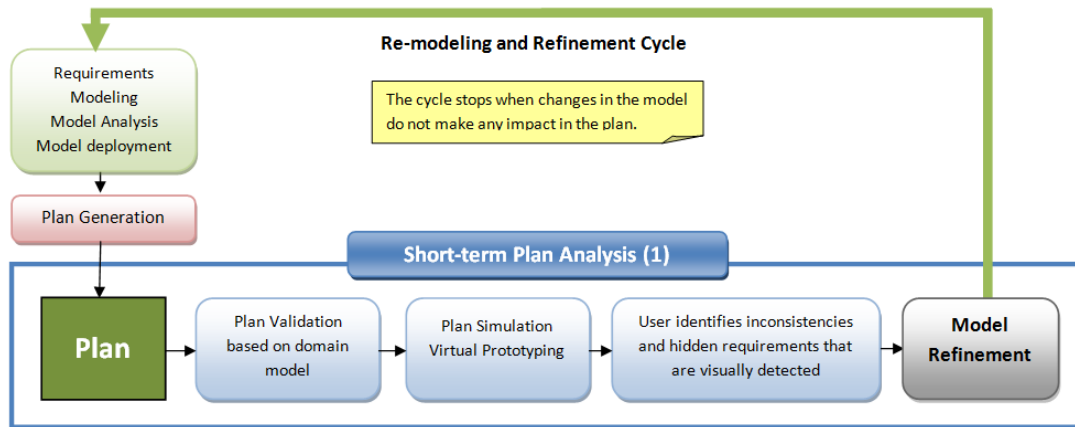


Figure 2: Short-term analysis in the postDAM framework

hibited) sequence of movements made by an autonomous air vehicle. Generally all these behaviors lead to a revision in the domain model. Besides the advantage in the model communication, a visual approach enriches the understating and familiarity with designs and resulting plans. Depending on the feedback provided during this analysis, early design phases must also be revisited.

Other plan visualization and simulation techniques would also fit on this layer of the framework (e.g., charts and diagrams). Similar to virtual prototyping, they serve as plan communication and plan analysis in the identification of inconsistencies in the model. However, we believe that virtual prototyping has a powerful communicative aspect via the visual interaction.

Creating virtual prototypes demands time and consumes resources, especially if applied to large and complex sys-

tems. However, its advantages in design support and decisions pay off, as seen in many real engineering problems (Cecil and Kanchanapiboon 2007). In addition, not all planning applications can be represented in visualization environments. For example, clinical decision support systems for oncology therapy planning (Fdez-Olivares, C  zar, and Castillo 2009) do not have a clear virtual representation to support analysis. For these cases, system or software prototyping becomes necessary to provide the same communication of the model for inconsistency detection.

The short-term layer and the re-modeling cycle are executed as many times as necessary. The cycle stops when there exists a minimal degree of acceptance (validation) from the design actors perspective and when changes in the model do not make any impact on the generated plans. After the short-term analysis, the generated plans are considered



valid, satisfying the goals and constraints specified in the model and having no inconsistencies or flaws.

In order to support the process described in Figure 2, the postDAM framework integrates a knowledge engineering tool (itSIMPLE (Vaquero et al. 2009)) for the modeling and re-modeling processes with a virtual prototyping environment. The former must provide a structured and disciplined design process while the latter must have a rich simulation environment that allows realistic representation of the real domain of application. In this work, we focus on domain-independent tools that are able to represent and model a variety of planning applications. It is worth to note that existing work and contributions in plan verification & validation, and plan execution & control (e.g. (Giannakopoulou et al. 2005; Fox, Howey, and Long 2005; Cesta et al. 2008)) could enhance, complement or fulfill the requirements of the short-term level, depending of the application.

## Second Layer: Long-term Plan Analysis

In real planning applications, distinct valid plans can solve the same planning problem; however, they might have different qualities and strategies. In some applications (such as space exploration (Cesta et al. 2008)) there are a number of criteria to evaluate a plan and complex trade-offs to be achieved. The definition of a quality metric must be made wisely and carefully in AI planning problems. The use of the appropriate quality metrics and the consideration of evaluation rationales are essential to the selection of a subset of valid plans that can be reliably executed. In real applications, trade-offs must be known and acknowledged.

The second layer of the postDAM framework aims to support plan evaluation and model refinement based on a set of specified metrics. Plan evaluation involves the analysis of quality metric values, plan classification, and rationale acquisition. The successive plan evaluations gathered in the long-term can provide important information that can be pieced together, accessed and explored. This information is the base for discovering hidden requirements, constraints, preferences and the real intentions of design actors (knowledge that was not identified during the virtual prototyping phase). Such discoveries feed the model refinement cycle. Differently from the first layer, the model refinement envisaged in this long-term analysis focuses on enhancing the plan quality as opposed to plan validity. Figure 3 illustrates the long-term plan analysis process proposed in the framework.

As illustrated in Figure 3, the first step in the layer refers to an initial plan evaluation based on a specified set of quality metrics. In this initial evaluation, the goal is to analyze the values of each metric, their weights (representing the importance among other metrics) and to provide a classification for each one of them (e.g., a satisfiability level ranging from bad to good, 0 to 1).

Depending on the specification and experts' familiarity with the domain, it is possible that some of the metrics might have predefined classification functions that map metric values to classification ranges. The work of (Rabideau, Engelhardt, and Chien 2000) describes an interesting approach for representing predefined metric classification functions in

which each metric is attached to a preference function that maps values to scores in the interval  $[0,1]$  (0 = unsatisfactory, 1 = satisfactory). This approach is used as a reference in the framework for representing predefined classification of quality metrics. If such predefined metric classification is available (for instance through a KE tool), the initial evaluation of the metrics can be made automatically. However, in most real scenarios predefined metrics functions are unknown. In fact, the complete set of necessary quality metrics might be unknown in advance. The lack of a complete set of metrics requires design participants to communicate and discuss the main aspects of their planning problem solutions during plan analysis.

As a second step, the framework introduces the process of acquiring plan metrics, plan classifications and plan evaluation rationales. Since the set of quality metrics might be incomplete, users can specify new ones during the initial plan evaluation. Based on individual metric classification and on the overall characteristics of the plan, design actors provide the final plan classification by using, for example, the interval  $[0,1]$ . Both metric and plan classification are done interactively with users.

Besides plan classification, the framework aims to acquire human-centric evaluation rationales. Different from existing work on rationales in planning (Polyank and Austin 1998; Wickler, Potter, and Tate 2006), we focus on acquiring human-centric rationales that emerge from user feedback, observations and justifications during plan evaluation. Based on general and individual criteria, interests, feelings and expectations, the rationales from plan evaluation generate explanations and justifications as to why a plan was classified into a specific quality level. Therefore, we extend here the concept of plan rationales described in planning literature with rationales that encompass "why a certain plan element does not or does satisfy a criterion" or "why a certain plan does not or does satisfy a preference". Moreover, these rationales could explain "why a certain metric does not or does satisfy a given criteria" and "what is the effect of a given plan characteristic or element in the plan quality" (e.g., it decreases or increases the quality). As an example of plan rationale, one might say that "the plan has a decreased quality because the robot left a block too close to the edge of the table" or "the plan has a high quality because the robot avoided repeatedly passing through the two most crowded areas of the building while cleaning it". We call these explanations *plan evaluation rationales*.

As illustrated in Figure 3, the acquired rationales for plan evaluations are checked to guarantee correctness and applicability to the plan being evaluated. Valid rationales are attached to the plan. Once analyzed and evaluated, the plan is then stored in a database of the postDAM framework, called *Plan Analysis Database*.

The *Plan Analysis Database* is an essential component of the postDAM framework and, in particular, of the long-term layer. The main role of the database is to support the reuse of rationales in each initial plan evaluation, i.e., the framework is responsible for accessing the database and looking up for rationales that can be applied. For example, if a previously analyzed plan in the database was annotated with a given

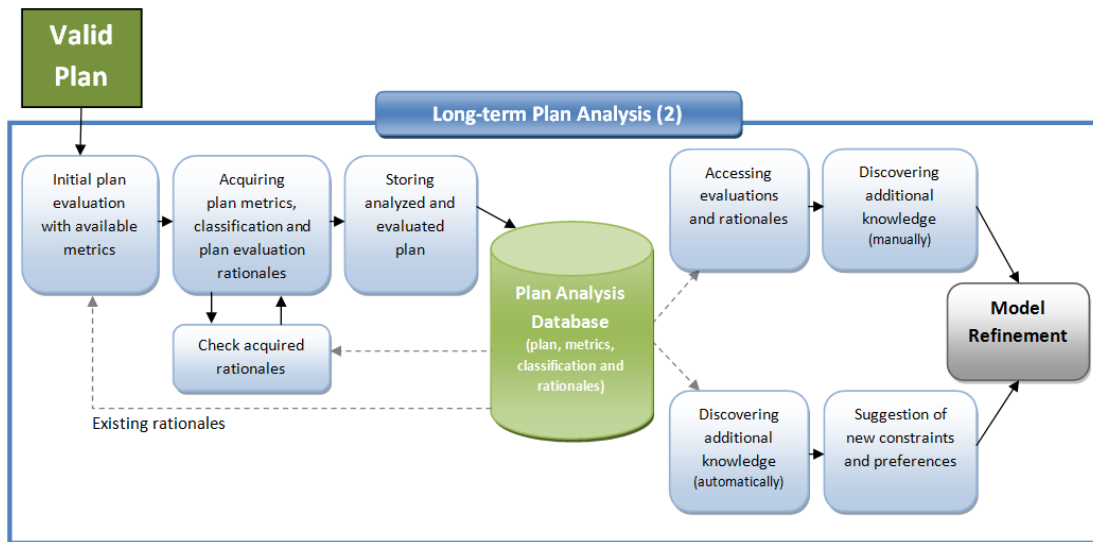


Figure 3: Long-term analysis in the postDAM framework

rationale that refers to a bad sequence of actions and if the plan currently being analyzed has the same bad sequence, the rationale can be reused to justify an initial classification during the evaluation.

Another role of the database is to support knowledge discovery in the long-term scenario. In the right side of Figure 3, we have split the exploration of the analysis data into two processes. The upper process refers to the manual access to the database and discovery of hidden requirements, preferences and constraints. The lower process refers to the automatic reasoning about plan evaluations and discovery (or extraction) of the hidden knowledge. Both processes address knowledge that directly impact in the quality of plans.

The manual process includes the access of the database to explore and study the information available from several plan analyses. A proper interface for accessing the data is required. By investigating and comparing different plans, classifications, and rationales, designers can perform modification to the model as they better understand actors' preferences, justifications and the correlations between a plan's properties and classifications.

Since manual identification of hidden requirements can be time consuming (but sometimes very efficient), an automatic process for knowledge extraction and suggested model adjustments becomes necessary. The postDAM includes such automatic process which can include specially designed algorithms and AI techniques. We do not get into the discussion of any algorithm or technique in this paper (an example is given in (Vaquero 2011)). Nevertheless, the input and output must be made clear. The input is a set of evaluated plans (along with their respective classification and rationales) and, if necessary, a domain and problem description. The output is a set of model modification suggestions to be analyzed by designers (e.g., a new precondition, post-condition, constraint, preference, metric, or action).

In order to support the process described in Figure 3,

the postDAM framework integrates the following: (1) the knowledge engineering tool itSIMPLE for plan evaluation (including metric acquisition, specification and classification), rationale acquisition, and manual discovery of hidden requirements; (2) a database for storing analyzed plans; and (3) AI algorithms and techniques for reasoning about plan evaluations and discovering model refinements. Besides these elements, it is necessary to consider a good representation language to capture plan structures, metrics, classification and rationales. According to (Polyank and Austin 1998), the most reasonable approach would be to consider new and existing extensions of Plan Ontologies. In this work, we propose the use of an ontology-based representation of plans and rationales to allow reasoning about plan evaluations. Such an ontology is an extension of existing plan ontologies that is able to support post-design analysis. The proposed ontology-based representation of plans and rationales is discussed in the implementation section.

### Third Layer: Cross-project Re-use Analysis

The two previous layers were designed to analyze a particular domain model. However, the experience gained in the design of one application can be applied to others with similar properties. The cross-project layer aims to abstract some of the experience gathered in the short and long-term analyses and make them available and reusable in the design of new planning applications. In this section, we focus on the reuse of quality metrics, rationales for plan evaluations and discovered knowledge.

Reusing past experience during design has long been used by other engineering areas, such as Software Engineering. The term *design pattern* is commonly used to refer to elements of reusable software. A design pattern is a general, reusable solution to a commonly occurring problem. Design patterns have been successfully used in software engineering to speed up design, improve quality, reduce cost, and

decrease problem fixing issues (Gamma et al. 1995).

Even though research on design patterns for AI Planning and Scheduling is scarce, studies like (Long and Fox 2000; Simpson et al. 2002) have shown that reusable elements can indeed be useful. Initial work on design patterns for Planning and Scheduling focuses on enhancing automated planners to produce plans faster through recognition of common patterns and the use of specialized heuristics (Long and Fox 2000; Clark 2001). The only work that focuses on the advantages of reusable knowledge during the design process is (Simpson et al. 2002). The most significant existing design patterns are: *transportation* encapsulating the behavior of mobile objects that traverse a network of locations (Long and Fox 2000); *construction* representing the behavior of objects that are built from other objects (e.g., assembly problems) (Clark 2001); and *bistate* referring to the behavior of on-off switches (Simpson et al. 2002). The existing design patterns are limited to classical problems, but this topic has great potential to evolve with richer domains.

Since design patterns already provide a structured foundation for reusing past experience, in the cross-project reuse analysis layer we focus on enhancing the patterns for planning with the knowledge gathered in previous plan evaluations performed in the short and long-term layers. Figure 4 illustrates the process proposed in the layer.

discussed

The main components of the layer described in Figure 4 are the *Plan Analysis Database* and the *Planning Design Patterns Database*. The former stores specific experiences acquired in plan evaluation cycles (in different domains), whereas the latter stores general encapsulated, reusable model elements for planning problems.

The first step represented in Figure 4 aims to generalize specific knowledge found in the plan analysis database and enrich the elements in the design patterns database. In this work we focus on making rationales and discovered knowledge available in design patterns. A matching process between design patterns and the existing knowledge from the plan analysis database is required. Conceptually, when a pattern is recognized in a particular domain model that has its plan analysis stored in the plan analysis database, the rationales, metrics and model modification referring to that particular pattern can be analyzed and transferred to the design patterns database. Since the plan analysis database can be highly dynamic (new data being stored in each analysis) an update cycle is necessary to filter new experiences and maintain the design patterns database. The techniques and methods for recognizing and updating design patterns are not in the scope of this paper. However, ontology matching techniques could be useful in such process (Euzenat and Shvaiko 2007). In fact, the use of an ontological representation of evaluated plans and rationales in the second layer provides a foundation for the application of such ontology matching techniques.

When a new design takes place for a planning application, existing patterns can potentially be used and explored. The framework is responsible for providing a design pattern catalog during the modeling phase. When designers import a selected pattern, all information can be re-used (e.g., quality

metrics, preferences, additional constraints, and evaluation rationales). Design patterns might also be applied in existing models. In these cases, the framework is also responsible for identifying existing patterns in the model, matching the available knowledge and bringing the attached experience, if it is not yet explicitly specified in the model.

In order to support the process described in Figure 4, the postDAM framework conceptually integrates the following: (1) the knowledge engineering tool itSIMPLE for performing the update cycle on the planning design patterns database and for providing such patterns during application design; (2) a database of design patterns; and (3) mechanisms and techniques for recognizing and matching model components. Note that it is necessary to consider formalisms and languages for representing and reasoning about design models. The KE tool has an important role to provide the right reusable information at the correct phase of the design.

## Implementation in itSIMPLE

In order to support the different layers of plan analysis, the postDAM framework has been implemented as a post-design tool for AI planning that integrates several tools, including an open-source KE tool, an open-source 3D content creation for virtual prototyping, an ontology-based reasoning system, and a database. The core of this integration is the KE tool, itSIMPLE (Vaquero et al. 2009). The KE tool has an important role in every layer of the framework; it supports human interactions, from metric and rationale acquisition to knowledge discovery and re-modeling. We have been designing a series of new extensions of itSIMPLE to fulfill such roles.

Currently we have completely implemented the first layer of the framework and partially implemented the second layer. Regarding the first layer, we have implemented an extension of itSIMPLE, integrating the KE tool with the plan validation VAL (Howey, Long, and Fox 2004) for plan verification and a virtual prototyping environment called Blender<sup>1</sup> for plan simulation. The work described in (Vaquero, Silva, and Beck 2010) provides a complete description of the short-term plan analysis process. The work introduces a re-modeling support tool that uses virtual prototyping techniques for plan analysis. While observing plan execution and evaluation in a 3D environment environment, users detect inconsistencies, unexpected behaviors and hidden requirements that guide a re-modeling process. In (Vaquero, Silva, and Beck 2010) we show, through experiments with benchmark domains, that the impact of a re-modeling tool on the planning process can be quite impressive, affecting not only the plan quality, but run-time and solvability. In these experiments, we have detected missing constraints on the definition of operators, as well as new predicates which were added to avoid specific scenarios. Such experiments open a large road for the investigation and understanding of knowledge captured on post-design.

In the long-term plan analysis, itSIMPLE (Vaquero et al. 2009) has also a central role; it supports human interactions and reasoning processes, from plan evaluation to ratio-

<sup>1</sup>Blender, available at [www.blender.org](http://www.blender.org).

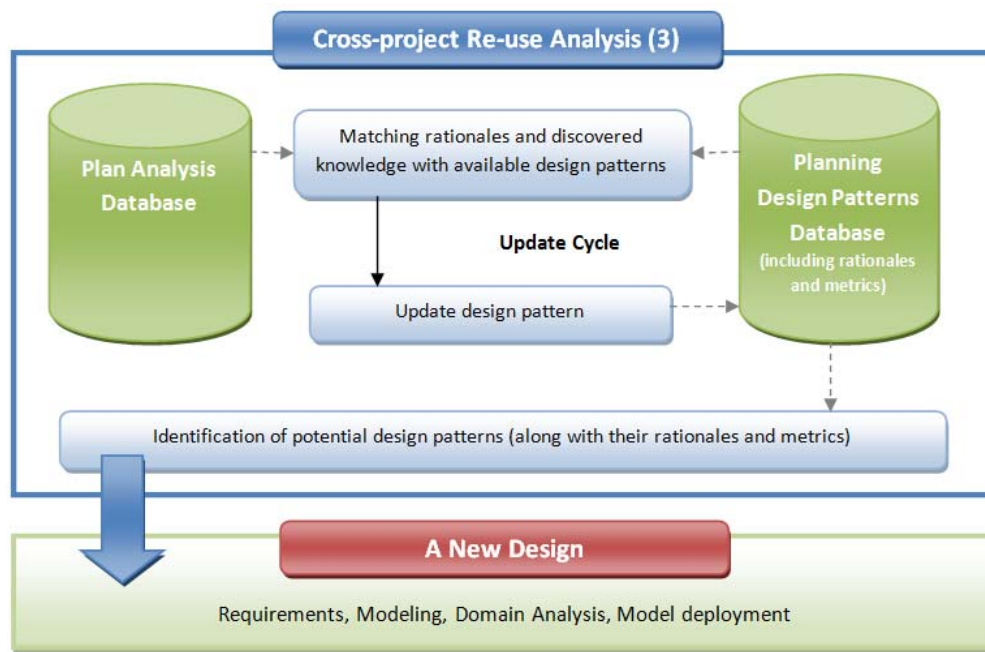


Figure 4: Cross-project re-use analysis in the postDAM framework

nales acquisition and knowledge extraction. We have implemented the following processes: the interactive plan evaluation in which users are able to modify and adjust plan classification based on their impressions; and the acquisition and re-use of plan evaluation rationales. This implementation is describe in the work (Vaquero, Silva, and Beck 2011).

In order to give here an overview of the implemented components of the second layer, we have integrated itSIMPLE with a database system, PostgreSQL,<sup>2</sup> and with a reasoning system, based on tuProlog.<sup>3</sup> The database is responsible for storing plan evaluations and their respective rationales, while the reasoning system is responsible for inferring rationale re-use in new plan evaluations. In order to capture, represent and re-use the rationales, we have used an extension of the Process Specification Language (PSL). PSL is an expressive ontological representation language of processes (plans), including activities and the constraints on their occurrences. PSL has been designed as a neutral interchange ontology to facilitate correct and complete exchange of process information among manufacturing systems such as scheduling, process modeling, process planning, production planning, simulation, project management, workflow, and business process applications (Grüniger and Koppena 2005).

We have not covered the development and investigation of all three layers of plan analysis and knowledge re-use. We have focused on the short and long-term analyses while the third layer is left for future work.

## Conclusion

This paper presents the conceptual model of the postDAM framework. We have emphasized the structure of the proposed framework, i.e., its different layers of plan analysis that provide a basis for the model modification and refinement. Such refinement aims at the improvement of plan quality and, as a consequence, the planning performance. The layers of analysis include a short-term support for detection of missing requirements, a long-term support for identification of hidden knowledge, a cross-project analysis and software tools for knowledge re-use.

We have described the different tools that can be integrated in the framework to assist the discovery of missing requirements, to support evaluation rationale acquisition and re-use, and to guide the model refinement cycle. Not all processes have been implemented, but some of them have been developed. In previous work we implemented for example the first layer of postDAM. We demonstrated in (Vaquero, Silva, and Beck 2010) that following a careful post-design analysis, we can improve not only plan quality but also solvability and planner speed. However, in this paper we have shown the big picture of our research on post-design, aiming at supporting the different stages of plan analysis. In a real planning application, the analysis process that follows design becomes essential to have the necessary knowledge represented and adapted in the model.

## References

- Cecil, J., and Kanchanapiboon, A. 2007. Virtual engineering approaches in product and process design. *The International Journal of Advanced Manufacturing Technology* 31(9-10):846–856.

<sup>2</sup>PostgreSQL is available at <http://www.postgresql.org/>.

<sup>3</sup>tuProlog: see <http://alice.unibo.it/xwiki/bin/view/Tuprolog/>.

- Cesta, A.; Finzi, A.; Fratini, S.; Orlandini, A.; and Tronci, E. 2008. Validation and verification issues in a timeline-based planning system. In *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS 2008) Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.
- Clark, M. 2001. Construction domains: A generic type solved. In *Proceedings of the 20th U.K. Planning and Scheduling Workshop*.
- Euzenat, J., and Shvaiko, P. 2007. *Ontology matching*. Heidelberg (DE): Springer-Verlag.
- Fdez-Olivares, J.; C  zar, J.; and Castillo, L. 2009. OncoTheraper: Clinical Decision Support for Oncology Therapy Planning Based on Temporal Hierarchical Tasks Networks. In Ria  o, D., ed., *Knowledge Management for Health Care Procedures*, volume 5626 of *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer-Verlag. 25–41.
- Fox, M.; Howey, R.; and Long, D. 2005. Exploration of the Robustness of Plans. In *Proceedings of ICAPS 2005 Verification and Validation of Model-Based Planning and Scheduling Systems workshop*.
- Gamma, E.; Helm, R.; Johnson, R. E.; and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.
- Giannakopoulou, D.; Pasareanu, C. S.; Lowry, M.; and Washington, R. 2005. Lifecycle Verification of the NASA Ames K9 Rover Executive. In *Proceedings of ICAPS 2005 Verification and Validation of Model-Based Planning and Scheduling Systems workshop*.
- Gr  ninger, M., and Kopena, J. B. 2005. Planning and the Process Specification Language. In *Proceedings of ICAPS 2005 workshop on the Role of Ontologies in Planning and Scheduling*, 22–29.
- Howey, R.; Long, D.; and Fox, M. 2004. VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL. In *ICTAI'04: Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence*, 294–301. Washington, DC, USA: IEEE Computer Society.
- J  nsson, A. K. 2009. Practical Planning. In *ICAPS 2009 Practical Planning & Scheduling Tutorial*.
- Long, D., and Fox, M. 2000. Automatic Synthesis and use of Generic Types in Planning. In *Artificial Intelligence Planning and Scheduling AIPS-00*, 196–205. Breckenridge, CO: AAAI Press.
- McCluskey, T. L., and Simpson, R. M. 2006. Tool support for planning and plan analysis within domains embodying continuous change. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS 2006) Workshop on Plan Analysis and Management*.
- McCluskey, T. L. 2002. Knowledge Engineering: Issues for the AI Planning Community. *Proceedings of the AIPS-2002 Workshop on Knowledge Engineering Tools and Techniques for AI Planning*. Toulouse, France 1–4.
- Myers, K. L. 2006. Metatheoretic Plan Summarization and Comparison. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS-06)*. Cumbria, UK: AAAI Press.
- Polyank, S., and Austin, T. 1998. Rationale in Planning: Causality, Dependencies and Decisions. *Knowledge Engineering Review* 13(3):247–262.
- Rabideau, G.; Engelhardt, B.; and Chien, S. 2000. Using generic preferences to incrementally improve plan quality. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*. Breckenridge, CO: AAAI Press.
- Simpson, R. M.; McCluskey, T. L.; Long, D.; and Fox, M. 2002. Generic Types as Design Patterns for Planning Domain Specification. In *Knowledge Engineering Tools and Techniques for AI Planning: AIPS'02 Workshop*.
- Vaquero, T. S.; Romero, V.; Tonidandel, F.; and Silva, J. R. 2007. itSIMPLE2.0: An integrated Tool for Designing Planning Environments. In *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS 2007)*. Providence, Rhode Island, USA: AAAI Press.
- Vaquero, T. S.; Silva, J. R.; Ferreira, M.; Tonidandel, F.; and Beck, J. C. 2009. From Requirements and Analysis to PDDL in itSIMPLE3.0. In *Proceedings of the Third International Competition on Knowledge Engineering for Planning and Scheduling, ICAPS 2009*, 54–61.
- Vaquero, T. S.; Silva, J. R.; and Beck, J. C. 2010. Improving Planning Performance Through Post-Design Analysis. In *Proceedings of ICAPS 2010 workshop on Scheduling and Knowledge Engineering for Planning and Scheduling (KEPS)*, 45–52.
- Vaquero, T. S.; Silva, J. R.; and Beck, J. C. 2011. Acquisition and Re-use of Plan Evaluation Rationales on Post-Design. In *Proceedings of ICAPS 2011 workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*.
- Vaquero, T. S. 2011. *Post-Design Analysis for AI Planning Applications*. Ph.D. Dissertation, Polytechnic School of the University of S  o Paulo, Brazil.
- Wickler, G.; Potter, S.; and Tate, A. 2006. Recording Rationale in <I-N-C-A> for Plan Analysis. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS 2006) Workshop on Plan Analysis and Management*.

# **System Demonstrations**

# An Interactive Tool for Plan Visualization, Inspection and Generation

Alfonso E. Gerevini and Alessandro Saetti

Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Brescia, Brescia, Italy  
{gerevini, saetti}@ing.unibs.it

## Abstract

In mixed-initiative planning systems, humans and AI planners work together for generating satisfactory solution plans or making easier solving hard planning problems, which otherwise would require much greater human planning efforts or much more computational resources. In this approach to plan generation, it is important to have effective plan visualization capabilities, as well to support the user with some interactive capabilities for the human intervention in the planning process. This paper presents an implemented interactive tool for the visualization, generation and revision of plans. The tool provides an environment through which the user can interact with LPG, a state-of-the-art domain-independent planner, and obtain an effective visualization of a rich variety of information during planning, including the reasons why an action is being planned or why its execution in the current plan is expected to fail, the trend of the resource consumption in the plan, and the temporal scheduling of the planned actions. Moreover, the proposed tool supports some ways of human intervention during the planning process to guide the planner towards a solution plan, or to modify the plan under construction and the problem goals.

## Introduction

In the AI-planning literature, many approaches to plan generation or revision combining automated techniques with human-driven decisions have been proposed, e.g., (Allen & Ferguson 2002; Cox & Zhang 2005; Cox & Veloso 1997b; Currie & Tate 1991; Ferguson *et al.* 1996; Ferguson & Allen 1994; Myers *et al.* 2003; Tecuci 2003; Veloso *et al.* 1997; Zhang 2002). The rationale of these mixed-initiative approaches is that the collaborative joint work of a human and an AI planner can be much more effective than either human planning or fully automated planning alone, in terms of problem solvability, planning speed, or user satisfaction about the quality of the generated solutions.

Mixed-initiative planning systems with interactive user-machine interfaces can be essential for making modern planning technology usable in real-world applications (e.g., (Bresina *et al.* 2005; Castillo *et al.* 2005)). For instance, consider the Mars Exploration Rovers Mission project, which involved two NASA rovers for the ground exploration of Mars (Bresina *et al.* 2005). As argued in (Bresina *et al.* 2005), the complexity of this project and the aggressive operation plans made using an automated tool for generating

the daily activity plans necessary. However, also the human involvement during the planning process was needed. The activity plan needed to be presented, critiqued and, hopefully, accepted. Another concern in this application was the infeasibility of formally encoding and effectively utilizing during automated plan generation all the knowledge that characterizes plan quality.

As argued by Ferguson & Allen (1994), in mixed-initiative planning the description of a plan that the system provides to the user should be richer than just a list of action names with the associated temporal information, e.g., for each action, its start time and expected duration, as in PDDL2.1 plans (Fox & Long 2003). In particular, for an interactive planning tool it is essential to have specialized user-interface capabilities explaining the reasons why an action has been planned, or why, in the context of the plan under consideration, it is expected that its execution will fail. Moreover, it is desirable that the system supports an effective visualization and inspection of the plan, which helps the user to understand the ongoing planning process, the decisions taken by the planner, and the feasibility and quality of the solution plan proposed by the system to the user.

An adequate description of the current plan and the planning process that led to its generation is very useful to the user for deciding the possible human interventions in order to (a) guide the planning process for a faster generation of a solution, (b) constrain the plan under construction so that, e.g., it contains certain actions or crosses some particular intermediate states specified by the user, or (c) modify the problem goals during planning. However, plan visualization is a scarcely investigated area in AI planning, and only very few planning systems currently incorporate a user interface with effective plan visualization capabilities, e.g., (Lino & Tate 2004; Lino *et al.* 2005; Daley *et al.* 2005).

In the recent years, automated domain-independent planning has dramatically improved in terms of planning performance and especially speed. However, to the best of our knowledge, the state-of-the-art domain-independent planners are not equipped with an interactive tool supporting plan visualization and mixed-initiative planning capabilities. This limits their applicability to real-world applications where, often, the domain experts want to analyze the plan generated by the planner and possibly refine some portion(s)

of it, before committing to its execution. The output information of all recent efficient domain-independent planners is given to the user only in a simple high-level textual form, indicating some very general information about the planning process and describing the generated plan as a simple list of actions with the relative scheduled start times and durations. Moreover, there is no possibility of human intervention to guide the planning process, to inspect the generated plan and to possibly revise it.

In this work, we focus on a successful approach to fully automated domain-independent planning with the aim of making it more suitable for mixed-initiative planning. This approach is implemented in the well-known LPG planner (Gerevini *et al.* 2003; 2006; 2009), a state-of-the-art planner supporting the standard language PDDL2.2 (Hoffmann & Edelkamp 2000) and belonging to the so-called “satisficing” style of planning, which in the last ten years has received significant interest in the AI planning community.<sup>1</sup>

The main contribution of the work presented in this paper is a new interactive environment for the visualization, inspection, generation and revision of plans. The tool implementing this environment is called InLPG, and uses LPG as the underlying automated planning system. The user can interact through InLPG with the underlying planner about:

- the plan under construction or revision (e.g., the user requests a temporal scheduling for some planned action that is different from the one decided by the planner, or imposes that some particular action must be in the solution plan);
- the planning problem under consideration (e.g., the user communicates to the system that some particular goal can be ignored, or adds some new goals);
- the automated planning process (e.g., the user pauses the search of the planner and modifies the planner decision about the next search state to be explored).

Moreover, the proposed tool supports plan visualization through various (dynamic) views of the plan, such as: a Gantt chart of the planned actions, a constraint graph for the temporal constraints in the plan, a resource graph for monitoring and describing the trend of the resource consumption in the plan, a graphical representation of the main data structure representing the search states and partial plans explored by the automated planner, several plots describing the trend of the search process.

The paper is organized as follows. Section 2 introduces the necessary background about LPG planner. Section 3 describes the main components of the proposed interactive planning tools. Finally, the last section is devoted to the conclusions and possible future work.

## The LPG Planner

LPG is a versatile system that can be used for plan generation, plan repair and incremental planning in PDDL2.2 do-

<sup>1</sup>Differently from optimal planning, satisficing planning addresses the problem of quickly computing a good quality solution plan rather than a necessarily optimal solution plan (which usually is a more expensive computational task).

1. Set  $\mathcal{A}$  to the action graph containing only  $a_{start}$  and  $a_{end}$ ;
2. *While* the current action graph  $\mathcal{A}$  contains a flaw or a certain **number of search steps** is not exceeded *do*
3.   **Select a flaw**  $\sigma$  in  $\mathcal{A}$ ;
4.   Determine the search **neighborhood**  $N(\mathcal{A}, \sigma)$ ;
5.   Weight the elements of  $N(\mathcal{A}, \sigma)$  using a **heuristic function**  $E$ ;
6.   Choose a graph  $\mathcal{A}' \in N(\mathcal{A}, \sigma)$  according to  $E$  and **noise**  $n$ ;
7.   Set  $\mathcal{A}$  to  $\mathcal{A}'$ ;
8. *Return*  $\mathcal{A}$ .

Figure 1: High-level description of LPG’s search procedure.

mains (Hoffmann & Edelkamp 2000). The planner is based on a stochastic local search procedure that explores a space of partial plans represented through *linear action graphs* (LA-graph), which are variants of the very well-known planning graph (Blum & Furst 1997). A linear action graph is a directed acyclic leveled graph that alternates between a proposition level, i.e., a set of domain propositions, and an action level, i.e., one ground domain action and a set of special dummy actions, called “no-ops”, each of which propagates a proposition of the previous level to the next one. If an action is in the graph, then its preconditions and positive effects appear in the corresponding proposition levels of the graph. Moreover, a pair of propositions or actions can be marked as (permanent) mutually exclusive at every graph level where the pair appears (for a detailed description, see (Gerevini *et al.* 2003)). If a proposition appears at a level of the action graph, then its no-op appears at that level and at every successive graph level until a level containing an action that is marked mutually exclusive with it is reached (if any). The initial and last levels of every action graph contain the special actions  $a_{start}$  and  $a_{end}$ , where the effects of  $a_{start}$  are the facts of the problem initial state and the preconditions of  $a_{end}$  are the problem goals.

The plan represented by an action graph is a valid plan if and only if the graph contains no *flaw*, where, intuitively, a flaw is an action in the graph with a precondition that is not supported by the propagation of an effect of another action appearing at a previous graph level.

Starting from the initial action graph containing only two special actions representing the problem initial state and goals, respectively, LPG iteratively modifies the current graph until there is no flaw in it or a certain bound on the number of search steps is exceeded. LPG attempts to resolve flaws by inserting into or removing from the graph a new or existing action, respectively. Figure 1 gives a high-level description of the general search process performed by LPG. Each search step *selects a flaw*  $\sigma$  in the current action graph  $\mathcal{A}$ , defines the elements (modified action graphs) of the *search neighborhood* of  $\mathcal{A}$  for repairing  $\sigma$ , weights the neighborhood elements using a *heuristic function*  $E$ , and chooses the best one of them according to  $E$  with some probability  $n$ , called the *noise parameter*, and randomly with probability  $1 - n$ . Because of this noise parameter, which helps the planner to escape from possible local minima, LPG is a randomized procedure.



## Architecture of InLPG

In this section, we describe the architecture of the proposed interactive tool for the visualization, generation and revision of plans, along with its main components.

### Architecture Overview

The architecture of InLPG is sketched in Figure 2. It consists of five main components that are integrated in the LPG planner:

- the *input module*, which inputs the files containing the description of the planning problem under consideration;
- the *search process monitor*, which monitors the search process, and at each step of the search displays the information about the current search state;
- the *search state monitor*, which provides different views of the current state during the search process;
- the *plan editor*, which provides some tools for human-driven changes to the plan under construction;
- the *search process editor*, which provides some tools for human-driven changes to the search process and to the current planning problem.

Our environment includes an *open-controllable* version of LPG, i.e., all the decision points of the search procedure sketched in Figure 1 can be controlled by an external process that, in our context, is under the control of a human user. In particular, at each search step, the user can select a plan flaw to repair (step 3 of the procedure in Figure 1), modify the definition of the search neighborhood (step 4), and select a graph modification among those that generate the elements in the search neighborhood (step 6). Basically, LPG runs as a separate process, and it communicates with the rest of the environment through socket messages. The decisions about the search process taken by the user through InLPG overwrites the decisions taken by the heuristics of LPG.

Figures 3 and 4 shows two screenshots of the user-interface. The left frames show the Gantt chart of the plan computed at the 368th search step (upper screenshot) and the trend of some resources during the execution of the plan (bottom screenshot). The plan is flawed, because it contains actions that cannot be executed (the darker boxes in the Gantt chart, which in the actual screen are red). The information in this frame can be moved to the secondary frame (right frame), as displayed by the bottom screenshot, or into different windows. (This latter option is particularly useful if the user wants to compare different plans.) The Gantt chart in this frame also displays the temporal constraints between some actions in the plan, that can be activated or deactivated in the chart by clicking the action boxes.

The quality of the plan is automatically measured according to the metric expression specified in the problem formulation. In this example, the quality is defined as the duration of the plan, and for the displayed plan it is 1350.

The right hand side of the upper screenshot contains four plots. The first three plots (starting from the top) show, for each search step, the number of flaws (1st plot), the number of actions (2nd plot) and the makespan of the plan con-

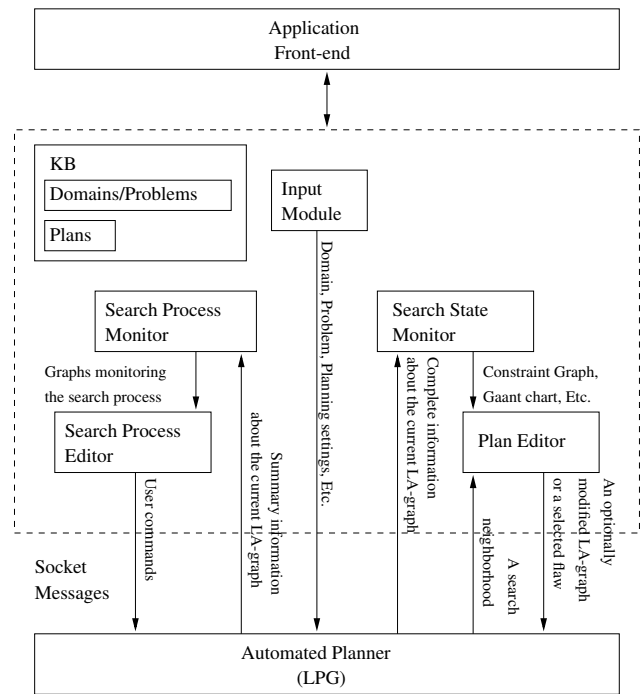


Figure 2: A sketch of the main components of InLPG and their interactions.

structed at the corresponding search step (3rd plot). The bottom plot informs the user about the trend of the quality of the solutions computed so far (LPG is an incremental planner computing a sequence of plans with increasing quality): for the example of Figure 3, LPG first found one solution with quality 7836.6 using 0.171 CPU-seconds; subsequently, a better solution with quality 4868.6 was found using about 0.4 CPU-second, and finally another slightly better solution with quality 4522.8 was generated using about 1.5 CPU-second.

In the next subsections, we will give a detailed description of the components integrated in our environment.

### Input Module

By using standard acquisition tools the user inputs the basic planning information to our environment: (i) a PDDL *domain file*, containing the action schemata and the predicates of the planning problem to solve; (ii) a PDDL *problem file* describing the problem initial state and goals; and, optionally, (iii) a plan file containing the PDDL description of a plan taken from a plan library. The possibility of loading a plan is particularly useful for solving plan adaptation problems (Gerevini & Serina 2000), in which the input plan is modified to become a solution of the planning problem. In addition, the user can change the default values of some technical parameters of the search algorithm implemented in LPG. A complete list of such parameters is described in (Gerevini *et al.* 2004).

After the necessary information has been acquired, the input module verifies the syntax of the PDDL files, and, if they are syntactically correct, it sends a message to LPG in or-

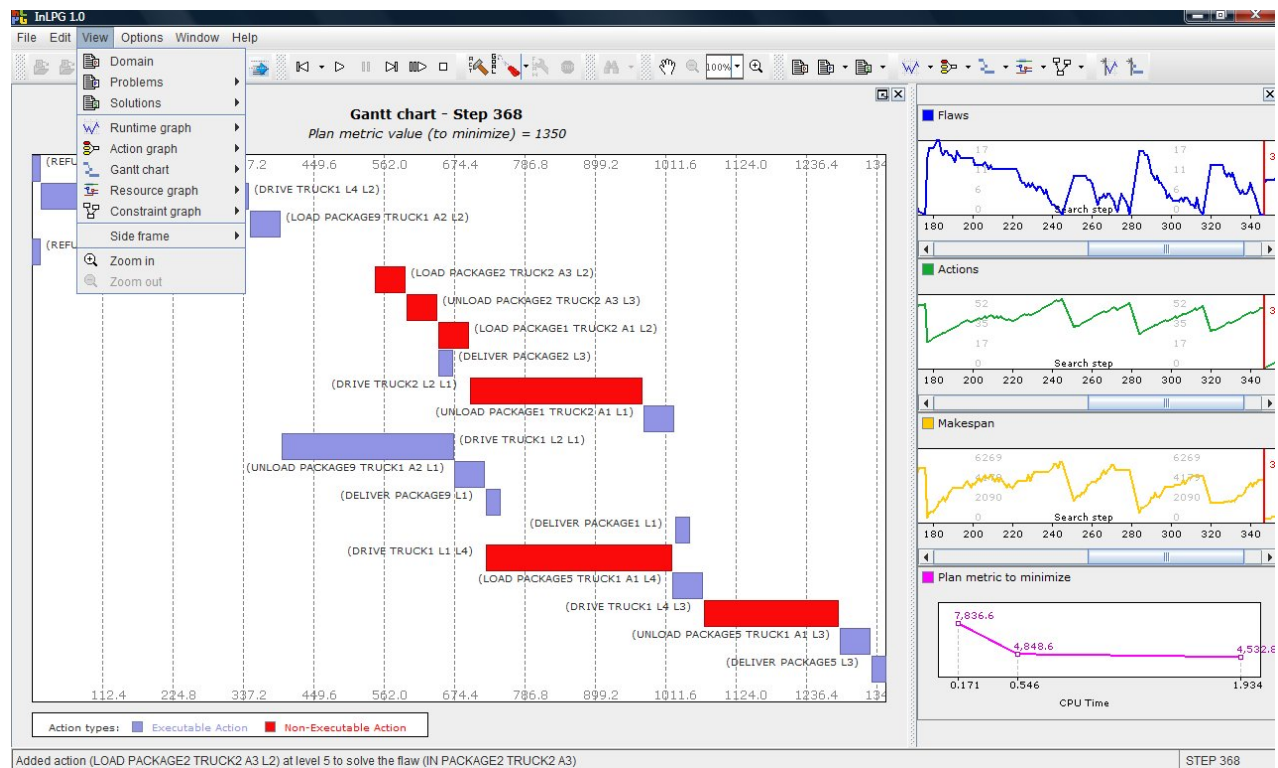


Figure 3: A screenshot of the graphical user interface of InLPG showing the Gantt chart of the current partial plan (main frame) and four graphs for monitoring the search process (right frame).

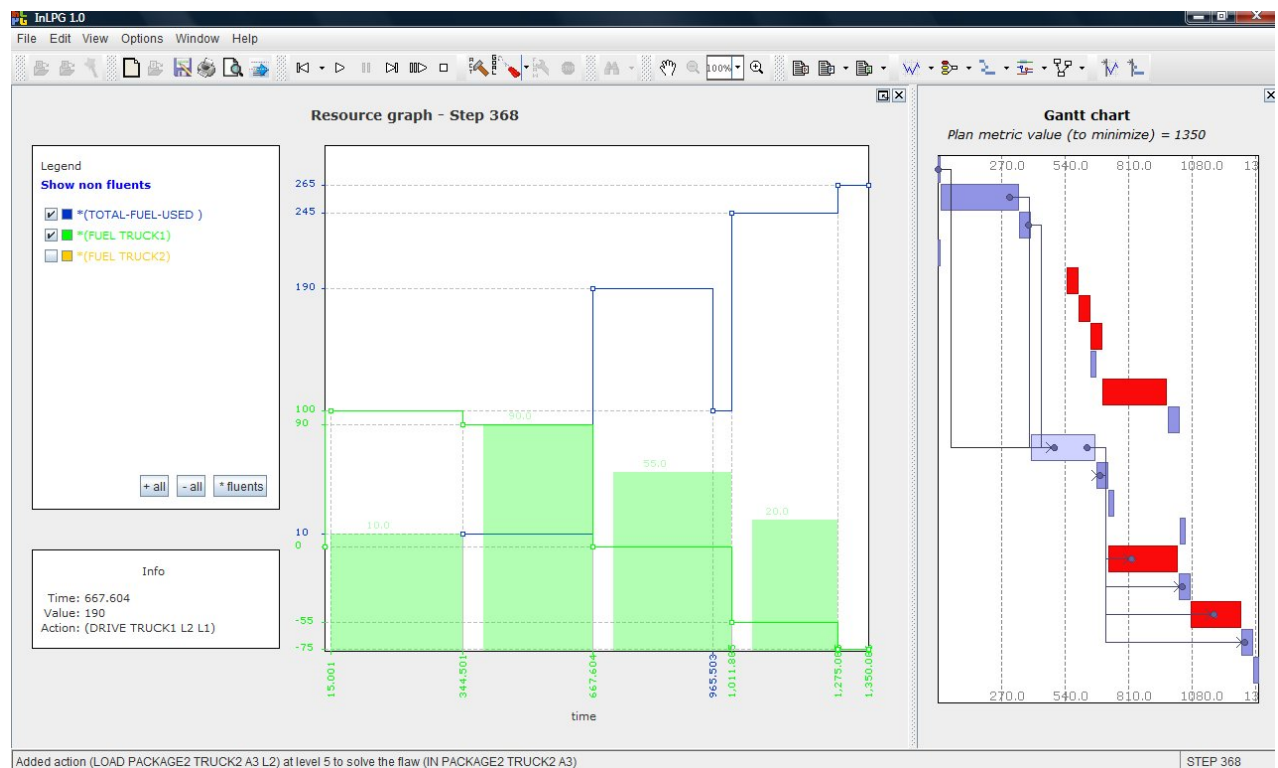


Figure 4: A screenshot of the graphical interface of InLPG showing the resource graph and the Gantt chart of the current plan.

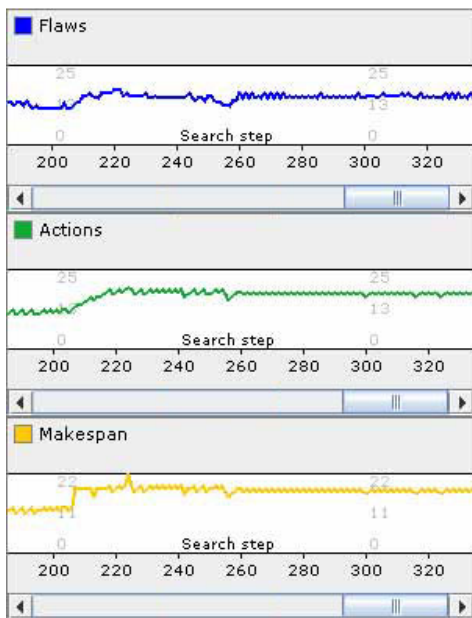


Figure 5: A portion of the screen of the user interface showing an example of the plots displayed by the search process monitor: the number of flaws (upper plot), the number of actions in the current LA-graph (middle plot), and the number of time steps of the plan under construction (bottom plot). On the  $x$ -axis we have the number of performed search steps.

der to start a planning process for solving the given planning problem.

### Search Process Monitor

At each search step, LPG sends a message to the search process monitor containing the following basic information about the current LA-graph: the number of flaws in the LA-graph, the number of actions in the represented plan and the makespan of this plan. The search process monitor processes this information and plots the corresponding graphs in order to visualize a variety of information about the ongoing search process. For example, if the user sees that the number of actions or the plan makespan is much higher than the desired value, then he is informed that the search process is most likely visiting a portion of the search space that is faraway from the portion where the desired solution is located. Moreover, if the number of flaws does not decrease with the search steps, then the search might be trapped in a local minimum or plateau. Figures 3 (right frame) and 5 show two examples of these plots.

Identifying which are the search steps where the planner makes wrong decisions that are crucial for the success of the search process could be a difficult task. The plots of the search process monitor help the user to identify them. The intervention of a human to revise the wrong decisions made by the planner for these steps could be very important in order to effectively guide the process towards a solution plan.

In our context, when the search process visits a portion of the space which contains no solution LA-graph, the number of flaws does not significantly decrease. The plot of the number of flaws can indicate this problematic behavior in which the planner is continuously making wrong decisions, and hence it can suggest that a human-driven choice could improve the search.

For example, according to the right frame of Figure 3, the planning system is working well: at several search steps the number of flaws in the current LA-graph is zero.<sup>2</sup> On the contrary, according to the plots about the number of actions, flaws and duration of the plan in Figure 5, LPG is not approaching a solution LA-graph.

### Search State Monitor

When the user intervenes in the search process of LPG, e.g., by pausing the search process and undoing the search steps back to a certain search state (LA-graph), LPG sends a detailed description of the current LA-graph to the search state monitor. The search state monitor processes such information by computing the following information:

- a complete graphical representation and some compact representations of the current LA-graph,
- a graphical representation of the temporal constraints involving the actions in the plan represented by the current LA-graph;
- a textual (PDDL-like) description of the plan represented by the current LA-graph;
- a Gantt chart of the actions in the current plan; and
- a graph showing the trend of the involved resources during the plan.

All these graphs are dynamic. At each search step (forward or backward), they are automatically recomputed and, in order to guarantee their readability, they are automatically scaled and appropriately displayed. For example, at each search step, the temporal constraints involving the actions in the plan change because either an action is removed or inserted, and thus the search state monitor recomputes an appropriate graphical organization of the nodes in the revised constraint graph, in order to avoid edge crossing and to reduce the edge length. The new constraint graph is computed by GRAPHVIZ (<http://www.graphviz.org/>), an automated tool for layered drawing of directed graphs. Moreover, the nodes of the graphs can be clicked to obtain information on the represented action or to change some property of the action (e.g., a new start time for the represented action). Figures 3, 4 and 6 give examples of the Gantt chart, the graph of the resources and the linear-action graph.

By looking at the graphical representations of the computed plan, the user can evaluate the current search state and

<sup>2</sup>When a search step reaches an LA-graph with no flaw, the planner has found a valid plan. However, this plan is given in output only if its quality improves the quality of the previous output plan. In the example of Figure 3, some valid plans are computed, but only the one found at about the 350th step is given as the third output plan.

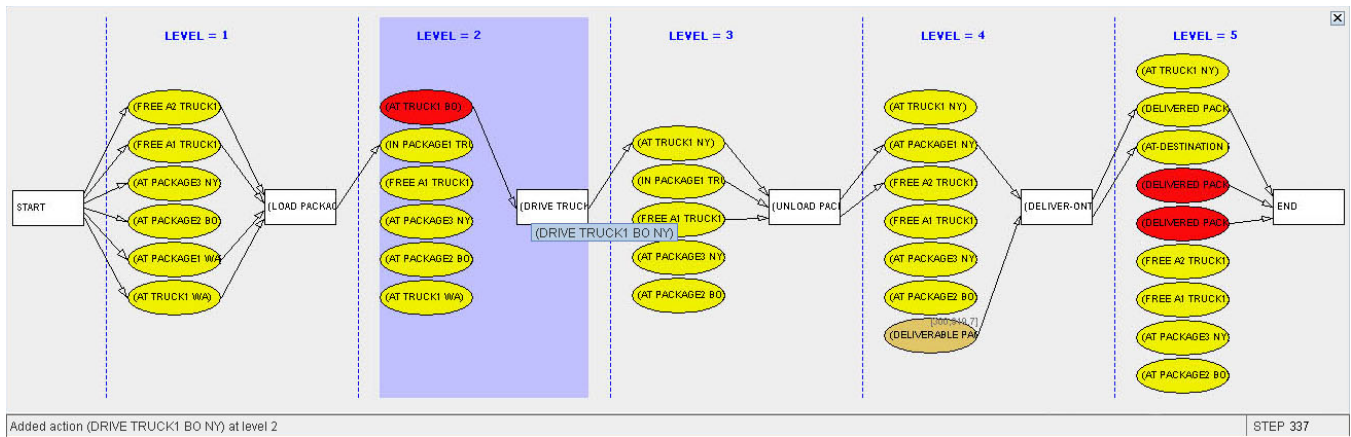


Figure 6: A portion of the screen of the user interface showing a compact representation of the current LA-graph. Square nodes are action nodes; elliptical nodes are fact nodes. Dark elliptical nodes (red nodes in the actual screen) are plan flaws (unsatisfied action preconditions). For lack of space, the label of some nodes is abbreviated. By moving the mouse on a node, a tooltip displays the corresponding full label. The darkened level (blue area in the actual screen) is the level of the last change performed by the planning process.

realize (or at least hypothesize) how the current plan has to be modified. In particular, the user interface highlights the graphical objects corresponding to plan actions that are not executable, and provides information exploiting why they are not executable (for instance, because of a precondition is not satisfied, a scheduling constraint is violated, or not enough resources are available in the state where the action is executed).

### Plan Editor

The plan editor is activated when the user revises the plan under construction. In a mixed-initiative planning context, the possibility for the user to inspect a plan and revise it through a plan editor is useful both during the process of constructing high quality plans and during the process of adapting an existing plan to satisfy the requirements of a new planning problem.

The plan editor allows the user to remove undesired actions, to add new actions (possibly satisfying some preconditions that currently are unsatisfied), and to reschedule an action in the plan. In the following, we describe the tools provided by the plan editor for supporting these plan revisions. If the user adds (removes) an action from the current plan, the plan editor sends a special message to LPG containing the desired (undesired) action selected by the user. Then, the planner computes a new LA-graph obtained from the previous one by adding (removing) the corresponding action node. Similarly, the plan editor allows the user to constrain the start time of an action in the plan to be after a desired time, and the end time of an action to be before a desired time. Moreover, the plan editor allows the user to restart the search from an empty plan, instead of continuing it from the current plan, which can be a good option when the current plan contains too many undesired actions or too many actions violating the desired scheduling constraints.

When the plan under construction is not valid, the plan

editor allows the user to repair “by hand” a flaw in the current partial plan. This interaction requires the usage of some handshaking messages. The plan editor sends a message to the planner containing the flaw selected by the user; then, LPG sends a new message to the component containing the elements in the search neighborhood for the selected flaw. The plan editor displays such a neighborhood (each element is compactly represented by the corresponding action addition/removal that eliminates the flaw under consideration), while the user selects an element from the neighborhood (for an example, see Figure 7). Finally, the plan editor sends a message describing the graph modification selected by the user to the planner, which then computes a new LA-graph obtained by applying the selected graph modification.

### Search Process Editor

The search process editor allows the user to control the progress of the search. The user can inspect and run the search by different modalities: with “no interruption”, “forward step by step”, “forward multi steps” and by “backward step by step”. For instance, the user can “pause” the search at any time, inspect the current plan and LA-graph and then continue the search step by step, i.e., the search progresses only one step and then waits that the user clicks the command button to proceed for the next step. If the user observes that the heuristics of LPG make an incorrect choice when repairing the selected flaw, the search process editor allows her to move the search one step *backward*, so that she can intervene and forces the planner to make an alternative decision among a set of alternatives provided by the system. The backward step-by-step modality can be repeated to undo the search back to any of the last  $k$  states previously generated, where  $k$  can be set by the user. The multi-step modality is very useful to obtain a graphical animation of the search progress. Under this modality, for each search step all graphs provided by the environment are automati-



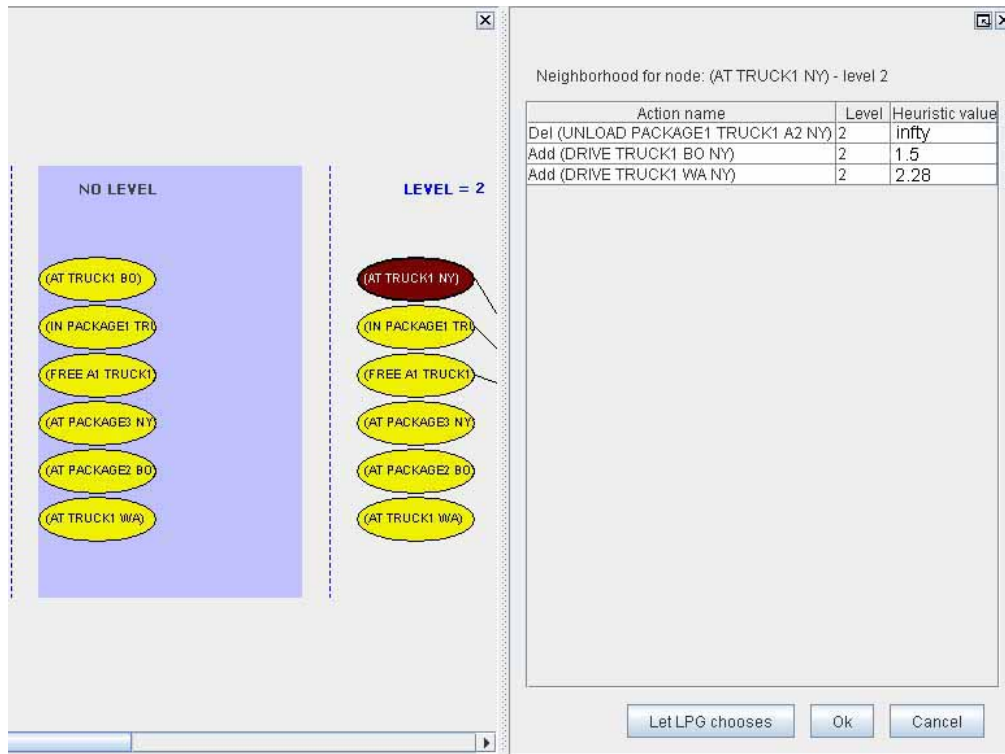


Figure 7: Two frames of the user interface containing a portion of the current LA-graph (left frame) and the search neighborhood of such a graph for repairing the flaw (attruck1NY) at level 2 of the LA-graph (right frame). The user can choose one of the three indicated possible graph modifications or let LPG choose using the heuristic value.

cally updated and re-displayed after  $k$ -milliseconds, where  $k$  is the speed of the animation that can be set by the user.

Moreover, the search process editor gives the user a tool for affecting the future search steps of the planning process by modifying the definition of the search neighborhood for every flaw in the current plan. For example, by inspecting the (partial) plan computed so far, the user realizes that, in order to achieve a desired solution plan, some actions should never be removed from the plan under consideration. The search process editor allows her to specify this constraint to the search, and, in this case, the search process editor sends a special message to the planner imposing that in the rest of the search process the removal of these actions will not be part of any search neighborhood.

The search process editor also allows the user to associate a *breakpoint* with a flaw in the plan under construction. When this happens, the editor sends a special message to the planner containing the selected flaw, which modifies the standard behavior of the planner in the following way. Whenever the planning process of LPG selects such a flaw to repair, the process is interrupted; the system presents all possible options for repairing the flaw to the user; and the user chooses one of these options repairing the flaw “by hand”.

For each flaw  $\sigma$  repaired by hand and search neighborhood  $N$ , the search process editor memorizes the successor action graph (a graph modification) selected by the user. In the successive search steps, if the planning process of LPG

attempt to repair flaw  $\sigma$  again, evaluating a search neighborhood *similar* to  $N$ , then the successor action graph selected by the planner is the graph obtained by performing the graph modification previously selected by the user, which could be different from the action graph that the planner would select from the neighborhood according to its heuristic evaluation. Let  $N_{curr}$  be the neighborhood for solving the flaw  $\sigma$  under consideration, the similarity between  $N_{curr}$  and  $N$  is measured by  $\frac{|N_{curr} \cap N|}{\max\{|N_{curr}|, |N|\}}$ . A similarity threshold  $t$  can be customized as an input setting of the graphical user interface. Thus, the search process repairs flaw  $\sigma$  using the graph modification previously chosen by the user for  $\sigma$  with neighborhood  $N$ , if the similarity measure between  $N_{curr}$  and  $N$  is greater than or equal to  $t$ ; it repairs flaw  $\sigma$  using the graph modification selected according to the default criteria of LPG, otherwise.

Finally, the search process editor allows the user to impose *intermediate goals*, i.e., facts that must be true at some point in the plan.

## Conclusions

We have presented InLPG, an implemented interactive tool for the visualization, inspection, generation and revisions of plans, which supports a form of “human-in-the-loop” control of planning that is typical in the mixed-initiative approach to plan generation. InLPG includes a graphical in-

terface through which the user can interact with a state-of-the-art domain-independent planner, obtaining an effective visualization of a variety of information about the plan under construction or inspection, as well as about the undergoing planning process. Moreover, the tool provides some capabilities allowing the user to intervene during the planning process to modify the problem goals, the plan under construction or the planner heuristic decisions at search time.

In a preliminary experiment using hard problems from the IPC benchmark domains *Philosophers* and *Storage*, we observed that specifying a few human-driven search steps or intermediate goals through the user interface of InLPG significantly helps the underlying planner to reach a solution plan, which otherwise would be much harder to find for the planner alone.

Current and future work concerns the extension of InLPG with further innovative techniques of information visualization for improving the readability of large plans. We also intend to augment the options offered by the tool to the human intervention during the plan construction, and to conduct some experiments to test the usability of the tool for novice and expert users of planning technology. Finally, we believe that the proposed tool can be useful also in an educational context to support teaching and learning AI planning, as observed with the students of an AI course (master degree level) recently taught by the authors of this paper.

**Acknowledgments.** We would like to thank Fabrizio Bonfadini and Maurizio Vitale for their help with the implementation of InLPG's graphical interface.

## References

- J. Allen and G. Ferguson. 2002. Human-machine collaborative planning. In *Proc. of the 3rd Int. NASA Workshop on Planning and Scheduling for Space* (2002).
- A. Blum and M. Furst. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.
- J. Bresina, A. Jonsson, P. Morris, and K. Rajan. 2005. Activity planning for the mars exploration rovers. In *Proc. of the 15th Int. Conf. on Automated Planning and Scheduling*.
- L. Castillo, J. Fdez-Olivares, O. García-Pérez and F. Palao 2005. SIADEX. An integrated planning framework for crisis action planning. In *Proc. of ICAPS-05 System Demonstrations*.
- M. T. Cox and M. Veloso, 1997. Controlling for unexpected goals when planning in a mixed-initiative setting. In *Proc. of the 8th Portuguese Conf. on Artificial Intelligence*.
- M. Cox and C. Zhang, 2005. Planning as mixed-initiative goal manipulation. In *Proc. of the 15th Int. Conf. on Automated Planning and Scheduling*.
- K. Currie and A. Tate. 1991. O-plan: the open planning architecture. *Artificial Intelligence* 52:49–86.
- P. Daley, J. Frank, M. Iatauro, C. McGann, W. Taylor. 2005. PlanWorks: A debugging environment for constraint based planning systems. In *Proc. of the 1st Int. Competition on Knowledge Engineering for Planning and Scheduling*.
- G. Ferguson and J. Allen. 1994. Arguing about plans: Plan representation and reasoning for mixed-initiative planning. *Proc. of the 2nd Int. Conf. on AI Planning Systems*.
- G. Ferguson, J. Allen, and B. Miller. 1996. Trains-95: Towards a mixed-initiative planning assistant. *Proc. of the 3rd Conf. on Artificial Intelligence Planning Systems*.
- M. Fox and D. Long. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20:61–124.
- A. Gerevini, A. Saetti, and I. Serina. 2003. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research* 20:239–290.
- A. Gerevini, A. Saetti, and I. Serina. 2004. An empirical analysis of some heuristic features for local search in LPG, *Proc. of the 14th Int. Conf. on Automated Planning and Scheduling*.
- A. Gerevini, A. Saetti, and I. Serina. An approach to temporal planning and scheduling in domains with predictable exogenous events. *Journal of Artificial Intelligence Research* 25:187–231.
- A. Gerevini, A. Saetti, and I. Serina. 2009. An Approach to Efficient Planning with Numerical Fluents and Multi-Criteria Plan Quality. *Artificial Intelligence* 172(8-9):899–944.
- A. Gerevini and I. Serina. 2000. Fast plan adaptation through planning graphs: Local and systematic search techniques. in *Proc. of the 5th Int. Conf. on Artificial Intelligence Planning and Scheduling*.
- J. Hoffmann and S. Edelkamp, 2005. The deterministic part of IPC-4: An overview. *Journal of Artificial Intelligence Research* 24:519–579.
- N. Lino and A. Tate. 2004. A visualisation approach for collaborative planning systems based on ontologies. In *Proc. of the 8th Int. Conference on Information Visualisation*.
- N. Lino, A. Tate, and Y.-H. Chen-Burger. 2005. Semantic support for visualisation in collaborative AI planning. In *Proc. of the Workshop on The Role of Ontologies in Planning and Scheduling*.
- K. L. Myers, P. A. Jarvis, W. M. Tyson, and M. J. Wolverton. 2003. A mixed-initiative framework for robust plan sketching. In *Proc. of the 13th Int. Conf. on Automated Planning and Scheduling*.
- H. A. Simon. 1957. Models of Man. John Wiley & Sons Inc., New York, USA.
- G. Tecuci. *Proc. of the IJCAI Workshop on Mixed-Initiative Intelligent Systems*. AAAI Press, Menlo Park, California, USA.
- M. Veloso, M. Mulvehill, A., and T. Cox, M. 1997. Rationale-supported mixed-initiative case-based planning. in *Proc. of the 9th Conf. on Innovative Applications of Artificial Intelligence*.
- C. Zhang. 2002. Cognitive models for mixed-initiative planning. *PhD thesis*, Wright State University, Computer Science and Engineering Department, Dayton, Ohio, USA.

## An extended HTN knowledge representation based on a graphical notation

**Francisco Palao**

IActive Intelligent Solutions  
Granada, SPAIN

**Juan Fdez-Olivares**

Dept. of Computer Science and A.I  
University of Granada

**Luis Castillo and Oscar García**

IActive Intelligent Solutions  
Granada, SPAIN

### Abstract

This work presents both an extended HTN Knowledge Representation based on a graphical notation inspired in commercial standards and a suite of tools, named *IActive Knowledge Studio*, based on this representation aimed at fully supporting a knowledge engineering process that, starting with the acquisition and representation of planning knowledge, ends with the integration and deployment of planning applications. The suite is also intended to be a knowledge engineering workbench for HTN planning applications deployment. This workbench includes several interesting features like a fully integrated representation of problem and domain knowledge and a new graphical and intuitive notation for easily representing HTN domains. It also provides tools for data integration with external data sources as well as an enhanced visual environment for HTN domains and plan validation.

### Motivation

AI Planning and Scheduling (AIP&S) has revealed as an enabling technology to develop "assistant applications" which support human decision making in domains where the accomplishment of tasks to carry out a given activity or achieve a goal is mandatory. Most of these applications are aimed at supporting Human-Centric processes (Dayal, Hsu, and Ladin 2001) for knowledge workers (experts or decision makers). These processes are collections of tasks which support decisions and help to the accomplishment of workflow tasks for such knowledge workers in several and diverse application domains. Indeed, the features of AIP&S are really aligned with key requirements in these application field: human-centric processes mainly embody expert knowledge, processes need to be dynamically generated through a process that must be aware of the context in which they will be executed, and the execution environments are highly dynamic, thus requiring adaptive behaviour and rapid response to the new, changing situations.

The development of such knowledge-intensive applications, where intelligent planning becomes a key component, require a great modeling and engineering effort in the main application development stages: acquisition and representation of planning knowledge, validation of such knowledge, integration with external sources of information or already existing legacy software systems, and deployment of the fi-

nal planning application. Most planning applications developed at the time being (Fdez-Olivares et al. 2006), (Fdez-Olivares et al. 2011), (Boddy and Bonasso 2010), (Cesta et al. 2010) are based on a standard life-cycle based on the above stages, but each of the steps in this cycle are performed following different, ad-hoc and difficult to couple techniques or tools, not easily scalable nor reproducible to other applications (thought the same technology is being used). For example, given a set of protocols and procedures to be operationalized, it is common to manually encode such protocols in a textual planning language (PDDL (Gerevini and Long 2006), HTN-PDDL (Fdez-Olivares et al. 2006) or ANML (Boddy and Bonasso 2010) to cite some), then to generate plans in order to validate this knowledge, then to develop ad-hoc algorithms to integrate external sources of information (for example mapping data from ontologies or external data bases to PDDL data models (Castillo et al. 2010a)), then to develop extra code in order to integrate the output of the planner with existing systems, etc. This is neither an agile nor easily reproducible development process that clearly impact negatively the goal of AI Planning to be a widespread and widely used technology. Moreover, a widely recognized bottle-neck in the development of such applications is that the languages used to represent both expert knowledge and domain dynamics are textual and oriented to expert AI planning researchers, and normally, little attention is devoted to the domain objects model as well as the plan representation and integration.

This handcrafted way of working in our area is in contrast with the extremely high-technological tools developed in other, no so distant areas like Business Process Management (BPM (van der Aalst, ter Hofstede, and Weske 2003)), where a plethora of tools may be found which give support to the whole life-cycle of development in such areas: modeling, deployment, execution and monitoring of processes. BPM is, perhaps, the dominant area in IT solutions for Human-Centric processes, but BPM technology is mainly focused on the management of static and perfectly predictable tasks/processes and, at present, there is a clear trend (González-Ferrer et al. 2010) to incorporate features like dynamic composition, context awareness or adaptiveness of processes into software solutions: clearly these features may come from the incorporation of AIP&S technology into these systems. However, in order to compete on

equal terms with this extremely developed area, and to convince IT engineers and managers about the real capabilities of AI Planning technology, the current engineering process of planning applications must be radically improved.

Compared with the modeling and engineering processes of the not so different BPM area, AI P&S lacks of integrated tools (also called suites) that cover an important gap, still not fully bridged, between the conceptualization and design of a planning domain and the final deployment of a fully functional planning application. In this sense, the contribution of this work is two fold:

- On the one hand, we present an extended knowledge representation based on a graphical notation, built on both the concepts of HTN planning languages (concretely an HTN extension of PDDL, called HTN-PDDL (Castillo et al. 2006) which has been used in several applications) and inspired in the graphical notation of industrial standards devoted to the modeling of business processes (called BPMN (White 2004)).
- On the other hand, a suite of strongly coupled planning tools developed by IActive Intelligent Solutions<sup>1</sup>, integrated into a product called *IActive Knowledge Studio*<sup>2</sup>, conceived as an integrated development environment for planning applications. It includes several *visual working environments* in order to support the main steps of Knowledge engineering: *acquisition and representation* of planning knowledge based on the graphical notation, *validation by inspection* based on planning process debugging and plan visualization and analysis, *knowledge integration* with external sources of information and *planning application deployment*.

Next sections are devoted to describe, firstly, the main issues concerned with the extended graphical representation and, secondly, the main features of IActive Knowledge Studio.

## The graphical knowledge representation

The graphical knowledge representation introduced in this work can be considered as an evolution from a former, textual HTN language called HTN-PDDL (Castillo et al. 2006),(Fdez-Olivares et al. 2011) towards a more usable one, closer to the modeling practices of general purpose IT engineers and strongly focused on developing commercial planning applications. HTN-PDDL is a hierarchical extension of PDDL which incorporates all the standards concepts of the PDDL 2.2 version (Edelkamp and Hoffmann 2004). Concretely HTN-PDDL supports the modeling of planning domains in terms of a compositional hierarchy of tasks representing compound and primitive tasks at different levels of abstraction, where primitive tasks maintain the same expressiveness that PDDL 2.2 level 3 durative actions (allowing to represent temporal information like duration and start/end temporal constraints, see (Castillo et al. 2006) for details). In addition, HTN methods used to decompose compound

tasks into sub-tasks include a precondition that must be satisfied by the current world state in order for the decomposition method to be applicable by the planner. The problem representation in HTN-PDDL is thus almost the same that in PDDL 2.2 problems, but with the only difference that the goal is described as a set of high-level tasks to be decomposed, instead of than a set of states to be achieved.

It is worth to note that this textual representation, though can be seen as a general-purpose hierarchical planning representation, based on the HTN paradigm, is specific for a temporally extended HTN planner, formerly known as SIADEx (Fdez-Olivares et al. 2006),(Castillo et al. 2006), which has evolved as a commercial product, developed by our start-up *IActive Intelligent Solutions*, now called *Decisor*<sup>3</sup>. Both, the textual language and the former planner have already been applied in several applications in diverse domains like crisis intervention (Fdez-Olivares et al. 2006),(Castillo et al. 2010a), e-learning (Castillo et al. 2010b), e-tourism (Castillo et al. 2008) or e-health (Fdez-Olivares et al. 2011), many of them being at present commercially exploited by IActive<sup>4</sup>.

In the following, we will detail how this textual representation has evolved into a graphical knowledge representation based on three pillars: **1)** a domain objects representation based on UML that is called *Context Model* that clearly overcomes the classic representation of PDDL domain objects, **2)** a planning description language, still based on predicates but incorporating object-oriented concepts, named EDKL (Expert Knowledge Description Language) used for writing logical expressions for preconditions, effects, rules and temporal constraints in both compound tasks and actions, and **3)** a graphical notation named EKMN (Expert Knowledge Model Notation) used to represent the main concepts of an HTN domain (compound tasks, methods, primitive tasks as well as hierarchical, compositional and order relations) intended to be understandable by both, IT engineers and domain experts.

## Context Model: UML-based domain objects model

Domain objects are represented in the Context Model following the standards recommendations of UML (Unified Modeling Language (Booch, Rumbaugh, and Jacobson 1999)), a standardized general-purpose modeling language in the field of object-oriented software engineering. UML includes a set of graphic notation techniques to create visual models of object-oriented software-intensive systems. Although UML supports the modeling of many aspects related with planning applications (like activities diagrams or state machine diagrams) we have opted to use UML only with those issues related with data and domain objects modeling, trying to bring the strong modeling capabilities of this standard to domain objects in AI Planning.

Indeed, all the HTN-PDDL concepts (mostly inherited from the classical PDDL types and objects representation) have their associated representation in UML. The Context

<sup>1</sup><http://www.iactive.es>

<sup>2</sup>Download at <http://www.iactive.es/productos/iactive-knowledge-studio/>

<sup>3</sup><http://www.iactivecompany.com/products/iactive-intelligent-decisor/>

<sup>4</sup><http://www.iactive.es/casos-de-exito/casos-de-exito/>



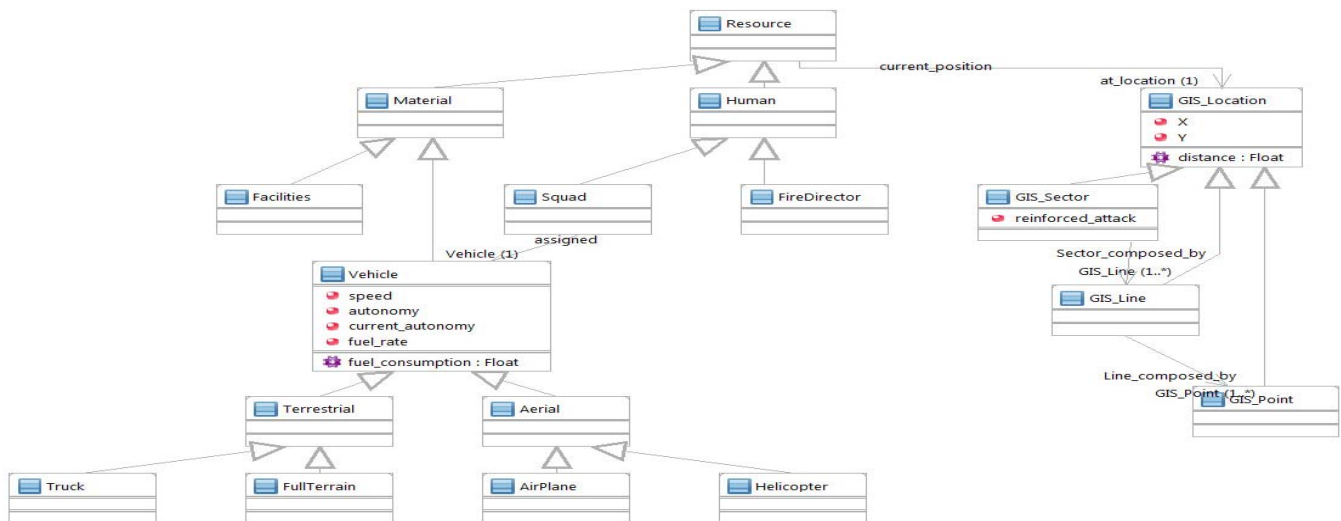


Figure 1: A (simple) UML model of a fire-fighting ontology: a Resource is located at a GISLocation (represented by a UML association). Every location is represented by two coordinates and an associated operation to compute the distance between two points (special Google libraries are provided by the language to implement this operation). There are material (Vehicles and Facilities) or Human (Squads and Fire Director) resources. A Vehicle has an speed, autonomy, current autonomy and a fuel rate. The fuel consumption of every vehicle (Terrestrial or Aerial) is a dynamic value computed by a procedure. Attributes, operations and relations are inherited throughout the is-a hierarchy.

Model is based on five key UML concepts: *Class*, *Attribute*, *Operation*, *Association* and *Generalization*. Object types are represented as UML Classes (see Figure 1), and the hierarchy of types is represented as an is-A hierarchy, using the UML standard Generalization relationship between classes; object properties are represented as UML Class attributes and relations as UML Association. Domain objects are represented as instances of the UML classes defined. In addition, UML operations are used as special attributes of objects that need to be computed by a procedure, thus allowing to manage and represent resources. Finally, it is also possible to represent temporal information as attributes of a class the value of which is of a special type called *DateTime* (supporting the representation of time and dates associated to objects).

This modeling approach based on UML eliminates any barrier between the modeler and the AIP&S technology, and it is intended to offer a standard way to model planning domain objects close to IT engineers. Moreover, as seen in the next section, the Context Model defined as a UML Class Diagram is directly incorporated and managed by the planning language. The incorporation of this object-oriented approach into the planning language provides new features that increase both the expressiveness of the planning language and the user-friendliness with respect to previous languages like PDDL or HTN-PDDL. Among others enhancements, these features allow to introduce a solid interpretation of inheritance of properties and operations in the language as well as a better modeling of objects relations, now much easier to manage. Perhaps the most important one is the integration with external data models. That is, most external sources of information are based on relational data bases or

ontologies. Both models can be directly mapped into UML models, thus increasing the integration capabilities of planning domains and problems with external sources information, a key issue to be addressed in the development of any planning application.

### EKDL: Expert Knowledge Description Language

As opposed to other well known approaches, like for example *itSIMPLE3.0* (Vaquero et al. 2009), the aim of EKDL<sup>5</sup> is not to translate the UML Context Model into PDDL, but maintaining this object-oriented representation as modeled in the Context Model and introducing it into the planning language that will be described in this and following sections. Predicates are still the basic construction of EKDL, but the standard syntax used in most planning languages has been modified in order to cope with objects and classes.

The first consequence is that predicates now respond to the syntactical pattern ( $C.p\ o\ v$ ), where  $C$  stands for the name of a class,  $p$  stands for the name of a property or a relation,  $o$  stands for an instance or a variable of type  $C$ , and  $v$  stands for an instance or variable of the range of  $p$  (that is, the Class which the allowed values of the property  $p$  belong to). For example, a typical PDDL predicate ( $at\ ?x - Person\ ?y - Place$ ) shall require firstly to define a Class *Person*, with a property *at* of type *Place* (*Place* must also be defined as a class). This definition will result in an EKDL predicate ( $Person.at\ ?x\ ?y$ ) that can be used as desired in any logical expression. This notation (indeed a prefix form of the standard  $< object\ attribute\ value >$ ) forces to represent all the

<sup>5</sup>A manual (in spanish) describing EKDL and EKMN can be found in <http://help.iactive.es/>

expressions as logical combinations of binary predicates, but has the key advantage that is aligned with standards knowledge interchange formats, like for example RDF triplets, a common way to serialize knowledge present in ontologies or relational data bases, thus opening the way to easily integrate the planning knowledge modeled under EKDL with other standard knowledge representation formalisms.

In summary, EKDL is an adaptation and extension of PDDL and HTN-PDDL that tries to maintain the expressiveness of logical expressions based on predicates, but also maintaining the object-oriented approach in logical expressions.

Parameters:	?g:Squad, ?v:Vehicle, ?p1:GIS_Location, ?p2:GIS_Location
Actors:	
Conditions:	
1	(and (Squad.Vehicle ?g ?v) (Vehicle.at_location ?v ?p1)
2	(Squad.at_location ?g ?p1)
3	(= ?dur / (GIS_Location.distance ?p1 ?p2) (Vehicle.speed ?v)))
Effects:	
1	(and (not (Vehicle.at_location ?v ?p1)) (not (Squad.at_location ?g ?p1))
2	(Vehicle.at_location ?v ?p2) (Squad.at_location ?g ?p2)
3	(- (= (Vehicle.current_autonomy ?v) ?dur) )

Figure 2: A primitive task represented in EKDL. It is like a PDDL durative action representing the movement of a human group transported by a vehicle. Temporal (duration of the transport) and resource constraints (the autonomy of the vehicle decreases) are also represented.

Primitive tasks are represented textually in EKDL (see Figure 2), but supported by visual forms. They are represented as a name, typed parameters (now referring to Context Model classes), and logical expressions to describe preconditions and effects. Preconditions and effects are represented as logical expressions of predicates taking into account the object-centered syntax above described. Numerical function are also allowed and, therefore, it is also possible to represent discrete numerical resources. Indeed numerical resources are represented as numerical properties of objects, for example the remaining autonomy of a Vehicle can be represented as the attribute *current-autonomy* of a class *Vehicle*. Then EKDL provides arithmetic and increment/decrement operators that can be applied to numerical attributes. For example, the expression  $(= (Vehicle.current\_autonomy?v)?dur)$  stands for a decreasing  $?dur$  time units the autonomy of a Vehicle  $?v$ .

In addition, primitive tasks representation inherits the concepts of PDDL 2.2 level 3 durative actions (allowing to represent temporal information like duration and start/end temporal constraints). EKDL also allows the representation of PDDL 2.2 axioms. As shown in next sections, the introduction of this knowledge is supported by graphical interfaces, with powerful capabilities to make the introduction of this knowledge an easier task.

It is worth to note that neither UML activity diagrams nor state machine diagrams are intended to capture and repre-

sent all the categories of knowledge found in temporal HTN paradigms. Although they may be useful to represent some aspects of domain dynamics related to changes of state, and thus able to model basic primitive actions, they lack of necessary mechanisms to adequately represent temporal constraints as well as compound tasks and alternative decomposition methods. UML can represent relationships between activities at different levels of abstraction, but it is not designed to represent alternative decompositions to be dynamically managed at reasoning time by a planner (one of the key aspects of HTN). Moreover, though some time constraints on activities can be represented in UML, they are intended to be used on sets of predefined, already fixed activities. The temporal constraints management in HTN is much more expressive, since it supports to represent which temporal constraints must be satisfied on a dynamically generated, and initially unknown set of tasks. Therefore, the modeling of HTN planning domains requires additional characteristics to the ones presented in UML, since UML is not designed to be a knowledge representation language. Because of this, we have defined a new graphical approach for this purpose, detailed in the next section.

## EKMN: Expert Knowledge Model Notation




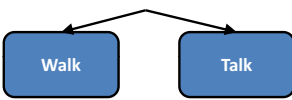
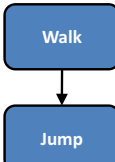
EKMN Graphic Elements	
Graphic notation	Description
	<b>Compound Task or Goal</b> ...
	<b>Method</b> ...
	<b>Task</b> ...
	<b>Parallel Task Network</b> ...
	<b>Sequential Task Network</b> ...

Figure 3: Main elements of the graphic notation.

Compound tasks, decomposition methods and primitive actions represented in an HTN planning domain mainly encode the procedures, decisions and actions that experts must follow, according to a given protocol, when they deal with a given decision problem.

Therefore, the main advantage of HTN planning approaches is their capability of capturing planning expert knowledge in form of either protocols or operating procedures. Hence, an HTN planning model should be seen as

a knowledge representation mechanism to represent human expertise and operating procedures as well as using them as a guide to the planning process.

The main goal of EKMN is to face these aspects and provide a notation that is understandable by IT modelers (responsible of encoding the knowledge finally managed by the planner) as well as domain experts (containers of the knowledge). It is inspired in BPMN (White 2004), the current standard notation for process modeling, and its aim is to display an intuitive visual metaphor of the main categories of knowledge found in an HTN domain. On the one hand, *compound tasks (or goals)*, *methods*, *task networks* and *primitive tasks*. On the other hand, the relationships between these categories: *hierarchical relationships* between tasks and methods, *is-part-of* relationships between methods and subtasks, and *order relations* between tasks inside a method). Figure 3 shows the visual representation used for compound tasks (goals), methods, primitive tasks and sequential and parallel tasks. The main idea behind this graphic notation is to support a modeling process that starts with the development of a visual skeleton of the task hierarchy, carrying out (if possible) a collaborative process between knowledge engineer and expert, and then to detail this skeleton by filling out more detailed knowledge, using EKDL, in successive refinement steps. Figure 4 shows an example of the skeleton of an HTN domain which incorporates the above described elements.

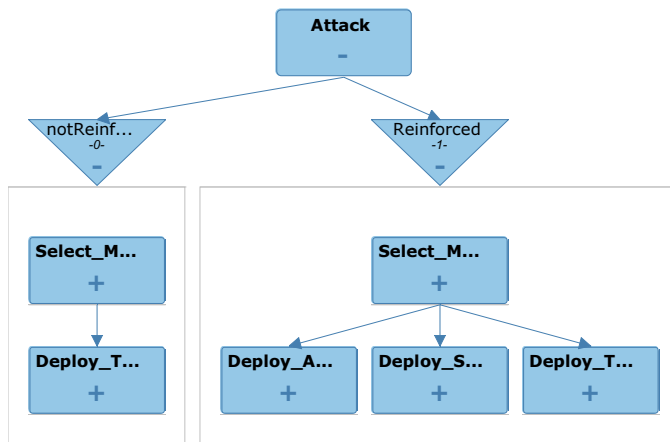


Figure 4: An example domain using EKMN implementing a standard operating procedure to attack a sector in a forest fire. The doctrine establishes that, in case of not being a reinforced attack, first select appropriated human means, then deploy and mobilize them. In case of reinforced attack, first select means, then deploy every human, aerial and terrestrial mean selected.

More concretely, the knowledge representation language as well as the planner are also capable of representing and managing different workflow patterns (van Der Aalst et al. 2003) which can be found in most business process models. A knowledge engineer might then represent control structures that define both, the execution order (sequence, parallel, split or join), and the control logic of processes with

conditional (represented by alternative methods) and iterative ones (represented by recursive decomposition schemas).

In addition, this knowledge representation supports to explicitly represent and manage time and concurrency at every level of the task hierarchy in both compound and primitive tasks, by allowing to express temporal constraints on the start or the end of an activity. Any sub-activity (either task or action) has two special variables associated to it, *?start* and *?end*, that represent its start and end time points, and some constraints (basically  $\leq$ ,  $=$ ,  $\geq$ ) may be posted on them (it is also possible to post constraints on the duration with the special variable *?duration*). In order to do that, any activity may be preceded by a logical expression that defines a temporal constraint. For example, it is possible to encode constraints of the form  $((\text{and } (>= ?start \text{ date1}) (<= ?start \text{ date2})) (t))$  what provides flexibility for the start time of *t*'s execution, indicating that *t* should start neither earlier than *date1* nor later than *date2*. These constraints are represented using EKDL and can be posted using a user-friendly interface, embedded into the integrated development environment that is described in the next section.

## IActive Knowledge Studio

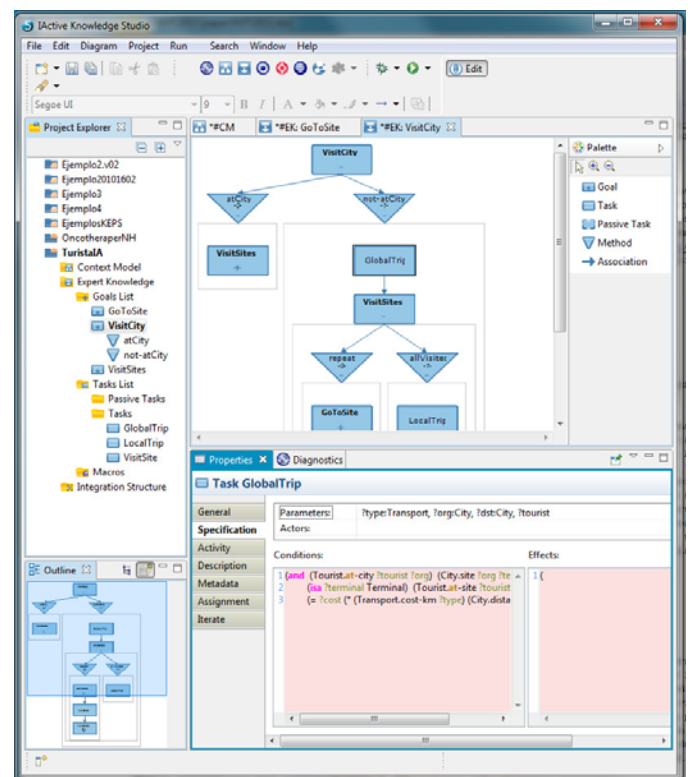


Figure 5: The knowledge edition environment of IActive Knowledge Studio. From left to right: the *project tree* showing the main parts of a planning project, the task hierarchy, an outline of this hierarchy, and a properties view showing the EKDL representation of a primitive task.

The above described knowledge representation and

graphical notation is the cornerstone of a suite of planning tools developed by IActive Intelligent Solutions, integrated into a product called *IActive Knowledge Studio*. This suite is conceived as an integrated development environment for planning applications. It is intended to support the main steps in the process of Knowledge Engineering for Planning and Scheduling: *Knowledge Acquisition and Representation* based on the above described extended graphical representation, *Knowledge Validation* based on a validation-by-inspection process supported by an enhanced debugging process and plan analysis tools, *Knowledge Integration* with external sources of information, based on the Context Model above described and *Planning Application Deployment*. IActive Knowledge Studio has been developed using Eclipse Java technology, and it includes several *visual working environments* in order to support each one of these steps, which will be detailed in the four following sections.

### Edition environment

The edition environment is intended to facilitate the acquisition and representation of planning domain knowledge. It allows to model planning objects based on the UML Context Model, to describe the task hierarchy based on EKMN and to fill out the properties associated to every category of knowledge through EKDL. Figure 5 shows a snapshot of the main perspective. It provides a *Project Tree* where the main categories of a planning project are shown: the *context model* in the terms above described, the *expert knowledge* showing an expandable list of tasks and the *integration structure* devoted to represent planning problems (see section below). A region to graphically model both the Context Model objects and the EKMN domain is also provided. Finally, the environment also includes a *properties window* in order to represent in EKDL the knowledge required for action preconditions and effects, temporal constraints, axioms, etc.

### Integration environment

The integration environment has a triple role: (1) it is intended to define the necessary data definitions to allow the integration of both the input (initial state) and the output (the plans) of the planner with external systems, (2) to describe the initial state, and (3) to define the goal. Through this environment (see Figure 6), a domain modeler not only is able to represent the initial values of object properties or relations, but to define the way in which external sources of information can be accessed from the planner. Aimed at preserving its integration capabilities at the maximum, the data definition schemas managed by the planner (input data of initial state, input goal and plans) are stored as XML files (XSD templates), and these definitions are intended to both, be used to easily define the mapping from external data sources into the internal structure defined in the Context Model, and to integrate the plans obtained with external systems that need as input the output of the planner (for example, a BPM running engine).

The integration environment turns around the concept of *Integration Structure*, what can be seen as a metaphor of the "old" concept of planning problem, but redefining it into a

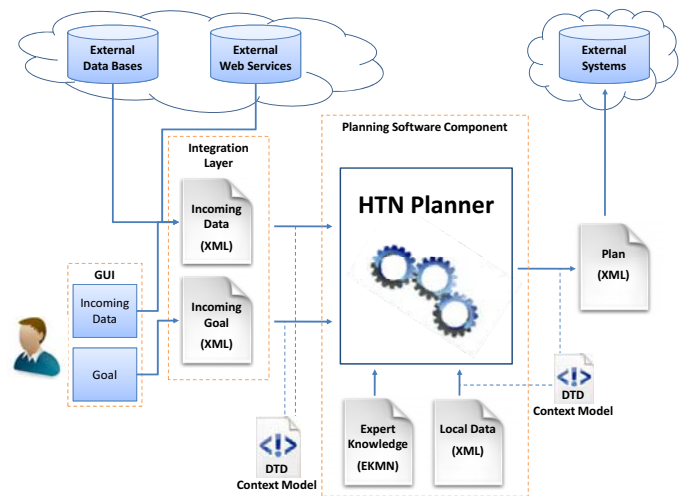


Figure 6: A diagram of the integration environment.

more ambitious way, mainly centered on exploiting and enhancing the integration capabilities of the planner. Through this metaphor a Knowledge Engineer is able to define a planning problem by using the following concepts: *Local Data*, *Incoming Data* and *Incoming Goal*. **Local Data** are intended to be used in the first validation tests of the planning domain. The working environment allows to describe de initial conditions of every object as set of data tables (locally managed by the tool) that are automatically generated from the Context Model. These tables are automatically generated in such a way that the attributes of every table  $T$  corresponds to the properties and relations defined for a corresponding class  $C_T$ . In addition, each row in the table corresponds with an object instance of type  $C_T$ . Therefore, a domain modeler can describe the initial state of the world in a friendly and commonly accepted interface based on relational data base tables. The section **Incoming data** is aimed at managing the Data Template Definition (DTD), stored as XML files, that an external source should fit in order for the planner to access these external data. This is the cornerstone of the integration features of this approach: on the one hand, the XML schemas so defined can be used to easily develop data mappings to acquire external information, on the other hand these files can be used to provide the XSD specifications to third-party developers responsible of integrating the planning application with other systems. Finally, through the **Incoming Goal** Section the modeler defines the goal, that is, the task at the highest level to be decomposed as well as its parameters and start/end temporal constraints. These definitions are also stored as XML files that are given as inputs to the planner in order to carry out a planning process. Finally, the resulting plan is also stored as an XML file which can be used to integrate this output with other external systems.

### Execution and debugging environment

Once the domain knowledge as well as the information required for the planner has been defined, the next step con-



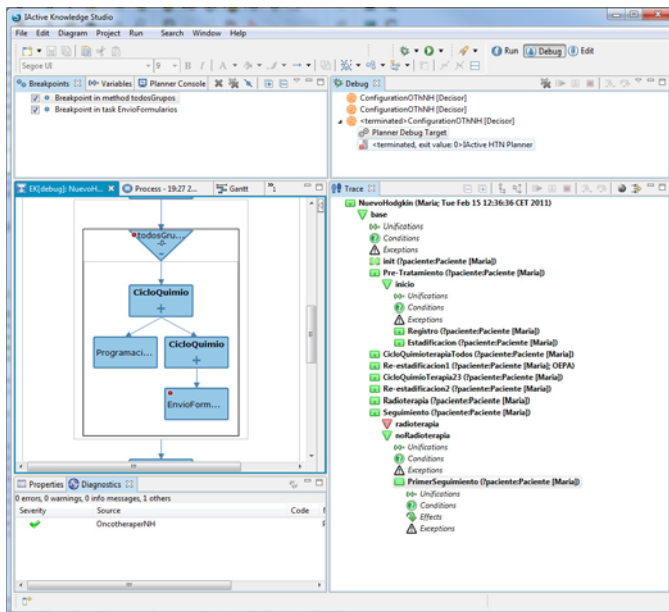


Figure 7: A snapshot of the debugging environment.

sists on using the execution and debugging environment in order to validate the knowledge. The execution and debugging environment provides the necessary functionalities to perform a validation-by-inspection process. On the one hand, it provides a trace facility that allows to execute step by step the planning process guided by the knowledge described in the EKMN notation (see Figure 7). The trace window allows to visualize and analyze through an expandable decision tree the decisions performed by the planner during the planning process. In addition, the tool also supports to define *breakpoints* associated either to compound tasks (goals), methods or primitive tasks, thus allowing to interrupt the knowledge-based reasoning process at any point defined by the modeler. The trace tree informs about the selected HTN methods as well as the discarded options during a given problem solving episode. In addition, this environment also allows to show the intermediate states produced by the planner. Moreover, it provides powerful tools for plan analysis and validation. Firstly, the plan obtained can be visualized either as a sheet or as a gantt diagram. The gantt diagram visualization also allows to intuitively analyze order dependencies between actions. In addition, there is a section to show several statistics about the resulting plan (resource usage, actions duration, etc.).

## Deployment environment

The deployment environment allows to obtain a software component with the functionality defined in the previous steps and which will be able to be executed as a standalone application. The deployment process starts when the planning knowledge has been acquired and represented, when the data models necessary to integrate external information have been defined and when the knowledge has been validated. The deployment environment allows to deploy a plan-

ning project either as a local application (a *jar* file that can be accessed through an API) or as a web service (allowing to access to the planning application through remote procedure calls). In both cases, an API (application programming interface) is automatically provided. This API allows, among other operations, to perform calls to the planner and to obtain plans. Concretely, the planning component developed will return a plan from a given goal, specified as an XML file and a given set of data specified as an XML file accomplishing the data schema defined in the integration environment. The plan representation is based on an internal model that can also be known through the API. Therefore, this opens the way to integrate (by third party developers) the plans obtained with external systems that require as input the solutions offered by the planner.

## Related work

Regarding other Knowledge Engineering Tools, GIPOII (McCluskey, Liu, and Simpson 2003) and itSIMPLE3.0 (Vaquero et al. 2009) are pursuing similar purposes as the work here presented. On the one hand, GIPOII is mainly aimed at supporting an object-centered Knowledge Acquisition process, focused on how domain objects change their properties or relations by specifying object transitions. Then these transitions are grouped to finally form action specifications. As opposite, the approach here presented supports a task-centered knowledge engineering process, based on the HTN paradigm. With respect to itSIMPLE3.0, one of the main common points is the domain objects model, also based on UML diagrams. itSIMPLE3.0 is devoted to capture user data requirements by using class diagrams and state based diagrams in order to represent the dynamics of actions in non-hierarchical domains, and then to translate this model into PDDL. Our approach, on the contrary, maintains the object-oriented approach to its last consequences and, so, the planning language (EKDL) has been designed in order to adopt the object-oriented paradigm. Perhaps, one of the stronger points of itSIMPLE3.0 is its capability to perform domain analysis by translating the UML and PDDL specification into Petri-Nets, what supposes a great help when focusing on validating the dynamic aspects of the domain. However these techniques are not fully applicable to hierarchical domains like the one addressed in our approach. On the other hand, itSIMPLE3.0 does not deal with temporal constraints, nor faces the representation of hierarchical planning knowledge.

Another work somehow related is (Boddy and Bonasso 2010) where authors describe a knowledge engineering process that includes the representation of operating procedures in NASA's PRL (Procedure Representation Language) (Kortenkamp et al. 2008), and then the (manual) translation into ANML (Action Notation Modeling Language) in order to be manageable for a planner. Indeed the representation of operating procedures is supported by a different tool (Kortenkamp et al. 2008), than the one used to represent planning domains with ANML (Boddy and Bonasso 2010). Our knowledge representation, based on the graphical HTN notation here presented is indeed intended to represent operating procedures (as HTN domains) and, moreover, to make

these protocols directly interpretable by a planner, thus offering a more integrated approach than the one described by Boddy et al.

## Conclusions

In this work we have presented an extended Graphical Knowledge Representation for HTN domains with three main features: (1) it allows to model domain objects following an object-centered approach based on UML visual diagrams, overcoming several weaknesses of PDDL, mainly related with expressiveness and user-friendliness issues, specially the most relevant one is that this planning domain objects representation is closer to the modeling practices of IT engineers; (2) this Context Model is also directly embedded into the planning language and, in order to manage this object-centered model, EKDL (an object-oriented redefinition of basic PDDL constructs) has been described; (3) the most important aspect of this knowledge representation is EKMN, a graphical notation based on standard BPM modeling notations, which is aimed at visually and intuitively representing HTN domains as hierarchical and expandable diagrams based on compound tasks (goals)/methods/primitive tasks and the relationships between them. This graphical knowledge representation is intended to be understandable by both, IT engineers and domain experts. On the other hand, the knowledge representation is the basis on which IActive Knowledge Studio has been built. It is a development suite intended to support a knowledge engineering process that bridges the gap between the conceptualization of a planning domain and the final deployment of a planning application. The tool may also be seen as a workbench that can be used for academic and research purposes, including interesting features like a fully integrated representation of problem and domain knowledge and a new graphical and intuitive notation for easily representing HTN domains. IActive Knowledge Studio, also provides tools for data integration with external data sources, plan statistics and visualization methods for plan validation. But its most distinguishing features is that it has been designed to be used by IT engineers when developing commercial planning applications. Indeed, we have achieved to significantly increase the number of users of AIP&S technology through this development suite, since it is a commercial product that is being used in several industrial projects developed in collaboration with the partners of IActive Intelligent Solutions.

## References

- Boddy, M., and Bonasso, R. 2010. Planning for human execution of procedures using ANML. In *Scheduling and Planning Applications Workshop (SPARK), ICAPS*.
- Booch, G.; Rumbaugh, J.; and Jacobson, I. 1999. *The unified modeling language user guide*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA.
- Castillo, L.; Fdez-Olivares, J.; García-Pérez, O.; and Palao, F. 2006. Efficiently handling temporal knowledge in an HTN planner. In *Proceeding of ICAPS06*, 63–72.
- Castillo, L.; Armengol, E.; Onaindía, E.; Sebastián, L.; González-Boticario, J.; Rodríguez, A.; Fernández, S.; Arias, J.; and Borrajo, D. 2008. samap: An user-oriented adaptive system for planning tourist visits. *Expert Systems with Applications* 34(2):1318–1332.
- Castillo, L.; Fdez-Olivares, J.; González, Milla, G.; Prior, D.; Morales, L.; Figueroa, J.; and Pérez-Villar, V. 2010a. A knowledge engineering methodology for rapid prototyping of planning applications. In *Proceedings of FLAIRS 2010*.
- Castillo, L.; Morales, L.; González-Ferrer, A.; Fdez-Olivares, J.; Borrajo, D.; and Onainda, E. 2010b. Automatic generation of temporal planning domains for learning problems. *Journal of Scheduling* 13:347–362.
- Cesta, A.; Finzi, A.; Fratini, S.; Orlandini, A.; and Tronci, E. 2010. Validation and verification issues in a timeline-based planning system. *The Knowledge Engineering Review* 25(03):299318.
- Dayal, U.; Hsu, M.; and Ladin, R. 2001. Business process coordination: State of the art, trends, and open issues. In *Proceedings of the 27th VLDB Conference*.
- Edelkamp, S., and Hoffmann, J. 2004. The language for the 2004 international planning competition. <http://ls5-www.cs.uni-dortmund.de/edelkamp/ipc-4/pddl.html>.
- Fdez-Olivares, J.; Castillo, L.; García-Pérez, O.; and Palao, F. 2006. Bringing users and planning technology together. Experiences in SIADEx. In *Proceedings ICAPS06*, 11–20.
- Fdez-Olivares, J.; Castillo, L.; Czar, J. A.; and Prez, O. G. 2011. Supporting clinical processes and decisions by hierarchical planning and scheduling. *Computational Intelligence* 27(1):103122.
- Gerevini, A., and Long, D. 2006. Plan constraints and preferences in PDDL3. *ICAPS 2006* 7.
- González-Ferrer, A.; Fdez-Olivares, J.; Sánchez-Garzón, I.; and Castillo, L. 2010. Smart Process Management: automated generation of adaptive cases based on Intelligent Planning technologies. In *Proceedings of the Business Process Management 2010 Demonstration Track*.
- Kortenkamp, D.; Bonasso, R.; Schreckenghost, D.; Dalal, K.; Verma, V.; and Wang, L. 2008. A procedure representation language for human spaceflight operations. In *Proceedings of i-SAIRAS-08*.
- McCluskey, T. L.; Liu, D.; and Simpson, R. M. 2003. GIPO II: HTN planning in a tool-supported knowledge engineering environment. In *13th ICAPS*.
- van Der Aalst, W.; Ter Hofstede, A.; Kiepuszewski, B.; and Barros, A. 2003. Workflow patterns. *Distributed and parallel databases* 14(1):5–51.
- van der Aalst, W.; ter Hofstede, A.; and Weske, M. 2003. Business process management: A survey. *Business Process Management* 1019–1019.
- Vaquero, T.; Silva, J.; Ferreira, M.; Tonidandel, F.; and Beck, J. 2009. From requirements and analysis to PDDL in itSIMPLE3. 0. In *ICKEPS'09: Proceedings of the 3rd. International Competition on Knowledge Engineering for Planning and Scheduling*, 54–61.
- White, S. 2004. Introduction to BPMN. *IBM Cooperation* 2008–029.

# VisPlan – Interactive Visualisation and Verification of Plans

**Radoslav Glinský, Roman Barták**

Charles University in Prague, Faculty of Mathematics and Physics

Malostranské nám. 25, 118 00 Praha 1, Czech Republic

radkog1@gmail.com, bartak@ktiml.mff.cuni.cz

## Introduction

Plan analysis is an inevitable part of complete planning systems. With the growing number of actions and causal relations in plan, this analysis becomes a more and more complex and time consuming process. In fact, plans with hundreds of actions are practically unreadable for humans. In order to make even larger plans transparent and human readable, we have developed a program which helps users with the analysis and visualization of plans. This program called VisPlan finds and displays causal relations between actions, it identifies possible flaws in plans (and thus verifies plans' correctness), it highlights the flaws found in the plan and finally, it allows users to interactively modify the plan and hence manually repair the flaws.

## Existing Tools

Though the number of planners rapidly grows, the number of available tools for user interaction with planners is still limited. Three complex systems are worth mentioning as they are publicly available and provide graphical user interface supporting the planning process: itSimple (Vaquero et al. 2010), GIPO (Simpson et al. 2007), and VLEPPO (Hatzl et al. 2010). They are effective tools for modelling and updating planning domains, however, their plan analysis lacks some handy features such as:

- recognizing causal relations of actions
- compact overview of actions' preconditions and effects
- support for plans with flaws
- information about world state at a specific plan step
- a user friendly interface to modify, insert, and delete actions in a plan and to re-verify the plan in real-time.

VisPlan focuses on all above features.

## VisPlan

VisPlan is a graphical application (Figure 1) written in Java with the ultimate goal to visualize any plan, to find and highlight possible flaws, and to allow the user to repair these flaws by manual plan modification.

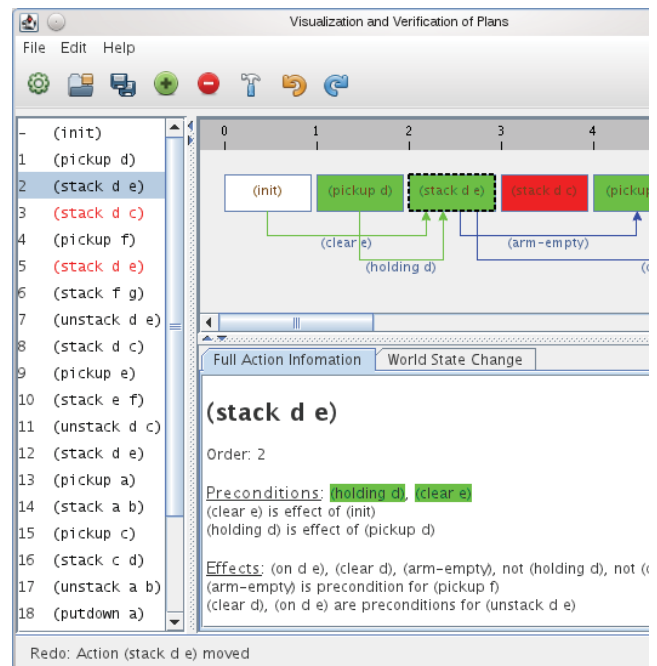


Figure 1. Graphical user interface of VisPlan.

## Program Input

VisPlan works with three types of files that the user should specify as program input:

- planning domain file in PDDL
- planning problem file in PDDL
- plan file specified in text format

VisPlan supports STRIPS-like plans and temporal plans. The program recognizes the plan type (strips/temporal) automatically and verifies and visualizes it based on its type. The plan type is determined by the planning domain – durative actions indicate a temporal plan, actions with no duration indicate a STRIPS-like plan. The following PDDL requirements are currently supported in the program: strips, typing, negative-preconditions, disjunctive-preconditions, equality, existential-preconditions, universal-preconditions, quantified-preconditions, durative-actions.

Planning domain and problem need to be syntactically correct and mutually consistent (separately parsed planning domain and problem files can be linked with each other). Otherwise, visualization and verification is not performed and errors from the PDDL parser are displayed. Sometimes, PDDL parser encounters errors and issues which are not critical. In these cases, warning and non-critical error messages are displayed and the program continues. Recognized plan actions are given in the following format:

```
start_time: (action_name param1 param2
...) [duration]
```

In the plan file each action is supposed to be on a separate line. The parser recognizes the lines and creates actions given only in the above mentioned format. Other lines are ignored. Eventually, a modified plan can be saved either to the original file or to a new text file.

## Verification

Plan verification is automatically executed after the plan is initially loaded and then after each user interaction modifying the plan. The verification process is based on simulation of plan execution and the main idea is to incrementally construct “layers” of facts. Each fact layer is determined by a corresponding set of facts and an action due to which the layer has been created.

At the beginning of the verification, all possible facts (grounded predicates) are instantiated. This domain-specific data remains fixed and is computed only once at the beginning; re-verifications do not change the data. This attitude permits us not to manipulate with the facts during the whole verification process, but to work only with the indexes to the array of grounded facts. Because of that, operations like checking if an action is applicable, application of action’s effects, finding missing conditions, etc. are just logical bit-sets operations (where one bit-set has its bits set to true at indexes corresponding to the selected grounded facts). Such operations are very fast.

Unlike facts, only actions present in the plan are grounded (meaning related to an operator with grounded conditions and effects). The operator is found based on matching the planning-domain operator and concrete parameters of the action. As mentioned in the previous paragraph, conditions and effects of the grounded operator are represented by bit sets (pointing to the fix array of grounded facts). The verification process makes sure it has a matching operator available for each examined plan action (otherwise, for instance when a user adds a new action, the verification process additionally finds and stores the operator). Actions, which do not comply with any operator definition, are marked as invalid and omitted from the verification. Nevertheless, such actions are still displayed (but distinguished from others by a different colour and marked as invalid).

There are two special “actions” artificially added into the plan. They are called “init” and “goal” and their aim is to represent the initial state and the goal. A classical plan-

space approach is used to define these actions. The init action has empty preconditions and the facts that apply at the initial state are considered as its effects. The goal action has empty effects and the set of facts that need to be satisfied at the final world state are considered as its preconditions. By treating the initial state and the goal as regular plan actions we are able to recognise causal relations also at the margins of the plan without any further work. This way we easily find dependencies on the initial state and, eventually, marking the “goal” action as non-applicable means that the goal conditions are not satisfied.

## Finding action’s matching operator

In order to find a matching operator for an action we have to go through the planning-domain operator expressions and find an operator which:

- matches action’s name
- matches the number of action’s parameters
- each action’s parameter belongs to a (typed) domain of respective operator’s variable, where the domain is a set of concrete objects in the planning problem such that object’s type is equal to the variable’s type (or variable’s deduced type)

Upon correspondence, every couple (variable, parameter) is bound and added into a “substitution” object. This substitution is consequently applied on the operator’s conditions and effects, thus ensuring they are grounded since then. Afterwards, the algorithm separately converts the grounded conditions’ and effects’ compound expressions into a set of trivial expressions (for STRIPS-like actions each such expression is either a literal or, for durative actions, a timed expression including just one literal).

In the final step, an operator is created based on the trivial expression set from the previous step. For STRIPS-like actions the following bit-sets are instantiated: positive preconditions, negative preconditions, positive effects, and negative effects. If the literal from the literal set is an atomic formula, the index of atomic formula (which is, indeed, a grounded fact, one of the facts in the initially created array of facts) is added to positive preconditions/effects bit-set. On the other hand, if the literal is a “not (atomic formula)”, the index of atomic formula is added to negative preconditions/effects bit-set.

For durative actions the literal is obtained from a timed expression (one of the following: “at start (literal)”, “over all (literal)”, “at end (literal)”). And, similarly, index of literal’s atomic formula is added to one of the following sets: at start conditions, over all conditions, at end conditions, at start effects, at end effects (each positive or negative depending on the literal).

Artificial operators for special “init” and “goal” actions are constructed as well. Conditions for the “goal” operator are obtained in the same way as conditions for any regular plan action with an exception that the goal expression is separately taken from the parsed PDDL problem file. In contrast to the “goal” operator, for the “init” operator there is already a predefined and grounded set of facts (atomic



formulas) from the separately taken init expression. These facts (represented as a bit-set) are then assigned to the “init” operator’s effects. In addition to grounded facts, the init PDDL expression may contain equality comparison functions, for instance:

```
(= (drive-time l1 l2) 4.3)
```

Function name plus its arguments (the first argument of the above equality comparison function) is assigned a numerical value representing time duration (the second argument of the above equality comparison function). Couple (*duration function*, *duration value*) is stored and used when creating an operator matching the durative action. At this time, the duration of action is obtained from the parsed PDDL domain file, grounded (by the same substitution as action’s conditions and effects) and searched within previously stored duration functions. Duration value of the found function is assigned to the matching operator of the currently manipulated action.

### Verification of STRIPS plans

Verification is realised via simulation of plan execution. Firstly, we construct an empty layer of facts. After that, we consecutively try to apply a single action (in the order given by the sequential plan) to the current world state represented by the last fact layer. If the action is applicable, the action is applied and a new world state is computed based on the effects of the action. If the action is not applicable, its effects are not encountered and the verification starts processing the next action in the plan. For instance, after the first “init” action is successfully applied, we have constructed the initial world state as defined in the planning problem. An action is applicable to a given fact layer if and only if the layer contains all action’s positive preconditions and simultaneously excludes all negative preconditions. If the action is applicable, a new fact layer is created. The new set of facts is computed based on the previous fact layer extended by the facts from action’s positive effects and excluding action’s negative effects.

Fact layer against which an action under examination is trying to be applied is remembered. If applicable, the fact layer which the action has created is stored as well. For STRIPS-like plans the first and the second fact layer are next to each other. In temporal plans, a difference between these two layers can vary a lot, as there can be arbitrary number of other actions’ starts and ends between them (each start and end of durative action possibly creates a new layer). Such stored information will be used when finding how the world changes by applying the action (the actual set of facts prior and after the action).

Missing preconditions of the action (if any) and causal relations to previous actions in the plan are also computed for each action during its verification. In the visualization, an action is applicable if and only if its set of missing preconditions is empty. If a precondition of the action is not missing, we find the last fact layer from which the precondition fact is included (on the other hand, for

negative precondition we find the last fact layer from which the precondition fact is excluded). The precondition then depends on the action assigned to that fact layer.

After each modification the plan is immediately re-verified.

### Verification of temporal plans

Verification of temporal plans is similar to STRIPS plans’ verification regarding the plan execution simulation and fact layers’ construction. The main difference is that a single durative action, besides “at start” conditions (equivalent to strips preconditions), defines “over all” and “at end” conditions. Therefore, a durative action needs to be checked multiple times whether it is applicable or not. Similarly, in a general case one action can create more than one fact layer, when both “at start” effects and “at end” effects are encountered.

Prior the verification, at the time the algorithm is creating operators matching to actions, each durative action is checked to have the duration complying with the duration specified for the operator in the planning-domain. In case the two durations vary, a user is prompted (in a new question message dialog) to accept or deny modification of action’s duration to the one specified in the planning domain. Once a user has denied action’s duration modification, he/she is never prompted again for the same action. When many actions from the plan under examination have similar conflict, the user is given a possibility to accept/deny modifications for all actions. Nevertheless, this doesn’t affect new actions eventually added to the plan.

During the verification every action is processed twice. Firstly, at its start time, when “at start” and “over all” conditions are checked against the current fact layer. If the action is applicable it is applied (taking its “at start” effects into consideration), resulting in creation of a new fact layer. In addition, the action is remembered to be “in progress” internal state. Secondly, at action’s end time (start plus duration time), the action finds out whether it has been applied at its start. If so, “at end” and “over all” conditions are checked and, if satisfied, the action is applied (considering its “at end” effects). The action is removed from “in progress” actions at this phase.

When processing an action during verification either at its start time or its end time, besides checking its own conditions, the algorithm checks also “in progress” actions (those which have already started but haven’t finished yet) to verify their “over all” conditions. Such verification is performed only when the inducing action is applied (either at its start time or end time).

In case an action’s “at start” effects have been applied at action’s start time and it has later been found that any of action’s “over all” and “at end” conditions are not satisfied, the verification process is reverted back to the point when the affected action was applied at its start time, the action is omitted then and marked as non-applicable.

Similarly to STRIPS plans’ verification, possibly missing conditions for an action are found while

processing the action. However, for durative actions we store three different types of missing conditions: “at start”, “over all” and “at end” missing conditions sets. Thus, in a future plan analysis, missing conditions are already available without need to be computed.

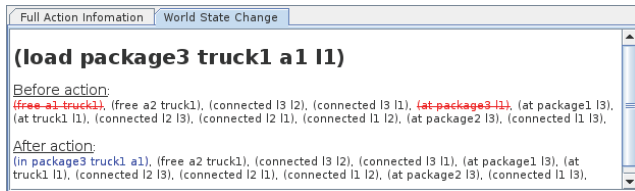


Figure 2. Example of information about world-state change.

## Visualization

As shown in the right-upper frame of Figure 1, plan’s actions are visualized as cells (boxes) of fixed size filled by the action name. Each action is coloured either green or red (or any other colour chosen by the user) depending on whether the action is applicable or non-applicable. Causal relations between the actions are visualized by edges. These edges are annotated by grounded facts that are “passed” between the actions. Only the causal relations for the currently highlighted action are displayed to remove a cluttered view. Display position of the edges is automatically adjusted every time an action is highlighted in order to assure that the edges do not overlap and their labels (describing the causal relations) are fully readable. The edge position adjustment is vertical (with fixed space size between edges), as well as horizontal (source and target points of edges on the same cell have regular space between themselves).

If the process of verification is still going on, actions whose state has not been decided yet are coloured gray (or any other colour chosen by the user). The state of an action can be one of the following:

- invalid (action doesn’t match any definition in the planning domain file),
- un-decided (action is still being checked by the validation module),
- applicable (action is valid and can be used),
- non-applicable (action cannot be used due to non-satisfied preconditions).

Two special actions, “init” and “goal” are coloured differently to distinguish their special meaning. These are the only two actions which cannot be modified in any way.

For the highlighted action, the system displays complete information about the action including the satisfied and violated preconditions and actions giving these preconditions (the right-bottom frame of Figure 1), as well as world change caused by the action (Figure 2). World change illustrates which facts are true prior the action and which after the action. Naturally, world state information is not available for non-applicable actions. Facts that were subject of change (either added or deleted)

are marked (by colour and/or by strike through their names).

On the left side of the window a list of actions is shown to provide a brief plan summary (the left frame of Figure 1). Actions in the list are sorted by their order/start time and are visually differentiated based on their states. The list gets updated every-time a modification is done to the plan. Selecting an action in the list results in adjusting the scrollbar view to comprise the visualized action in the graph and vice versa. If the user needs more space for graphical plan analysis he/she is free to hide the action summary list completely (as well as informative tab pane at the bottom of the application).

During a plan analysis, the ruler (Figure 1) helps to orientate within a time axis. Its default size of units is one inch (without dependence on user’s screen resolution). Size of units can be adjusted by the combo box (upper-right frame of Figure 1) or by dragging any tick of the ruler.

While dragging an action (to change its position), actions providing preconditions and actions using effects of the dragged action are dynamically highlighted, so that the user knows where he/she can drop the action. When actions are swapped it usually changes causal relations between the actions significantly. Due to this fact, highlighting preconditions and effects partially wouldn’t provide enough information. Therefore the plan is re-verified when an action changes its order while dragging. Having such information the program chooses the correct actions to highlight. Colour for highlighting is the same as colour for preconditions/effects edges. If actual colour of an action is the same as the colour for edges when highlighting, another (but similar) colour is used then.

Each user has an opportunity to set his/her own user preferences regarding the visual appearance and behavior of software according to the personal needs. The user preferences are saved in the home directory of the user and include various (mostly graphical) settings, for instance:

- colors for actions (each state has its own color), edges (both preconditions and effects) and ruler,
- font size (for different GUI components),
- automatic loading of last successfully loaded files (domain, problem, plan) at start-up,
- default action width in STRIPS-like plans.

## Visualization of STRIPS plans

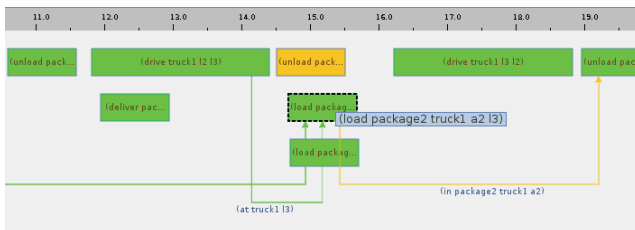
As the STRIPS plans are sequential, cells representing the actions are displayed in a row. When changing the order of an action by drag & drop, the new order is computed after each movement by checking the horizontal position of the cell being dragged and ruler’s units. In the case the new position is different from the current one, a cell placed at that moment on the “new order” position is immediately repositioned to the “current order” position, and thus these two actions swap their position. When the action is finally dropped, it is just placed in the row.

## Visualization of temporal plans

Ruler units in temporal plans reflect durations of actions. However, as individual durations of actions within a plan can vary a lot, the median duration has been chosen to be the initial ruler unit. Auxiliary ticks are also present on the ruler. All actions (meaning cells) are also guaranteed to have a minimum horizontal size (in order to be visible even if real duration is too small).

Horizontal position of an action is fully determined by its start time and duration. Although actions in temporal plans can overlap with each other, cells representing the actions are positioned in order to be fully visible. This is performed by placing the cells in rows. All cells in the same row have the same vertical position. Cells position adjustment is iterative and cells are positioned into the first row (from top) where the cell would not overlap with other cells (Figure 3).

When an action is being dragged, in contrast to STRIPS plans, the start time of the action is determined by the horizontal position only (multiplied by the current ruler units). In such a situation re-verification of the plan is done only when the action has changed its position significantly, meaning the relative order of the dragged action margins (start/end) changed with respect to other actions.



**Figure 3.** Example of visualisation of temporal plans.

## Plan Modifications

In addition to visualization of plans the software supports interactive modification of the plan. The following operations with plans are supported:

- inserting new actions (selection of actions and their parameters is automatically restricted to the current planning domain and the problem and offered in the corresponding number of pre-filled combo boxes),
- removing actions,
- modifying actions,
- changing the order of actions in STRIPS plans and start time of action in temporal plans by drag & drop technique.

Modifications are revertible and are under control by undo manager. Undo manager waits for performing an undoable (revertible) modification, which is any of the above. When an undoable change is fired, undo manager clones and saves both the current plan and vericator state (this includes the constructed layers of facts, the causal relations among actions, actions' indexes to layers before and after

application, missing conditions). On the one hand, this approach is more memory consuming, due to the fact that undo manager saves as many plans and verifier states as is the limit of possible “undo”s. On the other hand, the approach is time-saving. Re-verification is not needed to be performed after each “undo”/“redo”. All the necessary steps include just retrieving previous/next plan and verifier state plus redrawing the graph based on the retrieved plan. In comparison with a memory-saving approach, which would save only modifications’ description and would perform opposing action during “undo”/“redo”, the chosen approach is easier and more “defect-resistant”. That is because it coherently maintains entire plans and states.

Besides the already mentioned plan and verifiator state, undo manager saves two more items for user-friendliness and informative purposes. These include id of an action causing an undoable change (in order to select this action and to adjust view to comprise it) and a string describing the change (in order to print informative message onto status panel at the bottom of the application).

Modified plans can be saved in the text format to either the same (initially loaded) file or to a new file (save as).

## Future Development

The up-to-date version of the software can be downloaded from <http://glinsky.org/visplan>. The program is under continuous development and in a near future it will support additional features such as:

- wider support of PDDL requirements
- support of plans in PDDL+
- own planning module
- support for finding possible plan modifications in order to solve flaws in the plan
- graphs visualizing a timeline of predicates and numerical variables during plan execution

## Acknowledgements

The research is supported by the Czech Science Foundation under the contract P103/10/1287.

## References

- Simpson, R.M.; Kitchin D.E.; McCluskey, T.L. 2007. Planning Domain Definition using GIPO. *The Knowledge Engineering Review* 22(2): 117-134.
- Vaquero, T. S.; Silva, J. R.; Beck, J.C. 2010. Analyzing Plans and Planners in itSIMPLE3.1. In: Proceeding of the ICAPS 2010 Knowledge Engineering for Planning and Scheduling Workshop. Toronto, Canada, pp. 45-52.
- O. Hatzi, D. Vrakas, N. Bassiliades, D. Anagnostopoulos, I. Vlahavas. VLEPPO system, A Visual Programming System for Automated Problem Solving, *Expert Systems With Applications*, Elsevier, Vol. 37 (6), pp. 4611-4625, 2010.