Proceedings of the Workshop on Knowledge Engineering for Planning and Scheduling

Toronto, Canada
May 13, 2010

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@poli.usp.br

Program Committee

Piergiorgio Bertoli, Fondazione Bruno Kessler, Italy

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

Adi Botea, NICTA/ANU, Australia

Amedeo Cesta, ISTC-CNR, Italy

Stefan Edelkamp, Universität Dortmund, Germany

Susana Fernández, Universidad Carlos III de Madrid, Spain

Antonio Garrido, Universidad Politecnica de Valencia, Spain

Arturo González-Ferrer, University of Granada, Spain

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

Ugur Kuter, University of Maryland, U.S.A.

José Reinaldo Silva, University of Sao Paulo, Brazil

Dimitris Vrakas, Aristotle University of Thessaloniki, Greece

Foreword

Knowledge Engineering is a wide area of artificial intelligence, spanning many branches of knowledge-based systems, and is concerned with the elicitation, formulation, validation and maintenance of a knowledge base. Planning and Scheduling are problem areas that demand knowledge of dynamical systems, and in particular, knowledge of change, of action and of process. Whereas in engineering and mathematics, knowledge of dynamical systems is used by people, for automated planning and scheduling this knowledge has to be encoded in such a way that programs can manipulate and reason with it. To make matters even more complex, planners and schedulers not only require input in an expressive, accurate, operational and precise form (encoded in the ubiquitous "domain model"), but also perform the synthetic task of creating plans and schedules to achieve goals. Hence, when considered in relation to the problems areas of automated planning and scheduling, knowledge engineering is considered to be particularly difficult.

For the purposes of this workshop, we define knowledge engineering for automated planning and scheduling to cover the processes of acquisition, validation and maintenance of domain models, as well as the selection and optimization of appropriate machinery to work on them. These processes impact directly on the success of real planning and scheduling applications: to function efficiently applications still need to be fed by careful problem description, and need to be fine tuned for particular domains or problems. The importance of knowledge engineering techniques is clearly demonstrated by a performance gap between domain-independent planners and planners exploiting domain dependent knowledge.

The KEPS series of workshops has been running for some time interleaving with ICKEPS (The International Competition on Knowledge Engineering for Planning and Scheduling) with the goal to promote knowledge engineering aspects of planning and scheduling. KEPS 2010 was held in May 2010 during 20th International Conference on Automated Planning and Scheduling (ICAPS). The current edition includes seven technical papers all reviewed by international program committee and four system demonstrations. The width and variety of papers in this volume demonstrates the span and importance of the field. The papers cover topics such as planning extensions, eliciting planning information, acquisition of hierarchical descriptions, automated model updating, translation between representations, verification of plans and post-design analysis. The applications span from computer games, through business processes to space exploration. System demonstrations include both academic and industrial systems; two winners of recent ICKEPS demonstrate advancement of their tools.

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 2010 Organizers
May 2010

Contents

Full Technical Papers

Eliciting Planning Information from Subject Matter Experts 5
Pete Bonasso, Mark Boddy

How Hard is Verifying Flexible Temporal Plans for the Remote Space Agent? 13
Amedeo Cesta, Alberto Finzi, Simone Fratini, Andrea Orlandini, Enrico Tronci

Ontology Oriented Exploration of an HTN Planning Domain through Hypotheses and Diagnostic Execution 21
Li Jin, Keith S. Decker

Model Updating in Action 29
Maria V. de Menezes, Leliane N. de Barros, Silvio do L. Pereira

Integrating plans into BPM technologies for Human-Centric Process Execution 37
Juan Fdez-Olivares, Inmaculada Sánchez-Garzón, Arturo González-Ferrer, Luis Castillo

Improving Planning Performance Through Post-Design Analysis 45
Tiago Stegun Vaquero, José Reinaldo Silva, J. Christopher Beck

An XML-based Forward-Compatible Framework for Planning System Extensions and Domain Problem Specification 53
Eric Cesar E. Vidal, Jr., Alexander Nareyek

System Demonstrations

Constraint and Flight Rule Management for Space Mission Operations 62
Javier Barriero, John Chachere, Jeremy Frank, Christie Bertels, Alan Crocker

Using Knowledge Engineering for Planning Techniques to leverage the BPM life-cycle for dynamic and adaptive processes 64
Juan Fdez-Olivares, Arturo González-Ferrer, Inmaculada Sánchez-Garzón, Luis Castillo

Analyzing Plans and Comparing Planners in itSIMPLE3.1 66
Tiago Stegun Vaquero, José Reinaldo Silva, J. Christopher Beck

Visual design of planning domains 68
Jindřich Vodrážka, Lukáš Chrpa

Full Technical Papers

Eliciting Planning Information from Subject Matter Experts

Pete Bonasso* , Mark Boddy**

*Traclabs, Inc, 1012 Hercules, Houston, TX 77059, bonasso@traclabs.com
**Adventium Enterprises, 111 3rd Ave South, Suite 100, Minneanapolis, MN 55401, mark.boddy@adventiumlabs.org

Abstract

Over the past several months, we have been engaged in layering planning information onto execution procedures for supporting NASA operations personnel in planning and executing activities on the International Space Station (ISS). The procedures are captured in the Procedural Representation Language (PRL). The planning information is to be integrated with the procedural information using a PRL authoring system. This paper describes an initial design for eliciting planning information from the domain experts who created the procedures. The goal is to generate actions in standard planning languages that automated planners can use to generate executable plans. Of particular note is that the resulting action representations support both goal and action HTN decompositions.

Introduction and Motivation

There have been a number of recent efforts, most notably the Automation for Operations (A4O) initiative (Frank 2009), to provide NASA flight controllers with activity planning and execution aids by leveraging maturing execution (Vera et al 2006) and planning technology (e.g., Chien et al, 2003, and Bedrax-Weiss et al 2005). One of those technologies is the development of a procedure representation language (PRL) that both captures the form of traditional procedures and allows for automatic translation into code that can be executed by NASA-developed autonomous executives. PRL provides for access to spacecraft and habitat telemetry, includes constructs for human-centered displays, allows for the full range of human interaction, and allows for automatic methods of verification and validation. As well, PRL is being developed with a graphical authoring system, known as PRIDE, that enables non-computer specialists to write automated procedures (Kortenkamp et al 2007).

Given a set of procedures cast in PRL, one of our current research goals has been to enhance the PRL language to include planning information related to each procedure, i.e., a) time, for both task duration and for temporal constraints among procedures, b) resources that are required, produced or consumed by a procedure, c) pre-conditions, 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. Our target flight disciplines have been Extravehicular Activity (EVA) and Power, Heat and Light Control (PHALCON). The two disciplines often work together because spacewalks usually entail the installation or removal of power equipment around the International Space Station (ISS). (Bonasso & Boddy 2009) details the results of our first year efforts in both "chunking" large EVA and PHALCON procedures into primitives for planning as well as developing PRL representations for time, resources, preconditions and effects that can easily translate into standard planning languages, our target being ANML (Smith & Cushing 2008).

A second major research goal is to design an interaction scheme as an addition to PRIDE that will elicit these planning data from the EVA and PHALCON flight controllers, the same experts who developed the PRL procedures. These subject matter experts have little or no understanding of automated planning technology. This paper describes our initial approach to obtaining from these experts planning information sufficient to be used by automated planners.

Goal versus Action Decomposition

Much of the activity planning done by PHALCONs and virtually all done by EVA flight controllers lends itself to Hierarchical Task Net (HTN) planning. Standard HTN decomposes a task into actions, but some planners, e.g., SIPE (Wilkins & Myers 1998) and AP (Applegate et al 1990), a planner we've used for several NASA applications, use goal decomposition. To illustrate, consider an EVA task to retrieve an external light known as a Crew and Equipment Translation Aid light, or CETA-light. A stripped down action description would be:

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Define-action: retrieve-light
  Parameters: ev – crew, light – CETA-light
  Variables: bag – ORU-bag, light-loc – location
  Expansion:
    Sequence
     Pick-up (ev, bag)
     Translate-by-handrail (ev, light-loc)
     Extract-item-to-bag (ev, light, bag)

Basically, the crewmember gets the orbital replacement unit (ORU) bag, travels to the light location and unbolts and stores the light in the bag.

A plan using the above definition will always have three sub-actions. So the first action will still be planned even if the conditions of the initial situation include the fact that the crewmember already possesses the bag.

A goal-decomposition of the above expansion might be:

Expansion:
  Sequence
   Possessed-by (bag, ev)
   Located (ev, light-loc)
   Extracted-into-bag (ev, light, bag)

This form asks the planner to find actions that will bring about the goals (states) in the order specified. However, if any goal already holds, no action need be planned. So if the crewmember already possesses the bag at the outset, only actions for locating the crewmember and getting the light in the bag will appear in the plan. Additionally, a goal decomposition does not specify what action to take to bring about a desired goal, so, in the above example, any action that will position the crew member at the light location can be used, like traveling on a CETA cart or on the space station robotic arm. In essence, an expansion of goals is a template for many action decompositions. In practice there are always actions whose goal/state/intent is just that the action be successfully completed, which is the case for extract-item-to-bag.

While our design favors goal decomposition, our approach to building an interactive aid to elicit planning information will produce a representation from which either or both action and goal decompositions can be derived.

The Interactive Paradigm

We now describe a query-response flow of interaction in PRIDE to obtain the planning information needed to construct complete action descriptions, including actions with decompositions. We assume in this design that all the executable level actions – called procedures – have been defined in files with PRL representations. We also assume a domain ontology is available to the PRIDE system. Obtaining those is a non-trivial effort – we spent a year constructing these for our domain. The examples below are taken from our models of the EVA domain.

Goal Representation

First we analyzed the primitive actions/procedures to develop a set of domain relations that can serve as goals or actions (this set will need to be expanded as users determine there are other relations that should be modeled). Here are examples of a goal and an action relation:

Relation:
  Name: located
  Type: fluent
  Function?: yes
  Args:
   object – thing
   location – geographicarea
  Verb-form: "locate object at location"
  Prefix-form: "object is at location"

Relation:
  Name: extract-item-to-bag
  Type: action
  Function?: no
  Args:
   crew - agent
   object - station-object
   bag - ORU-bag
  Verb-form: "crew extract object to bag"
  Prefix-form: "crew has extracted object to bag"

The type field is used to distinguish goals that can have a fluent form and those that are purely actions, that is, there is no corresponding state condition that could be used as a goal. The function field allows planners to take advantage of single-value fluents. For example, rather than

Variables: loc1 – location, crew1 crew 2 - agent
Conditions: located(crew1, loc1)
      located(crew2, loc1)

one can write:

Variables: crew1 crew 2 - agent
Conditions: located(crew1) = located(crew2)

The prefix forms are used to display the relation to the user as either effects or conditions; the verb form, as actions in a decomposition (see Building Decompositions below).

Obtain the Action

A subset of the relations described above corresponds to the PRL procedures assumed to exist for this endeavor. PRIDE will derive the action name as well as the intent of the action from this set. For this example we'll be using the procedures known as `pick-up, travel-by-handrail, travel-by-SSRMS, and extract-item-to-bag.`

So PRIDE will first direct the user to select the procedure to which planning information is to be added. Our user selects `extract-item-to-bag` and the template shown in Figure 1 appears.

```
Action:    extract-item-to-bag
Agents:
Duration:        minutes

Conditions:


Effects:

Comment:
```

Figure 1 Action Template

This is a two-step procedure, written in PRL, wherein the crewmember unbolts the item with a power grip tool and stows it in an ORU bag.

Obtain the Intent

Next PRIDE directs the user: *Select a goal that is the intent of this action*. The user selects from the list of relations described in the section on goal representation above and presented to the user in their prefix form. Our user selects "crew has extracted object to bag" and the template updates as in Figure 2:

```
Action:    extract-item-to-bag
Agents:    crew1
Duration:        minutes
Parameters: object1 is a station-object
            bag1 is an oru-bag
Conditions:


Effects: crew1 has extracted
           object1 to bag1

Comment:
```

Figure 2 Action Template with Intent

When the relation is selected, PRIDE uses the type information for the relation's arguments to fill in the Agents and Parameters fields. Instances of parameters are constructed from the argument names. The agents are called out separately from the parameters so that other non-planning applications can use that information from the final result.

Determine Needed Tools

Another source of parameters will involve tools used in the procedure. So the user is now asked: *Are any tools needed for this procedure?* A taxonomy of the tools in the EVA domain are presented to the user as shown in Figure 3. The

```
TOOL
 RATCHET-WRENCH
  ratchet-wrench1
 SQUARE-SCOOP
  square-scoop1
 POWER-GRIP-TOOL
  PGT-WITH-TURN-SETTING
   pgt1
   pgt3
  PGT-WITH-TORQUE-BREAK-SETTING
   pgt2
```

Figure 3 Tool Taxonomy (with instances in lower case)

user selects a power grip tool with a precision ratchet, which shows up as pgt1 in the parameters.

Obtain the Decomposition

If this were a new action, the user would be asked at this point to define the decomposition. Since the current action is a primitive, PRIDE will not query for a decomposition (but see Building Decompositions below).

Obtain the Preconditions

Rather than asking the user an open-ended question like,

```
Action:    extract-item-to-bag
Agents:    crew1
Duration:        minutes
Parameters: object1 is-a station-object
            bag1 is-a oru-bag
            pgt1 is-a power-grip-tool
            loc1 is-a geographic-area
Conditions: crew1 has bag1
            crew1 has pgt1
            crew1 is at loc1
            object1 is at loc1
Effects: crew1 has extracted
           object1 to bag1
Comment:
```

Figure 4 Action Template with Tools and Conditions

What conditions must be true for this procedure to be applicable?, we use a series of "wizard" questions keyed on common conditions such as location, possession and containment. For possession, PRIDE assumes the crew

will be the default possessor and so asks, *Should crew1 possess any items?*, and the user selects from a pop-up menu of the parameters. In this case, user selects bag1 and the pgt1. PRIDE then uses the relational form of possessed-by to construct the appropriate preconditions.

Similarly, PRIDE will ask if any of the crew and/or parameters need to be co-located. In this case, crew1 needs to be at the same place as object1. The same process is used for containment, but we will illustrate that in the effects query below. The resulting template is shown in Figure 4.

As this is a work in progress, we are for now assuming

```
Action: extract-item-to-bag
Agents: crew1
Duration:        minutes
Parameters: object1 is-a station-object
            bag1 is-a oru  tool
            pgt1 is-a pow  ratchet-wrench
            loc1 is-a geo  square-scoop
Conditions: crew1 has bag  power-grip-tool
            crew1 has pgt  ceta-light
            crew1 is at loc1
            object1 is at loc1
Effects: crew1 has extracted
           object1 to bag1
Comment:
```

Figure 5 Focusing parameters with a type pull-down menu

all conditions are pre-conditions until we work out how to elicit temporal information from the user or develop some reasonable intelligent "wizard" questions to obtain it.

Focus the Parameters

PRIDE then tells the user, *Use the type drop-down menus to adjust the type of any parameter to be more specific.* A drop-down list under each parameter's type in the template contains all the subtypes for that parameter. The user activates the drop-down for station-object and selects the subtype CETA-light, as in Figure 5.

Obtain Side Effects

Again, rather than asking the user an open-ended question like, *What other effects will be true at the end of this procedure?,* we'll again use a the wizard approach and ask a series of questions keyed on common conditions such as location, possession and containment.

In this example, PRIDE uses the fact that there is a container and an object to ask the question, *At the conclusion of this action will bag1 contain an item?*, and gives a list of parameters less any containers. The user knows that the extracted item will be put in the bag so she checks object1. PRIDE uses the containment relation and the selected parameters to construct the effect as in Figure 6.

```
Action: extract-item-to-bag
Agents: crew1
Duration:        minutes
Parameters: object1 is a CETA-light
            bag1 is a oru-bag
            pgt1 is a power-grip-tool
            loc1 is a geographic-area
Conditions: crew1 has bag1
            crew1 has pgt1
            crew1 is at loc1
            object1 is at loc1
Effects: crew1 has extracted
          object1 to bag1
         bag1 contains object1
Comment:
```

Figure 6 Action Template with Side Effect

Establish Duration

The user is now asked: *How long in minutes will this procedure take?* The user can specify an integer amount of minutes, in this case, 12, or she can specify a computation (see the translate-by-handrail action below).

Provide a Text Description

Finally, the user will be asked to provide a description of

```
Action: extract-item-to-bag
Agents: crew1
Duration:  12   minutes
Parameters: object1 is a CETA-light
            bag1 is an oru-bag
            pgt1 is a power-grip-tool
            loc1 is a geographic-area
Conditions: crew1 has bag1
            crew1 has pgt1
            crew1 is at loc1
            object1 is at loc1
Effects: crew1 has extracted
          object1 to bag1
         bag1 contains object1
Comment:unbolt CETA-light and put in bag
```

Figure 7 Completed Action Template

the action in free-form English text. Our user enters, "Unbolt the CETA light and place in bag." The final action is shown in Figure 7.

Internal Representation

The main objective of this interactive exercise is to construct an internal representation that can be translated into standard planning languages, such as PDDL and

ANML. Our proposed representation, the instance resulting from the template above, is show below.

```
Action: extract-item-to-bag
Agents: crew1
Duration: 12
Parameters: crew1 - agent
            object1 - CETA-light
            bag1 - ORU-bag
Variables: loc1 - geographicarea
           pgt1 - power-grip-tool
Preconditions: operator: "=="
                var: loc1
                relation: located
                  args: object1
                relation: located
                  args: crew1
               operator: predicate
                relation: possessed-by
                  args: bag1, crew1
               operator: predicate
                relation: possessed-by
                  args: pgt1, crew1
Effects: relation: extract-item-to-bag
          args: crew1, object1, bag1
         relation: contained-by
          args: object1, bag1
Comment: "Unbolt ceta-light and put in bag"
```

Note that any parameters not in the effects are moved to the variables slot. Also, the operator slot allows the definition of functional fluents. This internal representation, along with the set of relations defined earlier, holds sufficient information to generate the following PDDL and ANML actions.

```
(define-durative-action extract-item-to-bag
  :parameters (?crew1 - crew
               ?object1 - ceta-light
               ?bag1 - oru-bag)
  :vars (?loc1 - geographicarea
         ?pgt1 - power-grip-tool)
  :duration 12.0
  :condition
  (and
   (at start (located ?crew1 ?loc1))
   (at start (located ?object1 ?loc1))
   (at start (possessed-by ?bag1 ?crew1))
   (at start (possessed-by ?pgt1 ?crew1)))
  :effect
   (and
    (at end
     (extract-item-to-bag
      ?crew1 ?object1 ?bag1))
    (at end (contained-by ?object1 ?bag1)))
  :comment "Unbolt ceta-light and put in
bag"
```

PDDL 2.1 can't take advantage of functional fluents. As well, our current planner, AP, uses only goal decomposition, so a goal form of the action, constructed by the action name and parameters is included in the effects.

```
action Extract_item_to_bag
       (agent crew1, CETA_light object1,
         ORU_bag bag1)
{
 duration := 12
 [start]{located(object1) == located(crew1);
         possessed_by(bag1) == crew1;
         exists (power_grip_tool pgt1) {
                  possessed_by(pgt1) ==
                  crew1}
  };
  [end] contained_by(object1) := bag1
}
```

ANML on the other hand, takes full advantage of functional predicates and can use both goal and action decompositions (e.g., see Building Decompositions below).

Add More Actions

We continue the example by building three more primitives, but we show only the final results (parameters are in italics). The next action is pick-up, whose intent is based on the possessed-by relation.

```
Action: pick-up
Agents: crew1
Duration: 5
Parameters: object1 is a station object,
            loc1 is a geographicarea
Conditions: crew1 is at loc1
            object1 is at loc1
Effects: crew1 has object1
Comment: "Crew untethers item and attaches
to suit."
```

Next we develop a translation action based on the located relation.

```
Action: travel-by-handrail
Duration: function: distance, path1
Agents: crew1
Parameters: loc1 is a geographicarea
            path1 is a path
            loc2 is a geographicarea
Conditions:
  the start location of path1 is loc2
  the end location of path1 is loc1
  crew1 is at loc2
Effects: crew1 is at loc1
Comment: "Crew uses handrails to go to
loc1."
```

Here the duration is computed from the path using a distance function. A list of such available domain functions will reside in the PRIDE system.

Next we develop a similar action based on the crewmember being mounted on the space station remote manipulator system (SSRMS).

```
Action: travel-by-SSRMS
Agents: crew1
Duration: function: GCA, loc2, loc1
Parameters: loc1 is a geographicarea
            arm1 is a robotic-arm
            loc2 is a geographicarea
Conditions: arm1 is located at loc2
            crew1 is on arm1
Effects: crew1 is at loc1
Comment: "Crew GCAs arm to loc1"
```

Here the duration is computed using the Ground Controlled Approach (GCA) function with the start and end locations as arguments.

Building Decompositions

There is no existing PRL procedure for a complex action; by definition they are composed of an ordered set of other complex actions or primitives. Our user wishes to build a retrieve action wherein a crewmember obtains a bag, travels to a worksite and extracts a CETA-light to the bag. She will go through the steps as before, with certain differences alluded to earlier because this is a new action with a decomposition.

Action Name and Intent. The user will create an action name for the new action and PRIDE will generate a relation based on an assumption that at least one crewmember is involved and at least one object. If there is no object involved in the preconditions, effects or decomposition, PRIDE will excise it from the final internal representation. In this case the user types in "retrieve item" and PRIDE generates

Relation:
  Name: retrieve-item
  Type: action
  Function?: no
  Args:
    crew - agent
    object - station-object
  Verb-form: "crew retrieve item object"
  Prefix-form: ""

and the action template that appears is:

```
Action: retrieve-item
Agents: crew1
Parameters: object1 is a station-object
Effects: crew1 retrieve item object1
```

Decompositions. As mentioned earlier, a new planning action may include a decomposition, so PRIDE asks, *Does this action have a decomposition?* In this case, the user answers in the affirmative and PRIDE presents a list of verb forms for existing relations. A subset of that list is shown below:

1) "locate object at geographicarea "
2) "object possess another object"
3) "object contain another object"
4) "crew extract CETA-light to bag"

The user selects 2) and 1), focusing object to crew, and 4), which results in the following template:

```
Action: retrieve-item
Agents: crew1
Duration: derived
Parameters: object1 is a CETA-light
            loc1 is a geographicarea
            bag1 is an ORU-bag
Expansion: sequential
             crew1 possess bag1
             locate crew1 at loc1
             crew1 extract object1 to bag1
Effects: crew1 retrieve item object1
```

The default ordering is sequential but is associated with a pull-down menu that includes `unordered`, `simultaneous` and `parallel`.

Note that the parameter object1 has been further specified by the addition of the extract action where the object type is a CETA-light. As well, parameters from the actions other than object1 and crew1 are added to the variables list. Finally, the duration is set to `derived`, since it will be an accumulation of the durations of the actions in the decomposition.

Tools. In this first pass at our design we do not query for tools in a complex actions; the bottom-up approach to building actions should cover the needed tools at the primitive level.

Preconditions and Effects. In this first pass at our design we do not allow side effects for a decomposition; the bottom-up approach to building actions should cover the needed effects at the primitive levels.

For tasks with decompositions, the usual suspects for preconditions – e.g., `crew1 has bag1`, are brought about by the actions in the decomposition. But PRIDE can reason about some aspects of this action and ask, *Is loc1 the location of object1 or bag1?* The user selects object1.

After adding a text description the action is:

```
Action: retrieve-item
Agents: crew1
Duration: derived
Parameters: object1 is a CETA-light
            loc1 is a geographicarea
            bag1 is an ORU-bag
```

```
Conditions: object1 is at loc1
Expansion: sequential
             crew1 possess bag1
             locate crew1 at loc1
             crew1 extract object1 to bag1
Effects: crew1 retrieve item object1
Comment: "crew gets bag, goes to loc and
extracts light."
```

Internal Representation. The internal representation for the above complex action is:

```
Action: retrieve-item
Agents: crew1
Duration: derived
Parameters: crew1 – agent
            object1 - ceta-light
Variables:  bag1 - oru-bag
            loc1 - geographicarea
Conditions: operator "=="
               var: loc1
               relation: located
                 args: object1
Expansion:
  Order: sequential
    Tasks:
      relation: possessed-by
        args: bag1, crew1
      relation: located
        args: crew1, loc1
      relation: extract-item-to-bag
        args: crew1, object1, bag1
Effects:  relation: "retrieve-item"
           args: crew1, object1
Comment: "Crew gets bag, goes to loc and
extracts light."
```

Here are the resulting ANML and PDDL actions.

```
action Retrieve_item (agent crew1,
                      ceta_light object1,
                      oru_bag bag1)
[duration]
{location current_location :=
                    located(object1);

[all]contains
  ordered(ach_possessed_by(bag1,crew1),
        ach_located(crew1,current_location),
      extract_item_to_bag(crew1,object1,bag1
    ))
}
```

The first two items in the expansion are goals. ANML uses an achieve action for each goal, e.g.,

```
Action ach_possessed_by(station_object item,
crew agent)[duration]
{
[start] possessed_by(item, agent) == TRUE ||
{[start] possessed_by(item, agent) == FALSE;
  [end] possessed_by(item, agent) == TRUE}}
}
```

that can be interpreted as: if the state doesn't hold at the start, find an action that will bring it about.

PDDL uses the goal form for all the actions in the decomposition:

```
(define-durative-action retrieve-item
  :parameters (?crew1 – crew
               ?object1 – ceta-light))
  :vars (?bag1 – oru-bag
         ?loc1 – geographicarea)
  :condition (at start (located ?object1
                        ?loc1))
  :expansion
    (sequential
      (possessed-by ?bag1 ?crew1)
      (located ?crew1 ?loc1)
      (extract-item-to-bag ?crew1 ?object1
        ?bag1)
:effect (at end (retrieve-item ?crew1
        ?object1))
:comment "crew gets bag, goes to loc and
extracts light."
```

Resulting Plans

For planning, the PDDL or ANML actions are selected as tasks to be planned. So the user could, for example, ask the planner to plan `bob retrieve ceta-light1` and a resulting plan might be:

```
sequence
 bob pick-up medium-oru-bag2
 bob travel-by-handrail to ceta-light1-loc
 bob extract ceta-light1 to medium-oru-bag2
```

Given an initial situation where Bob already possessed an ORU-bag, however, the plan would be:

```
sequence
 bob travel-by-handrail to ceta-light1-loc
 bob extract ceta-light1 to medium-oru-bag1
```

Given a starting situation where Bob possessed an ORU bag and was positioned on the SSRMS, the plan would be:

```
sequence
 bob travel-by-SSRMS to ceta-light1-loc
 bob extract ceta-light1 to medium-oru-bag1
```

Thus, the decompositions serve as templates of several different action combinations that could bring about the top-level goal.

Relation to Other Work

The bulk of the efforts in knowledge engineering for planning deal with AI programmers eliciting planning information from domain experts, and then using KE aids to model and validate this information. Examples are (Fernández et al 2004) and (Simpson 2007). The effort in this paper is aimed at developing planning actions from an existing set of executable procedures, by asking the procedure authors – non-AI-programmers – leading questions about the procedures. Our hope is that, through a set of focused questions to these non-AI users, we can obtain planning actions that can be used to generate valid, though possibly inefficient plans.

Like our work here, related KE efforts target standard planning languages like PDDL, NDDL and OCL. Besides PDDL, we have selected the ANML planning language, because it is based on strong notions of action and state, uses a variable/value model, supports rich temporal constraints (Smith & Cushing 2008 mention ongoing development of an ANML to NDDL translator), and provides simple, convenient idioms for expressing the most common forms of action conditions, effects, and resource usage. The language supports both generative and HTN planning models in a uniform framework and has a clear, well-defined semantics.

(Boddy and Bonasso 2010) is a companion paper that discusses the semantics of ANML including goal versus action decompositions.

Summary and Future Work

With the interactive paradigm described in this paper, we believe we can enable non-AI programmers to construct primitive and complex actions from existing procedures that can be used by AI planners to generate executable plans. Our design favors goal-based HTNs as templates for multiple methods of bringing about top-level goals. We are in the process of coding an interactive plug-in to our PRIDE system that will execute this paradigm.

What we've reported on here is of course the tip of the iceberg. Our approach using "wizard" questions will quickly become too restrictive as our domain models become more complicated. So, allowing the user to break out of the restricted question set and manage the action templates directly will involve the development of many more checks and balances to help the user avoid inadvertent errors.

And as with all KE for planning efforts, the planning models developed by this interactive paradigm must be validated. We envision running our AP planner endowed with the PRIDE-developed action set, on a set of situations to obtain valid plans. We then need to develop schemes for failure diagnosis and feeding back the results of that diagnosis to the authoring system when the planner cannot find valid plans. Initially, we will construct a single thread of that closed loop, concentrating on one or two types of authoring errors, using our existing planning aid (Bonasso et al 2009).

References

Applegate, C., C. Elsaesser, and J. Sanborn. 1990. An Architecture for Adversarial Planning. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(1): p. 186-194.

Bedrax-Weiss, T., et al., 2005. *EUROPA2: user and contributor guide*, NASA Ames Research Center.

Boddy, Mark and Bonasso, Pete. 2010. Planning for Human Execution of Procedures Using ANML. In ICAPS 2010 Scheduling and Planning Applications Workshop (SPARK). Toronto, Canada.

Bonasso, Pete, Boddy, Mark, Kortenkamp, D. 2009. Enhancing NASA's Procedure Representation Language to Support Planning Operations. In Proceedings of IWPSS09, Pasadena, CA.

Chien, S., et al. Autonomous Science on the Earth Observer One Mission. In i-SAIRAS 2003. 2003. Nara, Japan.

Frank, Jeremy. 2009. Automaton for Operations. http://ti.arc.nasa.gov/news/a4o-demo-for-hdu/

Simpson, R. M. 2007. Structural Domain Definition using GIPO IV. *The Knowledge Engineering Review*, 22, 117-134. Cambridge University Press

Susana Fernández, Daniel Borrajo, Raquel Fuentetaja, Juan D. Arias and Manuela Veloso. 2004. PLTOOL: A Knowledge Engineering Tool for Planning and Learning. *The Knowledge Engineering Review*, Vol. 00:0, 1–24. Cambridge University Press

Kortenkamp, D., R.P. Bonasso, and D. Schreckenghost. 2007. Developing and Executing Goal-Based, Adjustably Autonomous Procedures,. in AIAA InfoTech@Aerospace Conference.

Smith, D.E. and W. Cushing. 2008. The ANML Language, in iSAIRAS. Los Angeles, CA.

V. Verma, A. Jónsson, C. Pasareanu, and M. Iatauro, Universal Executive and PLEXIL: Engine and Language for Robust Spacecraft Control and Operations, American Institute of Aeronautics and Astronautics Space 2006 Conference.

Wilkins, D. and Myers, K. 1998. A Multi-agent Planning Architecture, in *Artificial Intelligence Planning Systems*, Pittsburg, PA.

How Hard is Verifying Flexible Temporal Plans for the Remote Space Agent?

A. Cesta† and **A. Finzi**‡ and **S. Fratini**† and **A. Orlandini**† and **E. Tronci**§
† ISTC-CNR, Via S.Martino della Battaglia 44, I-00185 Rome, Italy
‡ DSF "Federico II" University, Via Cinthia, I-80126 Naples, Italy
§ DI "La Sapienza" University, Via Salaria 198, I-00198 Rome, Italy

Abstract

Timeline-based planners have been shown quite successful in addressing real world problems. Nevertheless they represent a niche technology in AI P&S research as an application synthesis with such techniques is still considered a sort of "black art". Our current work aims at both creating a rational reference architecture for timeline-based planning and scheduling and developing a knowledge engineering environment around such problem solving tool. In particular we are integrating verification tools in such engineering environment to enhance typical capabilities of a constraint-based planner. In this paper we present recent results on the connection between plan generation and execution from a particular perspective: the static verification of plans before their execution. In particular, we present a verification process suitable for a timeline-based planner and show how a temporally flexible plan verification problem can be cast as model-checking on timed game automata. We here discuss the effectiveness of the proposed approach in a thorough experimental analysis based on a realistic domain called "The Remote Space Agent".

Introduction

In the past, several planning systems were endowed with development environments to facilitate application design (e.g., O-PLAN (Tate, Drabble, and Kirby 1994)). More recent examples of software development environments are EUROPA (EUROPA 2008) and ASPEN (Sherwood et al. 2000). Such environments can be enriched in several directions. In a recent work (Cesta et al. 2010b), these authors have envisaged the synthesis of knowledge engineering environments in which constraint-based and validation and verification techniques concur in creating an enhanced software environment for P&S. In particular, we are working on verification and validation methods for timeline-based planning investigating the use of model checking techniques for verifying properties of specific planning software applications.

An important problem in timeline-based planning as used in (Muscettola 1994; EUROPA 2008; Sherwood et al. 2000) is the connection with plan execution which is instrumental in several challenging real domain (e.g., the aspect is relevant for both autonomy in space and robotics). Broadly speaking such architectures return an envelope of potential solutions in form of a flexible plan which is commonly accepted to be less brittle of a single plan when coping with execution. But the general formal properties of such a representation are far from being statically defined. Some aspects of such plans have been studied by working on the temporal network which is underlying the constraint based plan representation often used by such systems – see for example (Vidal and Fargier 1999; Morris and Muscettola 2005). We have addressed the more general question of verifying flexible plans working on the more abstract plan view as set of timelines with formal tools like model checkers.

These authors have been investigating one aspect which we consider as missing: the interconnection between timeline-based planning and standard techniques for formal validation and verification (V&V). The broad aim here is the one of building a powerful environment for knowledge engineering (Cesta et al. 2010b) and also that of exploring properties that concern temporal plans and their execution (Cesta et al. 2009a; 2009b). In particular, (Cesta et al. 2009a) provides a feasibility study for the approach, while, in (Cesta et al. 2009b), some formal properties are further investigated. In this paper we mainly address a limitation of (Cesta et al. 2009b): the fact that experiments were very preliminary.

Here, that work is carried on by: (a) introducing a benchmark problem which is realistic and rich enough to allow experiments along different directions; (b) presenting a complete experimental analysis considering incrementally complex scenarios and configurations in the benchmark domain. The collected results show that the approach based on model checking can be effective in practice. Indeed, despite the increasing complexity of the verification tests, the verifier performances remain acceptable for static analysis in a knowledge engineering environment.

Preliminaries

This section shortly present the two basic ingredients we combine in our knowledge engineering environment: timeline-based planning and timed game automata. It is worth mentioning that in (Abdedaim et al. 2007) the same ingredients are put together for a different purpose than ours, namely the mapping from temporal constraint-based planning problems into UPPAAL-TIGA game-reachability problems.

Timeline-Based Planning and Execution

Timeline-based planning is an approach to temporal planning which has been applied in the solution of several real world problems – e.g., (Muscettola 1994). The approach pursues a general idea that planning and scheduling consist in the synthesis of desired temporal behavior for complex physical systems. In this respect, the set of features of a domain that needs control are modeled as a set of temporal functions whose values over a time horizon have to be planned for. Such functions are synthesized during problem solving by posting planning decisions. The evolution of a single temporal feature over a time horizon is called the *timeline* of that feature.

In the rest of this paper, the time varying features are called multi-valued *state variables* as in (Muscettola 1994). As in classical control theory, the evolution of controlled features are described by some causal laws which determine legal temporal evolution of timelines. Such causal laws are specified for the state variables in a *domain specification* which specifies the operational constraints in a given domain. In this context, the task of a planner is to find a sequence of control decisions that bring the variables into a final desired set of evolutions always satisfying the domain specification.

We assume that the temporal features we want to represent as state-variables have a finite set of possible values assumed over temporal intervals. The temporal evolutions are sequences of operational states – i.e., stepwise constant functions of time. Operational constraints specify which value transitions are allowed, the duration of each valued interval (i.e., how long a given operational status can be maintained) and synchronization constraints between different state variables.

More formally, a state variable is defined by a tuple $\langle \mathcal{V}, \mathcal{T}, \mathcal{D} \rangle$ where: (a) $\mathcal{V} = \{v_1, \ldots, v_n\}$ is a finite set of *values*; (b) $\mathcal{T} : \mathcal{V} \rightarrow 2^{\mathcal{V}}$ is the *value transition* function; (c) $\mathcal{D} : \mathcal{V} \rightarrow \mathbb{N} \times \mathbb{N}$ is the *value duration* function, i.e. a function that specifies the allowed duration of values in $\mathcal{V}$ (as an interval $[lb, ub]$). (b) and (c) specify the operational constraints on the values in (a).

In this type of planning, a *planning domain* is defined as a set of state variables $\{\mathcal{SV}_1, \ldots, \mathcal{SV}_n\}$. They cannot be considered as reciprocally decoupled but a set of additional relations exist, called *synchronizations*, modeling the existing temporal and causal constraints among the values taken by different state variable timelines (i.e., patterns of legal occurrences of the operational states across the timelines). More formally, a synchronization has the form

$$\langle \mathcal{TL}, v \rangle \longrightarrow \langle \{ \langle \mathcal{TL}'_1, v'_1 \rangle \ldots, \langle \mathcal{TL}'_n, v'_n \rangle \}, \mathcal{R} \rangle$$

where: $\mathcal{TL}$ is the reference timeline; v is a value on $\mathcal{TL}$ which makes the synchronization applicable; $\{ \langle \mathcal{TL}'_1, v'_1 \rangle \ldots, \langle \mathcal{TL}'_n, v'_n \rangle \}$ is a set of target timelines on which some values v'_j must hold; and $\mathcal{R}$ is a set of *relations* which bind temporal occurrence of the *reference* value v with temporal occurrences of the *target* values.

Timeline based planning. The temporal evolutions of a state variable will be described by means of *timelines*, that is a sequence of state variable values, a set of ordered transition points between the values and a set of distance constraints between transition points. When the transition points are bounded by the planning process (lower and upper bounds are given for them) instead of being exactly specified, as it happens in case of a least commitment solving approach for instance, we refer to the timeline as *time flexible* and to the plan resulting from a set of flexible timeline as a *flexible plan*.

It is worth mentioning that *planning goals* are expressed as desiderata of values in temporal intervals and the task of the planner is to build timelines that describe valid sequences of values that achieve the desiderata.

A *plan* is defined as a set of timelines $\{\mathcal{TL}_1, \ldots, \mathcal{TL}_n\}$ over the same interval for each state variable. The process of *solution extraction* from a plan is the process of computing (if exists) a *valid* and completely specified set of timelines from a given set of time-flexible timelines. A solution is *valid* with respect to a domain theory if every temporal occurrence of a reference value implies that the related target values hold on target timelines presenting temporal intervals that satisfy the expected relations.

Plan execution. During plan execution the plan is under responsibility to an executive program that forces value transitions over timeline. A well known problem with execution is that not all the value transitions are under responsibility of the executive but event exists that are under control of *nature*. As a consequence, an executive cannot completely predict the behavior of the controlled physical system because the duration of certain processes or the timing of exogenous events is outside of its control. In such cases, the values for the state variables that are under the executive scope should be chosen so that they do not constrain uncontrollable events. This is the *controllability problem* defined, for example, in (Vidal and Fargier 1999) where *contingent* and *executable* processes are distinguished. The contingent processes are not controllable, hence with uncertain durations, instead the executable processes are started and ended by the executive system. Controllability issues underlying a plan representation have been formalized and investigated for the Simple Temporal Problems with Uncertainty (STPU) representation in (Vidal and Fargier 1999) where basic formal notions are given for *dynamic* controllability (see also (Morris and Muscettola 2005)). In the timeline-based framework, we introduce the same controllability concept defined on STNU as follows. Given a plan as a set of flexible timelines $\mathcal{PL} = \{\mathcal{TL}_1, \ldots, \mathcal{TL}_n\}$, we call *projection* the set of flexible timelines $\mathcal{PL}' = \{\mathcal{TL}'_1, \ldots, \mathcal{TL}'_n\}$ derived from $\mathcal{PL}$ setting to a fixed value the temporal occurrence of each uncontrollable timepoint. Considering N as the set of controllable flexible timepoints in $\mathcal{PL}$, a *schedule* T is a mapping $T : N \rightarrow \mathbb{N}$ where $T(x)$ is called *time* of timepoint x. A *schedule* is *consistent* if all value durations and synchronizations are satisfied in $\mathcal{PL}$. The *history* of a timepoint x w.r.t. a schedule T, denoted by $T\{\prec x\}$, specifies the time of all uncontrollable timepoints that occur prior to x. An *execution strategy* S is a mapping $S : \mathcal{P} \rightarrow \mathcal{T}$ where $\mathcal{P}$ is the set of projections and $\mathcal{T}$ is the set of schedules. An execution

strategy S is viable if $S(p)$ (denoted also S_p) is consistent for each projection p. Thus, a flexible plan $\mathcal{PL}$ is *dynamically controllable* if there exists a viable execution strategy S such that $S_{p1}\{\prec x\} = S_{p2}\{\prec x\} \Rightarrow S_{p1}(x) = S_{p2}(x)$ for each controllable timepoint x and projections $p1$ and $p2$.

Timed Game Automata

Timed game automata (TGA) model have been introduced in (Maler, Pnueli, and Sifakis 1995) to model control problems on timed systems. In (Cassez et al. 2005), definitions related to TGA are presented in depth. Here, we briefly recall some of them that we shall use in the rest of the paper.

Definition 1 *A* **Timed Game Automaton (TGA)** *is a tuple* $\mathcal{A} = (Q, q_0, \mathrm{Act}, X, \mathrm{Inv}, E)$ *where: Q is a finite set of* locations*; $q_0 \in Q$ is the* initial location*;* Act *is a finite set of* actions *split in two disjoint sets,* Act_c *the set of* controllable *actions and* Act_u *the set of* uncontrollable *actions; X is a finite set of a nonnegative, real-valued variables called clocks;* $\mathrm{Inv} : Q \rightarrow B(X)$ *is a function associating to each location $q \in Q$ a constraint* $\mathrm{Inv}(q)$ *(the* invariant *of q); $E \subseteq Q \times B(X) \times \mathrm{Act} \times 2^X \times Q$ is a finite set of* transitions*. Where $B(X)$ is the set of constraints in the form $x \sim c$, where $c \in Z$, $x, y \in X$, and $\sim \in \{<, \leq, \geq, >\}$. We also write $q \overset{g,a,Y}{\rightarrow} q' \in E$ for $(q, g, a, Y, q') \in E$.*

A *state* of a TGA is a pair $(q, v) \in Q \times R^X_{\geq 0}$ that consists of a discrete part and a valuation of the clocks (i.e., a value assignment for each clock in X). An *admissible* state for a $\mathcal{A}$ is a state (q, v) s.t. $v \models \mathrm{Inv}(q)$. From a state (q, v) a TGA can either let time progress or do a discrete transition and reach a new state.

A *time transition* for $\mathcal{A}$ is 4-tuple $(q, v) \overset{\delta}{\rightarrow} (q, v')$ where $(q, v) \in S$, $(q, v') \in S$, $\delta \in R_{\geq 0}$, $v' = v + \delta$, $v \models \mathrm{Inv}(q)$ and $v' \models \mathrm{Inv}(q)$. That is, in a time transition a TGA does not change location, but only its clock values. Note that all clock variables are incremented by the same amount δ in valuation v'. This is why variables in X are named *clocks*. Accordingly, δ models the *elapsed time* during the time transition.

A *discrete transition* for $\mathcal{A}$ is 5-tuple $(q, v) \overset{a}{\rightarrow} (q', v')$ where $(q, v) \in S$, $(q', v') \in S$, $a \in$ Act and there exists a transition $q \overset{g,a,Y}{\rightarrow} q' \in E$ s.t. $v \models g$, $v' = v[Y]$ and $v' \models \mathrm{Inv}(q')$. In other words, there is a discrete transition (labeled with a) from state (q, v) to state (q', v') if the clock values (valuation v) satisfy the *transition guard* g and the clock values after resetting the clocks in Y (valuation v') satisfy the invariant of location q'. Note that an admissible transition always leads to an admissible state and that only clocks in Y (reset clocks) change their value (namely, to 0).

A *run* of a TGA $\mathcal{A}$ is a finite or infinite sequence of alternating time and discrete transitions of $\mathcal{A}$. We denote with $\mathrm{Runs}(\mathcal{A}, (q, v))$ the set of runs of $\mathcal{A}$ starting from state (q, v) and write $\mathrm{Runs}(\mathcal{A})$ for $\mathrm{Runs}(\mathcal{A}, (q, \vec{0}))$. If ρ is a finite run, we denote with $\mathrm{last}(\rho)$ the last state of run ρ and with $\mathrm{Duration}(\rho)$ the sum of the elapsed times of all time transitions in ρ.

A *network* of TGA (nTGA) is a finite set of TGA evolving in parallel with a CSS style semantics for parallelism. Namely, at any time, only one TGA in the network can change location, unless a synchronization on labels takes place. In the latter case, the two automata synchronizing on the same label move together. Note that time does not elapse during synchronizations.

Given a TGA $\mathcal{A}$ and three symbolic configurations *Init*, *Safe*, and *Goal*, the *reachability control problem* or reachability game $RG(\mathcal{A}$, *Init*, *Safe*, *Goal*$)$ consists in finding a *strategy* f such that $\mathcal{A}$ starting from *Init* and supervised by f generates a winning run that stays in *Safe* and enforces *Goal*.

A strategy is a partial mapping f from the set of runs of $\mathcal{A}$ starting from *Init* to the set $\mathrm{Act}_c \cup \{\lambda\}$ (λ is a special symbol that denotes "do nothing and just wait"). For a finite run ρ, the strategy $f(\rho)$ may say (1) no way to win if $f(\rho)$ is undefined, (2) do nothing, just wait in the last configuration ρ if $f(\rho) = \lambda$, or (3) execute the discrete, controllable transition labeled by l in the last configuration of ρ if $f(\rho) = l$.

Using nTGA to model timeline-based planning specifications

Timed Game Automata are particularly suitable for modeling controllability problems because the uncontrollable activities can be modeled as adversary moves. Following this approach, we perform flexible timeline-based plan verification by solving a Reachability Game using UPPAAL-TIGA. To this end, this section describes how a flexible timeline-based plan, state variables and domain theory can be modeled using nTGA. Our strategy is the following. First, timelines and state variables are mapped to TGA. Second, we model the flexible plan *view* of the world by partitioning state variables/timelines into two classes: controllable and uncontrollable. Finally, an *Observer* TGA is introduced in order to check for value constraints violations as well as synchronizations violations.

Modeling a Planning Domain as an nTGA. Let $\mathcal{PD} = \{\mathcal{SV}_1, \ldots \mathcal{SV}_n\}$ be the set of state variables defining our planning domain. We will model each $\mathcal{SV} \in \mathcal{PD}$ with a TGA $\mathcal{A}_{SV} = (Q_{SV}, q_0, \mathrm{Act}_{SV}, X_{SV}, \mathrm{Inv}_{SV}, E_{SV})$. Then the set $SV = \{\mathcal{A}_{SV_1}, ..., \mathcal{A}_{SV_n}\}$ represents our planning domain $\mathcal{PD}$ as an nTGA.

The TGA $\mathcal{A}_{SV}$ is defined as follows. The set Q_{SV} of locations of $\mathcal{A}_{SV}$ is just the set $\mathcal{V}$ of values of $\mathcal{SV}$. The initial state q_0, of $\mathcal{A}_{SV}$ is the initial value in the timeline of $\mathcal{SV}$. The set of clocks X_{SV} of $\mathcal{A}_{SV}$ consists of just one local clock: c_{sv}. The set Act_{SV} of actions of $\mathcal{A}_{SV}$ consists of the values $\mathcal{V}$ of $\mathcal{SV}$. If $\mathcal{SV}$ is controllable then the actions in Act_{SV} are controllable (i.e., $\mathrm{Act}_{SV} = \mathrm{Act}_{cSV}$), otherwise they are uncontrollable (i.e., $\mathrm{Act}_{SV} = \mathrm{Act}_{uSV}$). Location invariants Inv_{SV} for $\mathcal{A}_{SV}$ are defined as follows: $\mathrm{Inv}_{SV}(v) := c_{sv} \leq u_b$, where: $v \in Q_{SV} = \mathcal{V}$ and $\mathcal{D}(v) = [l_b, u_b]$. The set E_{SV} of transitions of $\mathcal{A}_{SV}$ consists of transitions of the form $v \overset{g,v'?,Y}{\rightarrow} v'$, where: $g = c_{sv} \geq l_b$, $Y = \{c_{sv}\}$, $v \in Q_{SV} = \mathcal{V}$, $\mathcal{D}(v) = [l_b, u_b]$, $v' \in \mathcal{T}(v)$.

Modeling a Flexible Plan as an nTGA. Let $\mathcal{P} = \{\mathcal{TL}_1, \ldots, \mathcal{TL}_n\}$ be a flexible plan for our planning do-

main $\mathcal{PD}$. We will model each $\mathcal{TL} \in \mathcal{P}$ with a TGA $\mathcal{A}_{\mathcal{TL}} = (Q_{\mathcal{TL}}, q_0, \text{Act}_{\mathcal{TL}}, X_{\mathcal{TL}}, \text{Inv}_{\mathcal{TL}}, E_{\mathcal{TL}})$. Then the set $Plan = \{\mathcal{A}_{\mathcal{TL}_1}, ..., \mathcal{A}_{\mathcal{TL}_n}\}$ represents $\mathcal{P}$ as an nTGA.

The TGA $\mathcal{A}_{\mathcal{TL}}$ is defined as follows. The set $Q_{\mathcal{TL}}$ of locations of $\mathcal{A}_{\mathcal{TL}}$ consists of the value intervals (*plan steps*) in $\mathcal{TL}$ along with a location l_{goal} modeling the fact that the plan has been completed. Thus, $Q_{\mathcal{TL}} = \mathcal{TL} \cup \{l_{goal}\}$. The initial state q_0, of $\mathcal{A}_{\mathcal{TL}}$ is the first value interval l_0 in $\mathcal{TL}$. The set of clocks $X_{\mathcal{TL}}$ of $\mathcal{A}_{\mathcal{TL}}$ consists of just one element: the *plan clock* c_p. Let $\mathcal{SV}$ be the state variable corresponding to the timeline $\mathcal{TL}$ under consideration. The set $\text{Act}_{\mathcal{TL}}$ of actions of $\mathcal{A}_{\mathcal{TL}}$ consists of the values of $\mathcal{SV}$. If $\mathcal{SV}$ is controllable then the actions in $\text{Act}_{\mathcal{TL}}$ are controllable (i.e., $\text{Act}_{\mathcal{TL}} = \text{Act}_{c\mathcal{TL}}$), otherwise they are uncontrollable (i.e., $\text{Act}_{\mathcal{TL}} = \text{Act}_{u\mathcal{TL}}$). Location invariants $\text{Inv}_{\mathcal{TL}}$ for $\mathcal{A}_{\mathcal{TL}}$ are defined as follows. For each $l = [lb, ub] \in \mathcal{TL}$ we define $\text{Inv}_{\mathcal{TL}}(l) := c_p \leq ub$. For the goal location l_{goal} the invariant $\text{Inv}_{\mathcal{TL}}(l_{goal})$ is identically true, modeling the fact that once plan is completed we can stay there as long as we like. The set $E_{\mathcal{TL}}$ of transitions of $\mathcal{A}_{\mathcal{TL}}$ consists of *intermediate* and *final* transitions. An intermediate transitions has the form $l \overset{g,v!,Y}{\rightarrow} l'$, where: $g = c_p \geq lb$, $Y = \emptyset$ with l and l' consecutive time intervals in $\mathcal{TL}$. A final transition has the form $q \overset{\emptyset,\emptyset,\emptyset}{\rightarrow} q'$, where: $q = l_{pl}$ (pl is the plan length), $q' = l_{goal}$. Note how using state variable values as transitions label we implement the synchronization between state variables and planned timelines.

Modeling Synchronizations with an Observer TGA. We model synchronization between SV and $Plan$ with an *Observer*, that is a TGA reporting an error when an illegal transition occurs.

The observer TGA $\mathcal{A}_{Obs} = (Q_{Obs}, q_0, \text{Act}_{Obs}, X_{Obs}, \text{Inv}_{Obs}, E_{Obs})$ is defined as follows.

The set of locations is $Q_{Obs} = \{l_{ok}, l_{err}\}$ modeling *legal* (l_{ok}) and *illegal* (l_{err}) executions. The initial location q_0 is l_{ok}. The set of actions is $\text{Act}_{Obs} = \{a_{fail}\}$. The set of clocks is $X_{Obs} = \{c_p\}$. There are no invariants, that is $\text{Inv}_{Obs}(l)$ returns always the empty constraint. This models the fact that $\mathcal{A}_{Obs}$ can stay in any location as long as it likes. The set E_{Obs} consists of two kind of uncontrollable transitions: *value transitions* and *sync transitions*. Let $s_p \in \mathcal{TL}$ be a plan step and $v_p \in \mathcal{SV}$ its associated planned value. A value transition has the form $l_{ok} \overset{g,a_{fail},\emptyset}{\rightarrow} l_{err}$, where: $g = \mathcal{TL}_{s_p} \wedge \neg SV_{v_p}$. Let $\langle \mathcal{TL}, v \rangle \longrightarrow \langle \{\mathcal{TL}'_1, \ldots, \mathcal{TL}'_n\}, \{v'_1, \ldots, v'_n\}, \mathcal{R} \rangle$ be a synchronization. A sync transition has the form $l_{ok} \overset{g,a_{fail},\emptyset}{\rightarrow} l_{err}$, where: $g = \neg\mathcal{R}(\mathcal{TL}_v, \mathcal{TL}'_{1v'_1}, \ldots, \mathcal{TL}'_{nv'_n})$. Note how, for each possible cause of error (illegal value occurrence or synchronization violation), a suitable transition is defined, forcing our Observer TGA to move to the error location which, once reached, cannot be left.

The nTGA $\mathcal{PL}$ composed by the set of automata $PL = SV \cup Plan \cup \{\mathcal{A}_{Obs}\}$ models Flexible plan, State Variables and Domain Theory descriptions.

Time flexible plan verification

Given the nTGA $\mathcal{PL}$ defined above, we can define a Reachability Game that ensures, once successfully solved, the plan validity with respect to all the domain constraints and dynamic controllability.

nTGA and Flexible Plans

In (Cesta et al. 2009b), we demonstrated by construction that we obtain a one-to-one mapping between flexible behaviors, defined by $\mathcal{P}$, and automata behaviors, defined by $\mathcal{PL}$, with the Observer automaton holding the error location if either an illegal value occurs or a synchronization is violated. More specifically, it is possible to show that the set of automata $Plan = \{\mathcal{A}_{\mathcal{TL}_1}, ..., \mathcal{A}_{\mathcal{TL}_n}\}$ captures all and only the possible evolutions enabled by the flexible plan $\mathcal{P}$, that is: each automaton $\mathcal{A}_{\mathcal{TL}_i}$ describes the sequence of values for the $\mathcal{TL}_i$ timeline within the planning horizon $\mathcal{H}$; by construction, each automata in $SV = \{\mathcal{A}_{SV_1}, ..., \mathcal{A}_{SV_n}\}$ represent the associated state variable in one-to-one correspondence; finally, the Observer automaton checks for both values consistency (between planned timelines and state variables) and synchronizations satisfaction.

Plan Verification in UPPAAL-TIGA

Once we have represented flexible plans as nTGA, the plan verification problem can be reduced to a Reachability Game.

For this purpose, we introduce a Reachability Game $RG(\mathcal{PL}$, *Init*, *Safe*, *Goal*) where *Init* represents the set of initial locations, one for each automaton in $\mathcal{PL}$, *Safe* = $\{l_{ok}\}$, and *Goal* is for the set of goal locations, one for each $\mathcal{TL}_i$ in $\mathcal{PL}$.

In order to solve $RG(\mathcal{PL}$, *Init*, *Safe*, *Goal*), we use UPPAAL-TIGA (Behrmann et al. 2007). This tool extends UPPAAL (Larsen, Pettersson, and Yi 1997) providing a toolbox for the specification, simulation, and verification of real-time games. If there is no winning strategy, UPPAAL-TIGA gives a counter strategy for the opponent (environment) to make the controller lose. Given a nTGA, a set of goal states (*win*) and/or a set of bad states (*lose*), four types of winning conditions can be issued (Behrmann et al. 2007). Then, to solve the reachability game, we ask UPPAAL-TIGA to check the formula $\Phi =$ *A [Safe U Goal]* in $\mathcal{PL}$. In fact, this formula means that along all the possible paths, $\mathcal{PL}$ remains in *Safe* states until *Goal* states are reached. Thus, if the solver can verify the above property, then the flexible temporal plan is valid (again, see (Cesta et al. 2009b) for a formal account).

Whenever the flexible plan is not verified, UPPAAL-TIGA produces an execution strategy showing one temporal evolution that leads to a fault. Such a strategy can be exploited in order to understand whether the plan has some weakness or flaws are present in the planning model. In (Cesta et al. 2010b), the authors address this issue in a more general way.

Dynamic Controllability

If there exists a winning strategy for the Reachability Game RG, then the plan is also dynamically controllable. Indeed,

recalling the *dynamic controllability* definition for timelines introduced in the second section, we can notice that each possible evolution of the uncontrollable automata corresponds to a timeline projection p. Each strategy/solution for the RG corresponds to a consistent schedule T and a set of strategy represents a viable execution strategy S. Thus, the winning strategies produced by UPPAAL-TIGA represents a viable execution strategy S for the flexible plan $\mathcal{P}$. Furthermore, the use of forward algorithms (Behrmann et al. 2007) guarantees that S is such that $S_{p1}\{\prec x\} = S_{p2}\{\prec x\} \Rightarrow S_{p1}(x) = S_{p2}(x)$, for each controllable timepoint x and projections $p1$ and $p2$. That is, the flexible plan is dynamically controllable.

A new benchmark domain

An aspect worth being addressed is the following: does the method have any practical relevance? In this respect, we have investigated the possibility of tailoring our method in order to implement a realistic benchmark, collect a set of experimental results and show its actual feasibility.

In this section, we present a case study that we use in our experimental analysis. The domain is inspired by a Space Mission Long Term Planning problem as described in (Cesta et al. 2008; 2010a).

We consider a remote space agent (RSA) that operates around a target planet. The RSA can either point to the planet and use its instruments to produce scientific data or point towards a communication station (Relay Satellite or Earth) and communicate previously produced data. The RSA is controlled by a planner and an executive system to accomplish the required tasks (scientific observations, communication, and maintenance activities). For each orbit followed by the RSA around the planet, the operations are split with respect to 3 orbital phases: (1) the pericentre (the orbital segment closest to the target planet); (2) the apocentre (the orbital segment farthest from the planet); (3) the orbital segments between the pericentre and apocentre. Around pericentre, the agent should point toward the planet, thus allowing observations of the planet surface (Science operations). Between pericentre and apocentre passages, the agent should point to Earth for transmitting data. Communication with Earth should occur within a ground-station availability window. Ground-station visibility can either partially overlap or fully contain a pericentre passage. Maintenance operations should occur around the apocentre passages. The RSA is also endowed with a set of scientific instruments or payloads (e.g., stereo cameras, altimeters, spectrometers, etc.) whose activities are to be planned for during the pericentre phase taking into account physical constraints. In particular here we are assuming that instruments can be activated one at a time by following a fixed execution sequence of operations: warm-up, process, turn-off. Additionally, there are other constraints to be satisfied. Constraints on uplink windows frequency and duration require four hours uplink time for each 24 hours, and these uplink windows must be as regular as possible, one every about 20 hours. Apocentre slots for spacecraft maintenance windows must be allocated between 2 and 5 orbits apart, and the maintenance duration is of 90 minutes.

Timeline-based Specification. To obtain a timeline-based specification of the domain we use: *Planned State Variables* representing the timelines where there are activities under the agent control (they are planned for by the agent); *External State Variables*, representing values imposed over time which can only be observed

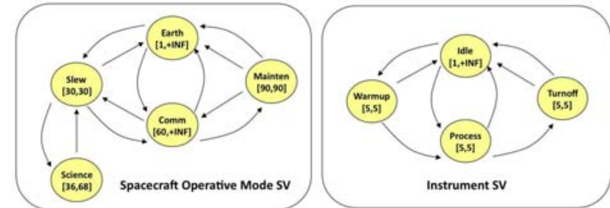


Figure 1: Value transitions for the *planned state variables* describing the Spacecraft Operative Mode (left) and any of the Instruments (right) correct behavior.

Planned State Variables. A state variable *Spacecraft Operative Mode* specifies the observation, communication, and maintenance opportunities for the agent. In Figure 1-left, we detail the values that can be taken by this state variable, their durations, and the allowed value transitions. Additional planned state variables, called *Instrument-1*..., *Instrument-n*, are introduced to represent the scientific payloads. For each variable *Instrument-i* we introduce four values: *Warmup*, *Process*, *Turnoff*, and *Idle* (see Figure 1-right).

External State Variables. The *Orbit Events* state variable (Figure 2, top) maintains the temporal occurrences of pericentres and apocentres represented by the values: *PERI* and *APO* (they have fixed durations). The *Ground Station Availability* state variables (Figure 2, bottom) are a family of variables that maintain the visibility of various ground stations. The allowed values for these state variables are either *Available* or *Unavailable*.

Synchronizations constraints. Any valid temporal plan needs synchronizations among the planned timelines (see Figure 2, middle) and the external timelines (represented as dotted arrows in Figure 2). They represent how (a) science operations must occur during pericentres, i.e., the *Science* value must start and end during a *Peri* value; (b) maintenance operations must occur in the same time interval as apocentres, i.e., the *Maint* value is required to start and end exactly when the *Apo* value starts and ends; (c) communications must occur during ground station visibility windows, i.e., the *Comm* value must start and end during an *Available* value on any of the ground stations. As for scientific instruments, we introduce the following constraints: (d) if *Instrument-i* is not in *Idle* then the other instruments need to be in *Idle*; (e) the *Warmup* is before *Process* which is before *Turnoff*; (f) these activities are allowed only when *Science* is active along the *Operative Mode* timeline.

Relaxed constraints. Besides synchronization constraints, we need to take into account other constraints which cannot be naturally represented in the planning model as structural constraints, but rather treated as meta-level requirements to be enforced by the planner heuristics and optimization methods. In our case study, we consider the following relaxed

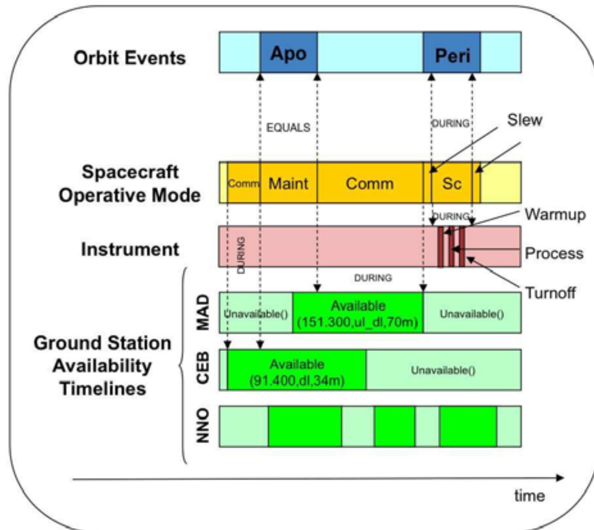


Figure 2: An example of complete plan for the Remote Space Agent domain. The synchronizations among timelines are highlighted.

constraints: (g) *Maint* must be allocated between 2 and 5 orbits apart with duration of about 90 minutes (to be centered around the apocentre event); (h) science activities must be maximized, i.e., during each pericentre phase a *Science* event should occur.

Experimental evaluation

In this work, we demonstrate the feasibility of our approach. In particular, in this section, we analyze the plan verification performances with respect to temporal *flexibility* and execution *controllability*. In particular, we deploy our verifier in different scenarios and execution contexts checking for dynamic controllability and relaxed constraints satisfaction.

More specifically, we analyze the performances of our tool varying the following settings: *State variables.* Here, we consider three possible configurations: the RSA endowed with zero, one, or two scientific instruments. This affects the number of state variables (and synchronization constraints). *Flexibility.* For each scientific instrument activity (i.e., warm-up, process, turn-off), we set a minimal duration (i.e. about 2 minutes), but we allow temporal flexibility on the activity termination, namely, the end of each activity has a tolerance ranging from 5 to 10 seconds. E.g. if we set 5 seconds of flexibility, we introduce an uncertainty on the activity terminations, for instance, the warm-up activity can take from 120 to 125 seconds. This temporal interval represents the degree of temporal flexibility that we introduce in the system. *Horizon.* We consider flexible plans with a horizon length ranging from 3 to 10 mission days. *Controllability.* We consider four different execution contexts: 1) all the instruments activities are controllable; 2) for each instrument the warm-up termination is not controllable; 3) for each instrument, warm-up and process terminations are not controllable; 4) for each instrument warm-up, process, and turn-off are not controllable.

Note that the higher is the degree of flexibility/uncontrollability, the larger is the space of allowed behaviors to be checked, thus, the harder is flexible plan verification.

In these settings, we analyze the performance of our tool considering the following issues: model generation, dynamic controllability checking, domain requirements checking. We run our experiments on a Linux workstation endowed with a 64-bit AMD Athlon CPU (3.5GHz) and 2GB RAM. In the following we illustrate the collected empirical results.

Model Generation. A first, preliminary, analysis concerns the model generation process and the dimension of the generated UPPAAL-TIGA specification. This analysis is needed because the complexity of the generated UPPAAL-TIGA models can affect the scalability of the overall verification method. In fact, for this purpose, we developed a tool that implements the nTGA modeling procedure described before (see Section "'Using nTga to model timeline-based planning specifications"') and automatically builds the UPPAAL-TIGA model given the description of the *planning domain* and the *flexible temporal plan* to be checked. Here, we want to assess the size of the generated model and the generation time with respect to the dimension of the planning domain and of the plan (state variables and plan length). In our experimental setting, we consider domain models with an incremental number of state variables (from 3 to 5) and plans with an incremental number of mission days (from 3 to 10). For each possible configuration, we consider the dimension of the generated model and the time elapsed for the generation. For all these configurations, the generation process is very fast and takes less than $200ms$, while the dimension of the generated model gradually grows with respect to the dimension of the flexible plan (in terms of number of timelines and plan length).

	3 timelines		4 timelines		5 timelines	
days	kb	nr. states	kb	nr. states	kb	nr. states
3	16	41	19	51	23	61
4	32	85	38	110	42	135
5	54	131	58	179	63	227
6	73	168	77	240	82	312
7	94	204	98	300	101	396
8	107	238	112	351	117	464
9	119	271	125	397	130	523
10	139	301	142	439	147	577

Figure 3: Size of the generated model (kb and number of states) with respect to the plan length and number of timelines.

In conclusion, the process of model generation is fast and the generated model grows linearly with the dimension of the plan, therefore, here the encoding phase is not a critical step.

Flexible Plan Verification against Fully Controllable Execution. Here, we collect the time performances (CPU time) of plan verification in different scenarios (changing the degree of plan flexibility) and execution contexts (changing the plan controllability).

Here, we analyze the plan verification performances in

checking dynamic controllability in the easiest condition of controllability. Indeed, in this initial experimental setting, we consider fully controllable plans assuming all the scientific tasks to be controllable.

Full Controllability			
days	0s flex	5s flex	10s flex
3	0,198	0,202	0,254
4	0,254	0,301	0,320
5	0,300	0,344	0,328
6	0,192	0,208	0,184
7	0,248	0,240	0,248
8	0,292	0,300	0,284
9	0,348	0,332	0,364
10	0,392	0,364	0,401

(a)

1 Uncontrollable Task			
days	0s flex	5s flex	10s flex
3	0,189	0,165	0,193
4	0,227	0,234	0,238
5	0,276	0,296	0,264
6	0,172	0,160	0,168
7	0,212	0,220	0,208
8	0,268	0,248	0,252
9	0,308	0,336	0,336
10	0,356	0,364	0,379

(b)

2 Uncontrollable Tasks			
days	0s flex	5s flex	10s flex
3	0,189	0,192	0,188
4	0,246	0,237	0,245
5	0,296	0,324	0,288
6	0,156	0,164	0,164
7	0,212	0,216	0,212
8	0,260	0,263	0,264
9	0,316	0,288	0,336
10	0,345	0,321	0,335

(c)

3 Uncontrollable Tasks			
days	0s flex	5s flex	10s flex
3	0,198	0,221	0,212
4	0,267	0,283	0,267
5	0,304	0,288	0,288
6	0,188	0,172	0,176
7	0,212	0,208	0,220
8	0,252	0,236	0,248
9	0,312	0,300	0,332
10	0,367	0,353	0,379

(d)

Figure 4: Verification with one additional instrument varying flexibility and controllability.

In Figure 4(a) and Figure 5(a), we illustrate the results gathered in the case of one and two instruments, respectively, considering the verifier performances under different plan length and flexibility conditions. The results in Figure 4(a) and Figure 5(a) show that an increment of temporal flexibility has a limited impact on the performances of the verification tool. This is particularly evident in the case of a single instrument, where the performances of the verification process seems not affected by the degree of temporal flexibility (Figures 4(a)). On the other hand, in the case of 2 scientific instruments (Figures 4(b)), we can observe a smooth growth of the verification time with respect to the allowed temporal flexibility. Of course, this is mainly due to the fact that in this case the verification process is to check all the synchronization constraints among the instruments, which are not considered in the case of a single instrument. However, even thought the increment of temporal flexibility enlarges the number of possible behaviors to be checked, in the presence of fully controllable activities a single execution trace is sufficient to show plan controllability, hence the verification task is reduced to correct plan termination checking.

Flexible Plan Verification against Partially Controllable Execution. In the following, we consider the verifier performances in checking *dynamic controllability* in the presence of uncontrollable activities. Interestingly, also in this setting the execution time for verification grows in a gradual manner. In the case of a single scientific instrument, the gathered results (see Figures 4b-c-d) are comparable with the ones collected in the fully controllable case. Even when we consider a setting where all the tasks are uncontrollable, our verification tool can easily accomplish plan verification for all the flexibility and plan length configurations (see Figure 4(d)). In the case of 2 instruments (hence, 5 timelines), the increment of flexibility gradually increments the time

Full Controllability			
days	0s flex	5s flex	10s flex
3	0,899	2,010	2,673
4	1,123	3,101	3,200
5	1,664	3,508	3,312
6	2,756	3,780	3,396
7	3,704	4,368	4,528
8	4,492	5,080	5,088
9	5,300	5,896	6,724
10	5,934	6,234	7,243

(a)

1 Uncontrollable Task			
days	0s flex	5s flex	10s flex
3	1,784	2,998	3,021
4	2,132	3,156	3,103
5	2,784	3,280	3,248
6	2,892	3,252	3,312
7	3,664	4,384	4,500
8	4,232	5,096	5,212
9	5,492	6,492	6,716
10	6,357	7,093	7,732

(b)

2 Uncontrollable Tasks			
days	0s flex	5s flex	10s flex
3	2,022	3,105	3,227
4	2,214	3,326	3,339
5	2,444	3,452	3,548
6	2,652	3,212	3,328
7	3,612	4,412	4,464
8	4,200	4,879	5,208
9	5,300	5,876	6,812
10	6,604	7,012	8,002

(c)

3 Uncontrollable Tasks			
days	0s flex	5s flex	10s flex
3	2,243	3,143	3,004
4	2,527	3,340	3,122
5	2,880	3,528	3,052
6	2,628	3,404	3,704
7	3,604	4,252	4,284
8	4,212	4,668	4,98
9	5,176	6,088	6,384
10	6,392	7,478	8,244

(d)

Figure 5: Verification with two instruments changing both flexibility and controllability.

needed by the verification tool to verify the plans (see Figures 5b-c-d). A similar increment can be observed when we increase the number of uncontrollable activities. If we keep constant the uncontrollable activities, the performances trend appears similar to the one of the fully controllable case. Nevertheless, even if we consider the worst case, i.e. all the activities uncontrollable and maximal temporal flexibility, the performances of the UPPAAL-TIGA verification tool are still very satisfactory: given flexible plans with horizon length up to 10 mission days and 5 timelines, plan verification can be successfully accomplished within few seconds (see Figure 5(d)).

Flexible Plan Verification against Relaxed Domain Constraints. We also perform tests to verify also other domain-dependent constraints, namely, the two *relaxed constraints* on maintenance and science activities introduced in the previous section. In this experimental setting, we assume the system endowed with 2 scientific instruments (5 timelines). In Figure 6, we report the experimental results collected increasing the degree of uncontrollability on the considered flexible plans.
Changing the plan flexibility, the verifier presents performances that are analogous to the ones reported in the previous case. Thus, the additional properties to be checked provide a low additional overhead to the verification process.

Conclusion

In our path to enhancing a knowledge engineering environment for timeline-based problem solving, we are investigating the integration of formal methods as a way of orthogonal contribution to analyze properties of plans. In recent work including the current one we are proposing the combined use of timeline-based planning and standard techniques for formal validation and verification. In particular, we have synthesized a verification process suitable for a timeline-based planner showing how a temporally flexible plan verification problem can be cast as model-checking on timed game au-

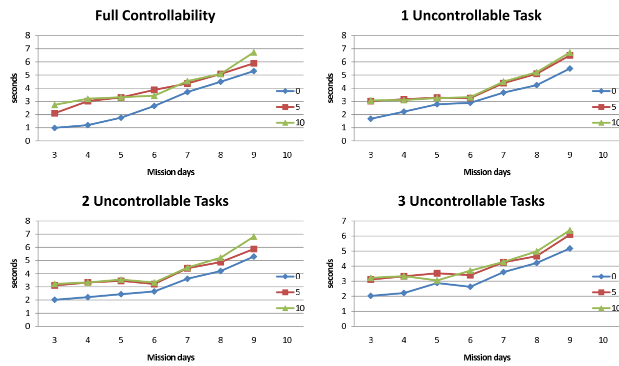


Figure 6: Experimental results collected validating flexible plans varying flexibility and controllability in case study with two additional instruments.

tomata. Then, we have investigated the possibility of tailoring our method in order to implement a realistic benchmark, collect a set of experimental results and show its actual feasibility. The experimental results presented in this paper demonstrate the feasibility of our method and the effectiveness of UPPAAL-TIGA in this setting. In fact, despite the increasing complexity of the verification configurations, the execution time gradually grows with the complexity of the task. Furthermore, the concurrent increase of temporal flexibility and plan uncontrollability does not determine the expected computational overhead. The UPPAAL-TIGA verifier can effectively handle the flexible plan verification task in all the considered configurations.

Acknowledgments. Cesta, Fratini, Orlandini and Tronci are partially supported by the EU project ULISSE (Call "SPA.2007.2.1.01 Space Science" Contract FP7.218815). Cesta and Fratini has been also partially supported by European Space Agency (ESA) within the Advanced Planning and Scheduling Initiative (APSI).

References

Abdedaim, Y.; Asarin, E.; Gallien, M.; Ingrand, F.; Lesire, C.; and Sighireanu, M. 2007. Planning Robust Temporal Plans: A Comparison Between CBTP and TGA Approaches. In *Proc. of the 7th International Conference on Automated Planning and Scheduling*, 2–10.

Behrmann, G.; Cougnard, A.; David, A.; Fleury, E.; Larsen, K.; and Lime, D. 2007. UPPAAL-TIGA: Time for playing games! In *Proc. of CAV-07*, number 4590 in LNCS, 121–125. Springer.

Cassez, F.; David, A.; Fleury, E.; Larsen, K. G.; and Lime, D. 2005. Efficient on-the-fly algorithms for the analysis of timed games. In *CONCUR 2005*, 66–80. Springer-Verlag.

Cesta, A.; Fratini, S.; Oddi, A.; and Pecora, F. 2008. APSI Case#1: Pre-planning Science Operations in MARS EXPRESS. In *i-SAIRAS-08. Proceedings of the 9th Int. Symp. on Artificial Intelligence, Robotics and Automation in Space*. JPL, Pasadena, CA.

Cesta, A.; Finzi, A.; Fratini, S.; Orlandini, A.; and Tronci, E. 2009a. Flexible Timeline-Based Plan Verification. In *KI 2009*, volume 5803 of *LNAI*.

Cesta, A.; Finzi, A.; Fratini, S.; Orlandini, A.; and Tronci, E. 2009b. Verifying flexible timeline-based plans. In *VVPS-09. Workshop on Verification and Validation of Planning and Scheduling Systems at ICAPS, Thessaloniki, Greece.*

Cesta, A.; Cortellessa, G.; Fratini, S.; and Oddi, A. 2010a. MRSPOCK: Steps in Developing an End-to-End Space Application. *Computational Intelligence*. Accepted for publication.

Cesta, A.; Finzi, A.; Fratini, S.; Orlandini, A.; and Tronci, E. 2010b. Validation and Verification Issues in a Timeline-Based Planning System. *Knowledge Engineering Review*. Accepted for publication.

EUROPA. 2008. Europa Software Distribution Web Site. https://babelfish.arc.nasa.gov/trac/europa/.

Larsen, K. G.; Pettersson, P.; and Yi, W. 1997. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer* 1(1-2):134–152.

Maler, O.; Pnueli, A.; and Sifakis, J. 1995. On the Synthesis of Discrete Controllers for Timed Systems. In *STACS*, LNCS, 229–242. Springer.

Morris, P. H., and Muscettola, N. 2005. Temporal Dynamic Controllability Revisited. In *Proc. of AAAI 2005*, 1193–1198.

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

Sherwood, R.; Engelhardt, B.; Rabideau, G.; Chien, S.; and Knight, R. 2000. ASPEN, Automatic Scheduling and Planning Environment. Technical Report D-15482, JPL.

Tate, A.; Drabble, B.; and Kirby, R. 1994. O-Plan2: An Open Architecture for Command, Planning, and Control. In Zweben, M., and Fox, S. M., eds., *Intelligent Scheduling*. Morgan Kaufmann.

Vidal, T., and Fargier, H. 1999. Handling Contingency in Temporal Constraint Networks: From Consistency To Controllabilities. *JETAI* 11(1):23–45.

Ontology Oriented Exploration of an HTN Planning Domain through Hypotheses and Diagnostic Execution

Li Jin and Keith S. Decker
University of Delaware
Department of Computer and Information Sciences
Newark, DE 19716, USA
{jin, decker@cis.udel.edu}

Abstract

We present a framework, HTN-Explorer, for case-based exploration of hierarchical task network planning domain oriented by ontological background knowledge. With an existing simple, incomplete model as a seeding model extracted from limited examples of plan solutions, HTN-Explorer explores more comprehensive planning domain knowledge by expanding the seeding model through a discovery circle of hypothesis generation, evaluation and diagnostic execution. HTN-Explorer proposes hypothetical task methods by adapting an existing model for those situations not covered by the original model. As well, hypothetical models are evaluated with heuristics that estimate the plausibility and discoveries of hypotheses. The executions of hypothetical plans based on hypothetical models provide information to diagnosis. This framework provides some desirable functionalities: (1) it automatically explores HTN models by integrating various strategies; (2) it proposes hypotheses for experimental testing based upon their evaluated plausibility and discoveries; (3) it facilitates encoding of background knowledge into the exploration processes. We use a variation of the UM Translog domain to evaluate our approach.

Introduction

Hierarchical task networks (HTNs) are an important, frequently studied approach to solve problems in AI planning research and have recently achieved several notable successes (Nau 2007). A *hierarchical task network* (HTN) planner solves a problem by following task decomposition descriptions to recursively decompose a complex task into simpler tasks until the tasks can be accomplished by actions directly. HTN planning was first presented in the mid-1970s (Sacerdoti 1975), and its formalisms and properties were well studied in the mid-1990s (Erol, Nau, and Hendler 1994). Over the past decade, many planning systems based on HTN decomposition (e.g. SIPE (Wilkins 1985), O-PLAN (Currie and Tate 1991), and SHOP (Nau et al. 2005)) have made successful achievements in the practical applications, such as the Mars Rovers (Estlin et al. 2003) and Bridge Baron (Smith, Nau, and Throop 1998).

Despite the achievements of HTN planning, there still exists a significant challenge, i.e. the difficulty of acquiring complete domain knowledge required by a planner to solve a problem. In many application domains, it is difficult or impossible to acquire accurate and complete HTN models hard-coded by human experts due to multiple reasons, such as the lack of necessary experiences or knowledge of the domains, the complexities in the domains, and time or effort consumption. Consequently, researchers have recently shown strong interest in developing algorithms or systems to (semi)automatically learn planning theories from solution cases/examples or from planning and execution experiences (Zimmerman and Kambhampati 2003). However, most of the previous research that learns a planning domain theory from examples and experiences are passive, namely, it depends on examples or previous planning experiences but cannot actively explore new models not represented by or not reasoned from example cases or experiences.

In this paper, we present HTN-Explorer, a general framework that aims at facilitating exploration of an HTN planning domain with minimum human intervention. Our work is motivated by many practical domains in which generally, background knowledge is available, but domain specific task decomposition or action descriptions may be only partially provided by a human or learned by limited example cases. In such a domain, there might be limited plan examples or solution experiences available that do not cover all situations in the real world; thus, only example-driven learning techniques (e.g. case-based learning) might not work very well. We present an approach to learn comprehensive models through hypothesis generation and diagnostic execution.

With an assumption that background knowledge can be represented in ontology, we develop HTN-Explorer as a self-directed automated system that utilizes an ontology to expand an incomplete model by presenting hypotheses with multiple strategies. The plausibility of a hypothesis is evaluated based on the assessment of the strength of its proposing strategy and the underlying computational method. The novelty of a hypothesis is estimated by new preconditions, new kinds of tasks, new constrains or new solutions that the hypothesis may cover or provide. HTN-Explorer is capable of presenting those hypotheses with high evaluations of plausibility and novelty to a human or an experimental system (e.g. a lab robot) for testing. The feedback information will be used to update the original domain theory and will be utilized to generate new hypothetical models.

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

This paper continues with brief introduction of HTN planning formalism. Next, we overview the architecture of HTN-Explorer. After presenting the hypothesis generation strategies implemented in HTN-Explorer, we will then explain the hypothesis evaluation heuristic function and the diagnosis procedures. Then, we demonstrate an empirical case study of exploring a variation of the UM Translog domain (Andrews et al. 1995). Finally, we discuss related works and make conclusions.

Preliminaries

In this paper, we follow the principles of the task decomposition formalism of HTN planning defined in Chapter 11 by Ghallab et al. (Ghallab, Nau, and Traverso 2004). HTN planning uses task decomposition description methods to decompose non-primitive (also called compound) tasks into simpler subtasks. Planning continues the decomposition process until primitive tasks are reached. A primitive task can be accomplished by an action. A plan solution is composed of a sequence of actions that can achieve the high-level tasks of a problem.

Generally, an HTN planning theory is composed of two sets of descriptions, one is a set of operators, the other is a set of methods. An operator describes how a state is changed if the operator is applied to the state when required preconditions are satisfied. A method describes how a compound task is decomposed into subtasks.

An operator is represented by a 4-tuple $O = (name(O), PreC, Add, Del)$, where $name(O)$ is the name of the operator, $PreC$ is a set of preconditions, and Add and Del are the adding list and deleting list that define how to modify the current state s (consisting of a collection of ground atoms) when O is applied to s by adding or deleting atoms in the lists to or from s. A method is formalized as a triple $M = \{T, PreC, SubTs\}$ specifying that T, a non-primitive task, can be achieved by the subtasks $SubT$ when the elements in the precondition set $PreC$ are satisfied. A task is of the form $T(r_1, \ldots, r_n)$, where T is a task symbol, i.e. the name of the task, and $r_1, \ldots, r_k$ are terms.

An HTN planning problem is a 4-tuple $P = (s_0, Ts, Os, Ms)$, where s_0 is the initial state, Ts is the initial sequence of tasks, and Os and Ms are sets of planning operators and methods respectively. A solution for P is a plan consisting of a sequence of actions that can achieve Ts from the initial state s_0. As a ground instance of a planning operator, an action is of the form $a = (name(a), preconditions(a), effects(a))$. An action can accomplish a ground primitive task t in a state s if a is applicable to s when the preconditions of the action are satisfied in s, and a is able to produce the effects (including adding list and deleting list) that can succeed in achieving the goals requested by the task.

We assume that an ontology describing background knowledge of a domain is available. We use predicate to represent the relationships of variables and constants in an ontology, such as *(class ?x C)* indicating that a variable *?x* is of a class type *C* and *(isa C C')* indicating that *C* is a subclass of *C'*. For instance, Figure 1 shows the ontology of the UM Translog domain that is simplified from its original version

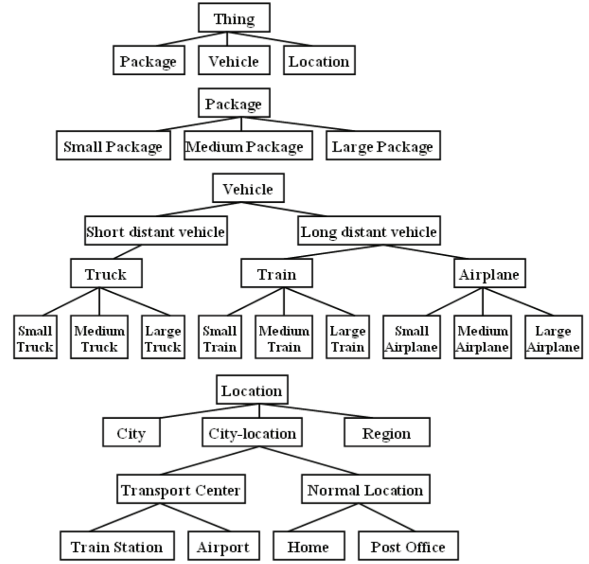


Figure 1: Ontology of the UM Translog domain.

(Andrews et al. 1995) and will be used in this paper to explain our approach. In Figure 1, the relationship that *Small Truck* is a subclass of *Truck* is represented by *(isa Small-Truck Truck)*. And the relationship of class *Small Truck* and its instance *st1* can be defined as *(class st1 Small-Truck)*. We also suppose that a simple incomplete model can be easily hard coded by hand or extracted from limited solution examples. Then the simple model works as a seeding one from which a more comprehensive model can be automatically generated by using some strategies with the aid of ontology knowledge.

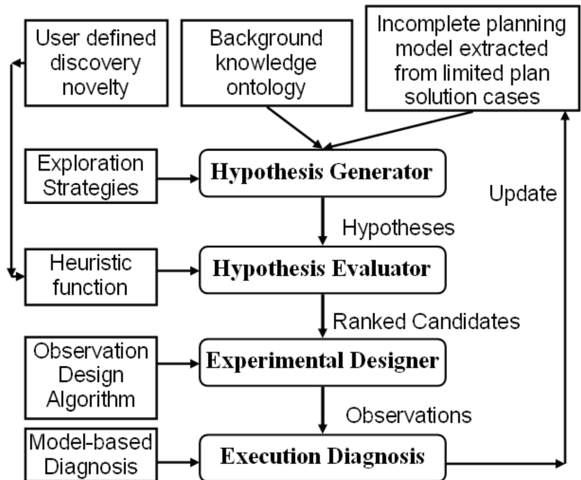


Figure 2: Information flow of HTN-Explorer.

Overview of HTN-Explorer

Figure 2 shows the top-level control of the exploration process in the HTN-Explorer. The process repeats the cycle of hypothesis generation, evaluation, experiment and diagnosis until no hypothesis has a plausible value above a stop criterion.

The HTN-Explorer's input consists of: (1) An initial set of cases of planning problems and their HTN solutions that are used to obtain the initial model to be expanded. The set may be limited; thus, the initial model may be incomplete. (2) Domain-specific background knowledge base stored in an ontology. (3) Domain-specific novelties of discoveries that are defined by a user. The output of HTN-Explorer is an enhanced planning theory that is more comprehensive and more accurate than the original one.

HTN-Explorer consists of the following main components:

- A hypothesis generator that proposes plausible hypothetical HTN methods or operators. It encodes multiple general HTN planning model exploration strategies to propose hypotheses based on the background knowledge organized in an ontology until reaching stopping values of the strategies.
- A hypothesis evaluator that estimates the plausibility and novelty of a hypothesis through a heuristic function. It selects candidates that are worth of the expense of experimental tests. It provides a flexible framework for the usage of specific background knowledge and user defined discoveries.
- An experiment designer that can design observations to be made for a robot or a system or a human that can execute experiments to test hypotheses, make observations, and maintain the records.
- An experimental diagnosis component that interprets the observed results and explains any divergence of the observations from the expected results predicted by the tests. The planning knowledge base is updated by the diagnoses.

Hypotheses Generation

With a set of HTN plan solution cases as input, HTN-Explorer first generates a simple model by abstracting an object with a variable of the deepest class in the ontology that the object belongs to. For the example shown in Figure 3, a method in the right side is generated from the case in the left side.

Then HTN-Explorer uses the following strategies to expand a simple model.

Analogical Expansion

One kind of hypotheses can be generated by modifying a method through replacing an object variable with a similar one in the object's ontology leaf nodes. The similarity between two classes in one ontology is estimated quantitatively as the division of the number of their common ancestors by the sum of the numbers of their individual distinguish ancestors as shown in Equation (1). Hypotheses are created by this method when the similarity between the substituting and original classes is higher than a prefixed value.

Task: deliver pac1 from loc1 to loc2	Task: deliver ?p from ?l1 to ?l2
Preconditions:	Preconditions:
pac1 class: Small Package	?p class: Small Package
loc1 class: Home	?l1 class: Normal Location
loc2 class: Post Office	?l2 class: Normal Location
loc1 in city1	differ ?l1 ?l2
loc2 in city1	?l1 in ?c1
city1 class: City	?l2 in ?c2
truck1 class: Small Truck	?c1 class: City
truck1 at loc1	?c2 class: City
Subtask:	same ?c1 ?c2
load pac1 truck1	?v class: Small Truck
drive truck1 loc1 loc2	?v at ?l1
unload pac1 truck1	Subtask: load ?p ?v
	drive ?v ?l1 ?l2
	unload ?p ?v

Figure 3: Generating simple model from plan cases.

$$Sim(C_1, C_2) = \frac{|Anc(C_1) \bigcap Anc(C_2)|}{|Anc(C_1) \bigcup Anc(C_2)|} \qquad (1)$$

where $Anc(C)$ is a set of all ancestors of C.

For example, based on the ontology shown in Figure 1, *Sim(Small Truck, Medium Truck)*=1.0 and *Sim(Small Truck, Small Train)*=1/5=0.2; thus, *Medium Truck* is decided more similar to *Small Truck* than *Small Train*. As shown in Figure 4, a hypothetical method can be generated by adapting the method in the right side of Figure 3 with modifying the class of *?v* from *Small Truck* to *Medium Truck*. After the hypothesis and testing processes, the method in the right side of Figure 3 will be adapted with all leaf classes in the ontology shown in Figure 1.

```
Task: deliver ?p from ?l1 to ?l2
Preconditions: ?p class: Small Package
               ?l1 class: Normal Location
               ?l2 class: Normal Location
               differ ?l1 ?l2
               ?l1 in ?c1
               ?l2 in ?c2
               ?c1 class: City
               ?c2 class: City
               same ?c1 ?c2
               ?v class: Medium Truck
               ?v at ?l1
Subtask: load ?p ?v
         drive ?v ?l1 ?l2
         unload ?p ?v
```

Figure 4: Hypothetical method generated by analogical expansion.

However, hypotheses generated by this strategy may be incorrect. For example, if a hypothetical method can be generated by adapting the method in the right side of Figure 3 with modifying the class of *?p* from *Small Package* to *Medium Package*, then a solution generated by this method will fail when it is executed. Therefore, hypotheses need experimental tests to prove it is correct or not. The testing and diagnosis processes will be discussed in later sections.

Negation

The above strategy expands a seeding model by adapting the model to more class types. There exist other kinds of preconditions that are not related to class types, such as *(same ?c1 ?c2)* in Figure 3. The strategy of negation makes a precondition unsatisfied by replacing it with a negative one; then, the old method will fail under the negative precondition. One way to repair the old method is adding a new subtask that can use an already known method if it exists to achieve the original precondition from the negative one. Thus, a new method that combines the old method and the added method for the new subtask will be generated to solve a problem under the negative precondition. For example, in Figure 3, the precondition *(?v at ?l1)* is denied to *(not (?v at ?l1))*, so a new task *(drive ?v ?l1' ?l1)* with *(not (same ?l1 ?l1'))* can be added to achieve *(?v at ?l1)*. Then the newly generated method can be applied to a problem with the preconditions *(?v at ?l1')* and *(not (same ?l1 ?l1'))* instead of *(?v at ?l1)*.

However, for those unsatisfied preconditions that cannot be achieved by a new task, the original solution should be modified by using the already existing methods or proved hypothetical methods. Then hypotheses can be proposed. For example, when *(same ?c1 ?c2)* is negated as *(not (same ?c1 ?c2))* for which multiple possible methods can be proposed: one is that a truck is still used to drive the package from *?l1* to *?l2* if there is route connecting *?l1* and *?l2*; another is that the task is re-decomposed into three tasks: first, deliver *?p* to a train station, then a variable of train type is used to deliver *?p* to another train station in *?c2*, then deliver *?p* to *?l2*.

Diversification

Diversification strategy selects a new precondition semantically different from the original preconditions but not conflicting with any of the original predictions; then combines this new precondition to the originals. Semantical difference is defined as that in an ontology, the concept related to a new precondition and the concepts related to original preconditions do not have any common ancestors. This strategy intends to generate methods for those rare situations that seldom happen in example problems and solutions.

For the example in Figure 1, a new subclass can be added to *Thing* such as *Weather* where *Weather* is considered as semantically different from *Package* and *Location*. Two preconditions can be related to *Weather*, such as *(Weather is good)* and *(Weather is snowing)*. Different methods may be preferred under the two weather preconditions, e.g. using a method employing a train may be preferred under the precondition of snow weather.

Generalization

This strategy generalizes a variable from its class to its parent class in an ontology. This strategy should be applied before all the class's siblings in an ontology have been examined. This strategy only chooses one hypothesis that can cover most of the individual methods that can be applied to the children classes. Exclusive preconditions are added to remove any conflictions from the children classes. This strategy makes the new method more general and provides a possible solution for those problems that are not solved by the original theory.

For example, when the hypothetical methods related to *Medium Package* and *Large Package* have been generated by adapting the method in Figure 3 and have been tested, the method shown in Figure 5 can be generalized from *Small Package*, *Medium Package* and *Large Package* to *Package*. Because *Large Truck* can work for all kinds of packages (*Small*, *Medium* and *Large*), this method is chosen as a general method. If *(?v class: Large Truck)* is changed to *(?v class: Medium Truck)*, then a precondition, *(not (?p class: Large Package))*, should be added to exclude the incorrect solution that *Medium Truck* is used to deliver *Large Package*.

```
Task: deliver ?p from ?l1 to l2
Preconditions:
    ?p class: Package
    ?l1 class: Normal Location
    ?l2 class: Normal Location
    differ ?l1 ?l2
    ?l1 in ?c1
    ?l2 in ?c2
    ?c1 class: City
    ?c2 class: City
    same ?c1 ?c2
    ?v class: Large Truck
    ?v at ?l1
Subtask: load ?p ?v
    drive ?v ?l1 ?l2
    unload ?p ?v
```

Figure 5: Hypothesis generated by generalization.

Ranking Hypotheses by Heuristics

When multiple hypotheses are generated, they should be evaluated to confirm that they are worthy experimental test operated by a robot or a human. In our approach, the hypotheses are ranked with the following considerations:

- assessing corresponding strength of a strategy that is used to propose a hypothesis;
- evaluating the plausibility of a hypothesis; namely, a hypothetical method has a higher likelihood to be chosen to solve a problem or to succeed when it is applied to a practical problem;
- estimating the novel discoveries that might be produced by a hypothesis.

Based on these considerations, we define the following heuristic function to evaluate a hypothesis:

$$Evaluate(h) = w_s * P_h * \sum(I_h) \qquad (2)$$

where, h is a hypothesis, w_s is the weight of the strategy used to propose h, P_h is the plausibility estimated for h based on the underlying computation methods used in the exploration strategies, and I_h represents the estimation of novelties of h's items interesting to a user or to a complete domain theory. The purpose of this function is to balance two factors related to a hypothesis, i.e. the appropriateness of a hypothesis and the interests of domain knowledge discovery or user preference. The appropriateness is encoded in the plausibility of execution success when the hypothesis is applied to a problem or the confidence that the hypothesis is selected from multiple choices to accomplish a task. The discovery of a hypothesis is represented by the summed discovery scores estimated for the items involved in a hypothesis.

For a general HTN domain, the plausibility of a hypothesis can be estimated differently for different exploration strategy together with the background knowledge and the already known planning theory. Our approach makes the estimation by follows:

- For analogical expansion, the computational method as shown in Equation (1) that estimates the similarity of two classes in an ontology can be used to estimate the plausibility of a hypothesis.
- For the strategy of generalization, the plausibility is estimated by the percentage of coverage of the new hypothesis over the methods of the children classes.
- For negation, the confidence of a hypothesis can be evaluated by the success possibilities of the primitive tasks that may estimated from the previous cases.
- Because analogical expansion provides various hypotheses of lower classes that are bases for the strategy of generalization, the strategy of analogical expansion is assigned higher priority.
- For diversification, the plausibility of a hypothesis may be based on background knowledge if related knowledge exists; otherwise, diversification will be assigned the lowest priority.

The general items of discoveries for an HTN planning domain can be categorized into groups with weights indicating preference to complete a domain theory or to satisfy a user's interest. As shown in Figure 6, this kind of estimated weights of discoveries can be defined or modified based on specific domains with the interestingness of specific items domain-dependently defined and estimated.

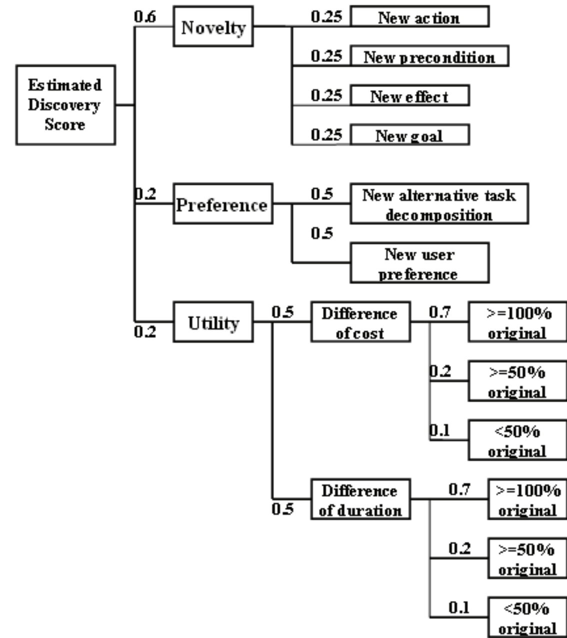


Figure 6: Abstraction of discoveries and their weights.

Plan Execution, Observation and Diagnosis

A hypothetical method generated may be incorrect or incomplete so that a plan solution based on hypothetical models (called a hypothetical plan) may fail when it is executed. Actually, a hypothetical plan provides information about what is expected to happen during an execution. When a prediction is different from what is observed in an execution, it means that an expectation failure happens. Such failures provide critical information for learning and refining a planning theory (Birnbaum et al. 1990; Ram, Narayanan, and Cox 1995), especially for testing and improving a hypothetical model in our approach.

We propose to apply model-based diagnosis technology (Reiter 1987) to diagnose a hypothetical plan execution by observing, comparing and analyzing the differences between predictions and observations of the plan. The incorrectness of hypothetical models can be identified and refined by the diagnosis information.

To test a generated hypothetical HTN task method, the following steps are necessary:

- Design a plan case to test the hypothetical method. A testing plan can be created by instantiating a task method with the instances of that variables. For the example of the method in Figure 4, a hypothetical plan case similar to the case in the left side of Figure 3 can be generated by replacing the variables with instances of the variable classes. For a hypothetical method containing variables of the upper class in an ontology, the variables should be instantiated with each instance of leaf classes; therefore, multiple hypothetical plan cases may be generated to test the model.
- Design observations to monitor during a plan execution. Figure 7 shows how to decide what to be observed for a hypothetical plan case. Here, we assume that a monitor robot or a human is able to decide when an action begins being executed and is able to know when this action is finished. We define that a precondition or an effect that can be monitored is observable.

- Diagnose a plan execution. Based on the difference between observations and predictions, decide if an execution succeeds or fails and give out the reasons of a failure. The diagnosis of one case may be saved in the background knowledge base to guide the future hypothesis generation.

function Observable(hp)
input : an HTN plan $hp = \{T, S_0, A, D\}$, T: task, S_0: initial state, A: action, D: domain knowledge
output : a set of atoms to be observed
$S \longleftarrow \emptyset$;
foreach *action* $a \in A$ **do**
  foreach *precondition* c *of* a **do**
    if c *is observable and* $(c, before(a))$ *not* $\in S$ **then**
      insert $(c, before(a))$ into S;
  endfor
  foreach *effect* e *of* a **do**
    if e *is observable and* $(e, after(a))$ *not* $\in S$ **then**
      insert $(e, after(a))$ into S;
  endfor
endfor
return S;

Figure 7: Designing an observable set of a plan.

Empirical Evaluations

To evaluate our approach, we adopt the variant of the UM Translog domain presented by Xu and Muñoz-Avila (2005) to do experiments for exploring HTN domain by generating hypotheses using the strategies described in the previous section. The domain contains trucks, trains and airplanes to transport packages of various sizes between different sites in different distance ranges (intercity and intracity) as shown in Figure 1. One region contains one or more cities with each city having one or more city-locations that may be transport centers or normal locations that are not transport centers. The transport centers include airports or train stations, while the normal locations serve as the origin or destination of a package. Different kinds of transportation tools are used for deliveries over different distance with different cost. For example, a truck is used for intercity delivery, a train or an airplane is used for intracity transportation. It assumes that any two intercity locations are connected by a truck route, two train stations are connected by a train route, and an airplane route connects two airports.

The initial states are generated with 5 cities each having one airport and one train station, 25 trucks, 20 trains, 20 airplanes and 20 packages. Each of the vehicles is initially located in a random city and randomly categorized into big, medium, and small types. The packages of different types are randomly located at various locations. We randomly generate a set of 150 solvable problems from which 50 problems are randomly selected to consist a seeding set and the left 100 problems become a testing set. For our purpose, JSHOP2 system (Ilghami 2005) is used to simulate solving a problem in the seeding set that requires a domain description, including operators and methods to generate plans. The problems and their corresponding plan solutions (including plans and task decomposition structures) are stored as cases in the seeding set from which HTN-Explorer abstracts a seeding model by replacing an instance object with a variable of its type. The HTN-Explorer will use the seeding model and the ontology to explore a more comprehensive model that will be tested by using JSHOP2 to solve the problems in the testing set.

We conduct evaluation by choosing various numbers of examples from the seeding set. The completeness of a seeding model depends on how much of the domain theory is covered by the selected solution cases. In detail, the experiments are conducted as the following:

1. for N=1 to 50 do steps 2 to 5:
2. Randomly choose N problems with their solutions from the seeding set. Extract an HTN model from each solution and combine the models together to be an initial seeding model.
3. Solve the problems in the testing set using the seeding model.
4. Apply the strategies described in the previous section to the seeding model to generate a more comprehensive model.
5. Use the comprehensive model to solve the problems in the testing set.

Figure 8 shows the results of an experiment in which one circle of hypothesis generation is applied to expand the seeding models that are respectively extracted from the various number of examples in the seeding set. From Figure 8, we can see that the HTN-Explorer can expand an incomplete domain theory efficiently. Figure 9 shows the average number of those strategies that the HTN-Explorer uses to do the experiments as shown in Figure 8. Because the experimental domain is not so complex that the strategy "diversification" is not applied.

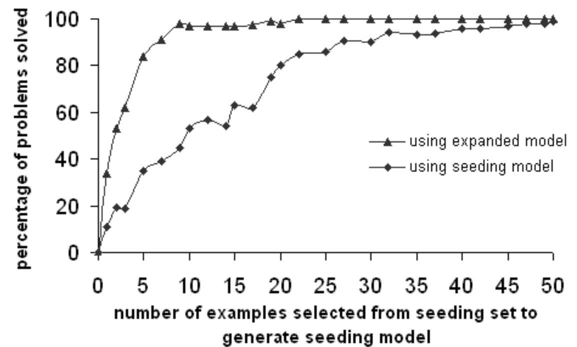


Figure 8: Experimental results of seeding models and expanded models after one circle of hypothesis generation.

Related Work

Automated HTN planning requires that a domain theory (descriptions of actions and methods) be present to a plan-

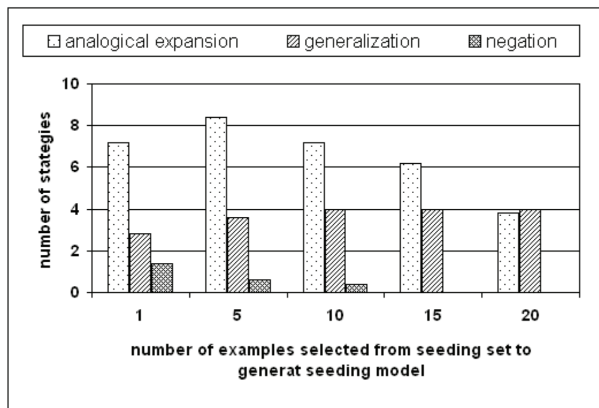


Figure 9: Strategies used to explore seeding models in Figure 8.

ner. Besides building up tools to facilitate effective planning domain acquisition and validation (e.g. GIPO (McCluskey, Liu, and Simpson 2003) and itSIMPLE (Vaquero et al. 2009)), researchers have been interested in pursuing (semi)autonomous programs to generate a domain theory for those complicated empirical domains.

CHEF and PRODIGY/ANALOGY (Veloso and Carbonell 1993) are two representative systems that extend planning domain knowledge with the analogical knowledge and case-based reasoning. While CHEF employs domain-dependent reasoning knowledge, PRODIGY/ANALOGY develops completely domain-independent analogical reasoning mechanisms and uses cases as search control knowledge. PRODIGY/OBSERVER (Wang 1996) automatically acquires and refines the preconditions and effects of operators by observing expert solution traces. PRODIGY/EXPO (Carbonell and Gil 1990) refines incomplete operator models by proposing experiments to test the explanations for those observed divergences. However, none of these systems does task model learning.

Instead of requiring a large numbers of training cases as those case-based reasoning systems, some authors have proposed an explanation-based learning approach that can learn from a single training example with the aid of the domain knowledge (DeJong and Mooney 1986). For example, GRASPHER (Bennett and Dejong 1996) implements a permissive planning approach to acquire and refine generalized plan schema through explanation-based learning.

To learn or improve the hierarchical structures relating tasks and subtasks (task models), one approach learns preconditions of HTN methods from HTN plan solution examples, e.g. CaMeL(++) (Ilghami et al. 2005) and DInCaD (Xu and Muñoz-Avila 2005). The other approach elicits the hierarchical structures of tasks from a collection of STRIPS action plans and hard-coded hierarchical annotation, e.g. HTN-MAKER (Hogg, Muñoz-Avila, and Kuter 2008). The generalization strategy proposed in this paper is similar to the approach described in DInCaD; however, the other exploration strategies presented in our work make our approach go further in expanding planning domain knowledge. The main difference between our approach and these previous case-based systems is that the previous works can only update their knowledge bases of HTN methods when they are provided with new problem solution cases; thus, they cannot produce methods that are not captured in the plan examples.

Another approach to exploiting hierarchies in planning is abstraction, such as ABSTRIPS (Sacerdoti 1974) and ALPINE (Knoblock 1990). These systems use both a collection of operators and an abstraction model that benefits the search process. Newton et al. (2008) learn control knowledge not captured by examples with genetic approach.

In summary, different from the previous approaches, HTN-Explorer integrates various strategies with an aim at self-directed exploring a bigger search space that example data does not provide. HTN-Explorer provides a flexible framework to integrate general domain-independent HTN exploration strategies and domain specific background knowledge represented in an ontology. The strategies HTN-Explorer implements to expand a domain space are not totally example-dependent. In addition, HTN-Explorer implements a heuristic function for evaluation of hypotheses. Generally, HTN-Explorer is like a knowledge discovery system, such as AM (Lenat 1982) and HAMB (Livingston, Rosenberg, and Buchana 2003). While AM focuses on mathematics domain and HAMB concentrates on chemistry discovery, our work focuses on exploring an incomplete HTN model.

Conclusion and Future Work

In this paper, we propose a framework that present hypotheses to explore an incomplete HTN planning domain. We present multiple exploration strategies and a heuristic function to estimate the value that a hypothesis deserves experiments to provide new theory for a domain. Finally, we demonstrate an empirical evaluation to test the effectiveness of our approach with the UM Translog domain.

For future work, more useful exploration strategies will be added to HTN-Explorer. In this paper, we have not considered the utilities of a plan execution, such as cost and duration. In real world, these properties are also important for hypothetical model generation. We will add utility consideration into a hypothesis evaluation. In addition, in this paper, we have only taken the advantage of the hierarchical structure of an ontology. The future work will study how to use the detailed properties of concepts in an ontology to propose and evaluate hypothetical models.

In our opinion, a planning domain model (HTN or non-HTN) can be improved by our approach. For example, an operator model can be expanded by the hypothesis strategies we presented. In addition, our approach can be applied to discover users' preference or to discover interesting knowledge for some real world domains whose background knowledge can be represented in HTN formalism. In the future, we will apply our approach to more planning domains to test its effectiveness.

References

Andrews, S.; Kettler, B.; Erol, K.; and Hendler, J. 1995. Um translog: A planning domain for the development and benchmarking of planning systems. *Technical Report, Dept. of CS, Univ. of Maryland at College Park*.

Bennett, S. W., and Dejong, G. F. 1996. Real-world robotics: Learning to plan for robust execution. *Machine Learning* 23(2-3):121–161.

Birnbaum, L.; Collins, G.; Freed, M.; and Krulwich, B. 1990. Model-based diagnosis of planning failures. In *Proceedings of AAAI'90*, 318–323.

Carbonell, Y. G., and Gil, Y. 1990. Learning by experimentation: The operator refinement method. *Machine Learning* 3:191–213.

Currie, K., and Tate, A. 1991. O-plan: the open planning architecture. *Artificial Intelligence* 52:49–86.

DeJong, G. E., and Mooney, R. J. 1986. Explanation-based learning: An alternative view. *Machine Learning* 1(2):145–176.

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

Estlin, T.; Castano, R.; Anderson, B.; Gaines, D.; Fisher, F.; and Judd, M. 2003. Learning and planning for mars rover science. In *Proceedings of IJCAI'03*.

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

Hogg, C.; Muñoz-Avila, H.; and Kuter, U. 2008. Htn-maker: Learning htns with minimal additional knowledge engineering required. In *Proceedings of AAAI'08*.

Ilghami, O.; Muñoz-Avila, H.; Nau, D. S.; and Aha, D. W. 2005. Learning approximate preconditions for methods in hierarchical plans. In *Proceedings of ICML'05*, 337–344.

Ilghami, O. 2005. Documentation for jshop2. Technical Report CS-TR-4694, University of Maryland, Department of Computer Science.

Knoblock, C. 1990. Learning abstraction hierarchies for problem solving. In *Proceedings of AAAI'08*, 923–928.

Lenat, D. 1982. Am: Discovery in mathematics as heuristic search. *Knowledge-Based Systems in Artificial Intelligence* 3–225.

Livingston, G.; Rosenberg, J.; and Buchana, B. 2003. An agenda- and justification-based framework for discovery systems. *Knowledge and Information Systems* 5:133–161.

McCluskey, T. L.; Liu, D.; and Simpson, R. 2003. Gipo ii: Htn planning in a tool-supported knowledge engineering environment. In *Proceedings of ICAPS'03*.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Muñoz-Avila, H.; Murdock, J.; Wu, D.; and Yaman, F. 2005. Applications of shop and shop2. *IEEE Intelligent Systems* 20(2):34–41.

Nau, D. S. 2007. Current trends in automated planning. *AI Magazine* 28(4):43–58.

Newton, M. A. H.; Levine, J.; Fox, M.; and Long, D. 2008. Learning macros that are not captured by given example plans. In *Supplementary Online Proceedings for Poster Papers at ICAPS'08*.

Ram, A.; Narayanan, S.; and Cox, M. T. 1995.

Reiter, R. 1987. A theory of diagnosis from first principles. *Artificial Intelligence* 32(1):57–96.

Sacerdoti, E. 1974. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence* 5(2):115–135.

Sacerdoti, E. 1975. The nonlinear nature of plans. In *Proceedings of IJCAI'75*, 206–214.

Smith, S. J. J.; Nau, D. S.; and Throop, T. 1998. Computer bridge: A big win for ai planning. *AI Magazine* 19(2):93–105.

Vaquero, T. S.; Silva, J.; Ferreira, M.; Tonidandel, F.; and Beck, J. 2009. itsimple3:0: From uml requirements and petri net-based analysis to pddl representation in the process of modeling plans for real applications. In *Proceeding of ICAPS 2009 Workshop on Knowledge Engineering for Planning and Scheduling*.

Veloso, M. M., and Carbonell, J. G. 1993. Derivational analogy in prodigy: Automating case acquisition, storage, and utilization. *Machine Learning* 10:249–278.

Wang, X. 1996. A mulitstrategy learning system for planning operator acquisition. In *Proceedings of the Third International Workshop on Multistrategy Learning*.

Wilkins, D. 1985. Recovering from execution errors in sipe. *Computational Intelligence* 1:33–45.

Xu, K., and Muñoz-Avila, H. 2005. A domain-independent system for case-based task decomposition without domain theories. In *Proceedings of AAAI'05*, 234–240.

Zimmerman, T., and Kambhampati, S. 2003. Learning-assisted automated planning: Looking back, taking stock, going forward. *AI Magazine* 24(2):73–96.

Model Updating in Action

Maria V. de Menezes and **Leliane N. de Barros**
Department of Computer Science
IME-USP

Silvio do L. Pereira
Department of Information Technology
FATEC-SP\CEETEPS

Abstract

Model updating is a formal approach to automatically correct a system model $\mathcal{M}$ with respect to some property φ not satisfied by $\mathcal{M}$. The well known model updating approaches are based on Computational Tree Logic (CTL), a branch time temporal logic which does not take into account the actions behind the state transitions. In previous work we have proposed a model checker and a planner based on α-CTL – a temporal logic whose semantics is based on actions – to solve extended reachability goals. In this paper, we present a model updating approach based on α-CTL that can be used to automatically suggest modifications in a state transition model induced by a set of actions and also is able to suggest changes directly in the action specification.

Introduction

Errors are common during the design of systems and their late detection and correction can be one of the major reasons for a high cost design. However, it can be reduced if the designer is able to early detect them, i.e., during system specification. By using formal methods to specify a system behavior, we can apply *model checking* techniques (Müller-Olm, Schmidt, and Steffen 1999) to automatically detect not met requirements. In order to understand how model checking works, let us consider the well-known *microwave oven* scenario presented by (Clarke E. 1999), which represents two main microwave usage processes: food heating and cooking.

The system designer starts by defining which properties will be used to describe the current state of a system. In the microwave oven example, the state properties are: *started* (indicating the microwave is operating), *closed* (indicating the microwave door is closed), *heated* (indicating the food inside microwave oven is heated) and *cooked* (describing that the food is cooked). Those properties are propositional atoms used to describe what is true in the system state and, their negation, describes what is false. Additionally, a state property *error* indicates the error detected during system operation (in our example, an error occurs in the situation where the oven starts and the door is open). Furthermore, system designer has to specify the actions that cause state transitions, that are: *start*, *finish*, *open-door*, *close-door*, *warm up*, *cook* and *reset*. Figure 1 shows a preliminary design model of a microwave oven given by a state transition diagram.

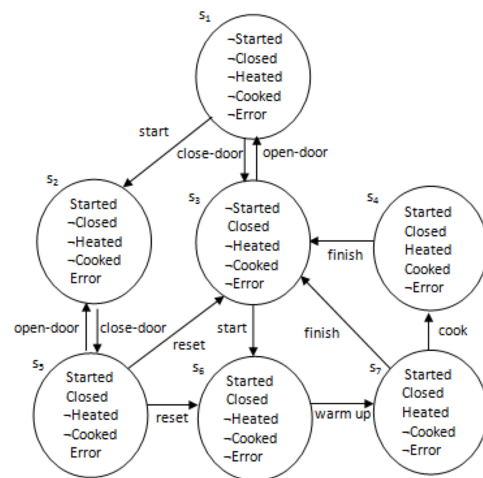


Figure 1: Formal model of a microwave oven, adapted from (Clarke E. 1999).

In the initial state s_1, a user can select one of the two actions: *start* and *close-door*. The *start* action takes the system to state s_2, which indicates the existence of an error. Notice that an error will persist even if the action *close-door* is selected (transition from s_2 to s_5). In this situation, the user has to reset (action *reset*) the microwave which can take the system to one of the two states: s_3 and s_6, both without error (note that *reset* is a non-deterministic action: an action with uncertain effects). If the *close-door* action is selected in the initial state s_1, the error does not occur and the food inside the microwave oven can be heated and\or cooked without making the user to reset the oven.

Suppose the designer wants to verify if the system specification in Figure 1 satisfies the temporal formula φ defined as: " *once the microwave oven is started, the food inside will be heated in some future state*". That means, φ is satisfied in a system model with no state where both *started* and $\neg$*heated* properties are true. The paths $[s_1, s_2, s_5, s_3, s_1, \cdots]$, $[s_1, s_2, s_5, s_2, \cdots]$ are examples where φ is not satisfied. The rationale behind φ can be "*the user should not have to reset*

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Figure 2: Model checker.

the oven in order to be able to heat and\or cook his food".

Although the microwave example seems to be easy to model and verify temporal properties, having defined 5 state properties (fluents or state variables) implies in 32 (2^5) possible states and 1024 (2^{10}) possible state transitions. Thus, to design a system that allows a user to achieve its intended goals (e.g. to heat or cook a meal) and yet guaranteeing to hold some temporal properties, is a complex task that gets harder with the size of state space (i.e. the number of state variables).

Model checking consists of automatically solving the problem $\mathcal{K} \models \varphi$, where $\mathcal{K}$ is a formal model of a system and φ is a formal specification of a temporal property to be verified in this system. Essentially, a model checker (Figure 2) is an algorithm that receives a pair $(\mathcal{K}, \varphi)$ as input and systematically visits the states of the model $\mathcal{K}$, in order to verify if the property φ holds. When all states in $\mathcal{K}$ satisfy property φ, the model checker returns success; otherwise, it returns a counter-example (e.g., a state in the model $\mathcal{K}$ where the property φ is violated). One of the limitations of model checking is that, when the property is violated, it only returns a counter-example leaving the task of modifying the system model to the designer.

Model updating is a technique that extends model checking functions in order to support the repair of a faulty system. Zhang and Ding(2008) proposed a model update algorithm that takes a given *Kripke model* $\mathcal{K}$ (Kripke 1963) - a state transition model without action specification - with respect to an arbitrary CTL formula φ and generates an updated model $\mathcal{K}'$ that: (1) satisfies φ and (2) has a minimal change with respect to the original model. To generate $\mathcal{K}'$, this approach uses *primitive update operations* such as: *add a relation element*, *remove a relation element*, *change labelling function on one state*, *add a state* and *remove a state*. For example, a possible correction in the model of Figure 1 is "*remove the transition between the states s_1 and s_2*". This means that "*It is not allowed to start the microwave oven with the opened door*".

Motivation: KE for planning vs. model updating

The microwave formal model from Figure 1 can be seen as a set of plans, specified by the system designer to achieve the goals: heat and\or cook a meal. Each plan is supposed to be executed by an user (according with a "system manual").

In artificial intelligence planning area, the task is to automatically generate a plan of actions given a goal specification and a system model (e.g., a factory or a robot environment model). A planning domain is specified in terms of a set of action schema which can be used to induce the system model. Nevertheless, it is very difficult, even for a simple planning domain, to specify a correct set of actions. Although automatic planning has been the subject of extensive study in the AI community since the early 1970s, low effort has been given to the task of modeling and verification of planning domains.

In order to model and verify a planning domain, a designer should start by describing a preliminary set of actions to be further refined. This refinement can be done using, e.g.: (i) a set of plan examples, for a given class of goals, specified in an ad hoc way by a domain expert (or possibly generated by an automatic planner); (ii) a state transition model induced by the semantics of the preliminary set of action schema for a small problem.

Notice that in the system model represented by Figure 1 the state transitions are labelled with actions. Traditional model updating approaches based on CTL (e.g (Zhang and Ding 2008)) do not take into account the actions behind the transitions and therefore can not be applied to update (refine) planning domains. Actions are not part of a CTL *Kripke* model, which is the formalism used in most of the model checking and updating approaches (Buccafurri et al. 1999; Harris and Ryan 2003; Zhang and Ding 2008). In this paper we show that by representing actions in the formal model of a system we can extend model updating techniques to be used as an important support tool for knowledge acquisition, modeling and verification of planning domains.

Since actions are not part of *Kripke* structure, we can only represent them using a temporal logic whose semantics is based on actions. Pereira and Barros (2008) proposed an extension of CTL, called α-CTL, whose semantics considers the transition actions. They have also developed a model checker based on this logic, named α-CTL *model checker*. This work presents a model updating approach based on α-CTL that can be used to automatically suggest modifications in a state transition model induced by a set of actions and therefore is able to suggest changes directly in the actions specification. We also define a criterion of minimal change for α-CTL model updating.

The remainder of this article is organized as follows: we first show the basic concepts of CTL model checking and CTL model updating; then we define a labelled transition system and a model checker based on α-CTL . Finally, we show how to perform model update, based on α-CTL , that can suggest modification in the set of actions of a planning domain.

CTL Model Checking and Update

In this section, we introduce the basic concepts of CTL model checking and CTL model update.

The branching time temporal logic CTL (COMPUTATION TREE LOGIC) (Clarke and Emerson 1982) allows us to reason about alternative time lines (i.e., alternative futures). In CTL the temporal operators must be preceded by some quantifier: $\exists$ (Figure 3) or $\forall$ (Figure 4).

- $\forall\circ$ (in *all next* states)
- $\exists\circ$ (at *some next* state)
- $\forall\Box$ (*invariant*, in *all future* state)
- $\exists\Box$ (*invariant*, at *some* future state)
- $\forall\diamond$ (*finally*, in *all* future states)
- $\exists\diamond$ (*finally*, at *some* the future state)

- $\forall[\varphi_1 \sqcup \varphi_2]$ (*since*, in *all* future states)
- $\exists[\varphi_1 \sqcup \varphi_2]$ (*since*, at *some* future state)

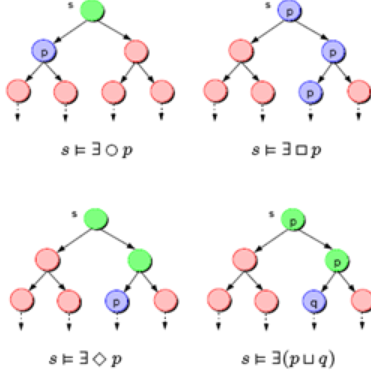


Figure 3: Semantics of the CTL temporal operators preceded by existential quantifier.

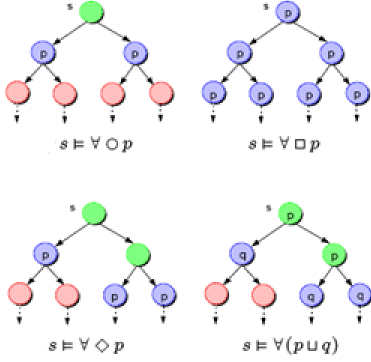


Figure 4: Semantics of the CTL temporal operators preceded by universal quantifier.

The CTL formulas are composed by atomic propositions, propositional operators and temporal operators. The symbols $\circ$ (next), $\square$ (invariantly), $\diamond$ (finally) and $\sqcup$ (until), combined with the quantifiers $\exists$ and $\forall$, are used to compose the temporal operators of this logic.

The syntax of CTL is inductively defined as:
$\varphi \doteq p \in \mathbb{P} \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \exists \circ \varphi_1 \mid \forall \circ \varphi_1 \mid \exists \square \varphi_1 \mid \forall \square \varphi_1 \mid \exists(\varphi_1 \sqcup \varphi_2) \mid \forall(\varphi_1 \sqcup \varphi_2)$.

The temporal operators $\exists\diamond$ and $\forall\diamond$ are defined as:

- $\exists \diamond \varphi \doteq \exists(\top \sqcup \varphi)$
- $\forall \diamond \varphi \doteq \forall(\top \sqcup \varphi)$

The semantics of CTL is defined over a *Kripke structure* $\mathcal{K} = \langle S, L, T \rangle$, where S is a set of states, $L : S \mapsto 2^{\mathbb{P}}$ is a state labelling relation and $T \subseteq S \times S$ is a transition relation. A path in $\mathcal{K}$ is a sequence of states $[s_0, s_1, \cdots]$ such that $s_i \in S$ and $(s_i, s_{i+1}) \in T$, for all $i \geqslant 0$.

Given a *Kripke structure* $\mathcal{K}$ and a state $s_0 \in S$, the CTL satisfiability relation is defined as:

Figure 5: Model updater receives $\mathcal{K}$ and φ and returns an updated model $\mathcal{K}$'

- $(\mathcal{K}, s_0) \models p$ iff $p \in L(s_0)$;
- $(\mathcal{K}, s_0) \models \neg\varphi$ iff $(\mathcal{K}, s_0) \not\models \varphi$;
- $(\mathcal{K}, s_0) \models \varphi_1 \wedge \varphi_2$ iff $(\mathcal{K}, s_0) \models \varphi_1$ and $(\mathcal{K}, s_0) \models \varphi_2$;
- $(\mathcal{K}, s_0) \models \varphi_1 \vee \varphi_2$ iff $(\mathcal{K}, s_0) \models \varphi_1$ or $(\mathcal{K}, s_0) \models \varphi_2$;
- $(\mathcal{K}, s_0) \models \exists \circ \varphi$ iff for some path $[s_0, s_1, \cdots]$ in $\mathcal{K}$, $(\mathcal{K}, s_1) \models \varphi$;
- $(\mathcal{K}, s_0) \models \forall \circ \varphi$ iff for every path $[s_0, s_1, \cdots]$ in $\mathcal{K}$, $(\mathcal{K}, s_1) \models \varphi$;
- $(\mathcal{K}, s_0) \models \exists \square \varphi$ iff for some path $[s_0, s_1, \cdots]$ in $\mathcal{K}$, for $i \geq 0$, $(\mathcal{K}, s_i) \models \varphi$;
- $(\mathcal{K}, s_0) \models \forall \square \varphi$ iff for every path $[s_0, s_1, \cdots]$ in $\mathcal{K}$, for $i \geq 0$, $(\mathcal{K}, s_i) \models \varphi$;
- $(\mathcal{K}, s_0) \models \exists(\varphi_1 \sqcup \varphi_2)$ iff for some path $[s_0, s_1, \cdots]$ in $\mathcal{K}$, there exists $i \geq 0$ such that $(\mathcal{K}, s_i) \models \varphi_2$ and, for $0 \leq j < i$, $(\mathcal{K}, s_j) \models \varphi_1$;
- $(\mathcal{K}, s_0) \models \forall(\varphi_1 \sqcup \varphi_2)$ iff for every path $[s_0, s_1, \cdots]$ in $\mathcal{K}$, there exists $i \geq 0$ such that $(\mathcal{K}, s_i) \models \varphi_2$ and, for $0 \leq j < i$, $(\mathcal{K}, s_j) \models \varphi_1$.

Model update framework

Let $\mathcal{K}$ be a formal model of a system and φ be a formal specification of a property that is not satisfied in this system, i.e., $\mathcal{K} \not\models \varphi$. *Model update* (Zhang and Ding 2008) consists of generating a new model $\mathcal{K}'$ that satisfies the input formula ($\mathcal{K}' \models \varphi$) and has a minimal change with respect to the original model $\mathcal{K}$. Then, a *model updater* (Figura 5) is an algorithm that receives a pair $(\mathcal{K}, \varphi)$, where $\mathcal{K} \not\models \varphi$, and returns a new model $\mathcal{K}'$, where $\mathcal{K}' \models \varphi$. The updated model $\mathcal{K}'$ can be viewed as a possible correction on the original system specification.

Zhang and Ding (2008) proposed a formal framework for CTL model update, defining primitive operations and specifying a minimal change principle for CTL model updating. Below, we list the Zhang and Ding (2008) primitive operations:

PU1: Adding one relation element. Let be $\mathcal{K} = \langle S, L, T \rangle$, its updated model $\mathcal{K}' = \langle S', L', T' \rangle$ is obtained from $\mathcal{K}$ by adding only one new relation element. That is, $S' = S, L' = L, T' = T \cup (s_i, s_j)$, where $s_i, s_j \in S$ and $(s_i, s_j) \notin T$.

PU2: Removing one relation element. Let be $\mathcal{K} = \langle S, L, T \rangle$, its updated model $\mathcal{K}' = \langle S', L', T' \rangle$ is obtained from $\mathcal{K}$ by removing only one existing relation element. That is, $S' = S, L' = L, T' = T - (s_i, s_j)$, where $(s_i, s_j) \in T$ for two states $s_i, s_j \in S$.

PU3: Changing labelling function on one state. Let be $\mathcal{K} = \langle S, L, T \rangle$, its updated model $\mathcal{K}' = \langle S', L', T' \rangle$ is obtained from $\mathcal{K}$ by changing labelling function on a particular state. That is, $S' = S, T' = T, \forall s \in (S - s^*), s^* \in S, L'(s) = L(s)$ and $L'(s^*)$ is a set of true variable assigned in s^* where $L'(s^*) \neq L(s^*)$.

PU4: Adding one state. Let be $\mathcal{K} = \langle S, L, T \rangle$, its updated model $\mathcal{K}' = \langle S', L', T' \rangle$ is obtained from $\mathcal{K}$ by adding only one new state. That is, $S' = S \cup s^*, s^* \notin S$, $T' = T$ and $\forall s \in S, L'(s) = L(s)$ and $L'(s^*)$ is a set of true variables assigned in s^*.

PU5: Removing one isolated state. Let be $\mathcal{K} = \langle S, L, T \rangle$, its updated model $\mathcal{K}' = \langle S', L', T' \rangle$ is obtained from $\mathcal{K}$ by removing only one isolated state. That is, $S' = S - s^*$, where $s^* \in S$ and $\forall s \in S$ such that $s \neq s^*$, neither (s, s^*) nor (s^*, s) is not in T, $T' = T$ and $\forall s \in S', L'(s) = L(s)$.

Model update should obey minimal change rules. Although, the primitive operations PUi where $i = 1, 2, ..5$ can be used to define minimal change criterion for CTL model update, within this framework it is not possible to make updates considering the actions of the system specification, as we propose in the next sections.

Labelled transition system

Let $\mathbb{P} \neq \emptyset$ be a finite set of atomic propositions, denoting properties of a system, and $\mathbb{A} \neq \emptyset$ be a finite set of actions, representing the events of a system.

Definition 1. *A labelled transition system with signature* $(\mathbb{P}, \mathbb{A})$ *is defined by* $\mathcal{M} = \langle S, L, T \rangle$*, where:*

- $S \neq \emptyset$ *is a finite set of states;*
- $L : \mathcal{S} \mapsto 2^{\mathbb{P}}$ *is a state labelling function;*
- $T : S \times \mathbb{A} \times S$ *is a state transition relation.*

A labelled transition system with signature $(\mathbb{P}, \mathbb{A})$ can be represented as a *transition graph*, where states are labelled with subsets of $\mathbb{P}$ and transitions are labelled with elements of $\mathbb{A}$. Set $\mathcal{S}$ has all possible states of a model, state labelling function designs for each state $s \in S$ a proposition set $L(s) \in 2^{\mathbb{P}}$ and labelling function T designs for each transition $t \in T$ an action $a \in \mathbb{A}$. Given two states $s_i, s_j \in S$ and an action $a \in \mathbb{A}$, a transition between s_i and s_j is represented by $(s_i, a, s_j) \in T$.

α-CTL model checking

An example where representing actions may allow for a more rational model checking and updating is shown in Figure 6. Suppose that φ is a desired property: "*from the initial state* s_0*, all transitions take to state in which* p *is true*". A traditional model checker (MC) would represent the system model by Figure 6(a), i.e., by a *Kripke* model. Since the transition (s_0, s_2) does not satisfy φ, the CTL based MC would detect this error and a model update would indicate " *remove the relation* (s_0, s_2)". However, if we do represent the transition actions (Figure 6(b)) a model checker would not detect an error, since there is an action a that takes the system to state s_2 that satisfies φ. Figure 6(c) shows another example of how model checking can be more rational when we represent the actions in the state transition model: by knowing that the two transitions correspond to non-deterministic effects of action a, removing one transition implies removing the other.

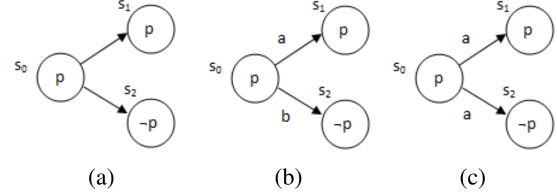


Figure 6: (a) Kripke structure. (b) and (c) Labelled transition model.

In this section, we present the branching time temporal logic α-CTL and a model checker based on this new logic.

The new temporal logic α-CTL

Differently from CTL, the branching time temporal logic α-CTL, proposed in (Pereira and de Barros 2008), can discern among various actions that produce state transitions.

Syntax of α-CTL In CTL, a formula $\forall \circ \varphi$ holds on a state s if and only if it holds on all successors of s, independently of the actions labeling the transitions from s to its successors. In α-CTL, to enforce that actions play an important role in its semantics, we use a different set of "dotted" symbols to represent temporal operators: $\odot$ (next), $\boxdot$ (invariantly), $\diamond$ (finally) and $\sqcup$ (until).

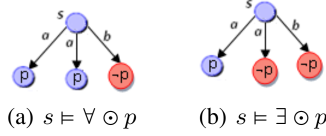
Definition 2. *Let* $p \in \mathbb{P}$ *be an atomic proposition. The syntax of* α-CTL *is inductively defined as:*

$\varphi ::= p \mid \neg p \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \exists \odot \varphi \mid \forall \odot \varphi \mid \exists \boxdot \varphi \mid \forall \boxdot \varphi \mid \exists(\varphi_1 \sqcup \varphi_2) \mid \forall(\varphi_1 \sqcup \varphi_2)$

According to the α-CTL syntax, well-formed formulas are in *negative normal form*, where the scope of negation is restricted to the atomic propositions. Furthermore, all temporal operators are prefixed by a path quantifier ($\exists$ or $\forall$). The temporal operators derived from $\diamond$ are defined as: $\exists \diamond \varphi_2 \doteq \exists(\top \sqcup \varphi_2)$ and $\forall \diamond \varphi_2 \doteq \forall(\top \sqcup \varphi_2)$.

Semantics of α-CTL Let $\mathbb{P} \neq \emptyset$ be a finite set of atomic propositions and $\mathbb{A} \neq \emptyset$ be a finite set of actions. An α-CTL *temporal model* over $(\mathbb{P}, \mathbb{A})$ is a transition graph where states are labelled with subsets of $\mathbb{P}$ and transitions are labelled with elements of $\mathbb{A}$.

Intuitively, a state s in a *temporal model* $\mathcal{M}$ satisfies a formula $\forall \odot \varphi$ (or $\exists \odot \varphi$) (Figure 7) if there exists an action α that, when executed in s, *necessarily* (or *possibly*) reaches an immediate successor of s which satisfies the formula φ. In other words, the modality $\odot$ represents the set of α-successors of s, for some particular action $\alpha \in \mathbb{A}$; the quantifier $\forall$ requires that all these α-successors satisfy φ; and quantifier $\exists$ requires that some of these α-successors satisfy φ.



(a) $s \models \forall \odot p$ (b) $s \models \exists \odot p$

Figure 7: Semantics of the temporal operator $\odot$. (a) p is true for all sucessors of s troughout action a. (b) p is true for some sucessor of s troughout action a.

Before we can give a formal definition of the α-CTL semantics, we need to define the concept of *preimage* of a set of states.

Definition 3. *Let be $Y \subseteq S$ a set of states. The weak preimage of Y, denoted by $\mathcal{I}_{\exists}^{-}(Y)$, is a set $\{s \in S : a \in \mathbb{A}$ and $\exists(s, a, s') \in T, s' \in S, s' \subseteq Y\}$; and the strong preimage of Y, denoted by $\mathcal{I}_{\forall}^{-}(Y)$, is the set $\{s \in \mathcal{S} : a \in \mathbb{A}$ and $\forall(s, a, s') \in T, s' \in S, s' \subseteq Y\}$*

The semantics of the global temporal operators ($\exists\boxdot$, $\forall\boxdot$, $\exists\sqcup$ and $\forall\sqcup$) is derived from the semantics of the local temporal operators ($\exists\odot$ and $\forall\odot$), by using least (μ) and greatest (ν) fixpoint operations.

Definition 4. *Let $\mathcal{M} = \langle S, L, T\rangle$ be a temporal model with signature $(\mathbb{P}, \mathbb{A})$ and $p \in \mathbb{P}$ be an atomic proposition. The intention of an α-CTL formula φ in $\mathcal{M}$ (or the set of states satisfying φ in $\mathcal{M}$, denoted by $[\![\varphi]\!]_{\mathcal{M}}$, is defined as:*

- $[\![p]\!]_{\mathcal{M}} = \{s \in S : p \in L(s)\}$
- $[\![\neg p]\!]_{\mathcal{M}} = S \setminus [\![p]\!]_{\mathcal{M}}$
- $[\![\varphi_1 \wedge \varphi_2]\!]_{\mathcal{M}} = [\![\varphi_1]\!]_{\mathcal{M}} \cap [\![\varphi_2]\!]_{\mathcal{M}}$
- $[\![\varphi_1 \vee \varphi_2]\!]_{\mathcal{M}} = [\![\varphi_1]\!]_{\mathcal{M}} \cup [\![\varphi_2]\!]_{\mathcal{M}}$
- $[\![\exists \odot \varphi_1]\!]_{\mathcal{M}} = \mathcal{T}_{\exists}^{-}([\![\varphi_1]\!]_{\mathcal{M}})$
- $[\![\forall \odot \varphi_1]\!]_{\mathcal{M}} = \mathcal{T}_{\forall}^{-}([\![\varphi_1]\!]_{\mathcal{M}})$
- $[\![\exists \boxdot \varphi_1]\!]_{\mathcal{M}} = \nu Y.([\![\varphi_1]\!]_{\mathcal{M}} \cap \mathcal{T}_{\exists}^{-}(Y))$
- $[\![\forall \boxdot \varphi_1]\!]_{\mathcal{M}} = \nu Y.([\![\varphi_1]\!]_{\mathcal{M}} \cap \mathcal{T}_{\forall}^{-}(Y))$
- $[\![\exists(\varphi_1 \sqcup \varphi_2)]\!]_{\mathcal{M}} = \mu Y.([\![\varphi_2]\!]_{\mathcal{M}} \cup ([\![\varphi_1]\!]_{\mathcal{M}} \cap \mathcal{T}_{\exists}^{-}(Y)))$
- $[\![\forall(\varphi_1 \sqcup \varphi_2)]\!]_{\mathcal{M}} = \mu Y.([\![\varphi_2]\!]_{\mathcal{M}} \cup ([\![\varphi_1]\!]_{\mathcal{M}} \cap \mathcal{T}_{\forall}^{-}(Y)))$

A model checker for α-CTL

A model checker for α-CTL can be directly implemented from its semantics. Given a model $\mathcal{M} = \langle S, L, T\rangle$ and an α-CTL formula φ, the model checker computes the set C of states that do not satisfy the formula φ in $\mathcal{M}$; then, if C is the empty set, it returns success; otherwise, it returns C as counter-example.

```
α − MODELCHECKER(φ, M)
1 C ← S ∖ INTENTION(φ, M)
2 if C = ∅ then return success
3 else return C
```

The basic operation on this model checker is implemented by the function INTENTION, that inductively computes the intention of the formula φ in the model $\mathcal{M}$. The efficiency of α − MODELCHECKER can be highly improved with use of BDDs (Bryant 1992), resulting in an extremely efficient symbolic version of this model checker. More details about the α-MODELCHECKER can be found in (Pereira and de Barros 2008).

α-CTL model updating

In this section, we define the basic concepts about the proposed model updating system, which is based on the branching time temporal logic α-CTL (Pereira and de Barros 2008).

First, consider that a *complete model* $\mathcal{M}^\star = \langle S^\star, L^\star, T^\star\rangle$ (eventually induced by a formal specification in $\mathbb{A}$) is a labelled transition system with signature $(\mathbb{P}, \mathbb{A})$. Thus, according to Definition 1:

- $S^\star \neq \varnothing$ is a finite set of states;
- $L^\star : S^\star \mapsto 2^{\mathbb{P}}$ is a state labelling function;
- $T^\star \subseteq S^\star \times \mathbb{A} \times S^\star$ is a state transition relation.

Moreover, consider that the model which the system designer wants to correct is a *partial model* $\mathcal{M} = \langle S, L, T\rangle$ such that $\mathcal{M} \subseteq \mathcal{M}^\star$ or, more precisely:

- $S \subseteq S^\star$ is a finite set of states;
- $L : S \mapsto 2^{\mathbb{P}}$ such that, for all $s \in S, L(s) = L^\star(s)$;
- $T \subseteq T^\star$ such that, if $(s_i, a, s_j) \in T$ and $(s_i, a, s_k) \in T^\star$, then $(s_i, a, s_k) \in T$.

In Figure 8, we have a labelled transition system representing a complete model $\mathcal{M}^\star$, where the highlighted substructure is the partial model $\mathcal{M} \subseteq \mathcal{M}^\star$ given by system designer.

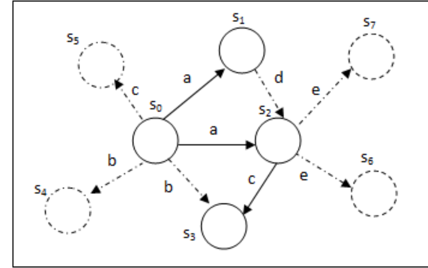


Figure 8: A labelled transition system. Solid lines represent the partial model $\mathcal{M}$ and dashed lines represent the complete model $\mathcal{M}^\star$.

Given a partial model $\mathcal{M} \subseteq \mathcal{M}^\star$, a initial state $s_0 \in S$ and an α-CTL formula φ such that $(\mathcal{M}, s_0) \not\models \varphi$, the α-CTL *model updating problem* consists of generating a new partial model $\mathcal{M}' \subseteq \mathcal{M}^\star$, called *updated model*, such that: (*i*) $(\mathcal{M}', s_0) \models \varphi$ and (*ii*) $\mathcal{M}'$ has a minimal change with respect to the original partial model $\mathcal{M}$.

To update a partial model $\mathcal{M}$, w.r.t. a complete model $\mathcal{M}^\star$, in order to satisfy an α-CTL formula φ, we define the follow primitive operations:

PUA1: Adding transitions induced by an action. Given a partial model $\mathcal{M} = \langle S, L, T\rangle$, a corresponding updated model $\mathcal{M}' = \langle S', L', T'\rangle$, with respect to $\mathcal{M}^\star$, can be obtained from $\mathcal{M}$ by adding a transition between states $s_i, s_j \in S$. In other words:

- $T' = T \cup \{(s_i, a, s) \in T^\star : \exists a \in \mathbb{A}, (s_i, a, s_j) \in T^\star\}$
- $S' = S \cup \{s : \exists a \in \mathbb{A}, (s_i, a, s) \in T'\}$
- $L'(s) = L^\star(s), s \in S'$

Adding a transition between two states s_i and s_j in the partial model is possible only if there is some transition between these states in the complete model. For example, in Figure 8 we cannot add a transition between states s_2 and s_4 in the partial model (solid lines), because there is no transition between s_2 and s_4 in the complete model (dashed lines). In Figure 8, it is possible to add a transition between states s_0 and s_3, since a transition labelled with action b exists in the complete model. However, all the effects of action b must also be added. In Figure 8 in order to add a transition between states s_0 and s_3, using action b, we must also add state s_4 and the transition between s_0 and s_4 in the updated model.

PUA2: Removing transitions induced by an action. Given a partial model $\mathcal{M} = \langle S, L, T \rangle$, a corresponding updated model $\mathcal{M}' = \langle S', L', T' \rangle$, with respect to $\mathcal{M}^\star$, can be obtained from $\mathcal{M}$ by removing an existing transition between states $s_i, s_j \in S$, which is labelled by some action $a \in \mathbb{A}$. More formally:

- $T' = T - \{(s_i, a, s) \in T^\star : (s_i, a, s_j) \in T\}$
- $S' = S$
- $L'(s) = L(s), s \in S'$

It is important to observe that to remove an existing transition between states s_i and s_j labelled by an action a, we must also remove all transitions from s_i using action a, i.e., all non-deterministic effects of the action a in state s_i. For example, to remove the transition between states s_0 and s_1 in the partial model of Figure 8, it is also necessary to remove the transition between states s_0 and s_2.

PUA3: Adding a new state. Given a partial model $\mathcal{M} = \langle S, L, T \rangle$, a corresponding updated model $\mathcal{M}' = \langle S', L', T' \rangle$, with respect to $\mathcal{M}^\star$, can be obtained from $\mathcal{M}$ by adding only one new state. That is:

- $S' = S \cup \{s\}$, for some $s \in S^\star$, such that $s \notin S$
- $L'(s) = L^\star(s), s \in S'$
- $T' = T$

It is possible to add one state s in the partial model if and only if this state exists in the complete model.

PUA4: Removing an isolated state. Given a partial model $\mathcal{M} = \langle S, L, T \rangle$, a corresponding updated model $\mathcal{M}' = \langle S', L', T' \rangle$, with respect to $\mathcal{M}^\star$, can be obtained from $\mathcal{M}$ by removing only one isolated state. That is:

- $S' = S - \{s\}$, for some $s \in S$, such that for all $s_i \in S$, $s_i \neq s$, and $a \in \mathbb{A}$, we have $(s_i, a, s) \notin T$ and $(s, a, s_i) \notin T$
- $L'(s) = L(s), s \in S'$
- $T' = T$

Defining minimal change for α-CTL model updating

Based on the work of Zhang and Ding (2008), we can establish a *minimal change criterion* for a labelled transition system using the primitive update operations (PUA1 – PUA4), defined in previous section.

By using a primitive update operation PUAi, a partial model $\mathcal{M}$ given by a system designer can be updated in different ways. Thus, we need a criterion which allows us to measure the changes in the different possible updated models of $\mathcal{M}$ and choose the one which is more close to the original model $\mathcal{M}$.

Given a labelled transition system $\mathcal{M} = \langle S, L, T \rangle$ and a corresponding updated model $\mathcal{M}' = \langle S', L', T' \rangle$, for each operation PUA$i$, for $i = 1..4$, we use $Diff_{\text{PUA}_i}(\mathcal{M}, \mathcal{M}')$ to denote the differences between these two models.

$$Diff_{\text{PUA}_i}(\mathcal{M}, \mathcal{M}') = |T' - T| + |S' - S|$$

We also define $Diff(\mathcal{M}, \mathcal{M}')$ as the following tuple: $(Diff_{\text{PUA1}}(\mathcal{M}, \mathcal{M}'),\ Diff_{\text{PUA2}}(\mathcal{M}, \mathcal{M}'),\ Diff_{\text{PUA3}}(\mathcal{M}, \mathcal{M}'),\ Diff_{\text{PUA4}}(\mathcal{M}, \mathcal{M}'))$.

Now, we can precisely define the ordering $\leq_{\mathcal{M}}$ on labelled transition system.

Definition 5. *(Closeness ordering) - Given $\mathcal{M}$ a partial model and $\mathcal{M}'_1$, $\mathcal{M}'_2$ two corresponding updated models, with respect to a complete model $\mathcal{M}^\star$. We say that $\mathcal{M}'_1$ is at least as close to $\mathcal{M}'_2$, denoted as $\mathcal{M}'_1 \leq_{\mathcal{M}} \mathcal{M}'_2$, if and only if for each set of PUA1 – PUA4 operations that transform $\mathcal{M}$ into $\mathcal{M}'_2$, there exists a set of PUA1 – PUA4 operations that transform $\mathcal{M}$ into $\mathcal{M}'_1$, such that the following condition hold:*

$$Diff_{PUAi}(\mathcal{M}, \mathcal{M}'_1) \leq Diff_{PUAi}(\mathcal{M}, \mathcal{M}'_2), \text{ for } i = 1..4$$

We also denote $\mathcal{M}'_1 <_{\mathcal{M}} \mathcal{M}'_2$ if $\mathcal{M}'_1 \leq_{\mathcal{M}} \mathcal{M}'_2$ and $\mathcal{M}'_2 \nleq_{\mathcal{M}} \mathcal{M}'_1$. For example, if $\mathcal{M}, \mathcal{M}'_1$ and $\mathcal{M}'_2$ are models such that: $Diff_{\text{PUA}_1}(\mathcal{M}, \mathcal{M}'_1) = 5$; $Diff_{\text{PUA}_2}(\mathcal{M}, \mathcal{M}'_1) = 2$; $Diff_{\text{PUA}_3}(\mathcal{M}, \mathcal{M}'_1) = 4$; $Diff_{\text{PUA}_4}(\mathcal{M}, \mathcal{M}'_1) = 6$; $Diff_{\text{PUA}_1}(\mathcal{M}, \mathcal{M}'_2) = 3$; $Diff_{\text{PUA}_2}(\mathcal{M}, \mathcal{M}'_2) = 1$; $Diff_{\text{PUA}_3}(\mathcal{M}, \mathcal{M}'_2) = 1$ and $Diff_{\text{PUA}_4}(\mathcal{M}, \mathcal{M}'_2) = 5$. We say that $\mathcal{M}'_2 <_{\mathcal{M}} \mathcal{M}'_1$, i.e., $\mathcal{M}'_2$ is more close to $\mathcal{M}$ than $\mathcal{M}'_1$ is.

Definition 5 presents a measure on the difference between two labelled transition system with respect to a partial model given by designer. Intuitively, we say that model M'_1 is closer to M relative to model M'_2 if M'_1 is obtained from M by applying all primitive update operations that cause fewer changes than those applied to obtain model M_2. Having the ordering specified in Definition 5, we can define a α-CTL model updating formally.

Definition 6. *(Admissible Update) Let be a partial model $M = \langle S, L, T \rangle$, $\mathcal{M} = (M, s_0)$, where $s_0 \in S$, and an α-CTL formula φ, a $Update(\mathcal{M}, \varphi)$ is called an admissible model if the conditions below hold:*

- *$Update(\mathcal{M}, \varphi) = (M', s'_0), (M', s'_0) \models \varphi$, where $M' = (S', L', T')$ and $s'_0 \in S'$;*
- *There does nor exists another updated model $M'' = \langle S'', L'', T'' \rangle$ and $s''_0 \in S''$ such that $(M'', s''_0) \models \varphi$ and $M'' <_M M'$.*

Model Update in Action

Like in the CTL model update proposed by Zhang and Ding (2008), the primitive updating operations *PUA1-PUA4* can suggest modifications by adding or removing states and transitions but:

- by considering an action labelled transition model, the effects of nondeterministic actions imply in different model updates; and
- the temporal formula φ is expressed in α-CTL which, as shown in Pereira and Barros (2008) can specify more expressive planning goals.

In this section we extend model updating to modify an action specification, i.e., an action precondition and effect. The idea is to add two more updating operations to *PUA1-PUA4*, as we show next.

First, we define how to induce the complete model $\mathcal{M}*$ from a set of actions, described in a language based on its preconditions and nondeterministic effects.

Definition 7. *An action is defined by a triple action(name,pre,pos), where* pre *is the set of preconditions that must be true in a state* s *where the action is applied; and* pos *is the set of nondeterministic effects that becomes true in the state resulting from the execution of the action* a *on state* s*. We use* $pre(a)$ *refering to the preconditions of action* a *and* $pos(a)$ *for the effects of action* a*.*

For example, action $(a, \{p, q\}, \{\{r, s\}, \{u\}\})$ indicates that $pre(a) = \{p, q\}$ and the two possible nondeterministic effects of a are given by $pos(a) = \{\{r, s\}, \{u\}\}$

Given an action set $\mathbb{A}$, the preposition set $\mathbb{P}$ of the *complete model* is determined as:

$$\mathbb{P} = \{p : a \in \mathbb{A} \text{ and } pre(a) \cup pos(a)\}$$

that means, all the proposition symbols involved in the description of the set of actions $\mathbb{A}$ are elements of $\mathbb{P}$.

Definition 8. *The* complete model $\mathcal{M}^*_A$ *induced by a set of actions* $\mathbb{A}$ *is a structure* $\langle S^*_A, L^*_A, T^*_A \rangle$*, where:*

- S^*_A *is a finite set of enumerated state symbols such that* $|S| = |2^{\mathbb{P}}|$ *;*
- $L^*_A : S \mapsto 2^{\mathbb{P}}$ *is the labelling function that assigns to each state a set of atomic propositions ;*
- $T^*_A = \{(s_x, a, s_y) : s_x, s_y \in S, a \in \mathbb{A}, pre(a) \subseteq s_x,$ *eff* $\in pos(a)$*, eff* $\subseteq L(s_y)$*.*

Notice that the semantics of actions defines that a transition labelled with action a is added to the complete model $\mathcal{M}^*_A$ if: $pre(a)$ is satisfied in the state s_x; and $pos(a)$ is also satisfied in state s_y. That means, the induced model $\mathcal{M}^*_A$ contains transitions between states without preserving properties in s_x that are not modified by action a. One can justify this kind of transitions with the occurrence of exogenous events (a possible explanation for nondeterministic actions) or as a relaxed state transition model useful to suggest the updates on the partial model (or in a preliminary set of actions specification, as we will see in the next section). We could also induce a complete model from a set of actions respecting the classical planning assumptions (i.e., the STRIPS-like semantics). However, it would be too restrictive while modeling a new planning domain (specially in the case of a nondeterministic domain).

E.g., let us consider the following set of actions $\mathbb{A}$:

- action$(a, \{p, q\}, \{\{r, s\}, \{u\}\})$
- action$(b, \{r\}, \{t\})$
- action$(c, \{u\}, \{\{t\}\})$
- action$(d, \{p, u\}, \{\{t\}\})$

The set of proposition atoms is given by $\mathbb{P} = \{p, q, r, s, u, t\}$. So, the complete model $\mathcal{M}^*_A$ induced by $\mathbb{A}$ has 64 states and all possible transitions, according with the semantics (Definition 7) of actions in $\mathbb{A}$.

Planning Domain Model Updating

In order to develop a real world planning application, a knowledge engineer must specify a correct set of actions which can guarantee the synthesis of correct plans. Our claim is that the use of a formal method, such as the model update approach presented in previous section, can offer an important support to knowledge acquisition, modelling and verification of planning domains. In this paper we use model update in a planning domain w.r.t. a preliminary set of actions $\mathbb{A}$, by making the following correspondences:

- the complete model $\mathcal{M}^*_A$ is induced by the actions $\mathbb{A}$ according with Definition 8 (we may call this complete model as *weakly induced* by $\mathbb{A}$);
- a *partial model* $\mathcal{M} \subseteq \mathbb{A}$ can be seen as a part of the *complete model* $\mathcal{M}^*_A$ that can correspond to (i) a set of plans for a given class of goals, specified in an ad hoc way by a domain expert (or possibly generated by an automatic nondeterministic planner) or ; (ii) a state transition model induced by a stronger semantics of actions (with frame axioms) and
- an α-CTL temporal formula φ is a planning goal (which can since it is expressed by α-CTL can be more complex than a simple reachability goal).

Formally, given a complete model $\mathcal{M}^*_A$ induced by a set of actions $\mathbb{A}$; a partial model $\mathcal{M} \subseteq \mathcal{M}^*_A$; and a α-CTL formula φ defining a planning goal, the set of primitive updating operations $PUA1 - PUA4$ can be used to refine and validate the partial model $\mathcal{M}$. Plus, in order to perform updates directly on the action specification, we need to define two extra primitive updating operations, named $PU5 - PU6$, as follows.

PUA5: Adding transitions induced by a modified action (precondition change). Given (i) a complete model $\mathcal{M}^*_A = \langle S^*_A, L^*_A, T^*_A \rangle$ induced by a set of actions $\mathbb{A}$; (ii) a partial model $\mathcal{M} = \langle S, L, T \rangle$ such that $\mathcal{M} \subseteq \mathcal{M}^*_A$ and (iii) s_i, $s_j \in S$, the corresponding updated model $\mathcal{M}' = \langle S', L', T' \rangle$ is obtained from $\mathcal{M}$ by adding a transition between states s_i and s_j labelled by action a_{new} which is a modified version of an action $a \in \mathbb{A}$ (where $pre(a)$ not satisfied in s_i needs to be relaxed), generating a new set of actions $\mathbb{A}'$. Formally:

- $\mathbb{A}' = (\mathbb{A} \setminus action(a, pre(a), pos(a))) \cup action(a_{new}, pre(a_{new}), pos(a_{new}))$, eff $\in pos(a)$, eff $\subseteq L(s_j)$, $pre(a_{new}) = L(s_i) \cap pre(a)$, $pos(a_{new}) = pos(a)$
- $\mathcal{M}^*_{A'} = \langle S^*_{A'}, L^*_{A'}, T^*_{A'} \rangle$ is a complete model induced by $\mathbb{A}'$, where $S^*_{A'} = S^*_A$, $L^*_{A'} = L^*_A$ and $T^*_{A'} = T^*_A \cup$

$\{(s_x, a_{new}, s_y)|s_x, s_y \in S^*_{A'}, pre(a_{new}) \subseteq L(s_x)$, eff $\in pos(a_{new})$, eff $\subseteq L(s_y)\}$

- $T' = T \cup \{(s_i, a_{new}, s) \in T^*_{A'}\}$
- $S' = S \cup \{s|(s_i, a_{new}, s) \in T'\}$
- $L'(s) = L^*_{A'}(s), s \in S'$

PUA6: Adding transitions induced by a modified action (postcondition change). Given (i) a complete model $\mathcal{M}^*_A = \langle S^*_A, L^*_A, T^*_A\rangle$ induced by a set of actions $\mathbb{A}$; (ii) a partial model $\mathcal{M} = \langle S, L, T\rangle$ such that $\mathcal{M} \subseteq \mathcal{M}^*_A$ and (iii) s_i, $s_j \in S$, the corresponding updated model $\mathcal{M}' = \langle S', L', T'\rangle$ is obtained from $\mathcal{M}$ by adding a transition between states s_i and s_j labelled by action a_{new} which is a modified version of an action $a \in \mathbb{A}$ (where $pre(a)$ is satisfied in s_i and we want to modify $pos(a0)$ to reach s_j), generating a new set of actions $\mathbb{A}'$. Formally:

- $\mathbb{A}' = (\mathbb{A} \setminus action(a, pre(a), pos(a))) \cup action(a_{new}, pre(a_{new}), pos(a_{new}))$, $pre(a) \subseteq L(s_i), pos(a_new) = pos(a) \cup L(s_j), pre(a_new) = pre(a)$
- $\mathcal{M}^*_{A'} = \langle S^*_{A'}, L^*_{A'}, T^*_{A'}\rangle$ is a complete model induced by $\mathbb{A}'$, where $S^*_{A'} = S^*_A$, $L^*_{A'} = L^*_A$ and $T^*_{A'} = T^*_A \cup \{(s_x, a_{new}, s_y)|s_x, s_y \in S^*_{A'}, pre(a_{new}) \subseteq L(s_x)$, eff $\in pos(a_{new})$, eff $\subseteq L(s_y)\}$
- $T' = T \cup \{(s_i, a_{new}, s) \in T^*_{A'}\}$
- $S' = S \cup \{s|(s_i, a_{new}, s) \in T'\}$
- $L'(s) = L^*_{A'}(s), s \in S'$

The principle of minimal change for $PUA5 - PUA6$ follows the same criterion we have defined for $PUA1 - PUA4$. Notice that $PUA5 - PUA6$ can update a model when a transition between two states s_i and s_j does not exist in the complete model $\mathcal{M}^*_A$, i.e., there is no action $a \in \mathbb{A}$ such that $pre(a) \subseteq L(s_i)$ and $pos(a) \subseteq L(s_j)$ and therefore the only way to update the model is by using the operations $PUA5 - PUA6$. However, when the primitive operations $PUA5 - PUA6$ are used to modify the actions in $\mathbb{A}$ they also imply in modifications on the induced complete model $\mathcal{M}^*_A$ which can eventually cause too many changes in the partial model. Therefore, considering all the primitive updating operations, $PUA1 - PUA6$, it is up to the minimal change criterion to suggest a set of minimal changes to the planning domain designer.

Conclusion

In this work we have presented a model updating approach that considers the actions behind the transitions in a state model. We also formalized the principle of minimal change for α-CTL logic - a previous proposed branching time temporal logic that has been applied to planning based on model checking (Pereira and de Barros 2008). To perform model updating in action (Figure 9), we take (i) a set of actions $\mathbb{A}$ (which is used to induce the complete labeled transition system $\mathcal{M}^*$), (ii) a partial model $\mathcal{M}$ such that $\mathcal{M}^* \subseteq \mathcal{M}^*$ and (iii) a α-CTL formula (i.e. a planning goal) and returns an updated model $\mathcal{M}'$, that has minimal change with respect to the original partial model (an example of plan specification or a more restrictive model induced by a preliminary set of actions).

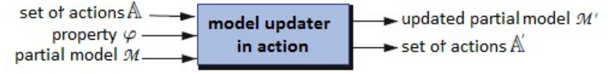


Figure 9: Model updater in action.

The minimal change principle proposed, as well the primitive operations $PUA1 - PUA4$ extended the work of Zhang and Ding (2008) to perform α-CTL model update. By using the description of actions, we proposed two extra primitive operations $PUA5 - PUA6$, that can be used to change an action specification (its precondition and effect, respectively). These two operations allow us to apply the proposed model updating as an important supporting tool for a planning domain designer.

As a future work we intend to implement an α-CTL model updater, based on our α-CTL model checker implementation and use it to refine some nondeterministic planning domain with complex goals. We also want to make experiments with new planning applications.

Acknowledgments. We thank CNPq and FAPESP (grant 2009/07039-4) for financial support.

References

Bryant, R. E. 1992. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys* 24(3):293–318.

Buccafurri, F.; Eiter, T.; Gottlob, G.; and Leone, N. 1999. Enhancing model checking in verification by AI techniques. *Artif. Intell.* 112(1-2):57–104.

Clarke, E. M., and Emerson, E. A. 1982. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, 52–71. London, UK: Springer-Verlag.

Clarke E., Grumberg O., P. D. 1999. *Model Checking*. San Francisco: MIT Press.

Harris, H., and Ryan, M. 2003. Theoretical foundations of updating systems. In *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings*, 291–294.

Kripke, S. 1963. Semantical considerations on modal logic. *Acta Philosophica Fennica* 16:83–94.

Müller-Olm, M.; Schmidt, D. A.; and Steffen, B. 1999. Model-checking: A tutorial introduction. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis*, 330–354. London, UK: Springer-Verlag.

Pereira, S. L., and de Barros, L. N. 2008. A logic-based agent that plans for extended reachability goals. *Autonomous Agents and Multi-Agent Systems* 16(3):327–344.

Zhang, Y., and Ding, Y. 2008. CTL model update for system modifications. *J. Artif. Int. Res.* 31(1):113–155.

Integrating plans into BPM technologies for Human-Centric Process Execution

Juan Fdez-Olivares and **Inmaculada Sánchez-Garzón** and **Arturo González-Ferrer** and **Luis Castillo**
Department of Computer Science and Artificial Intelligence
University of Granada

Abstract

This work presents a translation process from a standard representation of plans into a standard executable format for Business Process Management (BPM). This translation is conceived as a Knowledge Engineering for Planning process that bridges the existing gap between AI Planning and Busines Process Management and provides support for the direct execution of plans playing the role of Human-Centric processes.

Motivation

Human-Centric processes (Dayal, Hsu, and Ladin 2001) are collections of tasks, mainly organized in sequential and/or parallel control flows, which necessarily require human interaction in order to control and manage their execution. They are very common in any organization and they can be seen as complementary to System-Centric processes, which are devoted to exhaustively automate the data flow and processes of an organization, reducing human intervention to the minimum. This work is focused on a special kind of Human-Centric processes, concretely those oriented to *knowledge workers*(Myers et al. 2007): highly qualified personnel, like experts or decision makers, who need and produce knowledge in their daily work. These processes commonly support decisions and help to the accomplishment of workflow tasks in several application domains. Examples of such processes are a forest fire attack plan devoted to fire fighting technical staff, a medical treatment plan for a clinician, an e-learning course for a teacher, a military forces deployment plan for a commander, etc. For the sake of simplicity, we will designate these processes as *Smart Processes*.

The management and execution of *Smart Processes* demand special technological requirements, due to its main features(WorkflowManagementCoalition 2010): first, these processes respond to very complex, interacting sets of procedures and doctrine which reside either in an unstructured form in experts' mind or in partially structured documents, what makes difficult to generate and execute tasks in conformance with those constraints; second, they are unpredictable in the sense that both their composing tasks and order relations cannot be easily devised prior to their execution, since they strongly depend on the context of the organization and do not respond to a fixed pattern.

Hence they need to be somehow *modeled* and *dynamically generated*, their generation must be *adaptable to the context* of the organization and, finally, smart processes have to be *flexibly and interactively executed* by humans. In summary, they require some kind of intelligent management since they are very difficult to foresee and need to be adaptively generated depending on the context (current state) of an organization.

AI P&S has showed to be very suitable in many applications ((Fdez-Olivares et al. 2006; Castillo et al. 2007; Bresina et al. 2005; Fdez-Olivares et al. 2010)) as a technology that fulfills the above requirements. The role of AI P&S in this area, fundamentally HTN-based paradigms (Sacerdoti 1975; Castillo et al. 2006), is well known: starting from a planning domain where expert knowledge (in the form of actuation protocols or operating procedures) is *modeled* as a hierarchy of tasks networks, a plan (representing a course of actions to be accomplished at a given time) is *adaptively generated* as the result of a planning process. Then, the plan is *executed* by humans who, depending on the application may be ground operators, military personnel, experts in forest fire fighting, clinicians, etc., and this execution is supported by ad-hoc task visualization and execution models and tools (Fdez-Olivares et al. 2006; 2010).

On the other hand, a leading industrial area that has showed to be successful in the management and execution of Human-Centric processes is BPM (Business Process Management)(wfm), devoted to the modeling, deployment, execution and monitoring of business processes. From the concrete point of view of process execution, BPM technology provides: (1) *runtime engines* that support the execution of tasks based on robust task execution models, and (2) *visual consoles* (at present based on web portals) that support user interaction for the control of human-centric tasks. However, at the time being, BPM technology is mainly focused on static, repetitive, even perfectly predictable tasks/processes, mostly devoted to low qualification operators(WorkflowManagementCoalition 2010).

This is a widely known weakness in the BPM community (either industrial or academic) and, because of this, it is also recognized that new techniques must be devel-

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

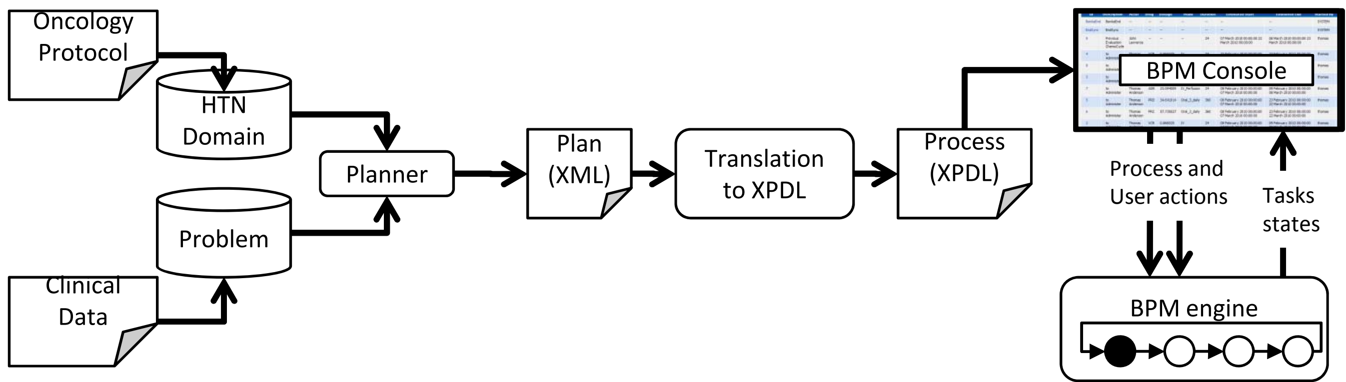


Figure 1: Integrating oncology treatment plans into both, a BPM console and a BPM engine

oped at the process modeling/generation step in order to fully cover the needs of knowledge workers on Smart Processes. With respect to AI P&S weak points, it is necessary to recognize that since most of the planning applications in Human-Centric processes are based on ad-hoc developments for task interactive execution (Bresina et al. 2005; Fdez-Olivares et al. 2006; Tate, Drabble, and Kirby 1994; Wilkins 1990), these developments are still far from being so stable, mature and usable like in BPM.

In summary, while AI P&S (concretely HTN paradigm) has proven to be successful on supporting the knowledge workers' *effort* (by modeling their expertise and helping them to adaptively produce plans to support their decisions), it can be seen that BPM is much more appropriate to support *the result* of this effort (by providing technological infrastructures in order to interactively execute and monitor processes). As a conclusion, it becomes relevant to analyze in what extent AI P&S technology might cover the lack of capability of BPM regarding the modeling and adaptive generation of plans. In addition, AI P&S may take advantage of an already tested, developed technology in order to enhance the user experience of a planning application at the execution and monitoring stage.

Consequently, this work faces the problem of integrating the capability of adaptive, dynamic generation of plans that we can find in AIP&S with the high-performance of BPM with respect to interactive execution of Human-centric processes. Concretely, we have interpreted the solution to this problem as the development of Knowledge Engineering techniques, focused on plan representation and postprocessing, in order to make the output of an AI planner understandable by a BPM runtime engine. The convergence of both technologies leads into an integrated environment for *Smart Process Management*, providing support for *modeling* (based on the representation of Hierarchical Task Networks), *adaptive generation* (based on Planning and Scheduling process) and *execution* (based on BPM runtime engines and consoles) of Smart Processes.

The adoption of this approach has many advantages for AIP&S in practical applications: the integration of a plan previously generated by a planning engine into an already developed, BPM standard environment for interactive execution of processes might support a rapid prototyping development life-cycle, saving development time at the first stages in the development of a AI P&S application. This also would allow to carry out a reliable acquisition of user requirements based on a rapid-prototyping methodology. In addition, user experience may be improved, helping to reduce/eliminate a constant bottle-neck in the adoption of AIP&S as a widely spread technology. From the BPM point of view, integrating a planner into its functional life-cycle, will leverage any BPM system allowing to fulfill all the requirements imposed by needs of knowledge workers.

In the following sections, in order to bring this arguments into reality, we will introduce a Knowledge Engineering approach based on the postprocessing and translation of plans into a BPM standard representations of processes. The result of this translation will be considered as the input of a BPM runtime engine that, highly coupled with a web console, will support the interactive execution of smart processes. In order to demonstrate the suitability of this approach, we have performed some experiments in the medical domain. Concretely we have achieved to execute, by using a commercial BPM runtime engine, pediatrics oncology therapy plans previously generated by a hierarchical planner. The therapy plans obtained by our planner are a clear example of what human-centric smart processes are, since they are primarily useful to support clinical decision making and need to be interactively executed by oncologists. Technical aspects of the plan representation and the translation process are detailed in last sections. Previously, the case study on therapy planning and some necessary background concepts on BPM are introduced.

Therapy planning case study

The work presented in this paper is being carried out in the framework of a research project aimed at developing a Clinical Decision Support System (called *OncoTheraper*), based on planning and scheduling techniques, in the pediatrics oncology area. OncoTheraper is intended to support oncologists's effort (they are the knowledge workers in this case study) when they deal with the problem of planning an on-

cology treatment for a given patient. These experts make their decisions following *oncology treatment protocols*, a set of evidence-based operating procedure and policies that are gathered in partially structured documents. The system is based on a temporally extended HTN paradigm (see (Fdez-Olivares et al. 2010) for more details) that, on the one hand, supports to model treatment protocols on the basis of an HTN temporal planning language. On the other hand, it allows to dynamically generate user-acceptable treatment plans, adapted to a context defined by a concrete patient profile, by following a planning process driven by the expert knowledge modeled in the planning domain.

In a previous experimentation, reported in (Fdez-Olivares et al. 2010; Fdez-Olivares, Czar, and Castillo 2009), a model of a concrete oncology clinical trial protocol (the one followed at present for planning the treatment of Hodgkin's disease and elaborated by the Spanish Society on Pediatrics Oncology) has been encoded in the temporally extended HTN planning language, following a knowledge elicitation process based on interviews with experts. This model contains knowledge about wokflow control structures included in the treatment protocol, temporal constraints to be observed between chemotherapy cycles, periodic patterns to administrate drugs as well as the representation of oncologists' working shifts.

In the experiments performed, the planner received the following inputs: a planning domain, representing this protocol; an initial state representing some basic information to describe a patient profile (age, sex, body surface, etc.) as well as other information needed to apply administration rules about drugs (dosage, frequency, etc.); and a high-level task representing the goal (apply the protocol to the patient) with temporal constraints representing the start date of the treatment plan. The output of the planner are plans that contain collections of (partially) ordered tasks representing drug administration actions to be accomplished on a patient. Since temporal information is crucial for oncology treatments, all the actions in a plan are temporally annotated with constraints on start and end dates which specify deadlines either for the estimated beginning and finalization of tasks. These plans are represented in a standard XML representation that allows to display them as Gantt charts in standard tools devoted to project management (like MS Project, see Figure 2)

Furthermore, OncoTheraper is also intended to support the execution of the treatment plan, that is, the *result* of the process followed by oncologists (now supported by the AI planning system) when planning a treatment. In the work here presented, we are exploring how the treatment plans, dynamically generated on the basis of medical knowledge, can be made executable in order to support the deployment and supervision, step by step, of all the planned treatment tasks. Clearly, the treatment plan generated by AI P&S techniques becomes a human-centric process and oncologists need a platform to visualize and interact with the plan generated, controlling the execution of the tasks defined in the treatment plan. Figure 1 illustrates this idea: since BPM consoles and runtime engines have shown to be successful in the execution of human-centric processes, it seems

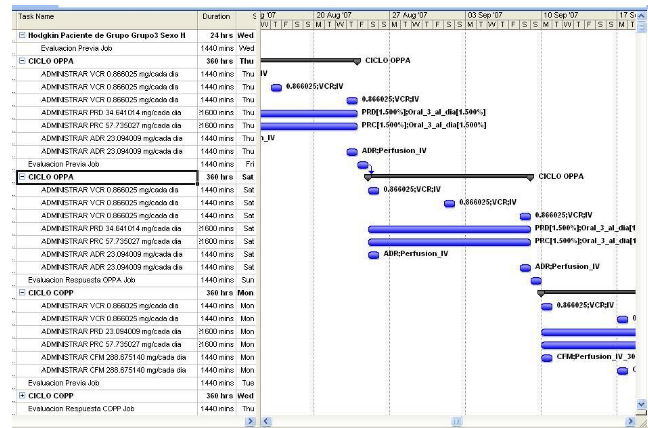


Figure 2: A temporally annotated and automatically generated therapy plan represented as a gantt chart. The plan represents the treatment for a patient following the Hodgkin's Disease Protocol. Start and end dates of every action are shown in the left-hand side. Drugs and their dosage are shown in the bars of the chart.

that a plan (in our example a treatment plan) adequately transformed into a standard business process representation, might be interactively executed by a standard/commercial runtime engine together with a BPM web console. In order to test this hypothesis we have considered the following steps:

1. The structure and content of plans generated by the hierarchical planner are postprocessed into an XML representation for plans.
2. The XML representation of plans has been translated into XPDL, a widely known standard representation of business processes. The result of this translation is the input to a BPM runtime engine.
3. The XPDL process is deployed into the BPM console in order to display its tasks and provide appropriate visual ”gadgets” to support the interactive control of the execution of tasks.
4. The execution of tasks is fully accomplished by a BPM runtime engine,following a task execution model based on a state-based automaton. The engine is also in charge of capturing user actions, sent by the console, an changing accordingly the states of the tasks. New states of tasks are sent back and visualized into the BPM console.

From a Knowledge Engineering for Planning point of view, the translation of a plan into a different model raises the question of which categories of information a plan should contain in order to be executable by a standard BPM engine. Next section is devoted to clarify this question, since an important part of the answer comes from the analysis of the information model of the target language of the translation process, as well as from the analysis of the execution model carried out by the runtime engine and from the requirements about visualization and interaction of the BPM console.

Technological Environment

A great part of the ideas developed in this work need to know in some extent the terminology, standard languages and concepts involved in the area of BPM. A business process is a collection of activities with order relations and control structures that define their execution flow. Processes are defined using a standard BPMN notation, through a Business Modeling tool. The result of such definition takes the form of a XPDL file. XPDL (*XML Process Definition Language*) is a standard language (wfm) , based on XML, devoted to promote process exchange between BPM engines. A BPM runtime engine is in charge of executing a process, usually represented in XPDL, by following the execution flow described. Since human-centric processes require the interaction of human users during its execution, most commercial runtime engines have also coupled a web console that support user interaction. The most relevant XPDL entities and attributes considered in our work are:

Activities. They comprise a logical, self-contained unit of work, which will be carried out by *participants* and/or computer applications. Activities are related to one another via *transitions*. **Transitions**. They result in the sequential or parallel operation of individual activities. **Participants**. They are used to define the organizational model over which a process is deployed and can be allocated to one or more activities.**Parameters and DataFields**. These entities are used to define the process data model. Information that is internal to the process is represented as *Data Fields* and information required outside the process is represented by *Parameters*.

In addition the information model of any XPDL entity may be extended by using *Extended Attributes*.

BPM runtime engines

Id	Description	Actor	Drug	Estimated Start	Estimated End	State	Actions
5	to Administer	Thomas Anderson	PRD	08 February 2010 00:00:00 07 March 2010 00:00:00	23 February 2010 00:00:00 22 March 2010 00:00:00		
6	to Administer	Thomas Anderson	PRC	08 February 2010 00:00:00 07 March 2010 00:00:00	23 February 2010 00:00:00 22 March 2010 00:00:00		
2	to Administer	Thomas Anderson	VCR	08 February 2010 00:00:00 07 March 2010 00:00:00	09 February 2010 00:00:00 08 March 2010 00:00:00		
7	to Administer	Thomas Anderson	ADR	08 February 2010 00:00:00 07 March 2010 00:00:00	09 February 2010 00:00:00 08 March 2010 00:00:00		
1	Previous Evaluation ChemoCycle	Thomas Anderson	--	07 February 2010 00:00:00 25 February 2010 00:00:00	08 February 2010 00:00:00 26 February 2010 00:00:00		
BonitaInit	BonitaInit	--	--	--	--		

Done

Figure 3: A typical BPM console showing a collection of tasks. For each one temporal information, execution state and execution controls are shown.

Most BPM systems include three main components: a Business Process Modeler (a tool oriented to IT professionals who visually design a business process), a BPM Runtime Engine (in charge of executing the activities represented in a XPDL process that, as said above, is a serialization of the business process visually designed) and a BPM console. From the user point of view, the BPM console is the most important component and it is closely related with the runtime engine, being responsible of (1) providing user interaction in order to deploy a business process previously defined by the Business process Modeler, (2) visualizing the process activities to be carried out, and (3) providing visual "gadgets" to interactively control the execution of process activities. Figure 3 shows a snapshot of the console used in the experiments of this work.[1]

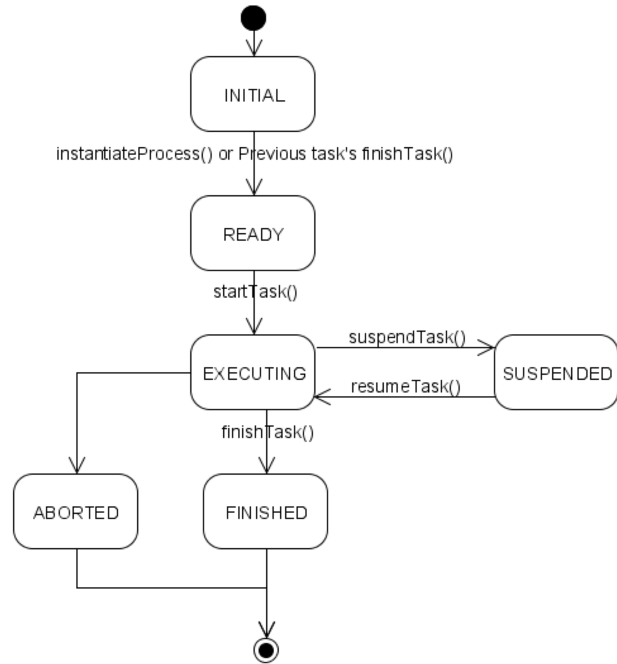


Figure 4: A BPM runtime execution model for Human-Centric tasks.

Regarding process execution, BPM engines are commonly endowed with the necessary machinery in order to execute every task in a process following an execution model based on state-based automata. Fig 4 illustrates the states and transitions of the automaton underlying task execution for the engine that we have chosen to test our concepts. It follows a standard life-cycle for task execution, and similar ones can be found in the literature for BPM engines as well as for clinical plans execution. Thus, there is not lost of generality on the concepts here explained and they can be applied to another different BPM engine. As shown int the figure, the engine allows to start, finish, suspend or abort any task, always upon user request. A task may be in a READY state if all its previously ordered tasks have been finished. Then, it can be started by the user and the engine changes its state to EXECUTING. At this point a task may change

[1]The console chosen for experimental purposes is Nova Bonita console that also includes the Nova Bonita runtime engine (http://www.bonitasoft.com/). Besides that both are Open Source projects and accept XPDL as input, the main reason for selecting these tools is that they support the interactive execution of tasks based on a configurable, simple yet expressive execution model.

either to SUSPENDED, ABORTED or FINISHED, depending on user actions. Every change of state has associated a *trigger* (a java method) that can be customized by the developer. This provides support to define the behaviour of the engine as required by users. Furthermore, triggers opens the possibility of communicating the engine with external systems like a plan monitoring service: the monitor may receive information on critical changes in the execution of a process and respond to them accordingly, for example raising a re-planning process.

Finally, on the basis on this execution model, the basic principles of an acceptable execution of plans can be obtained, with the following considerations:

1. Though most BPM engines support the execution of processes with conditional and repetitive control structures, due to the nature of clinical treatment plans, only sequential and parallel control structures are addressed.
2. BPM engines do not provide full support for the execution of processes with tasks incorporating time constraints(Gagné and Trudel 2009). Indeed, the engine used in this work is only capable of directly manage deadlines for the termination of tasks. This is a really weak point that forces to develop special monitoring services to provide full temporal information management like, for example rescheduling of dates upon user request. Therefore, the full treatment of temporal constraints falls out of the scope of this paper.

Next the XML plan representation used in this approach is explained.

Plan representation

The plans generated by the planner are represented in XML as collection of *Task nodes* (see Figure 5) where every Task node contains information about:

- Activities (*id* and *name*) and their parameters (*type*, *name* and the *value* assigned at planning time), its preconditions and effects.
- Temporal information of activities (*earliest Start* and *earliest End*), representing the estimated time (obtained at planning time) for the start and end of every task in the plan (start, end, duration). Indeed, the plan obtained includes richer temporal information, since it is deployed over a temporal constraint network that assigns to every task a start and end time points represented as time intervals with the earliest and latest start and end dates at which an action is allowed to be executed ($[a_{earlieststart}, a_{lateststart}]$ and $[a_{earliestend}, a_{latestend}]$). However, the information represented in the xml plan is enough, given the above explained execution model.
- Order dependencies. Every action a contains a collection of order dependencies, one for each action b ordered before a, which allow to establish sequential an parallel runtime control structures. This is a crucial item since its analysis will lead to inform the runtime engine about the set of actions that are immediately ordered after a given one.
- Metadata, which allow to represent additional knowledge required by either the user, the console or the runtime execution engine. They are syntactic structures that are managed (created, assigned, etc.) at planning time, and are intended to be interpreted by external systems. Indeed, they are the keystone to enrich a plan in order to facilitate plan postprocessing steps and integration with other systems. For a given action, the meta-data field is a collection of items of the form: `<metadata <name> <valuetype> <value> >`. The plan representation used in this approach embodies the following metadata:

  Description, a string containing user-friendly information about the task.

  Type, used to represent whether the execution of an action necessarily requires human intervention to be initiated ($Type = Manual$) or might be initiated by the engine ($Type = Auto$). This is a very common categorization of actions in Human-Centric processes.

  Actor, considering that we are focused on human-centric processes, it is mandatory that one of the parameters be considered as the resource (either a person or a system) that accomplishses the action.

  Performer, its value is the participant in the process in charge of executing the action from the BPM console.

Meta-data are a very convenient way to encode information that is not directly related with the reasoning process, but which is strongly required in practical applications. Furthermore, they can also be used to extend the knowledge model of actions with additional items requires for a given domain. Indeed, meta-data are encoded in the planning domain, as special tags associated to actions. It is important to note that they can be encoded independently from the BPM processes intended to be executed, as it will be shown in next sections.

Translating plans into XPDL

The translation process is focused mainly in transforming the following pieces of knowledge from a XML plan: activities and its parameters, temporal information, order dependencies and metadata. This knowledge is enough to generate a human-centric process that can be fully executed by a user through a standard BPM engine, since with this knowledge it is possible to generate information in order for the execution model to manage order relations, either sequential or parallel, of tasks as well as execution deadlines. The translation process has three main steps:

1. **Generation of XPDL DataFields/Participants.**The goal of this step is to generate the data model used by the runtime engine. Basically, consists on translating the objects hierarchy, their properties and initial values (defined in the initial state) into XPDL *DataFields*. This step is at present subject to further analysis, in the experiments this has been done semi-automatically.
2. **Generation of XPDL Activities.** For each action a_{plan} in the XML representation of plan, a XPDL activity a_{xpdl} is generated with the following information items:

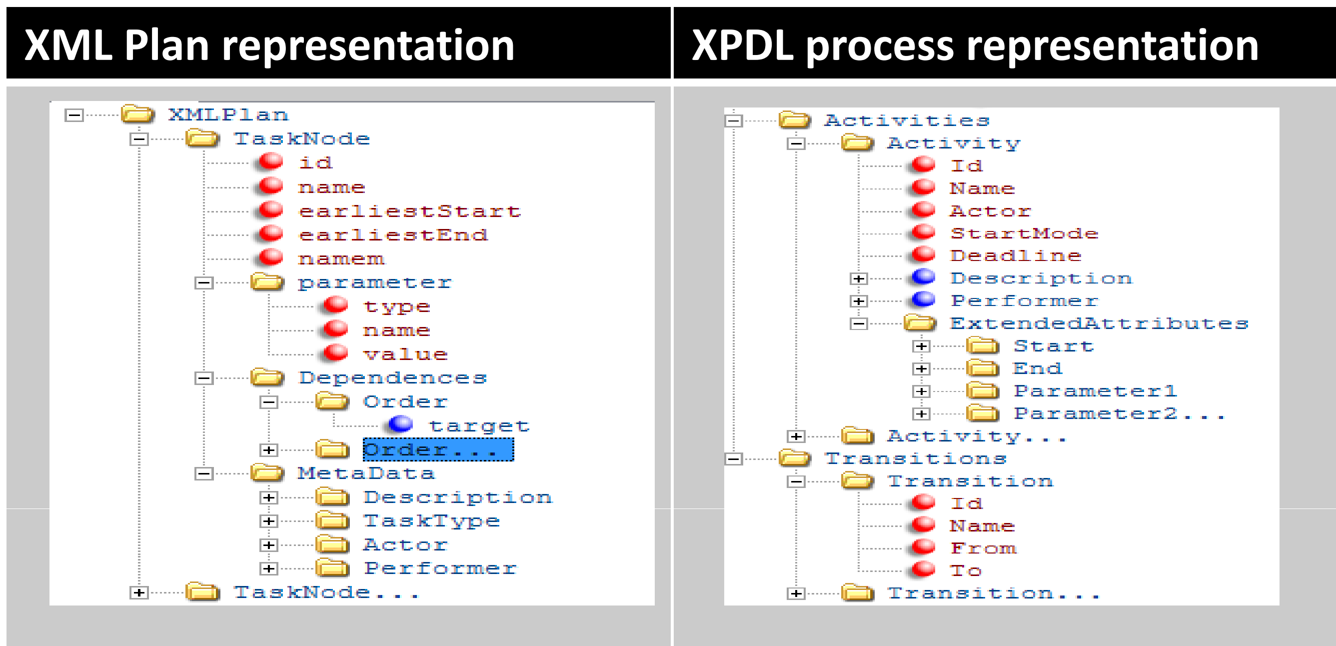


Figure 5: An XML structure of a plan and the XPDL schema of the relevant information for activities and transitions used in this work.

- $a_{xpdl}.name = a_{plan}.name$
- $a_{xpdl}.performer = a_{plan}.metadata.Performer$
- $a_{xpdl}.participant = a_{plan}.metadata.Actor$
- $a_{xpdl}.description = a_{plan}.metadata.Description$
- $a_{xpdl}.startmode = a_{plan}.metadata.Type$
- $a_{xpdl}.deadline = a_{plan}.duration$
- Start and end dates of a_{plan} are translated as XPDL *extended attributes* and added to the properties of a_{xpdl} `<ExtendedAttribute name=`*start* `value=`$start_{aplan}$`>` `<ExtendedAttribute name=`*end* `value=`end_{aplan}`>`
- Every paramenter p_i of a_{plan} is translated as a XPDL *extended attribute* of the form `<ExtendedAttribute name=`$p_i.name$ `value=`$p_i.value$`>`

3. **Generation of XPDL Transitions.** This step contains two stages:

(a) For each a_{plan} generate $SUCC(a_{plan})$ as the set of immediate succesors of a_{plan}. A task j is an immediate succesor of another task i when a dependence $i < j$ exists and there is no task k such that $i < k$ and $k < j$.

(b) For every a_{plan} and for each b in $SUCC(a_{plan})$, generate an XPDL transition `<Transition From=`a_{plan} `To=`b`>`

As said before, the XPDL so generated is given as input to the BPM console that deploys it upon user request. The information contained in the XPDL file is used by both, the console in order to show task items required by the user, and the runtime engine in order to execute tasks according to the information stored in the XPDL process. Thus, an XPDL file contains *informative-only* fields which are the following: *name, participant, description, parameters, and estimated start and end times*. These fields are usually showed in the console for user information purposes. *Operative fields* needed by the execution model are: *startmode* (used to determine whether a tasks starts automatically or upon user request), *deadline* (used to manage whether a task has finished correctly on time), *performer* (used to determine if the user is allowed to execute a given actions) and the *transitions* (used to determine the execution order of tasks).

The life-cycle of Smart Process Management

The translation process above introduced is the keystone of a process that allows to achieve a full connection between the output of an AI Planning system and the input of a BPM runtime engine. The convergence of both technologies leads into an integrated environment for *Smart Process Management*, providing support for *modeling* (based on the representation of Hierarchical Task Networks), *adaptive generation* (based on Planning and Scheduling process) and *execution* (based on BPM runtime engines and consoles) of Smart Processes.

Authors argue that this can be considered a contribution in the field of Knowledge Engineering for Planning due to the following reasons: first, it is a non-trivial transformation between a plan representation and a widely spread, standard language for business processes. Second, it also answers some questions about new information requirements that must be covered by domains and plans representations when they have to deal with plans that must be executed by using standard BPM technologies. The new pieces of knowledge like type of actions, actors and performers, are

not usually considered as part of a planning model, but it has been shown that they necessarily have to be incorporated in domain and plan representations.

Furthermore, the field of Knowledge Engineering for Planning also deals with the study and development of techniques and methodologies that might advance the life-cycle for engineering planning systems. In this sense, the transformation process above described bridges a common, important gap that prevented to adequately carry out a fast prototyping strategy for developing practical planning applications since, in order to develop a first prototype, it was mandatory to develop also ad-hoc execution monitoring processes and underlying preliminary interfaces, in order to convince users about the advantages of the system. Therefore, under this circumstances, building a first prototype of planning application is very costly in time and human resources.

As opposite, we have carried out a proof of concept based on the translation of a treatment plan previously generated by the planner and its execution based on the console. The plans obtained are then transformed into XPDL processes, following the translation process above described and interactively executed by oncologists on a BPM console. Therefore, the development effort in building a first prototype for oncologists has been reduced. Instead of fully developing both a specific interface for user interaction and a execution monitoring system to support the execution, we have studied and used the configuration techniques provided in the user manuals of both the console and runtime engine. The result, from the oncologist point of view, is a web application, obtained in few weeks, that provides both information about the treatment tasks to be performed, their time constraints and interaction to start/finish/suspend/abort tasks.

Figure 3 shows the configured interface which has as default behaviour to show information about the pending treatment tasks to be executed. The information in the console is structured as a table where each row shows task information that may be divided into three blocks:

1. Information about its name, its parameters and its estimated start and end dates (recall that XPDL does not provide specific fields to represent task parameters, nor start and end dates, but this information is extracted from *extended attributes* as detailed above).
2. Information about its state, shown in the form of flags.
3. Active buttons to control the execution of tasks. These controls are easily configurable, and include buttons to start, finish and suspend a task.

As the execution of the process progress, new pending task are added to the console. In addition, a basic time management at execution can be achieved, based on the capability of representing deadlines for tasks, as explained above.

Oncologists are highly skilled knowledge workers, but it is understandable the they have not a clear idea about what are their real usability needs with respect to a challenging and novel Clinical Decision Support System. Therefore the basic functionality above described is enough to capture user requirements, on the basis of this prototype, that would be almost impossible to detect on interviews-based knowledge/requirements acquisition.

Apart from Knowledge Engineering for Planning, the connection between a planner and a BPM runtime engine through the translation process introduced has advantages in the field of BPM. Mainly, it contributes to leverage the BPM life cycle incorporating capabilities for dynamic generation of emergent processes. As said in the introduction of this paper, BPM engines are oriented to execute processes the execution flow of which is completely defined a priori, and there is no place to the management of adaptive, context dependent process generation. The experimental proof of concept carried out shows that, starting from a ”smart process model” represented by an HTN planning domain, it is possible generate processes (originally plans and then translated into business processes) that vary on its execution flow depending on a variable context, defined by a patient profile. Then, these processes can be executed on the basis of standard BPM execution models.

Related work

The relationships between workflows (or business processes) and AI planning have been studied from different perspectives, but it is relevant for this work when considered as a technique to directly generate workflows executed by ad-hoc, application specific systems devoted to the interactive execution and monitoring of plans considered as workflows. For this last case, some works are oriented to autonomic computing (Srivastava, Vanhatalo, and Koehler 2005), others oriented to grid computing(Deelman et al. 2004), and we can also find works devoted to support knowledge workers in their daily work, in the form of intelligent task management assistants(Myers et al. 2007). Again, these approaches do not face the execution of plans, seen as smart processes, by using standard BPM technologies. AI P&S techniques have also been applied in the field of worflow generation for semantic web services. In this application area, AI planning techniques are mainly focused on the automated generation of sequences of semantic web services calls (that may be seen as semantic business processes)(S.A., T.C., and H. 2001). Some approaches in this field (P. and M. 2004) address the generation of BPEL code from plans representing web services processes. These approaches are focused on the management of System-Centric processes and, due to the nature of these processes, do not address the interactive execution of processes.

Regarding the concrete BPM field, the concepts described in this work are subject of study under the denomination of *Adaptive Case Management*(WorkflowManagementCoalition 2010). Nowadays this is an emergent concept in BPM. It tries to analyze and explore either already developed or new promising techniques, susceptible to be integrated into present BPM systems, in order to fulfill the requirements imposed by knowledge worker processes.

Conclusions

This work should be considered as a step forward in the pursuit of a methodology for rapid prototyping in Knowledge

Engineering for Planning. The main contribution consists on the integration of AIP&S techniques and BPM technologies through a process that translates plans into executable business processes, thus allowing to directly execute those plans into BPM standard runtime engines. From the point of view of a BPM runtime engine, the information model of plans, represented in XML, contains enough information to be directly executed what allows a fully automated translation process.

With respect to the flexibility of this approach, it is important to note that, although most of the knowledge embodied by plans obtained by any state-of-art planner can be reused by BPM runtime engines, without the transformation process presented, the plans obtained could not be directly executed in BPM engines. Therefore, for any planning system, it is mandatory to transform both, the structure of the plans and their content, as already explained. In this sense, whenever the plans obtained by another state-of-art planner fits to the plan representation here presented, it will always be possible to use our translation process in order to transform the plans into XPDL and then execute them on a BPM runtime engine. However, it is necessary to say that the domain model must be able to represent "special tags" for actions, in order to finally obtain plans containing all the information required for execution. This technique is not new in planning, and many planners, specially HTN planners (Myers et al. 2007), allow to introduce additional knowledge in the model of actions for postprocessing purposes. Under these considerations, authors argue that it is possible obtain a fully automated process that leads from domain modeling to human-centric business processes execution.

However the approach here introduced presents some weak points that need to be deeply studied. The translation of the planning domain object model is not completely addressed, and it must be faced in order to achieve a fully automated translation process. Precondition and effects management is neither addressed. Although it is possible to achieve a user acceptable execution of plans, this functionality is only permitted for prototype-level versions. The development of a full application requires to develop a complete execution monitoring based on the causal rationale of plans. Finally, a full treatment of temporal constraints at execution time is needed. All these issues are being faced at present and will be incrementally added to the current approach, according to the user requirements analyzed on the basis of this first prototype.

Acknowledgements

This work has been partially supported by the Andalusian Regional Ministry of Innovation under project P08-TIC-3572.

References

Bresina, J. L.; Jonsson, A. K.; Morris, P.; and Rajan, K. 2005. Activity planning for the mars exploration rovers. In *Proceedings of the ICAPS05*, 40–49.

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.; Fdez-Olivares, J.; Garca-Prez, O.; Garzón, T.; and Palao, F. 2007. Reducing the impact of ai planning on end users. In *ICAPS 2007, Workshop on Moving Planning and Scheduling Systems into the Real World*, 40–49.

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*.

Deelman, E.; Blythe, J.; Gil, Y.; Kesselman, C.; Mehta, G.; Vahi, K.; Blackburn, K.; Lazzarini, A.; Arbree, A.; Cavanaugh, R.; and Koranda, S. 2004. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing* 1:25–39.

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.; Cozar, J.; and Garcia-Perez, O. 2010. Supporting clinical processes and decisions by hierarchical planning and scheduling. *Computational Intelligence* To Appear.

Fdez-Olivares, J.; Czar, J.; and Castillo, L. 2009. *Knowledge Management for Health Care Procedures*, volume 5626 of *Lecture Notes on Computer Science*. Springer. chapter OncoTheraper: Clinical Decision Support for Oncology Therapy Planning Based on Temporal Hierarchical Tasks Networks, 25–41.

Gagné, D., and Trudel, A. 2009. "Time-BPMN". In *Proceedings of 1st International Workshop on BPMN*.

Myers, K.; Berry, P.; Blythe, J.; Conley, K.; Gervasio, M.; McGuinness, D.; Morley, D.; Pfeffer, A.; Pollack, M.; and Tambe, M. 2007. An intelligent personal assistant for task and time management. *AI Magazine* 28(2).

P., T., and M., P. 2004. Automated composition of semantic web services into executable processes. In *International Semantic Web Conference*.

S.A., M.; T.C., S.; and H., Z. 2001. Semantic web services. *IEEE Intelligent Systems* 2(16):46–53.

Sacerdoti, E. D. 1975. The nonlinear nature of plans. In *Proceedings of IJCAI 1975*, 206–214.

Srivastava, B.; Vanhatalo, J.; and Koehler, J. 2005. "Managing the Life Cycle of Plans". In *17th Innovative Applications of Artificial Intelligence Conference*, 1569–1575. AAAI Press.

Tate, A.; Drabble, B.; and Kirby, R. 1994. O-PLAN2: An open architecture for command, planning and control. In Zweben, M., and Fox, M., eds., *Intelligent scheduling*. Morgan Kaufmann.

Workflow management coalition. http://www.wfmc.org/.

Wilkins, D. E. 1990. Can AI planners solve practical problems? *Computational intelligence* 6:232–246.

WorkflowManagementCoalition. 2010. http://www.xpdl.org/nugen/p/adaptive-case-management/public.htm. Group on Adaptive Case Management.

Improving Planning Performance Through Post-Design Analysis

Tiago Stegun Vaquero[1,2] and **José Reinaldo Silva**[1] and **J. Christopher Beck**[2]
[1]Department of Mechatronic Engineering, University of São Paulo, Brazil
[2]Department of Mechanical & Industrial Engineering, University of Toronto, Canada
tiago.vaquero@poli.usp.br, reinaldo@usp.br, jcb@mie.utoronto.ca

Abstract

In this paper, we investigate how knowledge acquired during a plan analysis phase that follows model design affects planning performance. We describe a post-design framework that combines a knowledge engineering tool and a virtual prototyping environment for the analysis and simulation of plans. Our framework demonstrates that post-design analysis supports the discovery of missing requirements and guides the model refinement cycle. We present two case studies using benchmark domains and eight state-of-the-art planners. Our results demonstrate that significant improvements in plan quality and an increase in planning speed of up to three orders of magnitude can be achieved through a careful post-design process. We argue that such a process is critical for deployment of planning technology in real-world applications.

Introduction

Over the last decade, both research effort and industry interest have been directed towards the application of AI Planning techniques to solve real-life problems. As a result, it has become clear that the process of developing algorithms for synthesizing plans forms only one part of the complex design life cycle of a real-world planning application. Most of the problems identified as suitable to being solved with a planning approach are characterized by a need for substantial knowledge management, reasoning about actions and a careful consideration of quality metrics and criteria. The design process of real applications must have a strong commitment to these prerequisites in order to result in reliable, deployed planning systems.

Design 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 quality solutions. From a knowledge engineering perspective, lack of knowledge, ill-defined requirements and inappropriate definition of quality metrics 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 the former perspective, in which new algorithms are developed and tuned to obtain high performance and better plans. Not much investigation has been done on the knowledge engineering (KE) perspective, especially remodeling 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 and use of such human-centered feedback is still an unexplored area in the knowledge engineering for AI planning. Moreover, the extent of impact of such feedback and re-modeling on the planning performance is unknown. In order to deal with such post-design analysis, techniques such as simulation, visualization and virtual prototyping, commonly used in other disciplines (Cecil and Kanchanapiboon 2007), can help design teams identify new requirements and inconsistencies in the model.

In this paper, we present a post-design tool for AI planning that combines the open-source KE tool itSIMPLE (Vaquero et al. 2007) and a virtual prototyping environment to support identification of inconsistencies and hidden requirements. We describe two case studies showing that post-design not only improves plan quality, but also improves planning performance even in benchmark problems. The main contributions of this work are:

1. The creation of a framework to support post-design analysis for planning;
2. Two case studies that demonstrate that improvements in plan quality, an increase in solvability and a reduction of planning time of up to three orders of magnitude can be achieved through a careful post-design process.

This paper is organized as follows. First, we discuss concepts in knowledge engineering for planning and their role in plan analysis and post-design. Then, we present the post-DAM project describing the integration of itSIMPLE and a virtual prototyping tool. Next, we present two case studies and the results. We conclude with a discussion of our results.

Knowledge Engineering and Post-Design

Requirements engineering (RE) and knowledge engineering (KE) principles have become important to the success of the design and maintenance of real world planning applications.

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

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. 2009b; 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 and execution. 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 and investigate the solutions they generate. Some of the AI planning research on plan analysis has developed tools and techniques for plan animation (McCluskey and Simpson 2006; Vaquero et al. 2007), visualization (e.g. Gantt charts), and plan querying and summarization (Myers 2006). However, such work does not explore the effects of the missing knowledge and the re-modeling loop in the planning process. The investigation of modern analysis techniques such as simulation for planning it is still an emerging field.

The postDAM Project

The postDAM project aims to investigate post-design techniques to enhance the modeling cycle and increase the quality of plans. The project focuses on combining some of the recently developed tools in KE for planning with virtual prototyping. Virtual prototyping is commonly used in other engineering fields (e.g. mechanical engineering) to validate models and identify missing requirements (Cecil and Kanchanapiboon 2007) where producing a real prototype is impractical and costly.

The project proposes a framework that integrates the KE tool, itSIMPLE (Vaquero et al. 2009b), and the 3D content creation environment, Blender[1], for virtual prototyping. The former is a robust design system dedicated to AI planning in which a set of languages and validation engines are used to create domain models in a disciplined design process. The latter is an open source tool, widely used for creating games and animations. Blender provides several mechanisms for the definition and simulation of 3D elements, including their physical properties (such as mass, collision, gravity, inertia, velocity, strength, and sound effects), to mimic some real world characteristics.

The integration of these tools aims to close the design loop, from requirements acquisition to plan analysis in the post-design. In this loop, itSIMPLE is responsible for supporting users during design and re-design of the models while Blender, properly integrated with the KE tool, encompasses the simulation of plans provided by planners.

During model construction in itSIMPLE, designers perform the initial design phases to develop domain models in the *Unified Modeling Language*. An important step in this design process is the identification and specification of quality metrics, along with their respective importance. These quality metrics are characterized in the form of weighted domain or plan variables, i.e., numeric variables that are directly (or indirectly) related to the quality of plans. Examples of domain and plan variables are: the number of occurrences of a specific action in a plan; the total consumption of fuel; the number of robots used for a particular purpose; and the energy remaining in a battery. The definition of quality metrics in itSIMPLE uses the approach described in (Rabideau, Engelhardt, and Chien 2000). During simulation, the designer can analyze and evaluate different solutions while contrasting them based on the quality metrics. Different viewpoints can communicate during 3D visualization in order to validate and adjust the model based on their impressions. Figure 1 illustrates this iterative refinement process.

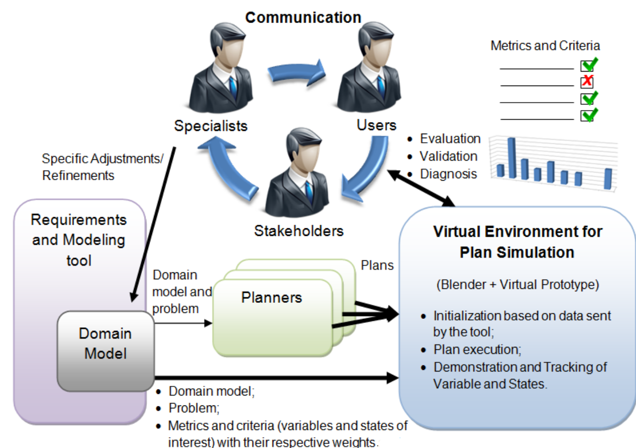


Figure 1: The post-design framework

In order to provide an integrated design iteration, a communication channel between itSIMPLE and Blender was developed in which data is sent from the KE tool to the 3D

[1]Blender, available at www.blender.org

environment. The data sent by itSIMPLE consist of the domain model, the problem instance and the quality metrics to be considered (all in an XML representation (Vaquero et al. 2009b)). Since users can run several state-of-the-art planner from the itSIMPLE's GUI, the generated plans are sent directly to the 3D simulator.

Blender reads the data from itSIMPLE and generates a *virtual prototype* of the model based on a predefined library of graphical objects and their physics. These objects are designed in such a way that they can perform and react to the actions defined in the model from itSIMPLE. The Blender application reads the main elements of the domain and problem instance such as classes of objects, the objects and their properties, and additional information regarding the graphical position of the elements that has been stated by the user. Classes are used to identify the necessary graphical elements from the predefined library, while objects, properties and location information are used to instantiate and initiate the graphical elements in the initial scene. All the elements in the problem instance definition are found in the 3D representation. Having the initial state established in the 3D scene, the plan provided by a planner is then simulated (we assume that plan actions are deterministic). In the simulation, the actions are sent to each involved object, step-by-step. Each object is implemented to act based on the instructions that it is given. In each step of the simulation, the values of the metrics are stored to provide a clear view of their changes over the plan.

At the end of simulation, Blender 3D produces a plan report that can be analyzed by users. The report contains the evolution of the chosen quality metrics along with the cost of the plan.

Case Studies

In this section, we present two case studies using benchmark domains from the *International Planning Competitions* (IPC). The domains are the Gold Miner domain from IPC-6 and the Storage domain from IPC-5. Both were chosen from recent competitions based on the clear correspondence between objects in the real and virtual world.

The procedure used for each case study is as follows:

1. We created an initial model in itSIMPLE guided by the original PDDL representation to simulate the design process. Since itSIMPLE generates PDDL output as a communication language to planners, we verify that such output is exactly the same as the original PDDL version of the benchmark domain. This model is called $Original$.
2. We selected three problem instances from the 30 IPC instances to be analyzed in-depth. The selected set of problem instances is called the *design set*. Eight planners were chosen to be run (using default arguments) with a 20-minute time-out for each problem instance.
3. In addition to the PDDL reproduction process, we used itSIMPLE to define quality metrics for the domain.
4. Using the virtual prototype in Blender, we studied every generated plan and its execution. With the analysis and plan reports, we manually introduced modifications to the model in itSIMPLE. We repeated the plan simulation and model refinement, going back and forth with new ideas and results, until we had two new models (A and B) each representing one major change and a third new model (AB) incorporating them both.
5. We then took the remaining 27 problem instances and tested all four models ($Original$, A, B, and AB) with the eight planners. We call this set of instances the *testing set*.

In order to analyze the impact of the refinement cycle, we compare the models A, B and AB to the $Original$ over all 27 problem instances from the testing set. This analysis considers the changes on plan quality, plan length, speed, and solvability. PDDL terms and elements are used to describe the adjustments made to the original model in the remodeling process to facilitate the explanation.

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[2] in which the robot must (1) get the laser, (2) shoot through the rocks (either soft of 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.

For our design set, we chose three problem instances considering the variety of the number of objects and difficulty. The first (gold-miner-target-5x5-02) and the second (gold-miner-target-5x5-01) instances have a 5x5 mine with distinct positions of gold, bombs, cannon, and soft and hard rocks. The third instance (gold-miner-target-6x6-05) has a 6x6 mine also with a particular position of the domain elements.

The quality metrics chosen for this study are (1) the travel distance of the robot (weight 2), (2) the bomb usage (weight 1), and (3) the laser cannon usage (weight 1). In itSIMPLE, we specified these metrics as counters of the actions *move*, *detonatebomb*, and *firelaser*, respectively. For this domain we selected SGPlan5, MIPS-xxl 2006, LPG-td, MIPS-xxl 2008, SGPlan6, Metric-FF, LPG 1.2, and hspsp to solve the problem instances.

During the first post-design analysis with the original model and the design set, we carefully investigated all 24 generated plans through the 3D simulation using Blender. Figure 2 shows an example of the simulation.

[2]IPC-6 2008. http://eecs.oregonstate.edu/ipc-learn/

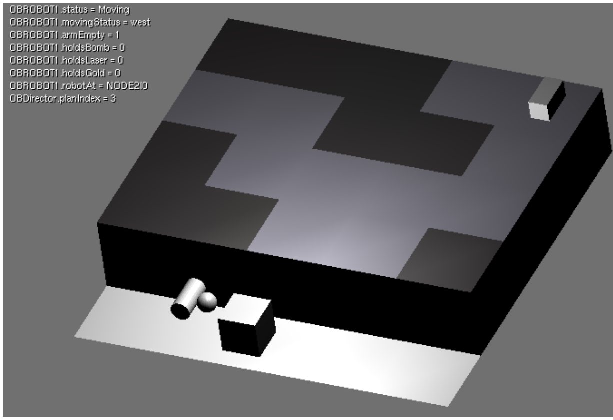


Figure 2: Virtual prototype and simulation of the Gold Miner domain. The robot is represented as a cube in the bottom. Soft and hard rocks are light and dark gray areas respectively. We used basic shapes to represent objects, however, there is no restriction on using complex 3D shapes and skins.

A number of observations were made in the first analysis:

- One planner generated invalid solutions in which the robot used the laser at the gold location, destroying the gold.[3]
- Some planners provided (valid) plans in which the laser cannon was fired at an already clear location.
- Unnecessary move actions were present in some plans.

In order to fix these non-optimal and flawed behaviors, we refined the original model. Concerning the planner assigning the robot to fire at the gold, the original model does not prevent such situation: there was no precondition on the *firelaser* operator that explicitly constrains this behavior. Therefore, a precondition to the operator was added: *(not (gold-at ?loc))*. Regarding the unnecessary *firelaser* occurrences, a second precondition was added to the same *firelaser* operator, in this case *(not (clear ?loc))*. We call this set of modifications A. The resulting model was sent to the planners to solve the same problem instances, resulting in a new post-design iteration.

During the second analysis process, additional observation were collected:

- Invalid plans were no longer being generated.
- The undesirable firing behavior from the initial observation was eliminated.
- In most of the plans, at the goal state, the laser cannon was left in a different position from the initial one. As a new requirement, the robot could leave the laser only at the same spot as the bomb source.

[3]This was observed with MIPS-xxl 2008. We contacted the authors and the bug appears to arise from the way we called the planner (e.g. a special script is necessary) but due to time limitations, we were unable to re-run these instances with the correct script.

- Unnecessary move actions were still being found in some solutions.

These new observations guided the second re-modeling loop. A precondition to the *putdownlaser* operator was added, forcing the robot to always drop the cannon at the location of the bombs. The precondition *(bomb-at ?loc)* was used for this purpose. We call this modification B. The plans generated with model B properly controlled the location where the cannon was left. The combination of the previous and the current set of adjustments is called AB.

As a result of the AB modification, most of the generated plans to the design set converged to the same solution. The main issues raised during post-design analysis were eliminated, except some unneeded move actions.

To analyze and compare the effects of the post-design analysis on planning performance, the $Original$ model was compared to the models A, B, and AB. The selected planners were run on the testing set for each of the four models. Table 1 illustrates the comparison concerning run-time (speed) and solvability on the testing set. The table shows the total time (including time-outs) for each planner to solve all 27 problem instances in each of the four models. In order to determine the speed-up values, we first define $t^M_{p,k}$ as the time planner p takes to solve problem instance k using model M. We then define the speed-up ratio for each of the new models (A, B, AB) compared to the $Original$ model as follows:

$$r^M_{p,k} = \frac{t^{Original}_{p,k}}{t^M_{p,k}}. \qquad (1)$$

For a particular planner and model, we calculate the mean and the median of the speed-up ratios. The mean speed-up value presented in Table 1 for each model is the mean of means of speed-up ratios over all planners. Similarly, the median speed-up is the median of the medians of speed-up ratios over all problem instances and planners for each model. Model AB shows a significant speed-up compared to the $Original$. Even A and B individually provide a significant speed-up. The maximum speed-up ratios observed on an individual problem instance were 7,547 with model A, 5,068 with model B and 5,185 with model AB. However, we also observed that in some cases the new models were slower than the original. The lowest ratio observed was 0.26 with model B. Considering the total time to solve the testing set, all planners perform better in the AB model. Because MIPS-xxl 2008 was run improperly it is not considered in the analysis showed in Table 1.

Table 1 also illustrates the number of instances solved by the planners in each model, as well as the percentage improvement of each new model compared to the original. All problem instances were solved in the AB model, a 22.7% improvement compared to the original model.

Improvements are not only found on speed and solvability, but also on plan length and quality. By looking at each problem instance and the four plans generated by a particular planner, we can determine the model that gives the best performance on three criteria: run time, plan length and plan quality (cost). Table 2 illustrates the number of times each

Time (s)					Problems Solved				
Planners	Original	A	B	AB	Planners	Original	A	B	AB
SGPlan5	3,303.42	1.93	3.65	1.99	SGPlan5	14	27	27	27
MIPS-xxl 06	3,995.34	983.79	2,015.28	22.13	MIPS-xxl 06	15	25	21	27
LPG-td	41.04	29.69	39.28	27.83	LPG-td	27	27	27	27
SGPlan6	3,925.85	3.13	4.65	3.51	SGPlan6	18	27	27	27
Metric-FF	1,707.00	3.79	4.35	3.78	Metric-FF	26	27	27	27
LPG 1.2	10.58	5.58	4.47	4.70	LPG 1.2	27	27	27	27
hspsp	37.53	16.77	8.22	8.19	hspsp	27	27	27	27
Mean Speed-Up		469.70	267.97	479.67	Total	154	187	183	189
Median Speed-Up		7.33	6.82	10.54	Improvement		21.4%	18.8%	22.7%

Table 1: Total time (including time-outs) required to solve all problem instances and the solvability comparison for the Gold Miner domain models.

	Best Time Occurrence				Best Plan Length Occurrence				Best Plan Quality Occurrence			
Planners	Original	A	B	AB	Original	A	B	AB	Original	A	B	AB
SGPlan5	0	10	5	13	14	13	24	13	14	17	24	17
MIPS-xxl 06	0	4	0	23	11	19	21	25	11	19	21	25
LPG-td	2	10	2	13	5	24	10	27	5	24	10	27
SGPlan6	0	19	2	7	17	19	24	19	17	20	24	20
Metric-FF	0	9	9	10	26	13	23	13	26	17	23	17
LPG 1.2	3	3	10	12	12	15	18	12	15	17	19	12
hspsp	0	0	19	8	27	27	27	27	27	27	27	27
Total	5	55	47	86	112	130	147	136	115	141	148	145

Table 2: Best time, plan length and plan quality comparison over the Gold Miner models.

model results in the best solution with respect to each of the three criteria for a given planner. For example, LPG-td generated the best plan lengths compared to the other models using LPG-td in 5 cases with the $Original$ model, 24 with model A, 10 with model B, and 27 with model AB. The numbers sum to greater than 27 due to ties. Better plans are usually found with refined models.

The Storage Domain

The Storage domain is one of the benchmark domains from the deterministic track of IPC-5 (2006). This domain involves moving a certain number of crates from containers to depots using hoists. Inside a depot, each hoist can move according to a specified spatial map connecting different areas of the depot, represented by a grid. Transit areas are used to connect depot areas to containers and also depots to depots. The domain has five actions: (1) lifting a crate with a hoist; (2) dropping a crate from a hoist; (3) moving a hoist into a depot; (4) moving a hoist from one area of a depot to another; and (5) moving a hoist outside a depot. At the beginning of each problem, all crates are inside the containers waiting to be transported to the necessary depot. For this case study we used the propositional version of the PDDL domain model.

The design set for this domain is composed of three problem instances with different numbers of elements and difficulty. In the first instance (p10), four crates must be allocated in one depot using a single hoist. In the second (p16), six crates must be carried from two containers into two depots by three available hoists. The third instance (p20) has ten crates stored in three containers, three depots, and three hoists.

The quality metrics specified for this domain are the numbers of occurrence of each operator in the plan: move (weight 2), lift (weight 1), drop (weight 1), go-out (weight 3); and go-in (weight 3). The weights used in this experiment were inspired by the PDDL numeric version of the domain. We selected the same planners used in the previous case study, except LPG 1.2 which was removed due to a bug identified while running the design set. Instead we used FF 2.3.

During the first post-design iteration, the plans for the design set were analyzed in the virtual prototype platform. Figure 3 shows the simulation of the first problem instance from the design set.

The main observation raised while analyzing the plans provided by planners were:

- In some solutions, the hoists were putting the crates back into the containers. This scenario happens mainly when hoists left other crates on the main access to the depots, blocking access. Such situation forced the hoist into a lift-drop loop in and out of the containers.
- The fact that hoists drop crates on the doorway of depots forced them to rearrange the crates, which culminated in unnecessary actions to correct the previous decisions.
- Some of the plans included unnecessary lifts and drops of the same crate. This lift-drop loop happened usually in

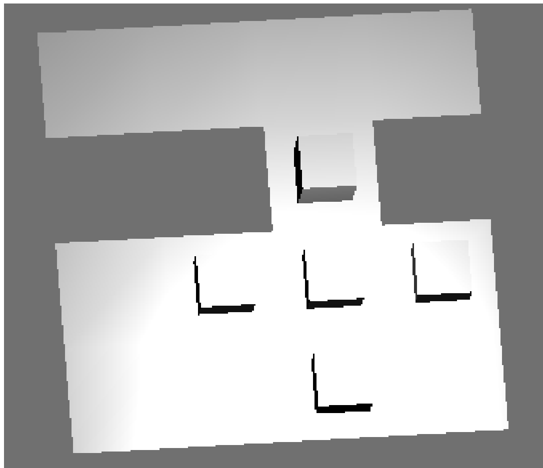


Figure 3: Virtual prototype and simulation of the Storage domain. Three crates at the depot and a hoist on the transit area.

transit and depot areas.

- Some solutions contained unnecessary move actions.

We focused on the fact that crates were being placed back into the containers. In order to constrain that situation, a simple modification of the parameters of operator *drop* was made. The original operator has the following PDDL representation *(?h - hoist ?c - crate ?a1 - storearea ?a2 - area* ***?p - place****)*. The adjusted version of the *drop* operator differs from the original in the last parameter: ***?p - depot***. The parameter *p* constrains the location where crates can be dropped. We call this modification A.

During the second iteration of simulation with adjustment A, the following observation were acquired:

- Crates were no longer being put back into the containers.
- Some of the plans still contained unnecessary lifts and drops of the same crate, but fewer than the original model.
- The unneeded move actions still occurs in some plans.

In the second re-modeling iteration, we focused on the unnecessary lift and drop cycles of the same crate. The approach taken for this problem was to make the hoist memorize the last lifted crate. For that, a new predicate was specified *(lastLifted ?h - hoist ?c - crate)*. This predicate was added to the precondition of the *lift* operator in the following form *(not (lastLifted ?h - hoist ?c - crate))*, as well as in the post-condition as *(lastLifted ?h - hoist ?c - crate)* to record the current crate. The precondition constrained the planner not to assign a hoist to lift the same crate again. The added post-condition defines the previously lifted crate. It is important to note that this is just one approach to attempt to tackle the issue. More elaborate re-modeling could be performed. We call this individual modification B. The combination of the refinements A and B generated the model AB. Note that A, B and AB do not require modification of the problem instances, even with the new predicate in B, since the hoists start with no previous lifted crates.

By analyzing the model AB in the post-design framework, the unnecessary lift-drop loops of the same crate were reduced drastically. However, some plans exhibited the problem in a different form: two hoists alternatively lift and drop the same crate. The occurrence of such situations was rare. The unnecessary move actions were reduced but not eliminated.

Table 3 shows the impact on run-time and solvability when running the selected planners over the testing set with the original and the new models. A and B are again analyzed separately for a better view of individual effects. In this table, the speed-up is not as impressive as the first case study. The maximum speed-up ratios observed for a single problem instance were 4,411 with model A, 1,597 with model B and 5,194 with model AB. The minimal ratio observed was 0.01 with model B. We do not have a significant improvement in the solvability with the new models. The highest one is the AB with a 9.4% improvement on the total number of problems solved. As above, the results of MIPS-xxl 2008 are not included due to the use of an incorrect script.

Table 4 illustrates in which model each planner provides it best run time, plan length and plan quality. For example, SGPlan6 provided the best plan lengths on 4 problem instances with the original model; 10 with model A, 17 with model B, and 18 with model AB. Similarly to the previous case study, the table shows that better plans are found more often using the refined models.

Discussion

In this section we present some of the main discussions raised by the case studies.

Knowledge Acquisition and Extraction

The case studies presented above demonstrate that even in benchmark domains missing requirements and modeling issues emerge in the post-design analysis. In real planning applications, we expect such gaps to be very common due to the difficulties of obtaining the necessary knowledge and requirements. The knowledge acquisition process in real-world applications is not the pure collection of already existing requirements during the beginning an application design (Studer, Benjamins, and Fensel 1998). Tacit knowledge and hidden and unknown requirements must be discovered and considered. Therefore, knowledge must be built up and structured during an iterative design process, especially during the initial phases and after design. Domain modeling is an iterative process in which new observations may lead to a refinement of the already built-up model (Studer, Benjamins, and Fensel 1998), even over time; moreover, the model itself may guide the further acquisition of knowledge.

In this work, both the KE tool and the 3D simulation environment have an important role in the discovery of missing requirements and the refinement cycle. The use of virtual prototyping, in particular, has shown to be a powerful technique on plan validation and new requirements identification as opposed to looking at plan traces. Visual and sound effects can give experts and non-experts a clear view of the domain model as well as the planning strategy. The

Time (s)					Problems Solved				
Planners	Original	A	B	AB	Planners	Original	A	B	AB
SGPlan5	11,562.25	10,137.26	7,686.23	7,470.18	SGPlan5	18	19	21	21
MIPS-xxl 06	3,433.82	3,334.95	3,913.46	3,347.97	MIPS-xxl 06	16	16	14	16
LPG-td	5,980.77	7,124.50	5,796.66	4,775.16	LPG-td	25	26	25	26
SGPlan6	8,230.74	10,177.03	7,754.42	7,508.96	SGPlan6	18	19	21	21
Metric-FF	13,441.49	9,767.35	13,205.63	12,003.99	Metric-FF	16	19	16	17
FF 2.3	14,416.93	9,935.53	15,603.44	12,117.64	FF 2.3	15	19	16	17
hspsp	21,782.20	21,744.39	21,117.63	20,989.91	hspsp	9	9	10	10
Speed-Up mean		37.22	13.03	43.49	Total	117	127	123	128
Speed-Up median		1.15	1.00	1.04	Improvement		8.5%	5.1%	9.4%

Table 3: Total time (including time-outs) required to solve all problem instances and the solvability comparison for the Storage domain models.

	Best Time Occurrence				Best Plan Length Occurrence				Best Plan Quality Occurrence			
Planners	Original	A	B	AB	Original	A	B	AB	Original	A	B	AB
SGPlan5	2	8	3	8	13	13	16	16	13	13	16	16
MIPS-xxl 06	2	7	1	6	13	14	12	13	10	13	13	13
LPG-td	5	6	4	11	9	9	17	18	9	9	15	18
SGPlan6	0	11	2	9	4	10	17	18	4	10	17	18
Metric-FF	1	10	3	9	15	16	10	12	14	17	10	12
LPG 1.2	2	12	1	5	12	15	14	17	13	15	14	16
hspsp	1	4	1	4	9	9	10	10	9	9	10	10
Total	13	58	15	52	75	86	96	104	72	86	95	103

Table 4: Best time, plan length and plan quality comparison on the Storage domain.

KE tool was also essential in the process, especially in the re-modeling phases. A metric-focused analysis, using for example the plan report, helps the designer to determine the subset of high quality solutions as well as the proper set of quality criteria.

Another important factor on discovering a lack of knowledge in the model and hidden requirements is the presence of different levels of quality over the analyzed plans. The identification of bad plans, for example, proved to be a powerful guidance on the re-modeling process. Bad plans not only raise the need for new constraints on the model, but also help designers to capture user's feedback and preferences. In our case studies, the generation of distinct plan qualities was enhanced by using a variety of planners. Since the lack of knowledge in the model can impact differently on the planners, their different responses also contribute to the identification of model issues. After the adjustment process these different responses are narrowed as many of the plans converge to the same solution over different planners.

We observed that planners can be very sensitive to the presence or absence of specific knowledge in the model. As an example, in the Gold Miner domain, the adjustment cycle made some of the planners perform impressively better; however, in the Storage domain, the addition of knowledge negatively affected the planners' internal heuristics. In fact, adding missing constraints not necessarily implies in faster responses from the planners; however even with a higher run-time we are moving toward better plans. These facts suggest that IPC results could be different if such issues were considered.

Modeling and Planning

The case studies showed that the planning performance indeed improved with a post-design analysis. We achieved speed-ups through a careful plan analysis and re-modeling process, without changing or adjusting planners. In some cases we have added obvious knowledge, from a human perspective, to the model; however, its explicit representation facilitates the search process of the planners. This evidence reinforces that both aspects, model and planner, must be carefully designed and refined. A sole emphasis on improving planners, neglecting observations and feedback from the design process itself, can *de facto* prevent or constrain planning use in real applications.

We believe that our results represent a challenge for practical planning research. The central justification for building general-purpose planners is that domain experts cannot be expected to also be planning experts. Domain experts should be able to concentrate on modeling the domain, treating the solver as a black-box. In practice, therefore, the *only* options available to a domain expert are domain and problem re-modeling. The use of planning technology by someone who is not a planning expert therefore depends entirely on the extent to which domains can be modeled (and remodeled) to allow planning algorithms to achieve satisfactory performance. Yet, we know very little about how domain modifications affect planning algorithms and we can provide little advice to domain experts (without becoming domain

experts ourselves) on what changes are likely to be positive.[4] Tools, such as the one presented here, to allow domain experts to investigate changes and planning experts to begin to develop an understanding of the impact of changes on their algorithms are therefore critical.

There is a fundamental mismatch between the target domains of the postDAM framework (i.e., real-world planning applications) and the case studies using IPC domains. A true test of our tool should be in the form of a case study with a real problem (e.g., (Cesta et al. 2008; Jónsson 2009; Vaquero et al. 2009a)). However, such case studies are not reproducible and often rely for success on external factors: significant interest from the client, success of other parts of the mission (e.g., landing on Mars), and larger economic forces. System building research is extremely valuable but sometimes inaccessible and difficult to generalize. Research in AI planning has shifted toward a more empirical style since the beginning of the IPC, where research innovations can be reproduced and directly compared. This style, too, has substantial benefits as can be observed from the gains in solver performance. The design of our case studies was an attempt to bridge the gap between "real" applications and "academic" benchmarks and to encourage further research on modeling in planning. We have shown that even in benchmark domains that, by definition, do not include a wealth of unrepresented knowledge, it is still possible to substantially increase solver performance by domain re-modeling.

Conclusion

In this paper, we have described a post-design framework to assist the discovery of missing requirements and to guide the model refinement cycle. We have demonstrated that following a careful post-design analysis, we can improve not only plan quality but also solvability and planner speed. The modifications made through the observations acquired during post-design resulted in impressive speed-up of state-of-the-art planners. In a real planning application, the analysis process that follows design becomes essential for having the necessary knowledge represented in the model. Post-design analysis is critical for deployment of planning technology in real-world applications.

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.

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). Sydney, Autralia.*

Jónsson, A. K. 2009. Practical planning. In *ICAPS 2009 Practical Planning & Scheduling Tutorial. Greece, Thessaloniki.* Available at: http://icaps09.uom.gr/tutorials/tutorials.htm.

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. Cumbria, UK.*

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.*

Myers, K. L. 2006. Metatheoretic plan summarization and comparison. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS-06).* AAAI Press.

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.*

Simpson, R. M. 2007. Structural domain definition using GIPO IV. In *Proceedings of the Second International Competition on Knowledge Engineering. Providence, Rhode Island, USA.*

Studer, R.; Benjamins, V. R.; and Fensel, D. 1998. Knowledge engineering: Principles and methods. *Data and Knowledge Engineering* 25:161–197.

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.*

Vaquero, T. S.; Sette, F.; Silva, J. R.; and Beck, J. C. 2009a. Planning and scheduling of crude oil distribution in a petroleum plant. In *Proceedings of ICAPS 2009 Scheduling and Planning Application woRKshop. Thessaloniki, Greece.*

Vaquero, T. S.; Silva, J. R.; Ferreira, M.; Tonidandel, F.; and Beck, J. C. 2009b. From requirements and analysis to PDDL in itSIMPLE3.0. In *Proceedings of the Third ICKEPS, ICAPS 2009, Thessaloniki, Greece.*

[4] We believe that the fact that the planners used in our experiments all tended to improve their performance is an indication that such advice may exist.

An XML-based Forward-Compatible Framework for Planning System Extensions and Domain Problem Specification

Eric Cesar E. Vidal, Jr. and Alexander Nareyek

NUS Games Lab
Interactive and Digital Media Institute, National University of Singapore
21 Heng Mui Keng Terrace, Level 2, Singapore 119613
ericvidal@nus.edu.sg, elean@nus.edu.sg

Abstract

Real-world planning problems, e.g., planning for virtual characters in computer games, typically come with a set of very specific domain constraints that may require specialized processing, like symbolic path planning, numerical attributes, etc. These specific application requirements make it necessary for planning systems to have an extensible design. We present a framework for a planning system that recognizes *planning extensions* (such as new data types or structures, sensing/acting functionality, and others). The framework is designed to be forward-compatible, exposing an XML-based domain language that allows current and future problems that use such planning extensions to be properly specified.

Introduction

In artificial intelligence, *planning* is a problem where, given a set of goals and possible actions, the necessary actions are to be determined (along with their proper temporal arrangement) to attain the given goals. A logistics problem, for example, can define a set of possible actions, such as "move a vehicle V from location A to location B" or "load/unload package P in vehicle V", and a set of goals such as "deliver n packages, labeled $P_{1..n}$ from locations $A_{1..n}$ to locations $B_{1..n}$". A solution is called a *plan*; in this case, the plan would involve multiple "move" and "load/unload" actions on a correctly-ordered, non-conflicting schedule. A plan is considered *valid* when it reaches the goal state without inconsistencies such as violations of action preconditions (e.g., moving a package requires the package to be loaded first) or forbidden overlapping of actions (e.g., a package cannot be unloaded while the vehicle is moving). A *planning system* or *planner* is a program that automatically creates such plans.

Many planning problems are simple enough such that a general (i.e., *domain-independent*) planning system is not needed, e.g., path planning in most computer games is usually implemented as a simple A* search. On the other hand, more complex problems (e.g., planning a dynamically-generated story for a computer game) can benefit from the solving capabilities of a general planning system. In order to do this, the properties of the problem must be formalized into a *domain definition*, using a specification language that can be understood by a planner.

To solve real-world problems, however, a general planning system usually needs to be extended to handle specific application requirements. A computer game, for example, will require extensions such as *online planning* (i.e., feeding the planner-selected actions into the game, and then sensing in real time the current state of the game world, updating the plan accordingly), *numerical resources* such as player health or money, and *specialized solving heuristics* to let the planner more efficiently handle specific sub-problems like symbolic path planning (where symbols are mapped to actual positions in the world, for faster planner reasoning about connectivity and distances compared to regular path planning). A planner written without such extensibility in mind will invariably need continuous re-design to handle these and future extensions. A better solution, from a software engineering point of view, is to adhere to a framework that readily integrates such extensions, making a planner *forward-compatible* with current and future planning problems.

Many such extensions may expose new planning constructs—for example, online planning introduces the concept of *actuators* and *sensors* into the domain ontology—thus making it necessary for the extensible planner architecture to tie in seamlessly with its domain specification language.

Background

In this section, we introduce the issues that accompany the design of an extensible planning architecture, and issues related to the domain specification of planning extensions.

This work was supported by the Singapore National Research Foundation Interactive Digital Media R&D Program under research grant NRF2007IDM-IDM002-051.

Monolithic versus General-Search-based Planning

We first discuss existing planning approaches to establish the context of our planner extensibility problem. Different planning approaches vary in the degree they can be extended.

Systems such as STRIPS (Fikes and Nilsson 1971) and Graphplan (Blum and Furst 1997) are *monolithic* systems. Although these systems are extensible to a certain degree, these systems use relatively rigid planning frameworks that are often optimized to exploit a particular problem representation and are not specifically designed with extensions in mind. Thus, the possibility of extending such systems ranges from impractical to impossible.

Planners that map to *general search frameworks* like propositional satisfiability (SAT), integer linear programming (ILP) or constraint programming (CP) can usually handle planning extensions much more easily, although they are often not as expressive as monolithic approaches for specific domains. SAT-based systems, such as Blackbox (Kautz and Selman 1998) and SatPlan-2006 (Kautz, Selman, and Hoffman 2006), can handle Boolean propositions, which somewhat limits the types of problems that can be expressed. ILP-based planners, such as LPSAT (Wolfman and Weld 1999), take into account numerical resources but are restricted to linear inequality constraints. CP-based planners, such as CPLAN (van Beek and Chen 1999) and the EXCALIBUR agent's planning system (Nareyek 2001), can theoretically handle more general constraints. See Nareyek et al. (2005) for a more detailed discussion.

A fully-extensible planning architecture should be able to handle flexible planning problem constructions such as the general search frameworks described above (including future refinements to these frameworks), while retaining the domain-specific expressiveness found in monolithic systems.

PDDL and Planner Extensibility

There are many available planning systems, often using very different internal representations of planning domains. The Planning Domain Definition Language (McDermott et al. 1998) was conceived to enable standardized comparisons and competitions between planning engines. PDDL solves a critical problem by exposing an *extensible language* to introduce new features to a planning system's model—by default, it recognizes STRIPS-style actions, but it also recognizes feature extensions such as conditional effects, hierarchical actions, durative actions and numerical reasoning (Fox and Long 2003), and as of version 3.0, preferences and soft constraints intended for CP planners (Gerevini and Long 2005). The `requirements` tag of PDDL invokes these extensions, which, in turn, change parts of the language's definition.

However, since PDDL is designed as a common language intended for academic planning competitions, it has distinct disadvantages in real-world applications. PDDL was conceived during a time when monolithic planners with STRIPS-like constructions were the norm, and the extensions were added stepwise as new planning paradigms were introduced. Consequently, these extension constructs, including but not limited to the simplified treatment of resource properties, durative actions, nonlinear numerical projections and unknown information, have been subject to criticism (Boddy 2003). Additionally, there is no direct way to expose sensing and actuating interfaces to the outside world, which is a requirement for specifying online planning problems. While a planner can add new or improved constructs in its private implementation of PDDL, this would result in the proliferation of non-standard extensions that are incompatible across planning systems. It may be possible to standardize certain extensions (PDDL versions 2.1 and above are indeed targeted towards providing standard extensions); however, as PDDL's intent is to provide a common interface that is not necessarily efficient nor sufficiently expressive (for example, continuous numerical effects in PDDL are assumed to be linear, making non-linear continuous functions hard to express), strict adherence to the language will impose an artificial restriction on a planning system's capabilities and will limit extensibility.

Furthermore, PDDL's `requirements`–based extensibility is *not a solution to support real-world applications*. As mentioned earlier in the Introduction, real-world applications using a planning system need to extend that planning system according to their special requirements by providing their own custom modules (e.g., new data types, new heuristics, or custom sensors and actuators). Ideally, external users (application developers or even third-party vendors) should be able to add new constructs to the planning problem definition without the domain modeler needing to recompile the planning system or its problem definition parser. This functionality is inherently absent from PDDL as it was intended to be an academic tool, with little consideration for a professional or industrial environment.

This paper proposes a solution to these problems by presenting a general planning system framework based on the Extensible Markup Language (XML), allowing a simple, modular way to extend the planning system and its model. The goal is to create a *pluggable system of planning extensions that neatly tie into the representation language*. Efficiency is not the main focus (although a clean problem representation that directly corresponds to the planning system's internal structure will naturally be more efficient than a poorly-fitting PDDL representation); rather, a planning system implementing our framework will be "future-proof", with a vast potential for new planning extensions to extend the capabilities of planning beyond what is currently being explored in academic circles.

The next section introduces an example scenario where extensions are needed, followed by a discussion of the framework itself, and the extension possibilities it allows.

An Example Scenario for Extensible Planning

The Crackpot planning system will be used throughout this paper as an example to show how our proposed framework can be implemented by a typical planning system. This section contains a brief introduction to Crackpot, along with a sample problem to be tackled by this planner.

The Example Planner

Crackpot[1] is the successor of the EXCALIBUR agent's planning system (Nareyek 2001). As such, it uses the same principle of local search based on iterative repair to make and improve plans—a plan with inconsistencies or *costs* (e.g., unmet goals, mutually-exclusive actions that overlap, unmet preconditions for an existing action, etc.) is iteratively improved by using one of several *repair* heuristics (e.g., add a new action, move an action's start/end times, etc.). Crackpot is intended to be an *online* planner, where agents other than itself might change the state of the world as time passes, and actions can only be added to the plan at positions at or beyond the current time.

Crackpot, as with most planners, separates the notion of a general *domain* from a specific *problem* of the domain, allowing modelers to create separate specifications of each. Crackpot internally models a domain/problem using an object-oriented design amenable to implementation in C++. Figure 1 depicts the relationships between Crackpot's domain specification constructs using a UML (OMG 2010) class diagram.

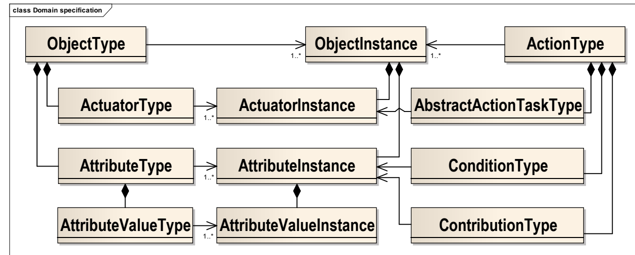


Figure 1. Crackpot's domain specification class structure. (This specification is a work-in-progress.)

In general, a distinction is made between the *type* and *instance* of a particular construct (e.g., ObjectType vs. ObjectInstance): the abstract types (e.g., the "person" type) appear in the domain specification, while the grounded instances (e.g., a "person" named "Joe") appear in the problem specification. (It is the planning system's task to create action instances and their components; hence the related classes are not shown in the above diagram.)

A domain *object* (e.g., a person) contains state resource variables called *attributes* (e.g., walking speed) whose projections over time can consist of one or more *attribute values* (e.g., 0.0 m/s at the start, 1.0 m/s at the end). An object also contains action resource variables called *actuators* (e.g., legs).

An *action* (e.g., walking from one place to another) is made up of *object parameters* (technically, "parameter object instances") specifying which objects are related to the action (e.g., which person is doing the walking), *conditions* on the object parameters' attributes that must be met for the action to execute (e.g., the person must be at the start location), *contributions* of the action to the attributes (e.g., the person ends up in the target location), and *action tasks* indicating how actuators are used throughout an action (e.g., a person uses his legs to walk).

Note that this design roughly maps to the EXCALIBUR planning system's model. In particular, actuators, attributes, and conditions/contributions directly map to action resource constraints, state resource constraints, and task constraints. (Nareyek 1998)

Because of efficiency reasons, Crackpot is *not* a completely-modular system, taking a middle ground between monolithic and fully-modular planning systems. Its design currently assumes a fixed flow of the planner's execution cycle (find a repairable cost in the plan, repair the cost, repeat). Third-party extensibility of the planner itself is currently restricted to introducing specialized attribute value types (e.g., a SymbolicLocation type extended from the provided Symbolic type to allow for path planning, or perhaps a collection-oriented Set type).

However, the next version of Crackpot that is currently being worked on will increase the expressiveness of domains, by introducing a *control flow actuator* (Nareyek 2003) to allow changes in the planning execution cycle (e.g., temporarily focusing on specific plan repairs), a *cost management system* to allow domain-influenced selection of the repair heuristic by specifying modifiers for each available cost (e.g., to cause the planner to prefer adding certain actions over others), *action-component relations* that specify additional constraints between an action's object parameters and other action-related parameters (e.g., to set the duration of the action according to the value of a resource), and *read-ins* that take in attribute values and feed them into action-component relations. These extension possibilities must be taken into account when designing a representation language for Crackpot.

A Sample Problem with Extension Possibilities

Consider a very simple planning problem: A person is currently in his living room, and he is hungry. Given the following scenario, what must he do to satiate his hunger?

- There is an apple in the kitchen. Conveniently, he can travel from one place to another by walking.
- The way to the kitchen is separated by a closed door. The only way to overcome this formidable obstacle is to open it with his hands.

To model the *domain* of this problem, a modeler may use Crackpot's constructs in the following manner. (The

[1] Crackpot is a work-in-progress planner available at the following URL: http://sourceforge.net/projects/crackpot

problem specification is omitted to save space, but the initial state and goals may be inferred from the above description.)

ObjectType: **Person**
AttributeType: **Hungry** { *true, false* }
AttributeType: **Location** { *livingroom, door, kitchen* }
ActuatorType: **Legs**
ActuatorType: **Hands**
ObjectType: **Apple**
AttributeType: **Existing** { *true, false* }
ObjectType: **Door**
AttributeType: **Open** { *true, false* }
ActionType: **EatApple**
Parameters: { p : Person, a : Apple }
Conditions: { p.Hungry = *true*, p.Location = *kitchen*,
a.Existing = *true* }
Contributions: { p.Hungry = *false*, a.Existing = *false* }
ActionTask: *uses* p.Hands
ActionType: **WalkFromLivingRoomToDoor**
Parameter: { p : Person }
Condition: { p.Location = *livingroom* }
Contribution: { p.Location = *door* }
ActionTask: *uses* p.Legs
ActionType: **WalkFromDoorToKitchen**
Parameters: { p : Person, d : Door }
Conditions: { p.Location = *door*, d.Open = *true* }
Contribution: { p.Location = *kitchen* }
ActionTask: *uses* p.Legs
ActionType: **OpenDoor**
Parameters: { p : Person, d : Door }
Conditions: { d.Open = *false*, p.Location = *door* }
Contribution: { d.Open = *true* }
ActionTask: *uses* p.Hands

The above representation is adequate for a planning domain with relatively simplistic assumptions. True enough, it is also possible to create a PDDL description out of this domain with predicates and actions (each with parameters, preconditions and effects), all but with a slight loss of fidelity to the modeler's intent; for example, the concept of actuators are lost in the translation. (The PDDL version is not shown here, again due to space constraints.) However, suppose that this is part of a more sophisticated computer game world, where a non-player character (NPC) agent has a relatively simple behavioral AI such as that described above (in order to tell a simple story, for example). Some problems with the above domain representation immediately become clear:

1. Game worlds, more often than not, have a running game clock, so actions don't occur instantaneously but are executed over certain durations.
2. Agent attributes such as hunger (or generally, health) in computer games are, more often than not, modeled as numerical resources that rise and fall over time, not just Boolean values as assumed here.
3. It is inadequate to specify game world locations as plain symbols. Without information about each location's actual Cartesian coordinates and its connectivity with other locations, this representation does not scale well to a real path-planning problem (as the current form requires many "walk" actions to be defined between each connected location).
4. The door's actual state may change irrespective of the agent's interactions—if a player closed the door immediately after our simplistic agent opened it, the agent will suddenly not be able to pass the "real" door in the game (although the agent thinks it has), nor would it know that it needs to re-open the door, unless the planner is notified of the change.

All these problems stem from a lack of expressiveness in the domain. What we need in this case are mechanisms to specify action durations, numerical attributes, symbolic path-planning, and some form of sensing functionality. These features will require extensions in the planning system. Perhaps just as importantly, these extensions must be properly exposed in the corresponding representation language. Note that if we had used a PDDL representation, we will be able to solve the first two representation problems (as PDDL 2.1 and above already support durative actions and numerical attributes), but we cannot solve the last two without extending PDDL's language specification.

Key Guidelines of the Proposed Framework

Having introduced our example planning system and problem, this section now presents the key guidelines of our planning extension/representation framework, explained via examples using the Crackpot system.

Correspondence between Language and Planning System Elements

The concept of an extensible planning language, introduced by PDDL, is quite essential for our proposed framework. However, PDDL is not able to handle nuances unique to a specific planning system, limiting its real-world use. In Crackpot, for example, it is non-trivial (although possible) to map PDDL predicates to ObjectTypes and AttributeTypes; worse, there is no direct PDDL analogue for ActuatorTypes.

This problem can be alleviated by designing the language around the planner, not the other way around. More succinctly, *form follows function*; this idea has been pointed out in critiques of PDDL (Boddy 2003). It must be noted that PDDL's "one-size-fits-all" representation stemmed from the need to provide common language elements across planning systems. Since our main focus is to extend planning systems into real-world applications such as games, with little to no use for inter-planner compatibility, we extend the basic idea into this

philosophy: *Develop a language that closely corresponds to the target planning system's internal representation.*

For example, since Crackpot recognizes ObjectTypes and AttributeTypes as first- and second-class constructs, respectively, they should be represented as-is in the language with their relative hierarchy unchanged (as opposed to representing their relationship as a predicate in PDDL). This has the advantage of easier extensibility system-wise, because new classes of constructs can be introduced to a domain language using the same class hierarchy of the planning system; for example, it is now trivial to add ActuatorTypes to the new language.

Planners conforming to the said philosophy will, of course, not be able to read each other's languages. Also, in the worst case, future planning problems and systems might require restructuring of the ontology: For example, Crackpot improves over EXCALIBUR (Nareyek 2001) by requiring resources to be grouped into objects for more expression possibilities (e.g., attributes can be references to objects), at the expense of incompatibility. However, language translation tools exist, such as proposed by Clark (1999), that allow wide-scale restructuring of a language, removing unnecessary data or even adding missing data, making it possible to import problems between planners.

Distributed Parsing of the Planning Language

A system that may be extended by external modules needs to have some form of *registration system*, which registers the cases when a planner needs to dispatch tasks to an external module rather than its internal constructs; for example, calling the constructor of an externally-created attribute instead of the system's built-in attributes.

The *Observer design pattern* (Gamma et al. 1995) is used as the basis of the registration system. This pattern is developed mainly for distributing events to *observers* or *listeners*, and it works well with our scenario—this allows modules to independently handle their own constructs. The Observer pattern effectively *distributes* the parsing of the representation language to the specific modules that are interested in smaller parts of the language.

For example, a SymbolicLocationAttribute external module can register as a listener on the same parts of the planning system that other attributes (SymbolicAttribute, NumericAttribute, etc.) also listen into. This way, whenever the language parser encounters the use of an attribute, a general "event" is fired, and the registered listener (in this case, SymbolicLocationAttribute) does the actual task, e.g., construction of the attribute, production of value instances, and managing of relations such as equality, comparison, etc. that are valid for the attribute.

XML as a Language Base

Theoretically, this planning framework can use a PDDL-like syntax as the language base. However, a much better option exists, in the form of the Extensible Markup Language or XML (Bray, Paoli, and Sperberg-McQueen 1998). Using XML as the language base has several advantages over maintaining a separate language:

- Since XML is widely considered as a standard, it enjoys vast third-party library support. Extension-aware planners will invariably have languages that change frequently. Existing XML libraries already allow users to change an XML-based language without needing to recompile the parser itself, which is perfect for a rapidly-evolving language.
- The XML SAX API (Megginson 2004) allows exactly the kind of distributed parsing that we need. SAX is a lightweight, event-driven API where each well-formed XML element (or "tag") fires an event; the target system's internal parser looks at an incoming XML "event" and distributes the event to the appropriate listener. The parser only needs to maintain a lookup table to find out which listener should be activated for which XML element.
- Using XSLT (Clark 1999), it is possible to do automatic translation between different languages (as recommended earlier). In fact, it is possible to transform the language into a version of PDDL with XML-style tokens, on which a simple token substitution can be performed to obtain pure PDDL.

XML is only used for planning system features where very high performance is not required. Generally, planning domains and problems are only loaded at the start of the planning process, so performance is not normally an issue, especially when taking into account the benefits that XML provides in terms of flexibility.

Example Implementation: Crackpot SLAP

A new domain specification language, dubbed "Scalable Language for Action Planning" or SLAP, is developed specifically for Crackpot. Two XML document schemas are created: The `<domain>` schema handles the formal definition of a domain (i.e., all *xxx*Type constructs), while the `<problem>` schema specifies a problem instance of that domain (i.e., all *xxx*Instance constructs).

The example domain presented earlier roughly translates to this form in SLAP, which corresponds with how Crackpot models the domain internally (for illustrative purposes only; details are left out due to space constraints):

```
<domain name="Apple Domain">
  <!-- definition for the location type -->
  <attribute_value_type name="LocType"
                        data_type="symbolic">
    <value name="livingroom" />
    <value name="door" />
    <value name="kitchen" />
  </attribute_value_type>

  <!-- object definitions -->
  <object_type name="Person">
    <attribute_type name="Hungry"
                    attribute_value_type="boolean" />
    <attribute_type name="Location"
                    attribute_value_type="LocType" />
    <actuator_type name="Legs" capacity="1" />
    <actuator_type name="Hands" capacity="1" />
  </object_type>
```

```
  <object_type name="Apple">
    <attribute_type name="Existing"
                    attribute_value_type="boolean" />
  </object_type>
  <object_type name="Door">
    <attribute_type name="Open"
                    attribute_value_type="boolean" />
  </object_type>

  <!-- action definitions -->
  <action_type name="EatApple">
    <parameter name="p" object_type="Person" />
    <parameter name="a" object_type="Apple" />
    <condition_type parameter="p"
                    attribute_type="Hungry"
                    relation="equals" value="true" />
    <condition_type parameter="p"
                    attribute_type="Location"
                    relation="equals" value="kitchen" />
    <condition_type parameter="a"
                    attribute_type="Existing"
                    relation="equals" value="true" />
    <contribution_type parameter="p"
                       attribute_type="Hungry"
                       value="false" />
    <contribution_type parameter="a"
                       attribute_type="Existing"
                       value="false" />
    <action_task_type parameter="p"
                      actuator_type="Hands" />
  </action_type>
  <action_type name="WalkFromLivingRoomToDoor">
    <parameter name="p" object_type="Person" />
    <condition_type parameter="p"
                    attribute_type="Location"
                    relation="equals"
                    value="livingroom" />
    <contribution_type parameter="p"
                       attribute_type="Location"
                       value="door" />
    <action_task_type parameter="p"
                      actuator_type="Legs" />
  </action_type>
  <action_type name="WalkFromDoorToKitchen">
    <parameter name="p" object_type="Person" />
    <parameter name="d" object_type="Door" />
    <condition_type parameter="p"
                    attribute_type="Location"
                    relation="equals" value="door" />
    <condition_type parameter="d"
                    attribute_type="Open"
                    relation="equals" value="true" />
    <contribution_type parameter="p"
                       attribute_type="Location"
                       value="kitchen" />
    <action_task_type parameter="p"
                      actuator_type="Legs" />
  </action_type>
  <action_type name="OpenDoor">
    <parameter name="p" object_type="Person" />
    <parameter name="d" object_type="Door" />
    <condition_type parameter="p"
                    attribute_type="Location"
                    relation="equals" value="door" />
    <condition_type parameter="d"
                    attribute_type="Open"
                    relation="equals" value="false" />
    <contribution_type parameter="d"
                       attribute_type="Open"
                       value="true" />
    <action_task_type parameter="p"
                      actuator_type="Hands" />
  </action_type>
</domain>
```

To implement the extensible framework itself, there were minimal changes to Crackpot's class structure. See Figure 2 for an overview. The Xerces-C++ parser (Apache Xerces Project 2010) was used for XML parsing, wrapping the library in the class XMLFactory using the Façade pattern (Gamma et al. 1995).

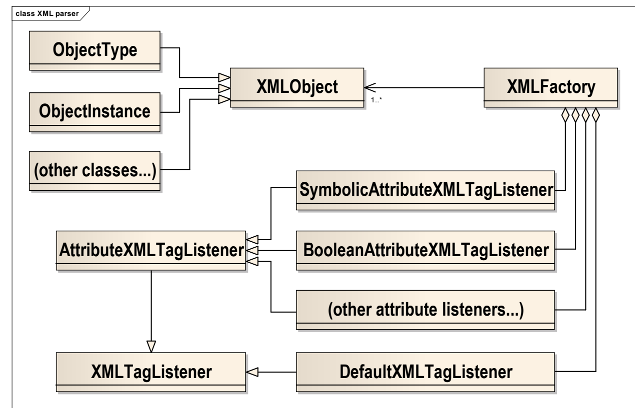


Figure 2. Crackpot's XML parser class structure.

For a custom module to register with the extensible framework, the interface XMLTagListener is provided. Since Crackpot also allows for third-party-supplied attributes, AttributeXMLTagListener is provided to further expand the XMLTagListener interface with helper methods that are relevant for attribute modules.

Registered listeners are stored in a lookup table on the tags that they listen to, e.g., all AttributeXMLTagListeners listen to the `<attribute_value_type>` tag. To resolve simple conflicts between multiple extension modules that listen to the same XML tags, Crackpot utilizes a *last-registered-first-called* rule, where the last listener to register is the first listener to be invoked by the parser. This rule makes sense because custom modules typically register with the system *after* the base modules. Future versions of the system may incorporate more sophisticated conflict resolution (more on this in the next section), but as it stands, the current system already allows for many interesting extension possibilities.

Possibilities for Planner Extensions

The advantages of our framework become apparent once the example problem is extended with new functionality.

Introduction of New Features

In our architecture, it is possible to add a new XML element for each new feature added to the system. For example, in Crackpot, a timing module[2] can be introduced to the system to handle action durations, which registers with our framework by listening to a new XML element, `<timing>`. This new element can be placed as a child under `<action_type>`. This allows many ways of implementing durative actions, such as a fixed duration:

[2] Crackpot currently implements durations in a different way; action-component relations will handle durative actions even more generally.

```
<action_type name="WalkFromLivingRoomToDoor">
  <timing duration="10" />
  ...
```

Or a condition- or contribution-related sub-duration (the overall duration is computed from all sub-durations):

```
<contribution_type parameter="p"
                attribute_type="Hungry" value="false">
  <timing effect_time="30" />
  ...
```

Using this framework, any planning system can decide how to model durations without being tied to a particular representation like that of PDDL, where durative actions needed a completely new construct `(:durative-action)` to support a single way of specifying durations.

Custom modules to add sensors to the outside world (in order to support online planning) are likewise easy to add in. An application can create hooks to attributes by adding a tag under the `<attribute_type>` tag:

```
<object_type name="Door">
  <attribute_type name="Open"
                attribute_value_type="boolean">
    <sensor_stream id="doorState" resolution="5" />
  </attribute_type>
</object_type>
```

In this example, the `<sensor_stream>` element is provided by a custom module, and specifies that a refresh of the door state is triggered every 5 time units. The actual sensor values may be transmitted to the planner via low-level means (i.e., not XML, for higher performance).

Overloading of Existing Features

It is also possible to extend the behavior of an XML element via *element overloading*, i.e., letting multiple modules listen-in on the same XML element. For example, custom modules to support new attribute value types like NumericRange and SymbolicLocation can provide listeners to the `<attribute_value_type>` tag, overloading its use when it encounters a `data_type` string that corresponds to what this module handles. They can make their own XML tags further down the hierarchy:

```
<attribute_value_type name="HungerType"
                      data_type="numeric_range">
  <range begin="0" end="100" />
</attribute_value_type>
<attribute_value_type name="LocType"
                      data_type="symbolic_location">
  <value name="livingroom">
    <coordinates x="0.0" y="0.0" />
    <connection to="door" />
  </value>
  <value name="door">
    <coordinates x="0.0" y="10.0" />
    <connection to="kitchen" />
    <connection to="livingroom" />
  </value>
  <value name="kitchen">
    <coordinates x="10.0" y="10.0" />
    <connection to="door" />
  </value>
</attribute_value_type>
```

These modules can then also override the `<condition>` and `<contribution>` tags to specify their own relations and operations:

```
<!-- a more natural model of hunger satiation -->
<condition_type parameter="p"
                attribute_type="Hunger"
                relation="greater_than" value="50">
  <timing check_time="0" />
</condition_type>
<contribution_type parameter="p"
                   attribute_type="Hunger"
                   operation="linear_decr" value="25">
  <timing effect_time="20" duration="30" />
</contribution_type>
```

These XML elements are handled directly by their respective modules, giving these modules the freedom to specify an entirely new XML hierarchy for their own data; for example, custom Set or Matrix attributes may include sizable amounts of formatted numeric data (potentially with the base functionality inherited from NumericRange).

Feature overloading may introduce problems when conflicting modules listen-in on the same XML elements (necessitating *conflict resolution*, mentioned in the previous section), but these issues are not unlike those encountered with OOP languages like C++; in future implementations, these problems may be solved using the same software engineering principles commonly used in these languages (such as disallowing multiple inheritance, adding support for public/private visibility, and so on).

Planner-Specific Exposure of the Solution Process

So far the preceding extensions simply modify existing planning constructs to support better expressiveness of a problem domain. However, *internal planner extensions* can also expose or even introduce changes to the solution process itself. Such extensions are not meant to be written by third-parties but by internal developers of the planning system. For example, Crackpot's forthcoming cost management system, an improved version of what is found in EXCALIBUR (Nareyek 2001), will allow cost modifiers to influence the selection of repair heuristics in a specific domain. These costs are specified in the form of *domain hints*. First, *cost collections* are specified by the domain:

```
<cost_collection name="satisfaction">
  <cost_type name="goal" cost_mapping="3x" />
  <cost_type name="aux" cost_mapping="2x" />
</cost_collection>
<cost_collection name="optimization">
  <cost_type name="optional" cost_mapping="default" />
</cost_collection>
```

Then, cost types may be registered for the different *cost centers* in the domain, i.e., attributes, actuators and (forthcoming) action-component relations:

```
<attribute_type name="Hungry"
                attribute_value_type="boolean">
  <cost_registration cost_name="unsatisfied"
                     cost_type="goal" />
</attribute_type>
<attribute_type name="Location"
                attribute_value_type="LocType">
```

```
  <cost_registration cost_name="distance"
                     cost_type="optional" />
</attribute_type>
<actuator_type name="Legs" capacity="1">
  <cost_registration cost_name="usage_overlap"
                     cost_type="aux" />
</actuator_type>
```

This allows for an expressive model of plan preferences that more closely mirrors Crackpot's internal planning architecture (which is based on cost repair via local search) than that of strong and soft constraints in PDDL 3.0 (Gerevini and Long 2005).

Conclusion

Our proposed framework solves two important problems: how to make a planning system support a level of extensibility to facilitate its use for real-world problems, and how to model and support an evolving domain representation language that allows problems to take advantage of such planner extensibility. The framework uses three key guidelines: *maintaining correspondence* between domain ontology and the planner's internal architecture, *distributing language parsing* to external modules through the use of the Observer design pattern, and *using XML as a language base* to facilitate language design, parsing and translation to other languages.

Possible future work include the development of more sophisticated forms of conflict resolution between planning extension modules, a common XSLT stylesheet library to allow translation of domain problems between planning systems (or to/from PDDL), and extensions to the Crackpot planning system itself, such as the aforementioned cost manager, control flow actuator, action-component relations, and a complete sensing/acting system to fully support real-world online planning.

References

Apache Xerces Project. 2010. Xerces-C++ XML Parser, Project documentation, available at http://xerces.apache.org/xerces-c, Apache Software Foundation.

Blum, A., and Furst, M. 1997. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence* 90(1-2): 281-300.

Boddy, M. 2003. Imperfect Match: PDDL 2.1 and Real Applications. *Journal of Artificial Intelligence Research* 20: 123-137.

Bray, T.; Paoli, J.; and Sperberg-McQueen, C. M. 1998. Extensible Markup Language (XML) 1.0. Technical Report, W3C recommendation, W3C.

Clark, J. 1999. XSL Transformations (XSLT) Version 1.0. Technical Report, W3C recommendation, W3C.

Fikes, R. E., and Nilsson, N. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 5(2): 189-208.

Fox, M., and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research* 20: 61-124.

Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA; Menlo Park, CA: Addison-Wesley Publishing Co.

Gerevini, A., and Long, D. 2005. Plan Constraints and Preferences in PDDL3, Technical Report, R. T. 2005-08-47, Università degli Studi di Brescia, Dipartimento di Elettronica per l'Automazione.

Kautz, H., and Selman, B. 1998. BLACKBOX: A New Approach to the Application of Theorem Proving to Problem Solving. In *Working Notes of the Workshop on Planning as Combinatorial Search, held in conjunction with AIPS-98,* 58-60. Pittsburgh, PA.

Kautz, H.; Selman, B.; and Hoffman, J. 2006. SatPlan: Planning as Satisfiability. In *Abstracts of the 5th International Planning Competition,* available at http://www.cs.rochester.edu/~kautz/satplan/index.htm.

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.

Megginson, D. 2004. SAX - Simple API for XML, Project documentation, available at http://www.saxproject.org/, Megginson Technologies, Ltd.

Nareyek, A. 1998. A Planning Model for Agents in Dynamic and Uncertain Real-Time Environments. In *Proceedings of the Workshop on Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments at the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS-98),* 7-14. Menlo Park, California: AAAI Press.

Nareyek, A. 2001. *Constraint-Based Agents: An Architecture for Constraint-Based Modeling and Local-Search-Based Reasoning for Planning and Scheduling in Open and Dynamic Worlds (LNAI 2062).* Springer.

Nareyek, A. 2003. Planning to Plan - Integrating Control Flow. In *Proceedings of the International Workshop on Heuristics (IWH'02),* 79-84.

Nareyek, A.; Fourer, R.; Freuder, E. C.; Giunchiglia, E.; Goldman, R. P.; Kautz, H.; Rintanen, J.; and Tate, A. 2005. Constraints and AI Planning. *IEEE Intelligent Systems* 20(2): 62-72.

OMG. 2010. Unified Modeling Language (UML), Version 2.2, Formal specification, available at http://www.omg.org/spec/UML/2.2/, Object Management Group.

van Beek, P., and Chen, X. 1999. CPlan: A Constraint Programming Approach to Planning. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99),* 585-590.

Wolfman, S. A., and Weld, D. S. 1999. The LPSAT system and its Application to Resource Planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence,* 310-316. Stockholm, Sweden.

System Demonstrations

Constraint and Flight Rule Management for Space Mission Operations

Javier Barriero and John Chachere and Jeremy Frank and Christie Bertels and Alan Crocker

NASA Johnson Space Center Mail Stop DS15 2010 NASA Parkway Houston, TX 77058
SGT, Inc. NASA Ames Research Center Mail Stop N269-3 Moffett Field, CA 94035-1000
NASA Ames Research Center Mail Stop N269-3 Moffett Field, CA 94035-1000
{Javier.Barreiro,John.M.Chachere,Jeremy.D.Frank,Alan.R.Crocker,Christie.D.Bertels}@nasa.gov

Abstract

Thousands of operational constraints govern NASA's human spaceflight missions. NASA's Mission Operations Directorate (MOD) develops, documents, and applies these constraints during mission planning to ensure the safety of the crew, as well as proper operation of the spacecraft systems and payloads. These constraints are currently written as documents intended to be read by MOD staff. Similar operational constraints are developed independently by different organizations, and manually transformed into machine-readable formats needed to drive tools such as automated planners, mission analysis and mission monitoring tools. The resulting process is inefficient and error prone. In response, NASA has developed Constraint and Flight Rule Management (ConFRM) software to centralize the capture of operational constraints and to transform these constraints into different products for different uses. ConFRM will help MOD staff create constraints using disparate information, update operational constraints using new information, and check sets of constraints for errors and omissions.

Human Spaceflight Mission Operations

NASA's Mission Operations Directorate (MOD) develops, documents, and applies operational constraints to ensure the safety of the crew and the proper operation of the spacecraft systems and payloads. During pre-flight planning, NASA and its partners systematically develop, document, and approve these constraints. MOD provides training and operations teams with approved constraint documents in paper form and online searchable databases. During training, flight controllers and related personnel learn all of the constraints relevant to their disciplines, and configure tools to help enforce those constraints. During nominal operations, the Flight Control Team and crew ensure that the constraints are continually satisfied. During off-nominal operations, these constraints indicate corrective actions the Flight Control Team should take in order to return to an acceptable mission state.

Many operational constraints are developed in order to mitigate a hazard documented in a *Hazard Report*. Many more are derived from engineering analysis of spacecraft systems. Yet more operational constraints are derived from the operational experiences of Flight Control Teams and crew. There are several types of operational constraints. The Flight Control Team uses *Flight Rules (FRs)* to avoid hazards or guide reactions to unexpected events. Mission planners, who are part of the Flight Control Team, use *Ground Rules and Constraints (GR&Cs)* and *Crew Scheduling Constraints (CSCs)* to plan the crew's daily activities. The Flight Control Team uses FRs and GR&Cs mitigate hazards that must be avoided, while Crew Scheduling Constraints are 'best practices' developed over years of operational experience. *Generic* constraints apply to all missions. *Flight-specific* constraints are specific to a mission's payload, objective or system configuration. For a typical six month period, the Flight Control Team manages 1000 generic FRs, 100 flight-specific FRs, 300 GR&Cs, and 100 CSCs.

Development of Operational Constraints

The following scenario illustrates a common life-cycle of operational constraints. Suppose a new Hazard Report specifies that hand-held radios onboard ISS interfere with some communications between ISS and Mission Control. The Flight Control Team may write a Flight Rule to ensure that the crew has powered off these radios prior to sending critical commands to ISS, and link this rule to the Hazard Report. In addition, the Mission Planners may document the GR&C to ensure the crew's plan contains these activities explicitly, and link this GR&C to the Flight Rule. Over the course of several missions, the specific details of the constraint may change; the number of radios that must be turned off, the type of radio, and the specific ISS commanding activities that require the radios to be turned off. Also, these mission specific constraints may be changed to generic, i.e. they impact every flight.

ISS crew activity planning happens in stages. The Long-Range Plan (LRP) is generated for roughly six months worth of ISS activities (equal to one ISS Increment and crew rotation). Versions of this plan are initially generated using Excel and Microsoft Word. Once the major mission milestones are decided, the LRP for the Increment is generated using the Consolidated Planning System, a model-based planner. CPS permits the declaration of state and resource requirements, as well as temporal constraints on activities. These constraints are manually translated from GR&C and CSC documents. CPS allows operators to add activities, delete activities, and automatically generate plans according to constraints on activities [1].

Operating constraints may change from mission to mission. A constraint created for a specific flight may be deemed generally applicable, or changes in vehicle configuration may lead to new crew scheduling constraints. Similar operational constraints are developed

independently by different organizations. A GR&C may contain identical information to a Flight Rule, but today these documents are created by different parts of the Flight Control Team, and may be mutually inconsistent because of changes to the Flight Rule that are not reflected in the GR&C. Manual input of constraints into machine-readable formats needed to drive tools such as automated planners, mission analysis and mission monitoring tools is performed after documentation of the constraints. The resulting process is inefficient and error prone.

ConFRM

The creation and management of consistent operational constraints to drive automated planning has been addressed previously by the AI planning community. Existing tools can detect ill-formed rules, mutually inconsistent rules and automatically infer rules from plans [2,3]. However, the task of managing these operational constraints for human spaceflight offers some unique challenges. First, the constraints must be documented so that both people and AI planners can use them. Second, the constraints will be created by a large, distributed team of knowledge engineers. Third, these tools will be used by experienced spaceflight operators and engineers who are not AI experts. While rules for automated planners have been extracted from documents e.g. for Orbital Express [7], this is not common practice today. Lastly, the gradual changes of constraints over long periods of time introduces the problem of 'lifetime rule management'.

NASA has designed and prototyped a software solution called Constraint and Flight Rule Management (ConFRM). ConFRM's approach to authoring and managing operational constraints addresses the problems described above. ConFRM's main features are: 1) ConFRM provides direct links to the many spacecraft command and telemetry descriptions, databases of hazards, previously created operational constraints, and analysis products that the constraint references. ConFRM automatically reads XML command and telemetry descriptions [5]. ConFRM can establish links to these products either manually or automatically. ConFRM can also detect changes to product content and location, so the constraints and links are always up-to-date. 2) ConFRM enables export of relevant information from operational constraints to planning and monitoring tools, thereby reducing the effort in mapping the documented constraint to the tools used to ensure the constraints are followed. ConFRM's approach to capturing constraint knowledge in a central database also allows each group to export the content it needs, reducing duplication of effort and operational constraint mismatches between groups. 3) The ConFRM prototype includes basic error and inconsistency detection supported by formal modeling. The NASA team is evaluating a promising enhancement to automatically integrate constraints with monitoring and planning software.

ConFRM's technical architecture has three layers: 1) A Storage Layer uses a relational database for scalability and rich searching and reporting functionality. 2) A Business Layer encapsulates document lifecycle, version control, error checking, authentication, and authorization. A plug-in mechanism allows for modular 2-way integration with external applications. Separating the Storage and Business layers enables flexibility in database technology and design. 3) A Presentation Layer provides a rich authoring UI with wiki formatting. An alternative, lightweight web UI can provide access for casual and external users (such as hardware manufacturers, whose role is limited to providing technical details for some constraints). The UI is built using Eclipse tools, following in the footsteps of other recent developments in mission operations software such as Mars mission operations [4] and human spaceflight procedure development [6].

We will demonstrate the features of ConFRM, particularly those used in configuring a planning system. We will show how Wiki formatting allows non-programmers to add various types of formal structure to text-based documents, and how GR&C's can be exported in order to configure a planner. We will show how ConFRM facilitates tracing back of GR&Cs to parent FRs, searching for products based on keywords, browsing imported command and telemetry, detecting errors in constraints, and automatic creation of both human- and machine-readable content.

References

[1] Frank, J., "When Plans are Executed by Mice and Men." Proceedings of the IEEE Aerospace Conference, IEEE, Big Sky, MT, 2010.

[2] Simpson, R. M. Kitchin D. E. and McCluskey T. L. Planning domain definition using GIPO. The Knowledge Engineering Review. Volume 22 , Issue 2 (June 2007) pp. 117-134, 2007

[3] T. S. Vaquero ; V. M. C. Romero;, F. Tonidanel; J. R. Silva. itSIMPLE 2.0: An Integrated Tool for Designing Planning Domains. Proceedings of the International Conference on Automated Planning & Scheduling 2007, Providence, Rhode Island. Menlo Park, California, USA, 2007.

[4] Aghevli, 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 with Ensemble." 5th International Workshop on Planning and Scheduling for Space. Baltimore, MD, 2007

[5] Simon, G. Shaya, E. Rice, K. Cooper, S. Dunham, J. Champion, J. "XTCE: A Standard XML-Schema for Describing Mission Operations Databases," IEEE Aerospace Conference, IEEE, Big Sky, MT, 2004

[6] Izygon, M., Kortenkamp, D., Molin, A., "A Procedure Integrated Development Environment for Future Spacecraft and Habitats," Space Technology and Applications International Forum, Albuquerque, NM, 2008.

[7] Knight, R., Chouinard, C., Jones, G., Tran, D. "Planning and Scheduling Challenges for Orbital Express." Proceedings of the 6th International Workshop on Planning and Scheduling for Space, 2009.

Using Knowledge Engineering for Planning Techniques to leverage the BPM life-cycle for dynamic and adaptive processes

Juan Fdez-Olivares and **Arturo González-Ferrer** and **Inmaculada Sánchez-Garzón** and **Luis Castillo**
Department of Computer Science and Artificial Intelligence
University of Granada

Motivation

The approach here presented deals with the development of AI P&S Knowledge Engineering techniques in order to: (1) automatically generate planning domains from expert knowledge described in BPM (Muehlen and Ho 2006) process models,(2) automatically generate plans, upon these domains, that can be interpreted as business process, and (3) automatically transform these plans back into executable business processes.

The interest of this work is focused on business processes the deployment and execution of which strongly depends on the given context of an organization and, therefore, do not respond to a fixed pattern. An example of such processes may be the organizational process to manage the collaborative creation of e-learning courses within a virtual learning center (a special case of product development processes). Upon a customer request, the manager of the organization needs an estimation of the tasks to be accomplished, the resources of the organization to be used in the elaboration of the course, as well as the time needed to deploy the requested course. Under these circumstances, since the final workflow to be carried out cannot be easily devised a priori, managers and decision makers rely on either project management or business process simulation tools to support decisions about activity planning (in order to find adequate dependencies between tasks and their time and resources constraints), by determining various scenarios and simulate them. But these tools force a knowledge worker to carry out a trial-and-error process that may become unrealistic when either the number of alternatives courses of action makes unmanageable to ascertain which tasks should be considered or the constraints get harder. Therefore, it is widely recognized (WfMC 2010) that, at this step, the life-cycle of BPM presents some weak points and new techniques must be developed at the process modeling/generation step, in order to fully cover the needs of knowledge workers for dynamic, adaptable processes.

From the AI P&S point of view, the need to obtain a context dependent, concrete workflow from a given business process model can be seen as the problem of obtaining a situated plan from an original process model. That is, a plan that represents a case for a given situation, and such that its composing tasks and order relations, as well as its temporal and resource constraints, strongly depend on the context for which the plan is intended to be executed. This is not a trivial problem which requires at least two strong requirement in order to be solved. On the one hand, since the (possibly nested) conditional courses of action that may be found in a process model lead to a vast space of alternative tasks and possible orderings, it is necessary to carry out a search process on this space in order to determine the sequence of actions to be included in the situated plan. On the other hand, the search process necessarily has to be driven by the knowledge of the process model, which in most cases takes a hierarchical structure. Therefore, considering that there is a structural similarity between BPM process models and HTN domain models, we opted in a previous work (Gonzalez-Ferrer, Fdez-Olivares, and Castillo) for the development of Jabbah: a Knowledge Engineering for Planning tool that allows to automatically extract and represent HTN planning knowledge from a business process model. Hence, by using Jabbah in order to generate HTN domain and problem files, from an original process model, it is possible to carry out a knowledge-driven HTN planning process that results in the generation of situated plans, that is, plans customized for a given situation.

These plans can be used either for supporting decision making about activity planning or process validation based on use-case analysis, leveraging the current BPM life-cycle at its process modeling/generation step. Furthermore, Jabbah has been extended (among other features below explained) in order for the business cases obtained to be executed in standard BPM runtime engines. Therefore, Jabbah fulfills, by using AI P&S knowledge engineering techniques, the needs of knowledge workers not yet completely covered by BPM technologies, in the management of dynamic, adaptable processes. Definitively, Jabbah supports most of the BPM life-cycle, from adaptive case generation (starting from a given process model represented in BPM standard languages), to the execution of such processes.

Transformation processes

The BPM models given as input to Jabbah are represented in XPDL, a standard BPM language that is a XML serialization of BPMN, the standard graphical representation for process

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

models. See (Gonzalez-Ferrer, Fdez-Olivares, and Castillo) for a description of the relevant XPDL entities managed by Jabbah.

Transformation from process models to planning domains. Given an XPDL process as input, Jabbah proceeds by identifying common workflow patterns (that is, sequential, parallel, subprocess and conditional structures) as process blocks in the process model, and then maps them into HTN decomposition schemes (a decomposition scheme is formed by a compound task and its associated decomposition methods, each one comprising a set of subtasks arranged by order constraints). Hence, it is possible to use a state-of-art HTN planner that takes this domain representation as input and use its output as activity plans helpful for management tasks. We have used the IACTIVE™ planner for this work, a temporally extended HTN planner which uses an HTN planning language that is a hierarchical extension of PDDL (we call it HTN-PDDL, see (Gonzalez-Ferrer, Fdez-Olivares, and Castillo) for more details).

In its first version, Jabbah was capable of detecting, on the one hand, sequential and parallel blocks, translating them into decomposition schemes with one single method, with subtasks arranged by either sequential or parallel order constraints, respectively. On the other hand, conditional blocks were mapped into decomposition schemes comprising as many methods as the number of alternative courses of action defined by the conditional gateways (see (Gonzalez-Ferrer, Fdez-Olivares, and Castillo) for more details). At present, Jabbah has been extended in order to identify both, subprocess relations between process activities, and more complex synchronization mechanisms between parallel branches described in a process model. Regarding the later extension, once a synchronization between parallel blocks has been detected (represented as a data flow between two activities), Jabbah generates the necessary predicates in the effects of the data producer as well as in the preconditions of the consumer, in order to establish a causal relationship between tasks in the HTN domain.With respect to the former extension it is important to note that it can only by managed by hierarchical planning approaches, and it endows Jabbah with a greater expressivity, allowing it to deal with a wider set of more realistic process models.

By following this process, it is possible to generate problem and domain files which are given as input to the HTN planner in order to obtain situated plans. These plans are generated by the planner for a given context represented in the problem file, and they can be interpreted as adaptive business cases since they are direct and automatically obtained from the initial process model. Furthermore, these plans can also be seen as process instances of the original process model. Thus, next we briefly describe how these plans are transformed back into XPDL process instances in order to be understandable, and so executable, by a BPM runtime engine.

Transformation from plans to executable process models. Given an XPDL process instance as input, BPM engines are commonly endowed with the necessary machinery in order to interactively execute every task (allowing to start, finish, suspend or abort it) in the process by following an execution model based on state-based automata. The plans generated by the planner, using the planning domains and problems generated by Jabbah, are represented in XML as a collection of *Task nodes* where every node contains information about: *actions (activities)* and their parameters; temporal information as *earliest start and earliest end* dates for the execution of every activity; *order dependencies* between actions which allow to establish sequential and parallel runtime control structures; and *metadata* which allow to represent additional knowledge like the user-friendly *description* of a task, its *type* (manual, auto) or its *performer* (that is, the participant of the activity). It is worth to note that metadata are generated at domain generation phase and are automatically extracted and generated by Jabbah.

Starting from this XML plan representation, we have implemented as an extension of Jabbah a translation process that automatically generates XPDL processes which can be directly executed in a BPM runtime engine and users can interact with them on an underlying BPM console (see (Fdez-Olivares et al. 2010) for more details). This process has three main steps: (1) generation of XPDL DataFields and Participants from the problem and domain files; (2) generation of XPDL activities from the information about actions, temporal constraints and metadata in the plan; (3) generation of XPDL transitions from the order dependencies between the actions of the plan.

Notes on experiments

We have applied the transformation processes here described in some experiments, by representing the whole process to develop and deploy a specific course within an e-learning center. Having this process model and an incoming course request, as well as some available workers with different capabilities, we have generated its corresponding HTN planning domain. Then, we have obtained a plan by using the temporally extended HTN planner and we have translated it into an executable XPDL process. Finally, this process has been used as input to a standard BPM runtime engine and console. In conclusion, the system not only allows to support the generation of dynamic, adaptive processes in order to support decision making, in addition it provides the necessary functionalities to execute these processes under user request.

References

Fdez-Olivares, J.; Sanchez-Garzon, I.; Gonzalez-Ferrer, A.; and Castillo, L. 2010. ”Integrating plans into BPM technologies for Human-Centric Process Execution”. In *ICAPS 2010. KEPS Workshop*.

Gonzalez-Ferrer, A.; Fdez-Olivares, J.; and Castillo, L. ”JABBAH: A Java Aplication Framework for the Translation between Business Process Models and HTN”. In *ICKEPS 2009*.

Muehlen, M., and Ho, D. T.-Y. 2006. *Business Process Management Workshops, LNCS 3812*. Springer. chapter ”Risk Management in the BPM Lifecycle”, 454–466.

WfMC. 2010. http://www.xpdl.org/nugen/p/adaptive-case-management/public.htm.

Analyzing Plans and Comparing Planners in itSIMPLE$_{3.1}$

Tiago Stegun Vaquero[1,2] and **José Reinaldo Silva**[1] and **J. Christopher Beck**[2]
[1]Department of Mechatronic Engineering, University of São Paulo, Brazil
[2]Department of Mechanical & Industrial Engineering, University of Toronto, Canada
tiago.vaquero@poli.usp.br, reinaldo@usp.br, jcb@mie.utoronto.ca

Introduction

Real planning problems arise from real application domains. One significant challenge to achieving satisfactory planner performance, in terms of both plan quality and planning speed, is the development of a clear understanding and accurate model of the application domain. Lack of knowledge or ill-defined requirements typically propagate to poor project specifications, then to an erroneous planning model and finally to unsatisfactory planner performance. Ideally, assumptions and models leading to incorrect and poor quality plans should be spotted and fixed in the design process. One useful approach to finding such problems is the analysis of plans generated by different planning techniques. Hence the need to include a re-modeling cycle in any real application design process.

Following this idea, itSIMPLE$_{3.1}$ not only allows users to test the PDDL model that is generated from a UML specification with a set of modern planners, it also provides a set of tools for plan analysis. Besides the existing capabilities of plan visualization in UML and the definition of plan quality metrics, itSIMPLE$_{3.1}$ brings extended features aimed to help users to (1) perform experiments with different planners, (2) evaluate plan quality, (3) compare planner performance, and (4) compare different model refinements. In all these new features, users observe and analyze automatically generated reports that contain useful information for investigating plans, the performance of planners, and the impact of certain modifications.

In this short paper, we briefly describe these new features in itSIMPLE$_{3.1}$. We start by introducing how users define and represent the quality metrics that guide the plan evaluation. We then show how user can set-up planning experiments to study and analyze planners and domain models. Finally, we describe the reports generated by itSIMPLE$_{3.1}$.

Defining Quality Metrics

Some of the research effort in the itSIMPLE project has been directed to plan quality analysis. One of the extended functionalities available in itSIMPLE$_{3.1}$ supports the definition of plan quality metrics and criteria acquisition. The main objective of this new functionality is to capture domain metrics and criteria from users and to use them to evaluate and compare plans. This feature aims to help the designer identify and explore their own metrics and their preferences on the metric values.

In order to capture metrics and criteria, itSIMPLE provides an interface in which users specify and select the variables that correspond to key parameters for measuring the quality of the plan. Metrics can be, for instance, a variable of the domain (e.g., *travel-distance* or *total-fuel-use*), an action counter that can involve specific characteristics (e.g., how many times action *move* appears in the plan with the first parameter being *loc1*), or an linear function involving several domain variables. These metrics can be maximized or minimized by planners or just observed by users. Each one of these metrics can have a preference function that maps variable values to scores in the interval [0,1] (where 0 is unsatisfactory and 1 is satisfactory). The definition of metrics in itSIMPLE was inspired by the work of (Rabideau, Engelhardt, and Chien 2000).

These metrics and their preference functions are used to evaluate the plans produced by planners. These plan evaluations can be used while analyzing models and planners or when performing planning experiments. The evaluations (metric values and plan scores) can support and lead designers to modified their models accordingly to their expectations. Such modification process is performed manually, but automatic refinement procedures have been investigated.

Performing Experiments with Planners

A number of planners can be used within itSIMPLE$_{3.1}$'s graphical interface: Metric-FF, FF, SGPlan, MIPS-xxl, LPG-TD, LPG, hspsp, SATPlan, Plan-A (IPC-6), blackbox (version 4.3), MaxPlan (IPC-5), LPRPG (beta version 1), and Marvin (IPC-4). Since itSIMPLE$_{3.0}$, the requirements tags of the (automatically generated) PDDL can be used to select the planners that can handle a given domain.

Generally, during research experiments in planning, we might want to do the following: test a specific domain model with different planners; test a particular planner with different planning domains; compare the performance of a set of planners in a given domain model; or compare the performance of a set of planners in a set of planning domains (what is generally done during the International Planning Compe-

Copyright © 2010, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

tition). itSIMPLE$_{3.1}$ allows the user to perform all these kinds of experiments.

In itSIMPLE$_{3.1}$, experiments are normally done as follows. Users first select which planners and domain models will be used in an experiment. The tool lets designers specify time-outs for all planners or a specific time-out for a particular planner. The tool, then, handles the experiment automatically, while showing to the user the progress and the status of the process. During the execution of every planner, itSIMPLE$_{3.1}$ records essential information and data: not only the speed (runtime) and solvability of the planner for a given problem instance, but also the quality of the resulting plans based on the defined metrics. All information and data from the experiments are recorded in a XML file which is used to display the results to the user in the form of a report.

Generating Reports

In this section we describe the reports that itSIMPLE$_{3.1}$ can generate from the data record of the experiments.

Plan Report

When a user wants to analyzed a particular plan, itSIMPLE$_{3.1}$ can generate a HTML report that shows basic information about the planner, the evolution of all metrics using charts (so user can identify peaks, maximum and minimum), the individual score for each metric, and, finally, the overall score of the plan.

Plan and Planner Comparison Report

When considering experiments with multiple planners and/or multiple domains, itSIMPLE$_{3.1}$ generates a comparison report that shows how planners perform for each problem instance concerning speed, solvability, number of actions, plan cost, quality of metrics, and plan quality. The report contains tables that list all these data. The report also contains two charts for every domain in the experiment: the first one correlates “number of actions” and “planners” considering every problem instance in the domain; the second correlates “speed” (time) and ”planners” also considering every problem instance in the domain. These charts are very similar to those presented in the IPC results. At the end of the comparison report, we provide a summary of the best planners concerning the categories speed, quality and plan length. This summary is made by counting how many times each planner dominates on problem instances in each category. This report, also in HTML format, can be very useful to identify better planners as well as critical domains and problem instances. In fact, it can simulate the evaluation process generally done in IPC.

Since every plan is stored in the experiments data file, users can quickly simulate or visualize a chosen plan using itSIMPLE’s interface to do a deeper investigation. Comments can be added to the plan which is stored in the XML file for further analysis and reuse.

Comparing Refined Domains

Plan analysis can help validation of the model and can also guide model modification and refinement. Recent research work with itSIMPLE (Vaquero, Silva, and Beck 2010) has shown that several observations, hidden requirements, and potential modifications to the model can be discovered while simulating the plan in a virtual environment. These modifications produce new (refined) models and a lead to subsequent plan analysis. The cycle of re-modeling and analysis naturally raises the need to compare the impact of inserted modifications, i.e., comparing different planners on different versions of the model. In order to help on such comparison tasks, itSIMPLE$_{3.1}$ produces a special report that combines several experiments on a particular domain. With such a combination of data, the tool can show (1) the model in which the planners produced the the best quality plans, (2) the model in which the planners had the fastest response and (3) the model in which the planners had the best plan length. The resulting report contains tables that show the performance of the planners in each problem instance of every (refined) model and also column charts illustrating the best models for each problem instance and for all experiment. Figure 1 illustrates an overall evaluation of four models where each criteria has been mapped so that higher means better. This figure shows that, in this case, Model AB is the best model for all criteria.

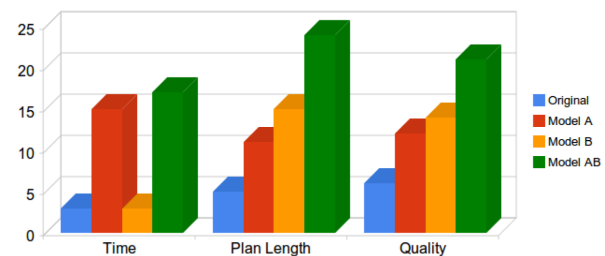


Figure 1: Overall comparison of models in itSIMPLE$_{3.1}$

Conclusion

The itSIMPLE project is in ongoing development. We have recently put some efforts on integrating itSIMPLE with other tools (such as virtual prototyping environments, model checking, and the Automatic Validation tool (VAL) for PDDL), as well as improving the modeling features. itSIMPLE$_{3.1}$ can be found in our website http://dlab.poli.usp.br.

References

Rabideau, G.; Engelhardt, B.; and Chien, S. 2000. Using generic preferences to incrementally improve plan quality. In *Proceedings of the Fifth AIPS, Breckenridge, CO.*

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). Toronto, Canada. To be published.*

Visual design of planning domains

Jindřich Vodrážka
Charles University in Prague
Faculty of Mathematics and Physics
Malostranské náměstí 25
118 00 Praha 1, Czech Republic
vodrazka@ufal.mff.cuni.cz

Lukáš Chrpa
Czech Technical University in Prague
Agent Technology Center
Karlovo náměstí 13
121 35 Praha 2, Czech Republic
chrpa@agents.felk.cvut.cz

Introduction

Description of planning domains and problems is the first critical task when using planning technology. It naturally belongs to the area of Knowledge Engineering as it involves knowledge extraction (from the user) and schematic formulation of problems.
To make the task more comprehensive for non-experts in planning we propose to use graphical representation for planning domains. This method has been already used in systems GIPO (Simpson *et al.* 2007) and itSIMPLE (Vaquero *et al.* 2007). In this paper we will describe system VIZ inspired by them. Unlike GIPO or itSimple, VIZ is a lightweight system which uses straightforward approach to model a planning domain. Users do not need to be familiar with PDDL syntax. VIZ provides a graphical user interface for description of planning domains and problems. The interface uses collection of simple diagrams which can be exported directly into PDDL.

Categorization

Planning domain designers usually start with informal description of some system. Available pieces of information need to be categorized with respect to their meaning. Concept of object oriented programming and formalism of first order logic (FOL) is incorporated in the following categories which are used in VIZ:

- *class* determines common properties for all *objects* which belong to it. It can be understood as a set of *objects*.
- *object* is a specific instance of some *class*
- *variable* can refer to any *object* from particular *class*
- *predicate* denotes an atomic statement of FOL language for a given planning domain

Design levels

It is convenient to split complex task of planning domain design into pieces. We can consider three levels of abstraction:

1. declaration of *classes* and *predicates*
2. definition of planning operators using *variables* and previously declared *predicates*
3. definition of planning problem using *objects* and *predicates*

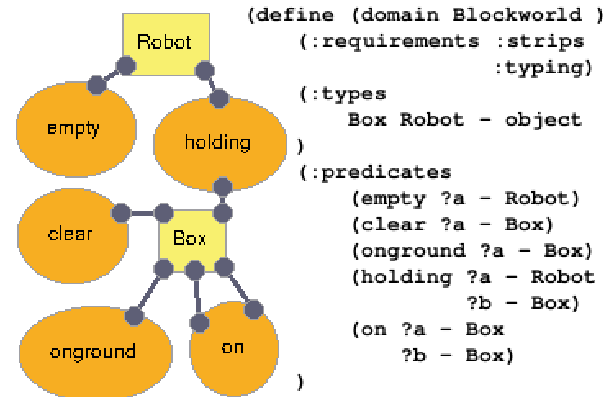


Figure 1: Declaration of language

Planning domain is described at the first two levels. Planning problems are described at the third level.

Program VIZ

Three types of diagrams are sufficient to describe simple planning domain with VIZ. All diagram types share the same idea. Semantics of diagrams correspond with three levels of abstraction described earlier. Number of diagrams depends on number of planning operators and problems.

Blockworld domain (Slaney and Thiébaux 2001) is used as an example in following Figures. Corresponding PDDL code is shown as well.

Declaration of language

In Figure 1 we can see rectangular nodes labeled `Box` and `Robot` representing *classes* used in the Blockworld domain. There are also eliptic nodes as a representation for *predicates*. By connecting e.g. *predicate* `empty` and *class* `Robot`, we can declare type of `empty`'s only argument. Generally we start declaration of *predicates* with zero arguments. Then we add new arguments by connecting the node which represents the *predicate* to some *class*. In this manner types of predicate arguments will be defined as well. Various information e.g. the order of predicate arguments can be displayed on demand in the VIZ's property editor.

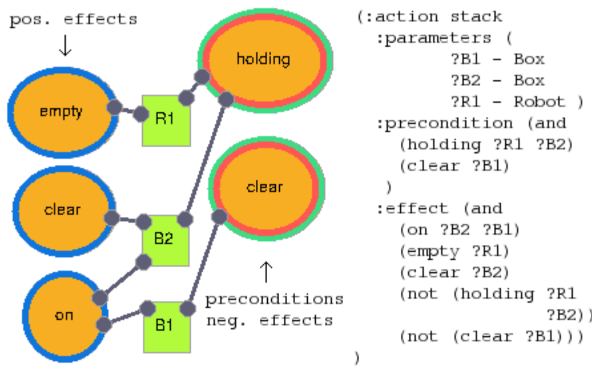


Figure 2: Planning operator example

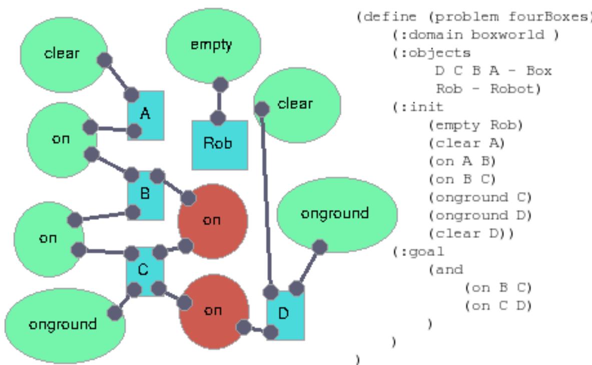


Figure 3: Planning problem example

Definition of planning operators

In Figure 2 we can see a diagram representing planning operator `stack`. Rectangular nodes in this diagram represent *variables*. We can see only their names, but VIZ allows to set their *class* as well. Notable difference is in visualisation of *predicates*. Predicates in operator *Op* are divided into three disjoint sets ($effect^+$,$effect^-$,*precond*). This is marked with three different colours. In the first set, there are predicates appearing only in $effect^+$ (we assume that such predicates do not appear in $precond$ neither in $effect^-$). The second set consists of predicates which appear in $precond$ but not in $effect^-$ (not shown in Figure 2). Finally, there is a set of predicates from $precond$ that also appear in $effect^-$.

Description of planning problems

The diagram for describing planning problems can be seen in Figure 3. Rectangle nodes represent *objects* and eliptic nodes (*predicates*) help to describe the state of the world. Distinct colors of *predicate* nodes are used to distinguish whether the described state is an initial state of the world or a condition which has to be fulfilled in the goal state.

Features and restrictions

Program VIZ is restricted to simple STRIPS planning domains with typing. It covers the following key features:

- class inheritance in the language declaration
- n-ary predicates with possibility of overloading
- basic consistency checking (e.g. missing predicate arguments, inconsistencies caused by changes in the language declaration)
- export of diagrams as `.png` images
- export and import from/to XML (in special format)
- export into PDDL (import from PDDL is not yet supported)

Conclusions

The presented system provides a comprehensive graphical interface which can assist users when designing simple planning domains through their visual representation. It can be also used for educational purposes. The proposed concept can be extended to allow design of more complex domains (e.g. functional symbols, conditional effects). Future development will be focused on the process of knowledge extraction from the informal problem description. VIZ is available from *http://clp.mff.cuni.cz/Viz.html*.

Acknowledgements

The research is supported by the Czech Science Foundation under the contract P103/10/1287.

We would like to thank Roman Barták for help with proof reading of this paper.

References

Simpson, R.M.; Kitchin, D.E.; McCluskey, T.L.:*Planning domain definition using GIPO.* Knowledge Engineering Review, 22 (2): 117-134 (2007)

Vaquero, T. S.; Romero, V. M. C.; Tonidandel, F.; Silva, J. R.:*itSIMPLE 2.0: An Integrated Tool for Designing Planning Domains.* In Proceedings of International Conference on Automated Planning & Scheduling (ICAPS 2007), pp. 336-343, AAAI Press (2007)

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; Wilkins, D.: *PDDL - the planning domain definition language.* Technical report, Yale Center for Computational Vision and Control (1998)

Slaney, J.K,; Thiébaux, S.:*Blocks World revisited.* Artif. Intell. (AI) 125(1-2):119-153 (2001)