Translating HTNs to PDDL
A Small Amount of Domain Knowledge Can Go a Long Way

Ron Alford    Ugur Kuter    Dana Nau

University of Maryland, College Park

IJCAI-09 Technical Presentation
Motivation

- HTNs allow domain authors to incorporate domain specific information on how to find solutions.

- Traditional HTN planners need everything fully specified.
  - They require complete HTN domain descriptions.
  - Otherwise they cannot generate solutions to the domain's planning problems.

- Writing HTN descriptions can be a complicated task.

```
(distance ?origin ?destination ?d)
(at-vehicle ?vehicle ?o1)
(distance ?o1 ?origin ?d1)
(gpm ?vehicle ?gpm)
(call >= ?gas (call * ?gpm (call + ?d1 ?d)))
(height-cap-r ?route ?height-capr)
(weight-cap-r ?route ?weight-capr)
(weight-p ?package ?weight)
(weight-v ?vehicle ?weight-v)
(or (and (not (expected ?vehicle ?route ?value))
    (call <= (call + ?weight-v ?weight) ?weight-capr))
    (and (expected ?vehicle ?route ?value)
    (call <= (call + (call + ?weight-v ?weight) ?value) ?weight-capr)))
(call >= ?height-capr ?height))
(:ordered (set-next ?vehicle ?origin)
    (:immediate add-exp-weight ?vehicle ?route ?weight)
    (:immediate at-vehicle ?vehicle ?origin)
    (:immediate !!delete-protection (next ?vehicle ?origin))
    (:immediate set-next ?vehicle ?destination)
    (:immediate load ?package ?vehicle ?origin)
    (:immediate !!delete-protection (next ?vehicle ?destination))
    (:immediate del-exp-weight ?vehicle ?route ?weight)
    (:immediate unload ?package ?vehicle ?destination)))
```
Simple Example: An Office Delivery Task

- Complete HTN

- Allow domain author to specify only part of the HTN
- Would rather let the planner figure out how to insert `move` and `open door` actions

- Incomplete HTN
Contributions

- An automatic translation from HTNs to PDDL
  - Classical solutions correspond to valid HTN decompositions
  - Translation runs in linear time & space

- Can translate *incomplete* HTN domain descriptions:
  - Domain descriptions that don’t contain enough for an HTN planner to work on
  - Classical planner can fill in the details in its own way

- Experiments using the Fast Forward (FF) planner.
  - Even small amounts of HTN knowledge substantially improved FF’s running time
Outline

- Translate each HTN operator
- Translate each HTN method
- Experimental Results
Some operators are uncontrolled by the HTNs
- I.e., they are missing from the HTN domain description
- Classical planner can insert these at any point during the decomposition

These operators pass through the translator unchanged
For controlled operators
- Can only insert operator when called as a subtask of a method
- Need to prevent planner from inserting operator until it’s called

How to achieve this? A control predicate
- Added as a precondition to operator
- Asserted by a calling methods
- Retracted when operator fires

Operator: \textit{pickup}(p, i)
\begin{align*}
\text{name} &= \textit{pickup}(p, i) \\
\text{pre} &= \{ \text{do}_{\textit{pickup}}(p, i), \ldots \} \\
\text{eff} &= \{ \neg \text{do}_{\textit{pickup}}(p, i), \ldots \}
\end{align*}
When translating a method, we need to:

- Check the method’s preconditions
- Manage the subtasks as they decompose
- Split the control structure over multiple operators

Method: \( \text{deliver}(p, i, g) \)

- \( \text{name} = \text{deliver}(p, i, g) \)
- \( \text{pre} = \{\ldots\} \)
- \( \text{subtasks} = \{\text{pickup}(p, i), \text{putdown}(p, g)\} \)
Method: \( \text{deliver}(p, i, g) \)

- \( \text{pre} = \{ \ldots \} \)
- \( \text{subtasks} = \{ \text{pickup}(p, i), \text{putdown}(p, g) \} \)

### Operator: method head

- \( \text{pre} = \{ \ldots \} \)
- \( \text{eff} = \{ \} \)

### Operator: subtask 1 controller

- \( \text{pre} = \{ \} \)
- \( \text{eff} = \{ \text{do}_{\text{pickup}}(p, i) \} \)

### Operator: subtask 2 controller

- \( \text{pre} = \{ \} \)
- \( \text{eff} = \{ \text{do}_{\text{putdown}}(p, g) \} \)

- **One operator for the method head**
  - Checks method preconditions

- **One controller operator for each subtask**
  - Each controller starts a subtask
  - Uses control predicates mentioned earlier
Method: \textit{deliver}(p, i, g)

\begin{itemize}
  \item Method head operator instantiates parameters for subtasks
  \item Each subtask controller starts next task
\end{itemize}

\begin{itemize}
  \item \texttt{pre} = \{\ldots\}
  \item \texttt{subtasks} = \{\texttt{pickup}(p, i), \texttt{putdown}(p, g)\}
\end{itemize}

\begin{itemize}
  \item Operator: method head
    \begin{align*}
      \texttt{pre} & = \{\texttt{do}_{\textit{deliver}}(p, i, g), \ldots\} \\
      \texttt{eff} & = \{\texttt{do}_{\textit{subtask}\,1}(p, i, g)\}\end{align*}
\end{itemize}

\begin{itemize}
  \item Operator: subtask 1 controller
    \begin{align*}
      \texttt{pre} & = \{\texttt{do}_{\textit{subtask}\,1}(p, i, g)\} \\
      \texttt{eff} & = \{\texttt{do}_{\textit{subtask}\,2}(p, i, g), \ldots\}\end{align*}
\end{itemize}

\begin{itemize}
  \item Operator: subtask 2 controller
    \begin{align*}
      \texttt{pre} & = \{\texttt{do}_{\textit{subtask}\,2}(p, i, g)\} \\
      \texttt{eff} & = \{\ldots\}\end{align*}
\end{itemize}
Need to distinguish between method invocations at different depths of recursion. So:

- Add a new parameter to each method: \textit{level}
- Parameter instantiated with current recursion depth
Method: \textit{deliver}(p, i, g)

\begin{align*}
\text{pre} &= \{\ldots\} \\
\text{subtasks} &= \{\text{pickup}(p, i), \\
&\quad \text{putdown}(p, g)\}\end{align*}

- Add a \textit{level} parameter to the subtask controllers
  - Instantiate to the current recursion depth
- Subtask controllers increment level before calling task
- Operators decrement level
- Last subtask operator leaves level alone
  - Makes tail recursion cheap

Operator: method head

\begin{align*}
\text{pre} &= \{\text{do}_{\text{deliver}}(p, i, g), \\
&\quad \text{level}(v), \ldots\} \\
\text{eff} &= \{\text{do}_{\text{subtask1}}(p, i, g, v)\}\end{align*}

Operator: subtask 1 controller

\begin{align*}
\text{pre} &= \{\text{do}_{\text{subtask1}}(p, i, g, v), \\
&\quad \text{level}(v)\} \\
\text{eff} &= \{\text{do}_{\text{subtask2}}(p, i, g, v), \\
&\quad \text{increment } v, \ldots\}\end{align*}

Operator: subtask 2 controller

\begin{align*}
\text{pre} &= \{\text{do}_{\text{subtask2}}(p, i, g, v), \\
&\quad \text{level}(v)\} \\
\text{eff} &= \{\ldots\}\end{align*}
For complete HTN descriptions:

- HTN problem solvable \rightarrow \text{Classical translation solvable} \leftarrow \text{HTN solution}
- \text{HTN solution} \rightarrow \text{Classical solution} \leftarrow \text{HTN solution}

For incomplete HTN descriptions:

- HTN problem solvable \rightarrow \text{Classical translation solvable} \leftarrow \text{HTN solution}
- \text{HTN solution} \rightarrow \text{Classical solution} \leftarrow \text{HTN solution}

Since the HTN description is incomplete, the classical planner can come up with solutions not mentioned in the HTN.
Experiments

- In domains that are hard for a classical planner, how much can we improve performance via partial HTN translations?

- Took FastForward (FF) [Hoffmann and Nebel, 2001]
  - Investigated how well it did on its own
  - Investigated how it did with a translated HTN
    - Used the simplest HTNs we could find
    - Some of the HTNs were incomplete

- 3 domains, about 5000 experiments:
  - Towers of Hanoi
  - Blocks World
  - An office delivery domain
Results: Towers of Hanoi

HTN Description:
- Move smallest ring to next tower
- Move whichever other ring can move
- Repeat

Varying number of rings
- Each point average of 100 runs
  - Run times vary!
- FF with the translated HTNs runs exponentially better

<table>
<thead>
<tr>
<th>Number of Rings</th>
<th>Total CPU Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FF w/ Original Domain</td>
</tr>
</tbody>
</table>

Alford, Kuter, Nau
HTN Description
- Pick up block
- Put down block either:
  - On table
  - Final location
- Repeat

“Complete” but inefficient HTN domain description
Vary number of blocks, 100 problems of each size.
- FF with translated HTNs can solve problems up to 85 blocks in size
- Can only do 25 block problems without HTNs
HTN Description:

- Pick up a package
- Put down package in goal location
- repeat

- Incomplete HTN description
- HTNs order ‘pickup’/‘putdown’ actions
- ‘move’ and ‘open’ actions uncontrolled by HTNs
- Planner is free to intersperse ‘move’ and ‘open’ actions in the HTN decomposition
Conclusions

- Translation from HTNs into PDDL
- Allows domain authors to specify small amounts of HTN domain knowledge
- Translated HTNs can increase FF’s performance by several orders of magnitude
  - Even when the HTNs are incomplete
Test against dedicated HTN planners
- Initial results show FF outperforming SHOP2 with the Towers of Hanoi HTN domain description.
See how well classical heuristics benefit HTN planning
Create a native incomplete-HTN planner
PDDL Planning

PDDL Planning Problem:
\[ P = (s_0, g, O) \]
- \( s_0 \) is the *initial* state
- \( g \) is the *goal* (a set of ground literals of \( L \))
- \( O \) is a set of operators

Each operator \( o \in O \) is a triple: \( o = (\text{name}(o), \text{pre}(o), \text{eff}(o)) \)
- \( \text{name}(o) \) is \( o \)'s name and argument list
- \( \text{pre}(o) \) a formula called \( o \)'s *preconditions*
- \( \text{eff}(o) \) a set of literals called \( o \)'s *effects*
HTN Planning:

- Tasks represent activities: \( t(x_1, \ldots, x_q) \)
  - *primitive* tasks correspond to operator names
  - *nonprimitive* tasks are implemented by methods
- Methods: \( m = (\text{name}(m), \text{task}(m), \text{pre}(m), \text{subtasks}(m)) \)
  - Methods take a task, and decompose it into multiple subtasks
  - For our purposes, the subtasks must be completed in order

```
    a_loop
   /   \
/     \
pickup  putdown  a_loop
```
Example: Office Delivery

Actions:

- $\text{pickup}(p, i)$ - Pickup package $p$ at location $i$
- $\text{putdown}(p, i)$ - Putdown package $p$ at location $i$
- $\text{open}(d)$ - Open a door between rooms
- $\text{move}(i_1, i_2)$ - Move between rooms
Subtasks totally ordered

Upper bound on depth of recursion
  - Except: tail recursion unbounded
  - Call this *non-tail height*
Non-Tail Height

- **non-tail height**: Level of method decomposition in a solution, ignoring tail decomposition.
- Often depends only on the methods, not the individual problem instances.
- Denote level by:
  - Constants $d_0, d_1, \ldots, d_H$
  - Ordering over constants: $\text{next}(d_1, d_2), \text{next}(d_2, d_3), \ldots$
  - Predicate level($d_i$) for current level
name = \textit{deliver\_one}_{\text{head}}() \\
\textit{pre} = \{\textit{do}_{\text{deliver}}(), \textit{level}(v), \textit{at}(p, i), \textit{goal}(p, g)\} \\
\textit{eff} = \{\neg\textit{do}_{\text{deliver}}(), \textit{do}_{\text{deliver\_all}}_{1}(p, i, g, v)\}

name = \textit{deliver\_one}_1(p, i, g, v) \\
\textit{pre} = \{\textit{do}_{\textit{deliver\_one}}_1(p, i, g, v), \textit{level}(v), \textit{next}(v, w)\} \\
\textit{eff} = \{\neg\textit{do}_{\textit{deliver\_one}}_1(p, i, g, v), \neg\textit{level}(v), \textit{level}(w), \\
\textit{do}_{\textit{pickup}}(p, i), \textit{do}_{\textit{deliver\_one}}_2(p, i, g, v)\}

name = \textit{deliver\_one}_2(p, i, g, v) \\
\textit{pre} = \{\textit{do}_{\textit{deliver\_one}}_2(p, i, g, v), \textit{level}(v), \textit{next}(v, w)\} \\
\textit{eff} = \{\neg\textit{do}_{\textit{deliver\_one}}_2(p, i, g, v), \neg\textit{level}(v), \textit{level}(w), \\
\textit{do}_{\textit{putdown}}(p, i), \textit{do}_{\textit{deliver\_all}}_2(p, i, g, v)\}
name = $\text{deliver}_\text{head}()$

pre = \{\text{do}_{\text{deliver}}(), \text{level}(v), \text{at}(p, i), \text{goal}(p, g)\}

eff = \{\neg\text{do}_{\text{deliver}}(), \text{do}_{\text{deliver}_\text{all}}(p, i, g, v)\}

name = $\text{deliver}_\text{one}_1(p, i, g, v)$

pre = \{\text{do}_{\text{deliver}_\text{one}_1}(p, i, g, v), \text{level}(v), \text{next}(v, w)\}

eff = \{\neg\text{do}_{\text{deliver}_\text{one}_1}(p, i, g, v), \neg\text{level}(v), \text{level}(w), \text{do}_{\text{pickup}}(p, i), \text{do}_{\text{deliver}_\text{one}_2}(p, i, g, v)\}

name = $\text{deliver}_\text{one}_2(p, i, g, v)$

pre = \{\text{do}_{\text{deliver}_\text{one}_2}(p, i, g, v), \text{level}(v), \text{next}(v, w)\}

eff = \{\neg\text{do}_{\text{deliver}_\text{one}_2}(p, i, g, v), \neg\text{level}(v), \text{level}(w), \text{do}_{\text{putdown}}(p, i), \text{do}_{\text{deliver}_\text{all}_2}(p, i, g, v)\}
Control Predicates

- deliver package
- pickup .................. putdown
- move, open, move, ...

Invoke uncontrolled tasks whenever the classical planner wants to invoke them

Controlled tasks only invoked as subtasks of the method

How to achieve this? A control predicate

- Put a new precondition into pickup and putdown
- Assert this precondition in the PDDL translation of the method

Alford, Kuter, Nau