Reducing the Representational Distance Between Application Domain Experts and AI Planning Technology: a compilation approach

Peter Jarvis
Artificial Intelligence Applications Institute
The University of Edinburgh
80 South Bridge, Edinburgh, EH1 1HN, UK
Peter.Jarvis@ed.ac.uk

Graham Winstanley
School of Computing and Mathematical Sciences
The University of Brighton
Lewes Road, Brighton, BN2 4GJ, UK
G.Winstanley@brighton.ac.uk

Abstract

We present a compilation-based approach to reducing the representational distance between application domain experts and AI planning technology. The approach combines a representation designed to match the structure of human expertise in the construction industry with an established planning technique. The design of this representation is derived from a study carried out with experts in the industry. This study shows that expertise in the industry is centred on the components of a building and organised into a subcomponent structure. We demonstrate by encoding the results of this study into a HTN formalism that such formalisms fragment expert knowledge. This fragmentation leads to a large representational distance between expert and formalism, making the task of encoding and maintaining a planner knowledge base a complex one. Our solution is to provide a representation designed around the modelling requirements of the construction industry and then to compile HTN schemata from that representation. We argue that this union reduces the representational distance between expert and formalism, thus lowering the complexity of the knowledge encoding and maintenance tasks, whilst still exploiting powerful AI planning techniques. We conclude by proposing further investigations of this type with the aim of providing a library of domain-oriented formalisms from which a knowledge engineer may choose an appropriate representation for a given domain.

Introduction

Knowledge engineering issues must be addressed if AI planning technologies are to move successfully from the laboratory to the "real world" [Chien 1996]. Significant progress has been made in this area over recent years. Planning methods, for example, have been categorised within a framework appropriate for guiding knowledge engineers [Valente 1995, de Barros et al 1996, Benjamins et al. 1996, Kingston et al. 1996] and tool support for verifying and debugging planner knowledge bases is emerging [Chien 1996].

In this paper, we address the issue of the representational distance between the application domain knowledge and planner formalism. Our approach is centred upon the compilation of a Hierarchical Task Network (HTN) planner [Tate 1977] formalism from a representation designed to match the structure of human expertise in the construction industry. We argue that current HTN formalisms, like previous planner method characterisations (e.g. [Kambhampati 1995]), are more oriented to the needs of the technology than the knowledge engineer. By providing a representation designed around the structure of human expertise, the representational distance that must be negotiated by the knowledge engineer is reduced.

Our motivation and approach are described in a number of stages. First, the results of a study that examined the structure of expertise in the construction industry are presented. The difficulties encountered when mapping the results of this study to HTN formalisms are then described. Second, we outline our Dynamically Assessed and Reasoned Task (DART) Network approach to the compilation of HTN schemata from a formalism designed around the expertise structure identified in the study. Finally, we discuss the contribution of our approach and suggest how it may be combined with related work to form a promising research direction for addressing knowledge engineering issues in AI planning.

Modelling Requirements of the Construction Industry

The modelling requirements presented in this section were derived from a knowledge elicitation and modelling process performed in the construction industry over twelve months. This process included meetings with experts and observations at construction sites. It was driven and conceptualised with the support of the KADS methodology [Schreiber et al. 1993]. We also drew on previous case studies carried out at The University of Brighton, UK and the Center for Integrated Facility Engineering (CIFE) at Stanford University, USA. These studies examined the human expertise applied during the planning of flight simulator construction [Marshall 1988, Winstanley et al. 1990] and the organisation of construction plans around work areas [Winstanley and Hoshi 1993].

The modelling requirements derived from this work are presented below in the stages: action modelling, and dependency modelling. These requirements are then placed into the context of the construction planning process through a KADS application model.

Action Modelling

In the construction industry, expert knowledge about actions is organised around the components of buildings. Actions are defined by their relationship with these components. It is from this observation that we define an activity as the union of an object and an action. Consider the example in Figure 1. The activity *Set Out Position (of a) Beam* is the union of the classes *BEAM* and *SET-OUT-POSITION*, where class *BEAM* is a subclass (triangle notation) of the class *OBJECT*, and class *SET-OUT-POSITION* is a subclass of the class *ACTION*.

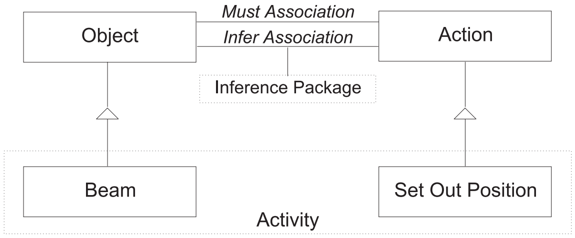


Figure 1: An Activity Defined as the Union of an Object and an Action

Experts use two types of relationship to associate actions with an object. The *Must* association means that an action must always exist as a result of its related object's presence. An *Infer* association means that inference is required to determine if an action is required. Figure 1 includes a graphical representation of the *Infer* relationship. The criteria for determining if a particular action should be associated with an object is represented by an *Inference Package*. The package evaluates to either *yes* (the action should be associated) or *no* (the action should not be associated).

To support reasoning at different levels of abstraction, experts organise objects into a structure of components and subcomponents. An example of this organisation is shown in Figure 2. In this figure, the class *BUILDING* has the subcomponents (diamond notation) class *FOUNDATIONS* through to class *PLANT*. Class *FOUNDATIONS* is further decomposed into the class *BEAM*. An expert reasoning at the level of the *FOUNDATIONS* would expect decisions made at this level to also affect all the subcomponents of the *FOUNDATIONS*.

The subcomponent relationship between objects introduces two types of relationship between actions and objects. Every component always has an *Abstract Action* associated with it (i.e. abstract actions always of the association type *must*). Abstract actions map to the different levels at which experts reason. When considering a building at the foundations level, an expert may place the constraint that the foundations must be laid before the drains. This constraint is placed on the abstract *LAY* action of class *FOUNDATIONS*, but it is to be followed by all the actions of the *FOUNDATIONS* subcomponents. *Primitive Actions* are only associated with components that do not have subcomponents and they correspond to the actions that will appear in the final construction plan for a building. Primitive actions may be related to components through both the *Must* and *Infer* relationship types.

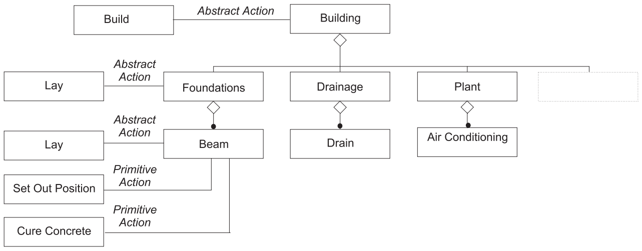


Figure 2: Subcomponent Structure with Abstract and Primitive Actions

Dependency Modelling

Like action knowledge, expert knowledge about dependency is organised around components. Figure 3 shows the relationship *Under* existing between the classes *BEAM* and *DRAIN*. The semantics are that an instance of class *DRAIN* will pass under an instance of class *BEAM*. Thus, the actions that lay the drain must be completed before work commences on the beam. This relationship can be expressed by placing the temporal ordering constraint *Drain.abstract-action* $<$ *Beam.abstract-action*, where *"<"* means before.

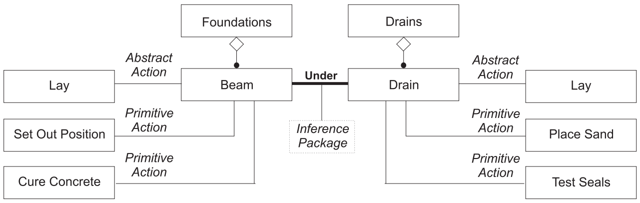


Figure 3: Temporal Dependency Causing Relationship Between Components

Dependency relationships are typed as either *Dependency-Must* or *Dependency-Infer*. The semantics are similar to the *Must* and *Infer* directives that occur between objects and actions. In the *Dependency-Must* case, an ordering constraint must always be placed between the abstract actions of the components joined by this directive. In the *Dependency-Infer* case, inference is required to determine if a constraint is required. There are, for example, occasions when a beam must be laid before a drain. This occurs when the equipment used to lay a beam will damage the drain if it is already in place. The inference attached to this relationship is responsible for determining if an ordering constraint is required.

Application Context

Figure 4 uses a KADS application model [Schreiber et al. 1993] to show the context of the construction planning process. The *Component Knowledge* input describes the set of components that may exist in a building together with their action and dependency assessment knowledge. This is the knowledge described in sections 2.1 and 2.2. Component knowledge is presently located within engineers' experiences, component manuals, and regulation documents.

The *Building Design* input is the design of a specific building. A design is a combination of components. It specifies the components in a specific building, their relationships, and the values of their properties. The building design input is currently represented in the form of design diagrams.

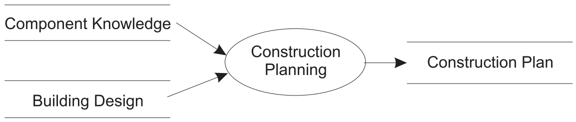


Figure 4: KADS Application Model of the Construction Planning Process

The *Construction Planning* process maps the building design to the component knowledge to generate the actual actions and dependency constraints required in a plan. The result of this reasoning is output as a *Construction Plan.*

Previous Applications of HTN Techniques in the Construction Industry

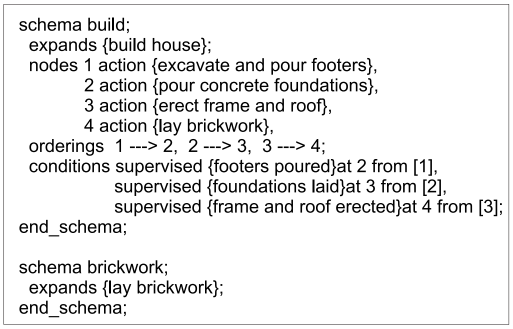
This section determines the extent to which past applications of HTN techniques in the construction industry have addressed the requirements established in Section 2.

The construction industry has been used in the form of Tate's House Building Domain [Tate 1976] to demonstration the capabilities of HTN planning systems since the technique's inception, e.g. [Tate 1977] and [Kartam et al. 1991]. A portion of the domain specification produced for the O-Plan system [Currie and Tate 1991] in the system's Task Formalism [Tate et al. 1994] is shown in Figure 5.

In Tate's House Building domain, planning is initiated by the user specifying that the task *build house* is to be planned. The planner searches its library of possible activities (listed in part in Figure 5) to find a method for achieving this initial task. Figure 5 contains only one suitable method, *schema build.* With a method identified, the planner follows the constraints encoded in that method by constructing a plan containing four actions; one for each of the *nodes* in *schema build.* These actions are constrained in accordance with the *orderings* and *conditions* statements in the schema.

The actions inserted into the plan from *schema build* are all non-primitives. That is, each is at a higher level of abstraction than the planner is required to reach. The planner therefore searches for methods that achieve each of these non-primitive actions. In the case of the action *lay brickwork*, the schema *brickwork* will be selected. This process continues until the plan contains only primitive actions. For a complete description of HTN planning, see [Erol 1995] or [Jarvis 1997].

Planning problems in the construction industry require the capability to reason at different levels of abstraction. The encoding of Tate's House Building demonstrates that HTN planners have this capability. This observation confirms Drummond's [1994] thesis that it is this capability that has enabled the industrial success of HTN technology. Knoblock [1996] also notes that HTN techniques underpin almost all industrial applications of planning technology.



```
schema build;
  expands {build house};
  nodes 1 action {excavate and pour footers},
           2 action {pour concrete foundations},
           3 action {erect frame and roof},
           4 action {lay brickwork},
  orderings  1 ---> 2,  2 ---> 3,  3 ---> 4;
  conditions supervised {footers poured}at 2 from [1],
                supervised {foundations laid}at 3 from [2],
                supervised {frame and roof erected}at 4 from [3];
end_schema;

schema brickwork;
  expands {lay brickwork};
end_schema;
```

Figure 5: Task Formalism Representation of Tate's House Building Domain

The representation of Tate's House Building Domain does not meet the application requirements of the construction industry specified in Section 2.3. To support construction planning, a representation must distinguish between the generic component knowledge and a specific design. The planner's task is to apply the generic component knowledge to a specific design. Tate's House Building Domain does not make this distinction. Instead, it encodes a specific design intertwined with the knowledge relevant to the components in that design. Such a representation would be of limited commercial utility in the construction industry. In order to plan a different design, a new encoding would have to be produced.

The following sections consider the issues encountered when attempting to provide an encoding of the construction industry that meets the application and representational requirements defined in Section 2.

Mapping the Modelling Requirements of the Construction Industry to HTN Representational Devices

The section provides a HTN encoding designed to meet the modelling requirements of the construction industry and draws conclusions about the suitability of the technique in this domain. It is the conclusions drawn from this analysis that motivate our DART-Network approach defined in Section 5.

Encoding Action Knowledge

This section outlines an encoding of the action knowledge identified in Section 2.1 in two stages. The first considers the issues surrounding objects, the subcomponent relationship, and abstract actions. The second considers primitive actions.

Figure 6 shows a HTN encoding of the *building design* and *component knowledge* inputs to the construction planning process. The *building design* input, at the top of the figure, states the components in a building, how they are related, and the values of their attributes. When writing the *component knowledge* input shown in the second part of the figure, the first issue is the provision of schema that will refine the top-level task of any design. Within the construction domain, all top-level task specifications will be of the form *build ?Building.* Class *BUILDING* is the highest level component in the sense that all other components are subcomponents of it, and it is not the subcomponent of any other component. We term the highest level component in a design as being the *Project* component. Schema *build* in Figure 6 will refine the initial task, *build ?Building.* The schema contains a node corresponding to the object - action pairing of all possible subcomponents of class *BUILDING*. From the *building design* specification in the example, class *FOUNDATIONS* is the only possible subcomponent of a building. Hence, schema *build* contains one node.

The second issue to consider is the provision of a schema that refines the action *lay ?FOUNDATIONS* that will be generated by *schema build* to include the subcomponents of class *FOUNDATIONS*. The only subcomponent of this class in Figure 6 is the class *BEAM*. Looking back at Figure 2, the subcomponent relationship between the classes *BEAM* and *FOUNDATIONS* is not as simple as that between the classes *BUILDING* and *FOUNDATIONS*. In this case, the relationship contains a multiplicity ball (black circle notation) at the beam end of the relationship. The ball notation indicates that any number of instances of class *BEAM* may be associated with a single instance of class *FOUNDATIONS*. The encoding of this second transformation is shown in the *lay_foundations* schema within Figure 6. To account for the multiplicity of the relationship, the Task Formalism's *foreach* construct is used. This construct will generate a *lay ?BEAM* action for each instance of class *BEAM* associated with an instance of class *FOUNDATIONS*.

```
;;; Building Instance Specification
;;; as the Building Design Input to the
;;; Construction Planning Process

supermarket: Building
the-foundations: FOUNDATIONS
beam1, beam2, beam3: BEAM

sub.supermarket   = {the-foundations}
sub.the-foundations = {beam1, beam2, beam3}

formwork.beam1 = {custom}
formwork.beam2 = {prefabricated}


;;; Component Knowledge
;;; input to the Construction Planning Process

schema build;
   expands {build ?BUILDING};
   nodes 1 action {lay ?FOUNDATIONS},
end_schema;

schema lay_foundations;
   expands {lay ?FOUNDATIONS};
   nodes N foreach action {lay ?BEAM} for ?BEAM over
              set of Beams related as sub components of
               ?FOUNDATIONS;
end_schema;
```

Figure 6: Encoding of the Component Knowledge and Building Design Inputs to the Planning Process in a HTN Formalism

The encoding scheme described above is formalised into a set of guidelines in Figure 7.

1. For each component C that may have subcomponents, write a schema with the name "C.abstract-action C.component-name"

2. For each possible subcomponent SC of C write a node statement within that schema of the form "SC.abstract-action SC.component-name"

3. If the cardinality between C and SC is not 1:1, use a foreach statement to generate the correct number of "SC.abstract-action SC.compoenent-name" actions.

Figure 7: Guidelines for Translating a Subcomponent Hierarchy into a HTN Formalism

The second stage of the encoding process must address the primitive actions that are associated with components at the bottom of the subcomponent hierarchy, accounting for the *infer* and *must* relationship types. We define the components at the bottom of the hierarchy as *primitive components* because they do not have subcomponents.

Each combination of *primitive actions* that can be used to construct a *primitive component* can be viewed as a possible construction method for that component. With this definition, each construction method can be mapped to a HTN formalism as a schema, and the conditions under which each method should be applied expressed as *only_use_if* filter conditions. The top part of Figure 8 sketches the semantics of this encoding for the methods that may be used to accomplish the task of laying a beam. Each of the methods contains a number of *primitive actions*. If *condition* is equal to *v1*, then the actions in *method-a* should be used to accomplish the task of laying the beam. Otherwise, *method-b* should be used.

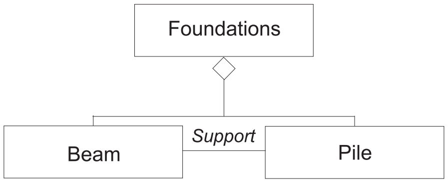
```
Semantics:
to accomplish task of laying a beam
   if condition = v1 then
         use method-a
   else
         use method-b
   end if

Encoding:
schema method-a
   only-use-if condition = v1 ;;; if condition
    …
end_schema;

schema method-b
    only_use_if condition !=v1 ;;; negation of the if condition for the else method
    ...
end_schema;
```

Figure 8: Semantics and Template for the Translation from Construction Methods to HTN Schemata

When encoding *if-then-else* structures, HTN filter conditions require the redundant specification of the condition under which the *else* branch should be exercised. In a HTN system, if the constraint *condition = v1* is placed on *method-a*, then that method cannot be used if this constraint does not hold. However, because of the way HTN planners operate, *method-b* may also be selected if *condition = v1*. For the semantics of the *if-then-else* structure to correctly hold, the domain writer must add the negation of the *if* condition into *method-b*. The lower part of Figure 8 provides a template for encoding of *if-then-else* structures into an HTN formalism.

Encoding Dependency Knowledge

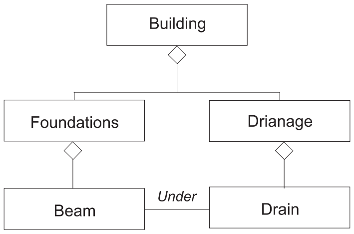
Figure 9 shows two classes related through the *Support* relationship. The implication of this relationship is that the actions relating to class *PILE* must be performed before those of class *BEAM*, as the *PILE* supports the *BEAM*. Following the encoding guidelines in Section 4.1 will produce a schema *lay-foundations* that will include the nodes *lay ?beam* and *drive ?pile*. The *Support* dependency constraint may be expressed as an ordering constraint between these two nodes. This encoding is also shown in Figure 9.



```
schema lay_foundations
   expands {lay ?FOUNDATIONS}
   nodes 1 action {lay ?beam},
            2 action {lay ?pile};
   orderings 2-->1;
end_schema;
```

Figure 9: Dependency Encoding Example, Same Parentage Case

The translation of the component structure shown in Figure 9 to the schema in that figure was possible because both components are subcomponents of the same component, class *FOUNDATIONS*. We term this situation as two classes sharing the same immediate parent. Figure 10 presents a case where two related components do not share the same immediate parent. When components share the same immediate parent, actions relating to those components will be reside in the same schema. Hence, it is possible to place an *ordering* condition between them. In the example in Figure 9, however, the actions for class *BEAM* will reside in the schema *lay-foundations* and the actions for class *DRAINAGE* in the schema *lay-drainage*. The encapsulation unit of the schema in HTN planning prohibits the placement of constraints directly between schemata. To specify an ordering constraint outside the encapsulation unit of a schema, the domain writer may specify a condition for the planning system to achieve. This encoding is shown in the bottom part of Figure 10.



```
schema lay_drain;
   expands {lay ?drain }
   nodes 1 action {primitive-lay ?drain};
   only_use_for_effects
      {state_of drain} = laid at 1;
end_schema;

schema lay_beam;
   expands {lay ?BEAM }
   nodes 1 action {primitive-lay ?BEAM};
   conditions unsupervised {state_of drain} = laid at 1;
end_schema;
```

Figure 10: Dependency Encoding Example, Different Parentage Case

The encoding in Figure 10 distributes the dependency knowledge between the schemata and requires the planning system to establish a condition that can be completely specified by the domain knowledge available.

HTN Encoding Conclusion

Encoding the construction domain within a HTN formalism is a non-trivial task. A critical contribution to this complexity is that HTN formalisms force the knowledge engineer to think in terms of the planning technology rather than the application domain. For example, the engineer must identify schema refinement as the appropriate HTN mapping of the domain's *subcomponent* relationship. When encoding dependency knowledge, the engineer must identify the relationship between the HTN schema unit of encapsulation, the mapping of the *subcomponent* relationship to schemata, and the resulting immediate parentage issue as the factors that determine how dependency is encoded.

An important secondary effect of this translation is the masking of a schema's rationale. Providing knowledge engineers with the rationale behind an encoding is essential to the efficient maintenance of a knowledge base. Without this information, it is difficult for the knowledge engineer to identify how the elements in a knowledge base are related, and therefore which aspects need to be updated to reflect changes in the application domain.

DART-Network Approach

The architecture of our Dynamically Assessed and Reasoned Task (DART)-Network approach is shown in Figure 11. In existing systems, the domain writer must encode a domain within the *Library of Possible Activities* using a formalism akin to the Task Formalism [Tate et al. 1994]. Section 4 identified the difficulties involved in mapping the construction domain to formalisms of this type. The DART-Network approach regards the *Library of Possible Activities* as the result of a compilation process. The innovation is that the compilation process takes as its input a representation that maps closely to the domain under consideration. Using a domain-oriented formalism simplifies both the encoding and maintenance tasks. By then compiling a domain representation in a *Library of Possible Activities*, the powerful techniques developed by AI planning research may be deployed.

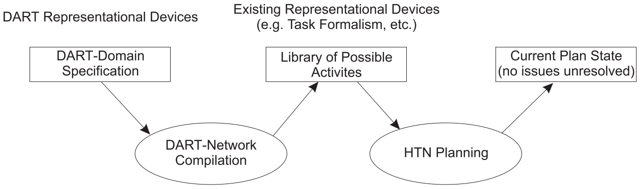


Figure 11: DART-Network Architecture

The following sections outline the DART representational devices and the stages in the compilation process. For a description of the standard HTN planning component of the architecture, see [Kingston et al. 1996].

DART- Network Representational Devices

The DART-Network representational devices are centred on the pattern derived from the modelling requirements of the construction industry that is shown in Figure 12. The domain writers task is to specialise this pattern to capture knowledge in an application domain. As the template is designed to matches the structure human expertise, this encoding process is simplified.

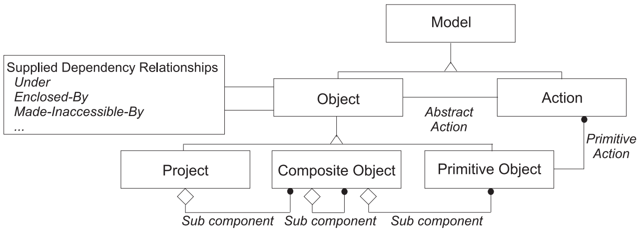


Figure 12: DART-Network Representational Pattern

Class *MODEL* is abstract and will therefore never be instantiated. The class assigns all other classes that are used to model a domain as being of type *MODEL*. Its purpose is to permit operators to be written that work over an entire model.

The abstract class *OBJECT* defines the relationship *abstract action* that is common to all classes that may be used to model the objects within a domain. Objects may be related to other objects through a set of relationships that determine the temporal ordering constraints between actions. The pattern includes a set of predefined relationships that the modeller may extend when tailoring the pattern to an application domain.

The pattern accounts for the different types of object that were identified in the construction domain. Classes *PROJECT, COMPOSITE-OBJECT, PRIMITIVE OBJECT*, and *ACTION* may be refined by the domain writer. Class *PROJECT* makes explicit a special type of class within a DART-Network domain representation. This class may not be the subcomponent of any other class, and is used to represent the overall problem. It the construction domain examples earlier, class *BUILDING* is of this type. Class *COMPOSITE-OBJECT* may be decomposed into other instances of class *COMPOSITE-OBJECT* and class *PRIMITIVE-OBJECT* through the subcomponent relationship. Instances of class *PRIMITIVE-OBJECT* can not have subcomponents but unlike the classes *PROJECT* and *COMPOSITE-OBJECT*, they must be associated with primitive actions.

The representation of action and dependency knowledge is described within the action synthesis and dependency synthesis stages in the following sections.

Action Synthesis

The stages within DART-Network Compilation process are shown in Figure 13.The action synthesis stage described in this section takes a DART problem specification, reasons with that specification, and returns the original specification augmented with actions. This section describes the specification of actions and the *action synthesis* process.

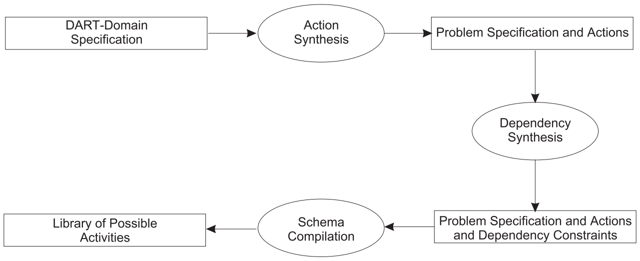


Figure 13: DART-Network Compilation Process

In the case of *abstract-actions*, the domain writer must specify the name of the action class to be associated with each refinement of the class *OBJECT*. The action synthesis algorithm generates a new instance of this action and adds it in the source object's *abstract-action* attribute. For the *primitive-actions*, the domain writer must specify the action class and the directive that is to be is used to determine when each action should be associated with each *primitive object*.

```
Class BEAM
  abstract-action = must(a, lay)
  primitive-action = must(a, test)
                     infer(a, set-out-position,
                     DETERMINE-SET-OUT-POSITION)
```

Figure 14: Encoding of class BEAM's Actions

The encoding of class *BEAM* is shown in Figure 14. In the case of the action *set-out-position*, inference is required to determine if the action should be associated. The *infer* directive includes the name of the inference package that must be invoked to perform this evaluation. An outline of the inference package *DETERMINE-SET-OUT-POSITION* is shown in Figure 15. The package will return *true* if the action should be associated, and *false* otherwise. This modelling approach meets the requirements of the construction domain set out in Figure 1.

The task of the action synthesis algorithm defined in Figure 16 is to visit each instance of a class derived from class *OBJECT* and instantiate the actions defined by the domain writer.

Dependency Synthesis

The dependency synthesis process, shown in context in Figure 13, takes the problem specification augmented with actions and adds the dependency constraints between them.

```
Inference Package DETERMINE-SET-OUT-POSITION
  goal (Known-Value (SET-OUT-POSITON))
  backward chain

  Rule1 cost 0
    if criteria holds then
      SET-OUT-POSITON = True
    end rule1

  Rule2 cost 0
     if criteria holds then
        SET-OUT-POSITON = False
     end rule2
```

Figure 15: Example Inference Package

```
Generate-actions
   For Each  object  O in a design
     Create (O, abstract-action)
     if O is a primitive-object
       Create (O, primitive-actions)
     end if
end Generate-actions
```

Figure 16: Action Synthesis Algorithm

The domain writer must provide a *determine dependency* method for each refinement of the classes *PROJECT*, *COMPOSITE-OBJECT*, and *PRIMITIVE-OBJECT*. Figure 17 shows the determine dependency method for the class *BEAM*. A set of directives are supplied akin to those for action synthesis. The *infer-abstract-action* directive will cause the abstract actions of all the instances related to an instance of class *BEAM* through a *support* relationship to be constrained as occurring before the class' abstract action. This constraint will only be placed if the *DETERMINE-SUPPORT-DEPENDENCY* directive returns true.

```
Class BEAM
  abstract-action = must(a, lay)
  primitive-action = must(a, test)
                     infer(a, set-out-postion,
                     DETERMINE-SET-OUT-POSITION)
  Determine Dependency
                     Infer-abstract-action(support,
                     DETERMINE-SUPPORT-DEPENDENCY)
```

Figure 17: Class BEAM's Determine Dependency Method

The dependency synthesis algorithm visits each instance that is refinement of class *OBJECT* in a model, and invokes their determine dependency methods. This algorithm is outlined in Figure 18.

```
Determine Dependency
  For each O that is a refinement of class OBJECT
     invoke O.determine-dependency
  End
End
```

Figure 18: Dependency Synthesis Algorithm

Schema Compilation

The schema compilation process, shown in context in Figure 13, takes the problem specification augmented with actions and dependency constraints and compiles the *Library of Possible Activities* input to the HTN planning process.

Figure 19 shows an example input to the compilation process. Dependency constraints are ignored for the moment for clarity. The compilation process navigates the instantiated model, generating the HTN schemata that are required to describe the construction of each component. The algorithm works in three stages: task definition, project and composite component definition, and primitive component definition. Each stage is outlined before the overall algorithm is presented.

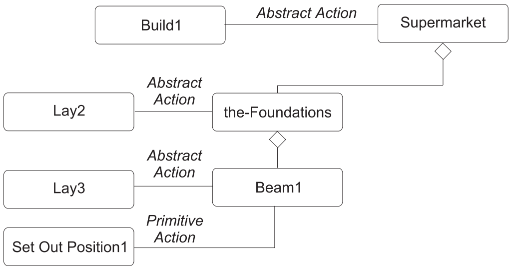


Figure 19: Instance Model with Action Synthesis Completed

The definition of the task the HTN planner is to achieve is derived from the union of a model's *PROJECT* class and that class' abstract action. Figure 20 shows the install task definition that will be compiled from the example in Figure 19.

With the HTN task compilation completed, the compilation process generates the schemata that describe the construction of each instance of the class *COMPOSITE-OBJECT*. The process starts with the single instance of class *PROJECT* in a model and then generates a schema that refines the initial task definition. The schema is generated by including the union of each subcomponent of the project instance and the abstract actions associated with them. In the case of the *Supermarket* component, the schema generated will include the union of each subcomponent and the abstract actions associated with those components. The completed *build supermarket* schema is shown in Figure 20. The results produced when the same process is applied to the component *the-foundations*, is also show in Figure 20.

```
task build-supermarket;
  nodes 1 action {build1 supermarket};
end_task;

schema build-supermarket;
  expands{build1 supermarket};
  nodes 1 action {lay2 the-foundations},
          2 …;
end_schema;

schema lay2-foundations;
  expands{lay2 the-foundations};
  nodes 1 action {lay3 beam1};
end_schema;

schema lay3-beam1
  expands{lay3 beam1}
  nodes1 primitive {set-out-pos beam1};
end_schema;
```

Figure 20: Compiled Task Definition and Schemata

With instances of the class *PRIMITIVE-COMPONENT*, the compilation process generates a schema with a node for each of the primitive actions attached these instances. The schema generated for the instance *Beam1* is shown in Figure 20.

The complete compilation algorithm is shown in Figure 21.

```
schema-compilation (p : PROJECT)
  /* Task Definition */
    create a new schema s with the name "p,"-",p.abstract-action"
    insert node into s "p.abstract-action," ",p)
 /* Schema Compilation */
   for each component c in a model that is either p or subcomponent of p
     if c is of type composite or project then
       create a new schema s with the name "c.abstract-action,"-",c"
       for each subcomponent sc of c
          insert node into s "sc.abstract-action," ",sc"
       end for each
    else ;;; c is of type primitive
       create a new schema s with the name "c.abstract-action,"-",c"
       for each primitive action pa of c
         insert node into s "pa," ",c"
       end for each
    end if
   end for
end schema-compilation
```

Figure 21: DART-Network Compilation Algorithm

Dependency constraints are handled in one of two ways. Ordering constraints between two components that share the same immediate parents are specified in the schema compiled for their parent as an ordering constraint. This encoding matches that shown in Figure 9. The handling of constraints between components that do not share the same parent requires a modification to the HTN planning process in order to relax the encapsulation unit of a schema. Ordering constraints of this type are recorded in the schemata generated for each of the participating components. In the dependent schema, a *known-dependent ?node* constraint is placed, where the *?node* parameter is instantiated to the name of the action object pair on which the action is dependent. In the producing schema, the constraint *provides-condition-for ?node* is placed, where the *?node* parameter is instantiated to the name of the action object pair which is dependent upon this action. Figure 22 shows an example of these constraints. In the example, *drain1* passes under *beam4*, hence the drain must be laid before the beam.

```
schema lay-drain1;
   expands {lay drain1 };
   nodes1 action {primitive-lay1 drain1 };
   conditions provides- condition-for primitive-lay2 beam4 at 1;
end_schema;

schema lay-beam4;
   expands {lay beam4};
   nodes 1  action {primitive-lay2  beam4};
   conditions known-dependent primitive-lay1 drain1 at 1
end_schema;
```

Figure 22: Example *known-dependent* and *provides-condition* Pair

The HTN planning process is modified so that when it encounters a node with a *provides-condition-for* or a *known-dependent* constraint it first examines its plan to see if the other side of the pair is present. If it is, the planner adds an ordering constraint between the two nodes and proceeds as normal. If the other side of the pair is not present, it suspends the processing of the node until the other half is inserted into the plan. When the other half of the pair is included in the plan, the planner adds an ordering constraint between the two nodes and marks the suspended pair as ready for further refinement.

Implementation Status

The DART-Network approach has been implemented in Intellicorp's KAPPA-PC and applied to construction problems containing in the order of 200 components. The knowledge representation and the plans produced have been evaluated and verified by domain experts [Jarvis 1997].

Discussion

We have presented a compilation-based approach that combines a formalism designed to match the structure of human expertise in the construction industry with an established planning technique. This union reduces the representational distance between a knowledge engineer or domain expert and the planning formalism, thus lowering the complexity of the knowledge encoding and maintenance tasks, whilst still exploiting powerful AI planning techniques.

Other approaches to the knowledge engineering issue centre upon the development of a framework for describing planning methods in a format that accounts for the requirements of a knowledge engineer. When viewed in isolation, the contribution of this characterisation framework is a clear understanding of planning methods in terms of the types of knowledge about a problem they must be provided with. Whilst this understanding will assist in the mapping of expert knowledge to planning methods, the knowledge engineer must still encode domain knowledge within representations that, as is demonstrated in this paper, fragment domain knowledge. When viewed in partnership with the DART-Network approach, a potentially more profitable research direction emerges. We view the DART-Network architecture as a demonstration of concept that is intended to motivate further research. By examining other domains, a library of representations designed to match the modelling requirements of knowledge engineers could be constructed. From such a library, it will be possible to identify emergent representations that are applicable to a number of domains. If these domain oriented representations are then mapped to the planning methods through a compilation approach, the resultant architecture would provide a general approach to reducing the representational distance between domain expert and planner formalisms. Thus, reducing the complexity of the knowledge encoding and maintenance tasks.

This paper refines our initial ideas published in [Jarvis and Winstanley 1996a, 1996b]. Complete details of the case study, encoding issues with HTN formalisms, and the compilation process are available over the web in [Jarvis 1997]. Our experiences with the Task Formalism Method obtained whilst encoding construction problems in the Task Formalism are reported [Jarvis and Winstanley 1998], whilst the impact of these conclusions on the development of the Task Formalism Method are reported in [Tate et al. 1998].

References

de Barros, L., Valente, A., and Benjamins, R. 1996. Modelling Planning Tasks. In *Proceedings of The Third International Conference on Artificial Intelligence Planning Systems*,11-18, Edinburgh UK: AAAI Press, ISBN 0-929280-97-0.

Benjamins, R., de Barros, L,. and Valente, A. 1996. Constructing Planners Through Problem Solving Methods. In *Proceedings of the 10th Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW 96)*: Banff.

Chien, S. 1996 Static and Completion Analysis for Planning Knowledge Base Development and Verification. In *Proceedings of The Third International Conference on Artificial Intelligence Planning Systems*, 53-61, Edinburgh UK: AAAI Press, ISBN 0-929280-97-0.

Currie, K., and Tate, A. 1991. O-Plan: the Open Planning Architecture. *Artificial Intelligence* 51(1): North-Holland.

Drummond, M. 1994. On Precondition Achievement and the Computational Economic of Automatic Planning. In: C. Backstrom and E. Sandewall, eds, *Current Trends in AI Planning*, 6-13: IOS press, ISBN 905199 153 3.

Erol, K. 1995. Hierarchical Task Network Planning: Formalisation, Analysis, and Implementation. Ph.D Thesis, Dept. of Computer Science, University of Maryland, USA.

Jarvis, P., and Winstanley, G. 1996a, Dynamically Assessed and Reasoned Task (DART) Networks. In *Proceedings of the Sixteenth International Conference of the British Computer Society Specialist Group on Expert Systems*, Cambridge, UK, ISBN 1 899621 15 6.

Jarvis, P., and Winstanley, G. 1996b. Objects and Objectives: the merging of object and planning technologies. In *Proceedings of the 15th Workshop of the UK Planning and Scheduling Special Interest Group*, Liverpool, UK.

Jarvis. P. 1997. Integration of Classical and Model-Based Planning. Ph.D Thesis, School of Computing and Mathematical Sciences, The University of Brighton, UK. http://www.aiai.ed.ac.uk/~paj/thesis/

Jarvis, P., and Winstanley, G. 1998. Using the Task Formalism Method to Guide the Development of a Principled HTN Planning Solution for the Construction Industry. Submitted to *the 17th Workshop of the UK Planning and Scheduling Special Interest Group*, to be held during September at the University of Huddersfield, UK. http://www.aiai.ed.ac.uk/~paj/paj-pubs.html

Kambhampati, S. 1995. Planning as refinement Search: a unified framework for evaluating design trade-offs in partial-order planning. *Artificial Intelligence*, 76, Special Issue on Planning and Scheduling.

Kartam, N., Levitt, R., and Wilkins, D. 1991. Extending Artificial Intelligence Techniques for Hierarchical Planning, *ASCE Journal of Computing in Civil Engineering.*

Kingston, J., Shadbolt, N., and Tate, A. 1996. CommonKADS Models for Knowledge Based Planning. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-96)*, Portland, Oregon, USA: AAAI
Press. http://www.aiai.ed.ac.uk/~oplan/oplan/oplan-doc.html

Knoblock, C. 1996. Editors Introduction. *IEEE Expert Intelligent Systems and Their Applications*, 11(6).

Marshall, G., 1988, PIPPA: The Professional Intelligent Project Planning Assistant. Ph.D Thesis, Information Technology Research Institute, The University of Brighton, UK.

Reece, G., Tate, A., Brown, D., Hoffman, M., and Burnard, R. 1993. The PRECiS Environment. In *the ARPA-RL Planning Initiative Workshop*, held during the 11th National Conference on Artificial Intelligence (AAAI-93), Washington DC, USA. http://www.aiai.ed.ac.uk/~oplan/oplan/oplan-doc.html

Schreiber, G., Wielinga, B., and Breuker. J. eds. 1993. *KADS A Principled Approach to Knowledge-Based System Development.* Academic Press, ISBN 0 12 629040 7.

Tate, A. 1976. Project Planning Using A Hierarchic Non-linear Planner, Department of Artificial Intelligence Research Report No. 25, Artificial Intelligence Library, University of Edinburgh, 80 South Bridge, Edinburgh, EH1 1HN, UK. House building domain encoding available from http://www.aiai.ed.ac.uk/~oplan/

Tate, A. 1977. Generating Project Networks. In *Proceedings of the International Joint Conferences on Artificial Intelligence*, 888-893.

Tate, A., Drabble, B., and Dalton, J. 1994. O-Plan Version 2.2 Task Formalism Manual. O-Plan Project Documentation, Artificial Intelligence Applications Institute, The University of Edinburgh, UK. http://www.aiai.ed.ac.uk/~oplan/oplan/oplan-doc.html

Tate, A., Polyak, P., and Jarvis, P. 1998. TF Method: An Initial Framework for Modelling and Analysing Planning Domains, In *Proceedings of the AIPS-98 Workshop on Knowledge Engineering and Acquisition for Planning: Bridging Theory and Practice*, held during the Fourth International Conference on AI Planning Systems, Carnegie-Mellon University, USA. http://www.aiai.ed.ac.uk/~oplan/oplan/oplan-doc.html

Winstanley, G., Boardman, J., and Kellett, J. 1990. An Intelligent Planning Assistant in the Manufacture of Flight Simulators. In *Proceedings of the ACME Research Conference*, University of Birmingham, UK.

Winstanley, G., and Hoshi, K. 1993. Activity Aggregation in Model-Based AI Planning Systems. *AI in Engineering Design and Manufacture*, 7(3), Academic Press, 209-228.

Valente, A. 1995. Knowledge Level Analysis of Planning Systems. *SIGART Bulletin*, 6(1), 33-41.

Acknowledgements

The authors would like to thank Stuart Aitken, John Kingston, and Austin Tate for their comments on this paper, and Nigel Shadbolt for his comments on the work that is described in this paper. The modelling requirements were developed with the assistance of The Llewellyn Group of Companies, UK. We also thank the anonymous reviewers for their insightful comments. Errors and omissions remain our own.