Generic Types as Design Patterns for Planning Domain Specification

R. M. Simpson and **T. L. McCluskey**
School of Computing and Mathematics, The University of Huddersfield, Huddersfield, UK
r.m.simpson@hud.ac.uk, lee@zeus.hud.ac.uk

Derek Long and **Maria Fox**
Department of Computer Science, University of Durham, UK
d.p.long@dur.ac.uk, maria.fox@dur.ac.uk

Abstract

In this paper we investigate the use of 'Generic Types' as design patterns to assist in the specification of planning domains. Current planning technology uses induced patterns discovered in a domain specification to speed up plan creation. We argue that such generic types can also be used to help a domain author to develop a design for a domain at specification time using concepts at a much higher level of abstraction than is normally provided by domain specification languages.

Introduction

Research into domain independent AI Planning and Scheduling, has traditionally focused on the development of algorithms to efficiently find solutions to planning problems within the domain. The problems of dealing with what is perceived to be realistically large problems has been very difficult but recent advances in algorithms appear to make the problems more tractable. Perhaps because of the difficulty in developing capable solution generating algorithms knowledge engineering for applications of AI Planning technology is still very much in its infancy. Recent successful AI planning applications (Muscettola *et al.* 1998; A. Tate (editor) 1996) have nonetheless highlighted the problems facing knowledge engineering in planning. Questions raised by such work include issues of how to encode knowledge into domain models for use with planning algorithms. Subsequently concern over the development of knowledge engineering issues in AI Planning has resulted in a set of workshops and initiatives, including (Benjamins, Nunes de Barros, Shahar, Tate and Valente (eds) 1998; PLANET 1999).

In this paper we describe a domain definition strategy and tools to support the knowledge acquisition phase, to be carried out by domain experts rather than experts in AI Planning. We show that planning domains can be constructed using concepts at a much higher level of abstraction than has traditionally been the case in domain independent planning. Traditional languages for the specification of planning domains allow the authors of a new domain great freedom in their choice of representation of the domain details. This freedom is we contend for the most part unnecessary and provides an unwanted conceptual barrier to the development of effective domain definitions. As part of our ongoing project to enhance the tools available for knowledge engineering in planning we recently released a "Graphical Interface for Planning with Objects" called GIPO (Simpson *et al.* 2001). This is an experimental GUI and tools environment for building classical planning domain models, providing help for those involved in knowledge acquisition and the subsequent task of domain modelling. The current work is an enhancement to the *GIPO* tools environment which is supported by EPSRC grant GR/M67421, within the PLANFORM project *http://scom.hud.ac.uk/planform*.

Generic Types

The primary purpose of *generic types* in planning as introduced by Fox and Long (Fox & Long 1997; Long & Fox 2000; 2001) was to provide control information to planning algorithms to boost the speed of finding solutions to planning problems. The underlying conjecture in that work is that if common structural elements can be detected in a domain definition then specialised algorithms can be brought to bear on those elements of the problem to speed up the detection of solutions. With this in mind researchers have now identified a number of candidate generic types that can be found in a range of the domains publically available. The purpose of our research is to investigate the possibility of using generic types as *design patterns* to assist the domain modeller in the construction of an initial specification. Defining a design pattern in his seminal work (Alexander *et al.* 1977), Alexander states that:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over without ever doing it the same way twice. *Christopher Alexander*

There are many potential advantages to presenting the domain modeller with a range of patterns around which a domain can be structured. Just as the concept of design patterns has promoted greater reuse at a higher level of abstraction in software engineering,

it can equally be beneficial for planning domain engineering. The domain modeller can benefit from encapsulated clean and elegant solutions to representation of common domain structures. Another advantage is, again as with software engineering practice, the designer is encouraged to conceptualise the domain at a higher level of abstraction than has typically been the case for modellers working in STRIPS derivative languages such as PDDL. A further benefit is that the use of canonical, "normalised" domain representations supports the opportunity for reuse of deeper domain knowledge associated with the patterns, such as control knowledge, specialised algorithmic problem-solvers and so on.

The language OCL (Liu & McCluskey 2000) which has a design suite of tools *GIPO* (Simpson *et al.* 2001) itself tries to lift the level of generality at which the modeller can design the domain by making the concept of an object and the changes of state that they undergo central to the conceptualisation of the domain. However the research group acknowledges that the task for the domain modellers who are not themselves experts in the field of A.I. planning is still too difficult. Along with other approaches being investigated by the team, this current research is seen as having the potential to help bridge the gap between the tools and techniques usable by a domain expert, who is not necessarily steeped in the technologies of AI, and those that may only be used by experienced practitioners in the field of AI Planning.

In software engineering, design patterns (Gamma *et al.* 1995) are described using stylised natural language templates, combined with UML class diagrams, describing the relationships between the fundamental building blocks of object-oriented software. In planning domains the corresponding notion of a generic type is described using relationships between the fundamental building blocks of planning domain descriptions: sorts (or types), predicates, object states and state transitions (associated with operators). These relationships can be captured graphically, in a diagram rather like a UML class diagram (Figure 1), or more formally, using declarative specifications of the necessary relationships between the components. The formalisation of these descriptions is still the subject of current research, since the precise language should combine expressiveness with precision and tractability. The role of generic types has expanded from being patterns to be identified and exploited, which demands a description that can be matched efficiently against domain descriptions, to that of domain engineering construct, which is not concerned with pattern-matching, but with expressiveness and ease of instantiation.

Broadly, a generic type then defines a class of classes of objects all subject to common transformations during plan execution. Within OCL we refer to *sorts* which are sets of objects all subject to the same characterisation and transformations, in typed-PDDL the range of a type identifies a set of objects all subject to the same characterisation and transformations. A generic

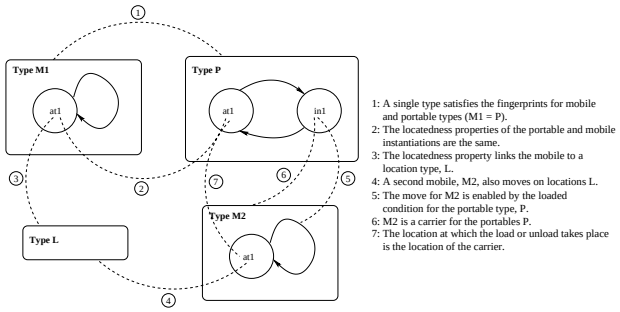


Figure 1: Pattern description for the driver generic type.

type accordingly ranges over the types or sorts of individual domains. The degree of commonality in the characterisation and transformations that these types or sorts must share have been described in the literature in terms of state machines describing the patterns of transformations that the objects undergo.

In the following section we illustrate the way in which a generic type can be used as a design pattern to support a more abstracted view of the structure within a planning domain during the engineering process.

An Extended Example

We will describe the generic type for a "mobile". A "mobile" can initially be thought of as describing the types of objects that move on a map. They can very simply be characterised by the state machine shown in figure 2

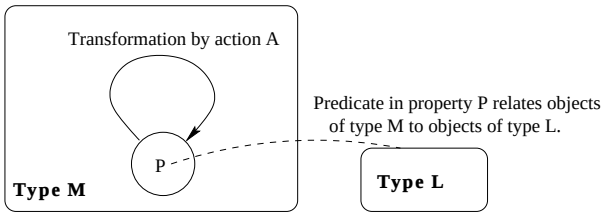


Figure 2: The Mobile Generic Type

In this one-state-machine the state is characterised by the property of the mobile object corresponding to its *locatedness* and the transition is identified by the action causing it to move. For this generic type to be applicable to a type or sort in a particular domain there must be a type such that there is an action that changes the truth value of a n placed predicate $N >= 2$, let us call it at, where one argument identifies the type in question, M, another a value L which changes as a result of the application of the operator. That is the value of L differs in the pre- and post-conditions of the action in the reference to the at predicate. No other predicate referencing M should change truth value in the same action definition in this most basic form of the mobile prototype. The namings of predicates and arguments

may (and will) be different in different instances of the generic type. This is a very weak characterisation of a mobile, in that domains that describe actions that perform transformation on some property of the objects in question might fulfill the above requirements and hence be characterised as a mobile. From the point of view of the designer of planning algorithms this does not represent a problem as it just means that any specialised algorithms identified to speed up processing domains containing such structures will have wider application. From the point of view of someone trying to produce tools to assist domain developers it poses a problem in conveying to the user precisely what is the scope and potential uses of the pattern we have described as mobile. This problem of communication is probably exacerbated when we realise that subtle variations of the pattern need to be distinguished from one another.

Flavours of Mobiles First we characterise the transition made by such a "mobile" in terms of the formula $[at(Mobile0, LocA) \wedge LocA \neq LocB] \Rightarrow [at(Mobile0, LocB)]$ where the compound predicate within square brackets to the left of $\Rightarrow$ describes the object $Mobile0$ prior to application of the action and the formula to the right describes it after the application of the operator. The question arises as to the nature of the relationship between the values of LocA and LocB. In the formula they are required to be distinct and, we will assume, of the same type but their relationship is not otherwise constrained.

In some domains a more complex relationship might be required to hold between the two locations in order to allow movement between them. For example if "a" and "b" are locations then the transition from "a" to "b" can only be made if there is a "road" from "a" to "b". In the case of the logistics domain transition are allowed if both locations are in the same city. In yet other cases e.g. the rocket world the only restriction may be the one we have assumed anyway that the locations are all of the same type and that they are distinct from one another. Distinguishing these differences may be necessary when analysing existing domains with the intention of speeding up solution detection but it does not follow from that that we should provide the domain developer with the freedom to create each variant, with the consequent additional burden of forcing the developer to distinguish conceptually between the different patterns. We must decide whether or not the differences are essential to capturing distinct behaviours of objects in the respective domains or whether they are merely alternative ways of describing the same behaviour and represent nothing deeper than differences in encoding strategies. We believe that there are advantages to requiring the developer to come to terms with a minimal toolkit of canonical concepts to allow them to model their domains. We should only deviate from this if minimality means that the developer must conceptualise a problem at a level of abstraction that is too far removed from a natural way of thinking of the domain, in which case we may introduce features which from the minimalist point of view are redundant. Determining what represents a "natural" way of thinking about domain structures is a matter of experience and of judgement — we anticipate the need to refine the collection of generic types offered as a domain design patterns as their use develops.

Data Structures

Consideration of the very simple model of a mobile described above leads us further to distinguish different elements of a generic cluster. In particular, we need to distinguish between *data structures* that are referenced in the cluster and the dynamic generic types. Data structures are elements within the domain that are captured in predicates that do not change truth value during the application of a plan. In PDDL, data structures are given in the initial state of a problem. An example is the set of connectedness propositions that define the road structure in the truck-world which form a connected graph. Such data structures may be referenced by multiple operator definitions within the planning domain. The dynamic generic types are types or groups of types characterised by the changes in state they make during plan application.

Data structures in planning domains are not normally identified in terms of their structure but are captured implicitly in collections of predicates. In PDDL data structures are to be found in a subset of the propositions defined as *true* in the initial state of a problem definition. In *OCL*, these static propositions that do not change truth value during the planning process are collected together in the *atomic_invariants* section of the domain specification.

Examples

- *Sets* In the logistics domain the proposition *in-city(pgh-po,pgh)* is used as a way of asserting that *pgh-po* is a member of the *set* of locations defined as part of the city *pgh*. Sets may be identified more simply than this. A set may simply be represented as the values of a particular argument in a predicate. We would describe the locations that can be visited in the brief-case world as forming a set, though they are never referred to in the domain specification other than as values of the location argument in the predicate that relates an object such as the "briefcase" to a location. Given this example it might seem that the range of every *typed* variable could be regarded as forming a *set* but we distinguish between *dynamic* and *static* types and only the ranges of static types are regarded as candidates to be identified as sets. The distinction between dynamic and static objects we have discussed in (Simpson *et al.* 2000). To summarise: *dynamic* objects are those that would normally be regarded as changing their properties or relations during plan execution, where as *static* objects do not change. Again referring to the briefcase world,

in the $at(briefcase1, home)$ predicate that describes the location of the briefcase1 as being at home, it would be normal to regard briefcase1 as changing location when moving from "home" to the "office", but we would not think of the locations themselves as changing state simply as a result of the arrival or departure of a briefcase. Hence we regard the "briefcase" as dynamic but the "home" as static.

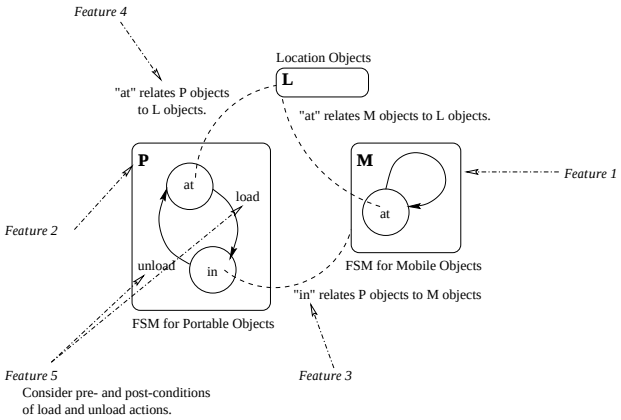
- *Maps* Examples of *maps* can be found in the "travel" world that contains collections of predicates such as $road(a, b)$ in the "init" sections of the problem specifications. The collection taken as a whole for each problem specifies a directed graph which has to be navigated by some *dynamic* object, the locations are again regarded as static.
- *Sequences* Examples of *sequences* are rarer and less obvious in the public domain planning domains. They occur in such domains as the "elevator" domain to show the relationships between floors but we would probably use a map in such an instance to model the relationship. In domains such as "truck" and "ferry" and "rocket" worlds there are single step sequences moving from "full" to "empty", which provides a very primitive way of representing resources in those domains. We generalise this and provide the notion of a sequence to enumerate the stages in the consumption or production of a resource. The use of this idea can be seen in the encoding of fuel levels and space resources in the Mystery and Mprime domains (AIPS'98 Planning Competition) and also in aspects of the encoding of the FreeCell domain (AIPS'00 Planning Competition).

Definitions

- A *map* we define as a named directed graph with nodes identified by simple labels/names and edges identified by a tuple containing the map name and node names. The tuple {x,a,b} identifies that there is an edge in map "x" from node "a" to node "b".
- A *sequence* we define as a fully ordered set with members identified by simple labels/names and a unique named $<$ relation. The tuple $\{< x$,a,b$\}$ identifies that "a" immediately precedes "b" in the "$< x$" sequence.
- A *set* we define as a *set* of items uniquely identified by labels and a predicate name identifying the set. The tuple {x,a} identifies "a" as a member of the set "x".

Defining Core Generic Types

The work done to date primarily identifies a family of types clustering around the notion of a "mobile". Figure 3 shows an initial pattern language for this collection of patterns. We distinguish two forms of mobile, those constrained to move on "maps" and those that move on "sets". The first we call "mobiles" the second we call "carriers". There is then a number of optional components that can be added to both mobiles and carriers. First both may be used to transport other objects. In which case the other objects will make a transition analogous to the "mobile" when the mobile moves but they will also make transitions when "loaded" into the mobile and "unloaded". These objects which we call "portables" can be characterised by state diagrams as shown in figure 4 The behaviour of portables are to be



The following components form the fingerprint for portability:

1. A previously identified mobile generic type, M, and its linked location generic type, L.
2. A new type, P, with a FSM containing two states linked by transitions in both directions.
3. One state of the FSM for P must include a property formed from a predicate linking the P type objects to the M type objects.
4. The *other* state of the FSM must contain a property formed from a predicate linking the P type objects to the L type objects.
5. The operators from which the two transitions in the P type FSM are derived must require an M object to be located at the same location as the P object is located at the appropriate end of the transition.

Note that the *names* of the operators and predicates are irrelevant and that the name of the predicate in feature 4 need not be the same as the locatedness predicate for type M.

Figure 4: The Portable Generic Type

determined by three actions, the action to move the mobile, an action to load the portable into the mobile and an action to unload the portable from the mobile. The state diagram for the portable does not however specify how the movement of the portable is to relate to the similarly structured movement state diagram for the mobile. Given that both describe the same "movement" action there are two plausible ways that they may relate to one another. First the transitions may both be required to take place together, in which case it will be a precondition of the "movement" action that the portable be "in" the mobile before any movement can take place. Portables of this sort we call "Drivers" and a specific transition needs to be defined for each driver participating in the action. The second

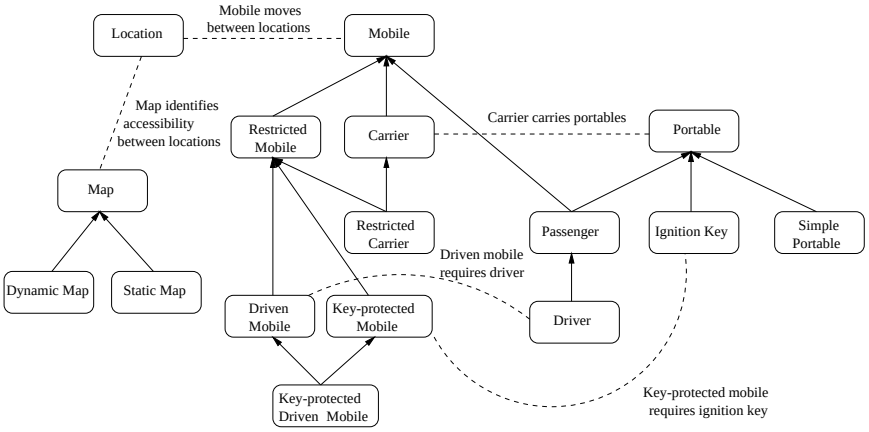


Figure 3: A Hierarchy of Mobile-related Generic Types

case arises when the transition of the portable is conditional on the portable being in the mobile but it is not required that a portable be in the mobile to allow the mobile itself to make the movement transition. In this case the conditional transition will apply to any instance of the portable "in" the mobile. We have therefore distinguished two types of mobile (a) carriers and (b) mobiles both being capable of being associated with two types of portable (i) drivers (ii) portables.

We have not yet exhausted our elaboration of the "mobile" generic type. In a number of domains, including one of the simplest, the "rocket" world, there is a notion of fuel which is a resource to be consumed as a result of the mobile moving. In the rocket world the fuel is consumed in one shot. The rockets starts with fuel full but any movement action results in the rocket being empty of fuel. We can easily see that the consumption of the resource could have been staged and to accomplish this we model the movement action as traversing a step of a sequence on each movement. Given the sequence {full, half, empty} a single movement action might take the rocket from *full* to *half* and from *half* to *empty*. With the notion of resources we can augment any transition as any transition might consume or produce some resource. The movement action may consume fuel, but equally loading or unloading some portable may consume energy.

In the discussion above we have not described features of mobiles that require "dynamic maps" nor "key" enabled actions. We have given an indication of the complexity and flexibility of the "mobile" generic type, viewed as a design pattern.

Composition of Generic Types

The problems of the composition of patterns falls broadly in two. The simple case is that already explored where a complex pattern has many optional but predictable variations. Examples are where a mobile requires a driver or consumes a resource. The more problematic case is where the domain contains two or more patterns where the same object type plays a role in more than one pattern instance. This can happen even in domains just containing mobiles. In a variation of the "hiking" domain we may have cars which can be used to transport the hikers from one centre to another but the hikers themselves may be mobiles in that they also walk from some locations to neighbouring mountain tops. In relation to the car(s) mobile pattern the hikers will play the role of either, or both, the roles of "drivers" and "portables" but in relation to their walking they play the role of mobiles perhaps even with their own portables such as the "tent" which they may carry on some walks.

The problem of composition also occurs where we have conceptually independent patterns. To illustrate, one of the patterns we are working with we call a "bistate" and it represents objects that typically exist in one of two states and there are actions to change back and forth between the states. A canonical example of a bistate would be a switch that can be "off" or "on". In the hiking domain the tent that the hikers sleep in may play a role as "portable" relative to the car and the hikers themselves, but may additionally be modelled as a bistate in that it is typically either "up" i.e. erected or "down". In this case the "erect" transition may be captured by the formula $[down(Tent0)] \Rightarrow [up(Tent0)]$
The "load" and "unload" transitions associated with the tent as portable may be captured as:
$[at(Tent0, LocA)] \Rightarrow [in(Mobile0, Tent0, LocA]$ and
$[in(Mobile0, Tent0, LocA] \Rightarrow [at(Tent0, LocA)]$.
The combination that we require is to associate the "down" state with the "at" state but we cannot simply replace the designation of the "down" state with that of the "at" state because the "at" state carries extra information about the location of the tent. We could not adequately describe the transition of the tent when we take it down as $[up(Tent0)] \Rightarrow [at(Tent0, LocA)]$ as there is no indication as to how the "LocA" variable is to be bound. Obviously the location of the tent is the same as that it had when the tent was

erected. Accordingly to preserve the location information we must merge the arguments in the "at" and "down" states and then propagate additional arguments to the other state descriptors in the merged patterns. In this case the "erect" transition now becomes $[at(Tent0, LocA)] \Rightarrow [up(Tent0, LocA)]$ and the "take down" transition is similarly enhanced.

The strategy shown here is the one that we generally follow in combining instances of patterns where a common state exists between the roles of the merged patterns.

Domain Definition using Generic Types

To enable the domain developer to use the identified generic types to structure a domain we have developed a series of dialogs which we have integrated into the *GIPO* domain development tool. The dialogues allow

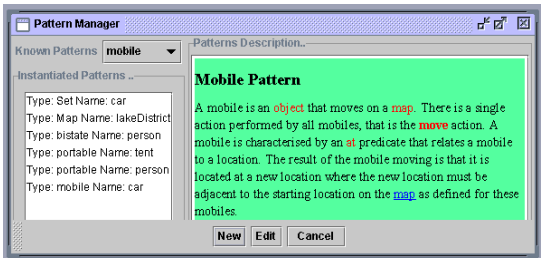


Figure 5: The Pattern Manager

the user to choose the relevant patterns and then tailor them to the problem in hand. In simple cases tailoring is simply a matter of naming the components of the pattern in an appropriate way. In more complex cases the user must add optional components to the pattern again by form filling and in the most complex cases ensure that domains using multiple patterns allow them to interact with each other in the correct way. The set of dialogues form a domain editor in such a way that the user committing her choices in the editing dialogues will result in the formal domain specification being automatically generated. We illustrate the process with snapshots taken from the "Pattern Manager" in figure 5 which is used to control the addition and editing of patterns known and instantiated within the domain. We also show the main dialog for defining the parameters of the "mobile" pattern in figure 6.

Evaluation

The implemented pattern editors that we have produced currently give good coverage of domains featuring "mobiles" of one sort or another. Our evaluation is currently limited to testing to see if we can produce using the pattern editors versions of the domains that have been made available as part of previous *AIPS* competitions. We are judging equivalence of domains not at

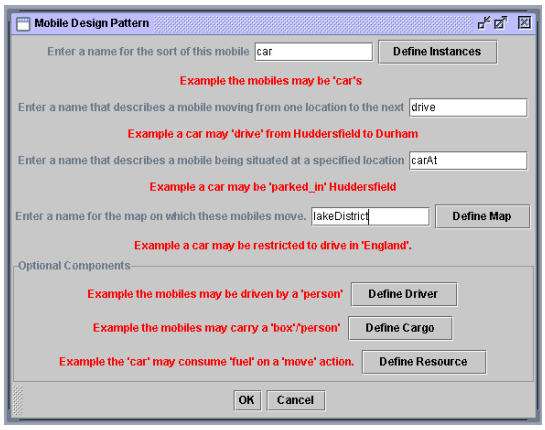


Figure 6: The Mobile Dialog

the level of encoding the operators and problems but rather at the level of generality that would allow us to say that derived solutions to equivalent problems are equivalent. We do not require for example that any planner that works with the original will work with our generated version, as this is not the case even in the rocket world as our encoding uses conditional effects whereas the commonly available originals typically do not. We would also judge domains to be equivalent even where they do not contain the same number of operators or predicates, for example a *move* operator may be either expanded into multiple *move* operators each responsible for moving objects of different sorts or conversely we may contract multiple operators into a single operator dealing with a more general *sort*.

An interesting observation is that though the pattern-directed reconstructions of classic domains are not always identical to the familiar encodings we consider it a strength of the use of patterns that a canonical encoding, with its attendant well-understood behaviours, is used to encode the domains. Nevertheless, it raises an important point about the expected behaviour that a domain is intended to capture. In reconstructing domains we expect to find that there is a correspondence between legal states of the reconstructed domain and the original encoding, with an induced correspondence between plans in the two domains (see figure 7). Confirmation that this correspondence exists forms a reasonable element of the evaluation of the use of the pattern-directed approach to domain construction.

Evaluation of the provision of support for domain construction by domain design patterns is difficult. It is intended that they make domain construction easier, but this is a matter of HCI and could only be empirically evaluated with access to a reasonable sample of potential users. Of course, the developers consider the

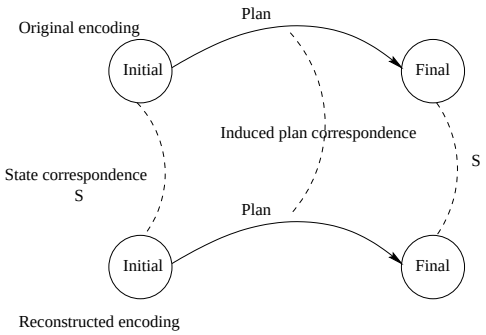


Figure 7: Equivalence between domain encodings.

approach to be an efficient and convenient way to generate domains. A separate dimension of evaluation is to consider the extent to which the patterns provide support across a wide range of domains. For example, one might consider how many benchmark domains can be reconstructed using the patterns, with the patterns providing support for a high-level view of the domain objects and their behaviours. It is of interest to note that many of the benchmark domains include a transportation element (Logistics, Gripper, Briefcase, Grid, Mystery and MPrime are all examples). The Tyreworld domain consists chiefly of interlocking bistate elements (hubs can be up or down, nuts are on or off, loose or tight, and wheels are on or off, inflated or deflated). Many of the other domains contain a construction component (Hanoi, Blocksworld, Assembly and Freecell) and we are currently exploring the implementation of a generic cluster to support construction.

Further Work

The work described above is still work in progress. We continue to develop it at a number of levels. We continue to work on incorporating known "generic types" into the *GIPO* tool and to enhance the facilities within the tool for creating, editing and combining patterns. At the level of the patterns themselves there is still more work to be done in identifying new patterns and elaborating further the existing patterns. We are also working at formulating more formally the rules for combining patterns with an ultimate goal of providing a formal description of patterns and an "algebra" for their composition. Ideally the outcome of this work would be tools to allow the domain designer to develop a wide range of domain definitions without the need to develop the domain in any way at the level of the underlying specification language such as *OCL* or *PDDL*. A further goal of the work is also to provide planning algorithms with information on the instantiated patterns to allow them to use this as control information to inform the planning process itself. We do not expect however that this will eliminate the need for further domain analysis to assist in speeding up planning solution production.

References

A. Tate (editor). 1996. *Advanced Planning Technology: Technological Achievements of the ARPA/Rome Laboratory Planning Initiative*. IOS Press.

Alexander, C.; Ishikawa, S.; Silverstein, M.; Jacobson, M.; Fiksdahl-King, I.; and Angel, S. 1977. *A Pattern Language*. Oxford University Press.

Benjamins, Nunes de Barros, Shahar, Tate and Valente (eds). 1998. Workshop on Knowledge Engineering and Acquisition for Planning: Bridging Theory and Practice. Proceedings of AIPS.

Fox, M., and Long, D. 1997. The Automatic Inference of State Invariants in TIM. *JAIR* 9:367–421.

Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. 1995. *Design Patterns: Elements of reusable software*. Addison-Wesley.

Liu, D., and McCluskey, T. L. 2000. The OCL Language Manual, Version 1.2. Technical report, Department of Computing and Mathematical Sciences, University of Huddersfield .

Long, D., and Fox, M. 2000. Automatic synthesis and use of generic types in planning. In *Proc. of 5th Conference on Artificial Intelligence Planning Systems (AIPS)*, 196–205. AAAI Press.

Long, D., and Fox, M. 2001. Planning with generic types. Technical report, Invited talk at IJCAI'01 (forthcoming Morgan-Kaufmann publication).

Muscettola, N.; Nayak, P. P.; Pell, B.; and Williams, B. C. 1998. Remote Agent: To Boldly Go Where No AI System Has Gone Before. *Artificial Intelligence* 103(1-2):5–48.

PLANET. 1999. *PLANET Knowledge Technical Coordination Unit*. http://scom.hud.ac.uk/planet.

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

Simpson, R. M.; McCluskey, T. L.; Zhao, W.; Aylett, R. S.; and Doniat, C. 2001. GIPO: An Integrated Graphical Tool to support Knowledge Engineering in AI Planning. In *Proceedings of the 6th European Conference on Planning*.