Reuse of Control Knowledge in Planning Domains

Luke Murray
Department of Computer Science
University of Durham, Durham, UK
l.c.j.murray@dur.ac.uk

Abstract

In recent years, the power of domain−independent planners which use hand coded domain−specific control knowledge has been demonstrated [AIP00]. This approach, though fruitful in terms of planner performance, has several issues asociated with it. Firstly, control rules need to be hand coded for each domain. This affords no scope for reuse, and the control rules formulated rely on the control rule writer's ability to recognise and exploit structure in the domain. Secondly, in comparisons between systems, it is often unclear to what extent off−line control rule construction governs the planners overall performance. This paper presents a new approach in automatically instantiating domain specific control rules from templates, with the use of existing domain analysis techniques. Using Generic Type information concerning the domain, template rules are instantiated from a library of *Generic Control Rules*. It is hoped that these instantiated rules may offer some of the benefits of hand−coded domain specific knowledge, but without many of the drawbacks.

1 Domain Analysis

Domain analysis is used in planning ultimately as a means of reducing the search that is essentially the burden of the planner. The concept behind domain analysis is that by analysing the domain in some way, it is often possible to recognise unfruitful paths in the search space (with respect to a problem instance) without having to actually traverse it. Typically, domain analysis is in the form of pre−processing before the planner is invoked, and the information unearthed by the domain analysis tool is passed to the planner along with the domain and problem specifications.

The field of domain analysis has developed to a point at which it is able to provide planners with important and valuable information relating to the structure of certain domains. Existing domain analysis techniques are capable of discovering a variety of different forms of information, e.g. State Invariants (conditions which are necessarily true of every state reachable from the initial state), Types (classification of domain objects according to the states they can be in and the operators that they can be changed by), etc.; State Invariants are used to prune any branch of the search space that violates them (as these are not reachable states) while Type analysis can can be used to disregard branches that violate type restrictions (e.g. those states reached by applying an operator to badly typed arguments).

Automatic domain analysis has been seen to recognise structure within a domain that humans have overlooked. A notable example of this is the Paintwall domain [FL00], in which the walls can be seen as traversing a map of locations represented by different coloured paints. In the 1998 AI Planning Systems competition, the Mystery domain (an encoding of the Logistics domain, with alternative object and predicate names [MD00]) was ’found out’ by a domain analysis tool, which allowed its planning system to invoke appropriate transportation heuristics.

2 Generic Types

Notably, the work of Fox and Long (which covers Type Analysis, Symmetry Analysis and Generic Type Analysis) in the field of domain analysis has provided the international planning community with the notion of Generic Types. These are higher−order types, populated by types identified in a given domain, and are identified by an extension of the type analysis machinery [FL98], [FL00]. Type classification is carried out by looking at the planning domain as a set of finite state machines (FSMs) whose states are representative of the individual argument positions in each predicate, and whose transition rules are governed by the operators described for the domain. Those domain objects that can traverse the same FSMs are grouped together into a type. Generic Type analysis takes this further and recognises templates or topologies of FSMs associated with a particular type and provide us with information relating to the shared behaviours and properties of types in totally distinct domains.

Each particular Generic Type has features which play specific roles in its behaviour. These features can be described at either the Generic Type level or at the level of the instance of the Generic Type. For example, one established Generic Type is the Safe Portable Object Type (SPOT). The distinguishing feature of a type identified as a SPOT is that its members can be transported between locations (according to some map) but never have any other role in a plan (commonly they have a specified goal location). Safe Portable Objects (SPOs) are distinguished from other Portable Objects precisely because it is safe to transport them, without affecting other processes in the domain. A SPO (like Portable Objects in general) changes location by being transported by a Carrier, which can pickup and deposit the object at any of the locations on its map.

It is possible to talk about the *locatedness* predicate of a SPO, meaning the predicate (relationship) which relates the SPO to the location at which it is situated. For instances of a SPO, such as packages in the Logistics domain, we can say that the locatedness predicate (or at−relation) is the *at* predicate. In the case of the Gripper domain, where the balls are identified as SPOs, *in−room* is the appropriate predicate. The other features that all members of a SPOT posess are a *contained_in* predicate (to show they are being carried by a Carrier) and the ability to have *load* and *unload* operations performed on them (to be loaded onto or deposited from a Carrier). Through these relations, is possible to talk about either the location or the carrier object to which the SPO is related in any given state. However, it is important to remember that a group of objects is only identified a SPOT if it meets the requirements; that those objects form a Type and that members of that type have no other role in the plan than to be transported between locations.

3 The Language of Control Rules

Control rules can be supplied with the domain description and problem instance and generally give planners heuristics for manipulating objects in the domain more efficiently. They can be explicitly goal−directed (of the form "if *P* is in the goal then do *Q")*, but need not be. They can simply offer efficient ways to achieve some desired state from some known state.

Historically, planners have generally had their own languages for the purposes of inputting domain specific knowledge. This is evident in the recent landmarks of TLPlan [BK00] and TALPlanner [DK99], both of which had knowledge expressed in temporal logics. The following is an example of a control rule for the Logistics domain, and captures the fact that any of the packages in the Logistics domain, once at their goal location, should remain at that location:

(1)

$\Box$ $\forall$ X:{package1,package2}. $\forall$ Y: {bos−po, pgh−po, bos−airport, pgh−airport} . at(X,Y) /\ Goal(at(X,Y)) => O(at(X,Y))

A proposal has been made to provide a standard environment for the exchange of domain knowledge in the form of DKEL [SH00]. However, these languages do not readily provide support for the proposed Generic Control Rules (i.e. control rules expressed in terms of Generic Types). A temporal logic will be proposed for expressing these control rules.

The logic proposed will incorporate the modal temporal operators O (Next), in order to be able to refer to progressions of states, $\Box$ (Always) to refer to all states and Goal to refer to the goal state. The language will be used at three distinct levels: at the highest level to express Generic Control Rules as in the library, at an intermediate level to enable the output of domain specific logical formulae (resembling existing control rule logics) and at a low level to provide object−specific queries for use with planners not necessarily capable of using general control knowledge.

4 Generic Types and Control Rules

Where several different types have the same structure of behaviour (i.e. equivalent states with equivalent state changing operators), as in Generic Types, it allows us to abstract any heuristics relevant to any particular instance to the abstracted level. This also means that heuristics can be formulated in terms of the abstraction, and then interpreted to apply to the all of the instantiations of those abstractions. It follows then that performance−enhancing heuristics can be expressed in terms of the Generic Type, as opposed to in terms of instances of that Generic Type. This abstraction allows the information to be applied in any instance of the Generic Type that is identified.

This aim has, to some extent, been achieved and was originally done in an integrated way, in the domain−analysis/planner partnership of TIM/STAN [FL98], in which hard−coded heuristics were triggered by Generic Type identification. There was no temporal aspect in the hard−coded control information, though, and no formalism was presented for the heuristics expressed. Also, the integrated approach does not allow the user easy access to the heuristics themselves, which can be embedded in the implementation. This makes it awkward to add to the heuristics, or indeed to add to the Generic Types that once identified, trigger those heuristics.

The integrated approach also means that the information discovered by the domain analysis tool tends to be in a format tailored for its partner system (the planner). The introduction of an API to TIM has begun providing access to parts of the domain anaylsis, of which Generic Type analysis is only a part. The work described in this paper aims to instantiate domain specific control knowledge (from reusable hand coded templates) using Generic Type information generated by existing domain analysis techniques.

5 Generic Control Rules

As outlined earlier, it is proposed that control rules be written in terms of Generic Types. The following is an example of a Generic Control Rule:

(2)

$\Box$ $\forall$T: SPOT. $\forall$X: T. (location_of$_T$ X) == (Goal (location_of$_T$ X)) => (O (location_of$_T$ X)) == (location_of$_T$ X)

This control rule expresses the heuristic that all members of a Safe Portable Object Type should remain at their goal location upon getting there. The antecedent ensures that the location of the object *X* in the current state is the same as it is in the goal state (the state qualification, to check that the rule is applicable). The consequent expresses that in the next state, the location of *X* is the same as in the current state (i.e the direction for *X* to stay where it is). The instantiation of the Generic Control Rule into a domain specific control rule would involve the specialisation of (2).

The *location_of* function is specific to the type *T*. This is important as every domain type *T* may have its own at−relation, argument positions within that relation and types associated with those argument positions. *(location_of$_T$ X)* should be able to return the argument that *X* is located at, expressed as *X* and its location in the at−relation specific to *T*. So the first step in instantiating (2) is to identify the appropriate at−relation (specific to *T*), with the types and positions of its arguments (this process becomes more complicated for relations with arities greater than two). This enables us to create a proposition from *(location_of$_T$ X)*, and in the Logistics domain would look like:

$$\text{at}(x{:}T_1,\ y{:}T_2) \tag{3}$$

where T_1 is the domain type identified as a SPOT, T_2 is the domain type identified as the SPOT's locations and *at* is the locatedness predicate. We can then re−write the rule in (2) using the substitution of (3) in place of *(location_of$_T$ X)*, as:

$$\Box\ \forall X{:}\ T_1.\ \ \forall Y{:}\ T_2.\ (\text{at}(X,Y) \wedge \text{Goal}(\ \text{at}(X,Y)) \Rightarrow (\text{O at}(X,Y)) \wedge \text{at}(X,Y)) \tag{4}$$

Notice that when the equality is evaluated, it is done with respect to a state (expressions without a temporal operator have an implicit 'Now' state argument,
indicating the current state). Now that we are dealing with propositions (see (3)) not values, we want to express that those propositions are true (in whatever their specified state), as opposed to equal (which wouldn't make sense). As a result, the equalities expressed in (2) become conjunctions as in (4), where the values that were bound by the equality are expressed as instances of the same variable. The introduction of the second argument in the at−relation (*Y*) requires the second quantification, but we have available to us the type information to do this (we want quantification over a type). Finally, the lemma

$$A \wedge B \Rightarrow A \wedge C \equiv A \wedge B \Rightarrow C \tag{5}$$

can be used to reduce (4) further to:

$$\Box\ \forall\ X{:}\ T_1.\ \ \forall\ Y{:}\ T_2.\ (\text{at}(X,Y) \wedge \text{Goal}(\ \text{at}(X,Y)) \Rightarrow (\text{O at}(X,Y))) \tag{6}$$

Individual object queries could be posted at this lower level, for planners not capable of using temporal control knowledge. This structure could be queried with domain objects *A: T_1* and *B: T_2*. If the antecedent was satisfied then the planner would know that the consequent should hold (in the plan, if the heuristic was adhered to).

Of course not all the domain specific control rules can be expressed in this manner. Rules which don't exploit types recognised as Generic Types cannot be expressed, possibly because there is no Generic Type currently identified for that particular structure or possibly because that particular rule exploits something other than Generic behaviour. However, as time progresses, it is expected that more and more Generic Types will be identified, encompassing increasing numbers of behaviours. This will allow increasing numbers of control rules to be identified for domains exhibiting the appropriate structure. Caution must be excercised, though, as the overhead cost will increase with the amount of work done by the domains analysis tool. This will have to be weighed against the benefits that the tool affords, but it is is feasible that the cost of rule instantiation for large numbers of rules would outweigh the time benefits in plan construction. Once there are large numbers of control rules being instantiated, there may also be issues regarding the precedence or priority of control rules. These points will need to be looked into as the work progresses.

6 Generic Control Rule Logics

The logic with which Generic Control Rules are expressed largely inherits its syntax and semantics from standard modal logics. However, there remain some points to note. First let there be a distinction made between the logic used to express the rules in their generic form, which we shall refer to as GCRLogic, and the logic used to express the rules in their instantiated form, which we shall refer to as DCRLogic.

DCRLogic is very much a standard modal logic, whose terms are propositions and whose modal operators are Next, Always and Goal. Universal quantification is allowed over domain types (sets of domain objects). The truth value of a proposition is determined by whether that proposition holds in the current (or otherwise specified) state.

GCRLogic, on the other hand, has a few more subtleties associated with it. Firstly, the ’of type’ operator (’:’) is overloaded. It can be used on two levels; to denote an object variable’s membership in a type and also to denote a type variable’s membership in a type of types. Universal quantification over these ’meta−types’ allows us to generalise over types that share behaviour, i.e. generalise over types that can be identified as being of a common Generic Type. In this sense, a Generic Type can be thought of as a higher−order type, populated by all domain types that can be shown to exhibit the appropriate behaviour.

In GCRLogic, the terms are the truth values of equalities between objects. The objects can be referred to through variables (introduced through quantification) or by function application to a variable. The application of a function to an object variable is not evaluated at the level of the GCRLogic. Instead, it is used to refer to an object that is related, through the relation expressed by the function, to the variable to which the function is applied. For example, in (2), the term

$$(\text{location\_of}_T\ X) \qquad (7)$$

refers to the object related to X through X’s location relation. We also want to be able to refer to objects that have a particular relationship with the variable in special states, i.e. in the goal state, the next state or in every state (remembering that a term unadorned with a modal operator has an implicit ’Now’ state argument). However, as in standard modal logics, in GCRLogic modal operators can only be applied to sentences, not objects. So in order to refer to, for instance, X’s location (as in (7)) in the Goal state, formally we must state

$$\text{Goal } ((\text{location\_of}_T\ X) == N) \qquad (8)$$

However, it must be noted that a short hand representation is in use for ease of reading GCRLogic statements. We allow the use of

$$\text{Goal } (\text{location\_of}_T\ X) == \text{O } (\text{location\_of}_T\ X) \qquad (9)$$

in order to represent the formally correct

$$(\text{Goal } (\text{location\_of}_T\ X) == N) \wedge (\text{O } (\text{location\_of}_T\ (X) == N)) \qquad (10)$$

NB this short form is applicable for all modal operators.

7 Control Rule Library

Generic Control Rules will be collected into a library, which the domain analysis tool will have access to. As a result, when domain analysis discovers Generic Types in a given domain, it can retrieve the relevant rules from the library and instantiate them with the specifics of the domain (see Figure 1 for proposed architecture). It is envisaged that this library will grow over time (as new Generic Types are described or additional rules are added for existing Generic Types), allowing the domain analysis tool to instantiate more control rules and ultimately improve the performance of the planner making use of the domain analysis. It is conceivable that

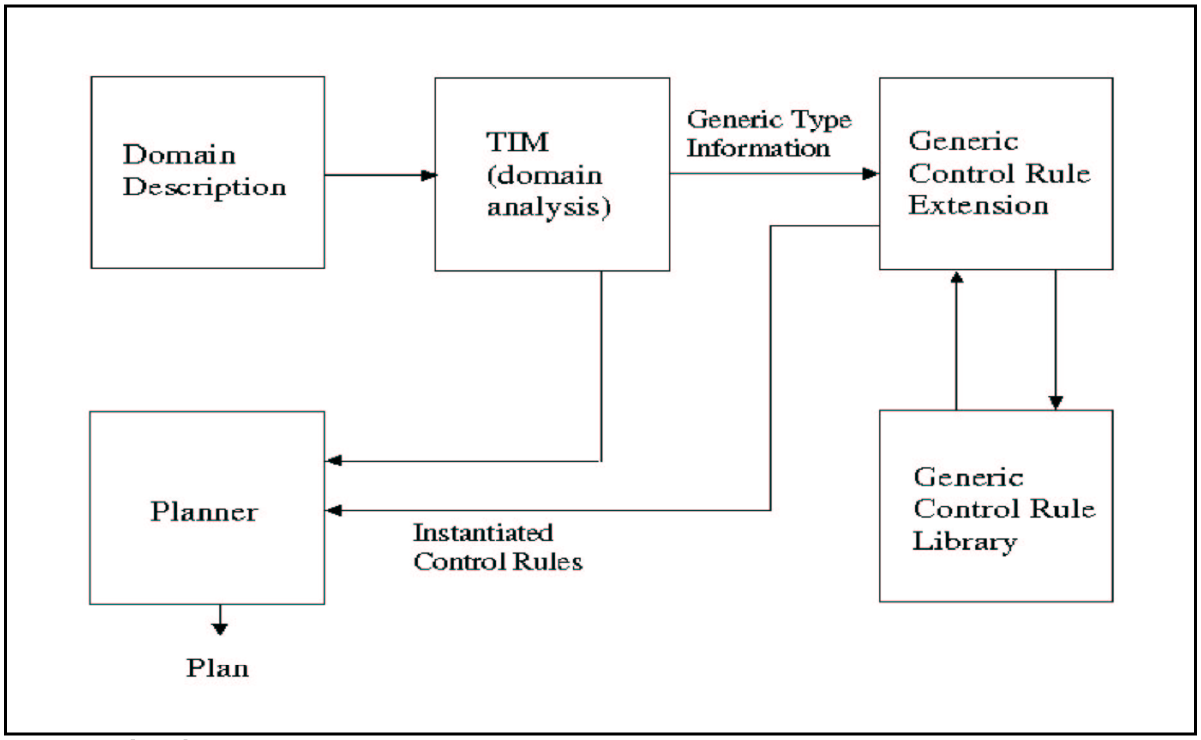
Domain Description
TIM (domain analysis)
Generic Type Information
Generic Control Rule Extension
Planner
Instantiated Control Rules
Generic Control Rule Library
Plan

Figure 1 Proposed Architecture

future work could involve the automatic generation of Generic Control Rules. However, static domain analysis techniques are not powerful enough to achieve this; this may perhaps be attainable through an adaptive tool (one able to learn through many examples), but it is not proposed here.

The library of control rules will not remove the need for control rules to be written, rather it will allow control rule writers to write reusable control rules. However, a point worth noting is that this does involve some standardisation. Writing the control rules in terms of Generic Types forces the individual Generic Types to be standardised. The parts of the Generic Types must have standard names in order for the Generic Control Rules to be meaningful. For example, the Generic Type *mobile* has a *locatedness* predicate, representative of the fact that mobiles are always located somewhere on a map of locations. This predicate, and its contents (i.e. the name of the location involved) must be accessable in a standard way. A possible solution is that each Generic Type is supplied with a prototype, giving the names of all its fields with some description for the benefit of those using the Generic Type. The logic for expressing the control rules must also adhere to a common standard, in order for the library to be portable.

8 Conclusion

Planners which take domain specific control knowledge in addition to the domain and problem descriptions have been shown to perform well over recent years. Unfortunately though, the control rules have only yet been expressed as domain specific, with no possibilty of reuse. This means that every domain (and to some extent, every different encoding of a domain) needs a new set of rules to be formulated, and then entered, by a human control rule writer. The control rules written rely on the writer's ability to recognise and exploit structure in the domain and this is not always an easy task as the names of objects and relationships in the domain can be misleading (as in the Mystery domain [MD00]).

Work has been started on imlementing the system described. At the time of writing, we are able to instantiate hard−coded Generic Control Rules for Safe Portable Objects, of the form described in (2), and be given the instantiated form as in (4). As of yet, we have not tackled the more general case, where multiple rules will be instantiated for any Generic Types identified in the domain, due to the fact that the library has not been implemented yet. We have not yet given the output (instantiated rules) as input to a planning system, but this is the aim in the near future. We expect the effect of supplying automatically instantiated rules to a planner will be similar to that of supplying hand coded ones; we will look for time improvements in plan construction. However, we expect that the library of Generic Rules will have to grow to include a significant number of control rules over a wide range of Generic Types in order to get anywhere near comparable performance to that where a team of domain engineers has coded domain specific control knowledge.

The work described, although still in progress, potentially offers many benefits to the planning community. The proposed will present a readily portable means of reusing abstracted control knowledge, that has till now required many man−hours to generate on a domain−by−domain basis. This abstracted control knowledge will be in a form that is easy to add to and edit, and will be available for use by the planner in the time it takes the domain analysis and template instantiation to occur. A possible extension to the work is the automatic abstraction of domain−specific rules into Generic Control Rules, and possibly even automatic generation of Generic Control Rules themselves, though this is a long way off yet.

References

[AIP00] Results of AIPS−2000 planning competition, http://www.cs.toronto.edu/aips2000/SelfContainedAIPS−2000.ps

[BK00] F. Bacchus and F. Kabanza, Using Temporal Logics to Express Search Control Knowledge for Planning, Artificial Intelligence, 116(1,2):123−−191, 2000.

[DK99] P.Doherty & J. Kvarnström, TALplanner: An Empirical Investigation of a Temporal Logic−based Forward Chaining Planner, Proceedings of the Sixth International Workshop on Temporal Representation and Reasoning, 1999.

[FL98] M.Fox and D.Long, The Automatic Inference of State Invariants in TIM, Journal of AI Research (JAIR), 9:367−−421, 1998.

[FL00] D Long, M Fox, Automatic Synthesis and Use of Generic Types in Planning, Proceedings of AIPS 2000, 196−−205, 2000.

[MD00] D McDermott, The 1998 Planning Systems Competition, Artificial Intelligence, 21(2):35−55, 2000.

[SH00] U Scholz and P Haslum. Enable Your Planner! Decoupling Domain Analysis and Planning. Tech report AIDA−00−05, Technische Universität Darmstadt, 2000.