Abstract control rules and their use in domain independent planning

Luke Caleb James Murray

Abstract control rules and their use in domain independent planning

by

Luke Caleb James Murray

Submitted in conformity with the requirements
for the degree of PhD
Department of Computer Science
University of Durham

for Jim, Liz, Dano and Sam. I am you.

"A sense of symbolism sees you through, you see,
Abstraction's no distraction,
Similarities will set you free."

Abstract

Abstract control rules and their use in domain independent planning

Luke Caleb James Murray

Control rule based planners that take advantage of hand coded domain specific control knowledge currently out-perform fully automatic planners. Generic types have been presented as abstractions of the behaviour of types identified in planning domains and we extend this abstraction to sets of interacting generic types, or *generic clusters.* A state based modal temporal logic is presented for expressing properties of sequences of states, the terms of which are features of generic clusters. The logic enables control strategies to be written in terms of generic clusters. Automatic generic cluster identification provides a way of automatically specialising expressions with the details of any domain level instances, yielding domain specific control rules. We show how control rules employed by current control rule based systems can be expressed in the framework presented and also demonstrate generic control rules being used to improve plan quality in a state of the art, fully automatic planning system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Planning

The general problem of classical planning is to construct some ordered sequence of operators to transform an initial state into a desired goal state. This involves searching for a solution in a problem space defined by the initial state and the mechanics of the domain in which the problem is posed. The problem space maybe in one of various forms, typically either a state space in which the nodes in the space are states of the domain and the transitions are operator applications or a partial plan space in which the nodes are partial plans and the transitions are additional constraints on those partial plans.

The domain description describes the mechanics of the domain. It specifies the relationships that can hold between objects in the modelled world and the operators that change those relationships. Operators are defined in terms of preconditions and effects. Preconditions are the conditions that must hold in order that an operator may be applied legally in a state, while effects are the the way in which an application of the operator affects that state. The domain description sets out the way the particular domain being modelled works and it does so through the use of variables (implicitly universally quantified over any constants that may inhabit that world), so both the predicates that describe properties of objects and the operators that affect these may be instantiated with constants from any given problem instance. Domain descriptions can contain more than a pure description of the workings of the model, but this is discussed in Section 2.4.

The initial state S_0 gives the constants involved in a given problem as well as their initial configuration (in terms of properties and relations). The goal state S_g gives the desired final state into which the planner must, through the use of legal operators, transform the initial state to successfully complete its task. S_0 must be a complete state description but S_g may only be a partial state description. A plan is valid if every operator is applied in a state in which its preconditions are satisfied. A plan solves a planning problem if it is valid and the state reached through the application of the plan from S_0, S_n, contains the goal conditions.

$$S_g \subset S_n$$

(Beyond the realms of classical planning we might look for

$$S_n \models S_g$$

where S_g is a formula instead of a conjunction of literals. Discussion on this difference is contained further in.)

1.2 The problem of search

Brute force search is not an effective way to search for solutions in an enormous search space. In the field of planning, generating a solution to a problem instance generally involves looking for the solution in just such an enormous search space, be it of states or partial plans. While solutions can often be found quite easily in very small instances, the task of locating a solution becomes insurmountable (propositional planning is PSPACE-complete [6]) very rapidly as more complex instances are attempted (other formal analyses of the complexity of planning has been conducted [10, 11, 7], as has work on the complexity of specific planning domains [27]).

Many algorithms for search have been presented. They can be described as complete or incomplete and systematic or stochastic. Complete algorithms, as the name implies, have the ability to search the entire search space for the desired solution (unlike incomplete methods). Systematic approaches work methodically through their available spaces, while stochastic methods are more 'hit and miss'. That is not to say that incomplete or stochastic methods are not valuable search strategies. In fact incomplete strategies can make unmanageably large search problems much more accessible [29] (obviously there is a risk of excluding a region of the search space in which a solution lies) and stochastic methods have shown their ability to find solutions quickly [51].

The basis for most complete search procedures is either *breadth-first* search or *depth-first search*, which consider siblings before children and children before siblings respectively. Search can also be guided through the use of *heuristic functions* (an estimate of the 'worth' of a node), such as *best-first* search or *A** search [48]. Heuristic searches order the sequence in which nodes are considered and expanded, and guarantees about the admissibility of a search strategy can be made on the strength of the admissibility of the heuristic function used. For example, the use of an admissible (conservative, never over-estimating) heuristic in *A** search guarantees that the solution found will be optimal. *A** search values a node by the sum of the cost to reach that node and the heuristic estimate of distance from the goal state. The fact that the heuristic does not over-estimate the distance to the goal state forces the search to consider the optimal solution (a complete non-optimal solution will necessarily have a lower estimation of worth than a partial solution that coincides with the optimal solution, forcing the algorithm to continue its search).

The ultimate goal of all of the search strategies presented is to find solutions in large search spaces, though there are necessarily trade-offs between such factors as time and space requirements. Whether the search is for an *optimal* solution (according to some metric) or simply *any* solution can affect these decisions.

1.2.1 Facing the problem of search

Planning problems are most commonly posed as a domain description and a problem instance in that domain. However, many benchmark domains implicitly contain structure which, if it can be made available to the planning algorithm, can be used to traverse the search space in a more intelligent way. This structure can range from domain invariants (propositions that hold for every possible state in the domain) to patterns of behaviour which, once identified, can totally transform the way in which one decides to tackle the problem. But, as in much of AI research, what is obvious and easy to us is very difficult to recreate in machines (e.g. natural language engineering is subject to ongoing research, but speaking and understanding our native language is trivial).

So having noted that there is sometimes implicit structure in a domain, the problem becomes how to recognise, access and exploit it. Static pre-planning domain analysis can identify various forms of information relating to the domain, including hypothesise-and-test generated domain invariants [26], operator based domain invariants [47], inferred state constraints [24] and rigorous type inference of objects in the domain [21].

Alternatively, the domain can be studied by a human domain engineer and any structure or features of the behaviour of the domain can be hand-coded and supplied to the planner in addition to the declarative description of the domain's mechanics (in some cases the declarative description is explicitly modified to incorporated any control strategies). There are many planners whose performance depends on this manner of additional input (e.g. TLPlan [3], TALPlanner[16], SHOP [46]). These will be discussed in Section 2.3.

1.2.2 Making use of extra information

There are a number of ways to make use of knowledge of these structures or behaviours in the context of planning but in essence they are all used to restrict the search space being considered. Although static domain analysis techniques can be very powerful and extract deeply hidden behavioural structure, it has been noted that planning systems that take hand coded domain-specific control knowledge perform best in terms of scalability [39].

There are two main problems with hand-coded domain-specific control knowledge, though; it has to be hand-coded and it is domain specific.

Firstly, there is a great onus on the author of the control knowledge. He has not only to analyse the domain in depth and translate high level observations into some formal language specified for control information but also bear the responsibility that the performance of the planning system is essentially dependent on his input. Any mistakes or inconsistencies could result in the planner failing to perform at all in the given domain, rendering useless all the work involved in supplying the additional information. Over the past few years, at international planning competitions, it has been commonplace for teams running control rule based planners to spend all night constructing and tweaking the control rules for their planners when new domains are introduced [1].

Where planning system authors are given significant time to construct control information for a particular domain (as in the international planning competitions), several questions are raised: Do comparisons of planners' performances compare the planning technology, the ability of the control rule author to identify and encode suitable strategies or the ease in which the relevant control information can be represented in the particular language accepted by the planner? Do we need to draw a distinction between planner-independent control knowledge and planner dependent control knowledge, and can the latter be seen not as supplying the system with control knowledge but rather tailoring a domain-independent planner to become more domain-specific? Khambampati presents an excellent discussion of some of these questions [30].This thesis will not answer all these questions, but the language and methods described herein make a valuable tool with which to address them.

Secondly, the encodings of the control information are not reusable outside of that particular manifestation of the given domain. The very fact that the control information is domain-specific means that even if the author recognises similar patterns of behaviour in other, possibly very similar, domains then the control strategies must be re-encoded with respect to the new specifics. Not only is this time-consuming but it also allows more room for human error.

However, much of the control information used can be seen to be exploiting analogous features across different domains. Bacchus and Kabanza [5] noted

"We have found that there are many 'meta-level' strategies that are

[1]Private communication with Fahiem Bacchus

applicable across different domains under slightly different concrete realisations."

A final problem with hand-coded control information is that humans, unlike machines, are not infallibly methodical. By this I mean that they may overlook some structure or feature in the domain because it does not appear in some expected fashion. The naming of objects and operators is a classic example of this, as, for instance, with the Mystery domain [42] but other examples exist (such as the PaintWall domain [38]).

1.3 The integrated approach

The thesis we present is that useful control knowledge can be expressed at an abstract level and efficiently and automatically translated into domain specific control rules. The work details an integrated approach to generating domain-specific control knowledge that draws on both static domain analysis as well as human observations (to compose the abstract control rules). Abstract behaviour based structures will be presented to represent sets of types exhibiting generic behaviours which have been shown to occur across many benchmark domains. This allows the control information to be written at a higher level, that is at the level of an abstract structure.

Existing domain analysis tools can already identify the structures that exhibit this generic behaviour and provide detailed information on their domain specific instances [38, 40, 22]. By identifying *generic types* in a domain, we are identifying types of objects that behave in ways that we recognise and understand. The recognition of generic types can be seen as identifying classes of domains (those in which an instance of a specific generic type is identified) or classes of types (those types from the universe of domain object types that exhibit particular behaviour).

The control information that is expressed at the abstract level can then be specialised using the details of the realisation of that abstract structure in the domain under consideration. The machinery that identifies generic types also supplies the details of their realisations. Once fully instantiated, the control information will be indistinguishable from that hand-coded specifically for the domain. It will then used as any other control information is, in restricting the search for a solution plan, the exact process of which will depend on the planning algorithm. An overview of the proposed system architecture is shown in Figure 1.1.

Let us see how this abstraction could be drawn. Consider two transportation domains in which the vehicles have different type predicates (such as $truck(x)$ and $lorry$(x) to represent that x is the transportation vehicle). Currently, though the behaviour of the domains may be identical, any control strategies must be explicitly written for each domain. The strategy of never immediately returning to the previous location (of the vehicle) would need to be expressed both in terms of *truck* and *lorry*. By expressing this strategy at an abstract level based on the behaviour of the objects, both domain specific rules are covered (as in 'for every object, x, whose type behaves as a transportation vehicle type, never immediately return x to its previous location'). Although this is the simplest of examples, it displays the behaviour based abstraction that we refer to.

As more generic types are recognised, and rules written for those types, a library of generic control knowledge will be built up and provided with the machinery to instantiate them (namely the generic type analysis currently performed by TIM [38]). This would provide a plug-in module for use with any planners in the community capable of making use of control knowledge (assuming that knowledge can be expressed in a suitable form). The library could be used, for instance, in comparisons between planners that take control rules; up till now such comparisons really compare not just the planners themselves, but the product of the planners, the work invested in constructing hand coded control knowledge and, not least, the control-rule writer's own ability to identify appropriate search strategies.

The work done in producing the abstracted control rules is totally reusable (so long as the control information can be instantiated or translated into a form usable by any given planning system). Where and whenever the appropriate structures are recognised in a domain, the instantiation mechanism can provide control rules as though they were hand-coded especially for that domain. In this sense, the investment made into constructing the rules has a higher return than simply constructing domain specific rules for every domain. The rules not only cover behaviours that occur in many benchmark problems (removing the need to manually construct these control strategies) but also provide support for domains yet to be encountered (that contain types exhibiting the appropriate behaviour patterns). The automation of this process also eliminates the drawbacks of manual domain analysis/control rule formulation.

It could be argued that the construction of abstract rules is more taxing than their domain specific counterparts. It is the opinion of the author that where a direct mapping between the instance and the abstract structure is given, the construction of rules in terms of the abstract behaviour structures is no harder than in terms of their instances.

With more work in the community on generic types and abstracted control information, it is conceivable that planners that have to date been classed as hand-coded could be run as totally automatic with comparable performance. We could also use the rules as a basis to compare planners that rely on control information for their performance, avoiding debates on the significance of the manual formulation process of those rules.

However, not all of the control knowledge used in current planners is classifiable as exploiting particular generic behaviour; perhaps that generic behaviour has not been identified yet or maybe the behaviour is not a product of anything less than the unique structure of domain in its entirety. Even if this is the case, the construction of control information for the generic behaviour that is the whole of some complex domain would have value. Its value would be precisely having 'off-the-shelf' control mechanisms to use whenever the particular behaviour is recognised in a planning problem.

1.4 Statement of thesis

This work attempts to verify the the following thesis:

> Behavioural cluster based abstractions of useful control strategies can be automatically instantiated to provide domain specific con-

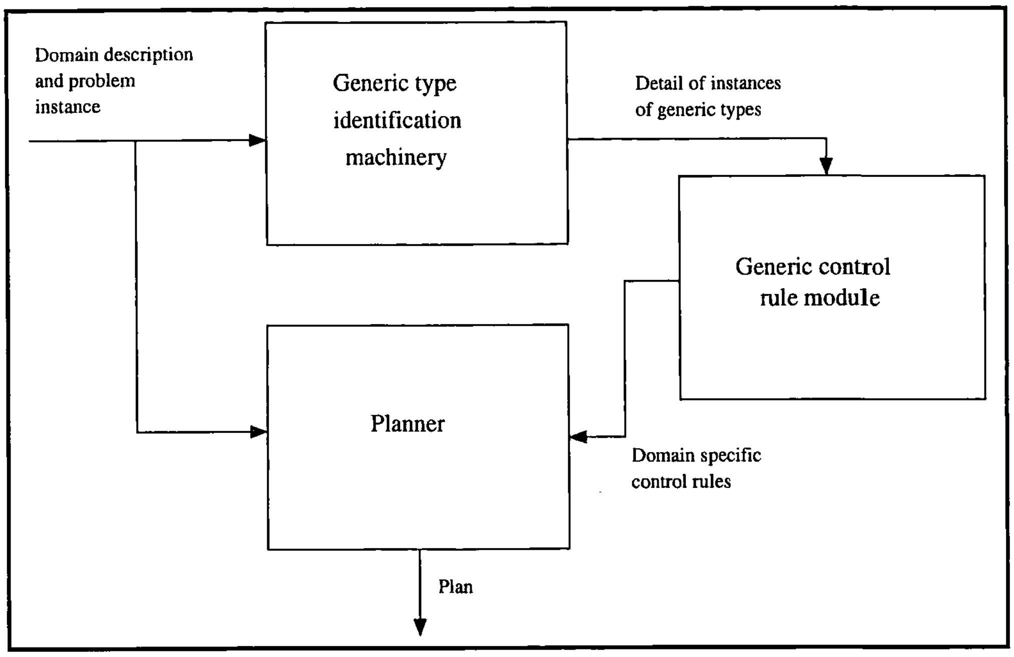

Figure 1.1: Overview of proposed system architecture

trol rules, which in turn can be used by planning systems to aid the generation of higher quality plans.

1.5 Thesis Map

Chapter 1 has provided an introduction to the main areas covered in this thesis. Chapter 2 provides a more detailed exploration of the literature concerning the appropriate aspects: control rule based domain independent planning, search control and domain analysis (with particular focus on type inference and generic type identification).

Chapter 3 presents the methods and structures employed to test the thesis in the form of a system that automatically generates domain specific control rules from abstract control rules. This comprises a system architecture and an in-depth presentation of each of the components. Firstly, generic types are explored and developed to present abstract behaviour based structures. The syntax and semantics of these structures is presented and discussed. Secondly, a linear temporal logic is presented whose terms are parts of the abstract behaviour based structures. This allows control information to be expressed at the abstract level by specifying properties of sequences of states. The syntax and semantics of the logic are presented. Thirdly, a domain level language is presented for the expression of domain specific control rules, again with its syntax and semantics. Fourthly, the instantiation process that specialises abstract control rules into their domain specific realisations is described. Finally, a proof is given that the semantics of the abstract logic are consistent with the combined processes of instantiation and evaluation with respect to a sequence of states. This proof is reminiscent of demonstrating the equivalence of expressions by normal order reduction, but involves transform operators so the resulting equivalences are based on the evaluation of expressions in different languages (the expressions all yield truth values on evaluation).

Chapter 4 discusses some of the subtleties implicit in the ideas presented in Chapter 3, including some decisions that needed to be made in the specification of the languages and structures.

Chapter 5 provides proof of concept of the thesis. These results take several forms. The integration of the generic control rule module with an existing planning system is described. Results are presented showing the effect of the additional automatically generated control information. Results are also presented demonstrating the time penalty incurred for the process of instantiating abstract control rules (without subsequently using them). The control strategies employed by a highly acclaimed control rule base planner are discussed and abstracted forms of that control knowledge are presented. The wider applicability of the approach presented is also demonstrated by considering another behaviour structure with a well acknowledged control strategy, which, in light of the work presented, can be expressed in an abstracted form.

Chapter 6 draws conclusions form the results of Chapter 5. The contributions that have been made to the field are stated, along with suggested directions for further research. Finally, Chapter 7 provides a summary of the thesis.

Chapter 2

Background

2.1 Introduction to background material

The work presented in this thesis falls under the umbrella term of domain independent planning and draws on two points of focus for the international community; domain analysis and domain specific control information. In addition to these areas, the work is related to generic behaviours across aspects of different domains. In this section, the background and state of the art in these areas is discussed and explored.

2.2 Domain independent planning

The task of domain independent planning is to produce planners that perform on a range of different tasks, rather than tailoring the system to solve only one type of problem. Although domain dependent planners are acknowledged and cited, this thesis concerns, in the main, domain independent planning.

2.3 Control rule based planners

Domain independent planning strives to develop planning systems that perform well across a range of domains. The trouble is that, by the very nature of their generality, domain independent algorithms are not as well suited to individual domains as those developed specifically for one application. An approach that has tried to overcome this problem is to supply additional information alongside the bare bones of the domain description and problem instance, in order to give the planning algorithm more knowledge about searching for a solution in a particular search space. This idea is not new; domain knowledge of some form was used by several early planning systems [44, 55, 13, 57].

2.3.1 Integrated control

The early attempts at incorporating domain specific knowledge were very much tied to the planning algorithm to which it was supplied. This meant that the domain engineer had not only the responsibility of correctly identifying appropriate search strategies, but also needed extensive knowledge of the planning algorithm and the ways it would make use of the additional information. This afforded no scope for reuse of search strategies across the community, let alone reuse of search strategies across domains. To some extent, even alternative domain encodings required the effort in constructing and integrating search strategies to be reinvested.

One of the early examples of a planning system making use of control knowledge is the PRODIGY system [44], which used control rules of various forms to control several different aspects of the planning process (there was also a module capable of learning control rules, discussed further in). The control rules used were very much dependent on the planning architecture and because of the way that PRODIGY works, four types of control rule were naturally identified. During plan generation in PRODIGY, there are four easily recognisable choice points. Firstly, there is the choice of which node in the search tree to expand. Once that node has been picked, a goal at that level must be chosen to be achieved (goals are often preconditions of operators to be applied later in the plan). Next, there is the choice of which operator to select to achieve that goal and finally there is the decision about variable bindings within an operator instance. At each of these points PRODIGY could use control information to guide its selection, though it could resort to blind search if none was supplied.

The control rules used by PRODIGY are all of the form of "if-then" statements, i.e. a left hand side for matching a particular state and a right hand side which offered one of three types of advice. *Selection*, *rejection* and *preference* rules were identified, each with a slightly different role within the control hierarchy employed at each decision point. First, any selection rules whose left hand side matched the situation were used to select a subset of nodes/operators/goals/variable bindings. Next, any applicable rejection rules were used to restrict that set. Finally, any applicable preference rules gave information on an ordering with which to explore the alternatives. The syntax for each of the three types of control rule is the same.

One of the features that made PRODIGY so appealing for research purposes was its modularity. The architecture was such that it was very easy to add on new functionality, such as the EBL module [43] (Explanation-Based Learning, the module previously mentioned, designed to learn control rules). This module learns control rules through the use of *target concepts* (some predicate over a universe of instances that characterizes some subset of those instances) and training examples. The training examples are instances of the target concept and the *explanation* is the proof demonstrating that the example satisfies the target concept. The control rules generated are implications whose antecedent is the weakest precondition of a proof and whose consequent expresses some preference based on the proof's conclusion. This approach suffers from the drawback that you must provide a high level target concept for the system to attempt to learn.

2.3.2 Modularised control

An important alternative to integrated control strategies was the seperation of the control knowledge from the planning algorithm. Bacchus and Kabanza were the first to present an independent formal language for representing domain specific knowledge, in the form of a state-based linear modal temporal logic [3]. They used this in conjunction with a forward chaining planner in the TLPlan system, using the control knowledge expressed in the logic to prune candidate branches in the search space that violated the control conditions.

This approach was very successful when compared with contemporary systems [42], and for a while remained unchallenged as the system to beat.

This was a successful attempt to introduce the idea of a language for expressing control knowledge with no reference to the the planning algorithm it was augmenting. The abstraction from 'integrated' to 'independent' control knowledge afforded its users several benefits. The control rules for a given domain no longer had to be written by someone with an intimate knowledge of the planning algorithm; the control rules that were written described only behaviours of the domain itself. This provided both implementational and conceptual modularisation. The control knowledge itself was specified in a first-order linear temporal logic, which allowed reasoning about statements in the language. The temporal modalities that were added to their first order logic were: $\bigcirc$ (next), $\Box$ (always), $\Diamond$ (eventually) and $\cup$ (until). Bounded existential and universal quantification were employed in order to avoid infinite conjunctions and disjunctions when evaluating quantified formulae (as would be the case if unbounded quantification was used in an infinite domain). Bounded quantification avoids this problem with the assumption that there will only ever be a finite number of objects in the domain that satisfy the bounding condition.

Importantly, statements written in the logic (control rules) could be subject to a 'progression algorithm' which allowed the semantics to be preserved over a progression of states. The progression algorithm provided an interpretation of the temporal modal operators used in the logic, and statements could refer to, through these modalities, particular states in a progression of states (via successive applications of the $\bigcirc$ modality).

Aside from temporal modalities, Bacchus and Kabanza added to their linear temporal logic a GOAL modality, to enable them to reference properties of the goal state. In empirical tests, they found that making reference to the goals of the problem under consideration was

> "essential in writing effective control strategies."[5]

For example, in a simple transportation domain in which a vehicle has a goal location somewhere on a map of sites, we might use a control strategy that states "do not move into a location from which there are no exits." Except in the case where the goal location of the vehicle is a dead-end, this control strategy will always be advantageous. However, the rule does not currently have the power to recognise situations when the goal location *is* a dead end and without admitting a goal modality this eventuality could never be accounted for. Allowing control rules access to problem specific goals (through the use of the GOAL modality) enables the control strategy to become "do not move into a location from which there are no exits unless that location is the goal location of the vehicle."

Originally in [3] the semantics of the goal modality were specified in terms of entailment. Given a goal expressed as a first order formula ϕ and the set of domain state constraints ψ, $\text{GOAL}(f)$ is true iff $\phi \wedge \psi \models f$. Bacchus and Kabanza note, however, that testing $\text{GOAL}(f)$ is intractable in the general case and in their original implementation of TLPlan they restrict a problem's goal to a set of positive literals. They then use the goal set under a closed world assumption. This makes checking for the goal condition easy as the entailment is reduced to set membership, i.e. $\phi \models f$ iff $f \in \phi$ where f is a positive literal. Their original implementation could not handle state constraints over the goal world. These restrictions are propagated through later work, and though no mention is made of the closed world assumption in [5], the same restrictions apply (i.e. $\text{GOAL}(f)$ is true iff $f \in \phi$ where ϕ is a set of literals given as the goal of the problem). These subtleties of the logic are particularly relevant to certain aspects of the work presented in this thesis (c.f. Section 3.10.1).

Doherty and Kvarnström, in their TALPlanner [16] system, provided a different approach to the use of temporal logic control rules. Again, the implementation was a forward-chaining planner which used domain specific control rules written in a formal logic to help it search more efficiently through its search space. The first apparent difference is the logic. Whereas TLPlan used a linear modal tense logic for reasoning about states of the world, TALPlanner used a linear temporal logic with explicit time for reasoning about actions and changes in the world. The basis for TAL (Time and Action Logic), the logic used by TALPlanner, was earlier work by Sandewall. A description of this work (with reference to its extension into TAL) can be found in [49].

Two planning algorithms were presented, one which progresses temporal statements (as in TLPlan) and the other which translated any statements containing temporal modal operators into equivalent statements without those modalities. Both versions of the planner worked with narratives, a representation of a plan which uses explicit time, and plans were output in the form of narratives. The work is successful in that there are obvious performance enhancements (the TALplanner system performed extremely well in both the AIPS-2000 and 2002 competitions, and is still the planner to beat in certain domains) and richer expressivity in the language (through the representation of durative actions with internal state). A full explanation and specification of the Temporal Action Logic can be found in [15].

Further work with TALPlanner has looked at compiling control information into operator preconditions [37]. Precondition control has been explored before [2], but only as a manual process. This carried with it all the drawbacks of manual control rule entry. The recent work with TALPlanner has investigated automatic precondition control, with marked success. Gabaldon [23] has also explored automatic precondition compilation using a technique that is proved to be equivalent to the progression algorithm employed by TLPlan in terms of the effect of pruning the search space.

Precondition control has an advantage over using control information to additionally guide search. By qualifying operators' applicability through the use of precondition control, the search space branches far less rapidly than with the unrestricted preconditions. This means that the planner is searching a greatly reduced search space, one which necessarily obeys the control information encoded in the operator preconditions. However, unlike standalone control information, work has to be done up front to compile the control strategies into the operator preconditions.

Kautz and Selman offered modularised control in the form of their four classes of heuristic (discussed in 2.6) as a

> "way to encode domain specific knowledge in a purely declarative, algorithm independent manner."[32]

However, this was in the realm of planning as satisfiability, and as such was not readily usable by those in the community taking different approaches. Gerevini and Schubert [26], on the other hand, constructed constraints in the form of STRIPS-style literals connected by logical operators. They also went on to compile these into SAT clauses (and in fact used Kautz and Selman's SATPLAN to demonstrate their results) but the original constraints generated were of a more widely applicable format. Gerevini and Schubert's methods are discussed in 2.6.

2.4 Domain Purity

Domain descriptions can be specified in a number of fashions, independently of the different languages that exist for different planning systems (e.g. PDDL, TAL). Domain descriptions that give only the declarative functionality of a domain are what we call *pure*. Not all planning systems have use for this kind of pure domain description and instead need the functionality to be integrated with the search control to solve problems in the domain. For example, SHOP [46] (and its successor, SHOP2 [45]) requires search control strategies to be encoded into its operators, methods and axioms in order to achieve the performance demonstrated in [1] (and [39]). We have also noted that although TLPlan will accept and use temporal control knowledge over and above a declarative domain encoding, in practice the control knowledge was sometimes encoded in the form of auxiliary predicates (that Bacchus and Kabanza called *defined* and *declared* predicates [3]), as in the TLPlan encoding of the *ZenoTravel* domain [39] (c.f Appendix A). This is not the independent, stand-alone temporal control information that they set out to define and use. The simplest and purest domain descriptions give only the declarative mechanics of the domain. The work described in this thesis supports the separation of domain functionality and search control strategies.

Including control knowledge in the domain description has a similar flavour to the early approaches of integrating search control with the planning algorithm, though the control strategies can still be domain specific (being tied to the domain description). This approach does have the advantage of restricting the problem space (the number of applicable actions in a given state is restricted according to the control strategies encoded in the operator preconditions). Precondition control

> "has proven to drastically reduce the number of states generated" [37]

but removes the modularisation of control knowledge. Control knowledge can be succinctly and declaratively specified *as* control knowledge, but once it is embedded in operator preconditions it becomes inextricably entwined with the domain's functionality. Search control knowledge is *not* part of a domain's functionality. Automatic generation of precondition control from stand-alone control strategies has been demonstrated [37] and can not, as an automatic process, be affected by human error no matter how complex the operators or strategies. This approach has the advantages of both modularisation of the control information and a search space with reduced complexity.

2.5 Control of search

There are many approaches to the process of guiding search through control information. Not only are there many different ways to express control strategies, there are various ways in which that information is used.

Early informal attempts at managing the control of search included implementation and problem specific methods such as operator ordering. By ordering action definitions in the domain description, the order in which actions are selected at solution generation time can be affected. This is a very basic way to assign sequential preference to actions, but requires detailed knowledge of the way the planning algorithm selects actions. It is a naive strategy as one ordering is imposed for all problem instances and for all action selection points in any given instance (it may also affect search negatively).

Considering actions based on their *primary effects* was also seen to improve solution generation. A primary effect of an action is not a well defined concept in the literature but tries to describe the intended effects of an action as opposed to any incidental consequences. For instance, the primary effect of boiling a kettle would be said to be heating the water to 100° C, but its secondary effects include electricity consumption and steam production. This is obviously an ill-defined term as the motivation for boiling a kettle could well be steam production (for example, to open an envelope without tearing the paper). Planners including PRODIGY used the notion of primary effects informally though later work tried to formalise and justify the concept [20].

Kautz and Selman offer up a taxonomy of what they call heuristics derived from their manual domain analysis (see Section 2.6). These are essentially presented as forms of control knowledge playing different roles in solution generation.

An excellent discussion of some of the issues involved in using control knowledge for domain independent planners (in the context of the international planning competitions) is given by Kambhampati [30]. Important questions are raised to do with the role of control knowledge in the comparisons of planning systems dependent on control strategies for their performance, as well as some observations regarding the acquisition of that control knowledge.

2.5.1 Explicit search control

We use the term *explicit search control* to describe stand-alone strategies, expressions or suggestions with the purpose of controlling the way a search space is explored. As has been stated previously, explicit search control has two main origins, namely manual and automatic domain analysis.

Search control can be described action-based or state-based. Action-based control knowledge reasons about the actions that should or should not be considered under given conditions, whereas state-based approaches reason solely about the states involved. Both types of search control declaration have been explored in the literature, and examples of particular relevance and interest include TALPlanner [16] and TLPlan [3] which use action-based and state-based logics respectively in forward-chaining planning algorithms. Both methods can describe similar strategies, though state-based examples are used in this thesis in keeping with the state-based logic presented further in.

2.5.2 Hard and Soft Control Information

Hard control information is definite and absolute. It describes courses of action that must necessarily be followed. Soft control information on the other hand is more suggestive than absolute. It is possible for a planner using soft control information to disregard that advice in situations where alternatives are considered more advantageous.

A particular type of soft control rule is what Delgrande, Schaub and Tompits call a *preference* [14]. This is not an optimising control rule for constructing better plans (as in the preference rules described for PRODIGY), rather they are additional opportunistic goals. Where a goal G can be achieved by plans P_1 and P_2, we may have a preference for how that goal is achieved. Though P_1 may satisfy the *necessary* goal conditions, we will select P_2, given the choice, if it adheres to our domain specific preferences. Failing the existence of P_2, we will be satisfied with the solution offered by P_1. They classify preferences as fluent or action preferences (and agree that both are analogous as stated previously) and choice or temporal preferences. Choice preferences are those concerning the method of attainment of a subgoal, whereas temporal preferences are those concerning the ordering of subgoals.

An example of hard control knowledge is the preclusion of sequential complementary actions. Where actions a and b exist such that a changes state S_1 to S_2 and b changes state S_2 to S_1, both with strictly no effect on any goals or subgoals in the plan, sequential applications of a and b will never have any positive outcome. Hard control information would specify that an a action never directly follows a b action and vice versa. There would never be any benefit in ignoring this rule, as doing so could only introduce redundant steps in the plan (according to the definition of a and b). If hard control information is not completeness preserving then this necessarily introduces incompleteness into the search space.

Examples of soft control include *preference* rules (see below) or the knowledge that some plan fragment $a_1, \ldots, a_n$ achieves some goal G (such as a macro-operator [36]). In general solution generation, this plan fragment is useful when G needs establishing. If, however, in the course of solution generation the goal

G arises then it may be the case that the actions $a_1, \ldots, a_m$ where $m < n$ are redundant and the whole plan fragment may not be needed. It is then pointless (and possibly illegal) to invoke the entire plan fragment. The planner in this case would need the option of overriding the suggested control information in order to plan efficiently. Consider a very simple *blocksworld* problem with one gripper and three blocks, a, b and c. a is on the table, b is on a and c is on b. The goal specifies that both b and c should be on the table. Our soft control information tells us that if we want c to be on the table, we should remove blocks from on top of c, then pick c up, then put it down on the table. If we try to achieve $on(b,$ *table*$)$ first, we will initially take c and put it on the table. When we go on to try and achieve $on(c,$ *table*$)$, we no longer need to use the entire plan fragment given as soft control information (in fact if we work sequentially through its suggested actions, we may end up picking up c only to put it straight back down again!). Both hard and soft control knowledge may have positive or negative manifestations. These will be called prescriptive and restrictive respectively.

Restrictive Rules

Restrictive (or rejection) rules give information on when *not* to follow certain courses of action. They can be used for simple or complex control strategies, one of the most basic uses being to stop successive applications of an action and its complement (see example above).

Hard restrictive rules are used to prune branches from the search space that do not adhere to control information. They can be described by sentences of the form

> Given the state S_1, never go to state S_2.

To see the effect of hard restrictive advice on pruning branches of the search space, consider the following restrictive rule.

> In any state where the literals $A(X, Y)$ and $B(X, Z)$ are true, $C(Y, Z)$ should not be true in the following state.

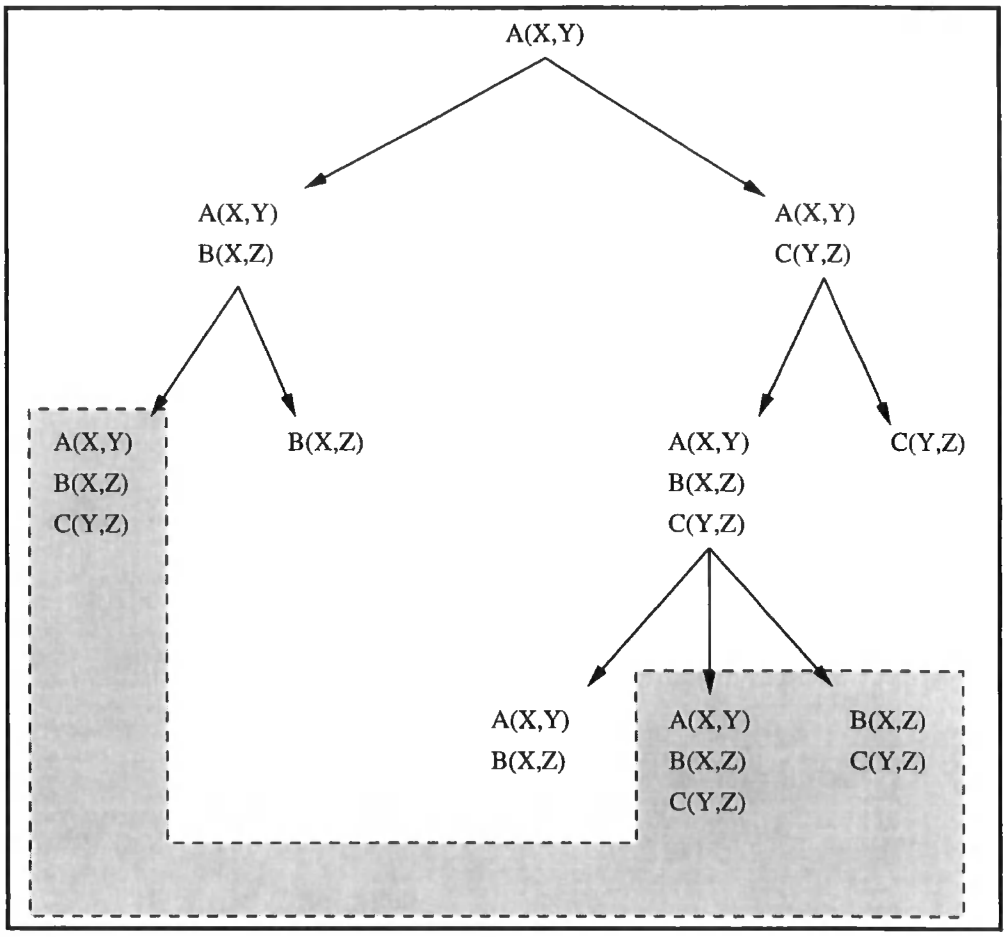

Figure 2.1: Rejection rules in the search space

Figure 2.1 show how a search space can be restricted using a rejection rule. The nodes are the collection of literals true in each state, the arrows show the transitions between states and the shaded region shows the nodes that would be pruned from the search space following application of the rule. Soft restrictive rules would not sever the branches concerned, rather queue those expansions at the end of the possibilities to be tried from a given current state. In this way it becomes a negative preference rule; that is, all other possibilities are tried in preference.

Prescriptive Rules

Hard prescriptive (or selection) rules offer positive advice. They are used to tell the planner exactly what to do in certain situations. They can be described by sentences of the form

Given the state S_1, go to state S_2.

To see the effect of hard prescriptive advice on traversal of a search space, consider the following prescriptive rule.

In any state where the literals $A(X,Y)$ and $B(X,Z)$ are true, $C(Y,Z)$ should be true in the following state.

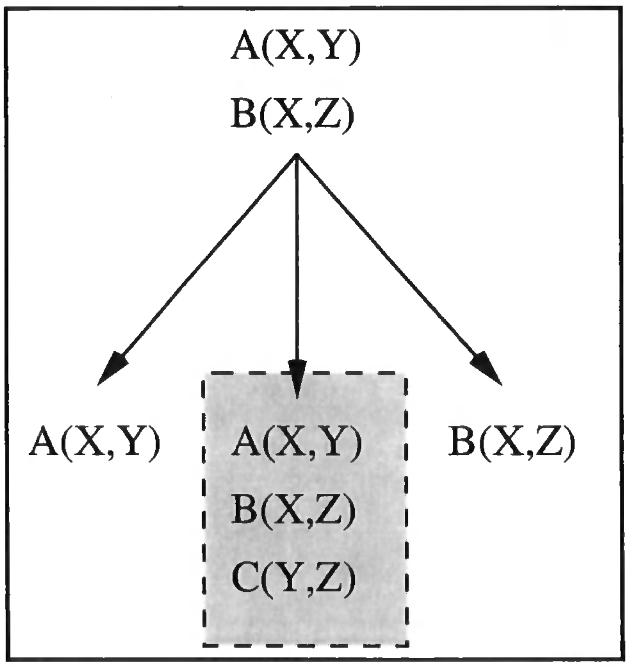

Figure 2.2: Selection rules in the search space

Figure 2.2 shows the selection of the successor node according to the prescriptive rule being used. The shaded area shows the successors as selected by the prescriptive rule.

There are, however, various issues concerning selection rules. It is not clear how to resolve conflicts between pieces of hard positive advice; in order to adhere to the control information, all prescriptive advice should be followed. In states where multiple selection rules are appropriate, the direction to the planner is that that the next state should be as specified by all the control rules. This could mean that the domain needs to be in many different states in the next state, which may be a contradiction (construction of parallel plans might deal with some cases, but if the prescriptive strategies interact at the level of preconditions and effects then problems still exist). A simple way to avoid such issues is the use of soft prescriptive rules, or *preference* rules. Instead of telling the planner exactly what it must do to progress at plan generation, they give advice on which courses of action are to be preferred. This avoids the problems of conflicts that prescriptive rules incur, as there is no obligation to follow the advice given. This allows the planner to delay or disregard any of the set of conflicting courses of action at its own discretion (obviously the planner still has the choice to make between conflicting actions, but this is encountered in any plan generation anyway).

2.5.3 Heuristics as control information

Heuristic evaluations as used in search are an attempt at formalising a value of worth, and allow a planner to make comparisons between alternatives. Most commonly, states are evaluated by some heuristic function to give some indicator of that state's worth in the search for an optimal solution, though heuristics relating to metrics other than plan length do exist (such as minimising fuel consumption in a transportation domain). Heuristic functions vary in their approach, from counting the number of literals in a state that match the goal specification to some estimate of a state's distance (in some defined sense) from the goal.

Nearly all planning systems use some form of heuristic for more efficient traversal of the search space. Those that do not must rely on brute force search (which is impractical in all but the most trivial of instances), or else are guided by explicit direction of which actions to consider (such as HTN based planners). Heuristics are a form of controlling the search for a solution, albeit a very different form of control to the domain specific rules that are usually intended with the term 'control information'.

The heuristics employed by a planning algorithm direct the global search strategy of the system, and in contrast to control knowledge do not concern a particular domain, type of domain, problem instance or top level goal. However, a planner's heuristics *do* hold significant influence over the traversal of the search space under consideration. Let us consider FF [29], the forward chaining state-based planner. FF uses the *relaxed plan* heuristic [28] to estimate the worth of a state, as an indication of the distance from that state to the desired goal state. The relaxed plan heuristic relaxes the planning problem by ignoring the delete effects of actions. This relaxation allows a solution to the particular planning problem to be found very quickly (not necessarily the optimal solution, however). The literature shows us that heuristic guided search can be a competitive approach to planning (FF is a prime example of how powerful the approach can be), but the performance of these systems can be further improved by employing domain specific control knowledge (see Section 5.2).

2.5.4 Domain specific solvers and policies

Hybrid systems can employ dedicated problem solvers to solve either the whole of, or various aspects of, a problem. There have been numerous examples of hybrid systems presented [22], [33], [17], though most use a selection of planning algorithms rather than specialist solvers. Using specialist solvers may have many performance benefits, but also suffers from significant drawbacks. Only exact problems can be tackled by the dedicated solvers, and given any other problem structure there is non-trivial work in the integration of solutions and plan construction, as suggest in [22]. An interesting new method of problem reformulation by Smith [52] is described in Section 2.8.

An alternative to the use of domain specific solvers is the construction of domain-specific *policies*. Policies are mappings from planning problems to actions, and can be seen as macro operators [36] in the domain (possibly for the entire planning problem). These can be constructed by hand, though recent work such as that by Baral and Eiter [8] and Fern, Yoon and Givan [19] have looked at automatic generation. Policies can be seen to represent control strategies within a domain.

2.6 Domain Analysis

Domain analysis is the process by which a domain is examined with a view to either discovering useful properties or truths that are not explicit in the domain description or verifying some properties that are expected to hold. The prime motivation behind domain analysis is to provide the planner with more information about a domain ultimately to aid the construction of a plan, although domain verification is also enabled (domain analysis techniques can highlight errors or unforseen consequences in domain specification). Typically, domain analysis is performed as pre-processing before the planner is invoked. The planner may then use the results of the analysis in many ways, including pruning the search space or guiding the search through preferences. Here we discuss the process of domain analysis itself from the stance of using the findings to help construct plans, as opposed to domain verification.

Broadly speaking, there are two ways to approach domain analysis; either we can do it ourselves, or we can try and get a computer to do it. The first attempts were by hand, using (as Kautz and Selman observed)

> "introspection to capture both 'obvious' inferences that are hard to deduce mechanically [41] and simplifying assumptions that follow from abstracting the essence of the domain."[32]

However, Kautz and Selman acknowledged that the manual approach has its limitations. A similar acknowledgement was made by van Beek and Chen [56] with regard to planning as constraint satisfaction (of which satisfiability is an instance), who noted

> "For each new domain, a robust CSP model must be developed. The modelling phase can require much intellectual effort..."[56]

Kautz and Selman, in their attempts at solving planning problems as satisfiability, augmented the domain description with the results of their manual domain analysis. They identified four classes of 'heuristics', though within this term they included state invariants as well as simplifying assumptions:

Conflicts and derived effects: Derived from the operator axioms only by comparing the precondition and delete lists, these represent the information about those operators that conflict with each other if applied to the same arguments.

State invariants: Derived from the operator and initial state axioms, these represent the invariants that hold in all reachable states of the problem, such as the single valuedness of the location of an object.

Optimality conditions: Derived from the operator and initial state axioms along with a plan length n, these represent the conditions that are true of a solution of minimal length.

Simplifying assumptions: Not derived, these simplify the domain and as such introduce incompleteness into the solution space.

Kautz and Selman make no mention of the labour or time spent analysing domains, nor of the possibility of human error, but it is natural to assume these were among the limitations they accepted.

Invariants (also referred to as *state constraints* or *domain constraints* in the literature) are truths about the initial state that are preserved over the application of any action. Many domain analysis techniques are concerned with invariants, which are implicit in the domain and problem descriptions. Other aims of domain analysis include typing of objects and identification of known sub-structures.

Following work on inferring operator parameter constraints [25] (or *parameter domains*) to speed up SATPLAN [31], Gerevini and Schubert went on to look at other automatic domain analysis techniques with their system, DISCOPLAN [26]. By looking at operator definitions, they were able to generate various kinds of invariants. The basis of their analysis was a hypothesise and test strategy, which allowed them to identify *implicative constraints* (of the form

$$((IMPLIES\ \phi\ \psi)\ \sigma_1, \ldots, \sigma_k)$$

where the implication holds when

$$\sigma_1, \dots, \sigma_k$$

are true), *single-valuedness constraints* and combinations of these two. An algorithm for determining predicate domains was also presented, based on IPP's code for plan graph construction [35]. They went on to extend the work to cover conditional effects within their previous analysis, typing of objects (according to type predicates), *antisymmetry constraints* (a special form of implicative constraints) and *XOR-constraints* (a form of constraint capturing the exclusive disjunctions of properties of objects).

There are many algorithms presented solely for the discovery of invariants. Kelleher and Cohn [34] present an early attempt at constructing invariants based on on restricted operator forms, in which pairs of facts are examined for particular relations (they introduced the notion of a *persistent precondition*). Later work has included invariant synthesis as an extension of type analysis [21] as discussed in Section 2.7, analysis of the initial state and successive operator applications [47] and analysis of operator descriptions only to search for *c-constraints* (based on the consumption of atoms by operators) [50].

Although work on the synthesis of domain specific knowledge is not new, many planners still use manually generated hand coded information. Their success in competition with fully automatic planners shows us that the automatic synthesis is currently still inferior to the human approach. The TLPlan and TALPlanner systems, for all their performance achievements, still rely on human domain analysis. The manual domain analysis in these cases is used to generate temporal logic control strategies, state based in the case of TLPlan and action based in the case of TALPlanner. Certainly humans are generally better (currently) than machines at constructing high level strategies, but humans are also more error-prone in their analysis. It has already been demonstrated that automated domain analysis techniques can uncover structure in a domain that humans have overlooked (the *paintwall* domain [38]).

There have been some recent attempts to integrate automatically derived information into solving planning problems. Policies can be learned from random walks in conjunction with approximate policy iteration, as demonstrated by Fern, Yoon and Givan [19]. These policies are shown to give performance competitive with FF in many domains from AIPS-2000, but there are decisions in the process that still require human input. Baral and Eiter describe the automatic construction of *k-maintainable* policies [8], a particular type of policy used for making guarantees about properties under time and exogenous event restrictions. This work uses a SAT encoding of the planning problem to construct its policies. Younes and Simmons describe the generation of policies for continuous-time probabilistic planning problems with concurrency [58]. This work uses a relaxation of the probabilistic element of the problem, and constructs a deterministic plan from which initial policies are generated. These initial policies are used to seed the policy search algorithm.

As the field matures, the available technology is improving towards the goal of totally relying on automatic analysis techniques to aid the construction of domain specific solutions, though as yet this remains a target.

2.7 Type inference

Some domain analysis is concerned with the typing of objects, that is meaningful classification based on particular properties. Fox and Long [21] described a systematic process of type inference through their Type Inference Module (TIM), which constructs a type hierarchy for objects in a domain. Type identification is achieved by viewing the planning domain as a set of finite state machines (FSMs) for each domain object. The states of the FSMs are the properties an object can have, where a property is the potential of that object to participate in a particular part of a relationship within the domain. Properties are distinguished by naming them according to the domain description, i.e. names are given to properties comprised of a predicate name and an argument position within that predicate.

For example, given the fact

$$X(Y, Z)$$

object Y is said to have property X_1 and Z the property X_2. From the initial state in a problem, a reachability analysis is performed to show every property it is possible for all objects to have; i.e. all actions are performed in each reachable state to give other reachable states, and the set of properties associated with each object monotonically increases. Objects that have like sets of properties are said to be of the same type, as they can share the same relationships with other objects and change those relationships by use of the same operators. This not only classifies the objects in the domain, but also gives type information about operator parameters. As a result, search can instantly be restricted to those actions whose arguments fit the type descriptions and any unit type predicates can be evaluated to boolean values (e.g. in the *logistics* domain [42], *truck*(*truck*1) would evaluate to TRUE while *truck*(*package*1) would evaluate to FALSE).

The type inference machinery provided by Fox and Long also allows the extraction of various kinds of invariants. Four classes of invariant are described, along with the algorithms for inferring them. *Identity invariants*, *state membership invariants* and *uniqueness invariants* are inferred from the property spaces integral to the type inference process and *fixed resource invariants* are constructed by examining the initial state and the operator definitions. The identity invariants are what Gerevini and Schubert called single-valuedness constraints [26], but the systematic analysis performed by TIM provide us with a richer set of constraints.

2.7.1 Generic Types

Following on from their work on type identification, Fox and Long presented the concept of the *generic type* [38]. These are meta-types, populated by classes of types according to some classification.

It was observed that various types across different domains shared some basic characteristics. Initially an attempt at problem decomposition in order to solve routefinding aspects of larger problems more efficiently, the ability to formally group types according to their behaviour has become a useful classification. Once type inference is complete, the FSMs relating to types can be examined. Regardless of the domain environment and details such as predicate and object names, types whose FSMs are of similar topology have analogous behaviour; i.e. the relationships they have with other objects and the operators used to change those relationships are directly equivalent in terms of their functionality.

Several generic types have been recognised and been given intuitive interpretations, including construction types [12] (where composite objects can be constructed out of component objects according to some rules), maps of locations with portable objects and vehicles and multiprocessor scheduling problems. However, it is worth noting that the interpretations that we attach to generic types are solely for our own understanding; the generic type itself only describes a behavioural pattern regardless of the actual realisation of the type in the domain.

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 *portable object type.* The distinguishing feature of a portable object is that is can be transported around a map of locations by an object of an associated *carrier* type. The portable object can be located at a location, loaded into a carrier and unloaded from a carrier. Carriers have the ability to make self-propelled movements from one location to another, according to the map. Figure 2.3 shows how these associated types relate to each other in a domain. This diagram is important as it shows how generic types exists in *clusters* of inter-related types.

Definition 1 *A* generic cluster *is a set of interacting generic types, whose member types are each required for the expression of the composite generic behaviour of the cluster.*

A special instance of the portable object type is the *safe portable object type* (or *SPOT*). A SPOT has all the features of the portable object type, but specifically *never have any other role in the plan* than to be located somewhere (commonly they have a specified goal location). *Safe portable objects* (or *SPO*s) are distinguished from other portable objects precisely because it is safe to transport them, without affecting other processes in the domain. A

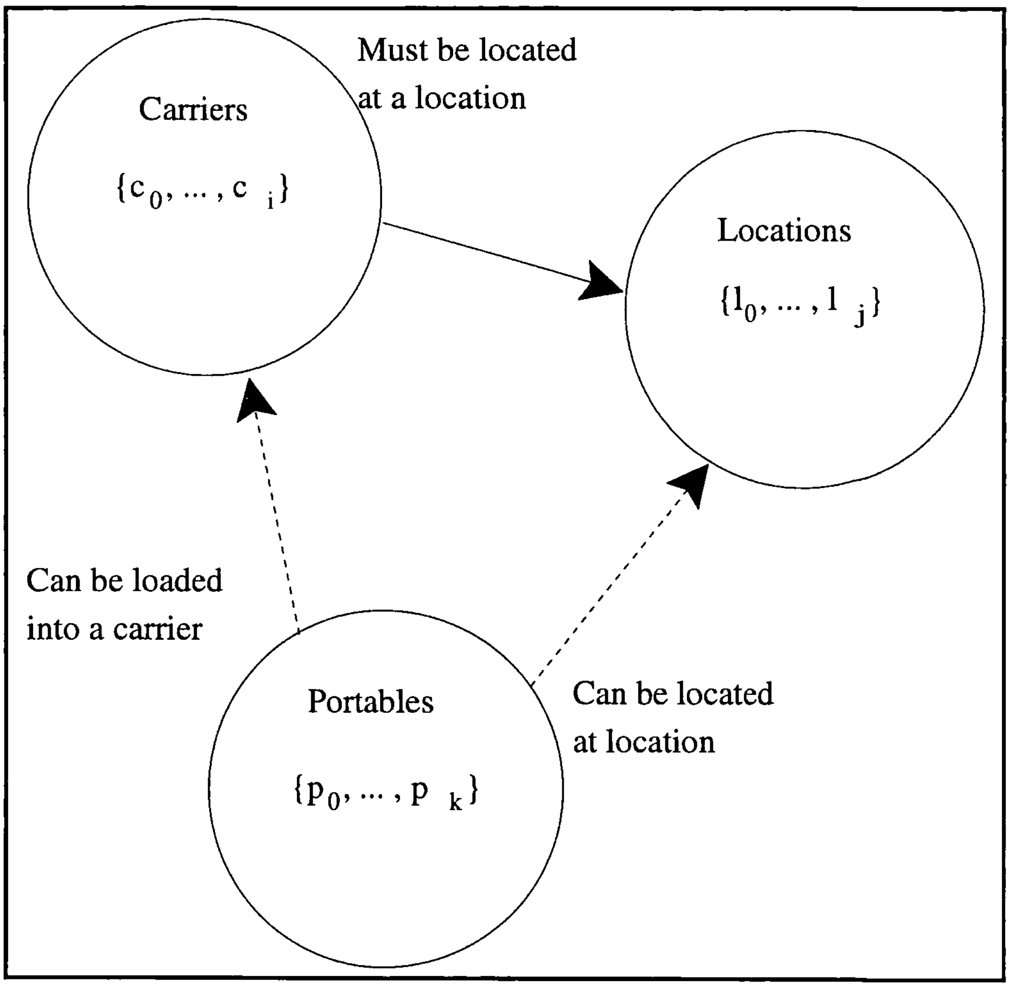
Carriers
{c0, ..., c i}
Must be located at a location
Locations
{l0, ..., l j}
Can be loaded into a carrier
Portables
{p0, ..., p k}
Can be located at location

Figure 2.3: The relation between generic types

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. For example, the *logistics* domain [42] is a simple transportation domain, in which packages can be transported around a map of cities by airplanes and around maps of city locations by trucks. The trucks and airplanes are identified as two types of carriers, each with an associated map of locations, and the packages are identified as portables. Furthermore, the packages are deemed *safe* portables as they have no other role than to be located somewhere; if the packages had in fact been keys for particular *unlock* operations they would not have been identified as safe.

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 [42] (a domain in which a number of ball objects must be moved between two rooms by a robot with two grippers) where the balls are identified as SPOs, *in-room* is the appropriate predicate. The other features that all members of a SPOT possess 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 portable object type and the members of that type have no other role in the plan than to be transported between locations.

There have been steps taken to find a suitable language with which to formalise the results of the generic type analysis, with a view to making the analysis available to the wider community; however, to date it is only STAN [38] that is able to make full use of the structures uncovered in domains.

2.8 Uniting domain analysis and control rules

An attempt to unite automatic domain analysis with search control in a planning algorithm has been made by Fox and Long in their graphplan-based system, STAN [38]. Originally only designed to recognise route-planning subproblems, the partnership of STAN and TIM (the earlier type inference module) employed pathfinding heuristics with great success in applicable domains (such as *logistics*). This work progressed as more generic types were identified, and became more directed towards problem decomposition. This is an approach to planning whereby a top level problem is decomposed into subproblems which in turn are solved by their own dedicated solvers. This approach reaps the benefits of solving smaller problems using dedicated technology, but integrating the sub-solutions into a solution to the top level problem is a non-trivial process.

More recently, Smith has offered an alternative approach to problem decomposition [52]. In this novel work, *over-subscription* planning problems are decomposed into orienteering problems, the solutions to which are used to offer goal ordering control information to a partial order causal link (POCL) planner solving the original problem. Over subscription planning problems model situations in which there are a large number of goals of varying value and the planner must decide on a subset of these to achieve within the time and resource constraints. Smith presents not only a manually controlled running example, but also describes techniques for automating the whole process (subject to manual tuning of the sensitivity analysis involved, though this too could be automated through a simple learning process). Although he admits the limitations of this abstraction of the problem, Smith's techniques are widely applicable. The identification of the orienteering subproblem reminds us very much of the classification offered by Fox and Long in their generic types. Although these two techniques are both based on the behaviour of the domain, the analysis is not similar and the approaches in identifying the subproblem are, as a result, quite different.

The use of the orienteering problem as a problem abstraction is also very similar to the use of relaxed planning graphs for heuristic evaluation (such as in FF [29]), but additionally considers orderings (according to the set of propositions that are used as the basis of the abstraction to the orienteering problem) that are necessarily lost in the relaxed planning graph. The information gained is also used differently (heuristic evaluation versus goal ordering information) and this is reflected in the decision to supply the information gathered to a POCL planner.

Recent work has considered generating pruning constraints from state invariants [37]. State invariants can be automatically generated by several different methods (see 2.6) and though the work currently involves inputting the state contraints (of the type that *can* be automatically generated) manually, an intended progression is to automate this part of the process too.

2.9 Chapter Summary

In this chapter the methods by which control information has been used by planning algorithms in the literature have been examined. Control information has been either integrated with or separated from the algorithms employing it. Where control strategies were not tied to a planning algorithm, those strategies were either integrated into the domain physics or given as declarative rules over and above the domain description.

Different types of control information have been considered, as have the different methods of expoiting each type. The effect of general search heuristics employed by planning algorithms on the traversal of the search space was also discussed.

Varying approaches to domain analysis were presented and comments made on the type of information extracted by such techniques. The work on type inference of objects in planning domains and its subsequent extension to generic types was introduced and a definition of the generic cluster was given as a set of interacting generic types.

Finally, attempts at marrying the processes of domain analysis and control of search were considered, in which the structures involved in or information discovered about a problem can be used to aid solution generation.

Chapter 3

Abstract control rules

3.1 Overview of Proposed Architecture

The integration of the generic control rule extension to provide a planner with domain specific control knowledge involves several components. The generic type identification machinery is required to discover the presence of any generic clusters. A collection of abstract control rules indexed by the generic clusters they describe forms a library. An instantiation component is needed to create domain specific realisations of the abstract rules from the library for the planner to use. This architecture is shown in figure 3.1.

Given a declarative domain description and problem instance, the generic type identification machinery will be invoked. This will give information on any generic clusters that are present in the particular domain. Any clusters identified will be passed to the generic control rule extension with the specifics of their realisation in the domain. The generic control rule extension then requests any rules in the library that pertain to the clusters that have been identified in the domain. The generic control rule extension has the information from the generic type identification machinery to provide domain specific instances of the abstract control rules in the library. The generic control rule extension then provides the planning system with the domain specific control rules it has instantiated. There may be some translation procedure needed to provide the planner with the domain specific rules in an appropriate form,

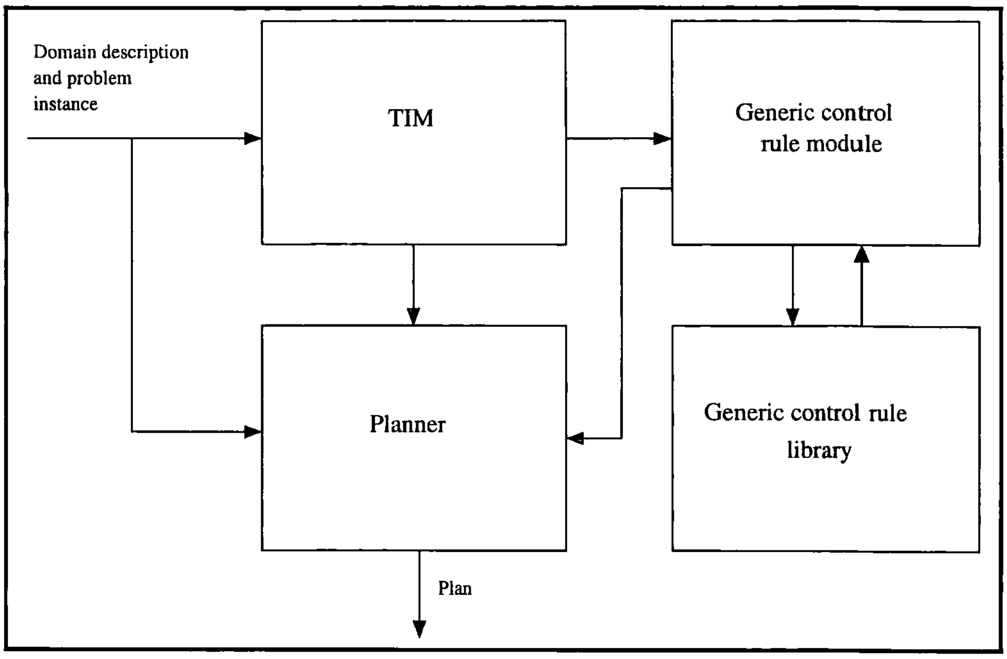
Domain description and problem instance
TIM
Generic control rule module
Planner
Generic control rule library
Plan

Figure 3.1: System architecture

depending on the planning system employed.

3.2 Components

3.2.1 Generic Types and Clusters

The idea of abstracting control knowledge across different domains is not new. The work of Fox and Long [38] explored using generic search control strategies for a particular behaviour pattern (specifically route planning across some map of locations). However, this was an integrated approach in partnership with the hybrid Graphplan [9] and forward-chaining planner, STAN4 [22] (the route planning subproblems were handled by the forward chaining planner, FORPLAN [22]). Though the work progressed to look at the more general case of problem decomposition (through the consideration of a larger number of generic types), no language was specified to represent the information relating to the generic structures outside the compound system of STAN4. This meant that the information that STAN4 had available could not easily be used by other planners or the rest of the planning community.

Generic types provide abstractions of the behaviour of types that occur in planning domains. They describe the way in which members of instances of generic types interact with other objects in a domain. However, it is rarely (if ever) useful to consider generic types in isolation. As previously stated, the very behaviour that is abstracted by generic types describes not just the type's own characteristics but its changing relationships with *other generic types* (or other objects of similar generic type). This has led to the proposal of a *generic cluster*, which is precisely a set of interacting generic types. A generic cluster is identified in the same manner as a generic type, i.e. by looking at the fingerprint of the finite state machines describing a type's properties. An instance of a cluster is identified by the fingerprint of the collection of finite state machines for the related types. By grouping together interacting sets of generic types into clusters, we encapsulate patterns of behaviour for interacting sets of types.

The generic types that form the member types of a cluster are meta-types (higher level types populated by domain level types). So the cluster is a similar abstraction of the behaviour of interacting sets of types. In the same way that generic types allow us to abstract from the concrete realisations of domain level types, the cluster affords us an abstraction of the concrete realisations of interacting sets of types. For example, in a mobile object cluster (in which mobile objects move on a map of locations) we are not concerned, at the cluster level, with the manifestation of the relationship that relates mobile objects to their locations. The domain realisation may be $at(< mobile >, < location >)$, $wibble(< location >, < mobile >)$, $at_{truck}(< location >)$, etc. but the abstraction to the cluster level is concerned only with the fact that the mobile is related, through *some* domain relation, to the location object (though that relationship must behave in a prescribed manner for the types to be identified as mobiles and locations).

Arguably, the grouping of related generic types has been implicit in any discussions about generic behaviour of any types. It simply does not make sense to look at *process* types without also considering *processor* types (as in multi-processor scheduling) or *mobile* types without considering their associated maps of *location* types (as in transportation domains). However, no formal concept of a group or set has been proposed to date.

The acknowledgment of a generic cluster allows the discussion of the interaction of the generic types involved to be more precise and can provide a basis for ensuring the right domain types are considered with respect to each other (e.g. for domains with multiple MPS aspects, relating a process type to the right processor type). It also facilitates dealing with domain types that show more than one kind of generic behaviour. The identification and classification of generic clusters is not, however, the focus of this thesis.

Objects within an instance of a generic cluster interact with each other in known ways (according to the nature of the cluster). Using the knowledge of how these objects behave allows us to write various control strategies for the objects involved. Furthermore, because the generic cluster represents an abstraction of the behaviour of the objects involved, any control information that we construct is similarly abstracted. This abstracted control information can be made available for any instance of the particular generic cluster, saving the work that would normally be invested in formulating and encoding similar control rules for every instance.

Definition 2 *A* generic control rule *is an abstracted control strategy expressed as a logical statement whose atoms are features of a generic cluster.* (the term GCR may be used as shorthand for *generic control rule*)

The recognition of an instance of a generic cluster gives us some information about how the search space for problems in that domain is structured. By writing control strategies at the level of the generic cluster, that structuring is described for all instances of the cluster.

3.2.2 Language for abstract control rules

As discussed in Section 2.5, control rules can be of several forms. Importantly, we shall be considering only pruning rules (hard restrictive rules). Issues regarding the preservation of completeness of the search space while using pruning rules are discussed in Section 4.7.

A control rule is a logical statement describing the properties of some sequence of states. Given an actual sequence of states, the control rule will have an evaluation that corresponds to whether that sequence has the specified properties or not. If the sequence of states is not inconsistent with the properties described by the rule it will evaluate to *TRUE*, otherwise it will yield *FALSE*. In the light of these evaluations, it is possible to identify paths in the search space that do not follow the control strategies described; they are those paths that cause an evaluation of *FALSE*. Conversely, those sequences of states that do follow the strategy represented by a control rule will cause that rule to evaluate to *TRUE*.

For example, given the modal control strategy

$$a(X,\ Y) \wedge GOAL\ a(X,\ Y) \rightarrow NEXT\ a(X,\ Y)$$

in a problem instance where the goal condition contains the literal $a(X,\ Y)$, the sequence

$$\begin{aligned} a(X,\ Y) &\in state_{current} \\ a(X,\ Y) &\notin state_{next} \end{aligned}$$

would evaluate to *FALSE* (indicating that the sequence contravened the control strategy), where as the sequence

$$\begin{aligned} a(X,\ Y) &\in state_{current} \\ a(X,\ Y) &\in state_{next} \end{aligned}$$

would evaluate to *TRUE*.

Having looked at previous languages for expressing control knowledge, such as the action-based TAL used by TALPlanner [16] and the state based modal logic employed by Bacchus and Kabanza [3], it was observed that a new language was required in order to be able to abstract the control knowledge above the domain level. None of the languages to date have been able to express meta-constraints on the search space or dealt with any form of generic behaviour across different types that the use of generic clusters allows.

The most obvious difference between GCR logic and other previously presented logics for expressing control information (LTL[18], MITL [4], etc.) is the level at which that information is expressed. To date, only object level languages have been presented that specify control strategies at the level of objects in a particular domain, though the objects that are used may be object variables, quantified either over all domain constants or some bounded set of domain constants.

The proposed language will be able to express generic control information at a level abstracted from any particular domain realisation. Specifically, this demands not only the abstraction of predicate and object names, but also of the predicate *structure* used to express some generic semantic relationship between objects.

Expressing abstract forms of control rules necessitates abstract entities in terms of which the control rules can be formulated. Generic clusters provide precisely those entities. The language that expresses the control rules at the abstract level expresses the control information *in terms of the generic cluster*. This describes the control information for all instances of the generic cluster. For the purpose of the work described in this thesis, control rules are assumed to use only a single generic cluster. This decision is discussed in Section 4.4.

The generic cluster structure tells us how objects that belong to the member types relate to each other. Using the knowledge that such relationships exist allows us to refer to objects *as a function of their relation to some known object.* For example, objects of generic type portable are related to objects of generic type location by some locatedness predicate. At the level of the generic cluster, we have no information about any particular manifestation of that relationship, but we do know that such a relationship exists, in some form, for any instance of the cluster. As such it is possible to refer to the location of some portable object X of type $T_{portable}$ by the locatedness relationship specific to $T_{portable}$, *regardless of the actual manifestation of that relationship for any type T_x that is an instance of $T_{portable}$*.

Let us look at an example control rule to see how the abstraction to the generic cluster applies. A simple strategy for safe portable objects is

Do not move the safe portable object once it is at its goal location.

In the *logistics* domain, this rule could be formulated as

$$\forall x : package\ .\ \forall y : location\ .\ at(x,y) \wedge GOAL\ at(x,y) \rightarrow NEXT\ at(x,y)$$

If we try and abstract this rule to the generic cluster level, we will have no information about the predicates that express locatedness (i.e. the 'at' predicate in the above example). We simply know that safe portables can be located at location objects, and that when a safe portable is located at the location object specified as its goal location, then the safe portable should maintain that location. So we can re-formulate the control strategy above as

$\forall$ safe portable objects x, if $currentLocationOf(x) = goalLocationOf(x)$
then $nextLocationOf(x) = currentLocationOf(x)$

In the above example, we see the reference to objects through their relationships with other known objects. In particular, equality is asserted on the location object to which the object x is related in both its current and goal states. This is done without any knowledge of how that relationship will be manifest in any given domain or any knowledge of the sequence of states in which the specified situation arises. Importantly, this rule can not only be re-instantiated into the *logistics* domain but can be instantiated into which ever domain in which the correct generic structure is recognised (i.e. the identification of safe portable objects and associated types).

If we consider taking this abstracted rule and specialising it back to the domain specific version, it is obvious that we need access to the particular predicates in the domain that manifest the relationships such as the locatedness of an object. The manifestation of these predicates is tied to the types involved in the concrete domain (in this case, the *logistics* domain). Put another way, the way in which the *currentLocationOf* function will be interpreted is tied to the type of the object x, let us call it T_x. To make this clear in the control rule, the function is subscripted with the type. This makes plain the dependence of the manifestation of the function on the type for which it is being instantiated. The control rule then appears as

$\forall$ safe portable objects x,
if $currentLocationOf_{T_x}(x) = goalLocationOf_{T_x}(x)$
then $nextLocationOf_{T_x}(x) = currentLocationOf_{T_x}(x)$

The proposed language of control rules already has certain features defined. As we will be working with abstractions rather than any particular domains, we will not have access to domain constants. The terms of the language must therefore be variables of some sort. We know also that we are classifying objects according to type, so the object variables will be typed. However, we know that at the abstract level, the details of any domain types are not known. This tells us our object variables will be typed, not with concrete types, but with type variables. Finally, we can see that the language will refer to objects not exclusively explicitly, but also by their relationships with other objects. This will be achieved using functions that abstract the domain level expression of some particular semantic relationship that the object's type is identified as exhibiting.

Definition 3 *A term describing an object by a function expressing its relationship to another object will be referred to as a* function term.

Giving a function term a state argument qualifies the state in which the relationship expressed through the function holds. The state in which the relationship holds is important, as the object referred to by the function term may be dependent on the state. The location of the mobile X may be different in the current and next states, for example. In this case, the function expressing locatedness of X with the state argument denoting the current state returns a different object to the same function with the state argument denoting the successor to the current state. Using a state argument denoting the goal state of the problem allows objects to be referred to through their relationships as specified by the problem specific goal set. In the above examples, the state argument is implicit in the function (there are different functions for the location of objects in the current, goal and next states). This approach, however, involves multiple incarnations of functions to represent every state that is referenced.

An explicit state argument allows the same function to be used, the state argument taking responsibility for the specification of state. The function

$$currentLocationOf_{T_x}(x)$$

becomes

$$LocationOf_{T_x}(x, now)$$

The function

$$goalLocationOf_{T_x}(x)$$

becomes

$$LocationOf_{T_x}(x, goal)$$

The function

$$nextLocationOf_{T_x}(x)$$

becomes

$$LocationOf_{T_x}(x, next)$$

The abstraction from state specific functions reflects the domain level relationships, which remain the same regardless of the state in which they hold (a predicate expressing locatedness is the same predicate no matter which state is being considered).

Imposing conditions such as equality on the objects involved in particular relationships facilitates the specification of configurations of those objects. We can, for example, describe any two mobile objects x and x' that are located at the same location object by asserting an equality between the objects referred to by functions expressing the locatedness of x and x'.

$$\begin{array}{l} \forall \text{ mobile objects } x,x' \\ LocationOf_{T_x}(x, now) = LocationOf_{T'_x}(x', now) \end{array}$$

Asserting conditions on objects specified by functions involving different state arguments allows the specification of conditions that span states. Objects that are in relationships that match their goal relationships can be described by equality on the function terms for both the current and goal states. To describe a mobile object x that is situated at the location given as its goal location, we simply assert an equality on the function expressing its location in the current state and the function expressing its location in the goal state (the function remains the same, the state argument for the former term denotes the current state and the for the latter denotes the goal state).

$$\begin{array}{l}\forall \text{ mobile objects } x \\ LocationOf_{T_x}(x, now) = LocationOf_{T_x}(x, goal)\end{array}$$

Using state arguments that refer to sequences of states, relationships can be described that hold over that sequence of states. For example, a mobile object X that changes location between the current and next states can be described by an inequality between the function expressing its location with respect to the current state and the same function with respect to the successor to the current state.

A control rule is a logical statement describing some properties of a sequence of states. Properties of states can be described by objects' relations to other objects (and conditions on those relations) and these properties can span states by determining in which states those relations hold. As a result, abstracted control information can be expressed by the relationships described at the level of the generic cluster.

Control information is expressed as relationships between objects that hold over sequences of states in a plan trajectory. However, at the level of the generic cluster, there is no information on the objects that will populate the member types of the cluster other than the knowledge that there *will* be those sets of objects. As a result, the terms of this language are object variables quantified by type. The type is given as a type variable, as the member types of the cluster describe sets of domain types that exhibit the appropriate behaviour.

Taken in isolation, expressions at the level of the generic cluster can have no evaluation. In order to be evaluated, the expression requires a plan trajectory (i.e. a sequence of states for a particular domain). For example, the expression

$$\begin{array}{l}\forall \text{ mobile objects } x \\ LocationOf_{T_x}(x, now) = LocationOf_{T_x}(x, goal)\end{array}$$

needs both the current state (in order to assess $LocationOf_{T_x}(x, now)$) and the goal state. If there were references to states following the current state, these state descriptions would be needed too.

However, it is not a pre-requisite that the domain contain the structures in terms of which the expression is given. If an expression is evaluated with respect to a trace in a domain that does not contain the generic cluster used in the expression, the expression trivially evaluates to $TRUE$ (in the example above, the expression would evaluate to $TRUE$ in the case where there did not exist an x such that x was a mobile object). If, on the other hand, the domain *does* contain the appropriate generic clusters, then in order to evaluate the abstract expression with respect to the trace there must be a direct and known mapping between the abstract structures and features of the domain itself. This should seem perfectly reasonable; in order to evaluate the abstract expression with respect to the concrete realisation, we must have a complete mapping from the abstract structures to their instantiations in the concrete domain. The process of evaluation is given by the semantics presented in Section 3.6.3.

3.2.3 Exploiting features of a generic cluster

Knowing that types belong to a generic cluster gives information on how those types interact in their domain. The structure representing a cluster must be able to refer to the types that form that cluster.

In order to exploit the features of a cluster, two pieces of information are needed. Firstly, the types that form the cluster must be identified and secondly, there must be an indication as to how the types interact according to their generic behaviour.

Given an instance X of a domain type T_X identified to be an instance of a particular generic type within a generic cluster, it is possible to refer to objects related to X through the generic behaviour described by the cluster. For instance, a vehicle moving on a map locations will always be related to one location through its *locatedness* predicate. It is possible to describe the location through function application, using a function that gives the location of a vehicle object. Functions in general will return the object (or set of objects) to which X is related. Bearing this in mind, X will always be an argument of the function and as the function expresses relationships inherent to X we adopt the syntax of a member function as if that object were a class in an object-oriented language.

Definition of functions

Given two domain types T_X and T_Y that express some generic behaviour through the domain level predicate

$$p(< X : T_X >, < Y : T_Y >)$$

there will be two functions, one for each type involved, to access the objects of the other type to which they are related through this predicate. If X relates to precisely one Y and Y to possibly many Xs through p (i.e. p is a partial function) then the following functions will be defined:

$$\begin{aligned} p' &:: T_x \rightarrow T_y \\ p'\ x &= y \text{ where } p(x,\ y) \\ p'' &:: T_y \rightarrow [T_x] \\ p''\ y &= \{x \mid p(x,\ y)\} \end{aligned} \tag{3.1}$$

(N.B. The syntax of function definitions and function type declarations used here is borrowed from functional programming)

The functions are adopted as member functions of objects of the appropriate type. Binding the functions to the particular type highlights the fact that every instance of that type may have its own expression of the relationship described by the function. As a result we get X's corresponding Y given by

$$X.p'_{T_X}$$

while conversely the set of Xs related to Y is given by

$$Y.p''_{T_Y}$$

(this example may return an empty set, if there are no objects related to Y through p). If T_X is a vehicle type whose location type is T_Y where the location predicate is

$$at(< X : T_X >, < Y : T_Y >)$$

then

$$X.at'_{T_X}$$

returns the location of the object X and likewise

$$Y.at''_{T_Y}$$

returns the set of objects located at Y.

The example above shows one particular relation involved in the expression of some arbitrary generic behaviour. It demonstrates how referring to objects through functions representing generic behaviour at the abstract level can be interpreted to describe objects at the domain level. Notice that the functions can return both single objects and sets of objects.

There must be some declaration of the functions available for the member types of a generic cluster in order that we know which functions are defined. The functions are declared along with the member types in a structure known as a *prototype*. Details of this structure can be found in Section 3.5

3.2.4 Domain specific control rules

There must also be a language for expressing the control information at the level of instances of the clusters for which that information was formulated. An instance of a cluster is a set of types in a concrete domain whose objects interact according to the behaviour abstracted by the cluster. This behaviour is expressed by domain level predicates in which members of the types can participate (and the change in those predicates through operator application).

Control information at this level describes relationships between objects expressed through predicates inherited from the domain description. The language must therefore describe the control information in terms of the concrete domain. The terms of the predicates are domain constants or variables ranging over domain constants.

The predicates populated by domain constants give propositions. The evaluation of the proposition is a boolean describing the proposition's inclusion in the state description. Temporal modal operators give qualification of the state in which a proposition is to be evaluated. Expressions at this domain level will give a boolean result when evaluated with respect to a plan trajectory, i.e. a sequence of states visited by a plan. The evaluation denotes whether or not the particular sequence of states adheres to the conditions of the expression.

The description of the features of this language may appear to fit the definition of Emerson's LTL [18] and in fact the language being described *is* a subset of this previously defined logic. A more formal description of the subset of LTL that is used is given in Section 3.8. What should be remembered is that we are not claiming the originality of this language, rather that it is used as the language into which expressions of a higher level language are instantiated.

3.3 Generic Types

As discussed in Section 2.7.1, generic type identification is an extension of the type inference machinery. Some features that are apparent at the level of type inference are carried through and affect the behaviour of generic types in a generic cluster. In fact, the return types of the functions introduced in Section 3.2.3 are dependent on characteristics of structures that are constructed as an intrinsic part of the type inference and invariant generation processes. In order to examine this, let us first describe those structures in more detail.

We will use the portable objects cluster as a running example in this discussion. An instance of this cluster is a classic transportation domain, with packages (the portable objects), carriers (the vehicles that transport the packages) and a map of locations.

The first stage of type inference examines the operators in terms of each of their arguments. A property relating structures (or PRS) is built for each argument of an operator, where a PRS is a triple of bags of properties. The bags show the properties the argument has in each of the preconditions, positive effects and negative effects of the operator (where an object has more that one occurrence of a property, the bag contains multiple instances of that property). The bags of the PRS will be used to construct the transition rules that describe objects' behaviour, where the properties in the preconditions form the enablers of the transition from the properties described in the deleted preconditions bag to the properties described in the added properties bag.

An example of a transition rule derived from the *move* operator

```
(:action move
    :parameters   (?vehicle ?source ?destination)
    :precondition (and (carrier ?vehicle)
                       (at ?vehicle ?source)
                       (link ?source ?destination))
    :effect       (and (at ?vehicle ?destination)
                       (not (at ?vehicle ?source))))
```

would be

$$at_1 \longrightarrow at_1$$

This transition rule is formed for the *?vehicle* parameter, and has no enablers (the parameter does not appear in any persistent preconditions).An example of a transition rule with and enabler would be

$$link_1 \Longrightarrow at_2 \longrightarrow null$$

If either the added properties or the deleted preconditions bag is empty, the transition rules constructed are used to form *attribute spaces* and not *property spaces.* A property space is comprised of FSMs (showing the changes in properties), the objects that traverse them, the properties the objects can have and the transition rules describing the manner in which the objects' properties are changed. An attribute space contains the objects that can have, lose or

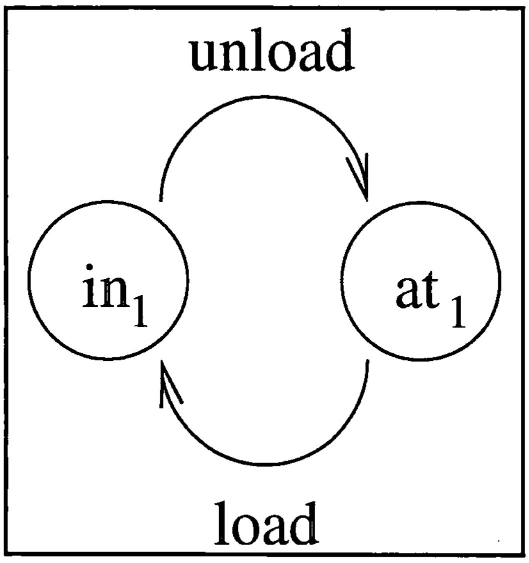

Figure 3.2: The properties of portable objects

acquire the described attributes, along with the transition rules pertaining to those losses and acquisitions.

An attribute is distinguished from a property by the inclusion of the *null* property in a FSM. The *null* property is used to denote the lack of a property lost or gained in a transition rule. This represents the object traversing the FSM acquiring or losing a property p without exchanging that property for another property q. Conversely, properties are always exchanged for properties (possibly the same property), which means the number of properties is fixed for objects of a specific type.

Figure 3.2 shows the FSM for portable objects (the packages in the transportation domain). The transitions represent classic *load* and *unload* operators, in which a package x changes from $at(x, < location >)$ to $in(x, < vehicle >)$. This FSMs is associated with a property space. Figure 3.3 depicts the FSM showing how location objects gain and lose the attribute of 'having something situated at them.'

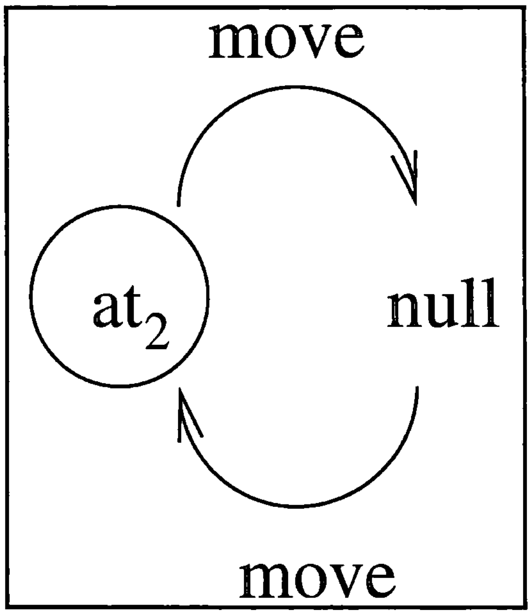

Figure 3.3: The attributes of location objects

The objects in the domain are grouped into types according to those attribute and property spaces in which they are included. The objects are in fact given a bit vector showing their inclusion and omission from each space constructed in the domain, and objects sharing the same vector are said to be of the same type.

Type analysis provides the basis for invariant generation. Only the property spaces are used for the generation of invariants, as the use of attribute spaces would yield incorrect invariants (the number of attributes that the associated objects can acquire is not encoded at the level of the attribute space). Three types of invariant are constructed from the property spaces, *identity invariants*, *state membership invariants* and *uniqueness invariants*.

State membership invariants show all the properties that objects of particular type can have, by creating a disjunction of propositions composed of predicates into which the objects can be instantiated (according to their properties). Uniqueness invariants show the mutual exclusivity of pairs of properties in the space, by asserting that the objects can not be simultaneously present in conflicting propositions. In both cases, the predicates are populated by existentially introduced variables to create propositions. Identity invariants show the single-valuedness of properties. When a property P_i occurs at most once in every state (where a state is the properties an object can have simultaneously, described by the initial state and reachability according to the transition rules) it is possible to restrict the propositions employing objects with the property P_i. This is achieved by asserting equality on the remaining arguments involved in multiple instances of an object x displaying the property P_i (an object displaying a property is that object in the correct position in the predicate P).

3.4 Inclusive and exclusive generic control rules

The question of the extent to which generic type identification describes a type's behaviour is interesting. A type is identified as of generic type if it displays the patterns of behaviour defined for that generic type. Whether or not that is the *only* behaviour it displays is not addressed in generic type identification. It is certainly possible that a type T identified as an instance of generic type T_G could have other behaviour in the domain separate from the behaviour described by T_G. From the point of view of GCRs, the important question is whether or not any of the other behaviours affects the behaviour described as generic or the interactions with the other generic types in the cluster.

This draws attention to the need to qualify both in which instances of a generic cluster the rules concerning that cluster apply and, conversely, which rules apply for varying instances of that cluster. Let us first address the latter, then return to the former. In fact we present a classification of rules into two classes, *inclusive* and *exclusive* and describe the conditions that necessitate this distinction. We also classify instances of generic types as inclusive or exclusive.

3.4.1 Inclusive and exclusive rules

Definition 4 Inclusive rules *are applicable based solely on the interaction of the types as described by the generic cluster. No matter what other behaviours may or may not exist for the types involved, the interaction of types according to the behaviour described by the generic cluster is the only thing that determines the applicability of the rule.*

An inclusive rule is based on behaviour that an instance of a cluster displays that is *totally described* by the cluster. That is to say, the behaviour on which an inclusive rule is based *must not* be affected by any other behaviours that the type exhibits outside its behaviour as described by the cluster to which it belongs.

Inclusive rules can include rules to prevent complementary action cycles (in which an action and its complement form a redundant cycle, reverting the world state to one strictly the same as before the cycle). An example of a complementary action cycle is unloading then reloading a package from truck in the *logistics* domain. A more subtle example of an inclusive rule is the rule that states that a member of a safe portable object type should never be moved once it reaches its goal location. In this particular example, the source of its inclusiveness stems from the restrictions involved in its type being identified as *safe* at the point of generic type identification (portables are qualified as safe precisely when they have no other role in the domain other than to be located somewhere).

Definition 5 Exclusive rules *exploit behaviours which* may *be affected by operations outside the scope of the generic cluster for which they are identified. They rely not just on the behaviour as described by the cluster, but the implied consequences of that behaviour* occurring exclusively.

This may sound like we are undermining the argument for abstracting control strategies to the level of the generic cluster, but let us examine the issue fully before any conclusions are drawn. Generic type identification (the basis for generic cluster identification) assigns a type the status of an instance of a generic type if it satisfies the appropriate conditions of behaviour. Except in special cases (such as qualifying a portable type as a *safe* portable type) this behavioural analysis is not exclusive with respect to any other behavioural aspects. In other words, type T identified as generic type C_G within generic cluster C may have not only behaviour other than its generic behaviour, but furthermore that behaviour may even affect its relationships with other member types of C.

Let us look at an example of how this situation might arise. The generic type *mobiles* are defined as able to make self-propelled (requiring no other objects/resources) transitions from being associated with one object $o_1 : T_x$ (through an identifiable predicate) to being associated with another object $o_2 : T_x$ ([38] gives a full description of the process of identifying mobile objects). As long as this required behaviour is a subset of the type T_{mobile}'s entire behaviour, it will be identified as being of generic type *mobile*. If, however, in addition to its mobile behaviour with respect to $\{o_1, o_2, \ldots, o_n\} : T_x$, T_{mobile} has a *BeamUp* operator which changes its members location by different rules to the previously outlined behaviour (this too would be identified separately as mobile behaviour), then certain control rules applicable to mobiles in general may have some unforseen or error-inducing consequences. For example, a general control strategy for mobiles is "do not enter a dead-end location (a location from which there are no exit routes) unless it is the mobile's goal location." This rule is completeness preserving in the case where the mobiles have no other behaviour (than its generic mobile behaviour) with respect to objects of the associated location type. However, if as suggested above there is further behaviour, for example in the form of the *BeamUp* operator (which teleports the mobile type between apparent dead end locations, according to the mobile map identified), then the completeness of the search space is threatened. It may be the case that applying the rule will prune branches of the search space in which solutions reside.

The example rule given above is a case of an exclusive rule. Exclusive rules are applicable where the interaction of the types involved is described by the particular cluster to which they belong *to the exclusion of any further* *interactive behaviour.* An interesting point mentioned in the example is the notion of preservation of completeness of the search space. The effect of control rules on the completeness of the space is discussed in Section 4.7.

3.4.2 Exclusive and inclusive instances

Instances of generic clusters can also be classified as *exclusive* or *inclusive.*

Definition 6 Exclusive instances *of a generic cluster are those instances in which the generic relationships do not appear in the preconditions or effects of any operators in the domain other than the generic operators associated with that cluster* and *in which those generic operators contain only preconditions and effects that are recognised in the generic cluster.*

This means that the generic behaviour of the member types is unaffected by any other patterns of interaction that they may be involved in in the domain, whether or not other such behaviour exists.

Definition 7 Inclusive instances *are those in which the predicates describing generic relationships* do *appear in the preconditions or effects of at least one operator that is not a generic operator* or *in which the generic operators contain preconditions or effects that are not recognised by the generic cluster.*

The member types' behaviour in the domain *includes* that described by the cluster, but is not exclusively defined by the cluster.

Inclusive rules are applicable to both exclusive and inclusive instances of the cluster over which they are quantified. Exclusive rules can be safely applied to exclusive instances of a cluster, but though it *may* be safe to apply them to an inclusive instance, there is no guarantee. The fact that inclusive instances of a cluster have additional behaviours (to those described by the cluster prototype) admits the possibility that those additional behaviours affect the interaction between the member types of the cluster. In this case, applying exclusive rules could result in solutions to the problems being lost (pruned) from the search space.

The classification of generic clusters as inclusive or exclusive goes some way towards answering the previous question, namely in which instances of a generic cluster the rules concerning that cluster apply. It also provides a basis for talking about hierarchies of types and clusters, and the implications on the applicability of rules within a hierarchy. This will be discussed in Section 4.3.

An examination into the relation between behaviour recognised as an instance of a generic cluster and additional behaviours (that may interact with the behaviour of the cluster) would provide an interesting extension to the work presented in this thesis. The classification of instances as exclusive or inclusive will also, in future work, assist the specification of rules for multiple clusters by qualifying to what extent instances of clusters interact with each other (c.f. Section 4.4).

The fact that exclusive rules are a subset of inclusive rules tells us that tighter constraints are possible using the former. This is intuitively verified, as we can be sure about the generic behaviour of an exclusive instance whereas we can only know that certain generic behaviour is *included* in the total behaviour of an inclusive instance. The generic type identification machinery identifies both inclusive and exclusive instances of clusters. For the sake of demonstration of concept, though we acknowledge the existence of inclusive and exclusive rules and instances of clusters, we will treat any rules and instances as exclusive. In most of the benchmark problems, this is a safe assumption.

3.5 Generic cluster prototypes

A generic cluster is an abstract structure that describes a collection of types and the interactions between those types that produce some known and interesting pattern of behaviour.

The abstraction of a set of interacting types that is provided by a generic cluster necessitates a way to describe the relationships between objects of the member types without reference to any domain specific realisation of those relationships. There also needs to be some description of the types involved in the cluster. A *generic cluster prototype* is provided to describe a cluster.

Definition 8 *A* prototype *of a cluster C is the structure that is used to declare the types of which C is comprised and to declare the relationships that hold between objects of the member types that produce the generic behaviour of C.*

The prototype describes, at an abstract level, the relations between parts of the structure that can be used to describe configurations of objects by their generic behaviour.

Where no reference can be made to the predicates that provide the relationships between objects, those relationships must be described in some other fashion. The information that must be captured is the fact that such a relationship exists, between which types of objects that relationship holds and what the characteristics of that relationship are (e.g. one-to-one, one-to-many). The functions as such are declared with an identifier, source and target types and the type of the relationship.

The prototype serves several purposes. Firstly, it gives the relationships that exist between objects of types that instantiate the cluster. In this role it introduces those relationships without any information over and above the fact that some relationship holds between the objects of the member types. This information specifies the interactions between the types that allow us to refer to objects by their relation to other objects.

Secondly, it allows us to say what it means for a control rule at the cluster level language to be well formed. That is, it specifies the functions that are available for objects variables according to type. The abstract syntax of the generic control rule language can be seen to describe a family of languages. Without any reference to the prototype of a cluster used in an expression, it is impossible to determine whether or not an expression is well formed or not as there is no information regarding the functions available for objects of the member types. The members of that family of languages are specified by the conjunction of that syntax and the generic prototype used in a given expression.

The declaration of the types and functions in a prototype gives a working model of the cluster. It outlines how the cluster can be used to describe objects that belong to its member types and as such how that information can be used in the context of writing control information. In this sense it provides a bridge between the abstract structures that are the clusters themselves and any instance of those clusters. The features that are defined for all instances of the cluster are given at the level of the structure that is the abstraction of any particular instance.

3.5.1 Prototypes and their features

A structure of a prototype is straight forward. There are several aspects of a cluster that need declaration in its prototype. Firstly the prototype needs a unique name. This allows the prototype to be related to the correct cluster. The issue of relating a prototype to a cluster is discussed in Section 3.5.4. Secondly the prototype needs a declaration of the types involved in the cluster. The types are given as unique type variables. Finally, a prototype needs the declaration of relationships that exist between objects of its member types as an expression of the behaviour of the cluster. These are given as member functions of the types declared.

The functions are given return types that show not only the type of object that is returned but also whether a single value or a set value is returned. The types that can appear in the return values are any subset of the union of all the member types declared for the cluster. The prototype also declares the single-valued functions that may return, in addition to a domain constant of the appropriate type, the special value $NULL$. The $NULL$ value may be returned where the function is descriptive of an exchanged property. This point is discussed further in Section 3.7. The functions that may return $NULL$ are suffixed with a "+".

The syntax of the prototype

The abstract syntax of a prototype is given by the following rules:

1. If C_{ID} is a unique cluster ID and B is a cluster body, then

$$\text{prototype } C_{ID} \; \{ \; B \; \}$$

is a cluster prototype.

2. If *tList* is a list of member generic types and *fList* is a list of member functions then

$$\begin{array}{l} \text{Types} : tList \\ \text{Functions} : fList \end{array}$$

is a cluster body.

3. If $GT_m, \ldots, GT_n$ are generic type IDs then $\{GT_m, \ldots, GT_n\}$ is a list of member generic types.

4. If f_{ID} is a function name, $GT_i \in \{GT_m, \ldots, GT_n\}$ and $GT_x \in \{\bigcup Ts | Ts \in \wp\{GT_m, \ldots, GT_n\}\ \}$ then

$$\begin{aligned} f_{ID} &:: GT_i \rightarrow GT_x \\ f_{ID} &:: GT_i \rightarrow GT_x+ \\ f_{ID} &:: GT_i \rightarrow \{GT_x\} \end{aligned}$$

are all member functions (where $a \rightarrow b$ is a many-to-one function, $a \rightarrow b+$ is a partial function and $a \rightarrow \{b\}$ is a many-to-many function or relation).

The prototype declares the types and relationships that exist for all instances of the cluster that it describes. This statement gives us an informal semantics for the prototype: The prototype states that, for any instance of the cluster, that instance will have the following features:

1. Types that relate to each of the type variables given in the prototype.

2. An expression of relationships declared as functions between objects of the member types of the cluster.

Note: the expression of the relationships may take several forms including predicates in which all arguments are explicit, predicates in which some arguments are implicit and totally implicit predicates (e.g. locations on a totally connected map that do not use an explicit predicate to show map connections).

However, the precise semantics of a prototype can only be given with respect to a domain in which the cluster described in the prototype is identified. This involves examining the relationship between the prototype and the domain level realisation of the cluster described by the prototype.

3.5.2 The functions of TIM

As has been stated, the identification of generic clusters through the recognition of their member types and interrelations is handled totally be TIM. That is not, however, TIM's only function in the use of generic cluster prototypes and control rules. The results of the generic type analysis not only identify the existence of generic types (and, by association, generic clusters) and their constituent features, they also provide the mapping from the abstraction to the domain instance of any given generic cluster identified. This should not come as a surprise, as to identify an instance of a generic cluster is precisely to identify those domain types that, through specific domain level predicates and operators, exhibit the appropriate behaviour.

The first task is to identify an instance of a generic cluster in a domain, from the universe of instances of that cluster. As has been discussed, this identification is carried out by TIM. The result of this identification process is the 'naming' of the instances of the cluster that are discovered in the domain. Let this identification function be known as $\mathcal{N}_D$, and let it be defined as

$$\mathcal{N}_D(C) = \mathcal{C} \text{ where } \mathcal{C} \text{ is the name of an instance of } C \text{ and } \mathcal{C} \in D$$

Note that in fact the function $\mathcal{N}_D$ may return a set of instances of the cluster C, as in

$$\mathcal{N}_D(C) = \{\mathcal{C} | \mathcal{C} \text{ is the name of an instance of } C \text{ and } \mathcal{C} \in D\}$$

representing the fact that multiple instances of a cluster may occur in a single domain. However, for the purposes of discussing the processes involved it will be assumed that $\mathcal{N}_D$ returns only one instance $\mathcal{C}$. For domains where $\mathcal{N}_D$ actually returns $\{\mathcal{C}_0, \ldots, \mathcal{C}_i\}$, the bindings of the prototype and likewise interpretations of control rules may be given for every instance $\mathcal{C}_x$.

The function $\mathcal{N}_D$ also gives us identifiers for the features of the cluster. Where a member type of a cluster C is given as C_T, the name of the type in an instance $\mathcal{C}$ of C will be given as $\mathcal{C}_T$. This preserves the naming convention of the member types in naming the concrete domain type representing the member type in an instance of C.

$$\mathcal{N}_D(C_T) = (\mathcal{N}_D(C))_T$$

The names of the functions that are attached to the member types of a cluster are also preserved, but a similar binding of the cluster to one of its instances allows the discussion of the manifestation of a function for a particular instance. This gives the manifestation of f_{C_T} for an instance $\mathcal{C}$ of C as $f_{\mathcal{C}_T}$.

$$\mathcal{N}_D(f_{\mathcal{C}_T}) = f_{(\mathcal{N}_D(C))_T}$$

The identifiers that the function $\mathcal{N}_D$ returns allow the instances of the cluster to be described and discussed. This will be shown to be important in the specialisation of an abstract control rule (see Section 3.9).

The second responsibility that TIM carries is to provide the system with the grounding function $\mathcal{F}$ that relates instances of a cluster to their concrete realisations in a domain. It is this function that allows the system access to the domain level specifics of the instance that are crucial to the power of the abstracted control rules. The grounding function gives concrete domain types as a realisation of the member types and propositional expressions in the language of the domain relating objects of those types as a realisation of the member types' functions.

$\mathcal{F}(\mathcal{C}_T) = \mathcal{T}$ where $\mathcal{T} \in \{\bigcup \mathcal{T}s | \mathcal{T}s \in \wp\{T_0, \ldots, T_i\}\}$,
$T_0, \ldots, T_i$ are primitive domain types and $\mathcal{T}$ is the instance of the type T within $\mathcal{C}$

$\mathcal{F}(f_{\mathcal{C}_T}) = \mathcal{P}[x, y]$ where $\mathcal{P}$ is the propositional expression with at most two free variables that is the manifestation of the generic relation captured by f in $\mathcal{C}$

$[x, y]$ represents the fact that x and y may be free in the expression $\mathcal{P}$. The decision to put an upper limit of two on the free variables in $\mathcal{P}$ is discussed in Section 4.5.

The responsibility of dealing with particulars of domain expressions of functions falls entirely on the function $\mathcal{F}$, including implicit arguments and even implicit predicates. The case of implicit arguments is covered by the expression $\mathcal{P}$ using domain level predicates with implicit arguments. For example, the gripper domain has an implicit argument to the predicate describing the location of robby the robot. The locatedness predicate of robby is *at–robby*(x), where x is robby's location (this is in contrast to a classic logistics domain, in which the locatedness predicate takes two arguments, the object and that object's location).

The case of implicit predicates is more interesting, but easily managed. In the case where the realisation of the relation f_{C_t} in a domain is an implicit predicate, $\mathcal{F}$ returns $TRUE$, as in $\mathcal{F}(x.f_{C_T}) = TRUE$. This point necessitates the consideration of truth values in the language and functions discussed in Sections 3.8 and 3.9.

In light of the two functions that are required from TIM, it is now possible to show the semantics of the prototype with respect to a domain D. The semantics are given as a set of interpretation rules, where $I_D[\![a]\!]$ represents the interpretation of a in D.

$$\begin{aligned} I_D[\![C]\!] &= \mathcal{F}(\mathcal{N}_D(C)) \\ I_D[\![C_T]\!] &= \mathcal{F}(\mathcal{N}_D(C_T)) \\ I_D[\![f_{C_T}]\!] &= \mathcal{F}(\mathcal{N}_D(f_{C_T})) \end{aligned}$$

3.5.3 An example prototype and its features

This example prototype is for the *SafePortable* generic cluster.

```
prototype SafePortableCluster {
  Types:        {SafePortable, Carrier, Location}
  Functions :   at::SafePortable→Location+
                at'::Carrier→Location
                in::SafePortable→Carrier+
                locationOf::Location→{SafePortable,Carrier}
                link::Location→{Location} }
```

This cluster prototype tell us that there are three types in the cluster, *SafePortables*, *Carriers* and *Locations*. Objects of type *SafePortable* have two functions available to them, the *at* and *in* functions, which return a possible *Location* object and a possible *Carrier* object respectively. The fact that *SafePortables* are either located at some location *or* being carried by some carrier necessitates the inclusion of the *NULL* value in the return type. Objects of type *Carrier* have one function available to them, namely the *at'* function. This function returns a single-valued variable of type *Location*. Objects of type *Location* have two functions available to them, namely *locationOf* and *link*. Both of these functions return set-valued variables, though *locationOf* returns a set of objects of type $Carrier \cup SafePortable$ while *link* returns a set of objects of type *Location*. It might be supposed that the *locationOf* function should have the option to return the *NULL* value, for the cases where there is indeed no objects located at a particular location. However, because this function returns a set of values, it simply returns the empty set in this situation.

Although the names of the member functions of objects (of certain generic type) are somewhat arbitrary (in the example above they are in fact suggestive of an intuitive interpretation of the generic types involved), it must be remembered that they refer to other objects through specific and *identifiable* relationships that, by the type's identification as an instance of a certain GT, the object necessarily has. The decision to name two of the functions in the above example *at* and at' reflects the fact that they both represent the same semantic relationship (i.e. that of being associated with a location object). It would in theory be possible to overload functions, under the condition that no two functions with the same type signature share a name. The overloading would be resolved by the type of the object calling the function and the function term's context (i.e. the types of the terms in the predicate using the function term).

Member function notation

The decision to give the relations as member functions of the types in the cluster was made for several reasons. What we are trying to achieve through the use of functions is the identification of an object (or set of objects) *by their relation to some known object.* With this in mind, any domain level realisation of that relation will either be a propositional expression in which the base object of the function term is required as an argument, or a dedicated propositional expression in which the base object is an implicit argument. Although it is true that the expression showing the relation is not dependent on the base object of the function term, the function expresses a relationship that is inherent to that object (as a direct result of its type). We adopt the syntax of a member function as if the base object were an object in a class in an object-oriented language to reflect that the relationships are dependent on the base object.

The member function notation not only allows us to abstract away from the names of the predicates that relate objects, but also provides us with a convenient way in which to abstract the structure of those predicates. We simply say that objects are related through the function and leave the details of the manifestation of that relation to be resolved when the function is interpreted with respect to the specifics of the domain level instance of the cluster.

3.5.4 Linking prototypes to generic clusters

A problem exists when we try to formalise the classification of generic types, and by association, generic clusters. The inclusion of a type into a generic type is based on a description of the FSMs for that type, though there is no language currently presented for that description in a formal sense. Currently, the identification process is hard coded implicitly in TIM (the type inference module that also handles generic type identification).

Although there is no language currently presented for the identification of generic clusters, such a language would require certain features that can be outlined. In the same way that generic types are identified by the fingerprints of the finite state machines associated with the type, so the a generic cluster is identified by the fingerprints of its member types.

The language would need to supply constructs to allow generic clusters to be described in terms of their constituent patterns (FSMs and other structural elements). This would involve a categorisation of the states involved as single properties, exchanged properties, etc. and of the transitions as self-connecting, connecting exchanged properties, etc..

The generic type identification machinery provided by TIM is able to identify interacting sets of generic types by the fingerprints associated with the finite state machines for each type. The properties involved in the fingerprints of related types refer to the domain level expression of the relation of the types involved.

In order to formally define generic clusters, the language would need to describe the relation of the properties of the fingerprints for each of the member types. The preconditions of operators in the domain can also be involved in the specification of features of associated types (e.g. the map of locations for a mobile type is inferred using the preconditions of the *move* operator). The language would as such need the capability of referring to operators and specifically identifying the predicates that appear in the preconditions according to conditions on the arguments of those predicates.

A useful extension to the work covered in this thesis would be to explore the definition of a language to formally classify and identify generic types and clusters.

3.6 The language of control rules

3.6.1 The need for a new language

In order to represent control knowledge at a level abstracted from any specific domain, two components are required. Firstly the abstract structures are needed in terms of which the control information will be formulated. Generic clusters have been presented as such structures, with their prototypes giving information about how they can be used to describe objects and types that instantiate them. So it remains to specify the language that makes use of generic clusters to formulate control information.

3.6.2 Abstract syntax

The logic used for expressing generic control rules (GCRs) is an abstract logic. That is, expressions have no evaluation in isolation. In order to evaluate an expression it must first be specialised with an instance of a generic cluster. The expression can then be evaluated with respect to a plan trace in the domain that provides the instance of the generic cluster.

Generic control rule logic (GCRL) is an extension of a first order predicate logic. The predicates are equality ($==$) and set membership ($\in$). The only terms of the predicates are variables, representing objects or sets of objects. Terms can be used explicitly or denoted by a function applied to an object variable (according to the functions available for objects of that type).

The set of formulas in GCRL is given by the following rules:

1. If p and p' are object terms and q is a set term, $p == p'$ and $p \in q$ are logical formulae

2. If P and Q are logical formulae, then $P \wedge Q$ and $\neg P$ are logical formulae

3. If t is an object variable, C_m is a type variable where m is a member type in the prototype for cluster C and P is a logical formula then $\forall t\ :\ C_m\ P$ is a logical formula and $\exists t\ :\ C_m\ p$ is a logical formula

4. If C is a generic cluster ID and p is a logical formula then $\forall\ C\ p$ is a formula

Standard abbreviations are employed, specifically $p \vee q$ abbreviates $\neg(\neg p \wedge \neg q)$ and $p \rightarrow q$ abbreviates $\neg p \vee q$. Note that the $\forall$ quantifier is overloaded. It is used to quantify over objects according to type (given as a type variable) and also to quantify over all instances of a generic cluster.

The set of terms is given by the following rules:

1. If x is a simple object variable of type C_m, then x is a term

2. If x is a simple object variable of type C_m, f is a function defined for C_m in the prototype of the cluster C, and S is a state term, then $x.f(S)$ is a term (also called a function term). The term is either a set or object term, according to the type of f in the prototype.

A state term is either a *temporal state* or a *special state.* The set of temporal state terms S_t is given by the following rules:

1. The constant symbol NOW

2. If s is a temporal state term then NEXT(s) is a temporal state term

NEXT abbreviates NEXT(NOW) which provides, we feel, a more intuitive manner to describe successor state terms and nested successor states. For example, the state term representing the successor of the state term NOW is simply NEXT, and the successor of NEXT is given as NEXT(NEXT).

The set of special state terms S_s contains only the constant symbol, GOAL.

The set of state terms, S, is given by the formula

$$S = S_t \cup S_s$$

Objects are structured (according to type), as different functions are available for objects of different types. The abstract syntax given above is intended to define a family of logics, whose individual members are parameterised by the generic cluster prototypes used in any given expression. The logical syntax of all the family members remains the same, but the constants that are the names of member types of the generic clusters and the names of the functions available for use in function terms by each member type are cluster-specific. As such it is the conjunction of the abstract syntax and the prototypes of the clusters involved that specify whether or not a particular expression is well-formed.

3.6.3 Semantics

The semantics of the language is given by the interpretation of the language constructs involved, as given in Section 3.6.2. The interpretation can only be given with respect to a domain, D, and to a sequence of states, $SqSts$, $< s_0, \ldots, s_n >$ and a goal state, G. The domain must contain the structure that is used in the expression being interpreted, i.e. the appropriate generic cluster. The interpretation of a formula f in domain D will be given by $I_D[\![f]\!](SqSts, G)$.

At the heart of the interpretation process is the interpretation of the language's terms. Terms can either be simple object terms or function terms. Simple objects can only represent single objects, but function terms can represent either single objects or sets of objects (according to the type of the function given in the prototype). The interpretation of simple objects terms is just the object that they represent,

$$I_D[\![X]\!](SqSts, G) = X$$

(N.B. Simple object terms will always be object variables, and well formed sentences will only use object variables that have been appropriately quantified. A simple object term will only ever interpreted within the scope of the quantification of the object variable involved.)

The interpretation of function terms where a function f describes the relation of objects through some generic relationship, which is expressed through some propositional expression in the language of the domain D in which the rule is being interpreted, is given as

$$I_D[\![X.f_{\mathcal{C}_m}(S)]\!](SqSts, G) = Y \text{ where } I_D[\![S]\!](SqSts, G) \models \mathcal{F}(f_{\mathcal{C}_m})[x/X, y/Y]$$

for f as a single-valued function and

$$I_D[\![X.f_{\mathcal{C}_m}(S)]\!](SqSts, G) = \{Y | I_D[\![S]\!](SqSts, G) \models \mathcal{F}(f_{\mathcal{C}_m})[x/X, y/Y]\}$$

for f as a set-valued function ($\models$ is used here as standard entailment of an expression by a planning domain state). In the case where f represents a function describing an exchangeable property (Section 3.7) of type $\mathcal{C}_m$, f may return a $NULL$ value according to the following rule

$$\begin{aligned} I_D[\![X.f_{\mathcal{C}_m}(S)]\!](SqSts, G) = NULL & \text{ where there does not exist any } Y \\ & \text{such that } I_D[\![S]\!](SqSts, G) \models \mathcal{F}(f_{\mathcal{C}_m})[x/X, y/Y] \end{aligned}$$

Notice that the interpretation of the function term involves the grounding function $\mathcal{F}$ that gives the domain level realisation of the function $f_{\mathcal{C}_m}$ according to the instance $\mathcal{C}$ of its prototype as well as the logical entailment of the resulting propositional expression with respect to the interpretation of the state term and the sequence of states. The term $\mathcal{F}(f_{\mathcal{C}_m})$ is the propositional expression with at most two free variables and the substitution $[x/X, y/Y]$ replaces all occurrences of x with X and y with Y.

The interpretation of state terms gives the state in which the propositions involved in interpreting function terms should be evaluated.

$$
\begin{aligned}
I_D[\![\text{NOW}]\!](SqSts, G) &= s_0 \\
I_D[\![\text{NEXT}(S)]\!](SqSts, G) &= s_{i+1} \text{ where } s_i = I_D[\![S]\!](SqSts, G) \\
I_D[\![\text{GOAL}]\!](SqSts, G) &= G
\end{aligned}
$$

The interpretation function for the remainder of the language is given by

$$
\begin{aligned}
I_D[\![X == Y]\!](SqSts, G) &= TRUE \\
&\quad \text{iff } I_D[\![X]\!](SqSts, G) \text{ equals } I_D[\![Y]\!](SqSts, G) \\
I_D[\![X \in Y]\!](SqSts, G) &= TRUE \\
&\quad \text{iff } I_D[\![X]\!](SqSts, G) \text{ is in } I_D[\![Y]\!](SqSts, G) \\
I_D[\![X \wedge Y]\!](SqSts, G) &= TRUE \\
&\quad \text{iff } I_D[\![X]\!](SqSts, G) \text{ and } I_D[\![Y]\!](SqSts, G) \\
I_D[\![\neg X]\!](SqSts, G) &= TRUE \\
&\quad \text{iff it is not the case that } I_D[\![X]\!](SqSts, G) \\
I_D[\![\forall_{object} X : \mathcal{C}_m \,.\, P]\!](SqSts, G) &= TRUE \\
&\quad \text{iff } I_D[\![P[X/t]]\!](SqSts, G) \text{ is } TRUE \text{ for all } t \in \mathcal{F}(\mathcal{C}_m) \\
I_D[\![\exists_{object} X : \mathcal{C}_m \,.\, P]\!](SqSts, G) &= TRUE \\
&\quad \text{iff } I_D[\![P[X/t]]\!](SqSts, G) \text{ is } TRUE \text{ for some } t \in \mathcal{F}(\mathcal{C}_m) \\
I_D[\![\forall_{cluster} C \,.\, P]\!](SqSts, G) &= TRUE \\
&\quad \text{iff } I_D[\![P[C/\mathcal{C}]]\!](SqSts, G) \text{ is } TRUE \text{ for every } \mathcal{C} \in \mathcal{N}_D(C)
\end{aligned}
$$

The semantics of the language is quite standard for a predicate logic, though there are several points to note. Firstly, the interpretation of the terms and specifically the function terms gives the language the power to express relationships between objects *in specified states* given by their state arguments. The overloaded $\forall$ operator allows not just quantification over types but also over instances of a generic cluster and as a result must be interpreted in two different ways. As with the syntax, the logical abbreviations $\vee$ and $\rightarrow$ are accepted. $NULL$ can be seen as a special constant value. It behaves just like a constant value in equalities, that is, it is equal to only itself.

3.7 Prototypes Detail

Interestingly, both properties and attributes are used as the basis of functions given in the prototypes to relate objects within a generic cluster. This is in contrast to the use of attribute and property spaces in invariant generation within TIM, where only the property spaces are used. Functions describe objects or sets of objects that are related to an object x through predicates. Where an object x is of certain generic type and has some generic behaviour given as a property describing the predicate and argument position within that predicate, it will have a function which describes the objects with properties relating to the other argument positions in that predicate.

The characteristics of the functions that are used to relate objects within generic clusters (c.f. Section 3.2.3) can be traced back to the PRSs (property relating structures) that are used in invariant generation (c.f. Section 3.3). The basis of the identification of identity invariants, namely the single occurrence of properties in states given by the PRS, determines whether a function will be single-valued or set-valued. If property P_i caused the identification of a identity invariant then the functions that relate some object x with property P_i to other objects through P will be single valued functions. For any other property P_j that did not cause the identification of a identity invariant, the functions that relate objects x with property P_j to other objects through P will be set-valued functions.

Where x's property P_i was based on a FSM with a singleton state, functions based on that property will always be defined. The singleton state tells us that objects that have the property P_i *always* have the property P_i, and hence will always be related to the other objects present in the predicate P. If in addition to this, the property P_i caused the identification of an identity invariant, functions based on the property P_i will always be single-valued objects.

However, where property P_i is an exchangeable property, i.e. a property that is exchanged for another in the FSM, the functions based on P_i may return a $NULL$ value. This represents the fact that in any given state, objects that *can* have property P_i *may not have that property currently.* For example, portable objects have the exchangeable properties of being either located at some location or contained in some carrier. Functions that describe the location of the portable will return $NULL$ if the portable is currently being carried and conversely functions to describe the carrier will return $NULL$ if the portable is not being carried. If the property P_i caused the identification of an identity invariant, functions based on the property P_i will always be single-valued, but the $NULL$ value may be returned. This effectively adds the special object $NULL$ to the set of objects that may be returned by the function. Within a set of exchangeable properties $\{P_k, \ldots, P_l\}$, $\exists P_m \cdot P_m \neq NULL$.

Functions based on any properties that did not cause an identity invariant to be generated or attributes will always return a set of objects. This may be the empty set if the object has no occurrences of the particular attribute in the required state (in the case of a function based on an attribute), or if the property does not hold at all for that object in the state being considered (it the case where the function is based on an exchangeable property of the object). For example, the attribute of a location as having some mobile or portable located at it may return an empty set if there are no objects located there. Note that the only case where property based functions can return a set of objects is in the case where an object can have multiple occurrences or a particular property in a state. In principle this allows the size of the set of objects described to be limited to the number of occurrences of the particular property for the type.

3.8 Domain level language

3.8.1 Abstract syntax

The domain level language used is a version of linear temporal logic presented by Emerson [18]. The base language is a first-order language, whose elements can describe a single state in a planning domain. The language includes constants from the universe of planning domain objects and predicates as well as *TRUE* and *FALSE*. Objects can also be represented by typed variables. Propositions in the language are predicates, populated with constants or object variables. Propositions are formulae in the language, as are propositions joined by standard logical connectives. The quantification of object variables is by type. The types identified by TIM are given as sets of domain objects. These may be named to avoid listing large sets of objects in expressions.

To the base language is added the temporal modality $\bigcirc$ (next). The base language is extended such that if p is a temporal proposition, then $\bigcirc p$ is a temporal proposition (all unadorned propositions are also temporal propositions). The intuitive interpretation of $\bigcirc p$ is that p holds in the sequence of states succeeding the state in which $\bigcirc p$ is asserted.

The modal operator GOAL is used to assert the inclusion of a literal in the problem-specific goal set. The base language is extended such that if P is a formula containing no temporal modal operators, then GOAL P is a formula. We call this a *weak* GOAL modality, as it does not bind tightly to a single proposition but instead can be used to qualify any atemporal formula. A *strong* GOAL modality would be such that if p is a proposition, then GOAL p is a formula. This decision is discussed further in Sections 3.8.3 and 3.10.1.

The set of formulae in the domain level language is given by the following rules:

1. If p is a predicate of i arguments in $\mathcal{P}$, the universe of domain predicates, and $\{x_1, \ldots, x_i\}$ are object variables from $\mathcal{O}$, the universe of domain objects, or object variables then $p(x_1, \ldots, x_i)$ is an atomic formula, as are the variables *TRUE* and *FALSE*. Atomic formulae are also temporal formulae.

2. If P and Q are formulae then $P \wedge Q$ and $\neg P$ are formulae.

3. If P is a temporal formula then $\bigcirc P$ is a temporal formula.

4. If P is a formula containing no temporal modalities then GOAL P is a formula.

5. If P is a formula, x is an object variable and O is an object type then $\forall_{object} x : O \,.\, P$ is a formula.

$\vee$ and $\rightarrow$ are accepted as abbreviations under the standard definitions.

3.8.2 Semantics

In order to evaluate formulae in the domain level language with respect to a plan execution trace, the semantics of the language must be given. A basic standard semantics, such as that given by Emerson [18], is described by the following interpretation rules over the language specified in Section 3.8.1. Let $SqSts$ be an infinite sequence of world states $\{s_0, s_1, \ldots\}$, G be the problem specific goal set, P be an unadorned proposition, f and g be formulae in the extended language and O be a set of domain constants.

$$\begin{array}{lll}
(SqSts, G) & \models_D & TRUE = TRUE \\
(SqSts, G) & \models_D & FALSE = FALSE \\
(SqSts, G) & \models_D & P \text{ iff } s_0 \models P \\
(SqSts, G) & \models_D & f \wedge g \text{ iff } (SqSts, G) \models_D f \text{ and } (SqSts, G) \models_D g \\
(SqSts, G) & \models_D & \neg f \text{ iff it is not the case that } (SqSts, G) \models_D f \\
(SqSts, G) & \models_D & \forall\, x \;:\; O \;.\; f \text{ iff } (SqSts, G) \models_D f[x/o] \text{ for all } o \in O \\
(SqSts, G) & \models_D & \bigcirc f \text{ iff } (tail(SqSts), G) \models_D f \\
(SqSts, G) & \models_D & \text{GOAL } f \text{ iff } G \models f
\end{array}$$

The function *tail* is defined in a standard manner as

$$tail(x : xs) = xs \tag{3.2}$$

The semantics use a modified version of entailment, namely $\models_D$. This is to reflect the fact that instead of standard entailment of an expression by a single state we are dealing with a tuple composed of a sequence of states (the plan trajectory) and a single state (the goal set). $\models_D$ does use standard entailment on single states, the particular state being dictated by the mode in which the expression is qualified. As with the syntax, the logical abbreviations $\vee$ and $\rightarrow$ are accepted.

3.8.3 Strong versus weak interpretation of the goal modality

There are two possible ways in which we can interpret the GOAL modality in the context of the domain entailment. The first option is to allow only literals to be adorned with the GOAL modal operator, i.e. a strong GOAL. This is dictated by the syntax of the language but would manifest itself in the semantics as

$$(SqSts, G) \models_D GOAL\ P \text{ iff } P \in G$$

The alternative is to use a weak GOAL, i.e. the modality can qualify any atemporal expression. This would manifest itself in the semantics as

$$(SqSts, G) \models_D GOAL\ P \text{ iff } G \models P$$

Notice that the entailment used in the interpretation of the weak GOAL would become simple set membership under a strong GOAL modality, since

$$G \models P \text{ where P is a single proposition iff } P \in G$$

Aside from the fact that the use of a strong GOAL gives us a subset of the language provided if a weak GOAL is used, the decision has an impact on the proof contained in Section 3.10. Further discussion on the GOAL modality can be found in Section 3.10.1.

Domain level expressions are obtained from abstract level expressions through the instantiation process. Instantiation provides us with the domain-specific versions of the abstract expressions. Figure 3.4 shows how the languages relate to each other through the processes of interpretation, instantiation and evaluation.

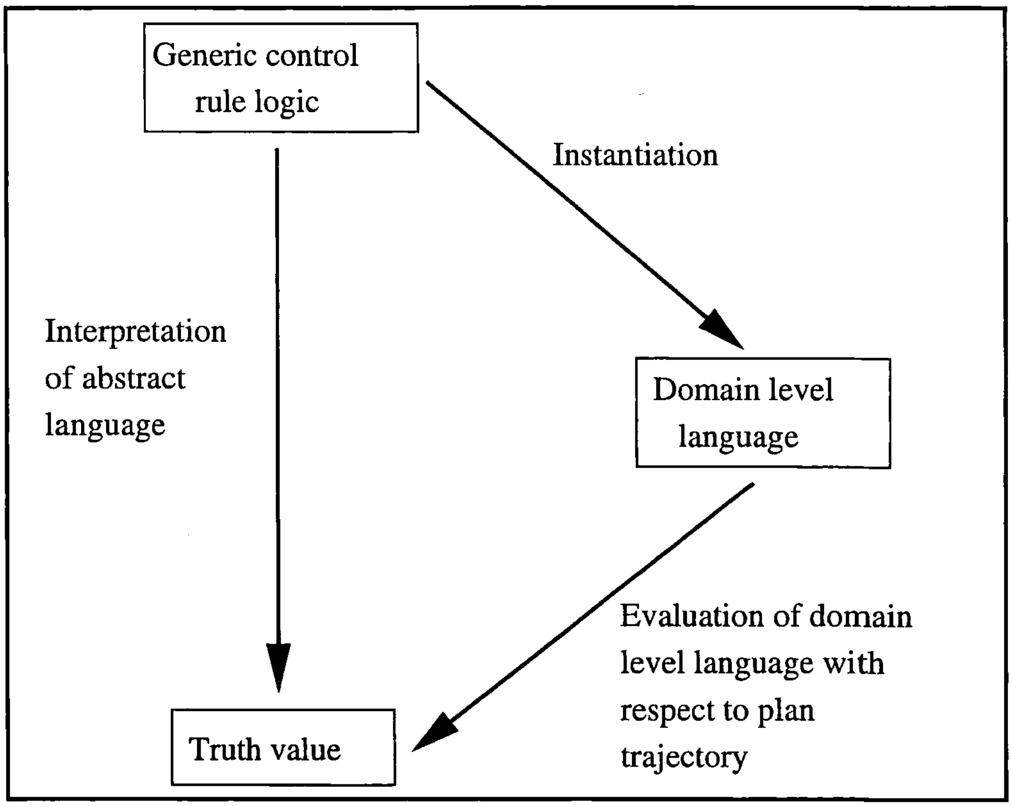
Generic control rule logic
Instantiation
Interpretation of abstract language
Domain level language
Evaluation of domain level language with respect to plan trajectory
Truth value

Figure 3.4: Language relation

3.9 Instantiation

Instantiation is the process by which an abstract control rule is specialised and results in a domain-specific control rule. This gives the rule in a form that can be evaluated in a concrete domain according to the semantics of the domain level language. As a result, instantiation is with respect to an instance of the cluster that occurs in that concrete domain.

The process of instantiation is achieved using the two functions supplied by TIM to give the realisation of the types and functions involved in the expression while preserving the logical structure. The instantiation procedure is defined over the language specified in Section 3.6.2.

$$inst_D[\![\forall_{cluster}\ C\ .\ B]\!] \ = \ \bigwedge inst_D[\![B[\mathcal{C}/C]]\!] \text{ for all } \mathcal{C} \in \mathcal{N}_D(C)$$

$$\begin{aligned}
inst_D[\![\ \forall_{object} X\ :\ \mathcal{C}_T\ .\ B]\!] &= \forall X : \mathcal{F}(\mathcal{C}_T).inst_D[\![B]\!] \\
inst_D[\![\exists_{object} X\ :\ \mathcal{C}_T\ .\ B]\!] &= \exists X : \mathcal{F}(\mathcal{C}_T).inst_D[\![B]\!] \\
inst_D[\![A \wedge B]\!] &= (inst_D[\![A]\!]) \wedge (inst_D[\![B]\!]) \\
inst_D[\![\neg B]\!] &= \neg(inst_D[\![B]\!]) \\
inst_D[\![X.f_{\mathcal{C}_T}(S) == Y]\!] &= inst_D[\![S\ (\mathcal{F}(f_{\mathcal{C}_T})[x/X, y/Y])]\!] \\
& \quad \text{for equalities using a function term} \\
inst_D[\![X == Y]\!] &= X == Y \\
& \quad \text{where } X \text{ and } Y \text{ are constants} \\
inst_D[\![Y \in X.f_{\mathcal{C}_T}(S)]\!] &= inst_D[\![S\ (\mathcal{F}(f_{\mathcal{C}_T})[x/X, y/Y])]\!]
\end{aligned}$$

(again, the auxiliary logical operators $\rightarrow$ and $\vee$ are accepted, being defined by the $\neg$ and $\wedge$ operators)

The construction of modal operators is also handled by the instantiation function:

$$\begin{aligned}
inst_D[\![\text{NOW}\ \mathcal{P}]\!] &= \mathcal{P} \\
inst_D[\![\text{SUCC(S)}\ \mathcal{P}]\!] &= \bigcirc(inst_D[\![S\ \mathcal{P}]\!]) \\
inst_D[\![\text{GOAL}\ \mathcal{P}]\!] &= GOAL\ \mathcal{P}
\end{aligned}$$

Notice that the $\forall_{cluster}$ instantiates a conjunction of the instantiations of its body for all instances of the cluster found in the domain. As mentioned in Section 3.5.2, we will consider only examples where a cluster appears at most once in a domain. This decision affects only the demonstration of the instantiation process by example, not the process itself which is robust to multiple instances.

Let us consider an example, through the instantiation of the control strategy of never moving a safe portable from its goal location. This is expressed as a GCR as

$$\forall_{cluster} C : SafePortableCluster\ \forall_{object} X : C_{SafePortable}\ \forall_{object} Y : C_{Location}$$

$$X.at_{C_{SafePortable}}(\text{NOW}) = Y \wedge X.at_{C_{SafePortable}}(\text{GOAL}) = Y$$
$$\rightarrow X.at_{C_{SafePortable}}(\text{NEXT}) = Y \tag{3.3}$$

Given a domain $\mathcal{D}$ in which TIM has identified an instance $\mathcal{C}$ of the safe portable cluster with the following types.

$$\begin{aligned} \mathcal{F}(\mathcal{C}_{SafePortable}) &= \{obj1, obj2, obj3\} \\ \mathcal{F}(\mathcal{C}_{Location}) &= \{loc1, loc2\} \end{aligned}$$

TIM also provides the binding function that tells us how to interpret the function terms used

$$\mathcal{F}(at_{C_{SafePortable}}) = sited(x : \mathcal{C}_{SafePortable}, y : \mathcal{C}_{Location})[x, y]$$

The first stage of instantiation is to apply the domain specific $inst_{\mathcal{D}}$ function to the expression 3.3. This yields

$$\bigwedge inst_D[\![\forall_{object} X : \mathcal{C}_{SafePortable}\ \forall_{object} Y : \mathcal{C}_{Location}$$
$$X.at_{C_{SafePortable}}(\text{NOW}) = Y \wedge X.at_{C_{SafePortable}}(\text{GOAL}) = Y$$
$$\rightarrow X.at_{C_{SafePortable}}(\text{NEXT}) = Y]\!] \text{ for all } \mathcal{C} \text{ in } \mathcal{N}_D(C)$$

Assuming there is only one unique $\mathcal{C}$ in $\mathcal{N}_D(C)$, the outer conjunction becomes a single term as in

$$inst_D[\![\forall_{object} X : \mathcal{C}_{SafePortable}\ \forall_{object} Y : \mathcal{C}_{Location}$$
$$X.at_{C_{SafePortable}}(\text{NOW}) = Y \wedge X.at_{C_{SafePortable}}(\text{GOAL}) = Y$$
$$\rightarrow X.at_{C_{SafePortable}}(\text{NEXT}) = Y]\!]$$

giving

$$\begin{aligned}&\forall_{object} X : \mathcal{F}(\mathcal{C}_{SafePortable})\ \forall_{object} Y : \mathcal{F}(\mathcal{C}_{Location})\\ &\quad inst_D[\![X.at_{\mathcal{C}_{SafePortable}}(\text{NOW}) = Y \wedge X.at_{\mathcal{C}_{SafePortable}}(\text{GOAL}) = Y\\ &\quad \rightarrow X.at_{\mathcal{C}_{SafePortable}}(\text{NEXT}) = Y]\!]\end{aligned}$$

Inside the body of the expression, the logical structure is preserved and $inst_D$ is propagated to the terms.

$$\begin{aligned}&\forall_{object} X : \mathcal{F}(\mathcal{C}_{SafePortable})\ \forall_{object} Y : \mathcal{F}(\mathcal{C}_{Location}\\ &\quad inst_D[\![X.at_{\mathcal{C}_{SafePortable}}(\text{NOW}) = Y]\!] \wedge inst_D[\![X.at_{\mathcal{C}_{SafePortable}}(\text{GOAL}) = Y]\!]\\ &\quad \rightarrow inst_D[\![X.at_{\mathcal{C}_{SafePortable}}(\text{NEXT}) = Y]\!]\end{aligned}$$

The grounding function, $\mathcal{F}$, that TIM provides alongside the identification of the instance of the cluster gives us the propositional expression representing the function for the type $\mathcal{C}_{SafePortable}$ (this may be a truth value as discussed in 3.5.2). According to $inst_{\mathcal{D}}$, the base object of the function term is given as the first argument to the propositional expression and the other argument to the predicate (equality or set membership) is given as the second argument. The first equality in the example is instantiated as

$$\begin{aligned}&inst_D[\![(X.at_{\mathcal{C}_{SafePortable}}(\text{NOW}) == Y)]\!]\\ &\qquad = \quad inst_D[\![\text{NOW}\ \mathcal{F}(at_{\mathcal{C}_{SafePortable}})[x/X, y/Y]]\!]\\ &\qquad = \quad sited(x : \mathcal{C}_{SafePortable}, y : \mathcal{C}_{Location})[x/X, y/Y]\\ &\qquad = \quad sited(X, Y)\end{aligned}$$

After all the function terms are instantiated, the expression becomes

$$\begin{aligned}&\forall_{object} X : \mathcal{F}(\mathcal{C}_{SafePortable})\ \forall Y : \mathcal{F}(\mathcal{C}_{Location})\\ &\quad sited(X, Y) \wedge GOAL\ sited(X, Y) \rightarrow NEXT\ sited(X, Y)\end{aligned}$$

An expression that is fully instantiated can then be evaluated according to the interpretation function for the domain level language (Section 3.8), with respect to a plan trajectory in the concrete domain. This yields a truth value that represents whether or not the plan trajectory adheres to the control rule.

The process of instantiation is very similar to the interpretation of abstract level expressions, but does not expand quantifications or check the entailment of the resulting expression (it having no plan trajectory against which to be evaluated). The interpretation of the abstract level language provides its semantics, but in the implementation of this work control rules are instantiated before they are evaluated.

3.10 Proof

The processes of interpretation of an expression in the abstract language and of instantiation followed by interpretation of the resulting domain level expression can be shown to be equivalent. Figure 3.5 illustrates the equivalence to be proved. The proof is reminiscent in structure of proofs showing the equivalence of normal forms.

The proof will follow a basic inductive proof on the structure of the language (whose grammar is given in Section 3.6.2), where we show the equivalence for the base cases and then extend that equivalence for each of the step cases. The base cases will show the equivalence of the primitive predicates, and the step cases will be the language constructs that form expressions from those predicates. The nature of the state argument means that an auxiliary inductive proof must be performed to demonstrate the equivalence of the primitive predicates.

$I_D[\![f]\!](SqSts, G)$ represents the interpretation of a GCRL expression with respect to a plan execution trace of goal condition G and states $SqSts = \{s_0, s_1, s_2, \ldots\}$. $inst_D[\![f]\!]$ gives the domain instantiation of f, and $(SqSts, G) \models_D g$ is the domain entailment of an expression g in the domain level language (see 3.8).

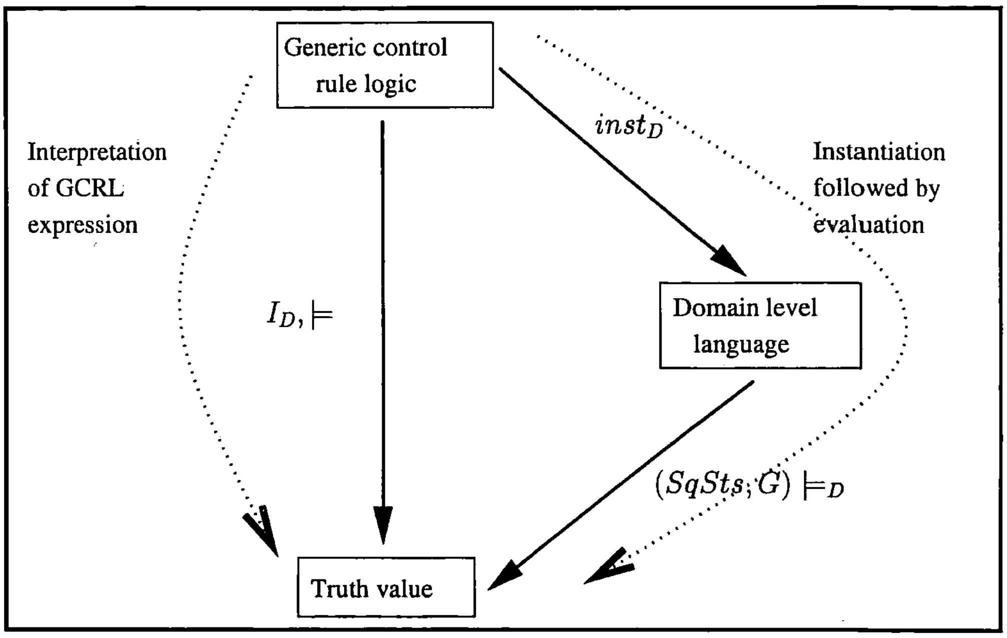
Generic control rule logic
Interpretation of GCRL expression
$inst_D$
Instantiation followed by evaluation
$I_D, \models$
Domain level language
$(SqSts, G) \models_D$
Truth value

Figure 3.5: Evaluation approaches for abstract level expressions

The proof requires the following lemma.

Lemma 1.

$$\forall \text{ states S}, \forall\, i$$

$$\begin{aligned} \text{If } S &= \textsc{next}(S') \\ \text{and } I_D[\![S']\!](SqSts, G) &= s_i \\ \text{then } I_D[\![S']\!](tail(SqSts), G) &= s_{i+1} \end{aligned}$$

This lemma is proved by induction on the structure of the state argument S'. The base case considers the state argument "NOW".

Base Case

$$\begin{aligned} S' &= \textsc{now} \\ I_D[\![\textsc{now}]\!](SqSts, G) &= s_0 \end{aligned}$$

The definition of the function *tail* 3.2 allows us to prove the base case, given $SqSts = \{s_0, s_1, s_2, s_3, \ldots\}$.

$$\begin{aligned} tail(SqSts) &= \{s_1, s_2, s_3, \ldots\} \\ I_D[\![\text{NOW}]\!](tail(SqSts), G) &= s_1 \end{aligned}$$

Inductive case

$$S' = \text{NEXT}(S'')$$

Inductive hypothesis

$$\forall S' \forall i$$

$$\begin{aligned} \text{If } S' &= \text{NEXT}(S'') \\ \text{and } I_D[\![S'']\!](SqSts, G) &= s_i \\ \text{then } I_D[\![S'']\!](tail(SqSts), G) &= s_{i+1} \end{aligned}$$

Assuming $I_D[\![S']\!](SqSts, G) = S_i \equiv I_D[\![\text{NEXT}(S'')]\!](SqSts, G) = s_i$

$$\begin{aligned} I_D[\![S'']\!](SqSts, G) &= s_{i-1} \\ &\quad \text{(Definition of } I_D \text{)} \\ I_D[\![S'']\!](tail(SqSts), G) &= s_i \\ &\quad \text{(Inductive hypothesis)} \\ I_D[\![\text{NEXT}(S'')]\!](tail(SqSts), G) &= s_{i+1} \end{aligned}$$

$\square$

The proof obligation we are undertaking is to show the equivalence of the interpretation of an expression in GCRL and the result of instantiation followed by the entailment of the domain level expression created can be stated as

$$\begin{array}{rcl} I_D[\![f]\!](SqSts, G) & = & TRUE \\ & \text{iff} & \\ (SqSts, G) & \models_D & inst_D[\![f]\!] \end{array}$$

There are two primitive predicates in GCRL, namely equality and set membership. These will be dealt with first.

Base case 1 (Equality) Atoms are propositions of the form

$$X.f_{C_T}(S) \quad == \quad Y$$

From the syntax of GCRL, Y *must* be a simple object (rather than a function term).

$$\begin{array}{rcl} I_D[\![X.f_{C_T}(S) == Y]\!](SqSts, G) & = & I_D[\![X.f_{C_T}(S)]\!](SqSts, G) \equiv Y \\ & = & Z \equiv Y \\ & & \text{where} \\ I_D[\![S]\!](SqSts, G) & \models & \mathcal{F}(f_{C_T})[x/X, y/Z] \end{array}$$

(remembering that $\mathcal{F}(f_{C_T})$is the propositional expression with at most two free variables that provides the domain level manifestation of the relationship represented by the function (f_{C_T}). Also note that there must only be one unique Z that satisfies the entailment.)

$$\begin{array}{rcl} & = & TRUE \\ & \text{iff} & \\ I_D[\![S]\!](SqSts, G) & \models & \mathcal{F}(f_{C_T})[x/X, y/Y] \\ & & \text{(by the equivalence of } Y \text{ and } Z\text{)} \end{array}$$

We must now show that this dependency holds for the three cases of the state argument S, namely $S = \text{NOW}$, $S = \text{GOAL}$ and $S = \text{NEXT}(S')$.

Case $S = \text{NOW}$

$$\begin{array}{rcl}
I_D[\![\text{NOW}]\!](SqSts, G) & \models & \mathcal{F}(f_{C_T})[x/X, y/Y] \\
s_o & \models & \mathcal{F}(f_{C_T})[x/X, y/Y] \\
 & & \text{(definition of } I_D\text{)} \\
 & \text{iff} & \\
(SqSts, G) & \models_D & \mathcal{F}(f_{C_T})[x/X, y/Y] \\
 & & \text{(since } (SqSts, G) \models_D P \leftrightarrow s_0 \models P\text{)} \\
 & \text{iff} & \\
(SqSts, G) & \models_D & inst_D[\![\text{NOW}\mathcal{F}(f_{C_T})[x/X, y/Y]]\!] \\
 & & \text{(since } inst_D[\![\text{NOW}P]\!] = P\text{)}
\end{array}$$

Case $S = \text{GOAL}$, where $\mathcal{F}(f_{C_T})[x/X, y/Y]$ is a literal.

$$\begin{array}{rcl}
I_D[\![\text{GOAL}]\!](SqSts, G) & \models & \mathcal{F}(f_{C_T})[x/X, y/Y] \\
G & \models & \mathcal{F}(f_{C_T})[x/X, y/Y] \\
 & & \text{(definition of } I_D\text{)} \\
(SqSts, G) & \models_D & GOAL\ \mathcal{F}(f_{C_T})[x/X, y/Y]
\end{array}$$

(using a weak version of the $GOAL$ modality,

$(SqSts, G) \models_D GOAL\ P \leftrightarrow G \models P$

c.f. Section 3.10.1)

$$\begin{array}{rcl}
(SqSts, G) & \models_D & inst_D[\![GOAL\ \mathcal{F}(f_{C_T})[x/X, y/Y]]\!] \\
 & & \text{(since } inst_D[\![GOAL\ P]\!] = GOAL\ P\text{)}
\end{array}$$

Case $S = \text{NEXT}(S')$

$$I_D[\![\text{NEXT}(S')]\!](SqSts, G) \quad \models \quad \mathcal{F}(f_{C_T})[x/X, y/Y]$$

$$\begin{array}{rcl}
s_{i+1} & \models & \mathcal{F}(f_{C_T})[x/X, y/Y] \\
 & & \text{(where } I_D[\![S']\!] = s_i\text{)} \\
I_D[\![S']\!](tail(SqSts, G)) & \models & \mathcal{F}(f_{C_T})[x/X, y/Y] \\
 & & \text{(lemma 1)} \\
(tail(SqSts, G)) & \models_D & inst_D[\![S'\mathcal{F}(f_{C_T})[x/X, y/Y]]\!]
\end{array}$$

(by application of the inductive hypothesis of the whole proof obligation)

$$(SqSts, G) \models_D inst_D[\![\text{NEXT}(S')\mathcal{F}(f_{C_T})[x/X, y/Y]]\!]$$

Having proved for each case of the state argument, we have now proved

$$I_D[\![X.f_{C_T}(S) == Y]\!](SqSts, G)$$

iff

$$(SqSts, G) \models_D inst_D[\![S\ \mathcal{F}(f_{C_T})[X/x, Y/y]]\!]$$

Base Case 2 (Set membership) Atoms are propositions of the form

$$Y \in X.f_{C_T}(S)$$

Again, from the syntax of GCRL, Y must be a simple object or simple object variable.

$$\begin{array}{rcl}
I_D[\![Y \in C_T(S)]\!](SqSts, G) & = & Y \in I_D[\![X.f_{C_T}(S)]\!](SqSts, G) \\
 & = & Y \in Z \\
 & & \text{where} \\
\{Z' | I_D[\![S]\!](SqSts, G) & \models & \mathcal{F}(f_{C_T})[x/X, y/Z']\} \\
 & = & TRUE \\
 & \text{iff} & \\
I_D[\![S]\!](SqSts, G) & \models & \mathcal{F}(f_{C_T})[x/X, y/Y] \\
 & & \text{(by the definition of the set } Z\text{)}
\end{array}$$

The proof concerning the three cases of state argument does not require repeating, as we have already (see above) proved

$$\begin{array}{rcl} I_D[\![Y \in C_T(S)]\!](SqSts, G) & & \\ & \text{iff} & \\ I_D[\![S]\!](SqSts, G) & \models & \mathcal{F}(f_{C_T})[x/X, y/Y] \end{array}$$

We have now proved

$$\begin{array}{rcl} I_D[\![f]\!](SqSts, G) & & \\ & \text{iff} & \\ (SqSts, G) & \models_D & inst_D[\![f]\!] \end{array}$$

for f = atomic propositions in GCRL. The dependency remains to be shown for the remaining language constructs.

Negation We need to prove:

$$\begin{array}{rcl} I_D[\![\neg P]\!](SqSts, G) & = & TRUE \\ & \text{iff} & \\ (SqSts, G) & \models_D & inst_D[\![\neg P]\!] \end{array}$$

$$\begin{array}{rcl} I_D[\![\neg P]\!](SqSts, G) & = & TRUE \\ & \text{iff it is not the case that} & \\ I_D[\![P]\!](SqSts, G) & & \\ & \text{iff it is not the case that} & \\ (SqSts, G) \models_D inst_D[\![P]\!] & & \\ & & \text{(inductive hypothesis)} \\ & \text{iff} & \end{array}$$

$$(SqSts, G) \models_D \; inst_D[\![\neg P]\!]$$

Conjunction We need to prove:

$$\begin{array}{rcl} I_D[\![P \land Q]\!](SqSts, G) & = & TRUE \\ & \text{iff} & \\ (SqSts, G) & \models_D & inst_D[\![P \land Q]\!] \end{array}$$

$$\begin{array}{rcl} I_D[\![P \land Q]\!](SqSts, G) & = & TRUE \\ & \text{iff} & \\ I_D[\![P]\!](SqSts, G) \text{ and } I_D[\![Q]\!](SqSts, G) & & \\ & & \text{(definition of } I_D) \\ & \text{iff} & \\ (SqSts, G) & \models_D & inst_D[\![P]\!] \text{ and } inst_D[\![Q]\!] \\ & & \text{(inductive hypothesis)} \\ & \text{iff} & \\ (SqSts, G) & \models_D & inst_D[\![P]\!] \land inst_D[\![Q]\!] \\ & \text{iff} & \\ (SqSts, G) & \models_D & inst_D[\![P \land Q]\!] \end{array}$$

Quantification Quantification appears in GCRL in three forms, so we must tackle each one. The first is universal quantification over objects. To prove:

$$\begin{array}{rcl} I_D[\![\forall X : \mathcal{C}_m.P]\!](SqSts, G) & = & TRUE \\ & \text{iff} & \\ (SqSts, G) & \models_D & inst_D[\![\forall X : \mathcal{C}_m.P]\!] \end{array}$$

$$\begin{array}{rcl}
\multicolumn{3}{l}{I_D[\![\forall X : \mathcal{C}_m.P]\!](SqSts, G)} \\
 & \text{iff} & \\
\multicolumn{3}{l}{I_D[\![P[X/t]]\!](SqSts, G) \text{ for all } t \in \mathcal{F}(\mathcal{C}_m)} \\
 & \text{iff} & \\
(SqSts, G) & \models_D & inst_D[\![P[X/t]]\!] \text{ for all } t \in \mathcal{F}(\mathcal{C}_m) \\
 & & \text{(inductive hypothesis)} \\
 & \text{iff} & \\
(SqSts, G) & \models_D & \forall X : \mathcal{C}_m.inst_D[\![P]\!] \\
 & \text{iff} & \\
(SqSts, G) & \models_D & inst_D[\![\forall X : \mathcal{C}_m.P]\!]
\end{array}$$

Next comes existential quantification over objects. To prove:

$$\begin{array}{rcl}
I_D[\![\exists X : \mathcal{C}_m.P]\!](SqSts, G) & = & TRUE \\
 & \text{iff} & \\
(SqSts, G) & \models_D & inst_D[\![\exists X : \mathcal{C}_m.P]\!]
\end{array}$$

$$\begin{array}{rcl}
\multicolumn{3}{l}{I_D[\![\exists X : \mathcal{C}_m.P]\!](SqSts, G)} \\
 & \text{iff} & \\
\multicolumn{3}{l}{I_D[\![P[X/t]]\!](SqSts, G) \text{ for some } t \in \mathcal{F}(\mathcal{C}_m)} \\
 & \text{iff} & \\
(SqSts, G) & \models_D & inst_D[\![P[X/t]]\!] \text{ for some } t \in \mathcal{F}(\mathcal{C}_m) \\
 & & \text{(inductive hypothesis)} \\
 & \text{iff} & \\
(SqSts, G) & \models_D & \exists X : \mathcal{C}_m.inst_D[\![P]\!] \\
 & \text{iff} &
\end{array}$$

$$(SqSts, G) \ \models_D \ inst_D[\![\exists X : \mathcal{C}_m.P]\!]$$

Finally we reach universal quantification over generic clusters. To prove:

$$\begin{array}{rcl} I_D[\![\forall C.P]\!](SqSts, G) & = & TRUE \\ & \text{iff} & \\ (SqSts, G) & \models_D & inst_D[\![\forall C.P]\!] \end{array}$$

$$\begin{array}{rcl} & I_D[\![\forall C.P]\!](SqSts, G) & \\ & \text{iff} & \\ & I_D[\![P[C/\mathcal{C}]]\!](SqSts, G) \text{ for every } \mathcal{C} \in \mathcal{N}_D(C) & \\ & \text{iff} & \\ (SqSts, G) & \models_D & inst_D[\![P[C/\mathcal{C}]]\!] \text{ for every } \mathcal{C} \in \mathcal{N}_D(C) \\ & & \text{(inductive hypothesis)} \\ & \text{iff} & \\ (SqSts, G) & \models_D & \bigwedge inst_D[\![\forall C.P]\!] \text{ for all } \mathcal{C} \in \mathcal{N}_D(C) \\ & \text{iff} & \\ (SqSts, G) & \models_D & inst_D[\![\forall C.P]\!] \\ \square & & \end{array}$$

3.10.1 Subtleties of the goal modality

As was mentioned in 2.3.2, the treatment of the GOAL modality involves certain subtleties. The strong goal modality presents a much easier task when interpreting expressions in the domain level language. We simply look for inclusion in the specified goal set of any particular proposition. However, in the context of proving the equivalence of the two interpretation approaches for expressions in GCRL, we are posed with a problem. The strong goal modality does not allow us to show the equivalence of the atomic predicates. It is the use of a weak goal that allows the following step in the proof:

$$\begin{array}{rcl} G & \models & \mathcal{F}(f_{C_T})[x/X, y/Y] \\ (SqSts, G) & \models_D & GOAL\ \mathcal{F}(f_{C_T})[x/X, y/Y] \end{array}$$

This step would not be provable in the general case if the semantics of the domain level language were given with a strong goal modality. However, as long as $\mathcal{F}(f_{C_T})[x/X, y/Y])$ evaluates to a single proposition in a domain, there is no difference in practice between the strong and weak versions of the goal modality.

$G \models P$ iff $P \in G$ where P is a single proposition

This means that the proof given above holds for all domains in which for all function terms $X.f_{C_T}(\text{GOAL})$, $\mathcal{F}(f_{C_T})[x/X, y/Y])$ is a single proposition. The fact that in GCRL state qualification is only given for function terms means that $inst_D$ never results in the body of a GOAL expression being more complex than the result of $\mathcal{F}(f_{C_T})[x/X, y/Y])$.

3.11 Chapter Summary

In this chapter the architecture of a system that uses generic control rules has been described and each of the components considered. This has necessitated various definitions. The generic cluster prototype wass introduced to represent clusters at an abstract level. A syntax and semantics was given for prototypes and the association between features of generic clusters and the structures that appear as part of the generic type identification process was examined.

A language in which control rules can be expressed (gerneric control rule logic, or *GCRL*) was declared. This language is a temporal logic whose terms are comparisons between features of a generic cluster. A domain level language in which domain specific instances of control rules can be expressed was identified, and an instantiation procedure which converts GCRL epressions into their domain specific instances was provided. The equivalence between the interpretation of GCRL expressions and the combined processes of instantiation followed by interpretation (of the resulting domain level expression) was proven.

Classifications were proposed to describe both generic control rules and instances of generic clusters. These classifications enable us to be more specific in stating which control rules apply to which instances of generic clusters.

Chapter 4

Issues concerning generic control rule logic

4.1 Unification of terms

The comparisons used in GCRL ('==' and '$\in$') provide forms of unification. However, only a restricted form of unification is used in that the structure of the terms is not matched. In that sense, unification is used only as substitution. Unification as substitution is the process by which a variable is bound to either a value or the same value as another variable.

In GCRL the right hand side of the == operator and the left hand side of the $\in$ operator are forced to be object variables. It is the value of these variables that the other terms in the comparisons are bound to. The right hand side of the == operator may be an object variable or a function term representing a partial function. The right hand side of the $\in$ operator may be a function term representing a set-valued partial function. The point of the comparisons is to bind the objects described by the function term to the object given as the other argument of the comparison, or in the case where function terms are not used, to bind the values of object variables to be the same.

An interesting point to note is that the terms involved in the unification operators can have a temporal aspect. This allows us to unify objects *that may be temporally separated*, i.e., the unification of objects that may be in different states. In fact it is this temporal aspect of objects that sets GCRL apart. In contrast to other forms of temporal logic, such as LTL, that allow only temporal qualification of *propositions*, in GCRL we can talk about a particular object *in a particular state.*

4.1.1 The Implicit Variable

Though we do not allow the unification of two function terms, the subject provides some issues worth discussing. Allowing unification of two function terms has the subtlety of implicitly introducing a variable. The unification describes the very fact that its two arguments express some relationship with the *same* implicit object. The variable introduced is the object to which the roots of the function terms are related through their respective functions. Let us take the equality

$$X.at_{T_X}(\text{NOW}) \quad == \quad X.at_{T_X}(\text{GOAL}) \tag{4.1}$$

This represents the fact that some object X of type T_X is related to the same location argument in both the current and goal states (importantly without naming or referencing that location object). In the evaluation of the terms at unification there would be an additional object involved (which would relate to X in both the current and goal states). The expression 4.1 is equivalent to the slightly longer

$$\exists Y \ . \ X.at_{T_X}(\text{NOW}) \quad == \quad Y \wedge X.at_{T_X}(\text{GOAL}) \quad == \quad Y \tag{4.2}$$

What was an implicit variable has been introduced accordingly. It might be supposed that universal quantification of the implicit variable would be used in the evaluation of set membership (as the function term on the right hand side must be set-valued) as in

$$X.at_{T_X}(\text{NOW}) \in Y.link_{T_Y}(\text{NOW}) \tag{4.3}$$

(where Y is a location object and the set-valued function *link* returns the locations accessible from Y) However, expression 4.3 describes the states in which X is situated at *any* of the locations accessible from Y, not the states in which X is situated at *all* of those locations. In the evaluation of the set membership at unification, a third object (the unnamed location) must be introduced. This means expression 4.3 is equivalent to

$$\exists Z \,.\, X.at_{T_X}(\text{NOW}) == Z \wedge Z \in Y.link_{T_Y}(\text{NOW}) \tag{4.4}$$

The implicit variable is introduced existentially as the term $X.at_{T_X}(\text{NOW})$ *must only refer to one object.* The decision not to allow comparisons between two function terms was made precisely to make explicit these subtleties and does not decrease the expressive power of the language.

An aside on $\forall$ and $\exists$

Unification involving two function application terms introduces a variable existentially (because of the way that unification is defined). Likewise, if that variable is explicitly introduced then it is done so existentially. However, as is evident in many of the examples in this thesis, the variable that is used to name the object referred to by a function term is introduced universally. This is a subtlety relating to implication, which is used in most of the control rules in this work to enforce the conditions described. If a variable is introduced existentially within the antecedent of an implication, its quantification can be brought outside the implication by changing it to universal quantification. This process groups the quantification of variables at the beginning of the expression and leaves the body without nested quantifiers. Both from a implementational and conceptual point of view, this is a simplification.

4.1.2 Relic Predicates

The decision not to allow equality between two function terms had the additional benefit of avoiding the unnecessary repetition of predicates in instantiated expressions. The term *relic predicate* refers to these repeated predicates. In order to demonstrate how relic predicates occur, the instantiation of equality must be examined in both cases (between two function terms and between a function term and an object variable). The control strategy "never move a safe portable object from its goal location" will be used in this example. Let us first look at the case without equality between function terms.

$$\forall X : T_x . \forall Y : T_y . X.at_{T_x}(\textsc{now}) == Y \wedge X.at_{T_x}(\textsc{goal}) == Y \\ \rightarrow X.at_{T_x}(\textsc{next}) == Y \tag{4.5}$$

This will instantiate to something of the form

$$at(p,\ q) \wedge GOAL\ at(p,\ q) \rightarrow NEXT\ at(p,\ q) \tag{4.6}$$

If we allow the generic control rule to state equality between function terms then the instantiation is slightly different. This is due to the introduction of the implicit variable (c.f. Section 4.1.1). Allowing functor equality, the above GCR would be written

$$\forall X : T_x . X.at_{T_x}(\textsc{now}) == X.at_{T_x}(\textsc{goal}) \tag{4.7}$$

$$\rightarrow X.at_{T_x}(\textsc{next}) == X.at_{T_x}(\textsc{goal}) \tag{4.8}$$

(N.B. The final term in expression 4.7 ($X.at_{T_x}(\textsc{goal})$) could be replaced by $X.at_{T_x}(\textsc{now})$ as equality of these terms is stated in the antecedent. In practice, the use of the GOAL state argument may be more efficient. This is because the set of goal facts is accessible throughout runtime (from the problem instance), whereas $X.at_{T_x}(\textsc{now})$ would need to be evaluated for each node in the search space.) Expression 4.7 will instantiate to something of the form

$$at(p,\ q) \wedge GOAL\ (at(p,\ q)) \rightarrow NEXT\ (at(p,\ q)) \wedge GOAL\ (at(p,\ q)) \quad (4.9)$$

Notice the repetition of the term $GOAL\ (at(p,\ q))$. It is not required in the consequent as part of the functionality of the control rule, rather it appears there as a relic of instantiation. In particular it is a direct result of allowing equality between two function terms where the object they both refer to has not been explicitly introduced. However, it should be pointed out that whether implicit or explicit variables are used, the resulting expressions are both logically equivalent.

The term $X.at_{T_x}(\text{GOAL})$ is needed to ensure the location object in the subsequent state (described by $X.at_{T_x}(\text{NEXT})$) is bound to the same value as in the current and goal states (expressed in the antecedent). However, using the term $X.at_{T_x}(\text{GOAL})$ commits to evaluation into a predicate.

By restricting the equality to only one function term, this situation can be avoided. A variable must be introduced and bound to the function term and that variable can then be used throughout the expression. This removes the necessity to repeat function terms in order to reference the same value and results in evaluating the function terms to predicates only when necessary (as in 4.5 and 4.6).

4.1.3 Terms and comparisons

The object from which a function is applied (along with a state argument) shall be referred to as the root object of a function term. Objects that appear without function applications will be referred to as simple objects.

Though the idea of comparison between objects (using either equality or set membership) through function terms is easy to comprehend, the process involves careful consideration to be totally understood. No novel or unusual way of interpreting the comparisons themselves is proposed, rather how to apply them in the context of terms that are more complex than simple objects. *What* and *when* we are comparing the objects referred to is of particular interest.

Take the equality

$$X.p'(S) \;==\; Y \tag{4.10}$$

This states that the object related to X through the relationship described by the function p' in the state S is equal to the object Y. Likewise the set membership expression

$$X \;\in\; Y.p''(S') \tag{4.11}$$

states that X is in the set of objects related to Y through the relationship described by the function p'' in the state S'.

Using the definition 3.1, 4.10 refers to the fact

$$p(X,\; Y) \tag{4.12}$$

being in the state S. This will obviously have an associated boolean value representing whether it actually *is* in the state description. Likewise, using the definition 3.1, 4.11 refers to the fact

$$p(X,\; Y) \tag{4.13}$$

being in the state S'.

The evaluation of a comparison involving a function term and an object corresponds to the membership in the specified state of the expression that relates the objects concerned through the predicate given by the function. Evaluation is really a two step process, as is shown in the examples 4.10 and 4.11 above. The first stage is to create an expression whose arguments are bound to the right objects. The function determines the manifestation of the propositional expression $\mathcal{P}$ (Section 3.5.2). The function term is then bound to the simple object, with the effect of substituting the free variables in $\mathcal{P}$ with the root of the function term and the simple object. Were the expression $\mathcal{P}$ to be an expression with more than two free variables, for example a predicate with more than two arguments, the other arguments would need to be existentially quantified over all values of the correct type. However, the responsibility for this lies with the interpretation of the function, which according to this work must return an expression with at most two free variables.

Having evaluated the relation between the objects involved to create an expression in the language of the domain, the next stage of the evaluation looks at entailment. We look for the entailment of the expression $\mathcal{P}$ in the complete state description, and yields $TRUE$ or $FALSE$. The state in which we look for entailment is given by the state argument of the function application term.

Simple objects play a different role to function application terms in comparisons. Simple objects are used to refer to domain objects by their identity only. They are not evaluated in a relation or a state; an object is a constant irrespective of state or the predicates it is involved in. The objects referenced by function terms *are* such objects, but they are identified by their relationships and importantly the state qualification of those relationships. The objects themselves are still constants, but instead of being identified by name they are identified by their relation to other named objects.

A comparison cannot be evaluated without being given the context in which it is said to hold; i.e. the relevant sequence of states. The only states that can be expressed in the language are the current state, future states (using the $NEXT$ modal operator on relations in the current state) and the goal state. The goal state is known throughout solution generation, but the current and (proposed) future states are specific to the particular plan trajectory at a given point.

Comparisons serve the purpose of identifying relations in a state with particular bindings. The control rule as a whole can be thought of as describing a specific pattern of relations between a set of objects, with advice on how to preserve or change that pattern in future states. The control rules can express strategies quantified over both a domain's objects as well as across domains themselves.

4.2 Persistent Relations

Persistence of generic relations can be expressed in GCRs. By stating that a particular relation r holds between objects a and b in the current state *and* in the next state (through the function f that describes the relation r), the persistence of that relation is enforced for the entire plan being constructed (a rule is quantified over all 'current' nodes in a plan).

$$\ldots a.f(\textsc{now}) = b \wedge a.f(\textsc{next}) = b \ldots$$

This is very reminiscent of the temporal modality $\Box$ (always), though we do not have an explicit analogous construct. The notion of persistence is expressed inductively, the base case of which is the assertion of persistence from the current state to its successor state. The inductive step is implicit in the quantification of a rule over all states in the plan trajectory (every state in the plan will be considered the current state at the point of extending the plan from that state).

The semantics of $\Box$ as given in Emerson's LTL as $x \models \Box p \leftrightarrow \forall j(x^j \models p)$ (Emerson actually uses G to represent the always operator $\Box$), where p is a formula in LTL. In GCRL, we can express $x \models \Box r$, where r is a relationship between objects of generic types.

We can also express the fact that *once* a particular relation r holds between objects, that relation persists. This is achieved by using an implication instead of a conjunction.

$$\ldots a.f(\textsc{now}) = b \rightarrow a.f(\textsc{next}) = b \ldots$$

A restricted version of the linear temporal modality $\cup$ (until) can be expressed through the persistence of generic relations. By a disjunction of a persistent relation r between objects a and b by function f and some other relation r' between objects a' and b' by function f', we can express the notion that either now or in some future state, r' holds between a' and b' and until that state, r holds between a and b.

$$\ldots (a.f(\textsc{now}) = b \wedge (a.f(\textsc{next}) = b \vee a'.f'(\textsc{next}) = b')) \vee a'.f'(\textsc{now}) = b' \ldots$$

Notice that it is not sufficient to state

$$\ldots a.f(\textsc{now}) = b \wedge (a.f(\textsc{next}) = b \vee a'.f'(\textsc{next}) == b') \ldots$$

as this does not allow the relation r' to hold in the first node of the search space. This example actually states a subtly different condition; that the relation r holds and persists until the relation r' holds (if it in fact ever does) i.e. there must be a state in which r holds before some state in which r' holds.

The restricted version of $\cup$ that we can express in GCRL is weaker than its true semantics in LTL. In LTL, $x \models (p \cup q) \leftrightarrow \exists j(x^j \models q \wedge \forall k < j(x^k \models p))$ where x is the timeline and x^i is the ith state in that timeline. This asserts that there *must* be a future state x^j in which p is true and that in all states preceding x^j, q is true. The GCRL notion of $r \cup r'$ is weaker in the sense that it cannot assert that a particular relation r' necessarily holds in some future state. The most it can express is that if r' holds in some future state then r holds up till that state, or the relation r persists. $x \models (r \cup r') \leftrightarrow (\exists j(x^j \models r') \wedge \forall k(k < j \wedge x^k \models r)) \vee \forall k(x^k \models r)$

In fact the weaker version of $\cup$ is what Emerson describes as *weak until* or *unless* (denoted as $\cup_\forall$). The semantics are given as

$$x \models p \cup_\forall q \leftrightarrow \forall j.((\forall k.k \leq j \wedge x^k \models \neg q) \rightarrow x^j \models p)$$

Within his PLTL (propositional linear temporal logic), Emerson describes the relationship between weak and strong until ($\cup_\exists$) as

$$\begin{aligned} p \cup_\exists p &\equiv p \cup_\forall q \wedge \Diamond q \\ p \cup_\forall p &\equiv p \cup_\exists q \vee \Box p \\ &\equiv p \cup_\exists q \vee \Box(p \wedge \neg q) \end{aligned}$$

In linear temporal logic, the modalities $\Box$ and $\Diamond$ (eventually) are equivalent to until assertions [18], though they are employed as intuitive short versions. In particular, $\Diamond x \equiv TRUE \cup x$ and $\Box x \equiv x \cup FALSE$. Through the use of persistent relations, $\Box$ and $\cup_\forall$ statements can be expressed in GCRL. It might therefore seem natural that $\Diamond$ expressions were also possible within the language, as reformulations of always or until expressions. This, however, is not the case. Because we do not have a way to express $\Diamond$ we can not express strong until.

The reason we cannot express $\diamond$ is to do with the way that the language uses the state argument of a function term. We can refer to, through the use of nested successor functions, relationships that hold in states beyond the immediate successor state. These states must be explicitly specified (for example, the third successor state is referenced by the state argument NEXT(NEXT(NEXT))). In order to express $\diamond$, we would need to existentially quantify over *the number of nested* 'NEXT*'s* in an expression, to denote that a relationship holds in *some* future state. This is a critical difference between GCRL and other proposed temporal logics such as LTL.

Expressions are evaluated at the point of selecting an action to progress the plan from its current state to its next state. Rules that refer to relationships at most one state into the future need not be progressed (at evaluation, a candidate successor state is known), though a progression algorithm would need to be employed to maintain the semantics of expressions across sequences of states beyond the immediate successor.

4.3 Hierarchies of generic clusters

Having introduced the ideas of inclusive and exclusive instances of generic clusters in Section 3.4, we are now in a position to discuss their implications on hierarchies of generic clusters.

It is evident that hierarchies of generic clusters do exist. There are several simple examples that demonstrate this. A cluster C involving mobile objects on a map of locations can be seen to be a base cluster for the derived cluster C' in which there are portables that can be transported by those mobile objects. C' in turn can be seen as a base cluster for the derived cluster C'' in which the portable objects are identified as safe portables. The interesting question here is that of the applicability of rules to the derived cluster that were formulated for the base cluster.

Derived clusters can be seen to be exclusive or inclusive with respect to the base cluster of which they are an instance. A derived cluster C' is exclusive with respect to C if and only if the generic relationships of C do not appear in the preconditions or effects of any of the operators that are additional generic operators in C'. Failing that condition, C' is inclusive with respect to C (i.e. C' includes the behaviour of C but has some behaviours in addition that may affect the behaviour derived from C).

A point worth emphasising here is that the inclusivity or exclusivity of a derived cluster with respect to a base cluster is independent from the inclusivity or exclusivity of the particular instantiation of a cluster in a domain. An instance of a cluster in a domain may be described as an inclusive or exclusive instance, according to its characteristics (see Section 3.4). This classification is distinct from the relationship between abstract forms of clusters, at which level one may be seen to be a derived instance of the other in either an exclusive or inclusive fashion.

Both inclusive and exclusive rules (applicable to some base cluster C) can be correctly applied to derived clusters if the derived clusters are exclusive with respect to the base cluster C. This is because the derived instance only adds behaviour that does not disrupt the behaviour described in C (for which the rule was formulated). If, however, the derived cluster is inclusive with respect to C, there is no guarantee that the behaviour for which the rule was formulated will be unaffected by any additional behaviours the derived cluster adds. In this case, it is clear that exclusive rules can not be safely employed by the derived instance, but the case of inclusive rules is not to straight forward.

Inclusive rules rely only on the behaviour described at the cluster level, irrespective of any additional behaviours that an instance of a cluster may display. It should then seem that inclusive rules be applicable to clusters derived from the cluster for which the rule was formulated.

However, having admitted the possibility that a derived instance may add behaviours that affect the behaviour described by the base cluster, is it possible that we cannot rely on the instance to exhibit the behaviour for which the inclusive rule was formulated? If the behaviour of the base cluster is not exhibited by the derived cluster, it is difficult to see in what way it is a derived instance. The crucial observation is that although the derived instance may add behaviour that *interacts with* the behaviour inherited from the base cluster, *that inherited behaviour must be present in the derived instance.* As the definition of an inclusive rule (3.4) is based solely on the existence of the appropriate behaviour, regardless of other behaviour that may exist for the types involved, we *can* apply inclusive rules to inclusively derived clusters.

4.4 Multiple generic clusters

All of the example abstracted control strategies in this thesis are quantified over only one generic cluster. The generic cluster provides us with an abstraction of a set of types of objects that necessarily interact or relate to each other in a determined fashion. It is precisely this behaviour that we have used to construct abstractions of optimising control strategies. We have not considered control strategies that are based on the interaction of multiple generic clusters. There are several reasons for making this simplifying decision.

Were we to consider strategies based on multiple interacting clusters, we would need methods of describing the relation between those clusters. The relationships that we would need to specify are the same as those described in Section 3.5.4 (namely the relationships required for generic type identification), but would relate features of one generic cluster to another. The proposed language outlined in Section 3.5.4 would in fact be able to accurately describe the relationship between instances of generic clusters.

The extent to which generic clusters interact is closely related to the definition of exclusive and inclusive instances of clusters (c.f. Section 3.4). If the interaction is between exclusive instances of generic clusters, the behavioural functionality of each cluster remains unchanged and the interaction must be between properties that the types have in addition to their generic features. In this situation, any control strategies that were applicable to the individual clusters would still be valid, with the scope for additional constraints based on the compound behaviour.

Consider a domain (let us call it *painted–blocksworld*) which extends the standard blocksworld domain by allowing the blocks to be painted (whether or not they are stacked). We know from the PaintWall domain that the painted objects can be seen as mobiles on a map of colours. The colours are identified as locations, where a link between locations a and b is encoded as the ability to paint over colour *a* with colour *b* [38]. As long as the stacking behaviour of the blocks does not interfere with the painting of those blocks, we have a domain in which there are two exclusive instances of clusters. The blocks are both construction objects and mobile objects.

If the interaction is between inclusive instances, the only strategies we have available are inclusive rules. By their definition, these inclusive rules are still applicable if the cluster to which they pertain is affected by other behaviour, be that in the form of other recognised generic behaviour or not. The depots domain [39] is an example of of the interaction between two inclusive instances of generic types. In this domain, crates are transported on trucks. In this respect the crates can be seen to be portable objects. On top of that behaviour, the crates can be stacked (either on the lorry or in the depot). This behaviour is that of a construction type (as with blocksworld, where the blocks are construction materials and the compound objects are the towers). Note that although the depots domain was designed to sythesise the elements of blocksworld and transportation domains, the existing generic type identification machinery does not identify the crates as portables as only *exclusive* instances of generic clusters are recognised (the fact that they display additional behaviour means that the crates are an *inclusive* instance of a portable type).

An interesting point raised in Section 3.4 is that we can give tighter constraints with an exclusive instance than with an inclusive one. This is because we know that the behaviour outlined by the generic cluster is unaffected in the instance, whereas all we can say about an inclusive instance is that it *includes* certain generic behaviour. With this in mind, it could be argued that an exclusive instance of a 'compound' generic cluster (a cluster describing the overall behaviour of interacting constituent clusters) would be of greater use than inclusive instances of two or more interacting clusters. This line of argument suggests the creation of new compound clusters whenever we have interacting clusters, but in the general case this would result in an unmanageably large collection of clusters. The exploration of the interaction of generic clusters is an important and interesting direction for further research.

4.5 Binary Predicates

There were several factors contributing to the restriction of the predicates dealt with to be at most binary.

Domains that contain predicates with arities greater than two can be transformed into equivalent domains in which all the predicates are at most binary [54].

Instantiating a function term results in a propositional expression $\mathcal{P}$ with at most two free variables. This is because the function term describes a relationship between two objects, namely the base object of the function term (x in $x.f_{C_{T_x}}(S)$) and the object that the function term represents. In the case of a set-valued function term, the function term denotes a binary relationship between the base object of the function term and each of the objects in the set that the function term represents. These binary relationships are reflected by the binary comparison operators.

4.5.1 Two free variables

As has been discussed (3.5.2), the binding function $\mathcal{F}$ that is supplied by TIM yields a propositional expression with at most two free variables when applied to a function f_{C_T}. To understand why this expression may have two free variables we must consider what the function f_{C_T} represents, namely that a recognisable relationship exists in a domain between the base object of the function and the object that the function describes (or in the case of a set valued function, that the relationship exists for every object in the set described). It is this binary relationship that dictates the number of variables that may occur free in the propositional expression given by $\mathcal{F}(f_{C_T})$. Section 3.5.2 describes how $\mathcal{F}(f_{C_T})$ may return an expression that actually contains fewer free variables, in the case of implicit arguments to predicates or implicit predicates in the domain. In these cases, the same binary relationship is described by the function, but it is the realisation of that relationship in the domain that may use one or more of the objects implicitly. Another interesting point worth noting is that where $\mathcal{F}(f_{C_T})$ gives an expression that involves objects other than the base object of f_{C_T} and the object(s) described by f_{C_T}, those objects must be introduced explicitly by $\mathcal{F}$. For example, if in a logistics style domain the locatedness predicate of a mobile was a predicate of three arguments $at(x, y, z)$(where x is the mobile, y the location and z the city in which the location is found), the GCRL proposition $X.at(\text{NOW}) == Y$ would be instantiated as $(\forall P : T_P \; . \; at(x, y, P))[X/x, Y/y])$

4.6 Local versus global constraints

We need to acknowledge the difference between *local* and *global* constraints (a control rule or strategy can be seen as an imposed constraint). The terms local and global refer to the structure of the plan; a global constraint is concerned with the global structure of the plan whereas a local constraint affects properties of a plan fragment. The terms local constraint and global constraint are used informally in the literature, so we provide definitions here for the purposes of concise discussion.

Definition 9 Local constraints *restrict a sequence of states,* $< s_0, \ldots, s_n >$*, based on a state description* (s_i *where* $0 \leq i \leq n$) *or sequence of state descriptions* ($< s_j, \ldots, s_k >$ *where* $0 \leq j \wedge k \leq n$)*, possibly in conjunction with the goal set* (s_{goal}).

An example of a local constraint is generic control rule that states "never move a safe portable object from its goal location." This is a local constraint as it constrains the subsequent sequence of states based on a local state description (the state in which a safe portable is located at its goal location).

Definition 10 Global constraints *restrict the structure of the plan as a whole, i.e. they necessarily constrain the sequence of states* $< s_{initial}, \ldots, s_{final} >$. *The restrictions may be based on individual state descriptions* s_i *where* $initial \leq i \leq final$*, but only in the context of the entire plan structure. The restrictions may be put in the context of the entire plan in two ways, in the form of goal ordering information (top level goals or subgoals) or by explicit state references (such as 'let proposition* $\mathcal{P}$ *be true in the* x*th state in the plan')*

An example of a global constraint is the ordering of subgoals, such as "move object x to its goal location before moving object y to its goal location. Notice that once x is located at its goal location, the achievement of y at its goal location is still a global constraint, precisely because we only want to achieve this *once we have located* x. It is not the case, however, that global constraints are necessarily object specific. Consider the strategy "locate all safe portable objects at their goal locations before locating mobile objects at their goal locations," which is not object specific. Both local and global constraints can be problem specific but are not necessarily so (in fact the local constrains used in this work are necessarily *not* problem specific, as a result of the abstraction to generic clusters).

Generic control rule logic can only describe relationships between objects in states that are *explicitly* related to the current state (or the goal description) and can not express object specific statements. Because of this, the logic can only specify a subset of local constraints. Let us examine the qualification of that subset.

4.6.1 Local contraints in GCRL

GCRL is not an object level language. As such it can only express those constraints that are not object specific. More than that, as the object abstraction is based on the behaviour of interacting generic types, the constraints we can express are limited to those that can be described within a generic cluster (or generic cluster*s*, c.f. Section 4.4). Finally, the expressible local constraints are restricted to those that use reference to explicitly related states in a sequence of states (rather than through modes such as 'until' ($\bigcup$) and 'eventually' ($\Diamond$). The following expression represents these restrictions.

$$C_{GCRL} = C/(C_a \cup C_b)$$

where C_{GCRL} are the local constraints expressible in GCRL, C is the universe of local constraints, C_a are those local constraints *not* expressible within a generic cluster and C_b are those constraints *do not* use explicit sequential relations between states.

4.6.2 Global 'local' constraints

There are two situations in which local constraints might be considered global constraints. The first is the case in which the plan fragment used by the local constraint happens to be the entire plan sequence. We would classify this as a local constraint that was incidentally global for that specific problem. The second is the case in which local constraints are quantified over all sequences of states in the plan. The quantified expression is a global constraint as it imposes the condition that every plan fragment adheres to the local constraint *throughout the entire plan.* This case is very interesting as it relates to the control rules used in this work (the rules are implicitly quantified over all states through the use of the NOW state argument, which refers to the current state at every stage in plan construction).

We argue that through the case of implicitly universally quantified local constraints, a subset of global constraints can be represented in GCRL. That subset is the set of local constraints, C_{GCRL} (see above), implicitly quantified over all state sequences in the plan (through the use of explicit state relations relative to the current state).

It should be noted that the language of GCRL does not explicitly exhibit features for the specification of global constraints. It is the way in which the rules are used in conjunction with plan generation that enables this phenomenon. However, it is the nature of the rules that suggests this use (unless a local constraint given in terms of a current state was intended to be universally quantified over *all* current states, precise qualification of *which* current state in a particular problem would be required).

4.7 Completeness preservation

Whenever control rules are used to direct search, the preservation of the completeness of the search must be considered. It is not the case that completeness must necessarily be preserved, rather that awareness of this issue should be acknowledged. Here we discuss the concepts involved and make statements regarding the completeness preservation of the methods described in this thesis.

Soft control information, such as the preference rules described in Section 2.5.2, is completeness preserving no matter what strategy is expressed. This is because is does not restrict the search space, rather the control information enforces orderings on the expansion of nodes (or regions) in it.

Hard control information can be completeness preserving but can also introduce incompleteness into the search space, depending on the strategy it expresses. Let us look at examples of both of these situations.

Consider the strategy, in the *logistics* domain, that states that packages should never be loaded into a vehicle from which they have just been unloaded. This is hard control information as branches will be pruned from the search space in which packages *are* put straight back into vehicles from which they've been removed. Though there may be solutions residing in the branches that are removed, those same solution states will remain in the search space elsewhere (intuitively, we will be left with paths to the solutions that do not contain irrelevant cycles of loading and unloading packages). This is represented in Figure 4.1.

Let us look at an example of hard control information that does introduce incompleteness into the search space. Consider a transportation style domain, in which packages must be delivered to a selection of destinations from a selection of sources, where the fuel of the carriers is considered. A strategy that is employed by control rule based planners such as TLPlan in transportation domains states that only packages with like goal destinations should be loaded into the same carrier. Depending on the available resources of fuel in the domain, this control strategy may threaten the completeness of the search space. There may only be enough fuel available in the domain to reach the goal state by transporting the packages in a more efficient manner, so adhering to the control strategy may in fact make the problem unsolvable.

The control strategies expressed in GCRL can be used as either hard or soft control information. They can also express completeness preserving and completeness threatening rules. The rules used in the system described in Section 5.3 are all completeness preserving and are used as hard control information (they are used to prune branches of the search space).

A possible use of GCRs is in the form of a control information toolkit. The

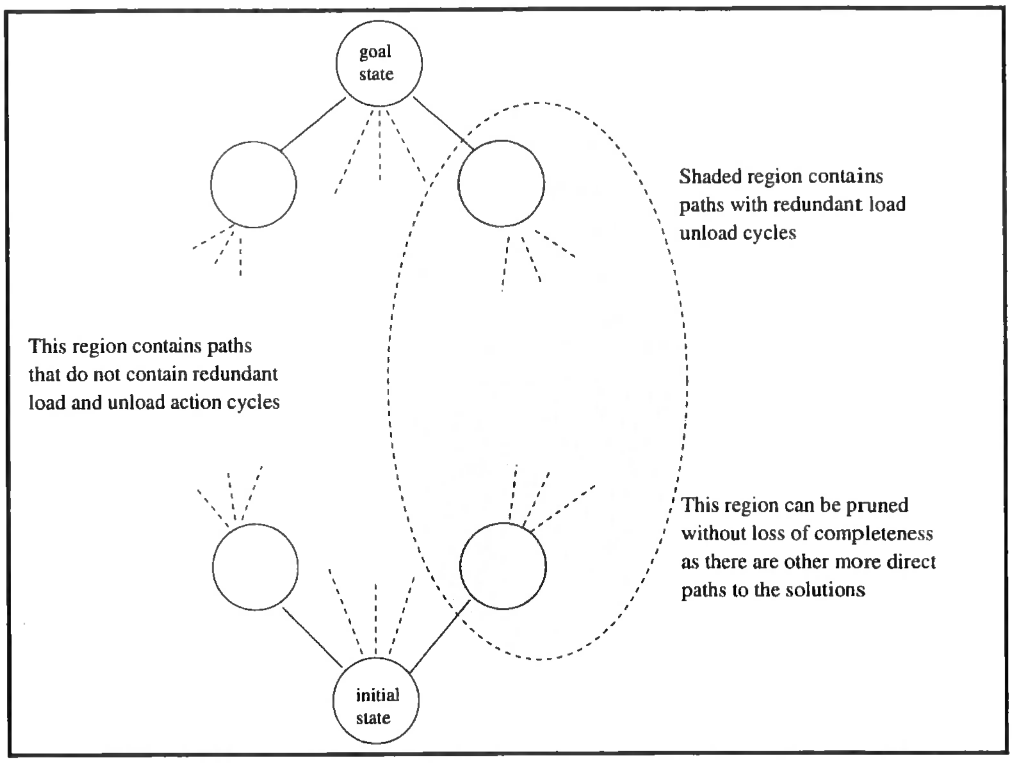
goal state
Shaded region contains paths with redundant load unload cycles
This region contains paths that do not contain redundant load and unload action cycles
This region can be pruned without loss of completeness as there are other more direct paths to the solutions
initial state

Figure 4.1: Completeness preservation in pruning the search space

user would be able to select which rules from a repository of control strategies should be used on a particular problem, and could specify whether those rules be used prescriptively or absolutely (as soft or hard control information). This process could offer an attractive way to generate good plans across a range of applications, better ways of combining control strategies for specific domains and a better understanding of the role of varying combinations of control rules in the search for better plans.

4.8 Chapter Summary

In this chapter both the theoretical and practical issues that arose in the definition and implementation of GCRL were presented. Subtleties involved in the unification of terms that the comparison operators provide were examined. Although there is no state qualification in GCRL equivalent to the $\Box$ and

$\mathcal{U}$ modal operators of LTL, the extent to which analagous strategies can be expressed was discussed.

The definitions of generic control rules and instances of clusters as exclusive or inclusive (c.f. Section 3.4) facilitated discussion of both hierarchies of generic clusters and rules involving multiple generic clusters.

The way in which using generic control rules affects the search space was considered. Both the types of constraint that can be imposed with GCRL and the notion of completeness preservation were examined.

Chapter 5

Proof of Concept

5.1 Overview of results

The results gathered for the work described in this thesis are in four sections. These demonstrate various aspects of the ideas presented.

Data is presented showing the time cost of generating domain specific instances of abstract control rules for several problem sets. The problem sets are collections of randomly generated instances of domains used in the 1998 planning competition [39], including domains into which none of the rules instantiate (instances of the appropriate generic clusters are not present). This demonstrates the cost of using the generic control rule module to generate domain specific rules. Analogous versions of these rules can be seen to be used by control rule based planners such as TLPlan and TALPlanner (but are object level instances of the abstract rules presented here). These results use a stand alone version of the control rule instantiation system, in which the abstract control rules are instantiated into domain specific control rules but are subsequently not used with any planning system. The results are important in their own right as they demonstrate the time cost of instantiation (and the cost involved where no rules are appropriate).

A method is described for linking the abstract control rule instantiation system to a competitive state of the art planner, FF [29]. This planner was not designed to have control rule input over and above the bare domain description and problem instance, but successful integration is achieved and improvement in the resulting plan quality is observed. Some initial results are presented demonstrating this improvement, with the addition of the generic control rule module. The subsumption, by the heuristics employed by FF, of many of the basic rules that are used by control rule based planners is discussed. It should be noted that the results supporting the thesis *do not* rest on these limited empirical results, but on all of the results presented in the context of the literature. These preliminary results are presented to demonstrate that the methods described can be successfully integrated with existing planning systems and have a positive effect on the quality of the plans produced.

We show the wider applicability of the use of generic control rules. A method is presented for instantiating an abstracted form of the well known 'good tower' heuristic [3]. This process is supported by the outlining of a generic cluster prototype for safe construction clusters (an extension to the work covered by Clarke [12]). We then present an abstracted version of the good tower rule in terms of that prototype and detail the instantiation of that rule for a particular domain. We also go on to propose a generic cluster prototype for a behaviour cluster that is identified by an alternative process to 'classic' generic type identification. We discuss the use of such a behaviour cluster with respect to writing reusable abstract control rules, and consider extensions to the language that may be demanded.

We present a translation of the control rules used in a TLPlan domain encoding. This involves interpreting that encoding, then demonstrating that analogous control information can be expressed in generic control rule logic. Not all of the control strategies are accommodated by the behaviour based abstract versions, but of those that are, some of the GCRL representations presented are more succinct (the TLPlan rules contain repetition). We explain why a full set of analogous control strategies is not presented here.

5.2 Results

The results show the times taken to run the stand alone version of the control rule instantiation system for the indicated domains (an independent version of TIM was also timed on the problem sets; these results were not explicitly included in order to simplify presentation in the table). The average running time was taken over ten randomly generated problem instance for each size of problem (where the size of the problem is given by the set of parameters used in problem generation). The systems (the generic control rule module and an independent version of TIM) were run on each problem in each problem set ten times, in order to get more reliable timing results irrespective of minor variations in actual running times. The time measured is the average user CPU time. This measure of the running time was selected to give a fair representation of how much time the program demanded of the CPU, irrespective of any other processes that happened to be running. All tests were performed on a 600MHz machine with 128Mb of RAM running Mandrake Linux 8.1. The systems described use a mixture of C and C++, and were all compiled using the *gnu* compiler.

The results also show the average difference between the running times for the generic control rule module and TIM. This gives a measure of any time penalty that is paid for using generic control rules *on top of* the existing TIM analysis. Finally, the results show timings based on the number of rules that the generic control rule module has that *may* be instantiatable (for one, two and three available rules).

Tables 5.1 and 5.2 demonstrate the cost of using the generic control rule module in domains where appropriate generic clusters are *not* identified. As we can see, the running time of the generic control rule module does not change significantly with the number of rules that it has available to instantiate. This should not surprise us, as, by the nature of the domains, we cannot instantiate any of the rules. The problem sets used increase in complexity as all their parameters are increased by a factor of two. Although there is some increase in running time for the harder problems, it is less than a linear relationship with this increase in the parameter values. We can also see that no matter what the complexity of the domains, nor the number of rules that are available, the additional time penalty of using the generic control rule module is only a few thousandths of a second (of the order of a few percent of the total running times).

Table 5.3 shows the running times for the stand alone generic control rule module on the logistics domain. This is a domain in which there *is* an instance of a safe portable cluster, so the times show the combined cost of generic cluster identification and instantiation of the indicated number of rules. Again, the problem sets represent problems generated with all parameters increased by a factor of two. As can be seen, there is an obvious increase in time as the system is given harder problems. There is a massive increase from around 0.13s to between 9s and 10s as we compare the times for the penultimate and ultimate problem sets. However, by looking at the results showing the additional time penalty for using generic control rules on top of the TIM analysis, we can see that it is TIM that dictates this increase. The additional penalty paid is still only of the order of a few percent of the total running time. There appears to be a definite increase in the additional time measurement as more rules are made available. For the first four problem sets, there is only a few thousandths of a second between results for all problem sets and numbers of rules. For the hardest problem set the additional time penalty is around a half a percent of the total running time but for one or two rules, but for three rules this metric jumps to around four percent.

5.3 Integration with FF

FF was chosen to test the utility of GCRs for a number of reasons. The fact that it is a forward chaining planner means that at every node in the search space a complete state description is known. With respect to state-based control rules, complete state descriptions provide all the information that could be required. The fashion in which FF extends its current solution path also provided an ideal opportunity to use control information of the form that GCRs supply. FF considers a set of successors to the current search node and selects the first candidate that improves on the heuristic value of the current state. It does so with an estimate distance to the goal state based on the length of the relaxed plan from that state to the goal. In the event of none of the successors offering a better heuristic evaluation, as in a plateau, FF considers the successors' successors, and so on. On top of this powerful heuristic, it has an additional strategy of selecting *helpful actions.*

5.3.1 Helpful actions

The idea of *helpful actions* is to restrict the candidate successor states from any search node. Based on the assumption that achieving any of the goals $\{g_i, \ldots, g_j\}$ in the first layer of the relaxed plan from a node will be useful in progressing the solution being generated towards the goal, helpful actions are precisely those actions that achieve a goal g_x where $g_x \in \{g_i, \ldots, g_j\}$. FF uses helpful actions to restrict the candidate successor states to a search node, but still uses the strategy of selecting the first candidate in the set that yields a better value than the current state according to the relaxed plan heuristic.

5.3.2 Control rule application

As FF was not designed to take control knowledge, the process of using the control rules generated was not as straightforward as it otherwise might have been. Rather than implement what would amount to a first-order formula interpreter to facilitate the use of control rules with FF, the instantiated control rules were fully grounded in the results presented. It is accepted that this approach is very naive and expensive but in the name of implementational simplicity it was decided that it would be sufficient to demonstrate proof of concept. In this respect, proof of concept entails the generation of plans whose sequential length is closer to the optimal length (i.e. shorter). We are *not* demonstrating competitive timing results for the working of the system, hence the decision to fully ground the expressions.

At the point where FF selects a candidate successor state, it has complete descriptions of both the current and the candidate next states. Given these, and the representation of the goal conditions (which FF can access at any point during search time), there is the perfect opportunity to analyse the proposed next state with respect to any control rules. This is the case whether the proposed next state comes from the set of helpful actions, or in the case of this set becoming empty, the full set of available actions.

N.B. In order to make use of control rules that refer to any successor states past the immediate successor, a progression algorithm would need to be employed. This would convert the control rule to show the appropriate constraints on the successor states. This has not been included in the implementation as considering only the immediate successor state in control rules has been enough to demonstrate proof of concept. This would, however, be a logical extension of the work described and a version of the progression algorithm described by Bacchus and Kabanza [3] could be employed.

Since the control rules used in this implementation only refer to at most the immediate successor state, the control rules can be evaluated with respect to the current and proposed next state. The control rules are given the goal, current and next states and on the basis of their truth evaluation, the candidate next state is either kept or rejected. All the control rule reasoning is in addition to FF's own search strategies.

5.3.3 The value of FF

As one of the most competitive current planning systems, it was decided that any improvement in FF's performance seen with the addition of GCRs would be significant. It was felt that improving the performance of simpler, outdated forward chaining planners would demonstrate proof of concept but that a greater show of value would be gained from using a state of the art system. This decision was not without consequences. The very fact that FF is so competitive meant that there was some difficulty in identifying control strategies that were not subsumed by FF's very powerful combination of the relaxed plan heuristic with helpful actions. Where much more naive planners would be expected to show a performance improvement with some basic control rules (such as the preclusion of complimentary action cycles), FF needed some more subtle examples. This point has had a major influence on the empirical results presented, which are forced to be only a preliminary proof of concept. However, the results for both the time penalty incurred (over and above the TIM analysis), the expressibility of proven control strategies and the wider applicability of the approach to other behaviour clusters and subproblems all support the limited empirical results. In particular, the ability to express established control strategies in their abstract forms in conjunction with the literature (where object specific analogous rules have shown their worth) is proof of concept in itself.

Attempts at finding suitable heuristic

As has been mentioned, FF's own heuristic which comprises a relaxed plan estimate of distance to the goal and helpful actions is very powerful. This meant that some examples of control rules that are employed by control rule based planners such as TLPlan and TALPlanner were simply subsumed by FF's own mechanics. A prime example of this is the rule that states that a safe portable object should never be moved from its goal location. FF never adds an action that moves a safe portable from its goal location, as the relaxed plan estimate would give a higher value for the distance to the goal state if a suitably located safe portable was picked up. A control rule was found that was not subsumed by FF's default behaviour and is described here.

> Forall Safe Portable clusters, when a carrier *c* contains a portable *p* that has the same goal location as another portable *p'* and the carrier can move directly between it's current location and the location of *p'*, *c*'s location in the next state should be the same as the location of a portable *p"* in the next state where *p"* has the same goal location as *p*.

(N.B. *p"* must be introduced to allow for the fact that there may be more than one possible value for *p'*, and we simply want to select one of those. Also, importantly, this rule only applies where the carriers are of unlimited capacity.)

The generic control rule expressing this is

$$\begin{aligned}
&\forall C : SafePortableCluster.\ \forall d, e, f, g : C_{Location}.\ \forall c : C_{Carrier}.\\
&\quad \forall x, y : C_{SafePortable}.\ \exists z : C_{SafePortable}.\\
&\quad x.in_{C_{SafePortable}}(\text{NOW}) == c \wedge c.at_{C_{Location}}(\text{NOW}) == d \wedge\\
&\quad y.at_{C_{SafePortable}}(\text{NOW}) == e \wedge x.at_{C_{SafePortable}}(\text{GOAL}) == f \wedge\\
&\quad y.at_{C_{SafePortable}}(\text{GOAL}) == f \wedge e \in d.link_{C_{Location}}(\text{NOW})\\
&\quad \rightarrow z.at_{C_{SafePortable}}(\text{NOW}) == f \wedge z.at_{C_{SafePortable}}(\text{NEXT}) == g \wedge\\
&\quad c.at_{C_{Location}}(\text{NEXT}) == g
\end{aligned} \tag{5.1}$$

The situation described in this rule can be see in Figure 5.1, where the goal location of both *spo* 1 and *spo* 2 are the same. Application of the rule would mean that the carrier would move, if it moves at all, to the location of *spo* 2 (or the location of some other safe portable with the same goal location as *spo* 1, if one exists).

5.4 Constructed domain

The domain used to demonstrate the partnership of FF and the control rule instantiation system is a simple logistics style domain (with no planes). The domain description is given shown in Figure 5.2. Various problem instances were constructed to highlight the rule employed. It is important to remember, however, that the control rule used in this example is expressed at the abstract level, and is instantiated into the domain automatically as problems posed in this domain are encountered (*the same rule would be instantiated into any domain encountered that exhibits the SafePortable cluster*).

5.4.1 Problem set and results

The problems were constructed to show situations in which the chosen control rule can improve on the plan quality delivered by FF. By plan quality we mean linear plan length, where higher quality plans have lengths closer to the optimal plan length. It should be stressed that we are not looking for improvements in solution generation time (we have acknowledged that a naive instantiation

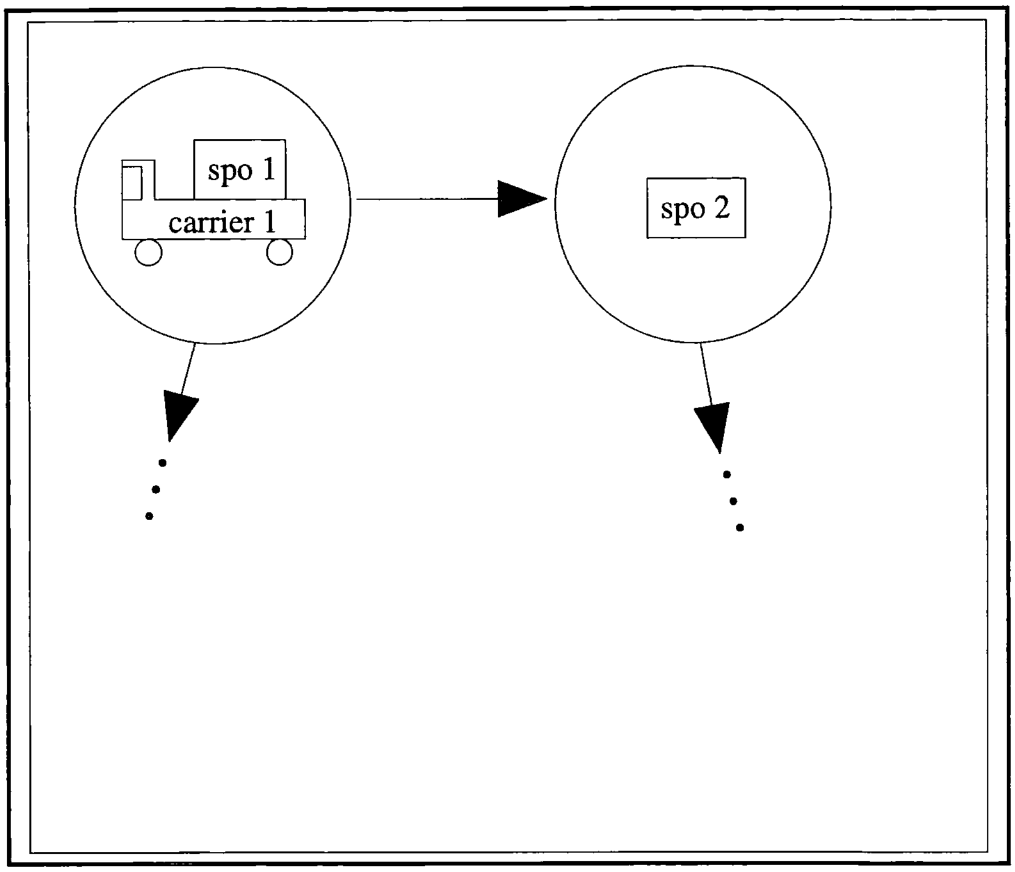

Figure 5.1: Example rule situation

method is used, though future work may address this). It should also be noted that only preliminary results are included, but that those preliminary results do show the positive effect of the addition of generic control rules.

For each problem, the compound system of FF plus generic control rules (we may refer to this as *FF++*) was run ten times, and the average CPU user time was recorded. Again, this measure of running time was chosen to most appropriately represent the work demanded of the processor. An independent version of FF was also run on the problems, to compare both plan length and solution generation time. This too was run ten times for each problem. Lastly, a stand alone version of the control rule module (instantiation of the rules without further use) was also timed on the problems. This system mirrored the work being done by the composite system, in terms of the available rules and domain and problem definition, and was also run ten times on each problem.

The initial state of problem 1 is shown in Figure 5.3. The goal is that *package*1 and *package*2 should both be located at Z. Problem 2 is the first variant on problem 1, with an additional location I on the path between X and Z. The initial state for problem 2 is shown in Figure 5.4 and the goal remains the same as for problem 1. Problem 3 is another variant of problem 1, with the addition of more locations and another package to be delivered at location Z. Problem 3 shows the way in which the simple situation captured in problem 1 can recur in a domain, enabling multiple steps in the plan to be saved by the use of the appropriate generic control rule. The initial state for problem 3 is show in Figure 5.5, and the goal location of all three packages is location Z. Table 5.4 shows the timing results for this problem set.

In each of the problems tested a reduction in solution length can be observed. Both FF and the instantiation only version of the generic control rule module have quite uniform running times across the problems. FF++'s running times increase as we look down Table 5.4 but we also note that the number of objects involved in the problems grows (for example, problem 1 has only three locations, four location links and two packages whereas problem 3 has six locations, nine location links and three packages).

5.5 Other generic clusters

5.5.1 Safe construction cluster

Another generic type that has been identified in the literature is the *construction* type [12]. This is the generic type whose members can be comprised with other objects in specified states to create new objects (such as towers in the blocks world). We propose a *safe construction object* cluster, whose member types are a *safe construction* type and a *state* type (denoting the state of construction objects). A safe construction type is similar to a safe portable type in that the construction type is identified as safe if and only if its members play no other role in the plan than to be 'built' with (comprised with other objects).

```
prototype SafeConstructionCluster {
  Types:  { Material, State }
  Functions :  neutralState:Material→TV
               composedOf:Material→Material+
               oldState:Material→State+ }
```

where TV is a truth value.

We know that a useful rule for construction objects uses the *good compound* [12] definition (which is itself an abstraction of the *good tower* definition [3]) but we must look at two natural extensions to the language. We use the term *definition* for a truth valued function, as in the *neutralState* function shown above. Definitions are needed when the type has a property that needs to be verified on a per object basis, where no other relationships exist that reveals this property. This addition alone does not, however, enable us to represent the good compound rule using the safe construction object cluster prototype as it is. To understand the reason for this, let us attempt to encode the good compound definition. A compound object with outermost object x (such as the top block on a tower in the blocks world) is a good compound if *either* x is in its neutral state in the current and goal states *or* x is composed of y in the current and goal states and y is a good compound. It is this last recursive call to the definition that causes the problem, as we cannot name the definition (as a control rule) in GCRL. This prevents us being able to call the defined rule from within itself as in

$$\forall \mathcal{C} : safeConstructionCluster\ .\ \forall X, Y : \mathcal{C}_{material}$$

$$X.neutralState_{\mathcal{C}_{material}}(\text{GOAL}) \wedge X.neutralState_{\mathcal{C}_{material}}(\text{GOAL})$$

$$\vee$$

$$(X.composedOf_{\mathcal{C}_{material}}(\text{NOW}) == Y \wedge X.composedOf_{\mathcal{C}_{material}}(\text{GOAL}) == Y$$

$$\wedge \text{ (recursive call on Y))}$$

A solution to this problem is to add the definition of good compound to the primitive functions of the prototype, as in

```
prototype SafeConstructionCluster {
  Types:  { Material, State }
  Functions :  neutralState:Material→TV
               composedOf:Material→Material+
               oldState:Material→State+
               goodCompound:Material→TV }
```

and leave it up to the grounding function $\mathcal{F}$ to interpret the manifestation of this property in the domain. The good compound rule could then be expressed as

$$\forall C : safeConstructionCluster\ .\ \forall X, Y : C_{material}$$

$$X.neutralState_{C_{material}}(\text{GOAL}) \wedge X.neutralState_{C_{material}}(\text{GOAL})$$

$$\vee$$

$$(X.composedOf_{C_{material}}(\text{NOW}) == Y \wedge X.composedOf_{C_{material}}(\text{GOAL}) == Y$$
$$\wedge Y.goodCompound_{C_{material}}(\text{NOW}))$$

It is feasible that we could expect this but would force the domain level language presented in Section 3.8 to be extended to allow definitions (the recursive nature of the definition would be passed on and would still need to be expressed). We do not see this as a problem in the language (the temporal logic of TLPlan was extended similarly for precisely the same reasons).

It could be argued that we are placing too much responsibility on the grounding function in terms of the amount of work we are expecting it to invest in delivering the manifestations of functions expressed at the cluster level. However, it is not evident exactly where we should draw the line. It is certainly reasonable to give the grounding function *some* responsibility, as in accounting for things such as implicit arguments or implicit predicates. A useful addition to the mobile object cluster might be the addition of a *reachable* function with the declaration

$$reachable :: Location- > Location$$

the intended meaning of which would be to return the set of locations that are reachable from a location (this could also be expressed as a truth valued function of two arguments). The interpretation of path existence is certainly more complex than dealing with implicit predicates, but nonetheless it would be useful to be able to rely on the grounding function to provide this. It has already been demonstrated that an abstract version of the good compound rule can be automatically interpreted and used to guide search in a planner [12]. It is therefore proposed that this is a reasonable expectation of the grounding function.

We do realise that as the number of behaviours that are captured at the cluster level increases, more work will be involved in instantiating the functions and definitions and that this may counteract any timing benefits of using the rules. However, extending the toolkit of the generic control rule writer can only add to the rules that can be abstracted, and as a result, reused automatically.

We are also aware of the fact that recursive rules may introduce the danger of infinite recursion when evaluating their interpretations. This would only be the case in domains in which the evaluation of the recursive function was not defined, and could in fact be a useful analysis tool for debugging domains.

It was envisaged, during the progress of the work, that as more clusters are identified a library of generic control rules would be augmented by these new additions. So too it is with functions defined for existing clusters; if a new and useful definition or function is presented (such as the good compound definition), the prototype is extended accordingly.

However, this again underlines the need for a logic with which to identify and discuss features of a generic cluster (in terms of the properties and attributes of the members of types), as the grounding function must be told how to interpret functions. As has been stated previously, the grounding functions used in the current implementation are all hard-coded (with respect to generic types; the automatic identification of types gives appropriate grounding functions automatically). This proposed logic would allow us to formally define the functions declared in the prototype. This is an obvious direction for further work.

5.5.2 Orienteering cluster

An interesting exercise would be to implement the *orienteering* cluster, which would represent the locations and reward based actions at those locations as proposed by Smith [52]. This would show the applicability of the logic and control rules to behavioural structures other than those discovered by TIM. This cluster too would demand extensions to the language of control rules, not least as the reward based actions return values. This would require the generic control rule logic to be extended to handle quantitative values. The orienteering cluster could be a derived instance of the mobile cluster (see 4.3) and would have a prototype such as

```
prototype OrienteeringCluster {
  Types:  { Mobile, Location }
  Functions :  at:Mobile→Location
               value:Location→Number+
               link:Location→ {Location } }
```

where *Number* is a value.

This cluster would test any language proposed for relating functions to generic cluster features, as the mapping is not clearly defined (as in clusters identified by generic type analysis). In fact, the identification of the orienteering subproblem in [52] is based on a kind of sensitivity analysis of the extent to which propositions affect the applicability of actions required for goals. Interestingly, the type of control knowledge that we would construct for the orienteering cluster would be global constraints rather than local constraints (c.f. Section 4.6), and this too would suggest more language extensions. Smith uses the orienteering subproblem as an object specific goal ordering technique but object specific subgoals could not be expressed in the abstracted language of GCRs. For subgoals that were not object specific (such as choosing *whichever* location has the highest valued reward-function in Smith's orienteering subproblem) ordering could be imposed but would require new state arguments such as 'until' and 'eventually.'

5.6 Comparison with the control rules used by TLPlan

In this section we will consider the control information encoded in the TLPlan encoding of a particular domain (the *zenotravel* domain [39]) and see the extent to which that control information can be captured by generic control rules.

5.6.1 Generic structures in the domain

The *zenotravel* domain contains an instance of the *SafePortable* generic cluster. The planes are identified as carriers, the people are identified as safe portables and the cities are the locations between which the carriers move and at which the people can be situated. The *board* and *debark* operators are the *load* and *unload* operators for loading and unloading the safe portables. The fact that there are fuel levels associated with the carriers does not affect the identification of the cluster as a safe portable cluster.

5.6.2 Assumptions made in the TLPlan encoding

There are several assumptions implicit in the TLPlan encoding of the *zenotravel* domain that are closely tied to the embedded control information:

The carriers (planes) have unlimited capacity.

Restricting carriers' cargo to portables with like goal locations does not make the goals unreachable.

The map that the carriers move on is totally connected, and there is no difference in cost associated with travel between different cities. This means that there is no need for route planning or efficient use of carriers (e.g. changing planes en route to a goal destination). The fact that the map is totally connected is the reason it is safe to restrict the cargo of carriers to portables with like goal locations (see below), as every city is just one step away from any other. It should be noted that this simplifying assumption precludes the generation of optimal plans for domains in which the optimal plans *do* involve packages changing planes.

5.6.3 Embedded control information

The full TLPlan encoding of the *zenotravel* domain can be found in Appendix A. As can be seen, the control information is not explicitly stated as temporal control rules, but rather manifests itself in the *defined* predicates (there are also auxiliary predicates declared, such as the *scheduled* predicate, that Bacchus and Kabanza call *described* predicates). These defined predicates are then used in the operator definitions, not unlike precondition control. We will now consider the defined predicates and propose analogous generic control rules where appropriate.

5.6.4 Analogous control information

The *ok-to-board* predicate is used in the preconditions of the *board* action, has two arguments (the person, *?p*, and the plane, *?a*) and has two clauses:

- The person is not currently at their goal location.
- The plane is scheduled to fly to the person's goal location (this will be the case if there are other passengers on board that have the same goal destination) or is not scheduled to go anywhere (this will be the case if there is currently no-one on board).

The first clause can be covered by the expression

$$\forall_{object}\ Y : \mathcal{C}_{Location}$$
$$?p.at_{\mathcal{C}_{SafePortable}}(\text{NOW}) == Y \land \neg(?p.at_{\mathcal{C}_{SafePortable}}(\text{GOAL}) == Y)$$

It should be noted that the GCR versions of the control strategies are quantified over all objects of the correct type - this is analogous to the implicit universal quantification of operator parameters or predicate arguments in the domain description. The second case can be expressed without the use of the additional *scheduled* predicate, once we understand that that predicate is descriptive of something being aboard the plane. This gives us the expression

$$
\begin{aligned}
&\forall_{object}\, Y : \mathcal{C}_{Location} \\
&\qquad (\forall_{object}\, X : \mathcal{C}_{SafePortable}\; X.in_{\mathcal{C}_{SafePortable}}(\text{NOW}) == ?a \wedge \\
&\qquad X.at_{\mathcal{C}_{SafePortable}}(\text{GOAL}) == Y \wedge ?p.at_{\mathcal{C}_{SafePortable}}(\text{GOAL}) == Y) \\
&\quad \vee \\
&\qquad \neg\exists_{object}\, X : \mathcal{C}_{SafePortable}\; X.in_{\mathcal{C}_{SafePortable}}(\text{NOW}) == ?a
\end{aligned}
$$

In the TLPlan encoding, the predicate *ok-to-board* is formed from the conjunction of the two cases above. By quantifying the analogous generic control rules over all the appropriate objects in the domain, we avoid the need to conjoin them into a single expression.

The *ok-to-fly* predicate is used in the preconditions of both the *fly* and *zoom* operators (these are instances of the *move* operators for carriers). This predicate has two arguments (the plane, *?a*, and the destination of the flight, *?c*) and a non-trivial logical structure, in that it contains nested conjunctions and disjunctions. We will treat the clauses individually, under the assumption that generic control rule logic has similar logical operators and the clauses could be reconstructed into a compound expression. The clauses of the definition of the predicate express:

- The plane is not scheduled to fly anywhere else.
- The plane is scheduled to fly, and there are no additional people at the current location with the same goal location as those currently aboard the plane.
- All the people at this location are at their goal location, and there are

people at the plane's destination who are not currently at their goal location and there are neither planes at that location nor any scheduled to go there.

- The plane's destination is its goal location and there are no people that need to travel anywhere.

The first clause can be expressed by the expression

$$\neg\exists_{object}\ X : \mathcal{C}_{SafePortable}$$
$$X.in_{\mathcal{C}_{SafePortable}}(\text{NOW}) ==?a \wedge \neg(X.at_{\mathcal{C}_{SafePortable}}(\text{GOAL}) ==?c)$$

The second clause can be expressed, remembering the definition of the *scheduled* predicate, by the expression

$$\exists_{object}\ X : \mathcal{C}_{SafePortable}$$
$$X.in_{\mathcal{C}_{SafePortable}}(\text{NOW}) =?a$$
$$\wedge$$
$$\neg\exists_{object}\ X : \mathcal{C}_{SafePortable}\ \forall_{object}\ Y : \mathcal{C}_{Location}$$
$$X.at_{\mathcal{C}_{SafePortable}}(\text{NOW}) == Y \wedge X.at_{\mathcal{C}_{SafePortable}}(\text{GOAL}) ==?c \wedge$$
$$?a.at_{\mathcal{C}_{Carrier}}(\text{NOW}) == Y$$

The third clause is expressed as

$$\forall_{object}\ X : \mathcal{C}_{SafePortable}\ \forall_{object}\ Y : \mathcal{C}_{Location}\ X.at_{\mathcal{C}_{SafePortable}}(\text{NOW}) == Y$$
$$\wedge ?a.at_{\mathcal{C}_{SafePortable}}(\text{NOW}) == Y \rightarrow X.at_{\mathcal{C}_{SafePortable}}(\text{GOAL}) == Y$$
$$\wedge$$
$$(\exists_{object}\ X : \mathcal{C}_{SafePortable}\ X.at_{\mathcal{C}_{SafePortable}}(\text{NOW}) ==?c \wedge$$
$$\neg(X.at_{\mathcal{C}_{SafePortable}}(\text{GOAL}) ==?c))$$
$$\wedge$$
$$(\forall_{object}\ Z : \mathcal{C}_{Carrier} \neg(Z.at_{\mathcal{C}_{Carrier}}(\text{NOW}) ==?c) \wedge$$
$$\neg(\exists_{object}\ X' : \mathcal{C}_{SafePortable}\ X'.in_{\mathcal{C}_{SafePortable}}(\text{NOW}) == Z \wedge$$

$$X'.at_{\mathcal{C}_{SafePortable}}(\text{GOAL}) ==?c)$$

The final clause is expressed as

$$a?.at_{\mathcal{C}_{Carrier}}(\text{GOAL}) ==?c$$
$$\wedge$$
$$(\forall_{object}\ X : \mathcal{C}_{SafePortable}\ \forall_{object}\ Y : \mathcal{C}_{Location}\ X.at_{\mathcal{C}_{SafePortable}}(\text{NOW}) == Y$$
$$\rightarrow X.at_{\mathcal{C}_{SafePortable}}(\text{GOAL}) == Y)$$

The *need-to-fly* predicate is only used in the definition of the *ok-to-refuel* predicate. This predicate has one argument (the plane, $?p$) and together with the *ok-to-refuel* predicate makes sure that a plane is only refuelled if it needs to fly somewhere. The definition of this predicate has clauses that express the following statements:

- The plane is scheduled to fly somewhere or someone has boarded the plane (these both describe the same property).
- There is at least one person who needs to be taken to their goal location, and there is neither a plane at their current location nor any scheduled to go there.
- The current location of the plane is not its goal location.

The first clause can be expressed by the generic control rule

$$\exists_{object}\ X : \mathcal{C}_{SafePortable}$$
$$X.in_{\mathcal{C}_{SafePortable}}(\text{NOW}) ==?a$$

The second clause can be expressed as

$$\exists_{object}\ X : \mathcal{C}_{SafePortable}\ \forall Y : \mathcal{C}_{Location}$$
$$X.at_{\mathcal{C}_{SafePortable}}(\text{NOW}) == Y \wedge \neg(X.at_{\mathcal{C}_{SafePortable}}(\text{GOAL}) == Y)$$

$$\begin{array}{l}\wedge \\ \quad \forall_{object}\ Z : \mathcal{C}_{Carrier}\ \neg(Z.at_{\mathcal{C}_{Carrier}}(\text{NOW}) == Y) \\ \quad \wedge(\neg\exists_{object}\ X' : \mathcal{C}_{SafePortable}\ X'.in_{\mathcal{C}_{SafePortable}}(\text{NOW}) == Z \\ \quad \wedge X'.at_{\mathcal{C}_{SafePortable}}(\text{GOAL}) == Y)\end{array}$$

The third clause can be expressed as

$$\begin{array}{l}\forall_{object}\ Y : \mathcal{C}_{Location} \\ \quad ?p.at_{\mathcal{C}_{Carrier}}(\text{NOW}) == Y \rightarrow \neg(?p.at_{\mathcal{C}_{Carrier}}(\text{GOAL}) == Y)\end{array}$$

The TLPlan encoding contains one more defined predicate, *ok-to-refuel.* We have already offered a generic control rule analogous to the defined predicate, *need-to-fly*, used in the definition of *ok-to-refuel.* However, this predicate references the fuel level of the plane. Within the safe portable cluster, there is no mention of fuel levels of carriers. In order to express this part of TLPlan's control strategy, we have three options. We could use a derived instance of the safe portable cluster in which carriers' fuel *is* taken into account (or conversely a derived instance of a fuelled mobile cluster in which safe portables are identified), we could construct a rule that uses *both* the safe portable cluster *and* a fuelled mobile cluster, or we could allow ourselves to supply additional rules by hand on top of those generated automatically. We do not present alternative clusters here for simplicity of the analogy, as the cluster in the existing analogy (the $SafePortable$ cluster) has been well introduced in previous examples in this thesis. Remarks on supplying additional information by hand are contained in Section 5.6.5.

The clauses in *ok-to-refuel* that do exploit relationships described by the safe portable cluster could be expressed in a similar fashion to those described above. In fact, there a lot of repetition in the individual clauses (with respect to the other defined predicates), a point that is discussed below.

A final and important remark is that the analogous control rules presented would would *totally* describe the control strategies of a similar TLPlan encoding where fuel levels of the vehicles were *not* considered (as in the classic *logistics* domain).

5.6.5 Differences between the alternative styles of control information

One of the main differences between the defined predicates as used in the TLPlan encoding and the GCRL (generic control rule logic) rules is the way that the defined predicates are used as preconditions in the standard operators for the domain. It is envisaged that the control rules will be used to actively guide search (though the use of generic control rules in precondition control is suggested in Section 6.6) and the differences between this and the role of passive preconditions raises some interesting points.

Firstly, the defined predicates (and their translation into GCRL) contain quite an amount of repetition. This is because the defined predicates are written as preconditions to specific actions, with some control strategy in mind for that particular action. For example, the second main clause in both *ok-to-fly* and *need-to-fly* performs the same check - namely that there are some SPOs at some location (not their goal location) and there are neither carriers at that location nor carriers heading to that location. The fact that each defined predicate is written for one action means that any information that two actions need must be repeated. We observe that while translating the defined predicates gives a subset of analogous control rules, a more concise set of control rules could be constructed by disregarding the structure imposed by those defined predicates.

This point also has consequences when we consider the problem of conflicting control advice. If the preconditions (defined predicates) for different actions contain conflicts or offer similar but slightly different advice, then this is not necessarily a problem. The reason this is not a problem is that as a precondition, where the specified condition does not hold, the operator will simply not be applied. However, once we translate the defined predicates into GCRL, we have a different situation. The GCRL rules tell us what will to be true in the next state, according to certain conditions that hold in the current state. If there are any conflicting courses of prescriptive action then we *may* end up with a contradiction in the specification of the next state (according to the control rules) which will need to be resolved.

One method, as employed in certain places in the defined predicates used above, is to provide disjunctive advice. This allows the control rule to suggest alternative courses of action, typically of the form 'do x or maintain the current situation.' This would allow the system using the rules to decide whether to take the action suggested (x) or wait and take that action at a later date (by maintaining the situation, the antecedent of the rule will be matched again in the subsequent state). This approach can also be used to force the resulting plan to be linear, whereby one of any pair of concurrent actions is delayed while preserving the preconditions required for its applicability.

The defined predicates used by TLPlan are unrestricted in referring to any predicates in the domain, whereas the analogous GCRs only have access to the relationships declared in the prototype of the cluster being used. The discussion of the fuel level of the vehicles above relates to this observation. However, the advantage with using GCRs is that the control information can be reused, not just when we encounter this particular domain again, but *whenever we encounter a domain with similar generic structure, under the same assumptions.*

With respect to giving a full analogy to the *ok-to-refuel* predicate, one option mentioned was to supply the additional strategies by hand on top of those strategies generated automatically. This suggestion, though seemingly contrary to the direction of this thesis, is easily justified. The performance of TLPlan is driven by the control information supplied. For such control rule dependent planners, a large portion of the work involved in producing control rules could be taken care of by automatic instantiation from generic control rules. The addition of *some* rules by hand to augment those automatically supplied involves less work than the construction of a full set of rules. It should also be noted that, where automatically generated rules are used by a planning system whose performance is *not* dependent on control information (such as in conjunction with a fully automated planner), that planning system will reap the benefits of any control strategies that are supplied and default to its natural behaviour in situations where none of those strategies are appropriate.

The temporal logic of TLPlan allows the expression of described predicates. This is an addition that could be made to GCRL without significant complications to the semantics. This addition to the language was also suggested by the discussions in Section 5.5.1.

5.7 Chapter Summary

Chapter 5 has presented various results which constitute a proof of concept of the thesis. These results take several forms.

Firstly, timing results were presented to show the time taken to instantiate control rules in a range of domains. Included in these domains were those in which the control rules *could not* be instantiated, these results demonstrating that no significant time penalty was incurred by the instantiation machinery when the appropriate generic clusters were not recognised. The timing results for the domain in which the control rules *could* be instantiated gave an indication of the time cost associated with the process of automatically instantiating those control rules. All of the timing results were compared with the running timeS of TIM on the same problems. This provided a measure of how much additional time was incurred by control rule instantiation over and above the analysis that enabled it.

Secondly, a method was described for integrating automatic control rule instantiation with an existing planning algorithm, namely FF, and subsequently using the domain specific strategies generated to influence the search for a solution. This successful integration demonstrated that generic control rules can be used by the community, and specifically that they can be used to supplement existing algorithms and techniques. Some preliminary results have shown an improvement in solution length with the addition of domain specific strategies automatically instantiated from generic control rules.

Thirdly, the utility of generic control rules in providing abstractions of strategies relating to other known subproblems from the literature was discussed.

Finally, a comparison was made between control strategies employed by TLPlan and analagous control strategies given as generic control rules. These results have shown that useful control information that is currently used by control rule based planners can be expressed at an abstract level using generic control rules.

Floors	Passengers	Average running times			Average running times minus average TIM running times		
		1 Rule	2 rules	3 rules	1 Rule	2 rules	3 rules
2	1	0.016	0.16	0.017	0.002	0.002	0.002
4	2	0.016	0.015	0.017	0.002	0.001	0.002
8	4	0.018	0.017	0.017	0.002	0.001	0.001
16	8	0.023	0.023	0.024	0.002	0.002	0.003
32	16	0.075	0.076	0.075	0.005	0.006	0.005

Table 5.1: Table showing average running times for abstract control rule instantiation system for randomly generated *miconic* – *STRIPS* domain problems

Locations	Max fuel	Spaces	Vehicles	Cargo	Average running times			Average running times minus average TIM running times		
					1 rule	2 rules	3 rules	1 rule	2 rules	3 rules
2	1	1	1	1	0.0194	0.0202	0.0185	0.001	0.002	0
4	2	2	2	2	0.0195	0.0199	0.02	0.001	0.001	0.001
8	4	4	4	4	0.0238	0.0228	0.0231	0	-0.001	-0.001
16	8	8	8	8	0.0424	0.0409	0.039	-0.002	-0.003	-0.005
32	16	16	16	16	0.3017	0.3032	0.2987	0.003	0.005	0

Table 5.2: Table showing average running times for abstract control rule instantiation system for randomly generated *mprime* domain problems

Planes	Cities	Size of cities	Packages	Average running times			Average running times minus average TIM running times		
				1 rule	2 rules	3 rules	1 rule	2 rules	3 rules
1	1	1	1	0.018	0.02	0.022	0.004	0.007	0.008
2	2	2	2	0.018	0.02	0.024	0.004	0.007	0.009
4	4	4	4	0.022	0.025	0.027	0.004	0.005	0.009
8	8	8	8	0.131	0.133	0.135	0.005	0.005	0.009
16	16	16	16	9.514	9.497	9.902	0.042	0.062	0.43

Table 5.3: Table showing average running times for abstract control rule instantiation system for randomly generated *logistics* domain problems

```
(define (domain lukeslogistics-strips)
   (:requirements :strips)
   (:predicates (obj ?obj)
                (truck ?truck)
                (location ?loc)
                (in ?obj ?truck)
                (at ?obj ?loc)
                (link ?loc1 ?loc2))
(:action LOAD-TRUCK
   :parameters
                (?obj
                ?truck
                ?loc)
   :precondition
                (and (obj ?obj) (truck ?truck) (location ?loc)
                (at ?truck ?loc) (at ?obj ?loc))
   :effect
                (and (not (at ?obj ?loc)) (in ?obj ?truck)))
(:action UNLOAD-TRUCK
   :parameters
                (?obj
                ?truck
                ?loc)
   :precondition
                (and (obj ?obj) (truck ?truck) (location ?loc)
                (at ?truck ?loc) (in ?obj ?truck))
   :effect
                (and (not (in ?obj ?truck)) (at ?obj ?loc)))
(:action DRIVE-TRUCK
   :parameters
                (?truck
                ?loc-from
                ?loc-to)
   :precondition
                (and (truck ?truck) (location ?loc-from) (location ?loc-to)
                (at ?truck ?loc-from) (link ?loc-from ?loc-to))
   :effect
                (and (not (at ?truck ?loc-from)) (at ?truck ?loc-to)))
)
```

Figure 5.2: Domain description for simple logistics style domain

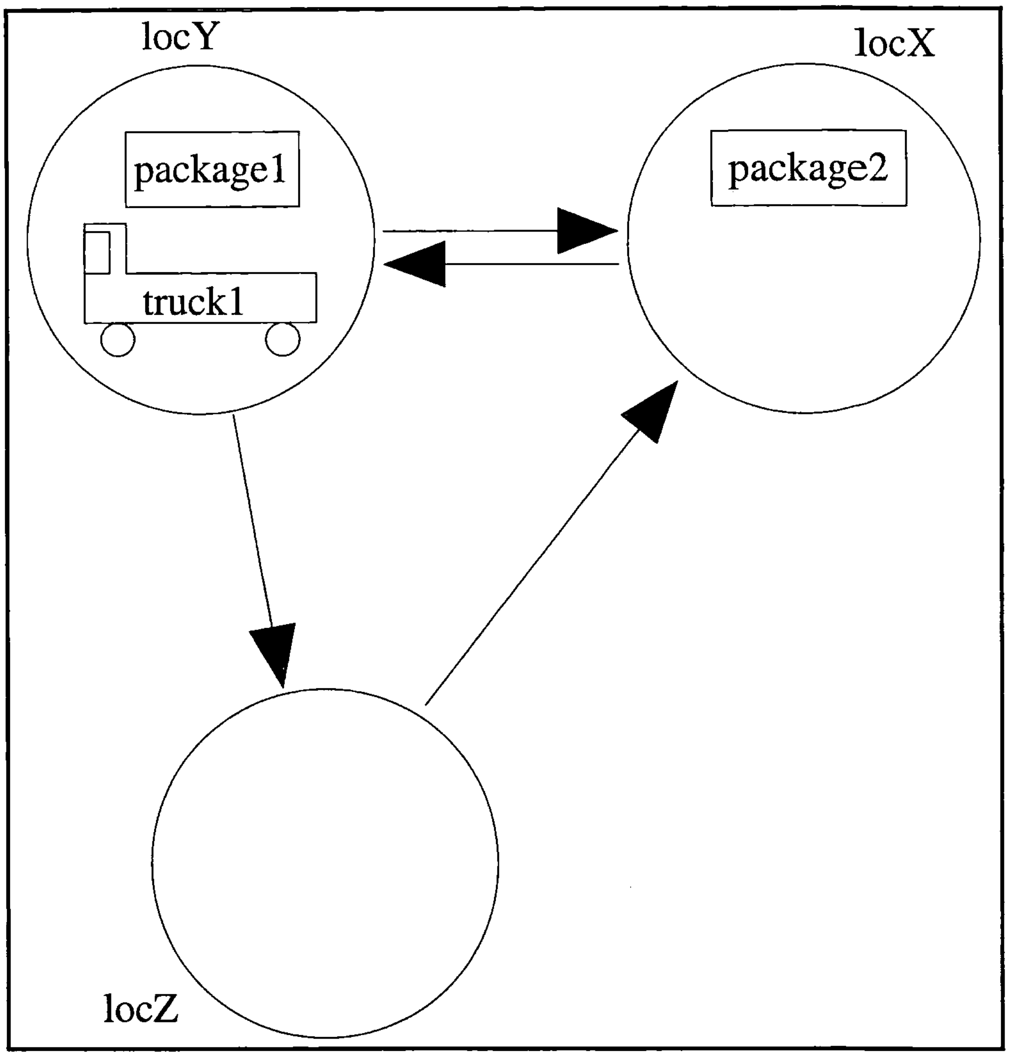

Figure 5.3: Problem 1

Problem	Average running times			Solution length	
	FF++	FF++ (instantiation only)	FF	FF++	FF
1	0.11	0.02	0.01	7	8
2	0.29	0.02	0.01	7	8
3	4.30	0.02	0.01	11	13

Table 5.4: Table showing average running times for abstract control rule instantiation system and FF for constructed simple logistics domain problems

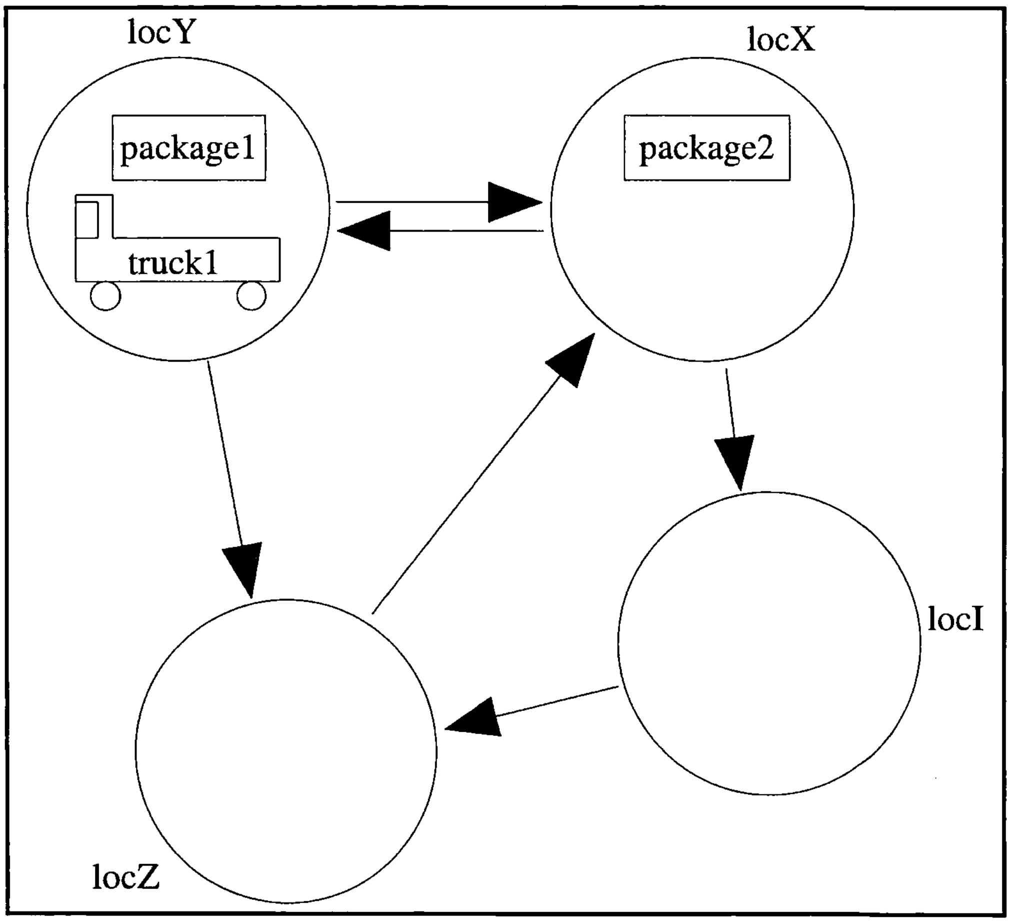

Figure 5.4: Problem 2

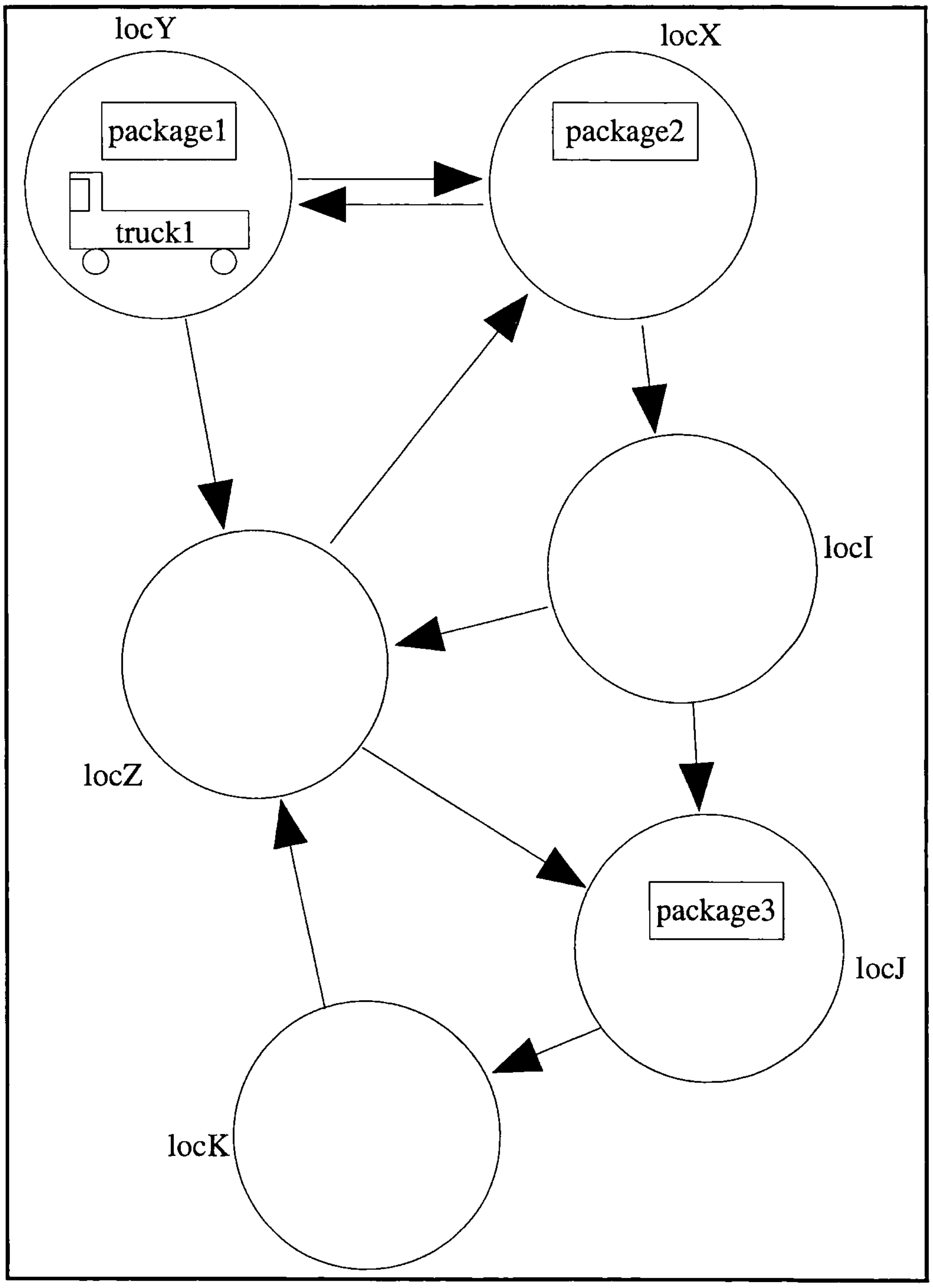

Figure 5.5: Problem 3

Chapter 6

Conclusions

The varied nature of the results presented in conjunction with the literature enable us to draw several conclusions and also suggest directions for further research.

6.1 Time penalty

The generic control rule module uses the already established TIM system. This system delivers the results of various forms of domain analysis which have been shown to aid search in planning algorithms. The time penalty paid for using the GCR module incorporates the time taken to conduct the TIM analysis. More than that, the running times for the GCR module have been shown to be dominated by the TIM analysis; the time spent instantiating the control rules is only a few percent of the total running time of the stand alone control rule instantiation system. This observation can be interpreted in a number of ways.

It could be argued that whenever the decision is made to use the results generated by TIM, no significant additional penalty is incurred by also taking advantage of GCRs. Admittedly, further experimentation may show that with vast numbers of rules available for instantiation, the additional time penalty may be more significant (see Section 6.6 for discussion of this). However, it has been shown that some rules used with great success by systems such as

TLPlan and TALPlanner can be instantiated automatically with minimal cost over and above the basic TIM analysis. This is a very positive and novel contribution of the work.

We cannot say for certain that the process of identifying the behaviour structures that allow generic control rules (generic clusters) is more costly than the process of using those results to instantiate template rules. This is due to the fact that TIM performs many forms of analysis other than the generic type identification that enables generic cluster identification (such as invariant generation); the timings of the TIM system on the problem sets incorporate all these analyses.

We can conclude that when the decision to use TIM to perform pre-planning domain analysis is made, we can also supply a collection of generic control rules (that have been shown to improve search in control rule based planners) that can be automatically instantiated into domain specific control rules. This additional reasoning is added with a small time cost relative to the TIM algorithm.

The time results presented in Section 5.4.1 are not presented to show any competitive performance. As can be seen, the additional time penalty for integrating generic control rules into FF grows rapidly even on this simple example domain. These results were included to describe fully the work undertaken, and require some explanation. As has been stated previously, the integration of domain specific control reasoning into FF has been implemented for proof of concept only, and as such adopts the very naive strategy of fully grounding control rules. The set of fully grounded control rules (subject to some restrictions, such as removing those that refer to goal states other than the goal of the current problem) were then used to verify whether proposed extensions to a partial plan adhere to control restrictions.

It is these processes of full grounding and then exhaustive verification that dominate the increase in running time that is seen in this problem set. We can see that the instantiation only version of the generic control rule system takes only two hunderedths of a second to generate domain specific rules quantified over domain types. Compared to the 4.3s that FF++ takes to generate a solution to problem 3, the time taken to instantiate the rule is not significant. We can see that this performance decrease must be incurred by the processes described, though unfortunately we can not break down this penalty further (by considering time to fully ground versus increased cost of searching).

It is worth noting that we consider the utility of generic control rules independently of these timing results. This point is discussed further in Section 6.4. The timing results are included here to demonstrate that the generic control rule module can be successfully integrated with existing planners, whether or not those planners were designed to accept control knowledge. We consider it an achievement that one abstracted rule has been used to improve the plan quality of FF, irrespective of the naive grounding and verification techniques employed.

Another point of note is that the tests were performed on, by current standards, a relatively low-spec machine. It was decided that as the tests were designed neither to showcase running times nor to tackle massive problems, a humbler machine was satisfactory (the machine is significantly less powerful, for example, than the machines used to run systems in recent planning competitions [39]).

6.2 Utility

The utility of generic control rules is supported by the literature. Control rules that are used by current control rule based planners can be expressed as generic control rules and automatically instantiated into domain specific forms (c.f. Sections 5.5 and 5.6). We are not suggesting that generic control rules currently match the power of hand coded control knowledge. What we are presenting is a method of representing a subset of control rules based on an abstraction of the behaviour of objects in domains (the generic cluster). This allows the expression of reusable forms of control knowledge that can be automatically instantiated. We have shown that useful control strategies can be abstracted in this manner and successfully instantiated automatically into domain specific rules.

We have indicated how to implement an abstraction of the powerful 'good tower' heuristic within the framework described. This involved the outlining of a prototype for the newly proposed safe construction cluster. Although no data is presented showing the utility of this heuristic, examples can be found in the literature. We discuss an abstracted version of the rule that would be available for use whenever the appropriate behaviour structure is identified.

Although we have not demonstrated that large numbers of rules can be generated with reasonable time cost, this does not detract from our findings. The fact that we can capture useful control rules is in itself justification that the process is beneficial, as those control rules can be abstracted and instantiated automatically.

The results presented in Section 5.4.1 describe the successful integration of the control rule module and an existing planning system. The results show that with the addition of an appropriate generic control rule and supporting instantiation and search restriction machinery, the solution quality of the state of the art planner FF can be improved. When we say that the quality has been improved, we mean that the length of the solution generated is closer to the optimal plan length. This result has been demonstrated in a small domain, in which problems were set to highlight the use of a particular rule.

The reason that control rules can improve plan quality in FF in particular is to do with the way that FF constructs its plans. FF uses a very powerful combination of a relaxed plan estimate of distance from the goal state and helpful actions (c.f. Section 5.3.1) to extend partial solutions. However, this approach can produce non-optimal solutions. FF extends a partial solution with the first action it considers that has a better heuristic evaluation (according to some restrictions based on the identification of helpful actions) than the current state. Because the relaxed plan estimate is based on a relaxation of the problem that ignores the delete effects of actions, the search space of the relaxed plan does not necessarily directly correspond with the entire problem. Thus the ordering of actions in the relaxed plan may give a misleading impression of the required ordering of actions in the top level plan. The use of generic control rules adds to FF's plan construction phase by pruning proposed extensions to partial plans according to domain specific control knowledge.

Let us consider an example of this process in detail, using problem 1 from Section 5.4.1. FF's own solution is sub-optimal as it accomplishes locating *package*1 at location Z before picking up *package*2. The reason it selects this course of action is the relaxed plan from the state where *package*1 is being carried by *truck*1 at location Y. From this state, the relaxed plan with the best heuristic estimate is:

$drive(truck1, Z)$ and $drive(truck1, X)$
$unload(package1, truck1)$ at location Z and $load(package2, truck1)$ at location X
$drive(truck1, Y)$from location X
$drive(truck1, Z)$from location Y
$unload(package1, truck1)$

FF will construct a helpful actions set of $\{drive(truck1, Z), drive(truck1, X)\}$, from which it will select the first action it considers that gives a better heuristic estimate than the current state. As both of these actions satisfy this condition, the action selected will be whichever is first in FF's own internal set.

The inclusion of domain specific control rules into FF prunes those states from this set that contravene the control rules. The domain specific control rule instantiated from the generic control rule described in Section 5.3.3 is

$$\begin{gathered}\forall g:2\ \forall e:2\ \forall d:2\ \forall C:0\ \forall X:1\ \forall Y:1\ \exists z:1\ \exists h:2\\ ((link(D,E)\wedge in(X,C)\wedge at(C,D)\wedge at(Y,E)\wedge GOAL\ at(X,g)\wedge\\ GOAL\ at(Y,g))\rightarrow(GOAL\ at(z,g)\wedge NEXT\ at(C,h)\wedge NEXT\ at(z,h))\end{gathered}\tag{6.1}$$

where the types identified are $0 = \{truck1\}$, $1 = \{package1, package2\}$ and $2 = \{X, Y, Z\}$. Application of this rule has the effect of pruning the action $drive(truck1, Z)$, which once removed from the set leaves only the action $drive(truck1, X)$. FF follows this course, and the resulting solution length is one step shorter.

6.3 The bigger picture

Generic control rules provide the community with a method of expressing abstracted control structures. We have shown that many widely accepted control strategies can be expressed in this framework. The fact that generic control rules can be automatically instantiated into domain specific rules means that they are not only reusable (saving the time and effort of control rule authors) but may also be *communal*; the community can share abstracted control rules and even libraries of control rules.

Generic control rules, however, are not being billed as a panacea. It is widely accepted that writing good control strategies for a particular planning system working on a known problem is a non-trivial task. The control rule author must have a good knowledge of both the domain and the language being used to capture control. He may, in addition, need to know the mechanics of the planning algorithm intimately. Even then, the quality of the control rules produced is reliant on the author's ability to identify useful strategies. Generic control rules allow the reuse of control knowledge, but it is not claimed that authoring them is any simpler than constructing domain specific examples (although by their nature, abstract control rules will not be tied to the mechanics of one planning algorithm). In fact, writing control rules at the abstract level could be considered a harder task than that of writing domain specific instances.

We are not, however, presenting generic control rules as a method of facilitating the layman in successfully managing a planning system. We accept that the current state of technology dictates that the field of planning is still largely academic and that only people with substantial knowledge with respect to the technology can use it (although some commercial systems do use planning technology, such as BridgeBaron [53]). This statement is all the more true if we consider the more technical tasks, such as constructing control strategies. It is the opinion of the author that this will remain the case for some considerable time, due to the nature of the process. What generic control rules *do* offer is a way to write, share and compare reusable abstract control strategies, though we readily admit that it will be people already in the field that will benefit from this. There is motivation in the community to apply planning technology to real world applications. Through the use of domain specific control rules, planning systems are producing better plans, faster. Reusable control knowledge can only add to this trend of increase in performance and may help move planning technology into practical applications.

6.4 Other issues

The claims of improved performance are based on solution length and not running times. This point highlights some interesting observations that might otherwise have gone unnoticed. When we supply control information to a planning system, we are affecting the way it searches for a solution in the search space. This can be through ordering, branch pruning and other forms of search space restriction. The desired effect is that plans will be found that have qualities we specify (through control knowledge) and that they *may* be found faster than without control knowledge (restricting the search space *may* lead to reduced search time).

The control rules used in the results were optimality rules. We would expect them to improve plan quality by pruning sub-optimal branches from the search space. As can be seen from the results, generic control rules can be used to improve plan quality (although they will not necessarily do so for all problems - control rules were used to enforce optimality constraints that the planning algorithm did not). In fact they improve plan quality in the very successful fully automatic system, FF. Solution generation times were not improved, but as has been discussed the implementation handled the control information in a naive way. There is also no guarantee that optimality constraints reduce search time, as they may make it more difficult to find a solution (in domains where non-optimal solutions are very dense and optimal solutions are very sparse).

There is comfort in the solution generation times for the instantiation only system, though this result is harder to qualify. The fact that the control rule instantiation is very quick with respect to the full TIM analysis is positive, and there is the possibility that a bare-bones version of TIM could be constructed that only put effort into identifying generic clusters with the intention of speed-ing this part of the process up. An interesting observation is that to compare the running times for the instantiation only system with the state of the art in control rule generation, we would need to time human control rule authors producing the same set of rules!

6.5 Contributions

The novel work presented in this thesis can be summarised by the following list of contributions to the field:

Demonstration of the thesis that useful control strategies can be represented in a language that uses behaviour based abstractions of interacting types and automatically instantiated into domain specific rules.

The proposal of the generic cluster as a basis for writing reusable, abstract control knowledge. Within this definition, cluster prototypes are presented as declarations of the available features of a cluster. The notion of a generic cluster is an extension of the idea of generic types and provides, we feel, a more powerful way of exploiting interacting generic behaviours. Where generic types abstracted the types involved in a subproblem, the generic cluster gives a behaviour based abstraction of the whole subproblem. The isolation or integration of subproblems is discussed using the novel terminology of inclusive and exclusive instances of generic clusters.

An extension to the domain analysis package offered by TIM, which to date had offered the results of various analyses (object typing, invariants, etc.). Generic control rules provide an interface for writing reusable abstracted control rules for the behaviour structures that TIM identifies. As such, the work contained in this thesis can be seen as providing a rational reconstruction of the processes of abstraction and reuse of control knowledge, as seen in the hard coded subproblem specific control strategies employed by the TIM/STAN partnership. The ability to provide abstract control strategies affords us many benefits. Some of the work provided by control rule authors can be fully automated, removing the problems associated with manual domain analysis and control rule formulation. They also provide a basis for the shared use of control information across the community.

A modal temporal logic for expressing properties of sequences of states. The logic uses the features of the behaviour based generic cluster to express configurations of objects over states. The logic itself is novel as it departs from the tradition of qualifying the temporal mode of propositions or expressions and instead considers the temporal mode of an object's relationship with some other known object or objects. None of the logics previously presented for the expression of control strategies have been able to express meta-constraints on the search space that the use of generic clusters enables (the languages used by both TLPlan and TALPlanner can only offer object level rules and as such can only offer object level control rules). A proof that the normal form of the generic control rule logic is the the same as the domain level language.

Direction on how to construct prototypes for a range of generic clusters, including a cluster relating to a subproblem that is *not* identified by the established generic type identification process. Demonstration of some natural, possibly necessary, extensions to the languages proposed. This includes the declaration of definitions (truth valued generic functions) such as *good compound*, the interpretation details of which are also presented.

Demonstration of the need for an additional language, the features of which have been described and discussed, for identifying and defining the features of a generic cluster in terms of the behaviour of the types involved. As has been stated, the vocabulary of this language would be the structures that enable the identification of generic types, i.e. properties of varying kinds, attributes, etc.. A question has been raised concerning the possibility of other such languages, potentially with identification techniques in addition to the established relational behaviour-based analysis (as would be required by the identification of the orienteering subproblem by Smith [52]).

Demonstration not only that behaviour based abstract rules can be automatically instantiated, but can also be used to aid the search process of a state of the art planning system that was not originally intended for such additional information. An example of the pruning that can be achieved is demonstrated. Timing results show that the additional work carried out by the generic control rule instantiation module is not significant with respect to the TIM analysis that is used to analyse the domain.

6.6 Further work

The work presented in this thesis suggests many directions for further research. Here we discuss some possibilities.

Although the machinery for automatic instantiation has been implemented, a more comprehensive library of rules for a range of generic clusters would give a clearer picture of the impact of using abstracted control rules. Building a library of applicable rules (that are not subsumed by the search strategy using them) would allow us to consider how the benefits of using large sets of rules compare with the increased time penalty paid for instantiating those rules. As new generic clusters are identified, the library can be extended to handle not just new rules but also new clusters.

A more comprehensive library would also allow control rule based planners to be evaluated on a level playing field. Giving different planners the same control information would facilitate fairer comparisons between these systems. This is in contrast to the international planning competitions, in which the performance of control rule based planners is dependent on the work done by human control rule authors in the available time.

We have not claimed that abstracted control rules can currently match the power of hand coded domain specific control information. We have shown that a subset of the rules that currently must be manually generated can be automatically instantiated from abstracted forms. This suggests a tool that could be used in conjunction with other forms of control rule generation that would be of use not just to those running planning systems but also to domain engineers (for domain verification purposes). An obvious inclusion in this tool would be the availability of both completeness-preserving rules *and* non-completeness-preserving rules. This would allow the selection of rules that introduce incompleteness into the search space on a per problem basis (for example, a control structure that is used to find *some* solution very quickly may be selected if we are more concerned with time performance than plan optimality).

A larger collection of rules for a range of generic clusters would allow the examination of any increase in running times entailed. If a significant time penalty is incurred by large rule sets, one way in which the work could progress is through the use of rule priorities. Rules could be rated in terms of their utility (with respect to the appropriate cluster), allowing the planner to select a subset of the rules available. This approach could also be applied in domains with multiple generic clusters, where only the best rated rules are selected irrespective of the cluster they exploit.

Abstract control rules were used in conjunction with a forward-chaining state based planner, and though this type of algorithm lends itself naturally to the pruning enabled by control rules, a dedicated control rule based planner could use the control rules in a less naive way (control rules were fully grounded in the implementation). FF's powerful heuristics meant that many rules that were tried were overshadowed by the natural search strategy of the planner. The full effect of pruning the search space could be seen with planning systems employing brute force search.

As has been described, it is possible to envisage declaring a prototype for a cluster that has not been identified by the deterministic behavioural analysis that TIM offers. The orienteering subproblem in planning domains described by Smith is just such a problem. Although we can quite easily construct a prototype for this subproblem (a generic cluster is really a recognised subproblem), it is not obvious how we would provide the appropriate binding function. In the work presented, the binding function is supplied automatically by TIM. We have demonstrated the need for a new language that would be used to define the functions of a generic cluster using the same vocabulary as that of generic type identification. However, the identification of the orienteering problem is not a deterministic behaviour based analysis. It involves a sensitivity analysis where results are compared with some user defined threshold. The proposed language for relating function definitions to behaviour aspects may not be appropriate for such behaviour structure. This seems to suggest that new logics would be needed to supply the definitions of functions for behaviour structures that are identified in different ways. We suspect that in reality, some base language (such as that outlined earlier) would be sufficient and could simply be extended to cater for structures that are identified in novel ways. However, this is an interesting area for exploration.

Extensions in line with the development of the domain description language. By this we mean the hand-in-hand development of the TIM/generic control rule module partnership to cater for aspects being introduced into the common language of the community such as time, durative actions, numerical functions, etc..

Much recent work has focussed on the automatic generation of control knowledge, be that in the form of goal orderings or policies. We suggest that the framework of generic clusters for discussing features of a subproblem may be useful in such control generation. For example, [19] uses a random walk strategy and approximate policy iteration to learn and refine policies. It is conceivable that the random walk strategy could be used to learn policies relating to the subproblem represented by the cluster by constructing a state space relating to the pure subproblem. This would improve on the policy generation described in [19] as the policies would be abstracted; the system would only have to learn the policy once but could apply it to every instance of the abstracted subproblem. This technique might not be so applicable to some forms of control knowledge, such as extracting goal orderings, as these strategies are problem specific and hence not abstractable with respect to the subproblem. This suggests the classification of control knowledge as either *problem specific* or domain specific, where problem specific control knowledge can only be guaranteed to be useful for the particular problem instance and not for any problem in general in the domain. Only domain specific control knowledge could be abstracted to the cluster level.

Generic control rules that refer to states beyond the immediate successor of the current state require a progression algorithm, or compiling into the operator preconditions (a compilation technique that is proven to be equivalent to TLPlan's progression algorithm is given in [23]). Implementation of either of these would allow control rules to specify longer sequences of states, and potentially offer new pruning opportunities. The compilation of control rules into preconditions simplifies the search space associated with the domain, and with an automatic compilation algorithm [23], the overhead of manually compiling those control rules is removed.

Considering the applicability of abstracted control rules demanded the classification of instances of generic clusters, rules for generic clusters and derived instances of base clusters as exclusive or inclusive. We also considered and discussed the relationships between hierarchies of generic clusters. Along with discussions on the completeness preservation of rules, these issues are all concerned with the appropriateness of control knowledge. Although many observations were made, these areas would benefit from further examination.

Chapter 7

Summary

We have successfully demonstrated the thesis of this work. This demonstration has involved many aspects.

We have extended the notion of generic types as abstractions of domain types with recognised behaviour to the notion of generic clusters as abstractions of sets of domain types that interact in known ways. This provides an abstraction of subproblems in planning domains. Generic cluster prototypes have been presented to encapsulate the behaviour of any given generic cluster.

A state based temporal logic has been specified, whose terms are features of generic cluster prototypes. This allows the description of sequences of states in terms of the features of generic clusters. Describing sequences of states provides a method of expressing control knowledge (properties that must hold over a sequence of states). Control knowledge expressed at this abstracted level is reusable.

Existing domain analysis techniques are used to automatically identify instances of generic clusters in planning domains. The details provided by generic cluster identification supply the information required to specialise any expression given in terms of that abstraction. This facilitates the automatic instantiation of abstract control rules into domain specific instances, removing the need for those rules to be manually constructed.

We have shown that abstractions of many control strategies that are employed by current control rule based planners can be expressed and automatically instantiated in the framework described. Furthermore, we have demonstrated that automatically instantiated control strategies can be used to improve plan quality in a competitive contemporary fully automatic planning system. The time penalty incurred by the provision of these rules is small compared to the TIM analysis on which that provision is based.

The specification of a logic for abstracted control knowledge provides the community with a common language in which to share, compare and compile control strategies. A repository of reusable control rules will reduce the onus of control rule authors and provide a level playing field on which to compare the performance of control rule based planning systems.

In addition to introducing the generic cluster as an abstraction of subproblems (through the representation of interacting sets of types) we have discussed the relationships that exist between derived instances of generic clusters and their base clusters. This discussion introduced the classification of derived instances of a cluster as exclusive or inclusive, a classification that was also introduced both with respect to control rules for generic clusters and instances of a cluster in a domain. These classifications were necessary in specifying the applicability of control strategies within the generic control rule framework.

The methods detailed in this thesis have been justified using both empirical data and their context in the field of planning. They have also set the foundations for many further avenues of enquiry.

Appendix A

The ZenoTravel domain

This is the complete TLPlan encoding of the ZenoTravel domain

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; ZenoStripsWorld.tlp
;;; TLPlan Zeno travel world.
;;; Based on the 2002 PDDL strips domain
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Initialization
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(clear-world-symbols) ;Remove old domain symbols
(enable pddl-support)
(set-search-strategy depth-first-no-backtracking)
(disable cycle-checking)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 1. The world symbols.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(declare-described-symbols
  (predicate at 2) ;domain predicates
  (predicate in 2)
  (predicate fuel-level 2)
  (predicate fuel-next 2 no-cycle-check)

  (predicate aircraft 1 no-cycle-check)
  (predicate person 1 no-cycle-check)
  (predicate city 1 no-cycle-check)

  (predicate scheduled 2)) ;?plane is scheduled to travel to ?city

(declare-defined-symbols
  (predicate ok-to-board 2)
  (predicate ok-to-fly 3)
  (predicate ok-to-refuel 1)
  (predicate need-to-fly 1))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 2. The defined predicates.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; OK for person ?p to board plane ?a

;; Don't board a plane unless we need to go somewhere else
;; We can board this plane either if it's not scheduled to go anywhere
;; or it's scheduled to go to our goal city

(def-defined-predicate (ok-to-board ?person ?aircraft)
  (and
    (forall (?city) (at ?person ?city) (not (goal (at ?person ?city))))
    (forall (?city) (scheduled ?aircraft ?city)
                    (goal (at ?person ?city)))))

;; OK for plane ?a to fly to city ?c

;; We can fly if we're scheduled and there is no one else who needs
;; a ride to our destination.
;; Or, there is no one here who needs to go somewhere else and there
;; are people at our destination and no other plane is there or no other
;; plane is scheduled to go there.
;; Or we need to get to our goal location and every person is at
;; their goal

(def-defined-predicate (ok-to-fly ?a ?c ?f) ;; ?f is dummy argument
  (and
    (forall (?c2) (scheduled ?a ?c2) (= ?c ?c2))
  (or
    (and
      (scheduled ?a ?c) ;; we're scheduled
      (forall (?c2) (at ?a ?c2) ;; no one else here going our way
      (not
        (exists (?p) (at ?p ?c2)
          (and
            (person ?p)
            (goal (at ?p ?c)))))))
          (and
            (forall (?c2) (at ?a ?c2) ;; we're no longer needed here
              (forall (?p) (at ?p ?c2)
                (implies
                  (person ?p)
                  (goal (at ?p ?c2)))))
            (exists (?p) (at ?p ?c) ;; persons need a ride at destination
              (and
                (person ?p)
                (not (goal (at ?p ?c)))))
            (or
              (not ;; there is no plane at destination
                (exists (?a2) (at ?a2 ?c)))
              (not ;; there is no plane scheduled for destination
                (exists (?a2) (scheduled ?a2 ?c)))))
          (and
            (goal (at ?a ?c)) ;; destination is our goal
            (forall (?p ?c2) (at ?p ?c2) ;; no person needs to travel
              (implies
                (person ?p)
                (goal (at ?p ?c2))))))))

;; We need to fly somewhere (so we'll need fuel)

(def-defined-predicate (need-to-fly ?a)
  (or
    (exists (?c) (scheduled ?a ?c)) ;; we're scheduled
    (exists (?p) (in ?p ?a)) ;; someone has boarded the plane
    (and
      (not
        (exists (?p ?a2) (in ?p ?a2)))
      (forall (?c) (at ?a ?c)
        (exists (?c2) (city ?c2)
          (and
            (not (= ?c ?c2))
            (exists (?p) (at ?p ?c2) ;; persons who need a ride
              (and
                (person ?p)
                (not (goal (at ?p ?c2)))))
            (or
              (not ;; there is no plane there
                (exists (?a2) (at ?a2 ?c2)
                  (aircraft ?a2)))
              (not ;; there is no plane scheduled for there
                (exists (?a2) (scheduled ?a2 ?c2)))))))
    (forall (?c) (at ?a ?c) ;; we're not at our goal
      (exists (?c2) (goal (at ?a ?c2))
        (not (= ?c ?c2))))))

;; OK to refuel plane ?a

;; Only refuel if we're out of fuel and we need to fly

(def-defined-predicate (ok-to-refuel ?a)
  (and
    (forall (?f) (fuel-level ?a ?f)
      (not (exists (?f2) (fuel-next ?f2 ?f))))
    (need-to-fly ?a)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 3. Initialization Formula.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(set-initialization-sequence

;; Delete any person without a goal location.

  (forall (?p) (person ?p)
    (or
      (exists (?c) (goal (at ?p ?c)))
      (and
        (del (person ?p))
        (forall (?c) (at ?p ?c)
        (del (at ?p ?c)))
        (forall (?a) (in ?p ?a)
          (del (in ?p ?a))))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 4. The temporal control formula.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 5. Operators.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; Debark a person from a plane

(def-adl-operator (debark ?person ?aircraft ?city)
  (pre
    (?person ?aircraft) (in ?person ?aircraft)
    (?city) (at ?aircraft ?city)
    (goal (at ?person ?city)))
  (add
    (at ?person ?city))
  (del
    (in ?person ?aircraft)))

;;; Board a person onto a plane

(def-adl-operator (board ?person ?aircraft ?city)
  (pre
    (?aircraft) (aircraft ?aircraft)
    (?city) (at ?aircraft ?city)
    (?person) (person ?person)
    (and
      (at ?person ?city)
      (ok-to-board ?person ?aircraft)))
  (add
    (in ?person ?aircraft))
  (forall (?c2) (goal (at ?person ?c2))
      (or
        (scheduled ?aircraft ?c2)
        (add
          (scheduled ?aircraft ?c2))))
  (del
    (at ?person ?city)))

;;; Fly a plane from one city to another

(def-adl-operator (fly ?aircraft ?from ?to ?fuel-from ?fuel-to)
  (pre
    (?aircraft) (aircraft ?aircraft)
    (?from) (at ?aircraft ?from)
    (?to) (city ?to)
    (?fuel-from) (fuel-level ?aircraft ?fuel-from)
    (?fuel-to) (fuel-next ?fuel-to ?fuel-from)
    (and
      (not (= ?from ?to))
      (ok-to-fly ?aircraft ?to ?fuel-to)))
  (add
    (at ?aircraft ?to)
    (fuel-level ?aircraft ?fuel-to))
  (implies
    (scheduled ?aircraft ?to)
    (del
      (scheduled ?aircraft ?to)))
  (del
    (fuel-level ?aircraft ?fuel-from)
    (at ?aircraft ?from)))

;;; Zoom a plane from one city to another

(def-adl-operator (zoom ?aircraft ?from ?to ?fuel-from ?fuel-mid ?fuel-to)
  (pre
    (?aircraft) (aircraft ?aircraft)
    (?from) (at ?aircraft ?from)
    (?to) (city ?to)
    (?fuel-from) (fuel-level ?aircraft ?fuel-from)
    (?fuel-mid) (fuel-next ?fuel-mid ?fuel-from)
    (?fuel-to) (fuel-next ?fuel-to ?fuel-mid)
    (and
      (not (= ?from ?to))
      (ok-to-fly ?aircraft ?to ?fuel-to)))
  (add
    (at ?aircraft ?to)
    (fuel-level ?aircraft ?fuel-to))
  (implies
    (scheduled ?aircraft ?to)
    (del
      (scheduled ?aircraft ?to)))
  (del
    (fuel-level ?aircraft ?fuel-from)
    (at ?aircraft ?from)))

;;; Refuel a plane

(def-adl-operator (refuel ?aircraft ?city ?fuel-from ?fuel-to)
  (pre
    (?aircraft) (aircraft ?aircraft)
    (?city) (at ?aircraft ?city)
    (?fuel-from) (fuel-level ?aircraft ?fuel-from)
    (?fuel-to) (fuel-next ?fuel-from ?fuel-to)
    (ok-to-refuel ?aircraft))
  (del
    (fuel-level ?aircraft ?fuel-from))
  (add
    (fuel-level ?aircraft ?fuel-to)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; World Print Routine
;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(def-defined-predicate (print-zeno-world ?stream)
  (and
    (forall (?x ?y) (fuel-next ?x ?y)
      (print ?stream "(fuel-next ~A ~A)~%" ?x ?y))
    (print ?stream "~%")

    (forall (?x) (city ?x)
      (print ?stream "(city ~A)~%" ?x))
    (print ?stream "~%")
    (forall (?x) (aircraft ?x)
      (print ?stream "(aircraft ~A)~%" ?x))
    (print ?stream "~%")

    (forall (?x) (person ?x)
      (print ?stream "(person ~A)~%" ?x))
    (print ?stream "~%")

    (forall (?x ?y) (at ?x ?y)
      (print ?stream "(at ~A ~A)~%" ?x ?y))
    (print ?stream "~%")

    (forall (?x ?y) (in ?x ?y)
      (print ?stream "(in ~A ~A)~%" ?x ?y))
    (print ?stream "~%")

    (forall (?x ?y) (fuel-level ?x ?y)
      (print ?stream "(fuel-level ~A ~A)~%" ?x ?y))
    (print ?stream "~%")

    (forall (?x ?y) (scheduled ?x ?y)
      (print ?stream "(scheduled ~A ~A)~%" ?x ?y))
    (print ?stream "~%")))

;(set-print-world-fn print-zeno-world)
```

Acknowledgements

If your name is not remembered, it does not mean that you are not ...
Beck
Maria and Derek
Amy, Badger, Curly Phil, Copley, David and Geni, Dom, Don J, Dyna White, Dyson, Ed, EPSRC, George, GOJ, Handy, Hazel, Jared, Jazz, JB, John, Ken, Keith, Lou-Lou, Mike and Derek, Naomi, Nick, Olly, Paddy, Parky, Paula, Rory B, Sash, Simon and Sally, Smith, Tom, Tori, WATN, Yaz.

Bibliography

[1] F. Bacchus. Aips-2000 planning competition, 2000.

[2] F. Bacchus and M. Ady. Precondition control. Unpublished manuscript, 1999.

[3] F. Bacchus and F. Kabanza. Using temporal logic to control search in a forward chaining planner. In *Proceedings of Second International Workshop on Temporal Representation and Reasoning (TIME)*, Melbourne Beach, Florida, 1995.

[4] F. Bacchus and F. Kabanza. Planning for temporally extended goals. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 1215–1222, Portland, Oregon, USA, 1996. AAAI Press / The MIT Press.

[5] F. Bacchus and F. Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[6] C. Bäckström. *Computational Complexity of Reasoning about Plans.* PhD thesis, Linköping University, Linköping, June 1992.

[7] C. Backström and B. Nebel. Complexity results for SAS+ planning. Technical Report LiTH-IDA-R-93-34, Linköping University, Linköping, Sweden, 1993.

[8] C. Baral and T. Eiter. A polynomial-time algorithm for constructing k-maintainable policies. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS04)*, 2004.

[9] A. Blum and M. Furst. Fast planning through planning graph analysis. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, 1995.

[10] T. Bylander. Complexity results for planning. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI 91)*, pages 274–279, 1991.

[11] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[12] M. Clark. Construction types: A generic type solved. In *Proceedings of the Twentieth Workshop of UK Planning and Scheduling Special Interest Group (PLANSIG 01*, pages 32–43, 2001.

[13] K. Currie and A. Tate. O-Plan: The open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

[14] J. P. Delgrande, T. Schaub, and H. Tompits. Domain-specific preferences for causal reasoning and planning. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS04)*, 2004.

[15] P. Doherty, J. Gustafsson, and L. Karlsson. Temporal action logics (TAL): Language specification and tutorial. Linköping University E-Press, 1998.

[16] P. Doherty and J. Kvarnström. TALplanner: An empirical investigation of a temporal logic-based forward chaining planner. In *TIME*, pages 47–54, 1999.

[17] S. Edelkamp and M. Helmert. The model checking integrated planning system. *AI Magazine*, 22(3):67–71, Fall 2001.

[18] E. Allen Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science*, B:997–1072, 1990.

[19] A. Fern, S. Yoon, and R. Givan. Learning domain-specific control knowledge from random walks. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS04)*, 2004.

[20] E. Fink and Q. Yang. Planning with primary effects: Experiments and analysis. In *IJCAI 95*, pages 1606–1611, 1995.

[21] M. Fox and D. Long. The automatic inference of state invariants in TIM. *JAIR*, 9:367–421, 1998.

[22] M. Fox and D. Long. Hybrid STAN: Identifying and managing combinatorial optimisation sub- problems in planning. In *IJCAI*, pages 445–452, 2001.

[23] A. Gabaldon. Precondition control and the progression algorithm. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS04)*, 2004.

[24] A. Gerevini and L. Schubert. Extending the types of state constraints discovered by DISCOPLAN. Proceedings of the Workshop at AIPS on Analyzing and Exploiting Domain Knowledge for Efficient Planning, 2000.

[25] A. Gerevini and L. K. Schubert. Computing parameter domains as an aid to planning. In *Proceedings of the Third International Conference on Artificial Intelligence Planning and Scheduling AIPS96*, pages 94–101, Menlo Park, CA, 1996. AAAI Press.

[26] A. Gerevini and L. K. Schubert. Inferring state constraints for domain-independent planning. In *AAAI/IAAI*, pages 905–912, 1998.

[27] M. Helmert. Complexity results for standard benchmark domains in planning. *Artificial Intelligence*, 143(2):219–262, 2003.

[28] J. Hoffmann. A heuristic for domain independent planning and its use in an enforced hill-climbing algorithm. In *International Syposium on Methodologies for Intelligent Systems*, pages 216–227, 2000.

[29] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.

[30] S. Kambhampati. A critique of knowledge-based planning track at ICP, 2003.

[31] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In H. Shrobe and T. Senator, editors, *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201, Menlo Park, California, 1996. AAAI Press.

[32] H. Kautz and B. Selman. The role of domain-specific knowledge in the planning as satisfiability framework. In *Artificial Intelligence Planning Systems*, pages 181–189, 1998.

[33] H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In Jack Minker, editor, *Workshop on Logic-Based Artificial Intelligence, Washington, DC, June 14–16, 1999*, College Park, Maryland, 1999. Computer Science Department, University of Maryland.

[34] G. Kelleher and A. Cohn. Automatically synthesising domain constraints from operator descriptions. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI 92)*, pages 653–655, 1992.

[35] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In *Proceedings of the Fourth European Conference on Planning (ECP 97)*, 1, 1997.

[36] R. Korf. Macro-operators: a weak method for learning. *Artificial Intelligence*, 26:35–77, 1985.

[37] J. Kvarnström. Applying domain analysis techniques for domain-dependent control in TALplanner. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 02)*, pages 101–110, 2002.

[38] D. Long and M. Fox. Automatic synthesis and use of generic types in planning. In *Artificial Intelligence Planning Systems*, pages 196–205, 2000.

[39] D. Long and M. Fox. The 3rd international planning competition: Results and analysis. *Journal of AI Research*, 20:1–59, 2003.

[40] D. Long and M. Fox. *Planning with Generic Types*, volume Exploring Artificial Intelligence in the New Millenium, chapter 4, pages 103–138. Morgan Kaufmann, 2003.

[41] D. McAllester. Observations on cognitive judgments. In *National Conference on Artificial Intelligence*, pages 910–914, 1991.

[42] D. McDermott. The 1998 AI planning systems competition. *AI Magazine*, 21(2):35–55, Summer 2000.

[43] S. Minton. Learning effective search control knowledge: An explanation-based approach. Technical Report CMU-CS-88-133, CMU, 1988.

[44] S. Minton, C. Knoblock, and J. Carbonell et al. PRODIGY 2.0: The manual and tutorial. Technical Report CMU-CS-89-146, Carnegie-Mellon University, 5, 1989.

[45] D. Nau, T. Au, O. Ilghami, U. Kuter, J. Murdoch, D. Wu, and F. Yaman. SHOP2: An HTN planning system. *JAIR*, 20:379–404, 2003.

[46] D. S. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila. SHOP: Simple hierarchical ordered planner. In T Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI 99)*, pages 968–975. Morgan Kaufmann, 1999.

[47] J. Rintanen. An iterative algorithm for synthesizing invariants. In *AAAI/IAAI*, pages 806–811, 2000.

[48] S. Russell and P. Norvig. *Artificial Intelligence.* Prentice Hall, New Jersey, 1995.

[49] E. Sandewall. Cognitive robotics and its metatheory. Linköping University E-Press, 1998.

[50] U. Scholz. Extracting state constraints from PDDL-like planning domains. Proceedings of the Workshop at AIPS on Analyzing and Exploiting Domain Knowledge for Efficient Planning, April 2000.

[51] B. Selman, H. J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.

[52] D. Smith. Choosing ojectives in over-subscription planning. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS04)*, 2004.

[53] S. J. J. Smith, D. S. Nau, and T. Throop. A planning approach to declarer play in bridge. *Computational Intelligence*, 12(1):106–130, February 1996.

[54] K. Stergiou and T. Walsh. Encodings of non-binary constraint satisfaction problems. In *AAAI/IAAI*, pages 163–168, 1999.

[55] G. A. Sussman. A computational model of skill aquisition. Technical Report AI-TR-297, MIT, 1973.

[56] P. van Beek and X. Chen. CPlan: A constraint programming approach to planning. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence*, pages 585–590. AAAI/MIT Press, 1999.

[57] D. E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, San Mateo, California, 1988.

[58] H. L. S. Younes and R. G. Simmons. Policy generation for continuous-time stochastic domains with concurrency. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS04)*, 2004.