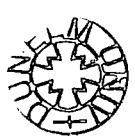
Object-orientated planning domain engineering

Mark Tully

Object-Orientated Planning Domain Engineering

Mark Tully

M.Sc. Thesis
2001



Planning Group
Department of Computer Science
University of Durham

Abstract

The development of domain independent planners focuses on the creation of *generic problem solvers*. These solvers are designed to solve problems that are declaratively described to them.

In order to solve arbitrary problems, the planner must possess efficient and effective algorithms; however, an often overlooked requirement is the need for a complete and correct description of the *problem domain*.

Currently, the most common domain description language is a propositional logic, state-based language called STRIPS. This thesis develops a new object-orientated domain description language that addresses some of the common errors made in writing STRIPS domains. This new language also features powerful semantics that are shown to greatly ease the description of certain domain features.

A common criticism of domain independent planning is that the requirement of being domain independent necessarily precludes the exploitation of domain specific knowledge that would increase efficiency. One technique used to address this is to recognise patterns of behaviour in domains and abstract them out into a higher-level representations that are exploitable. These higher-level representations are called *generic types*.

This thesis investigates the ways in which generic types can be used to assist the domain engineering process. A language is developed for describing the behavioural patterns of generic types and the ways in which they can be exploited. This opens a domain independent channel for domain specific knowledge to pass from the domain engineer to the planner.

Acknowledgements

This thesis would not have been possible without the inspiration given to me by my supervisor Dr. Maria Fox and my pseudo-supervisor Dr. Derek Long. I would like to thank them and the rest of the planning group for their guidance and support.

I would also like to extend my thanks to the EPSRC for their funding and the PLANFORM project for their grant, both of which allowed me to complete this work.

Copyright

Declaration

No part of the material offered has previously been submitted by the author for a degree in the University of Durham or in any other University. All the work presented here is the sole work of the author and no one else.

Table of Contents

1 Introduction

This section introduces the basic concepts of planning and planning domain engineering. An overview is given of the current state of domain engineering followed by a short discussion of the aims of this project. A general overview of the structure of the thesis can be found at the end of this section.

1.1 The Science of Problem Solving

Computers are good at following instructions. When a solution to a problem exists, computers can be programmed to implement a solution and will do so quickly and accurately.

When a computer is to be used to solve a problem, the usual approach is for a computer scientist to design an algorithm to solve the problem, and then execute the program on the computer. This approach will, more often than not, result in a highly efficient solver for that particular problem and the same algorithm can be deployed on future problems of the same kind to solve them just as quickly.

One of the requirements of this approach is that a solution to the problem at hand either exists, or can be developed. If there are a large number of varied problems that need to be solved, this can mean a lot of time spent developing algorithms and maintaining previous solutions.

An alternative approach is to develop a generic problem solver, which is capable of solving a wide range of problems given only a problem specification. A problem specification would be much quicker to write than a specialised solver meaning the time from problem conception to solution is reduced. The study and development of such generic solvers is called *domain independent planning.*

1.2 Domain Independent Planning

The disadvantage of a domain dependent problem solver is that the algorithm cannot generally be transferred to another domain. For example, a route-planning algorithm cannot be easily deployed on a map colouring problem[1].

Domain independent planning differs from domain dependent planning in that the domain is given to the planner as part of its input, where as a domain dependent planner has it implicitly encoded within its algorithms.

A domain description for a domain independent planner describes the "laws of physics" for that domain, it describes what is permissible and what is not. The description defines actions that can be used in the domain and what the actions achieve. Finally, a problem specification describing the initial state and desired states of the domain is written. The planner takes both the domain description and the problem description and produces a sequence of actions that, when applied, will take the domain from the initial state to the desired state. This action sequence is called a *plan*.

1.3 Domain Engineering

Planning as a research interest has been mainly fuelled by the desire to solve problems of increasing difficulty. Although this has doubtlessly lead to the development of better planning algorithms, the question of whether or not planners can be easily used to solve problems has been largely neglected.

Before a planner can be deployed on a given domain, a domain description must first be created in a process called *domain engineering*. Several different languages exist for describing domains, but the most commonly used one is a predicate logic based model called STRIPS. Engineering a domain in STRIPS means editing a text file by hand to encode into predicate logic the rules and actions of the domain. This is not always a straightforward process, mainly because it is often the case that a fair amount of experience is needed with STRIPS before correct domain descriptions can be created.

[1] Complexity theory demonstrates how problems can be translated between domains, allowing solutions for one domain to be deployed in another. However, the problem translators must still be hand written.

Domain engineering tools that assist the user in the process of creating a domain description are not very common, meaning that someone who wishes to deploy a planner for the first time may find the hurdle of writing their first domain in STRIPS too high to clear.

If planning is to become more widespread outside of the research field it is not just the planners than need work, good support from the domain engineering side is also needed so that planners can be deployed with the minimum of effort and experience on the user's behalf.

1.4 Exploiting Domain Specific Knowledge

As domain independent planners cannot be programmed for any particular domain, they cannot even exploit the most trivial of domain dependent knowledge. Although it is possible to pass the planner "clues" such as search heuristics to help guide its search, this is seen by some to be counter to the goals of domain independent planning. However, the performance boost given to planners by the introduction of domain dependent knowledge is too large to be ignored.

An approach more inline with the philosophy of domain independent planning is to perform automatic analysis on the domain in a pre-planning phase, this phase can lead to the production of domain invariants and higher level structures that were previously not apparent. This new information can then be directly exploited by the planner to improve its efficiency.

One form of this analysis is to recognise known patterns of behaviour in the domain and abstract them out into higher level types that can be reasoned with by more specialised algorithms. For example, route planning is often a common part of a problem, it may be possible to always identify maps in a domain description and abstract them out into a specialised data structures in the planner. This structure could then be manipulated by the planner by specialised route planning algorithms. These higher-level types are called *generic types.*

Generic types can also be used in domain engineering as well as planners. In domain engineering they can be used to focus the editing session more, introducing terms that are relevant to the domain in question, tailoring the

editing experience. Specialised sub editors can also be called upon, such as a map editor for the aforementioned map generic type.

1.5 Object-Orientation

Object-orientation is the process of viewing a domain in terms of its objects, their properties and the messages that they pass between them. Object-orientation was first talked about in the 1960s by those working on the SIMULA language, by the 1970s it was an important part of the SmallTalk language being developed at Xerox Parc [Coad and Youren, 1990]. Some authors claim that object-orientation is one of the most important developments in software engineering since its foundation [Coad and Nicola, 1993] [Cox, 1986].

Booch's work on object-orientated design [Booch, 1991] showed how the same ideas could be used to design programs. Object-orientation is closer to the real world model because the real world consists of objects. Work on object-orientated analysis has shown how object-orientation can reduces the semantic distance between the real domain and the domain model [Gardarin et al, 1997].

1.6 Project Aims

The aim of this thesis is investigate whether the use of an object-orientated domain description language benefits domain engineering. This is achieved though the development of a new domain description language called the object-orientated domain description language, or OODDL for short. OODDL is evaluated against the most popular domain description language by a usability study.

Further to the development of OODDL, methods for the exploitation of domain specific knowledge in a domain independent environment are discussed. This specifically relates to existing research concerning the recognition of common planning domain behaviour patterns in the form of generic types.

The role of generic types in domain engineering is discussed in detail and a new language for describing generic types and the manipulations that can be performed on them is developed. This language is called the generic type language, or GTL for short.

1.7 Structure of this Thesis

Introduction

The introduction provides an overview of planning, domain engineering and generic types. It also includes a statement of the aims of the thesis.

Background

This chapter provides in-depth background on domain description languages, generic types, object-orientation and existing domain engineering tools.

Design of OODDL

This chapter details the development of the new domain description language OODDL and compares and contrasts it with STRIPS. A discussion of the translation algorithm from OODDL to STRIPS is also included.

Design of GTL

This chapter details the development of the generic type description language: GTL. The use of generic types in domain engineering is discussed and the possibilities for future work in this area are explored.

Evaluation

The evaluation details tests that are designed to pit OODDL against STRIPS. The ideas behind the tests and a discussion of the expectations are included. The results of the tests are analysed and conclusions drawn.

Conclusion

This chapter provides a concluding discussion relating to the usefulness of OODDL and GTL. Possible future directions for domain engineering are also discussed.

Appendices

This chapter includes a formal grammar for both OODDL and GTL, followed by a copy of the tests designed to evaluate OODDL.

References

A list of referenced sources.

2 Background

This section details some of the existing methods available for domain engineering, including existing domain engineering tools and domain description languages. An introduction to *generic types* and their place in planning is detailed, followed by an overview of object-orientation and the benefits it could bring a modelling language.

2.1 The Process of Domain Engineering

Domain independent planners depend on an accurate domain description if a suitable plan is to be produced. It is the job of the domain engineer to describe a real world domain to the planner, by use of a domain modelling language. This language must be expressive enough to capture the semantics of the domain, yet it must not be so confusing that a high degree of expertise is required.

The easier the communication is between the domain engineer and the planner, the more flexible the planning system as a whole will become. A better domain modelling process will allow domain independent planners to be deployed quickly and effectively, for everyday problem solving.

2.2 Capturing the Domain

The first stage of creating a domain model is to understand the real domain that is to be modelled. Once the domain is understood, it can then be captured in a domain modelling language.

It is important to decide what the scope of the domain model will be, how much of the real world domain will it capture? How much of the domain can be abstracted out whilst keeping the model equivalent enough for the plans to be valid in the real world?

2.2.1 The Logistics Domain

A popular example from the field of planning can be used to explain this further. This problem is called *logistics* [Veloso, 1992] and is common benchmark for comparison of domain independent planners.

Consider a logistics problem relating to delivery of a number of packages to a number of possibly different destinations. The packages can travel by road in trucks, or by air in freight planes. The starting point for a package can be any location in a city and the destination can be any location in the same, or different, city.

The cities are too far apart for road travel to be feasible, and so the only method of getting a package from one city to another is for a truck to take it to the city's airport and for the freight plane to fly it to the other city, where another truck can complete the delivery.

The problem is to find the optimal pattern of truck movements and plane deliveries so that all the packages are delivered in the shortest possible time.

Once the domain engineer has a problem specification, they must decide how detailed the model must be. Should fuel usage be considered? Should the optimal routes for trucks within cities be found? Perhaps the capacity of the trucks should be taken into account? Should a schedule be imposed on the aeroplanes as might be necessary in the real world?

This is a very similar process to the situation in software designing when one must perform requirements capture [Sommerville, 1992]. The scope of the domain model need only be detailed enough so that the plans produced are valid in the real domain. With the logistics domain, the central problem is the efficient flow of packages between the initial and final destinations. It's perfectly acceptable to assume the trucks don't need to make fuel stops on their journeys, or that if they do, they don't need to be detailed in advance.

Study of the real world domain in this case will possibly bring about the following requirements:

"There are locations that can hold packages, the locations are grouped into cities. Each city has a single airport that can be visited by an aircraft. The

aircraft can ferry packages between the airports in the different cities. Within the cities, there are trucks that can move the packages between locations, it can be assumed that the trucks and aeroplanes have unlimited carrying capacity and unlimited fuel.

The packages must be moved from their start location to their final location in as efficient manner as possible."

2.3 Designing the Model

Once the real world domain is understood and it has been decided what is to be modelled, the next stage is to design the model. There are two main assumptions that are almost universally accepted across the field of domain independent planning, these are the *finite world assumption* and the *STRIPS assumption.*

The finite world assumption states that the domain is bound and finite; objects can neither be created nor destroyed. This assumption was originally made to simplify the task of the planner, however it has the effect of relieving the domain engineer of potential tasks such as memory management and dynamic object tracking.

The STRIPS assumption was made to tackle the *frame problem.* The frame problem is the question of whether facts in the domain not mentioned by an action persist after the action is applied. The STRIPS assumption, so called because the Stanford Research Institute Problem Solver assumed it, states that facts persist unless explicitly deleted by an action.

Planning domain modelling is quite different from software engineering, however there are similarities. Two of the most useful approaches to the design of a domain model can be taken from software engineering: the *functional model* and the *object model.*

2.3.1 The Functional Model

One method of creating a model from a specification is to progressively refine the model from a high level functional view into a more detailed design. The domain engineer selects areas of functionality in the real world domain and creates equivalent functionality in the model. The entire model is developed

concurrently with each action possibly influencing others. The functional model was popularised in software engineering by Jackson [Jackson, 1975] with his Jackson System Diagrams, and Wirth in his development of step-wise refinement [Wirth, 1971].

The functional method of design can work quite well with domain modelling languages that work with operator hierarchies, such as Hierarchical Task Network languages that are discussed later in this section. However, most domain modelling languages do not support the definition of operators in terms of other operators and so the functional model can only extend to one level.

The functional model does not attempt to capture the state of the domain, only the functionality of the domain. For this reason the functional model will need to be coupled with a method of representing state, such as the *propositional model.*

The propositional model uses first order predicates to describe the state of the world at any time. Actions in the model take affect by creating or deleting predicate instances from the world state.

This model is very flexible, and can support powerful functional operators including mathematical operators and logical quantification. Geffner has even explored a purely functional variant of STRIPS that is function orientated in a similar way to functional programming [Geffner, 2000]. However, the purely function or propositional model can seem unnatural to the inexperienced, mainly because people generally think in terms of objects and their properties, rather than predicates and populating objects[2].

Constructing a domain using the propositional model involves creating a list of predicates to instantiate on domain objects. All relations between objects must be captured in instances of these predicates.

Although some research has been done outside the field of planning on object-orientating predicate-based models [Conery, 1987], the propositional model used in planning is often quite flat and unstructured.

[2] Cognitive theory states that humans think primarily in terms of objects, images and object-propositions [Eysenck and Keane, 2000]

2.3.2 The Object Model

"Objects with encapsulated state and well defined messages have been found very suitable for describing real world entities and their dynamic behaviour"

[Chen and Warren, 1988]

Object-orientated programming and object-orientated design are arguably the most influential developments in software engineering. The object-oriented methods extend well to planning domain modelling where real world objects can be represented in the model directly.

With an object-orientated model, the system is organised as a cooperative collection of objects; because each object manages its own state information, the system state is decentralised. Each object is a member of a class that defines the object by declaring variables held by the object and methods that can be invoked upon it.

Object classes can *inherit* attributes from super-classes. This can be used in domain modelling to model one class of objects as being an extension to another. If a class derived from multiple superclasses it is called *multiple-inheritance.*

When a class inherits from another, it has the opportunity to replace part of the super class's behaviour by overriding. Overriding is the process of declaring a method with the same signature as one in the super class, causing it to be used by instances of the class instead of the original. This is a very useful feature in domain modelling because it allows physical domain objects to have default behaviour unless otherwise stated by an override from a subclass.

Planners working with an object-orientated domain description could theoretically produce plans showing the behaviour of individual objects in the domain, perhaps allowing resultant plans to be more easily deployed in multi-agent systems.

Object-orientated design is the process of creating a program or a simulation model consisting of objects, it was first formalised by Booch [Booch, 1991]. Booch explores diagrammatic methods for getting from a requirements list to a specification for object classes, his formal diagram notation has become widely

adopted in the software engineering field. Other methods for generating class specifications from a requirements list include CRC cards and the Noun-Verb method.

Beck and Cunningham first proposed CRC cards as a tool for teaching object-orientated programming [Beck and Cunningham, 1989]. They are simple 3x5 index cards upon which the name of a class is written at the top. The class's responsibilities are written on one half of the card and the class's collaborators are written on the other. By iteratively developing the description of each class it becomes very clear what the responsibilities of each class is. The cards can be spatially arranged to show patterns of collaboration, or task orientated functionality.

The Noun-Verb method was first proposed by Abbot as a simple way of generating a starting point for producing object specifications from a natural language description [Abbot, 1983]. His method involves underlining all nouns in the description and proposing them as objects (or in the general case: classes), and the selecting all the verbs in the description and proposing them as object methods. The Tokyo Institute of Technology has even demonstrated how this method can be automated [Saeki et al, 1989].

2.3.3 Visualising Domain Models

Designing domain models visually, whereby real world domain objects are represented in the domain model by icons or 3D models, could greatly enhance the domain engineers understanding of the domain model. However, creating domains visually poses problems with ambiguity in much the same way as describing a domain in natural language does.

Lowering the ambitions slightly one can imagine a domain engineering tool that assists the domain engineer by providing state diagrams and visual typing hierarchies, perhaps similar to entity relationship diagrams in database design [Date, 1999]. Such methods could ease the domain engineer's task by representing relationships between types or objects graphically rather than in syntax. Grant has already looked at using entity relationship diagrams in domain modelling, as well as other inductive modelling techniques [Grant, 1996].

Effective diagrams do allow for better understanding of data, for instance, Venn diagrams are generally seen as easier to understand than set theory; however, one must keep in mind that designing good diagrammatic representation is often a very difficult task.

"Too often it is taken for granted that a diagrammatic notation will be easier to read than a conventional one, an assumption that is not always correct."

[Green, 1979]

Whether a planning domain can be built visually or not, it will still require a solid underlying data model, be it propositional or object-orientated.

2.3.4 Summary of Models

Meyer made the prudent observation that:

"Finding an infallible technique of designing software is about as likely as finding an infallible technique for designing a house"

[Meyer, 1988]

This statement is true for planning domain models also, there are many different ways of encoding a real world domain, and no one model is necessarily better than another. However, the following observations can be made:

The object model has proved very popular in the software engineering field and object-orientated programming has showed the benefit of its application.

The propositional model has already shown itself to be a flexible method of encoding planning domains, although it is often not easy to clearly relate to a real world domain.

2.4 The Use of Ontologies

Once the basic model is laid out, it must be encoded into a domain modelling language for a planner to use. The most common way of doing this is simply to take the modelling language of choice and write the domain description by hand.

One branch of domain engineering research is concerned with the use of *ontologies*. An ontology is a "specification of a conceptualisation" [Gruber, 1993]. It is used to describe the relationships and behaviour of a set of agents. In

the context of planning domain engineering, an ontology is used as a framework for describing a particular domain. Joint-research at the universities of Salford and Huddersfield has been looking at ways that reusable libraries of ontologies can be used as a foundation for starting new domain descriptions.

Ontologies can be linked to the idea of software reuse easily. If a library of reusable domain components is maintained and updated, domains could be more rapidly developed by directly using, or extending from, existing domain descriptions.

2.5 An Overview of Domain Description Languages

There are several different domain description languages[3], which largely reflect the myriad of different planning architectures that are actively under research. The domain description languages discussed in this section were all developed to suit the different needs of the various planners, providing each with a domain representation that could be efficiently manipulated during the search for a plan.

Although the quest for an efficient and effective model is very important in the development of a planning system, this aim often opposes the ideal of making a domain language easy to use and productive to develop domain models in.

To use an analogy from software engineering, hand-optimised assembly may be faster and give the most efficient implementations of an algorithm, however higher level languages such as C++ provide implementations that are both easier to understand and more portable.

Portability is an issue in planning as well as the variety of domain description languages serves to reduce domain compatibility between the planners. This makes direct comparisons between planning systems more difficult as identical problem encodings cannot be used, meaning it is unclear whether all planners have the same amount of information.

[3] The terms "domain modelling language" and "domain description language" are used interchangeably throughout this document. Both terms refer to the same thing, the language that describes a domain model to the planner.

If a new domain description language is to be effective, it must address the concerns of efficiency, portability and usability. This section discusses some of the existing domain description languages, their features and limitations.

2.5.1 STRIPS

One of the most popular domain description languages in planning is the ubiquitous STRIPS language. STRIPS is an acronym for Stanford Research Institute Problem Solver [Fikes and Nilsson, 1971], which was one of the early domain independent planners.

The STRIPS system handled finite domains, that is, domains in which objects could not be created or destroyed, using logic predicates to model world state information. A STRIPS domain consists of a collection of predicate declarations and a collection of domain operators. The domain predicates are arbitrary arity first order predicates that are instantiated with problem domain objects to create facts. For example, using the aforementioned logistics domain, we could state that the truck, *truck1* was at location *location1* with the following grounded predicate:

```
at(truck1,location1)
```

STRIPS domain operators have three components:

- Parameters
- Preconditions
- Effects

An operator can be instantiated by the planner by matching any combination of domain objects to the operator parameters. However, the action is not a valid instace unless the preconditions of the action hold. The preconditions are a list of domain predicates based on the operator parameters[4]; the preconditions must be true of the parameters before the operator effects can take place. The effects

[4] A STRIPS operator's preconditions and effects lists in general involve only the operator's parameters, however domains can have *constant* objects that are globally available to all operators. These objects can be directly referred to in the operator description without use of a parameter, but do not allow the domain engineer to do anything that they could not do before.

are a list of predicates involving the parameters that are either created or destroyed when the action is applied.

If a fact is added that is already present, then a duplicate is created. This is perfectly valid and some domains rely on this behaviour to implement counting behaviour. However, a fact cannot be removed if it doesn't exist and so all negative effects must be declared as preconditions to the action.

A simple operator to move the truck in the previous example from one location to another would look like the following:

```
move(truck, from, to)
        preconditions:
            at(truck,from)
        effects:
            at(truck,to)
            ¬at(truck,from)
end
```

A sample STRIPS move operator from the logistics domain

2.5.2 The Issue of Typing

Basic STRIPS has no typing, meaning that any object can be assigned to any operator parameter by the planner. Although later editions of STRIPS supported typing, the types of objects can be verified easily enough in the operator preconditions. If the domain engineer declares a unary predicate for a type in the domain, and then instantiates it for each object of that type, operators requiring that predicate of the input parameters will be guaranteed an object of the correct type.

The logistics domain could use typing predicates: truck(x), aeroplane(x), package(x) and location(x) for example.

If a domain is encoded in un-typed STRIPS then the domain engineer has the responsibility of maintaining another set of predicates. Typed STRIPS is becoming more popular, however the majority of the STRIPS domains in circulation now are in un-typed STRIPS, using typing predicates instead.

2.5.3 PDDL

As research into domain independent planning became wider spread, the basic STRIPS concept was being stretched in different directions. Different planners used different STRIPS based encodings for the same data, making it hard to directly compare planners on the same set of domains.

In 1998 the Artificial Intelligence and Planning Scheduling (AIPS) conference was to feature a competition between various planning systems. Research institutes from around the world were to meet in Carnegie Mellon University in the USA to compare and contrast different planning techniques and pit their planners against each other.

It was realised that in order for this to be possible a standardised domain representation would be needed, one which encompassed not only the core features of STRIPS, but the additional features implemented by the myriad of planners it would be used by.

The resulting language was called the Problem Domain Description Language (PDDL) [McDermott et al, 1998]. It used a text based LISP structure to encode, in an extensible way, a solid standardised implementation of STRIPS. It also addressed a newer direction in planning being taken by Pednault called the Action Description Language, or ADL [Pednault, 1989].

PDDL also supported, as an option, typing for actions and predicates, to address the variants of STRIPS that required typing.

PDDL should really be seen simply as a "container" language for encoding domains in using other languages, such as STRIPS, ADL or HTN models (which are described in the following sections). Although PDDL does an admirable job of standardising domain descriptions, its syntax is often quite cumbersome to edit by hand, where the domain engineer can make simple syntax errors by getting confused by the multitude of nesting brackets.

PDDL has become the standard language for domain independent planners and it is constantly evolving to meet the researchers' needs. A recent revision of the language, PDDL 2.1, has introduced numbers (along with a limited number

of mathematical operations) and temporality (in the form of durative actions) [Fox and Long, 2001b].

2.5.4 ADL

One of the significant extensions to STRIPS is the *Action Description Language*, or ADL [Pednault, 1989]. ADL brings more powerful actions to STRIPS by allowing the use of quantification operators such as "for all" and "exists".

In standard STRIPS, effects and preconditions could only involve explicitly identified objects, e.g. the operator parameters. With ADL it is possible to use very general expressions to accomplish tasks that in standard STRIPS would require additional work.

For example, in the logistics domain mentioned earlier, it is stated that a package can only be loaded if it is not already in a vehicle. The most common way of accomplishing this in the basic STRIPS encoding of logistics is to delete the "at" predicate that records the location of the package. This means the package will not meet the preconditions of any other load operators and so cannot be loaded into another vehicle. Conversely, the unload operator would re-create the "at" predicate for the package with whatever location the vehicle is at when unloading.

```
(:action load-truck
        :parameters
            (?obj
            ?tru
            ?loc)
        :precondition
            (and (package ?obj)
            (truck ?tru)
            (location ?loc)
            (at ?tru ?loc)
            (at ?obj ?loc))
        :effect
            (and (not (at ?obj ?loc))
            (in ?obj ?tru))
)
```

Example load operator in basic PDDL STRIPS[5]

In a way, this could be seen as a trick on behalf of the domain engineer. Technically the package is still at the location; it's just in the truck as well. Using ADL, it is possible to leave the package's "at" predicate intact, while still retaining the mutually exclusive loading condition. The example below shows the use of ADL's quantification and typing to create a load operator that still meets the mutually exclusive loading conditions, but without deleting the "at" predicate.

[5] These domain excerpts are encoded in PDDL, to demonstrate the typical structures that a domain engineer must create by hand when using PDDL.

```
(:action load-adl
        :parameters
            (?obj - package
            ?veh - vehicle
            ?loc - location)
        :precondition
            (and (at ?obj ?loc)
            (at ?veh ?loc)
            (forall (x? - vehicle)
                (and (not (in ?obj ?x))))
        :effect
            (and (in ?obj ?veh)))
```

Example load operator in PDDL STRIPS ADL

The above operator introduces another problem that wasn't present in the basic STRIPS encoding, that is the problem of ensuring that when the vehicle moves, the packages move with it. As the basic STRIPS version deleted the "at" predicate from the packages as they were loaded, it did not need to update them when the vehicle was moved. (See the previous move operator definition in section 2.5.1).

The ADL version must use *conditional effects* to change the "at" predicate of all packages that are "in" the vehicle moving, for example:

```
(:action move-adl
        :parameters
            (?vehicle - vehicle
            ?from - location
            ?to - location)
        :precondition
            (and (at ?vehicle ?from))
        :effect
            (and (at ?vehicle ?to)
            (not (at ?vehicle ?from))
            (forall (?x - package)
                (when (and (in ?x ?vehicle))
                    (and (not (at ?x ? from))
                    (at ?x ?to))))))
```

Example move operator in PDDL STRIPS ADL

One further extension that ADL brings to STRIPS is the use of negative preconditions. Basic STRIPS only allows it to be asserted that a precondition is true; ADL allows preconditions to assert negative preconditions also.

2.5.5 The Usefulness of ADL

ADL allows some domains to be expressed much more concisely than would be possible with basic STRIPS. Some domains are very difficult, if not near impossible to model without access to ADL's conditional effects and quantifications.

Gazen and Knoblock have demonstrated that it is possible to automatically convert the most common features of ADL domain description into STRIPS at the cost of an exponential blow-up in the number of operators [Gazen and Knoblock, 1997]. However, one would be mistaken to interpret this as meaning ADL is no more expressive than STRIPS, in the same way one would be mistaken for thinking C++ was no more expressive than assembler just because it is complied down to it.

2.5.6 TLPlan

The University of Toronto in Canada are developing a planner called TLPlan, standing for temporal logic planner. The philosophy behind TLPlan is that a domain independent planner cannot succeed without domain specific knowledge [Bacchus and Fabanza, 2000], however a domain dependant planner is not flexible enough. There is also the performance issue related to domain independent planners; indeed, as Wilkins and Des Jardins observe [Wilkins and desJardins, 2000], there is still a significant difference in the performance of domain independent planners and domain dependent planners.

The approach taken with TLPlan is a compromise; domains are described for the planner as usual, but with the addition of search control rules that provide TLPlan with domain specific knowledge to help it plan.

TLPlan uses its own proprietary language for the domain descriptions, but it is based on STRIPS and ADL. Besides being able to declare standard predicates and STRIPS or ADL operators, TLPlan also supports the definition of custom first order logic equations. These custom predicates can be used to create expressions whose truth-value depends on other predicates in the current world state.

By building on these custom predicates, it is possible to create search heuristics recommending actions to the planner when faced with certain situations. For example, in the logistics domain when the truck is at the packages goal location, invoke the action to unload the package.

The following domain excerpt is a custom predicate in the TLPlan logistics domain that is true if, and only if, a package must be unloaded from the truck it is in. This is true if the truck is at the goal location of the package, or if the package is in the wrong city and the truck is at an airport:

```
;; We need to unload an object from a truck at the
;; current location iff, ?curr-loc is the goal
;; location of the object, or the object is in
;; the wrong city and the current-location is an
;; airport.
(def-defined-predicate
  (need-to-unload-from-truck ?obj ?curr-loc)
       (exists (?goal-loc)
            (goal (at ?obj ?goal-loc))
            (or
            (= ?curr-loc ?goal-loc)
            (and
            (in-wrong-city ?obj ?curr-loc ?goal-loc)
            (airport ?curr-loc)))))
```

Excerpt from a TLPlan logisistics domain showing

The domains used by TLPlan are obviously better for the planner, allowing plans to be produced in a much shorter time, but they do not help the domain engineer. A higher degree of expertise is needed to make an effective domain encoding for TLPlan, expertise in both the real-world domain and TLPlan's domain language. Essentially the user must already have a good idea about the solution of the problem in order to create effective heuristics and then they must be able to express these heuristics in such a way as to assist the planner.

2.5.7 Hierarchical Task Networks

An interesting way of modelling a planning domain is though the use of Hierarchical Task Networks, or HTNs [Erol, 1995, Erol et al., 1994]. The HTN model, like STRIPS, makes use of predicate based operators with add and delete lists; but in HTNs, these operators are called *primitive operators*. HTNs also support the notion of higher-level compound operators or *methods*.

A method is defined in terms of either other methods or primitive operators; each method can have more than one definition. Ordering constraints are defined for each decomposition definition, any of the actions on the decomposition list

agreeing with the ordering constraints can be selected by the planner as a substitution for the method instance.

A task is a request for a method to be recursively decomposed into applicable primitive operators; a resultant plan consists only of primitive operator instances. Goals are posed for the planner in forms of instantiated tasks.

The following is a pseudo code example of a HTN operator for delivering a package in the logistics domain:

```
method DeliverPackage(package,to)
          If package is at "to"
               End
          Else if package is in the correct city
               TransferByTruck(package,
                    currentloc,to)
          Else if package is in the wrong city
               1: TransferByTruck(package,
                    currentloc,city-airport)
               2: Fly(package,city-airport,
                    dest-airport)
               3: TransferByTruck(package,
                    dest-airport,to)
               Ordering constraints:
                    1 before 2, 2 before 3
          End if
end method
```

Excerpt from a pseudo code description of the logisitiocs domain in HTN

Similar methods would be needed for "Fly" and "TransferByTruck". The primitive operators in this domain could be identical to the standard STRIPS ones, i.e. load, unload, drive-truck and fly-plane.

This form of domain modelling is less about modelling the physics of the domain and more about modelling the functionality. To a certain extent, the domain engineer is encoding the solution into the domain. They must know the different possible ways of accomplishing various tasks and understand the

interactions between them. However, this is often not a problem for someone who is familiar with the real-world domain.

One problem type that can be expressed more easily in HTN than in STRIPS is the case where the goal state is a subset of the initial state. For instance, planning an itinerary where the ultimate goal is to be back in the starting location. The problem for STRIPS here is that the initial state and goal state are equal. In STRIPS the domain engineer would have to express each event of the day that was required as a goal before the planner would generate a plan to attend them all.

In HTN, the domain engineer could have a method called "MakeItinerary" that would decompose in different ways into all the subtasks that needed doing to attend all the events. The domain engineer wouldn't have to explicitly state that all events must be attended because that is implicitly encoded in the hierarchy of methods.

2.5.8 Object Centred Language

The object centred language, or OCL, was developed and is maintained by the University of Huddersfield [McCluskey and Liu, 1999]. OCL domains are much more structured than STRIPS domains because they revolve around objects and their states, rather than predicate literals.

In a STRIPS domain, the domain engineer creates predicates and instantiates them in the problem description to create a world state. STRIPS operators can add or delete these world predicates to change the world state.

OCL too makes use of predicates to describe the world state, however OCL also features invariants that allow the domain designer to restrict the domain model. For instance, the invariants can describe which grounded predicates are always true; the road links in the logistics domain would be declared as always true, because they do not change during plan execution.

A key structure in OCL is the *sort*. A sort is essentially an object class that describes the characteristics and behaviours of a group of objects. An important feature of sorts is that all possible states that objects of that sort can occupy must

be enumerated; this is contrast to STRIPS where the reachable states of objects, and their transitions, depend solely on the operator definitions.

OCL operators describe transitions between object states explicitly, making resultant condition of the object very clear. The operators have three parts: prevail conditions, necessary changes and conditional changes.

- **Prevail Conditions**: These are conditions on the objects that must be true before and after the action takes place.
- **Necessary Changes**: These show the conditions on objects that must be true before the operator is executed, and describes the new state of the object after execution.
- **Conditional Changes**: Each conditional change describes a new state of the object that will become true only if a certain condition is true before the action takes place.

The main difference between OCL operators and STRIPS operators are that there is no default persistence of facts. If a fact is not mentioned in the post-execution state of the operator, it becomes negated.

OCL domains are generally more unwieldy than equivalent STRIPS domains. OCL domains need states to be explicitly specified for all the different sorts of objects, and also additional information such as invariants. This places an extra burden on the domain engineer who must now calculate all possible states for the objects in the domain. However, once the states are defined writing operators is simply a case of declaring which states objects are in before and after the application. In STRIPS, the domain engineer must instead be careful to maintain consistency by careful construction of domain operators.

An OCL domain makes explicit the object states, transitions and invariants. A STRIPS domain encodes all of these implicitly in the operator definitions. Which approach is better is mainly a matter of preference for the domain engineer.

2.5.9 OCL*h*

OCL*h* is the HTN variant of OCL and is where all the OCL development is now focused. OCL*h* still makes use of operators, substates and all other features of OCL but it incorporates HTN task methods [McCluskey and Kitchin, 1998]. This allows domains to be expressed with the state transition model of OCL, while also being task orientated.

Its authors have described OCL*h* as a domain modelling language, but the lack of support for OCL*h* in rest of the community has so far stifled its uptake. OCL can be translated in PDDL [Simpson et al, 2000], but as yet OCL*h* cannot.

2.5.10 Summary of Domain Description Languages

Although OCL and OCL*h* are object centred languages they are not strictly object-orientated. Neither operators nor predicates are bound to OCL's *sorts* to create classes; the only data bound to a sort in OCL are the substate definitions.

Using object-orientated design, the domain engineer may want to have certain variables or actions related to certain classes and develop the classes to some extent independently. In OCL, all predicates and actions are in the "global scope" that is, they are not bound to any particular class. This means OCL is not ideal for object-oriented domain modelling.

The main issue with OCL is that the domain engineer must define the substates of each the sorts in the domain. The consideration of substates and declaration of domain invariants can be both a powerful consistency tool and a barrier to easy domain construction.

With correct substates and invariants defined, type-checking tools can check all domain operators for consistency. However, the development of substates and invariants also means the domain engineer has more information to encode and maintain.

STRIPS, by contrast, does not require the explicit definition of substates or invariants, this information is instead implicitly encoded in the domain operators by the way they add or delete predicates. This means that it is much harder to

verify domain consistency than with OCL because it cannot be done automatically. With STRIPS, domain analysis tools, such as TIM, can derive the domain invariants and substates allowing the domain engineer to verify them; however, whether it is better to check the derived invariants or to explicitly state the invariants is very much a matter or personal preference. It is certainly easier for inexperienced domain engineers not to have to enumerate sort substates and invariants.

This can be seen as one of the faults with the planner TLPlan's approach to domain modelling. In TLPlan it is necessary for the domain engineer, or planner operator, to describe a search heuristic that will effectively "guide" the planner to the correct solution. This can often require not only an in depth knowledge of the domain encoding and real world domain, but also an in depth understanding of the solution. This can mean that it is often difficult even for an experienced domain engineer to deploy TLPlan on a new domain, creating an obvious problem for inexperienced domain engineers.

HTN domain modelling views a domain as a collection of tasks that can be defined further in terms of other tasks. This leads to a different approach to the process of domain modelling because the domain engineer must, to a certain extent, understand how the goal of the problem will be encoded. One could see state based languages such as STRIPS as modelling the physics, while HTN languages like OCL*h* model higher level processes that can occur in the domain.

ADL, or the action description language, is an extension to STRIPS that bestows the power of first order logic to the domain engineer. ADL permits the use of quantification, existential preconditions, negative preconditions and conditional effects. Although most of these features can be encoded into basic STRIPS, doing so often means specialising the domain to a specific problem instance, and often results in an exponential increase in the domain size. ADL offers a concise way of describing many domain actions, although an equivalent basic STRIPS model can often be created by an experienced domain engineer.

This leads onto one of the main problems with STRIPS, which is that it is often necessary for the domain engineer to know tricks and methods to encode problems. For instance, the example STRIPS load-truck operator (in section

2.5.4) deleted the packages "at" predicate as it was loaded, allowing the domain engineer to update the packages easily as it is unloaded, whilst still maintaining the mutually exclusive loading condition.

Sometimes the methods needed are not obvious to an inexperienced domain engineer, which can make STRIPS a less than ideal language for beginners.

2.6 Domain Analysis and Generic Types

Domain independent planners are designed to solve planning problems without the aid of additional domain specific knowledge. However, extra domain knowledge is sometimes implicitly encoded within the domain in the form of invariants and recognisable object behavioural patterns. Domain analysis is a collection of methods designed to extract this information and provide a more enriched domain description.

Domains often exhibit common patterns of behaviour that can be abstracted upwards into higher-level operators. For example, the logistics domain has trucks whose behaviour can be abstracted into a representation of a mobile on a map of locations. This allows the exploitation of route planning algorithms for an increase in planning efficiency [Fox and Long, 2000b]. Such abstracted types like "mobile" and "location" are called *generic types* [Fox and Long, 1999].

Generic types are useful to planners because they allow the invocation of powerful heuristics and solvers that are not normally available. However, they are also very useful in the process of domain construction. In the context of domain construction, a generic type represents a conceptual component to the domain engineer. It maps onto a semantic object in the engineers mind, that is, one that they know manipulation rules for and even invariants.

If a domain tool uses generic types, then the engineer can manipulate the domain in higher-level ways, which are closer to their way of thinking for a particular domain. The tool can also focus the editing context more, allowing the engineer to manipulate what is, essentially, an arbitrary domain in terms familiar to the real domain it represents.

2.6.1 The Mobile Generic Type

The mobile is a generic type that conceptualises a self-propelled movable object that moves around on a network of locations. Example mobiles would include typical domain elements such as trucks, aeroplanes and robots.

The mobile is characterised by its ability to have a move operator invoked upon it to change its current location. Locations are generic types that are defined solely by their association with the mobile.

2.6.2 The Carrier Generic Type

A carrier is a simple extension of a mobile that is able to transport objects around the map of locations. Carriers can load domain objects that are classified as *portable* objects, then move to another location and unload them. Carrier-portable pairs include: trucks and packages, robots and balls and ferries and cars.

2.6.3 Generic Type Fingerprints

The signature or *fingerprint* of a generic type is a description of the pattern of behaviour in the domain that constitutes a generic type. This fingerprint should try to capture as many different encodings of the same behaviour as possible, however, even if the fingerprint does not match a description that a human would consider to be a particular generic type, it is not a disaster.

From the planner's point of view, generic types describe exploitable behavioural patterns. From a domain engineers point of view they represent concepts. If a generic type fingerprint does not match a particular domain description then the planner can still work with the domain, simply without the higher-level generic type manipulations. The same is true for the domain engineer, he can still work with the domain without generic types; the tool simply wouldn't use specialised terminology relevant to that generic type.

Generic types can offer assistance but they are not vital, therefore the fingerprints need not be all encompassing, they must simply recognise the most common patterns.

2.6.4 Generic Types and Planners

The STAN planner exploits generic types to increase its performance. It makes use of the TIM domain analysis tool to analyse a stock PDDL domain and automatically identify types, generic types [Fox and Long, 1999] and invariants [Fox and Long, 1998]. This extra information is used with a modified version of GraphPlan to improve search efficiency. The current version of STAN, STAN4, also makes use of specialised sub-solvers to tackle problems containing generic types and reintegrate the solutions back into the overall plan [Fox and Long, 2000] [Koehler, 1998].

This analysis allows STAN to process larger problem instances and at a greater speed whilst still maintaining optimality.

2.7 Existing Domain Engineering Tools

The importance of having tools to support domain engineering has been discussed by Des Jardins [desJardins, 1994], where she stated that an effective domain engineering tool would allow planning domains to be built and debugged by experts of the real domain, rather than planning experts.

One direction explored in early software engineering was the method of analysing a program that has been input by a programmer, and then describing back to them what has been understood by their input. If the programmer does not agree with what is presented, he can modify the input and try again [Klerer and May, 1965].

If domain analysis and domain engineering are tied together, a similar effect can be attained. The user could edit their domain, have it analysed and then re-presented to them. If they did not agree with the produced invariants, state models or generic types discovered by the analysis, it could indicate an error in the domain encoding.

The Draughtsman domain-engineering tool, developed by the author, took this approach. Draughtsman edited STRIPS domains by allowing the user to create or remove predicates, actions and objects though a GUI.

The TIM domain analysis tool gave Draughtsman the ability to analyse the domain. At any point, the user could run an analysis on the domain and have displayed to them state models, typing information, invariants and generic types. If the analysis indicated errors in the domain encoding the user could correct them and reanalyse the domain.

Although Draughtsman was very good at relating domain engineering and domain analysis, it was essentially founded on STRIPS. This meant that it assisted the user to construct a domain in STRIPS, but offered no help on how to go about it.

Research on the formalisation of domain modelling, including knowledge acquisition and constructing effective knowledge based models has developed into the idea of tools to support domain self-consistency [Porteous and McCluskey, 1997].

This research has prompted the development of another planning domain modelling tool called GIPO, the Graphical Interface for Planning with Objects. This tool is under development at the Universities of Salford and Huddersfield. GIPO allows creation of OCL*h* domains by allowing the editing of HTN based domain descriptions. It permits graphical state based display of the domain and maintains domain consistency and utilises ontologies to provide faster development of new domains from scratch.

However, the system is based around the OCL*h* language, which by GIPO's authors' own admittance, involves engineering constructs that are still too theoretical for an unskilled user [Simpson et al, 2001].

2.8 Summary

This chapter introduced some of the existing languages and tools available for domain engineering. Two different models were discussed: the functional model, used by languages such as STRIPS, and the object model, used by languages such as OCL. The domain description languages discussed have been summarised separately in section 2.5.10 and the different models have been summarised in section 2.3.4.

The ways in which generic types can be used in both domain-engineering tools and planners were discussed; overviews of the two most common generic types: mobiles and carriers were also included.

Domain-engineering tools such as an early edition of the Draughtsman tool for editing STRIPS domains, and the GIPO tool for editing OCL*h* domains were also introduced.

3 Design of OODDL

The user's main interface with planners is through the domain engineering language used to model their problems. If the user is unable to communicate a problem to the planner, then no matter how good the planner is it will not be able to solve the user's problem.

To this end, it is important that the domain language be easily understood and powerful. Section 2.5 discusses some of the existing planning domain description languages, the most common of which is PDDL STRIPS. One of the ideas explored in this chapter is that the use of object-orientated techniques in modelling domains can make domain modelling both more powerful and more easily understood, especially in comparison to STRIPS.

This chapter discusses the needs of a domain engineer and provides the motivation for, and a discussion of, a new object-orientated domain description language called OODDL.

3.1 Requirements

Object-orientated languages are generally regarded as being easier to relate to real world domains [Gardarin et al, 1997]. This refers to the physical organisation of real world domains, which are very easy to see as collections of objects. When a domain-engineer begins to write a domain model, they will be working from a real domain that will have real objects in it. If they are able work with objects in their design too, then it will be easier to relate the model to the real domain.

Existing domain modelling languages were discussed in section 2.5. If a new domain modelling language is to be proved useful, it must attempt to address some of the shortcomings of the existing languages.

Given that it has been decided that the new language will be object-orientated, the existing domain modelling language OCL should be given particular note. OCL is able to describe domains in an object centred manner, however it is fairly tricky to use; mainly because the domain engineer must

specify substates for the objects in the domain. This is perhaps useful for advanced domain engineers as it provides consistency checking, but it means the user must specify more information and it works against the ease of use of the language. OODDL will not require the user to specify substates or invariants explicitly and so this should make it easier to use for inexperienced users.

STRIPS has been serving the planning community for the past 30 years, but the problems with STRIPS have also been highlighted: such as the exploitation of "tricks" in domain encodings, meaning that domain engineers often require more experience to successfully create a domain. The common ways of encoding information in STRIPS are compared to the ways of accomplishing the same tasks in OODDL later in this chapter.

OODDL will try to move away from the need for users to have a large experience of domain modelling in order to create a domain. Where special tricks and techniques are used in STRIPS to model the domain, OODDL will instead use higher-level, more explicit, syntax to perform the same operations. This should make OODDL clearer than STRIPS.

The requirements of OODDL are therefore:

- OODDL will not rely on the user knowing tricks to be able to successfully encode domains. OODDL will use higher-level syntax to make explicit and obvious what before may have required greater experience to know.
- OODDL will not require the user to specify invariants or state models with their domain. Such information can be extracted using domain analysis tools such as TIM where necessary (see section 2.6.4).
- OODDL will be to be easy to understand and easy to relate to the real domain. It is hoped a combination of object-orientated structure and clear high-level syntax similar to Java or C++ will aid this end.

3.2 Compatibility

Inventing an entirely new language brings with it compatibility problems for existing planners, which would be unable to understand domains encoded using it. If OODDL is to be a viable option for domain engineers planners must

support it. This means that either planners must be updated to work with OODDL, or OODDL must be translatable into an existing supported language.

The translating approach has already been used in the field of software engineering to good effect; as Stroustrup demonstrated during the design of C++ [Stroustrup, 1994]. In order to provide easy compatibility with C, C++ was first implemented as a pre-processor for a C compiler. This meant programmers could utilise the new language without having to throw away their existing C compilers. C++ could then become more widely adopted, until eventually compilers implemented native support for it.

If the advantage to domain engineers given by the new language is great enough, then it will succeed on its own grounds. However, translation offers an easy path of adoption that would require no work to existing planners.

3.2.1 PDDL Compatibility

PDDL is the de facto language of domain-independent domain-engineering. Maintaining compatibility with PDDL, and in particular STRIPS, would provide good compatibility benefits to OODDL.

PDDL was designed to be an extendable language, so even though PDDL has no object-orientated characteristics, it is technically possible to add a new sub set of PDDL that could be object-orientated. However, any extensions to PDDL would still break compatibility with existing planners in that they would require new parser modules writing in order to understand the new sections. This makes a direct extension to PDDL less fruitful because it cannot be immediately exploited and if the extensions are too dissimilar from STRIPS then it could be hard for the planner to even get the domain to work with its algorithms.

Another approach would be to design a language that could be translated into PDDL STRIPS. Using this model, STRIPS can be seen as the assembly language that encapsulates the basic commands and data for the planner to work with; OODDL would be similar to a high level object-orientated language like C++, which is translated down into assembly for execution.

This would more than likely introduce inefficiencies in the STRIPS domains, as the output is unlikely to be as concise as a hand coded STRIPS domain.

However, by utilising domain analysis techniques (see section 2.6) it should be possible to optimise the resultant domain.

The benefits of this model are two fold: the module for translating into STRIPS could be replaced with one for generating another target language, such as ADL or OCL, with no change to the domains or planners. It gives the planners a degree of freedom whilst maintaining compatibility with the domain engineering side.

Secondly, it simplifies the job for domain engineers; they need only know one domain description language. This language could conceivably work with many different planners if the appropriate translator was written for the target language.

Translation from an object-orientated language to STRIPS is quite possible, because in essence all that needs to be done is throw structure away. Translation back from STRIPS would be considerably more difficult, in a similar way to decompiling being more difficult that compiling. This draws attention to the fact that some domain modelling languages are harder to translate than others. OODDL must have high enough level semantics to be "down translated" easily into the STRIPS and possibly other target languages in the future.

3.3 Introducing OODDL

The language that has been designed to meet these criteria is called OODDL, an acronym for Object-Orientated Domain Description Language. It has the same underlying expressive power as the ubiquitous STRIPS and is easily translatable into it, but it is easier to work with than raw STRIPS itself.

An OODDL domain description consists of a collection of classes. Each class has a set of typed variables and a set of actions. Every action is further divided into three parts: the typed parameter list, the preconditions and the effects. In contrast to STRIPS, OODDL actions work with variables rather than predicates. OODDL actions are discussed in detail later, in section 3.10.

The classes can inherit actions and variables from other classes if desired. This allows subclasses to reuse the super classes functionality and override their behaviour as necessary. Inheritance in OODDL is discussed in section 3.12.

Although OODDL is object-orientated, it is a planning domain modelling language rather than a programming language and as such, it doesn't represent all of the features of object-orientated programming languages. For instance, although inheritance and encapsulation (or more specifically aggregation) are both present and contribute to the usefulness of OODDL, abstraction, data hiding and polymorphism are not.

Abstraction and polymorphism do not have a place in OODDL because methods cannot invoke other methods. This is essentially a limitation of the underlying STRIPS target language; this is further discussed later in this chapter in section 3.12 and 3.13.

Data hiding cannot be implemented because all OODDL class variables are *public*, that is, any action can directly access any variable from any class. This is necessary because protected or private variables would require accessor functions and OODDL does not support functions.

3.4 OODDL vs. STRIPS

This section will compare and contrast OODDL and STRIPS, so that a general feeling of the scope of OODDL can be grasped in the familiar terms of STRIPS.

When domain engineers wish to describe a domain in STRIPS, they tend to think about the domain in terms of functionality. They think about all the actions that can be done and all the states that objects can be in, eventually distilling these into a list of predicates and actions.

In OODDL the domain designer can think in terms of objects and classes. The variables that concern a specific object are all found in its class, similarly so with its actions. This allows the domain designer to think, to a certain extent, about the objects in isolation.

This highlights the difference in the model used to develop with the two languages. OODDL follows the object model whereas STRIPS follows the functional model.

3.4.1 Addressing Typing Errors

OODDL's syntax is designed to allow some of the common errors of STRIPS to be bypassed. For instance, in OODDL all variables and action parameters are typed. This allows type checking of actions and stops any errors arising due to actions being applied to unsuitable objects by the planner. In STRIPS, it is possible to forget to assign a type to an object or parameter, whereas in OODDL, this will cause a parse error.

3.4.2 Addressing Omitted Negative Effects

A common error in STRIPS occurs when negative effects are omitted; this happens when a user is concerning themselves with the positive effects of an action and assumes that planner will somehow enforce trivially obvious negative consequences. It was explained by Lifschitz [Lifschitz, 1986] that the STRIPS notation of preconditions, add lists and delete lists is very sensitive to seemingly minor modifications and errors.

For instance, when attempting to create a STRIPS action which will move an truck from being at location A to being at location B, one must remember to remove the fact that the truck is at A after asserting that it is at B, otherwise it will be at both points simultaneously.

```
(:action drive-truck
          :parameters
               (?truck
               ?loc-from
               ?loc-to)
          :precondition
               (and (at ?truck ?loc-from))
          :effect
               (and (at ?truck ?loc-to))
)
```

Example of a flawed PDDL STRIPS move operator

This example is perfectly valid in STRIPS because there is no way to declare that an object can and *should* only be at once place at once. Errors like this can go undetected until the planner begins producing unexpected plans.

A simple analogy of this error in traditional imperative programming would be if when a program assigned a value to a simple integer variable, the variable could return either the old value or the newly assigned value next time it was accessed. It would hold both values at once. The difference with planners is that they will return the result that is most beneficial to the plan, in effect, exploiting the error to their advantage.

Thankfully, imperative languages do not suffer from this problem because deleting the previous value of a variable is an implicit requirement of the assignment operation. The use of variables and assignments gives a higher level semantic unit, the concept of a variable that can only hold one value, no matter how the rest of the program, or planning domain, is constructed.

In the generalised predicate logic sense, a single valued variable can be seen as a predicate that has an *invariant* stating that it only has one value. When a language supports variables, the language's compiler can enforce the variables' invariants resulting in correct typing and value maintenance. The user doesn't worry about the low-level data manipulations, they can think at the higher level.

STRIPS has no concept of these higher-level single-value variables, because it has no concept of enforcing invariants. This means that a domain engineer must form variables from the predicates and be careful to maintain whatever variable invariants they have decided on throughout the rest of their model.

3.4.3 OODDL Variables

As previously mentioned, OODDL directly supports variables and so relieves the domain engineer of the burden of maintaining simple single-value invariants. All variables in OODDL are typed; meaning an attempt to assign a variable with the wrong type will generate a parse error. Furthermore, by the nature of object-orientation, all domain variables in OODDL are bound to an owning object, akin to member variables in C++ or Java.

OODDL supports five variable types: object reference, maybe object reference, enumerated type, boolean and object-bag. The remainder of this chapter will discuss the five different types of variable implemented by OODDL, and why the need for them arose. Methods of modelling data in STRIPS and OODDL are compared and contrasted. The relations between the two are referred back to later in section 3.15 on translating OODDL.

3.5 Object References

An object reference is a way of recording a relationship between two objects. The previous sections have noted how the STRIPS method of creating a single object reference required more work to maintain its single valued constraint than the same variable would in OODDL. This section discusses the use of object references in domain descriptions and the operations that are performed on them in both STRIPS and OODDL.

3.5.1 Object References in STRIPS

In STRIPS, object references are created by use of a binary predicate. One argument is declared as the referring object, and the other is the object being referred to. Actions in the domain can instantiate this predicate with any arguments it chooses in order to establish a relationship; any action can also delete an existing predicate instance, thus removing the association.

3.5.2 Requiring the Absence of a Relationship

When an action requires that a relationship between two objects does not exist, it seems trivial enough that one would assert $\neg relation(a,b)$ as a precondition. Using a logical not in an action precondition is called a *negative precondition*.

Negative preconditions are a feature of ADL rather than basic STRIPS, if a domain is to be compatible with as many planners as possible, it should avoid the use of ADL. This means negative preconditions should be avoided and other methods for asserting the absence of a relationship must be used.

The simplest way of checking for the absence of a relationship is when it is acceptable to assert that the object isn't related to *any* other object. Using a

family tree as an example: if a person did not have a father, they could be said to not have a relationship with *any* father. This expressive power can be accomplished in STRIPS by the addition of a unary predicate to represent when an object has no relations at all:

ChildOf(offspring,father) – asserts that "father" is the father of "offspring"

NoFather(offspring) – asserts that "offspring" has no father

This would allows actions to assert preconditions such as:

NoFather(a) – "a" has no father

This dual predicate arrangement is useful because it allows the domain engineer to construct actions that require a specific object relationship, or require that there is no relationship.

One minor problem with this dual predicate model comes when an action wishes to establish a relationship, for example, if an Adopt() action wished to assign a father to a child. The child may currently have no father, in which case it would posses the NoFather() predicate, or it may already have a father, in which case it would posses a ChildOf() predicate. Because non ADL STRIPS actions do not support disjunctions, the domain engineer must create two Adopt() actions, one to handle the case where the NoFather() predicate must be deleted as ChildOf() is established, and one to handle the case where an existing ChildOf() predicate must be deleted as the new one is established. Having multiple actions that semantically achieve the same effect is wasteful, on both the domain engineer's time and the planners.

An alternative method of recording that a relationship doesn't exist is to establish a relationship with a special "null" object. For example, a special "null-father" object would exist in the domain, any child related to the null-father via a ChildOf() predicate would be seen as having no father. This means that only one Adopt() action would be needed as the NoFather() predicate would no longer be used. It would still be possible to assert that a child had no father as an action precondition by asserting that the child was a child of the specific null-father object.

To summarise, requiring the absence of a general relationship in STRIPS can be accomplished in two ways. The first uses a second predicate, this method is quite clear and easy to understand, but has more maintenance overhead if the relations need to be altered by the domain actions.

The second method uses a special null object; objects related to this object can be seen as being semantically related to no object. This method involves creating an additional object in the domain, but allows relationships to be easily altered.

3.5.3 Requiring the Absence of a Specific Relationship

The methods from the previous section are only able to establish that two specific objects have a relationship, or that an object has no relationships. These methods are useful for when an action wants to establish that there is no standing relationship before creating one. A more difficult case to encode is when an action is required to check that a specific relationship doesn't hold; for instance to check that a specific person is not the father of a specific child.

To accomplish this without the use of negative preconditions, the domain engineer, in the general case, must introduce another predicate to record the fact that the two objects do *not* have a relation.

Therefore, where previously only the fact that a child was related to a father was recorded, now the fact that the same child is *not* the child of every other father in the domain is also individually recorded. This requires the addition of a "NotChildOf()" predicate to the family tree domain.

This method can often generate a large number of predicate instances owing to the fact that the counter-predicate must be asserted in everyplace where the predicate isn't. It becomes easy for the domain engineer to make omissions that could invalidate the domain. This again shows the maintenance burden placed on the domain engineer when manually encoding domains in STRIPS.

An alternative method can be employed if the domain engineer knows that the relation is singled valued. In this case, he can make compare the current value of the relation with the object being tested; if they're not equal then the relation

doesn't exist. Testing if objects are not equal without using ADL requires a "NotEqual()" predicate to be instantiated for every pair of objects in the problem.

To summarise, object relations in STRIPS generally take the form of binary predicates. One argument represents the referring object, the other the referee. The predicate name provides the name of the relation.

The domain engineer must construct the actions carefully in order to maintain the single-valued invariant of object references. This requirement, along with the requirement of checking for the absence of relations, can lead to the extra burden of maintaining several auxiliary predicates.

3.5.4 Object References in OODDL

In STRIPS, it is necessary to use binary predicates and, in some cases, auxiliary predicates to maintain a single valued object reference. OODDL has language level support for single valued object references thus relieving the domain engineer of the burden of maintaining the variable's invariants manually. OODDL supports two forms of object reference variable: the *object reference* and the *maybe object reference*.

Object references in OODDL contain a reference to exactly one instantiated object at all times. The variable has a type associated with it and can only hold references to objects of that type or a sub class of that type. The object reference can be reassigned with any other correctly typed object by an action. An attempt to assign it with the wrong type will generate a parse error.

The object references in OODDL have no NULL value. This decision was made to assist the domain engineer when working with domain variables that must always refer to valid objects, and as such have no legitimate NULL value. From the compilation into STRIPS point of view, if an object has no NULL value then the compiler does not need to output auxiliary predicates or objects to track the NULL value, therefore declaring that a variable has no NULL value also leads to more efficient STRIPS domain generation.

The maybe object reference is very similar type of variable. It is identical to the object reference in every respect except that it may hold a NULL value. This

variant of the object reference can be used where a NULL value is needed, for instance to represent an optional object association.

The object reference variable is declared in OODDL with the following notation:

```
<object-type>        <variable-name>
```

For example:

```
Father               childOf
```

Maybe object references are declared in the similar way except the variable name is preceded with a '*'.

```
Father               *childOf
```

Both variables can be assigned and tested against other values, including the NULL value for the maybe object reference.

3.6 Managing Sets in STRIPS

Sets in STRIPS are a simple extension of the object reference; previously a single binary predicate was used to represent a single relation, here multiple instances of a binary predicate are used to represent a set.

The simple example below shows how a set could be constructed to record all the children of a given father in the family tree example:

FatherTo(father,child) – "father" is the father of "child"
FatherTo(f,a) - "f" is father to a, b and c
FatherTo(f,b)
FatherTo(f,c)

Sets are generally quite easy to work with in STRIPS, however a problem can arise when one wishes to stop duplicates being created in the set, i.e. to maintain a *strict* set rather than a *multi-set*[6]. To accomplish this, actions must check that the set relation doesn't already exist before allowing it to be created. As discussed in section 3.5.3 previously, this can cause problems for the domain

[6] Multi-sets are also known as bags

engineer, as it requires a counter-predicate to be instantiated for each object not in the set.

The most common way of maintaining a strict set it to mark objects that aren't in a set with a unary predicate. Then, only objects that have this predicate can be added to a set and when being added their availability predicate is deleted. Conversely when being removed they are marked as available again.

This has the side effect of allowing objects to be placed in only one set across the entire domain, useful for the previous example because a child can only have one father and so can only be in one FatherTo() set. This is the method used in section 3.5.2 previously, for checking that a child is not related to *any* father.

One hard to address problem with sets is that it is very difficult to test if the set is empty. To achieve this, a "set-empty()" predicate would have to be added when the last element is deleted from the set by an action. This is difficult because there is no way of counting how many elements are in the set and so the action cannot tell if the element being removed is the last one. This is a limitation of basic STRIPS that cannot be addressed without an exponential blow up in the number of actions [Gazen and Knoblock, 1997], or by implementing a specialised element counting scheme that is maintained in parallel to the set. Either method places a burden on the domain engineer.

In summary, STRIPS sets are very similar to object references. It is very easy to create a multi-set, where relations are added and removed without any special invariants to maintain. It is simple enough to maintain a strict set if set membership can be mutually exclusive with membership of other sets. Maintaining strict sets without this limitation can be accomplished by use of counter-predicates to indicate non-membership of individual sets. It is very difficult to test if a set is empty in an action precondition.

Any of these methods introduces a number of auxiliary predicates and places the onus on the domain designer to attempt to maintain the validity of the data model.

3.7 Managing Sets in OODDL

OODDL provides language level support for the multi-set, or object bag, discussed in the previous section. Support for the more specialised strict set hasn't yet been needed and so has not been implemented in OODDL at this stage. This could be added at a future date by the addition of a strict-set variable type that generates STRIPS counter-predicates as discussed in the previous section and enforces the single set entry invariant.

OODDL permits actions to add or remove objects of the correct type, or a sub class of the correct type, from the object bag. One cannot test if an object is not in the bag due to the problem of establishing non existent relationships for multi-sets in the compiled STRIPS domain description, as discussed in the previous section. This feature could be added to OODDL at a future date.

The object bag in OODDL is denoted by the name of the domain type it holds, followed by a variable name suffixed with a pair of square brackets ('[]').

For example:

```
Child           fatherTo[]
```

3.8 The Enumerated Type

A common domain feature in planning is the use of enumerated types. An enumerated type is a type that has a fixed collection of named discreet values. An example might be if objects in a domain could be one of a fixed set of colours. This set of colours would form an enumerated type, "colour", and could have the values red, green or blue.

Only values from the same enumerated type can be assigned to enumerated type variables. The enumerated type is useful for domain constants that do not vary from problem instance to problem instance.

3.8.1 Enumerated Types in STRIPS

There are two common ways of representing enumerated types in STRIPS. The first is to use a collection of unary predicates that encode the value in the predicate name, for example:

```
Object_Is_Red(x) - object x is red
Object_Is_Blue(x) - object x is blue
```

The second is to use binary predicates to separate out the value from the variable name:

```
Object_Colour(x,c) - object x is colour c
```

"C" would be a type of object that would possess one of the following predicates:

```
Red(c) - Used only by enum value objects
Blue(c)
```

Here special objects exist in the domain whose sole purpose is to be referenced as enumerated values. They are differentiated by the unary value predicates Red() and Blue().

Each of these methods has their advantages and disadvantages. A problem with the first method is that it doesn't scale very well. If the requirements were changed a little to require that all objects have two distinct colours recorded for them, a second set of predicates would be needed. For example:

```
A_Red(x) - object x's A colour is red
A_Blue(x) - object x's A colour is blue
B_Red(x) - object x's B colour is red
B_Blue(x) - object x's B colour is blue
```

Another problem with this method becomes apparent when actions are constructed to change the values of these variables. STRIPS actions cannot parameterise the names of predicates in their effects list; because the value of the enumerated type is encoded in the name of the predicate, separate actions must be written to change between any pair of colours:

```
ColourAFromRedToBlue(x) - Changes object x's A colour
                          from red to blue
ColourAFromBlueToRed(x) - Changes object x's A colour
                          from blue to red
ColourBFromRedToBlue(x) - Changes object x's B colour
                          from red to blue
ColourBFromBlueToRed(x) - Changes object x's B colour
                          from blue to red
```

These restrictions mean that this form of enumerated types is good only where it isn't necessary to reassign the colour variables, or where there are very few possible values meaning that the number of reassigning actions are small.

The benefits of this method lie in its simplicity. If the values do not need to be reassigned then this is a simple and effective way of implementing enumerated types.

The second method of accomplishing the enumerated type in STRIPS is to use binary predicates. This method requires additional objects to be added to the problem description, along with predicates to make each unique from the others. By having what are essentially object reference variables referring to these objects, it's possible to hold an enumerated value in a more flexible way. For instance, it is now possible to construct an action that can receive an enumerated type, such as colour, as a parameter:

```
ColourAChange(x,prevcol,newcol)
ColourBChange(x,prevcol,newcol)
```

The domain engineer now has a lot more freedom with his enumerated type, but he would have to be careful to restrict the values of "prevcol" and "newcol" in the action preconditions to ensure they're one of his colour objects and not some other arbitrary object. He must also ensure, as with the previous method, that the actions maintain a single value for both "A" and "B" colour variables.

3.8.2 Enumerated Types in OODDL

OODDL provides direct support for enumerated types, simplifying the job of the domain engineer considerably in cases like the above. The previous example would be encoded into OODDL by first declaring an enumerated type colour:

```
Enum Colour = { red, green, blue }
```

Now any class can have an enumerated type variable of type "Colour" as a member variable. Colour becomes a valid type for passing to actions and so it is trivial to construct actions for reassigning colours, and because all actions and variables in OODDL are typed, the domain engineer doesn't need to explicitly verify value types before assignment.

```
type ColouredOb
     Colour          colour-a
     Colour          colour-b

     ColourAChange(Colour newCol)
          e: colour-a:=newCol
     end
     ColourBChange(Colour newCol)
          e: colour-b:=newCol
     end
end
```

An OODDL class[7] with two colours made from enumerated types

OODDL also supports ordering on the enumerated types. Expressions can be formed to test if one enumerated value is greater than, or less than, another is. For example:

```
Enum CardValue = { ace, two, three, four, five,
six, seven, eight, nine, ten, jack, queen, king }
```

[7] The notation used here for the OODDL class is explained in section 7.1 in the appendix.

Actions that needed to ensure that a card was greater than another before it used it could use an OODDL precondition like the following:

```
p: card1.value < card2.value
```

Where "value" is an enumerated variable of type "CardValue".

STRIPS has no support for ordering in this form, all ordering must be explicitly encoded in the form of predicates. For example, more-than(two,ace), more-than(three,ace), ... This technique generates an exponential number of predicate instances, and takes effort to maintain by hand in STRIPS.

OODDL also predefines the commonly used *boolean* enumerated type. This means that domain engineers can make use of boolean variables without having to manually define the type themselves.

3.9 Actions in STRIPS

Domain actions are what allow a planner to form a plan; they affect the domain facts and thus change the world state. The only changes possible to a domain are those made by actions. Encoded into actions are the laws of physics for the domain, by carefully constructing the actions the domain engineer implicitly encodes the physics of the domain.

Actions in basic STRIPS are untyped; any typing restrictions for the parameters must be made in the action preconditions. STRIPS actions were discussed in detail in section 2.5.1.

3.9.1 The Light Switch Domain in STRIPS

A simple example of a STRIPS domain is declared here for comparison purposes with OODDL:

```
(define (domain light-switch)
     (:requirements :strips)
     (:predicates
          (on ?l)
          (off ?l)
          (light ?l))

     (:action switchon
          :parameters (?l)
          :precondition (and
               (off ?l)
               (light ?l))
          :effect (and
               (on ?l)
               (not (off ?l))))

     (:action switchoff
          :parameters (?l)
          :precondition (and
               (on ?l)
               (light ?l))
          :effect (and
               (off ?l)
               (not (on ?l))))
)
```

Light switch domain in PDDL STRIPS

The above domain makes use of a typing predicate *light()* to ensure that the parameter passed is of the correct type. In domains like this that feature only one type, typing predicates would normally be omitted.

Another case where the typing predicates are often omitted is if the action's other preconditions can only be true of a certain type anyway. For example the light_on() predicate above would only ever be given to the "light" type and so this establishes the type of the parameter. However, sometimes typing-predicates are added anyway, because their omission can make the domains less clear to a human reader.

3.10 Actions in OODDL

OODDL actions are quite similar to STRIPS actions in that they have a parameter list, a preconditions list and an effects list. However, unlike STRIPS, OODDL actions are bound to an owning class.

In STRIPS, an action could only refer to its parameters. In OODDL, when an action is instantiated there is a concept of the *this* object which, as with normal object-orientated languages, can supply the action with values without the explicit use of parameters. All member variables for the class are in the scope of the action and so the action can freely refer to them in its preconditions and effects. An action can also dereference its parameters in order to refer to their member variables too.

When a STRIPS action needs to assign a new value to an existing relationship, it requires the previous value to be passed as a parameter in order to delete the existing relationship. This results in the use of an extra parameter that may not be used for anything except deleting an old relation. For instance, instead of having an action with the title:

```
move(truck,destination)
```

We have:

```
move(truck,source,destination)
```

Because OODDL uses member variables that are in scope throughout the action, the previous value does not need to be passed in as a parameter. Instead, the member variable can be referred to directly, without the need for any extra parameters. This results in a more concise action definition.

The parameter lists in OODDL actions are typed and so the domain engineer doesn't need to worry about objects of the wrong type being passed to the

action. This reduces the number of preconditions the domain engineer has to maintain.

The preconditions are a list of boolean expressions written in a similar way to expressions from an imperative language. In OODDL, preconditions are made up of operations such as testing for equality, or establishing set membership rather than a list of predicates as with STRIPS.

Again, disjunctive preconditions are not allowed, only conjunctions. This is for simplicity when compiling into STRIPS, which doesn't support disjunctive preconditions. Disjunctive preconditions could be added to OODDL at a later date.

The effects are a list of operations on the parameters and the member variables of both the owning object and the parameters. Instead of consisting of predicates that are added and removed, the effects in an OODDL action are higher-level operations such as variable assignments or set insertions.

The effects in OODDL actions are seen as happening in parallel and so the ordering of the effects in the action is unimportant. If a variable is assigned a new value and then the variable is used in an expression further down the action, it is the original value that will be used, not the new one. This decision was made to ease the translation into STRIPS where all effects happen in parallel and there are no intermediate states in an action. However, a future version of OODDL could respect the ordering of expressions in actions, by having a special STRIPS translator that emulates ordering and intermediate states.

3.10.1 The Light Switch Domain in OODDL

The following is the light switch domain from section 3.9.1 in OODDL.

```
type Light
     boolean    on

     switchoff()
          e:on=false
     end
     switchon()
          e:on=true
     end
end
```

Light switch domain in OODDL

In contrast to the STRIPS domain, the OODDL domain appears clearer. There is no need for the type checks and so there are fewer preconditions in the actions. Because OODDL supports atomic assignments, the separate and delete effects from the STRIPS version have been collapsed into a single statement.

3.11 Inheritance in STRIPS

Although STRIPS is not an object-orientated language, it does allow inheritance of a sort. Section 2.5.2 discussed how typing is generally accomplished in STRIPS using unary predicates. Actions will check in their preconditions that an object has the correct type predicate before operating on it.

To implement inheritance, all that needs to be done is to take all the predicates from one type (including the typing predicate) and give them to an object from another type. Now the object implements both types and can be passed to actions from either type.

This is probably better classified as *unordered inheritance*, as it yields the same results no matter which type one views as the super type. Ordering however only becomes important when features such as overriding are allowed, but overriding isn't applicable in a flat modelling language like STRIPS.

3.12 Inheritance in OODDL

OODDL is an object-orientated language and so inheritance plays a more important role as a modelling method than it does in STRIPS. Classes in OODDL can inherit from one or more superclasses by using *multiple-inheritance.*

All attributes from each of the superclasses become part of the subclass. If methods have the same name then overriding takes place, and the most recently defined action, in terms of the inheritance chain, takes precedence. If member variables are redefined, an error is generated.

Polymorphism is a popular feature of object-orientated programming languages, however it is not appropriate to OODDL. Polymorphism allows a client object to call another object which is believes to be of a certain type. The receiving object however can be a subclass of that type, and if the subclass has a method with the same signature as the one being called, it is called instead. This means that when a method is called, the caller cannot be sure which class's methods will actually be called, only that the type that eventually is called will be the same as, or a subclass of, the type expected.

In OODDL, actions cannot invoke other actions and so there is no opportunity for polymorphism to take place. The planner is the only agent calling actions and it will select the action it deems appropriate based on the action definition. Overriding still has a place however, as that essentially stops an action being applicable to a sub class and replaces it with another, generally more specialised one.

3.13 Implementing Method Invocation in OODDL

It was mentioned in the previous section that actions couldn't invoke other domain actions as part of their effects. This is a fundamental feature of STRIPS, on which OODDL is based. In STRIPS all actions' effects are completely defined as a list of positive and negative predicates, to invoke another method would presumably mean the intention was to have that action's effects applied also. The action's preconditions would have to be met as well, so either they would have to be combined into the calling action's preconditions or the action

being called would only take affect if the preconditions were met at the time of invocation. Whichever option is taken would have increased the complexity of STRIPS and made implementing a STRIPS parser unnecessarily difficult.

Then there are other issues related to recursion and such which begin to create more problems, at the end of the day, invoking other methods in STRIPS is a lot more effort that it's worth.

OODDL builds upon STRIPS, if OODDL allowed method invocation then STRIPS would have to be extended to support it, or OODDL's compiler would have to emulate it by clever domain encoding.

The decision was to not have method invocation in OODDL either, it would be next to impossible to implement correctly on top of STRIPS and any extensions to STRIPS would require planners to be updated or rewritten to work with the new syntax.

With the absence of method invocation, abstraction and polymorphism lose their value and so OODDL does not support them either.

3.14 Draughtsman

It was mentioned in section 2.7 that an earlier version of the Draughtsman domain engineering tool edited STRIPS domains through a GUI. With the development of OODDL, a new version of Draughtsman has been created. This version allows OODDL domains to be edited through a CLI interface and then translated into PDDL STRIPS; this translation is outlined in the next section.

Draughtsman can also be compiled into a library that exports an API for parsing and manipulating OODDL domains, allowing existing tools to exploit OODDL more easily. This functionality has been demonstrated by attaching a Java GUI to Draughtsman and using it to edit and translate OODDL domains. Because Draughtsman uses wizards to help the user create domains, all the OODDL domains generated are guaranteed to be syntactically correct.

3.15 Overview of the OODDL to STRIPS Translation

The goal of OODDL is to give domain-engineers an object-orientated domain modelling language that is both easier to use and more powerful than STRIPS.

However, all of these aims are moot unless planners can make use of it. OODDL could theoretically be translated into languages such as OCL or ADL, but the most widespread language is currently PDDL STRIPS.

The translation between OODDL and STRIPS converts the object-orientated domain model, complete with class hierarchies and variables down to a "flat" predicate based STRIPS model. The algorithm used to do this is outlined in this section. There are five elements of the OODDL domain that contribute to the STRIPS encoding, these are: classes, enumerated types, variables, actions and the problem specifications.

3.15.1 Classes

Translating an OODDL domain's class hierarchy to STRIPS is the most trivial part of the translation. It was decided to target untyped STRIPS rather than typed STRIPS to keep the potential audience as large as possible. Planners that can deal with typed STRIPS can deal with untyped STRIPS, but not vice versa.

To translate the class structure, a single argument STRIPS predicate is declared for each class in the domain. Problem domain objects that are an instance of, or a subclass of, a particular OODDL class will possess this predicate. Actions will be able to type check their arguments by requiring a specific class's typing predicate in their preconditions.

3.15.2 Enumerated Types

The various ways of implementing enumerated types in STRIPS were discussed in section 3.8.1. OODDL will utilise the second approach that was discussed in this section; each enumerated type value will be translated into a problem domain object with a unique unary predicate to identify it. This allows STRIPS action's to directly refer to these values by name by requiring this unique predicate of an argument in the action's preconditions.

One of the features of OODDL's enumerated types is the ability to create ordered enumerated types, where OODDL expressions with ">", ">=", "<" and "<=" operators can be used. To record these relationships between the enumerated type value objects in the STRIPS domain, four binary predicates are

declared called "greater(a,b)", "greatereq(a,b)", "lessthan(a,b)" and "lessthaneq(a,b)".

These predicates are instantiated for the relevant values in the enumerated type when the STRIPS problem specification is written out. STRIPS actions can then test the relative value of enumerated type values by requiring these predicates in their preconditions.

3.15.3 Variables

OODDL class member variables relate a particular class instantiation and variable name to a value. In general, these relations can be represented by binary predicates where the first argument is the owning object (the instance of the class) and the second is the value.

To translate OODDL class member variables into STRIPS, a predicate is generated for each class variable formed from the class name and the variable name. These binary predicates address all OODDL's variable types: object reference, maybe object reference, object bag and enumerated type. Enumerated type variables, including the boolean type, are simply object references to OODDL generated problem objects, as discussed in previously in section 3.8.1.

The maybe object reference is identical to the object reference variable discussed in section 3.5.1, except it uses one OODDL generated problem object to represent the null value. The maybe object reference's implementation is described in section 3.5.2.

3.15.4 Actions

Actions in STRIPS consist of three parts, the parameters, the preconditions and the effects. The preconditions and effects are lists of domain predicates grounded by the action parameters.

OODDL has a similar action structure, but OODDL's preconditions and effects are made up from *expressions* formed using the parameters, member variables and dereferences thereof. This means that OODDL often makes use of temporary values that have to be made explicit on the conversion to STRIPS. The simplest of these is the "this" parameter, the object that the action is bound

to in the OODDL domain; this must be made explicit in the STRIPS domain and have preconditions generated for it to ensure it is of the correct type.

Further examples of generated parameters can be seen in the following example, which assigns an enumerated type variable "colour" with an enumerated type value "green".

```
e: colour:=green
```

As discussed previously, enumerated values like this are represented as objects when converted to STRIPS, so "green" has to become a parameter, with associated preconditions, to ensure that it can only be matched to the unique "green" object in the domain. Furthermore, the previous value of the "colour" variable has to be passed in so that the existing fact associating "colour" with it can be deleted.

OODDL expressions consist of three parts: the left side (the lvalue), the right side (the rvalue) and the infix operator. Each of these can generate parameters and preconditions in the resulting STRIPS actions. The expression operator can also generate effects.

If the lvalue and rvalue terms make use of dereferencing, e.g. "destination.surface.clear==true", then this will mean additional STRIPS parameters and preconditions have to be generated in order to get at the final dereferenced value. Using the above example, "destination" is a variable of the "this" object; it has a value that is an object, this in turn has a "surface" variable. The surface variable has a value that is an object, finally this object has a boolean variable called "clear", which also has value associated with it. The expression requires this value to be obtained and compared with the boolean value of true.

To generate STRIPS code for this lvalue term, three additional parameters must be generated (one for each dereference) and linked via preconditions to access the final dereferenced value. The STRIPS code for this expression would look like:

Parameters needed:

```
this, this_destination,
this_destination_surface,
this_destination_surface_clear
```

Preconditions needed:

```
MyType_destination(this,this_destination)
Table_surface(this_destination,this_destination
_surface)
Surface_clear(this_destination_surface,this_des
tination_surface_clear)
Boolean_true(this_destination_surface_clear)
```

This dereferencing technique allows access to any member variable and any dereferenced variable. Each expression generates its preconditions, effects and parameters independently of the other expressions in the action. By using a naming convention for the implicit parameters that are generated, and collapsing multiple needs of the same named parameters into one, it becomes possible to avoid duplicated parameters and preconditions when generating the action.

Once the appropriate lvalues and rvalues are obtained, the next stage is to generate effects. In general, the translation from OODDL infix operators to STRIPS effects is straightforward. For example, an object reference assignment will generate an add effect and a delete effect. The delete effect will remove the lvalue's association with the rvalue, and the add effect will create a new association with the new value. The assignment of enumerated types, including booleans, follows exactly the same structure. Object bag addition and removal operators simply omit the delete effect or the add effect respectively.

3.15.5 Overriding Actions

Overriding is a popular feature in object-orientated languages, it allows a subclass to selectively alter the behaviour of its parent by replacing its methods. As discussed in sections 3.12 and 3.13, the lack of method invocation in OODDL means polymorphism isn't applicable, however overriding is.

To implement overriding in the generated STRIPS domain, it must be ensured that an action can be applied to the owning class and its subclasses, but not to the subclasses (and their subclasses) for which an override has been defined. To accomplish this, each action in the OODDL domain is given a unary *enabling predicate* that objects must posses in order to be passed as the action's *this* parameter.

By using action specific enablers in this way, it is possible to turn individual actions on and off on an object-by-object basis in the generated domain. By coupling this with the inheritance hierarchy from the OODDL domain, actions that have been overridden can be turned off by not giving class instances the action enablers needed. Action enablers are granted for all the actions in the object's class, and all the actions in the inheritance hierarchy that have not been overridden.

3.15.6 Problem Specifications

The STRIPS problem specification contains two things, firstly it contains the actual OODDL problem in a STRIPS encoding, and secondly it contains meta-data needed to ensure the STRIPS domain functions correctly.

An OODDL problem specification has a value for every variable of every instantiated object. Given that the predicates for the class variables have already been generated, all that must be done is to instantiate these predicates with the values defined. A STRIPS object is generated for every OODDL domain object, and predicates are instantiated to record the initial state of every variable. Goal state predicates are also instantiated for those variables with goal states defined.

Besides the data from the actual problem specification, additional meta-data is also required. Meta-data is static data that must also be added to the STRIPS domain to ensure that certain actions in the OODDL domain function correctly. The majority of this is generated from OODDL's enumerated types. They require objects to be generated in the problem specification to represent each value in the enumerated type. Furthermore, instances of the relativity predicates such as greaterthan(), lessthan() etc. must also be generated.

The remaining static meta-data is generated from unary typing predicates, unary action enabler predicates and null values for maybe object references.

3.15.7 Resultant Plans

A STRIPS domain generated from OODDL contains names and predicates that are generated by the translator. Subsequently, it may not be clear from the resultant STRIPS plan what the actual OODDL plan is.

The names generated for STRIPS actions and predicates follow a rigorous naming convention, this means that instantiated actions can be translated back into an OODDL encoding with no reference to the original domain.

For instance, the OODDL generated logistics domain may generate the following plan:

```
Time:1
     carrier_load1(truck1,package1,po0,po0)
     truck_drive-truck1(truck2,airport1,po1,road1)
     aeroplane_fly-plane1(aeroplane1,airport0,
                                          airport1)
Time:2
     truck_drive-truck1(truck1,airport0,po0,road0)
Time:3
     carrier_unload1(truck1,package1,airport0)
Time:4
     carrier_load1(aeroplane1,package1,airport0,
                                          airport0)
Time:5
     aeroplane_fly-plane1(aeroplane1,airport1,
                                          airport0)
Time:6
     carrier_unload1(aeroplane1,package1,airport1)
Time:7
     carrier_load1(truck2,package1,airport1,airport1
)
```

```
Time:8
     truck_drive-truck1(truck2,po1,airport1,road1)
Time:9
     carrier_unload(truck2,package1,po1)
```

Plan found for an OODDL generated STRIPS domain

This can be automatically converted to OODDL notation by using the first parameter of each action as the "this" object, and dropping all the implicit parameters until only the original OODDL ones remain. The number of parameters originally in the action is appended to the end of the generated action for just this purpose. Dropping the number from the end of the name and removing the class prefix to the left of the underscore will translate the action names back to their original OODDL encoding.

```
Time:1
     truck1.load(package1)
     truck2.drive-truck(airport1)
     aeroplane1.fly-plane(airport0)
Time:2
     truck1.drive-truck(airport0)
Time:3
     truck1.unload(package1)
Time:4
     aeroplane1.load(package1)
Time:5
     aeroplane1.fly-plane(airport1)
Time:6
     aeroplane1.unload(package1)
Time:7
     truck2.load(package1)
Time:8
     truck2.drive-truck(po1)
```

```
Time:9
        truck2.unload(package1)
```

OODDL notation plan from the logistics domain

A simple shell script could be written that takes an OODDL domain, translates it to PDDL using Draughtsman, runs a STRIPS planner on it, and then translates and displays the resultant plan in OODDL notation. This functionality could even be executed from inside Draughtsman.

3.16 Summary

This chapter has detailed the development of the object-orientated domain description language: OODDL. OODDL is designed to be an easier and more powerful language for domain engineering than PDDL STRIPS.

OODDL is designed to lift the task of modelling a domain from the level of the predicate literal to the level of the object. It does this by employing the use of class definitions, variables and actions formed from lists of imperative statements, rather than add and delete lists. The motivation for OODDL's syntax and semantics is based upon common traits in existing domain encodings, especially STRIPS. This chapter has explained OODDL's features in relation to STRIPS, using case studies where appropriate.

4 Design of GTL

Domain analysis has demonstrated the usefulness of recognising common generic behaviours in planning domains in the form of *generic types* (see section 2.6). A domain exhibiting generic types can be planned with more efficiently because known heuristics and solvers can be employed that work with the generic types directly.

However, generic types can also be used in domain engineering, where they give higher-level semantic meaning to an otherwise abstract domain description. Once a generic type is identified, it is possible to deploy context sensitive sub-editors or to introduce terminology that is more relevant to the editing session.

This chapter first discusses some of the different ways a domain-engineering tool could implement generic types, particularly with reference to OODDL; it then moves on to the development of a generic type description language called GTL for allowing domain-engineering tools and planners alike to work with arbitrary generic types.

4.1 Modelling with Generic Types

Generic types are very useful for domain modelling because they represent a concept to the domain engineer. For instance, a *mobile* type (see section 2.6.1) represents the concept of something that moves around on a map of locations. If a domain-engineering tool allowed the user to manipulate the domains in terms of these concepts then the user could work at a higher level, in terms that are more representative of the true meaning of the domain.

For example, if the domain-engineering tool was able to identify the mobile and location relations, it could allow the domain engineer to edit the map of locations using a graph editor rather than by editing the relations directly. This would allow the engineer to visualise the domain more effectively.

4.1.1 Generic Types as Superclasses

A domain object that matches the criteria of a generic type could be seen as implementing the generic type's behaviour as a sub set of its own. In a sense, the object could be viewed as having inherited the behaviour from a superclass.

This leads on to the simplest way of implementing generic types in OODDL, which is to simply generate ready-made generic types classes for the engineer to derive from. This means the functionality of the generic type is captured in the superclass and the subclass must simply extend and customise the type as it sees fit.

Unfortunately, this approach has its problems, the main one being the lack of flexibility it offers. The default actions that operate on the generic types, such as the *move* operator for the mobile, often cannot be used in a stock, fixed manner. The actions frequently must be augmented with additional effects or preconditions to ensure that they work in harmony with the rest of the domain. To an extent this can be accomplished from the subclass by overriding, however sometimes this is not possible because the domain requires an action with a different signature[8] from the existing one; for example an additional parameter might be needed for a *move* operator. When this is the case, the original default action must be edited, or somehow disabled, to ensure the planner does not invoke it.

This often involves substantial re-engineering of the generic type classes to an extent that negates the usefulness of having them auto-generated in the first place. This problem can be partially addressed by the use of wizards to customise the generation of the generic types' actions. The wizards would present the user with a series of options for the addition of extra preconditions or effects; however, this method becomes increasingly cumbersome as more options are added, although the flexibility increases accordingly.

If such wizards were implemented in Draughtsman, the wizards would have to mark the generic types so Draughtsman would be able to recognise them and

[8] The signature of a method is a combination of its name and its typed parameter list. For example: drive(Truck t, Location from, Location to). If a subclass defines a method with the same signature as a superclass's method, it is overridden.

deploy the correct context sensitive editor on them. In turn, Draughtsman would have to protect the class in order to ensure that further editing does not destroy the behaviour that makes it meet the criteria of the generic type it implements. This could conflict with the user's ideas or designs, for example stopping them removing an action or property that is no longer needed, but is still required for the generic type to meet its criteria. This could happen if during the editing cycle, a type initially declared as a generic type turns out not to be a generic type at all.

A further problem with the superclass approach is revealed when the user doesn't use the generic types to build the domain; this could happen if they fail to recognise that their type meets the criteria of an existing generic type. If the wizards were not used to create the generic type based classes, then features that would allow the user to manipulate the domain more easily would not be enabled.

Further problems can also be caused if the engineer wants a type to extend multiple generic types, for example, when a type is seen as both a *portable* and a *carrier*. Mixing two generic types is not always trivial as sometimes there are subtle interactions between the generic types that must be addressed, but again this could be solved programmatically via wizards.

Using wizards in this way would however cause problems, the main one being the difficulty in writing the wizards due to the large number of possible combinations of generic types. Furthermore, adding new generic types would mean writing a new wizard and probably updating the older wizards to offer new combinations with the new generic type.

In summary, superclasses could be used to implement generic type editing in Draughtsman. The user would import stock generic types from a ready-made source by use of specialised wizards. The wizards would question the user in order to cater and combine existing generic types into a customised edition for the domain in question. The resulting classes would be tagged as generic types and Draughtsman would then be able to deploy specialised editors on them, such as map editors or visualisation aids. Draughtsman would have to protect the

resultant classes in some way to ensure they maintained the qualities that qualify them as generic types and allow the specialised editors to work with them.

4.1.2 Automatic Generic Type Recognition

The TIM tool, discussed in section 2.6, is able to analyse STRIPS domains and identify generic types. The resulting analysis is used to improve planner performance by allowing the deployment of specialised sub-solvers and exploitation of tailored heuristics. If generic types could be recognised automatically in Draughtsman, the user wouldn't need to first declare which generic types their domain uses.

TIM identifies its generic types by using hard coded recognition algorithms for each of its supported generic types. It also allows these algorithms to be employed by planners by providing an API to analyse the domain and provide access to each of the generic types. Unfortunately TIM cannot be used directly on OODDL domains, they must first be compiled into STRIPS. The problem with this is that it is often difficult to relate the resulting STRIPS analysis back to the original OODDL, making applying the TIM analysis to OODDL quite difficult.

Given that STRIPS is the domain engineering standard, the fact that TIM can't work with other languages is not really a shortcoming of TIM, but it does mean TIM cannot be directly applied to OODDL domains.

A solution to this would be to rewrite TIM's generic type recognition algorithms to work with OODDL domains. This would result in a duplicate code base, one designed to recognise each generic type in STRIPS and one for each generic type in OODDL. This approach has the consequence that the addition of new generic types would mean updating both the STRIPS and OODDL recognition algorithms.

The next evolution of this idea would be for TIM to use a declarative model for describing generic types. This would separate the description of the generic types from their recognition. As a result, this would require a single general-purpose generic type recognition algorithm for each domain language supported by TIM, the algorithm would apply generic type descriptions to the source

domain. TIM would have to report its results in syntax applicable to the source domain language; for example, predicates based results for STRIPS and class-based results for OODDL.

If TIM could be applied to OODDL domains then the domain engineer could reap additional benefits besides generic type recognition. TIM would be able to provide the same invariant extraction in OODDL as it does currently for STRIPS. This would allow the user to see a state model implied by their domain description, allowing them to identify and correct any errors in the domain actions.

This provides an opening for much future work on TIM that could mean it becomes as valuable a tool in the construction of OODDL domains as it has proved itself to be with STRIPS domains.

4.2 The Declarative Model

The problem of having to write the same generic type recognition algorithm for different domain description languages can be addressed if the generic type declaration is separated from the generic type recognition algorithm.

One implementation of this idea would be to develop a language independent way of describing generic types and then to apply this to both STRIPS and OODDL. This method would have the additional advantage of easy extendibility; a new generic type could be added by simply adding a new description to the descriptions repository.

It was mentioned earlier that generic types have benefits to planners in that they supply heuristics or recommend sub-solvers for problems. Using a declarative model, information such as heuristics or recommended sub-editors could be embedded within the description. For instance, if a domain expert recommends a particular heuristic for a domain, then a new generic type could be added to the repository with the heuristic attached, this would allow the domain specific knowledge to be exploited by the domain independent planner. In addition, perhaps crucially, other domains that can be seen in the abstract to implement the generic type would also benefit from the heuristics.

The same applies to sub-editors, if a domain expert recommends the best way of working with a particular domain type is through a graph editor, then a generic type could be added to recognise the type and recommend a graph editor interface for editing some of its parameters. Recommended heuristics, sub-solvers, sub-editors and so on provided by the generic type are called *services*.

Extendable generic types are very useful for domain-engineering and planning alike, they represent a valuable addition to the whole process from domain engineering through to resultant plan.

4.3 The Templated Approach

If a declarative model is to be used to recognise generic types and supply manipulation information for the domain-engineering tool, the task now becomes one of finding an effective way to describe generic types. One method would be to use some form of *template* to describe the generic type.

The templates could be used either to spot generic types in fully or partially constructed domains, or to create new classes implementing a given generic type. If generic types could be spotted as the domain engineer builds the domain, then the editing session could be focused by the application of a specialised sub-editor.

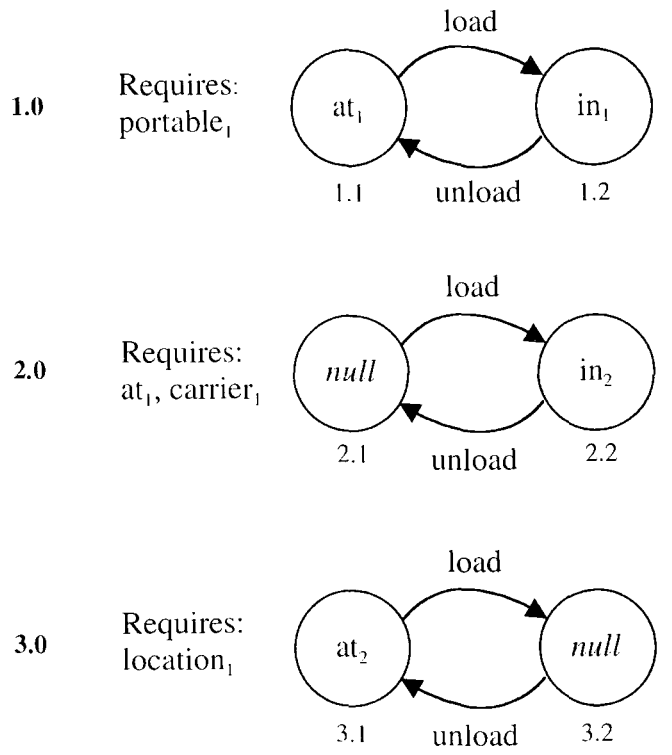
Because new templates can be added, this solution is extendable; furthermore, it has the additional benefit of standardising the representation of all generic types, allowing them to be accessed through an API that need not be specialised for each generic type.

A template approach presents two problems, identifying generic types in a domain and describing generic types in a template language.

4.3.1 Finite State Machine Representation

One approach to describing generic types is to show a generic type's behaviour diagrammatically using *finite state machines or FSMs*, as shown by Fox and Long [Fox and Long, 1999]. FSMs offer a concise way of showing the states that a generic type exhibits, and the transitions, via high-level actions, that

connect them. The example below shows a STRIPS FSM[9] representation for *portables* and *carriers*, as described in section 2.6.2.



FSM based definition of a carrier generic type; the names of the properties and actions are arbitrary, only the behaviour is important

This diagrammatic representation is good for graphically explaining the functioning of the generic types, but it isn't exact enough to precisely capture the complete picture. For example, the three FSMs depicted are intricately linked, when one FSM undergoes a transition, the other FSMs must also make a transition.

In this case, when a *load* action is executed, the portable will traverse the first FSM moving from 1.1 to 1.2, because its at_1 property will be deleted and it will gain an in_1 predicate instead. Meanwhile, the carrier will move from 2.1 to 2.2 on the second FSM as it gains an in_2 property. The in_1 and in_2 properties are created together because they are part of the same add effect; they are *related properties*. Similarly so, the at_1 from 1.1 is related to the at_2 from 3.1 and so this FSM must also make a transition as part of the load action.

[9] FSMs of this form are also known as property space diagrams. A subscripted predicate means that the object in that state appears in a grounded predicate at that argument position.

The problem of expressing the relationships between the FSMs gets worse when its realised that it cannot be any carrier, any location and any package undergoing these transitions simultaneously, the objects undergoing the transitions must also be linked. The location object that is traversing FSM 3.0 must be the one related to the portables at_l property in state 1.1, and the carrier's at_l in 2.0. This defines that the carrier must be at the same location as the portable that it is loading.

FSMs are good for diagrammatic representation, but they would not be able to define a generic type without a significant amount of meta-data describing the relations between the FSMs.

4.3.2 The Components of a Generic Type

In order to capture a more holistic view of the generic type's behaviour, it is necessary to formalise the generic types and break them into separate representable components. From examination and study of the currently available generic types, it was decided that a generic type can be seen as consisting of three main components:

- Variables
- Actions
- Services

Using the aforementioned transportation example, a carrier has two variables, an *at* variable defining its location and a *contents* variable describing its current load.

It has three high-level actions: one to *move* it around a network of locations, where the current location is defined by the *at* variable, and two more that allow it to *load* and *unload* portables from the same location into the *contents* variable.

A high-level action need not map onto a whole action in the target domain language, it could simply map onto a small section of it, or an expression inside it. This handles the case where an action may do several things and only part of it is the high-level generic type action.

The portable has two high level variables only: the location it is *at* and a boolean *locking* condition ensuring it cannot be loaded into two carriers at once. The definition of *load* and *unload* ensure that the portable is locked if it is loaded in a carrier and unlocked if not.

The variables and actions are enough to describe and capture the behaviour of a generic type; any domain object that matches the behaviour can be mapped onto the description and declared a generic type.

The third and final part of the generic type is the *services* it provides. These are things such as heuristics, state models or manipulation hints for domain engineering. For the transportation generic types, the heuristics may advise that a portable is not loaded into a carrier when it is already at its goal location for example.

A state model could be overlaid that declares states such as *loaded* and *unloaded* for the portables. This would allow them to be manipulated in terms that are more familiar by the domain engineer. Manipulation hints may advise that the network of values traversed by the *at* variables of the portables and carriers are represented by a graph where locations are represented by nodes and traversable paths by edges.

In summary, the template language will need to represent high-level variables that map onto whatever lower level representation is employed by the domain language; that would be variables in OODDL and predicates in STRIPS. It would also have to describe high-level actions that manipulate these variables, these actions would map onto parts of actions in the source domain.

Finally services will have to be described, it is likely that each service description will have to be designed on a case by case basis, however it is likely that they will all refer to the high level variables of their generic types.

4.4 A Case Study of Generic Types

Any candidate generic type description language must be at least able to describe the existing generic types. Ideally, it should be flexible enough to allow a domain expert to describe generic types that haven't yet been discovered, although of course there's no guarantee of this.

4.4.1 Mobile

The mobile is the simplest generic type. It has only one high level variable that dictates where the mobile is on a map of locations; the variable is generally called *at*. The mobile has one simple operator, the *move* operator. It assigns *at* with a new value.

To support this generic type the template would have to support different generic types in the same template, in this case *mobile* and *location*. The location type is defined only by its association with the mobile. A simple object reference variable is needed to represent *at*, and the move operator needs a way of representing an assignment.

4.4.2 Carrier

The carrier is an extension of the mobile generic type. A carrier has an additional high-level variable that represents its *contents*. It could carry only one object, or it could carry multiple objects. The carrier has a *load* operator and an *unload* operator that load and unload from *contents* to the current location that the carrier is *at*. The carrier defines the type *portable* by association. The portable type can only be in one carrier at once, so it has the notion of a mutually exclusive loading condition. This condition can be represented in many ways; it may not even be explicit.

4.5 Introducing the Generic Type Language

The template language that was designed to describe generic types is called Generic Type Language or simply GTL. GTL went through a rapid prototyping development cycle and its final form bears little or no relation to the original designs. The final version is able to describe all the existing generic types and is similar (although not identical) to OODDL's syntax.

Although the initial version turned out to be insufficient to represent all of the generic types, it is interesting in that it highlights some of the problems encountered when trying to capture the generic types, and it provides motivation for the features of the final version of GTL.

4.6 'Flat' GTL

The initial version of GTL was very simple, all the language did was to list variables, operator expressions and types that needed to be matched for the template to fit. Then it placed a few restrictions on these elements to ensure correct interoperability between them. There was no hierarchy or structure in the initial version of GTL, hence the use of the term 'flat'.

A simple example of the initial direction of GTL is shown below:

Elements	**Conditions**	**Element Names**
[1] Variable	Object reference to type [2]	*at*
[2] Type		*location*
[3] Expression	[1] := Value of type [2]	*move*

Prototype template for representing a mobile generic type

One of the design requirements of GTL was easy matching with OODDL, as the above snippet indicates. The matching algorithm here simply did a brute force search trying every possible combination of variable, type and expression for each type in the domain. Then it would check the conditions in the second column held, if they did then the values where recorded and the type was marked as a generic type.

The problem with this language is that is isn't quite powerful enough to achieve the more complex generic type descriptions elegantly. When it came to describing the *carrier* generic type, the conditions began to become very complex.

The first problem was that the definition of a carrier in OODDL varies depending on whether it is a *multi-carrier* or a *single-carrier*. A multi-carrier in OODDL would have an *object-bag* variable for its contents, so it could have more than one portable in its contents at once. A single carrier would have a *maybe object reference* as its contents as it is either referring to exactly one portable or none. This meant that the template had to match to both of these different encodings, and disjunctions had to be introduced into the conditions. This made the language more complex but still manageable.

The next problem was that due to its simplicity the language lacked any inherit structure, so simple relations between the various elements in the template had to be expressed explicitly. This became increasingly cumbersome as the need grew to group these elements into higher-level semantic units. For example, individual expressions were matched in the previous example; in that case, the *move* expression was identified. When a high-level generic type action consisted of more than one expression, additional syntax had to be added to the language to ensure the expressions matched were all from the same domain action.

An example of this is the carrier, which needed to find several expressions in the same domain action for it to qualify as one of its high-level operators. For instance, *load* needed an expression to check the location of the portable, one to add it into the carrier's contents and one to disassociate the portable with its location so it wouldn't be loaded again.

Finally, there was the problem related to the values in expressions. The expressions were identified by looking for a candidate expression in the domain that assigns a given variable with a value of a given type. This was insufficient for the carrier because besides the expressions being of the correct form, they also had to be referring to the same object. The object that is checked for location must not only be the same type as the object being added to the contents, it must be the same object. This is represented in OODDL and STRIPS by an action parameter, a value that is matched at the beginning of the action and then used in all the expressions in the action. Now additional syntax was needed to represent parameters in groups of expressions. This added further syntax and the language was by this point becoming very difficult to comprehend!

4.7 Object-Orientated GTL

The three problems with the flat version of GTL were: the lack of a good way of handling disjunctions, lack of easy means to group elements into higher-level semantic units and lack of a way to refer to values. The decision was made to move towards a more structured object-orientated template representation; in doing so, all three of these issues were resolved.

4.7.1 Structuring the Templates

In order to describe a generic type fully, it is necessary to group elements such as expressions into actions, and then actions and variables into types. The problem with the flat version of GTL is that extra definitions had to be given in order to do this. If the template contained more inherit structure, then these relationships would not have to be specified explicitly.

It was decided that the best structure would be one close to OODDL itself; this would make it very easy to match to OODDL, which was a high priority. The final version of GTL consists of object-orientated templates that matched against the domain description. The object-orientation allows variables and actions to be bound to a class, simplifying the conditions that need to be asserted while pattern matching.

High-level generic type operators look very much like OODDL actions, with parameters, preconditions and effects. Similarly to OODDL they are also embedded within the class declaration and they can refer to the *this* object and its member variables.

As before, GTL actions don't need to map directly onto a domain action, they can simply map onto some of the expressions in a domain action. Similarly, the use of parameters in the template need not be directly matched to action parameters in the domain description; a template parameter is simple a label for *any* value used in a domain action. Therefore, a parameter from a template action could refer to a member variable in the actual domain, or any other value that is used in the action being compared against the template. This allows templates to refer to values easily.

4.7.2 Addressing Disjunctions

The next problem that needed to be addressed was the issue of disjunctions in the templates. Disjunctions make the template harder to write because a disjunction in one place often means that other disjunctions have to be made in other places.

This is seen in the carrier template, where the *contents* variable can be either an object bag or a maybe object reference. This in turn changes what the load

and unload operators look like, because they will either assign the contents variable if it's an maybe object reference, or add an element if it's an object bag.

One way of eliminating the need for disjunctions would be to create two different templates, one for the multi-carrier and one for the single carrier, however this is not ideal as the template definitions would then greatly increase in size.

A better way of addressing the disjunctions is to abstract them out. All of the disjunctions that occurred were related to the type of variable, for example whether it was an object bag or an object reference. So instead of having disjunctions, two new equivalence variable types were introduced. These are the *single-object reference* and the *multi-object reference.*

These two variable kinds view object relationships in the most abstract manner possible, a variable either can refer to one object, or more than one object. Specifically, a single-object reference maps onto a variable that can refer to zero or one objects. It can be matched to OODDL object references, maybe object references, enumerated type variables and booleans. A multi-object reference refers to zero or more objects. It can map to anything a single-object reference can, plus it can also map to the OODDL object bag type.

Besides these two object reference variable types, there is also a boolean variable type. The boolean can be asserted as true or false and assigned like a normal variable. In the target domain however, it maps on to a boolean *expression* rather than a variable. By asserting its value to be true or false in the template action description, it is possible to assert state restrictions on the objects undergoing actions, without caring how these states are actually represented.

For instance, a boolean may be used to ensure a portable can only be loaded into one carrier at a time. It would map to an expression that would be true when the portable is unloaded and false when it is loaded. The exact expression would of course depend on the domain encoding; for example at!=NULL (the location is not NULL) or in-carrier==NULL (there is no carrier assigned).

A template assignment of true would map to an effect that makes this condition true, conversely a template assignment of false would map to an expression that makes this condition false. The boolean is a very flexible variable due to its high abstraction; it greatly increases the number of different generic type encodings that can be recognised by a template.

GTL is very similar to OODDL and this means that a domain engineer wishing to create a new generic type template doesn't need to learn an entirely new language. However, GTL and OODDL are not identical. In a sense, a GTL generic type template could be seen as a more abstract or "looser" version of an OODDL type definition. Where OODDL states the types of its variables, GTL states simply what they can be.

4.8 Transportation Template in GTL

The previous transportation generic types could be captured in GTL template using three generic type definitions as so:

```
type Carrier

    // at is a single-value variable
    // contents is a multi-value
    // variable
    Location        at
    Portable        +contents

    // Action definitions
    // "P" denotes a precondition
    // "E" denotes an effect
    Move(Location to)
        P: at!=null
        E: at:=to
    end

    // Ensure the carrier is on a map
    // and check the package is at the
    // same location before loading
    Load(Portable p)
        P: at!=null
        P: p.at==at
        P: p.loaded==false
        E: contents+=p
        E: p.loaded:=true
    end
```

```
        Unload(Portable p)
            P: at!=null
            P: p in contents
            E: p.at:=at
            E: p.loaded:=false
            E: contents -= p
        end
    end

    type Portable
        Location  at
        Boolean   loaded
    end

    type Location
    end
```

GTL template definition of the transportation generic types

In the above template there are three generic types defined. The carrier type is the main type in this template and it defines the functionality of the other two types by association. This template completely captures the behaviour of the transportation generic types.

The three actions, *move*, *load* and *unload*, are self-explanatory. An important point about these actions is that they need not map directly to domain actions, but, as previously stated, can be subsets of the actions instead. Therefore, an action may have other effects besides those of the generic type operator.

A domain-engineering tool that was editing a domain containing the carrier type could hide the statements that form the generic type actions and replace them with their respective high-level collective names, simplifying the editing. For example, an action in OODDL that was identified as performing, amongst other things, a load operation; could have that section of it hidden and replaced with the high-level equivalent, such as "load(thePackage)".

4.9 Generic Type Services

It has been previously stated that generic types can be beneficial to both planners and domain engineering tools. However, each type of tool may require different services from the generic type. For example, planners would be interested in search heuristics whereas domain-engineering tools would not.

Each GTL generic type description also describes exactly what services the generic type implements, allowing tools to use only the generic types that will bring it benefit. The ranges of services available from the generic types are far reaching, but some of the services so far proposed are outlined in the following sections.

4.9.1 Editing Tags for Domain Engineering

Editing tags supply information describing a method for editing instances of the generic type. For example, a graph structure tag would denote object relationships best edited with a graph editor. Such types include location networks in transportation domains, or complex value transition networks.

Linear parameters could also be identified, such as task length for the multi-processor-scheduling generic type [Fox and Long, 2001] or fuel capacity for fuelled mobiles. These could then be manipulated with a range slider GUI widget or another appropriate interface.

Construction domain instances could be better edited using a tailored graphical user interface. This would be better able to express the idea of an object being composed of other objects, as seen in the MacGyver domain [Clark, 2000].

The domain-engineering tool would have to support a number of different configurable editing methods that could be selected by the editing tag.

4.9.2 State Model

Object states allow another interpretation of the domain and actions to be overlaid. This would allow the various states that an object can partake in to be categorised and labelled. This could then form a basis for conveying further

features to the client, such as state invariants or qualifying an object's state [Gerevini and Schubert, 1998].

Each state definition would consist of a boolean expression involving the generic type's variables and a label. Whenever the expression is true of an instance of the generic type, it is labelled as being in that state. The states would be disjunctive.

States would also allow a visual representation for a domain-engineering tool. When creating a new object, a user could specify the state the object is in and then the necessary invariants could be automatically met.

4.9.3 Planner Assistance

This service would centre on providing information to assist a planner during search, such as heuristics or sub-solver specification.

Heuristics would present guidelines for manipulation of the generic types; they could be used to prune the search space and improve performance, by recommending situations where a specific action should never be performed, or conversely where one should always be performed.

Sub-solver recommendations would mean describing sub-problems in the plan in terms of generic type instances and then suggesting an algorithm for solving them. For example, the STAN planner is able to identify route-planning sub-problems involving mobile generic types; these are then handled by a separate path-finding algorithm [Fox and Long, 2000b].

Implementing the sub-solver service would require specifications of sub-solvers to be published and then implemented by planners. The planners would then be able to invoke the required sub-solver on the domain sub-problem to solve it. An ultimate version of the sub-solver could even define an algorithm in a simple scripting language or in terms of collections of closely interrelated heuristics.

4.9.4 Visualisation

Visualisation is a way of using graphics to interpret a domain description. Graphical descriptions could be suggested for the generic types and the generic

type parameters linked to how they are displayed. 3D object descriptions could even be parameterised in this way, allowing domain states to be viewed or edited in a more "physical" sense.

Plan execution could be visualised using a simple extension of this idea.

4.10 Applying GTL

Now that a formal declarative language is available for describing generic types, the problem becomes one of applying the template language to a source domain. Throughout the discussion of GTL, thought has been given about the application to OODDL.

It section 4.6, it was mentioned how the flat version of GTL could be matched against an OODDL source domain by a brute force pattern match. This involved a generate-and-test approach where every single possible assignment of values was made and then tested against the conditions attached to the template.

The same method can be applied to the final version of GTL, which is simply the flat GTL with additional structure. This method has been implemented in Draughtsman allowing generic types from GTL templates to be identified.

This simple method works well, however it is notoriously inefficient. A more efficient method would be one similar to the approach taken by TIM. TIM constructs finite state machines describing behaviours of types in the domain. As was discussed in section 4.3.1, generic types can also be viewed as FSMs. By applying GTL templates to the domain FSMs it should be possible to quickly search the domain for matching patterns without having to index though every possible potential match. This would also allow GTL to be potentially applied to STRIPS or other domain languages for which a FSM representation was available.

4.11 Summary

Generic types are types that are described by their semantics, not their implementation. They often represent common concepts in the mind of the domain engineer, such as mobile objects that move on maps of locations, or building blocks that build larger structures.

If the domain-engineering tool is able to assist the domain engineer in creating these common types then it will be providing a good starting point for new domains. Furthermore, if the tool is able to provide tailored sub-editors for some of these types, the editing session will become more focussed on the underlying semantics of the domain, rather than the syntax.

GTL is a template language for describing generic types. A GTL template can be used to create a new type in the domain, or it can be used to spot existing types in the domain, possibly giving the user a new viewpoint. Once a generic type is identified, the user can manipulate the domain using more relevant terminology, supplied by the template. GTL templates can potentially supply a wide range of services to the template client; examples include domain engineering hints, recommended heuristics and sub solvers for planning and state models.

5 Evaluation

The outcomes of this study have been two languages for domain modelling and a domain-engineering tool. The first language, the Object Orientated Domain Description Language (OODDL), is a language for describing planning domains and their associated problems. The domain-engineering prototype tool *Draughtsman* provides a CLI driven menu system for editing domains and providing specialised sub-editors for any higher-level generic types found in the domain. Draughtsman is also able to convert OODDL domains into PDDL STRIPS.

The second language, the Generic Type Language (GTL) is a language to describe generic types to both planners and domain construction tools alike. GTL provides a means for domain experts to express services of the generic types, such as heuristics, domain editing tags or invariants. Using an extendable "plug-in" approach, both editors and planners alike can be extended through GTL to recognise and manipulate new generic types. This allows domain-editing tools to become more context sensitive, for example using real-world domain terms instead of generalised terms, and it allows planners to exploit any performance benefits available from the generic types. In a sense, GTL allows domain specific knowledge to be passed from the domain engineer to the planner in a domain independent way.

5.1 Evaluation Aims

Of the three products of this study: OODDL, GTL and Draughtsman, it has been decided to evaluate only OODDL.

Draughtsman assists creation of OODDL domains by providing a text menu driven interface that uses wizards to create action expressions that are always well typed. However, Draughtsman isn't user friendly enough in its current prototype to be usable for domain construction by inexperienced users, so OODDL will have to be evaluated without it.

Draughtsman's OODDL to STRIPS algorithms could be evaluated, however work on this conversion module has so far been focused on simply creating a sound and complete STRIPS domain, very little work has been done on optimising the output. This means that although the STRIPS domains produced are sound and complete, comparing the generated domains to hand coded STRIPS at this stage would prove very little.

Evaluating GTL is difficult on two fronts: firstly, the language currently has not been implemented beyond a specification and parser. GTL specifications have been created for existing generic types (e.g. construction, mobile, carrier, mps), however identifying these in a domain will require new domain analysis algorithms that either build on the work of TIM or are entirely new.

Currently Draughtsman is able to read GTL specifications and instantiate some templates by use of a brute force pattern match, however this approach has little future when larger domains and greater numbers of templates are used.

However, OODDL can be evaluated by direct comparison to STRIPS by use of a test on planning literate undergraduates. These tests will attempt to address how easily a candidate can understand an existing domain excerpt in both STRIPS and OODDL, and how easily they can create one.

The features that OODDL presents over STRIPS are:

- Object-orientated
- Easy maintenance of single valued-ness
- Explicit inheritance with overriding
- Enumerated values
- Typing

The aims of evaluating OODDL will be two fold: firstly to try to attain if OODDL's object-orientated approach is easier for candidates to work with than STRIPS flat approach. Some features of OODDL will be hard to evaluate on simple tests, such as the possible benefits of object-orientated domains mapping to real world domains more easily. The effectiveness of inheritance in modelling

will also be hard to evaluate, as this will require large domains with many classes, this is beyond the scope of a simple test.

The second objective when evaluating OODDL is to rate the effectiveness of OODDL's variables against STRIPS predicates. OODDL's variables automatically enforce single-valued-ness constraints, meaning that the scope for errors due to missing negative effects is greatly reduced.

5.2 Designing the Tests

The test design for OODDL consisted of two parts, an understanding part and a construction part. It is a written test and does not require a computer. The same questions were asked in both STRIPS and OODDL to see which language the candidates were more successful with.

5.2.1 Question 1: The Understanding Test

Initially the understanding test presented the candidate with a logistics domain with all names mangled, similar to the mystery domain. The candidate's task was to examine the underlying semantics of the domain and through understanding the semantics, to suggest possible labels for actions and predicates in the domain.

It was hoped that this would test how well OODDL's syntax displayed the underlying semantics of the domain in comparison to how STRIPS was able to; the hypothesis being that the structure exhibited by OODDL's object-orientated approach would yield more clues as to the domain's meaning.

It was decided however that this test was too complex and difficult to grade, it was very likely that candidates would either completely understand it or not even know where to start.

The next approach for the understanding test was to present a flawed blocks world operator and ask the candidates to identify the error. This would require that the candidates understand what the operator was trying to do at the semantic level and be able to relate it to the syntax for either the flawed STRIPS or OODDL action on the test.

It was decided to use a simple STRIPS notation rather than PDDL STRIPS because PDDL would most likely prove too confusing for the inexperienced. Using PDDL would have made the test results more applicable to the real world where PDDL STRIPS is the standard, however for the purposes of this test a comparison with STRIPS was sufficient.

The simple STRIPS notation will essentially follow the PDDL STRIPS format but omit the LISP syntax. Instead actions will have clearly marked precondition and effects lists. The same notation will be adopted for OODDL to minimise the number of unnecessary differences between the tests.

STRIPS Domain Description

Predicates:
- on(a,b)
- clear(a)
- on-table(a)
- arm-empty()
- is-held(b)

Actions:

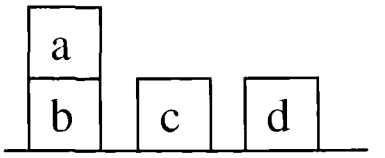
pickUpFromTable(block)
- preconditions:
  - arm-empty()
  - clear(block)
  - on-table(block)
- effects:
  - is-held(block)
  - ¬on-table(block)
  - ¬arm-empty()

end

putOnTower(block,to)
- preconditions:
  - is-held(block)
  - clear(to)
- effects:
  - on(block,to)
  - *arm-empty()*
  - ¬is-held(block)
  - ¬clear(to)

end

STRIPS Problem Description



Objects:
- a,b,c,d

Initial State:
- on(a,b)
- clear(a)
- clear(c)
- clear(d)
- on-table(d)
- on-table(c)
- on-table(b)
- arm-empty()

Goal State:
- on(c,d)
- on(d,a)
- on(a,b)

Completed STRIPS version of question 1

The flaw in the operator would be an omitted effect for marking the arm as available after it stacks a block (marked in *italics* in this completed question). The candidates would have to understand the preconditions and effects of the action and explain why it is not possible for the action to be executed more than once.

In the OODDL version of this question, the arm's empty status is recorded in a boolean in the arm class. Both the OODDL versions and STRIPS versions of these questions are expected to be of the same difficulty.

5.2.2 Question 2: Negative Effects Test

The purpose of the rest of the test is to identify whether OODDL was able to address two particular theorised shortcomings of STRIPS. The first shortcoming was to do with missing negative effects breaking the domain model. This occurs in STRIPS, when a particular relation between two objects is supposed to be single-valued and an operator adds a new predicate to change the relation and forgets to remove the previous one. This was discussed in detail in section 3.4.2.

OODDL addresses this by its use of variables that automatically maintain their single-valued-ness when assigned. The test should highlight this difference and attempt to show OODDL's benefits in that area.

The question designed to address this issue was a domain that involved moving an object from being in one bucket to being in another.

STRIPS Domain Description

Predicates:
in(thing,bucket)
bucket(ob)
thing(ob)

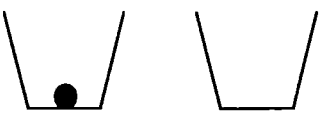
Actions:
Fill out the contents of this action!
move(thing,from,to)

preconditions:

effects:

end

STRIPS Problem Description



bucket1 bucket2

Objects:
bucket1, bucket2, thing1

Initial State:
bucket(bucket1)
bucket(bucket2)
thing(thing1)
in(thing1,bucket1)

Goal:
in(thing1,bucket2)

STRIPS version of question 2

The candidate would have to complete the preconditions and effects of the existing operator. This question would attempt to catch candidates who are not

thinking about negative effects and who fail to remove the existing **in** predicate instance when instantiating a new one.

There will be two OODDL variants of this question. The first variant will track the object's location by use of an object reference variable. This variable can only have one value at a time and so all the candidate need do is assign the variable with the new bucket.

```
type Thing
        Bucket          *inside

        move-into-bucket(Bucket b)
            preconditions:
            effects:
                inside:=b
        end
end

type Bucket
end
```

Maybe object reference version of OODDL question 2

The text in italics would be the part the candidates would need to write

The second variant of this OODDL question would represent the object's location by each bucket having a contents *object-bag* variable. The move operator would have to remove the object from the contents of one bucket, and add it to the contents of the other.

```
type Bucket
        Thing          contents{}

        move-into-bucket(Bucket b,Thing t)
            preconditions:
            effects:
                b.contents-=t
                contents+=t
        end
end

type Thing
end
```

Object bag version of OODDL question 2

The text in italics would be the parts the candidates would need to write

This variant of the question was designed to see if the candidates were as likely to omit negative effects (in this case the effect of removing the object from the previous bucket) as readily in OODDL as it is predicted they will in STRIPS. It is hoped that by explicitly representing the contents relation with an object bag, the candidates will realise that moving will require the removal of the object from one contents, and the addition of it to the other. This should result in candidates being more aware of negative effects in ODDDL than they would in STRIPS.

5.2.3 Question 3: Enumerated Types Test

The second area where it was theorised STRIPS has a weakness is the use of enumerated types. In STRIPS, domain constants often have to be implemented as objects with unique predicates to identify them, as discussed in section 3.8.1. This can cause confusion for inexperienced users because the distinction between the unique predicates and the unique objects that possess this unique predicates becomes blurred.

For example, in a lift domain it is required that a lift be recorded as either being at floor one, two or three. To represent this in STRIPS a unique predicate

could be assigned to each floor to allow operators to differentiate between them in their preconditions: e.g. "floor_one(x)", "floor_two(y)" and "floor_three(z)".

OODDL, by comparison, directly supports enumerated types, allowing the user to simply assign a pre-declared value of either "one", "two" or "three" to the object's "floor" variable.

To highlight this difference the tests require the candidate to write two operators in either OODDL or STRIPS. The operators have to move the lift from floor two to floor three and from floor one to floor two. This requires the candidate to establish the identity of the floors passed to the action, and then assert the correct add and delete effects to move the lift. This gives the candidates the opportunity to forget negative effects and to fail to establish preconditions for differentiating between the floors.

STRIPS Domain Description

Predicates:

at-floor(lift,floor)
lift(ob)
floor-1(ob)
floor-2(ob)
floor-3(ob)

Actions:

moveFrom1To2(lift,wasFloor,newFloor)

preconditions:

at-floor(lift,wasFloor)
floor-1(wasFloor)
floor-2(newFloor)

effects:

at-floor(lift,newFloor)
¬at-floor(lift,wasFloor)

end

moveFrom2To3(lift,wasFloor,newFloor)

preconditions:

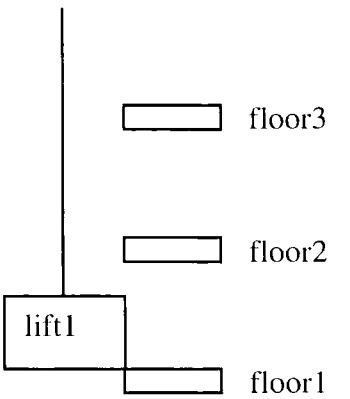
at-floor(lift,wasFloor)
floor-2(wasFloor)
floor-3(newFloor)

effects:

at-floor(lift,newFloor)
¬at-floor(lift,wasFloor)

end

STRIPS Problem Description



Objects:

lift1, floor1, floor2, floor3

Initial State:

floor-1(floor1)
floor-2(floor2)
floor-3(floor3)
lift(lift1)
at-floor(lift1,floor1)

Goal:

at-floor(lift1,floor3)

Completed STRIPS version of question 3

In OODDL, the use of a "FloorNumber" enumerated type simplifies the domain description:

```
enum FloorNumber = { one, two, three }

type Lift
        FloorNumber          floor

        moveFromOneToTwo()
            preconditions:
                floor==one
            effects:
                floor:=two
        end

        moveFromTwoToThree()
            preconditions:
                floor==two
            effects:
                floor:=three
        end
end
```

Completed OODDL version of question 3

It is hoped that candidates constructing the STRIPS versions will make errors both due to missing negative effects and confusion between the unique domain objects representing the floors in the domain and the unique floor identifying predicates.

5.3 The Tests

The tests consisted of four A4 typed sheets per language. Each test sheet came with a concise manual for the language for the candidates to refer to if they couldn't understand the domain excerpts. The candidates were allowed as much time as they needed, but they were not allowed to discuss the tests with one another.

The candidates chosen were second year undergraduates form the Artificial Intelligence degree at the University of Durham. A total of 25 candidates were recruited which resulted in 11 results for STRIPS and 7 for each of the OODDL papers. The candidates were only allowed to take one paper.

The tests were conducted in two phases; the initial phase was a pilot test that was used to identify errors or potential improvements in the tests themselves. This identified a couple of typographical errors that were subsequently fixed, but also showed that none of the candidates doing the STRIPS tests missed out negative effects.

It was thought that this might have been because the operator template on the test had a field for both negative effects and positive effects; causing the candidates to ask themselves what negative effects there were. In the next edition of the tests STRIPS operator templates had only a single effects field, as would be the case if the candidates were working with PDDL where negative effects are created by prefixing effects with a *not* operator. The OODDL tests already had a combined effects field and so this evened the field between the tests.

The tests can be found in sections 7.3 and 7.4 in the appendix.

5.4 Expected Results

One of the main errors expected from STRIPS candidates is the failure to maintain single-valued-ness during construction of actions. This shouldn't pose a problem on the OODDL version of the tests.

It is anticipated that candidates on the STRIPS test will break single-valued-ness invariants on questions 2 and 3 by omitting negative effects.

The first variant of question 2 on OODDL paper 1 allows scope for candidates to omit negative effects also, however it is hoped that fewer candidates will make this error on the OODDL question. The OODDL question uses object bags to represent the bucket contents and so the candidate could add the object to one bucket and forget to remove it from the previous one. However, is hoped the explicit declaration of the bucket contents as an object bag will mean the

candidates think about the move operation in two stages and subsequently both add the object to the new bucket and remove it from the old one.

In the second variant of question 2 on the OODDL paper, an object reference variable is used to represent the bucket containing the object. It is anticipated that most, if not all candidates will get this question trivially correct.

Question 3 offers further opportunities for STRIPS candidates to omit negative effects and thus break single-valued-ness invariants. Furthermore, this question may prove quite difficult to candidates who are not familiar with the technique of using transitive assertions to assert facts. For instance, stating:

```
at-floor(lift,floor)
floor-1(floor)
```

To encode the fact that "lift" is at floor-1.

The OODDL version of this question makes use of OODDL's enumerated types to provide a trivial encoding. It is anticipated that a significantly higher percentage of candidates will correctly answer the OODDL version as opposed to the STRIPS version.

Question 1 should be of equal difficulty in both STRIPS and OODDL, this question will highlight the "instinctive" understanding candidates possess of predicate based language and object-orientated languages respectively. It is expected than OODDL will be generally more easily understood.

5.5 Results

The following table shows the length taken for each question by the candidate, along with the result of the question.

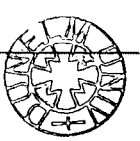
Paper ID	Paper	Q1 time	Q2 time	Q3 time	Q1 correct	Q2 correct	Q3 correct
1	OODDL(1)	3	12		✓	✓	✓
2	STRIPS	8	2	7	×	✓	×
3	STRIPS	4	2	4	✓	✓	✓
4	STRIPS	8	2	3	✓	✓	×
5	OODDL(1)	10	5		×	×	×
6	OODDL(2)	9	4	3	✓	×	✓
7	OODDL(2)	9	5	2	×	✓	✓
8	OODDL(1)				×	×	✓
9	OODDL(1)	9	2	5	×	×	×
10	STRIPS	10	6	4	×	×	×
11	OODDL(2)	10	2	3	✓	×	×
12	STRIPS	7	3	5	✓	✓	×
13	STRIPS	5	2	5	✓	✓	×
14	STRIPS	6	5	3	✓	✓	×
15	STRIPS	6	6	5	✓	✓	×
16	STRIPS				×	×	×
17	STRIPS	6	5	3	×	✓	✓
18	OODDL(2)				✓	✓	✓
19	OODDL(2)	9	1	5	✓	✓	✓
20	OODDL(2)	3	3	1	✓	✓	✓
21	OODDL(1)	8	3	5	×	×	✓
22	OODDL(1)	8	7	1	✓	✓	✓
23	OODDL(2)	5	5	3	✓	✓	✓
24	OODDL(1)	5	5	3	✓	✓	✓
25	STRIPS	7	2	4	✓	✓	✓

Results of tests, blank timings represent missing times from the candidates

The following table holds additional information on the test results in the form of comments:

Paper ID	Paper	Comments
1	OODDL(1)	All correct.
2	STRIPS	In Q3, candidate gives correct effects however he got his predicates and objects confused in the preconditions.
3	STRIPS	Fully correct, candidate even put in typing preconditions in the actions, showing a good understanding of the available predicates.
4	STRIPS	In Q2 the candidate uses negative preconditions. In Q3, candidate gives correct effects however he got his predicates and objects confused in the preconditions.
5	OODDL(1)	In Q3, candidate gives correct effects however he got his predicates and objects confused in the preconditions.
6	OODDL(2)	In Q2, candidate has the correct action structure, however he makes direct reference to problem objects instead of using the action parameters.
7	OODDL(2)	Candidate misunderstands Q1.2 and proposes a new action instead of correcting the existing one.
8	OODDL(1)	In Q2, candidate produces correct structure but systematically swaps the left and right side of every expression. In Q3, candidate produces correct preconditions and then refers to a problem object directly in the effects, instead of using the action parameters.

9	OODDL(1)	This candidate didn't really understand the test at all. Some questions left blank. Made use of natural numbers and invented variable names in Q3.
10	STRIPS	Candidate fails to comprehend any of the domain excerpts.
11	OODDL(2)	On Q2 and Q3, this candidate has correct structure, but fails to access member variables correctly. On Q3 integers are used.
12	STRIPS	On Q3, candidate breaks the single-value-ness of the lift floor. He also shows confusion between objects and predicates.
13	STRIPS	Candidate adds typing preconditions on Q2. In Q3 he fails to maintain single-value-ness and he makes use of equality operators for a variable that to him is clearly single-valued.
14	STRIPS	In Q3, candidate fails to maintain single-value-ness and also confuses objects and predicates completely.
15	STRIPS	In Q3, candidate fails to maintain single-value-ness and also confuses objects and predicates completely.
16	STRIPS	In Q2 and Q3 the candidate fails to maintain single-value-ness. In Q3 he confuses objects and predicates completely.
17	STRIPS	All correct + candidate makes use of typing on Q3.
18	OODDL(2)	On Q2 the candidate checks value is not already set in preconditions.
19	OODDL(2)	All correct + candidate checks value is not already set in preconditions.
20	OODDL(2)	All correct.



21	OODDL(1)	In Q2, the candidate had the right idea but made use of the wrong operator. He attempted to assign to the contents set instead of using add/remove operations on it.
22	OODDL(1)	All correct.
23	OODDL(2)	All correct.
24	OODDL(1)	All correct.
25	STRIPS	All correct + candidate makes use of typing in Q2.

5.6 Discussion of Results

The following table summaries the number of correct answers, along with the time taken, for each of the questions on the tests.

Paper and Question	% Correct	Average Time for Correct Answers in Minutes
OODDL Q1	9/14 (64%)	6.2
STRIPS Q1	7/11 (64%)	6.2
OODDL(1) Q2	3/7 (43%)	8[10]
OODDL(2) Q2	5/7 (71%)	3.5
STRIPS Q2	9/11 (82%)	3.9
OODDL Q3	11/14 (79%)	2.9
STRIPS Q3	3/11 (27%)	3.7

5.6.1 Question 1

The difference between results on question 1 is startling small. Both the OODDL and STRIPS questions took, on average, the same amount of time to solve and had the same percentage of correct answers.

This question was designed to evaluate how easily a domain description in both OODDL and STRIPS could be understood by the candidate. If they

[10] This high number is suspected to be due, at least in part, to candidate 1 misreporting the time. If candidate 1 is omitted from this mean then the result is 6 minutes.

understood it to a good enough degree, they should have been able to identify the error in the operator. As candidates answered the question in the same amount of time and with the same accuracy regardless of the domain language, it would indicate that STRIPS is no harder to understand than OODDL.

Although it was expected that OODDL would be demonstrably easier to understand than STRIPS, the domain was necessarily very trivial. It is now hypothesised that the benefits of object-orientated domain descriptions may only become relevant on larger domains with many more classes and actions. This is perhaps analogous to small programs being equally well understood in C and C++, but larger programs being easier to comprehend in C++. A larger domain could not have been used in this test due to the time restraints placed on the tests.

Overall, this question could have been improved by using a larger domain example where OODDL's inherit structuring of the domain would have been better used. Although that could not have been done here, perhaps a test purely addressing the understandability could use larger and more in-depth domains.

5.6.2 Question 2

Question 2 was designed to evaluate OODDL's facilities for preventing the omission of negative effects. It was expected that answers for the STRIPS question would frequently omit negative effects, in this case, adding the fact that the object was in the destination bucket but failing to assert the fact it was no longer in the source bucket.

Unfortunately, a firm confirmation of this hypothesis cannot be drawn from these results. There was only one instance of missing negative effects: from STRIPS candidate 16. Question 2 on OODDL paper 1 was designed to allow candidates to omit negative effects on an OODDL question. Although no candidates omitted negative effects on the OODDL version, the statistical variation between a single candidate making an error on STRIPS and no candidates making an error on OODDL is not significant enough to draw a conclusion.

It is still felt that missing negative effects are a common fault in STRIPS domains, however it is likely that this particular question was simply unable to capture this fact. For a domain engineer to omit negative effects in a domain description it would probably require a larger domain description with more complicated actions. In this question, the candidate only had to maintain one predicate, the object's location. If they had been required to maintain multiple predicates, it is more likely that some negative effects would be omitted.

This question was carefully designed to be simple and to the point because it was believed that the candidates would not be experienced enough to work with more complex domains. Although some candidates were unable to complete the question, most candidates found it very trivial. It is now believed that the candidates were able to draw on other areas of experience, such as undergraduate logic courses and Java courses, more than was expected. It is now anticipated that the candidates would be able to complete a more complex question in future tests, one that would provide greater scope for errors, but time and resource constraints prevented further testing.

Although question 2 did not yield the expected result with regard to negative effects, it did yield other useful information that can be analysed. Primarily the results indicate that a higher percentage of candidates were able to correctly answer the question in STRIPS than in OODDL. Indeed only two candidates made errors in the STRIPS paper on this question. One as mentioned was due to missing negative effects, the other indicated confusion between predicates and objects that resulted in badly formed expressions. As discussed later in the analysis of question 3, this confusion between predicates and objects is shown to be a common error in the STRIPS tests.

In OODDL paper 2, or OODDL(2), all the candidates were required to do was to simply assign a single variable that was already defined with the only parameter passed into the action. Two candidates made errors with this: the first directly accessed the object instance by name from the problem definition. Clearly the action should not refer directly to object instances in the domain, but should instead work with either the implicit "this" object (as in this case), or with a parameter of the action. As will be discussed in the analysis of question 3,

the confusion between problem domain object instances, and the action parameters that are actually available to use, proved to be a very common error in OODDL.

The second candidate's error on the OODDL(2) test was much in the same vain: he made the error of prefixing the variable with the name of the class, as one would do to access a static class variable in C++ or Java. Static class variables are not a feature of OODDL.

The difference between the accuracy percentages of STRIPS and OODDL(2) appear large but, due to the small sample, they only evaluate to approximately a single candidate. The difference in the timings is likewise too small to be considered significant.

The OODDL(1) variant of question 2 used object bags to represent the contents of the bucket. The structure of the action that the candidates were required to produced was identical to the STRIPS question. The candidate needed to add the object to one bucket's contents with OODDL's add operator ("+=") and remove it from the other using the remove operator ("-=").

Of the seven candidates who attempted this question, four answered incorrectly. Two of the candidates attempted to assign the bucket contents to be equal to the object. This is simply incorrect use of the object bag, or indeed any vector data structure where an update is required. Of the remaining erroneous candidates, one was simply unable to answer any questions correctly and indicated a complete lack of understanding of the test; the remaining candidate gave the correct structure of action, but swapped the left and right sides of the add and remove expressions around, creating a badly formed expression.

The most likely reason for incorrect use of the object bag variable in the OODDL question is that none of the other questions on the paper made use of one. This meant that the candidate had no "working example" to base their use of object bag add and remove expressions on, meaning they weren't sure how to add or remove objects from the bag. It had been originally believed that the candidates' experience with set theory, Java or C++ would make the object bag seem obvious, however this did not seem to be the case. Although they had been

given a concise OODDL reference sheet showing all operators and variable types, hardly any candidates used it at all.

With a few more examples or a little experience, the candidates would almost certainly recognise how to add and remove objects from an object bag in OODDL. STRIPS, in essence, has two effects: add fact or remove fact, different data structures yield no new syntax and subsequently no opportunity for ignorance.

This does not mean it can be stated that STRIPS is easier because it has less syntax, any more than it could be stated that binary notation is easier for humans than ASCII, just because it has a smaller set of symbols.

The final point about question 2 on the OODDL(1) paper is the significantly larger amount of time spent on the question. It is incredibly likely that this was simply due to the candidates who did answer the question, having to refer to the language reference sheet to answer the question. It is hypothesised that if the candidates were to be asked another question using object bags the time would be significantly shorter.

Furthermore, candidate 1 took 12 minutes to answer this question, which severely upset the mean time. It is expected that this candidate misreported the time and combined it with the time for question 3, which he doesn't supply.

OODDL doesn't have a large amount of syntax and a user could easily learn the syntax and work without a reference sheet, so it would be unfair to extrapolate this example into a statement about OODDL's ease of use.

Overall, question 2 could have been improved by asking candidates to do more things in the action. This would have widened the scope for the expected omission of negative effects. Candidates may have benefited from a working example of object-bag addition and deletion in the OODDL(1) variant of the question, this would have precluded simple syntax errors arising from the lack of familiarity with the object bag.

5.6.3 Question 3

The third and final question of the tests was designed to evaluate OODDL's enumerated type features. In this test, the OODDL candidates gave significantly more correct answers than the STRIPS candidates did; the OODDL candidates also took, on average, less time.

This question described a simple lift domain. The lift could be at one of several floors at any time. In STRIPS the floors were described by objects, each of which has a different typing predicate to differentiate them. In OODDL, the floors were described by an enumerated type. The candidates were asked to construct two operators, one to move the lift from floor 1 to floor 2, and one to move it from floor 2 to floor 3.

In OODDL, this required the candidates to establish in the preconditions that the floor variable was equal to the enumerated value for the floor required. The effect needed was a simple assignment to the floor variable with the new value. In OODDL, the floors did not need to be passed in as action parameters; they could be referred to directly. This is shown in the specimen answer in section 5.2.3.

In STRIPS however, the candidate was required to do a little more work in order to achieve the same effect, they had to assert both that the lift was at the floor passed as the source floor, and that the source and destination floor passed in were the ones required. This gave greater scope for errors from the STRIPS candidates.

On the OODDL test, only three candidates answered this question incorrectly. One candidate referred directly to problem domain objects instead of making use of the action parameters. The other two made the trivial error of substituting the numeral for the constant name of the floor, i.e. putting "1" instead of "one" etc. Otherwise, their structure and intentions were perfectly clear.

The STRIPS candidates made two common errors, the most overwhelming common of which was confusion between predicates and objects. As discussed in section 5.4, it was required that the candidate assert transitively that the lift be at a specific floor. This should have been achieved with:

```
at-floor(lift,floor)
floor-1(floor)
```

Yet, a number of candidates attempted to assert this fact with:

```
at-floor(lift,floor-1)
```

Besides these mistakes, there were two cases of missing negative effects, which broke the single-valued-ness invariant for the lift's floor relation.

The difference in the time spent between the two questions is most likely an indication of the extra work required for the STRIPS version. In STRIPS, the candidates have more preconditions to both write and think about.

The techniques of transitively asserting facts in preconditions is a very common one in STRIPS, it is used in most domains, including the standard logistics domain where it is used to assert that a package is at the same location as the vehicle loading it.

```
at(package,loc)
at(vehicle,loc)
```

If candidates do not understand how to use transitive assertions in this way, then it is very likely that they would run into a number of problems when attempting to encode domains in STRIPS.

OODDL doesn't make use of transitive assertions; instead, variables can be directly compared to one another. An equivalent statement in OODDL for the previous example would be:

```
package.at==vehicle.at
```

OODDL's use of enumerated types and facility to directly compare variables is a powerful tool when encoding domains and, as this test has shown, is easier and quicker to work with than transitive assertions.

5.7 Summary of Tests

The tests were successful in demonstrating that OODDL can address some common errors in STRIPS domain modelling. It was hypothesised that omitted negative effects would be a prevailing error in the answers from the STRIPS

tests, however instead the prevailing error was found to be a confusion between predicates and objects possessing the predicates.

OODDL's enumerated type support allowed candidates to easily construct the lift domain used in question 3. By contrast, a high proportion of the STRIPS candidates were unable to correctly answer this question, mainly due to confusing predicates and objects.

There were only three cases of omitted negative effects across all of the STRIPS tests. It is believed that this small number of errors is related to the simplicity of the domains used in the tests. It is anticipated that omitted negative effects would be a prevailing error in more complex questions featuring more state maintenance from the domain actions. In these cases, OODDL's use of single-valued variables would allow for easier domain construction.

A further hypothesis was that OODDL is easier to understand than STRIPS. These tests have demonstrated only that OODDL is no harder to understand than STRIPS. However, it is postulated that with larger domain instances, OODDL's inherit structure would make the description more understandable than the equivalent encoding in STRIPS.

Another interesting point with the test results is that some STRIPS candidates put in typing preconditions in the actions, to ensure the parameters were of the correct type. In these specific questions, the lack of different types in the domains meant that the domain description was still completely correct without typing preconditions, however it is unlikely that the candidates who omitted them did so for that reason. If the tests had have required the candidates to differentiate between similar types, it is likely that many of the candidates would have omitted typing preconditions and subsequently produced incorrect answers.

As parameters to OODDL actions are all typed, it is not possible to omit typing requirements in OODDL. This could have been used to demonstrate another key difference between STRIPS and OODDL.

One of the common errors from the OODDL candidates was confusion between problem domain object instances and actions parameters. In the interactive domain-engineering tool Draughtsman, action expressions are built using wizards that present lists of variables, followed by operations that can be

performed on them, finally followed by values that can be used with the operators. Draughtsman would have forced candidates to make well-formed expressions and it would have been impossible for the candidates to refer to problem domain objects, as they wouldn't be listed in the expression creator wizard. Essentially, none of the errors made by the OODDL candidates on questions 2 and 3 could have been made if the domains had been created in Draughtsman.

It was discussed in section 2.7, that an earlier version of Draughtsman edited STRIPS domains with wizards, thus ensuring preconditions and effects were well formed. Although this tool would have stopped STRIPS candidates becoming confused about the difference between objects and predicates, as seen in question 3, it would not have assisted them in creating the transitive assertions required, nor would it have offered assistance in enforcing single-valued-ness invariants. This level of assistance is not available in STRIPS because it is not structured enough to allow users to declare these requirements.

6 Conclusion

This work has been an investigation into the usefulness of object-orientated techniques in planning domain engineering. The products of the study were: an object-orientated domain modelling language (OODDL), an object-orientated generic type description language (GTL) and an interactive domain engineering tool that worked with these two languages (Draughtsman).

6.1 OODDL

OODDL was developed by analysing the common requirements of planning domain models and taking into account the shortcomings of STRIPS. OODDL's object-orientated structure was created to allow planning models to relate more easily to real world domains.

The decision to use variables instead of predicates was taken to make OODDL more accessible to a wider technical audience, such as those familiar with C++ or Java, but not necessarily comfortable with predicate logic. The use of variables also allowed the creation of explicit object reference relations between objects, references that were explicitly stated as only having a single value; this invariant was enforced by the language design.

OODDL gave the domain engineer access to tools such as full typing, inheritance, method overriding and enumerated types. The hypothesis being that a large number of computer programmers are already familiar with such tools and would be able to model domains easily.

To evaluate how useful OODDL is, it was pitted against STRIPS in a written test taken by undergraduate candidates familiar with both Java and planning in general. The tests gave domain excerpts that the candidates had to either understand and answer questions on, or complete by means of adding preconditions and effects to existing operator skeletons.

A simple STRIPS syntax was used rather than the bracket intensive PDDL because it the tests were designed to compare the ideas behind OODDL to those

behind STRIPS. PDDL would have added an unnecessary layer of noise to the results.

The results indicate that both OODDL and STRIPS are equally easy to understand. However, it was further hypothesised that large OODDL domains may be easier to comprehend than large STRIPS domains; in the same way that large object-orientated C++ programs are generally easier to work with than large C programs.

The use of variables in OODDL had a generally positive effect on the models, eliminating some errors and making domains descriptions more concise. However, it caused confusion for some candidates; these candidates made fundamental errors that would have been errors in *any* language, such as referring to variables outside the scope of the action. The use of the domain-engineering tool Draughtsman would have eliminated these errors by helping the user build valid actions.

Although it was not possible to evaluate all features of OODDL in depth, the evaluation of OODDL's enumerated types was very beneficial. Candidates working with OODDL were able to model the specified domain with over three times the success rate of the STRIPS candidates; they also created the models in less time.

OODDL clearly demonstrates that well designed higher-level semantics can assist the domain engineer in domain modelling.

6.2 GTL

The usefulness of GTL, the generic type language, could not be directly evaluated. This was due to a need for tools that applied GTL descriptions to domains or used them to build domains. Draughtsman was being extended to do this but was not yet at a testable stage.

GTL has been used to model several generic types, including:

- Mobiles
- Carriers
- Construction
- Multi-processor schedules

The potential for GTL to offer a secondary channel of communication between the domain engineer and the planner is great, and there is a large scope for future work with GTL.

6.3 Draughtsman

Draughtsman is a tool capable of editing OODDL domains and translating them into PDDL STRIPS. It can also be compiled into a library and used to parse and obtain information about these domains. This functionality was demonstrated by attaching a Java GUI to the library and using it to edit and translate OODDL domains. By using Draughtsman, domain engineers can also be assured that their domains will be syntactically correct.

Draughtsman is able to apply some GTL templates to a domain to identify generic types. In particular, it can recognise mobiles and location maps, and then provide a customised map editor allowing the user to manipulate the map. With further work, Draughtsman could apply GTL templates to a domain and then allow clients to access the results though its library API.

The high-level nature of OODDL allows different underlying domain modelling languages to be targeted for translation, for example ADL or OCL. Draughtsman provides a solid base from which such work could begin.

6.4 Scope for Future Work

The scope for future work in this area is huge. Good domain engineering tools will allow planners to be more easily deployed, and raise the status of planning in the wider community.

OODDL can provide a foundation for this work, further development on OODDL could include:

- Static class variables
- Disjunctive preconditions
- Conditional or quantified effects
- HTN extensions

Preliminary work has already been done on implementing conditional effects by using multiple STRIPS actions to represent a single OODDL action. Each STRIPS action has different preconditions to ensure its effects only take place when the original condition is true.

GTL is also a useful tool, for both domain engineer tools and planners. It will allow domain specific information to be encoded in a domain independent manner, allowing heuristics and editing hints to be reapplied to any domain that has a similar structure. Future development on GTL could include:

- Extending TIM to use GTL templates
- Development of more GTL services, such as a detailed grammar for temporal heuristics or sub-solver specifications

Common to the development of both OODDL and GTL is Draughtsman. Draughtsman provides a flexible tool for working with, or allowing other tools to work with, OODDL and GTL. Future development on Draughtsman could include:

- Further development on the Java based GUI
- Use of TIM domain analysis techniques to optimise STRIPS output
- Improve the GTL application algorithms for efficiency and compatibility
- Ability to create new domains based on GTL templates
- Integration with existing planners to create a one stop planning solution from modelling through to resultant plan
- Translation from OODDL to other languages such as ADL or OCL

6.5 Summary

This work has been an investigation into the usefulness of object-orientation in planning domain engineering. This involved the development of the object-orientated domain description language, OODDL, which was shown to be more effective at describing certain domains than the existing STRIPS language.

The role of generic types in domain engineering was also investigated, cumulating in the development of the generic type description language, GTL. GTL offers great potential as a new tool for conveying domain-dependent information from the domain engineer to the planner in a domain-independent manner. Some of the potential uses for GTL and its possible future developments are discussed.

A domain-engineering tool called Draughtsman was developed as a means of working with OODDL and GTL. Draughtsman can translate OODDL domains into PDDL STRIPS meaning that OODDL will be compatible with many existing planners; this should assist its adoption as a domain engineering language.

7 Appendices

7.1 Notation for OODDL Domains

During the development of OODDL, all domains were edited by the domain-engineering tool Draughtsman. Draughtsman uses a binary file format to store domains rather than an ASCII-based meaning that a parsable formal grammar was not required, nor developed, for OODDL.

However, a text-based notation has been developed for OODDL, simply as a way of writing down domain descriptions in a human readable form; although this notation is largely self-explanatory, this section provides a very informal description of it. OODDL is discussed in detail in chapter 3.

7.1.1 Domain Description

An OODDL domain description consists of a set of type declarations that are similar to Java or C++ classes. Each type can have member variables and member functions (called *actions*).

7.1.2 Member Variables

A member variable can be of four types:

Boolean:

A simple boolean variable that can be either true or false.

E.g. `boolean myBool`

Enum:

An enumerated variable, you must declare which enumerated type it refers to.

E.g. `enum Kind = { apple, orange, pear }`

`Kind myKind`

Object Reference:

Refers to an object of the declared type (or a subtype thereof) in the domain. If the variable name is prefixed with a * then it means the variable is a maybe object reference and can hold a NULL value. If there is no * then the variable is an object reference and *cannot* hold the NULL value.

```
E.g.  MyType     *myMaybeObRef   ' Maybe object ref
      Mytype     myObRef         ' Object ref
```

Note: Object reference variables can be dereferenced using the *dot* notation as seen in Java and C++.

```
E.g.  a.b.c=d
```

Object Bag:

The object bag is an unordered collection of objects. Objects in the bag are of the type specified, or a subtype thereof. The object bag is declared in a similar way to the object reference, except that it is suffixed with a pair of brackets.

```
E.g.  MyType     mySet[]
```

7.1.3 Actions

Actions in OODDL are divided into three parts: the parameters list, the preconditions and the effects.

Parameters:

A typed list of variables that can be used to form expressions.

Preconditions:

A set of expressions that must be true before the action can be invoked. Precondition expressions are prefixed with a "p:", this is differentiate them from effect expressions, which are prefixed with an "e:".

Possible expressions are:

`a == b` **Equality**

`a != b` **Inequality**

Variables are equal/not equal. Can be used with *booleans*, *enums* and *object references*. You can compare object references to NULL.

`a ∈ b` **Set Membership**

The set **b** contains the object **a**. Can only be used with *sets*. *Note: There is no ∉ operator.*

Effects:

A set of expressions that are applied when the actions preconditions are met. Effect expressions are prefixed with a "e:" in the action definition to differentiate them easily from the preconditions, which are prefixed with a "p:".

Possible effects expressions are:

`a = b` **Assignment**

Variable **a** becomes equal to value **b**. This can be used for *parameters, booleans*, *enums* and *object references.*

a += b Add to bag

a -= b Remove from bag

Object **b** is added to/removed from the set **a**. This can be used for *object bags* only.

7.1.4 A Problem Specification

A problem specification in OODDL has two parts. The first is a list of objects in the domain, along with their type and initial values for all their member variables. The second is a list of goal conditions that are required to be true at the end of the plan.

7.1.5 OODDL Example: Blocks World

This is an encoding of the ubiquitous blocks world in OODDL.

```
type block
    block       *on
    boolean     clear

    put_on_block(block block)
        p:block!=this
        p:block.clear=true
        p:on=<None>
        p:clear=true
        e:block.clear=false
        e:on=block
    end
    put_on_table()
        p:clear=true
        e:on=<None>
        e:on.clear=true
    end
end
```

7.2 Formal Grammar for GTL

This section contains a formal grammar for the GTL generic type template language. GTL is used to describe generic types declaratively rather than the current method of describing them procedurally.

GTL's syntax is similar to the notation used for OODDL, however it is a more abstract way of describing a domain than OODDL. This allows several different OODDL domain definitions to be matched to the same GTL template.

GTL is discussed in detail in chapter 4.

7.2.1 Template Tags

A GTL template file is divided into sections denoted by tags. The main tag is the *types* section, which is where the generic type template descriptions actually reside. Related to the types tag is the *instances* tag. This tag provides two functions; firstly, it lists the type combinations that must be matched in order to instantiate the template. Secondly, it names these instances so that other templates can extend the template and refer to matched generic type instances.

Other simple tags include the version of the file format and the name of the template. The option of implementing other tags is for future expandability viz., to offer new generic type services. Currently there are no implemented GTL services and so there are no grammars for the services tags. Only two tags currently exist: the types tag and the instances tag.

7.2.2 The Types Tag

This is the grammar for the types tag. The types tag defines the generic type structures.

```
%%start=gtlTypesSection
```

```
gtlTypesSection := ENDLINES typeslist | typeslist
typeslist := typeslist type | null
type := typeHeader propList actionList END ENDLINES
typeHeader := TYPE STRING extends ENDLINES
extends := EXTENDS STRING | null


propList := propList prop | null
prop := STRING STRING ENDLINES | STRING PLUS STRING
                ENDLINES | BOOLEAN STRING ENDLINES


actionList := actionList action | null
action := actionHeader expsList END ENDLINES
actionHeader := STRING OPENB plist CLOSEB ENDLINES
plist := param commaedplist | null
commaedplist := COMMA param | null
param := STRING STRING


expsList := expsList exp | null
exp := actionInstance | EXPTYPE STRINGPATH OPERATOR
                                STRINGPATH ENDLINES
actionInstance := STRING OPENB strpathList CLOSEB
                                        ENDLINES


strpathList := STRINGPATH commaedStrPathList
commaedStrPathList := COMMA strpathList | null


STRING := std identifier
STRINGPATH := std identifier with optional "."
              breaking fields (eg a.b.c)
TYPE := "type"
EXTENDS := "extends"
ENDLINES := "\n"*
```

```
END := "end"

PLUS := "+"

COMMA := ","

OPERATOR := ":=" | "==" | "!=" | "in" | "+="
                                  | "-=" | "~="

CLOSEB := ")"

OPENB := "("

EXPTYPE := "p:" | "e:"

BOOLEAN := "boolean"

REM := "'".*

WHITESPACE := " \t"
```

7.2.3 The Instance Tag

The instance tag denotes which types from the types tag must be instantiated (matched) in order for the template instance to be valid. This is useful for complex templates where only one of several possible generic type definitions must be matched, or where multiple occurrences of the same generic type must be matched. If the instance tag is omitted, then one instance of each root type (or one if its subtypes) is matched.

The types listed, or a subtype thereof, must be matched and named. Because instances are named, other templates can import the template and refer to instantiated generic types.

```
%%start = typeInstanceList


typeInstanceList := typeInstanceList typeInstance |
                                    null
typeInstance := STRING STRING ENDLINE
```

7.2.4 GTL Example: Mobile

The mobile represents the notion of a self-propelled object that can move around a map of locations. The mobile is discussed in section 2.6.1.

```
#gtlversion 1
#name Mobile

' describes the Mobile and Location types
#section types
     TYPE Mobile
          Location  at

          move (Location to)
               p: at != null
               e: at := to
          end
     end

     TYPE Location
     end
#endsection


' matches a single mobile and a location
#section instance
     Mobile          aMobile
     Location        aLocation
#endsection
```

7.2.5 GTL Example: Construction

Construction is a more complex GTL template definition. The construction generic type attempts to capture the notion of a construction, or building component composed of other components. There are two operators, *join* and *split* for building and destroying components respectively.

```
#gtlversion 1
#name Construction

#section types
     type Component
          boolean         available
          Component       +subcomponents

          join (Component c)
               p: this != c
               p: available == true
               p: c.available == true
               e: c.available := false
               e: subComponents += c
          end
     end

     type GeneralComponent extends Component
          split (Component c)
               p: c in subcomponents
               p: available == true
               e: subComponents -= c
               e: c.available := true
          end
     end
#endsection

' by not including an instance section, either
' GeneralComponent or Component will be matched
```

7.3 Test: Understanding STRIPS

This test is designed to evaluate how easily the candidate can understand the intricacies of a domain encoded in STRIPS.

Task

This test consists of three parts. Answer all parts. You may refer to the language notes when answering the questions.

Question 1: Action Application

This description describes a world containing a tower of blocks. It has an action to lift a block from the table into the air (only one block can be in the air at once) and another to put a block down on top of another block.

STRIPS Domain Description

Predicates:
- on(a,b)
- clear(a)
- on-table(a)
- arm-empty()
- is-held(b)

Actions:

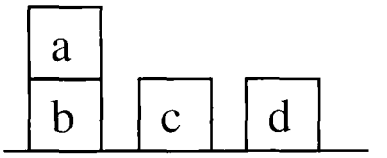
pickUpFromTable(block)
- preconditions:
  - arm-empty()
  - clear(block)
  - on-table(block)
- effects:
  - is-held(block)
  - ¬on-table(block)
  - ¬arm-empty()

end

putOnTower(block,to)
- preconditions:
  - is-held(block)
  - clear(to)
- effects:
  - on(block,to)
  - ¬is-held(block)
  - ¬clear(to)

end

STRIPS Problem Description



Objects:
- a,b,c,d

Initial State:
- on(a,b)
- clear(a)
- clear(c)
- clear(d)
- on-table(d)
- on-table(c)
- on-table(b)
- arm-empty()

Goal State:
- on(c,d)
- on(d,a)
- on(a,b)

1. Is it possible to execute the action *pickUpFromTable(d)*?

2. There is a problem with one of the actions in the domain, for some reason the planner cannot stack block c **after** it has stacked d on top of a. Can you identify the problem and correct it?
   Hint: Write out the states after each action execution if it helps.

Question 2: Action Construction

This domain features two buckets and a "thing" that can be placed in only one bucket at a time. Given a partially complete world description you must construct an action to move a thing from one bucket to another.

You are given the types and an empty action. Complete the action so that it moves a thing from one bucket to another. Remember that an action has two parts: the preconditions and the effects. Your action can only work with the action parameters and the predicates declared in the domain. An example problem has been declared to aid in your understanding of the question.

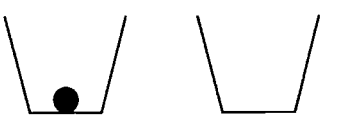
STRIPS Domain Description

Predicates:
in(thing,bucket)
bucket(ob)
thing(ob)

Actions:
move(thing,from,to)
Fill out the contents of this action!
preconditions:

effects:

STRIPS Problem Description



bucket1 bucket2

Objects:
bucket1, bucket2, thing1

Initial State:
bucket(bucket1)
bucket(bucket2)
thing(thing1)
in(thing1,bucket1)

Goal:
in(thing1,bucket2)

Question 3: Action Preconditions

For this question, you will once more have to fill out the contents of two actions. This domain concerns a lift. The lift can only be at one floor at once and can only move one floor at a time.

Below is a partial domain description for this domain. You need to fill out the two actions for the lift. One moves it from being at floor one to floor two, and the other from floor two to floor three. All the parameters and predicates needed have been declared for you. An example problem has also been declared to aid in your understanding of the question.

IMPORTANT: Ensure that the lift is only permitted to visit the floors in the correct order. Floor 2 and then floor 3.

STRIPS Domain Description

Predicates:
- at-floor(lift,floor)
- lift(ob)
- floor-1(ob)
- floor-2(ob)
- floor-3(ob)

Actions:

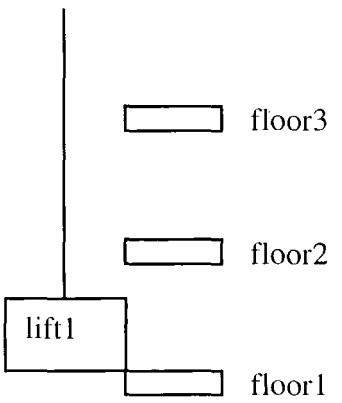
moveFrom1To2(lift,wasFloor,newFloor)
- *Fill out the contents of this action!*
- **preconditions:**
- **effects:**

end

moveFrom2To3(lift,wasFloor,newFloor)
- *Fill out the contents of this action!*
- **preconditions:**
- **effects:**

end

STRIPS Problem Description



Objects:
lift1, floor1, floor2, floor3

Initial State:
- floor-1(floor1)
- floor-2(floor2)
- floor-3(floor3)
- lift(lift1)
- at-floor(lift1,floor1)

Goal:
- at-floor(lift1,floor3)

7.4 Test: Understanding OODDL

This test is designed to evaluate how easily the candidate can understand the intricacies of a domain encoded in OODDL.

Task

This test consists of three parts. Answer all parts. You may refer to the language notes when answering the questions.

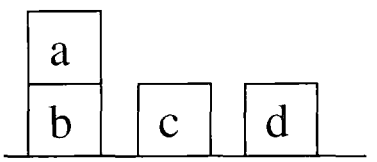
Question 1: Action Application

This description describes a world containing a tower of blocks. It has an action to lift a block from the table into the air (only one block can be in the air at once) and a different action to put a block down on top of another block.

OODDL Domain Description

```
type Block
  Block        *onTopOf
  boolean      clear
end

type Arm
  Block        *holding

  pickUpFromTable(Block b)
    preconditions:
        holding==NULL
        b.clear==true
        b.onTopOf==NULL
    effects:
        holding=b
  end

  putOnTower(Block b)
    preconditions:
        holding!=NULL
        b.clear==true
    effects:
        holding.onTopOf=b
        b.clear=false
  end

end
```

OODDL Problem Description



Initial Values:

```
object a : Block
  onTopOf=b
  clear=true
end

object b : Block
  onTopOf=NULL
  clear=false
end

object c : Block
  onTopOf=NULL
  clear=true
end

object d : Block
  onTopOf=NULL
  clear=true
end

object arm : Arm
  holding=NULL
end
```

Goal Conditions:

```
c.onTopOf==d
d.onTopOf==a
a.onTopOf==b
```

1. Is it possible to execute the action *arm.pickUpFromTable(d)*?

2. There is a problem with one of the actions in the domain, for some reason the planner cannot stack block c **after** it has stacked d on top of a. Can you identify the problem and correct it?
   Hint: Write out the values of the objects' variables after each action execution if it helps.

Question 2: Action Construction (Variant 1)

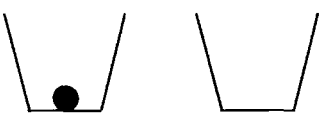
This domain features two buckets and a "thing" that can be placed in only one bucket at a time. Given a partially complete world description you must construct an action to move a thing from one bucket to another.

You are given the types and an empty action. Complete the action so that it moves a thing from one bucket to another. Remember that an action has two parts: the preconditions and the effects. You do not need to modify the domain, all the variables and parameters you need are there. An example problem has been declared to aid in your understanding of the question.

OODDL Domain Description

```
type Thing
end

type Bucket
  Thing         contents{}

  move-into-bucket(Bucket b,Thing t)
        Fill out the contents of
        this action!
        preconditions:


        effects:


  end
end
```

OODDL Problem Description



bucket1 bucket2

Initial Values:

```
object bucket1 : Bucket
        contents={thing1}
end

object bucket2 : Bucket
        contents={}
end

object thing1 : Thing
end
```

Goal Condition:
thing1 ∈ bucket2.contents

Question 2: Action Construction (Variant 2)

This domain features two buckets and a "thing" that can be placed in only one bucket at a time. Given a partially complete world description you must construct an action to move a thing from one bucket to another.

You are given the types and an empty action. Complete the action so that it moves a thing from one bucket to another. Remember that an action has two parts: the preconditions and the effects. You do not need to modify the domain, all the variables and parameters you need are there. An example problem has been declared to aid in your understanding of the question.

OODDL Domain Description

```
type Thing
    Bucket          *inside

    move-into-bucket(Bucket b)
        Fill out the contents of
        this action!
        preconditions:


        effects:


    end
end

type Bucket
end
```

OODDL Problem Description

bucket1 bucket2

Initial Values:

```
object bucket1 : Bucket
end

object bucket2 : Bucket
end

object thing1 : Thing
        inside=bucket1
end
```

Goal Condition:

thing1.inside==bucket2

Question 3: Action Preconditions

For this question, you will once more have to fill out the contents of two actions. This domain concerns a lift. The lift can only be at one floor at once and can only move one floor at a time.

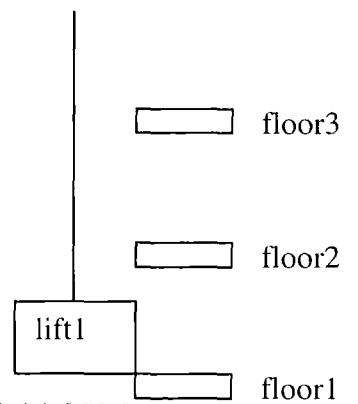
Below is a partial domain description for this domain. You need to fill out the two actions for the lift. One moves it from being at floor one to floor two, and the other from floor two to floor three. All the variables and types needed have been declared for you. An example problem has also been declared to aid in your understanding of the question.

IMPORTANT: Ensure that the lift is only permitted to visit the floors in the correct order. Floor 2 and then floor 3.

OODDL Domain Description

```
enum FloorNumber = { one, two, three }

type Lift
        FloorNumber          floor

        moveFromOneToTwo()
                Fill out the contents of
                this action!
                preconditions:

                effects:

        end

        moveFromTwoToThree()
                Fill out the contents of
                this action!
                preconditions:

                effects:

        end
end
```

OODDL Problem Description



Initial Values:

```
object lift1 : Lift
        floor=one
end
```

Goal Condition:

lift1.floor==three

8 References

[Abbot, 1983] *Program Design by Informal English Descriptions*
R. Abbot
Communications of the ACM, Vol. 26 (11)

[Bacchus and Fabanza, 2000] *Using Temporal Logics to Express Search Control Knowledge for Planning*
F. Bacchus, F. Kabanza
Artificial Intelligence, Vol. 116

[Beck and Cunningham, 1989] *A Laboratory for Teaching Object-Orientated Thinking*
K. Beck, W. Cunningham
SIGPLAN Notices, Vol. 24 (10)

[Booch, 1991] *Object-Orientated Design with Applications*
G. Booch
Benjamin Cummins

[Chen and Warren, 1988] *Objects as Intensions*
W. Chen, D. S. Warren
Proceedings of 5th International Conference on Logic Programming, pages 404-419

[Clark, 2000] *Construction Domains: Their Detections and Exploitation in TIM*
M.S. Clark
Proceedings of PLANSIG 2000

[Coad and Nicola, 1993] *Object-Orientated Programming*
P. Coad, J. Nicola
Yourdon Press

[Coad and Youren, 1990] *Object Orientated Analysis*
P. Coad, E. Youren
Prentice Hall

[Conery, 1987] *Object Oriented Programming with First Order Logic*
J.S. Conery
University of Oregon Tech Report CIS-TR-87-09

[Cox, 1986] *Object-orientated Programming: An Evolutionary Approach*
B. Cox
Addison Wesley

[Date, 1999] *An Introduction to Database Systems, 7th Edition*
C.J. Date
Addison Wesley

[desJardins, 1994] *Knowledge Development Methods for Planning Systems*
M. Des Jardins
Proceedings of the AAAI Fall Symposium on Planning and Learning, New Orleans

[Erol, 1995] *Hierarchical Task Network Planning: Formalization, Analysis and Implementation*
K. Erol
Ph.D. Thesis, University of Maryland

[Erol et al., 1994] *UMCP: A Sound and Complete Procedure for Hierarchical Task Network Planning*
K. Erol, J. Hendler, D.S. Nau
Proceedings of AIPS '94

[Eysenck and Keane, 2000] *Cognitive Psychology: A Student's Handbook, 4th Edition*
Michael Eysenck, Mark T. Keane
Psychology Press

[Fikes and Nilsson, 1971] *STRIPS: A New Approach to Theorem Proving in Problem Solving*
R. E. Fikes, N. J. Nilsson
Artificial Intelligence, Vol. 2, pages 189-208

[Fox and Long, 1998] *The Automatic Inference of State Invariants in TIM*
D. Long, M. Fox
Journal of Artificial Intelligence Research, Vol. 9, pages 367-421

[Fox and Long, 1999] *The Automatic Synthesis and use of Generic Types in Planning*
D. Long, M. Fox
Proceedings of the 18th Workshop of the UK Planning and Scheduling Special Interest Group

[Fox and Long, 2000] *Hybrid STAN: Identifying and Managing Combinatorial Optimisation Sub-Problems in Planning*
D. Long, M. Fox
Proceedings of PLANSIG 2000

[Fox and Long, 2000b] *Extracting Route Planning: First Steps in Automatic Problem Decomposition*
D. Long, M. Fox
Workshop on Analysing and Exploiting Domain Knowledge for Efficient Planning, AIPS-2000

[Fox and Long, 2001] *Multi-Processor Scheduling Problems in Planning*
D. Long, M. Fox
Proceedings of 2nd IC-AI, Special Session on "Learning and Adapting in AI Planning", Las Vegas

[Fox and Long, 2001b] *PDDL 2.1 Language Specification*
D. Long, M. Fox
http://www.dur.ac.uk/d.p.long/competition.html
Valid: 1st October 2001

[Gardarin et al, 1997] *Object Technology: Concepts and Methods*
M. Bouzeghoub, G. Gardarin, P. Valduriez
International Thomson Computer Press

[Gazen and Knoblock, 1997] *Combining the Expressivity of UCPOP with the Efficiency of Graphplan*
B.C. Gazen, C.A. Knoblock
Proceedings of ECP 1997

[Geffner, 2000] *Functional STRIPS: A More Flexible Language for Planning and Problem Solving*
H. Geffner
Logic-Based Artificial Intelligence, Jack Minker (Ed.), Kluwer Academic Publishers

[Gerevini and Schubert, 1998] *Inferring State Constraints for Domain Independent Planning*
A. Gerevini and L. Schubert
Proceedings of AAAI-98

[Grant, 1996] *Inductive Learning of Knowledge-Based Planning Operators*
T. J. Grant
Ph.D. Thesis, Universiteit Maastricht, Maastricht

[Green, 1979] *When do Diagrams Make a Good Computer Language?*
M. Fitter, T.R.G. Green
International Journal of Man-Machine Studies, Vol. 2, pages 235-261

[Gruber, 1993] *A Translation Approach to Portable Ontologies*
T. R. Gruber
Knowledge Acquisition, Vol. 5(2), pages 199-220

[Jackson, 1975] *Principles of Program Design*
M.A. Jackson
Academic Press

[Klerer and May, 1965] *A User-Orientated Programming Language*
M. Klerer, J. May
Computer Journal 8, No 2, July 1965

[Koehler, 1998] *Solving Complex Planning Tasks through Extraction of Sub-Problems*
J. Koehler
Proceedings of AIPS-98

[Lifschitz, 1986] *On the Semantics of STRIPS*
V. Lifschitz
Proceedings of the 1986 Workshop on Planning and Reasoning about Action, Timberline, Oregon

[McCluskey and Kitchin, 1998]
A Tool-Supported Approach to Engineering HTN Planning Models
T. L. McCluskey, D.E.Kitchin
Proceedings of the Tenth International Conference on Tools with Artificial Intelligence (TAI'98)

[McCluskey and Liu, 1999] *The OCL Language Manual*
D. Liu, T.L. McCluskey
Technical report, Department of Computing Science, University of Huddersfield

[McDermott et al, 1998] *The Planning Domain Definition Language*
D. McDermott, AIPS-98 Competition Committee
http://cs-www.cs.yale.edu/homes/dvm/
Valid: 1st October 2001

[Pednault, 1989] *ADL: Exploring the Middle Ground between STRIPS and Situation Calculus*
E. P. D. Pedault
Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning

[Porteous and McCluskey, 1997]
Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency
J. Porteous, T.L. McCluskey
Artificial Intelligence, Vol. 95 (1), pages 1-65

[Saeki et al, 1989] *Software Development Process from Natural Language Specification*
M. Saeki, H. Horai, H. Enomoto
Proceedings of the 11th International Conference on Software Engineering

[Simpson et al, 2000] *Knowledge Representation in Planning: A PDDL to OCL Translation*
R.M. Simpson, T. L. McCluskey, D.Liu, D.E.Kitchin
Proceedings of ISMIS 2000, Charlotte, NC.

[Simpson et al, 2001] *GIPO: An Integrated Graphical Tool to Support Knowledge Engineering in AI Planning*
R.M.Simpson, T.L. McCluskey, W. Zhao, R.S. Aylett, C. Doniat
Proceedings of ECP 2001

[Sommerville, 1992] *Software Engineering, 4th Edition*
I. Sommerville
Addison Wesley

[Stroustrup, 1994] *The Design and Evolution of C++*
B. Stroustrup
Addison-Wesley

[Veloso, 1992] *Learning by Analogical Reasoning in General Problem Solving*
M. Veloso
Tech. Report CMU-CS-92-174, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA

[Wilkins and desJardins, 2000]
A Call for Knowledge Based Planning
D. E. Wilkins, M. Des Jardins
Workshop on Analysing and Exploiting Domain Knowledge for Efficient Planning, AIPS-2000

[Wirth, 1971] *Program Development by Stepwise Refinement*
N. Wirth
Communications of the ACM, Vol. 14 (4), page 221-7