Chapter 4
Planning

Planning with Generic Types

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

Abstract

Domain-independent, or knowledge-sparse, planning has limited practical application because of the failure of brute-force search to scale to address real problems. However, requiring a domain engineer to take responsibility for directing the search behavior of a planner entails a heavy burden of representation and leads to systems that have no general application. An interesting compromise is to use domain analysis techniques to extract features from a domain description that can exploited to good effect by a planner. In this chapter we discuss the process by which generic patterns of behavior can be recognized in a domain, by automatic techniques, and appropriate specialized technologies recruited to assist a planner in efficient problem solving in that domain. We describe the integrated architecture of STAN5 and present results to demonstrate its potential on a variety of planning domains, including two that are currently beyond the problem-solving power of existing knowledge-sparse approaches.

1 Introduction

Research in planning has been an active branch of AI since Newell and Simon's work on GPS (Newell and Simon 1963) and Green's work on QA3 (Green 1969) in the 1960s. Planning is a loaded term, carrying strong connotations for most people. In AI the most widely accepted usage of the term refers to the activity of selecting between alternative sequences of actions that can be executed from a given initial state of a modeled world in order to achieve some goal. The world is modeled in terms of the domain of objects, relations, and state transitions that capture its key problem-solving aspects. The planning

problem is simplified in classical planning by assuming that states are sets of propositional atoms drawn from a finite universe, that the goal is characterized by a finite set of propositions, and that actions are deterministic state-transition functions. Under these assumptions, the general planning problem is PSPACE-hard (Bylander 1992).

An important division in the planning field exists between *knowledge-rich* and *knowledge-sparse* planning. Knowledge-rich planning systems are able to exploit carefully crafted advice, in the form of expert domain knowledge, produced by a domain engineer. The underlying planning strategies they are based upon involve traversing the encoded knowledge with a minimum of search. Search is expensive and the knowledge-rich approach to avoiding it is to design *choice-points* out of the domain representation. In contrast, knowledge-sparse planning systems exploit weak general heuristic strategies to combat the combinatorial explosion inherent in a domain description that captures only the physical dynamics of the world being modeled. The decision making that supports the planning process is uninfluenced by advice from the domain engineer, and search is the means by which a potentially very large solution space is explored.

Experience has shown that, although we can learn many useful lessons about general problem-solving activity by the study of knowledge-sparse planning, it is knowledge-rich planning that leads most readily to application (Currie and Tate 1991; Jónsson, Morris, Muscettola, Rajan, and Smith 2000; Muscettola 1994; Nau, Gupta, and Regli 1995; Wilkins and desJardins 2000). This is unsurprising—planning in all but the most trivial of domains, even under the simplifying assumptions of classical planning, is too hard for exhaustive search to be a reasonable approach to take. Advice given by knowledgeable human problem solvers can sensibly prune the search space a planner must explore to a manageable size and can even allow a relaxation of some of the strong assumptions of classical planning. Unfortunately, it is hard to generalize from the experiences of problem solving with knowledge-rich systems, since the nature of the advice they exploit is typically strongly bound to the domain and does not easily transfer to new domains.

The work we describe in this chapter is motivated by the observation that many practical problems share common, or *generic*, features. For example, planning to achieve parcel deliveries between depots shares its underlying *transportation* nature with the apparently different problem of ensuring that building materials are available at the appropriate locations when needed on a building site. In these two domains, different predicates and object names are used to encode essentially the same underlying transportation behavior. The heart of the transportation problem, which arises in both cases, is to move objects (e.g., parcels or building materials) between locations (e.g., depots or building sites), identifying the best routes for traveling between locations independently of the roles the objects might play at their destinations. Planning technology can be made more powerful if it is able to recognize the generic structure of a domain behind the domain-specific language used to encode it and to exploit its presence by decomposing the planning problem around efficient handling of the associated subproblems.

If a knowledge-sparse planner can recognize generic structures of this kind and configure specialized technologies to handle these parts of a planning problem, it can compete, in performance terms, with a knowledge-rich planner equipped with domain-specific advice. An essential weakness of a knowledge-sparse planner lies in the application of weak search heuristics to hard problems, often embedded within planning problems, when specialized techniques for handling these problems already exist. An example is the planning of routes

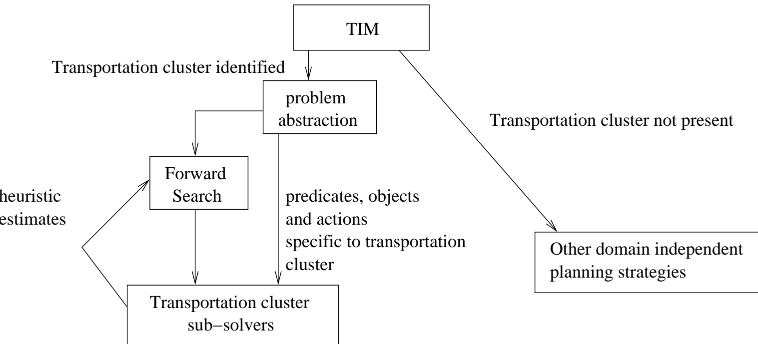


Figure 1 A hybrid planning architecture showing the main components of STAN5.

for vehicles that must visit several locations to fulfill tasks at each of them. This problem can appear as part of a planning problem, but there are already well-known techniques for finding efficient paths between locations and for deciding what order to visit locations so as to be efficient. A general planning strategy that uses the same weak heuristics to influence choices in solving this problem as to allocate tasks between different processors, or to schedule resources to activities, cannot be hoped to solve the problem as effectively as a specialized solver. A specialized solver can isolate the choices that are relevant to the subproblem from other decisions that the planner must make and can solve the subproblem with the benefit of richer and more specialized heuristics.

The successful integration of specialized solvers with a knowledge-sparse planning system presents many challenges. We have investigated these challenges in successive versions of STAN (STatic ANalysis planner), a planning system that exploits domain analysis techniques in order to bring domain-specific knowledge to bear on reducing search. The successive versions of STAN (indicated by version number) exploit increasingly sophisticated static analyses. STAN version 4 (Fox and Long 2001a) illustrates a first step towards using domain analysis to configure the integration of appropriate specialized techniques with a general-purpose planner. In (Fox and Long 2001b), we discuss how the architecture of the system can be made more robust and general, supporting the integration of a planner with multiple subsolvers through a uniform interface. Figure 1 shows the main components of STAN version 5 (STAN5). The integration of the system depends on the domain analysis techniques implemented in the *Type Inference Machinery* (TIM) system, discussed here (TIM is described in detail in Fox and Long 1998). Our work on STAN versions 4 and 5 has emphasized the automatic recognition of certain commonly occurring generic structures by analyzing the domain description prior to planning. In STAN5, this analysis constructs a profile of the domain in terms of its generic structure and enables the selection of appropriate subsolvers from a library. In Sections 4 and 5, we discuss the domain analysis techniques used to identify and isolate distinct subproblems within a planning domain.

In this chapter, we concentrate on describing the automatic recognition of a hierarchy of transportation-related generic types and behaviors. Transportation-related types occur in almost every planning problem, with varying degrees of centrality. Effective treatment of problems relating to transportation is therefore critical to achieving efficient plans in

most domains. This treatment can only be achieved by application of appropriate problem-solving technology, for route-planning, load-scheduling, driver-allocation, and so on, which, in turn, demands that these elements of planning problems be recognized and passed to these appropriate subsolvers. The role of generic types extends beyond transportation-related types as we and others have shown elsewhere (Long and Fox 2001; Long et al. 2000; Fox and Long 2001b; Clark 2001), but the predominance of the transportation-related generic types in planning applications makes their exploitation particularly powerful, regardless of whether other generic types are identified and exploited.

1.1 Overview

The chapter continues with a brief overview of planning domain descriptions, the status of control knowledge in these descriptions, and the development of automatic domain analysis techniques, among which fits the work described here. The foundations of this work, implemented in the TIM system, are reviewed in Section 3 and illustrated in a running example. In Section 4, we introduce *generic types*: the abstract patterns of behavior found in the fundamental structures constructed by TIM, that recur across different domains. We concentrate on a key sample generic type—the *mobile type*—together with related components, such as the locations on which instances of a mobile type move and the map that links them. We illustrate these ideas with an example drawn from the running example, showing how a further stage of structural analysis helps to uncover important supporting structure in this case. In Section 5, we consider other generic types related to the mobile type and then, in Section 6, we extend the treatment of the mobile generic type to include a broader generalization. There is a brief discussion, in Section 7, of the architecture of our planner, STAN5, that is capable of exploiting the generic type analysis, and we review some of the results of doing so in Section 8, before concluding the chapter.

In the interests of focusing on domain analysis techniques, we do not provide the details of how the components of STAN5 are integrated. Although we briefly overview the structure of the hybrid system in Section 7, the technical details of the integration are omitted and can be found in (Fox and Long 2001b), a related paper that is dedicated to the description of how plan generation is achieved in STAN5.

2 Planning Domain Description and Domain Analysis

The first domain descriptions for planning were given in terms of first-order axiomatizations of the dynamics of the worlds being modeled (Green 1969; McCarthy 1968; Newell and Simon 1963). In order to express how the application of an action changes the state of the world, axioms prescribe how actions transform states into their successor states. This modeling approach is known as the *situation calculus* (McCarthy 1968), and it formed the foundation for the development of domain representation languages. However, axiomatizing behavior in this way leads to a serious practical problem known as the *frame problem.* This is the problem of correctly modeling what stays stable, after application of an action, as well as what changes. If inertia is not correctly modeled, then inconsistent world models can be constructed. Unfortunately, in all but the most trivial worlds, correct modeling of inertia requires too many axioms to be written for the situation calculus to be a feasible approach to domain description.

A solution to the frame problem was proposed by Fikes and Nilsson (1971) in the development of the Stanford Research Institute Planning System (STRIPS). This used a language that explicitly modeled only what changed when an action was applied. In maintaining a consistent view of the world state, as planning progresses, the planning system makes the assumption that every relation in the world state, not explicitly listed as affected by the action last applied, remains stable. This assumption became known as the STRIPS assumption, and remains a fundamental principle of domain modeling to this day. Indeed, the STRIPS language, which represents each (parameterized) action in terms of its *preconditions* (the facts that must be true in a state in order for the action to be applicable to that state), its *add-effects* (the new facts that become true after application of the action), and its *delete-effects* (the facts that are false after application of the action), continues to be a strong influence on the development of modern domain languages because of the convenient way in which it models parameterized state change.

Throughout the 1970s, 1980s, and early 1990s, planning researchers developed a large family of domain description languages, all sharing the STRIPS assumption and all restricted to the modeling of finite, deterministic worlds. This lack of standardization resulted in it being difficult to compare planning systems or to accumulate a suite of benchmark domains. In 1998, Drew McDermott released the PDDL language—the first attempt to standardize planning domain modeling (McDermott 1998). PDDL has a number of levels, the lowest of which supports the STRIPS-style modeling of simple domains. Other levels support the modeling of simple numeric computations, the modeling of conditional effects, quantified effects, and preconditions—called the *Action Description Language* (ADL) level of the language, after the work of Pednault (1989), who originally considered the modeling of such domain features)—and language extensions to support hierarchical modeling of a domain.

Domain descriptions expressed in the standard planning languages, both the STRIPS and ADL variants of PDDL, comprise unstructured collections of operators that give a restricted picture of the highly structured domain model in the mind of the domain designer. It has often been observed that PDDL (and other domain-description languages based on the STRIPS assumption) is similar to an assembly language in terms of the level of abstraction it offers in the representation of a domain. Not only is the language difficult for the nonspecialist to use, it also offers no search-control assistance to the planner. Indeed, a PDDL domain description is an implicit description of the search space, which the planner simply expands and then searches using a mixture of brute force and weak heuristics.

The advantage of this approach to modeling is that it captures what *can* be done in a domain, rather than what *should* be done. While this places a greater burden on the planner, which has to search to determine which choices to eliminate, there is more scope for original solutions to be identified by the planner than if the domain designer constrains choice by embedding search control information in the domain description. McDermott (2000) argues strongly for a distinction between the encoding of *physics* and the encoding of *advice*. A key reason for this is that the way that advice is used, its interpretation, and the impact it has on the completeness of the planning process is all very much planning-system dependent, and failure to separate the advice from the physics both obscures that dependence and makes the domain description far less reusable.

Of course, the domain designer often has knowledge of the domain that must be exploited to obtain reasonable problem-solving efficiency in that domain, and it can be argued that preventing the domain designer from expressing this important knowledge simply has

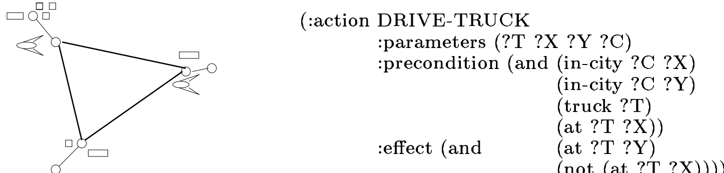


Figure 2 A simple logistics model and one operator from the domain. Thick lines are flight paths; thin lines are roads. The long rectangles are trucks, the squares are packages, and the other objects are planes. DRIVE-TRUCK is written in PDDL with parameterized pre- and postconditions: question marks annotate formal parameters. The operator is a schema that can be instantiated, using domain objects, giving rise to distinct *actions*.

the effect of artificially restricting the planner. Clearly there is a case for allowing such knowledge to be made available to a planner, but there are other interesting possibilities as well. In recent years, the field of automatic domain analysis has been developing as a subfield of planning (Fox and Long 1998; Fox and Long 1999; Gerevini and Schubert 1996a; Gerevini and Schubert 1996b; Gerevini and Schubert 1998; Kelleher and Cohn 1992; Morris and Feldman 1989; Nebel et al. 1997). This work has produced results that show that domain preprocessing can recover some of the structure that is lost in the representation process, and even identify additional structure not apparent to the domain designer. This information can be presented to the planner in a form that can be readily exploited.

The generic behaviors discussed in Section 4 are obtained by static analyses that depend on the underlying functionality of the TIM system. We refer to this underlying functionality as the *basic analysis* performed by TIM. The main features of the basic analysis are discussed in the following section.

3 The TIM System

A planning domain description is a model of the dynamics of a physical or software world. It comprises a concise representation of the states that objects can occupy and the actions, or operators, that cause them to change state. For planning to be decidable, there must be a finite collection of objects and hence a finite collection of states that they can occupy.

In most domains, objects are partitioned according to the states they can be in and the state transitions they can make. For example, in the logistics domain pictured in Figure 2, trucks and planes can be in states of location, and they make state transitions by driving or flying. Packages can also be in states of location, but they make state transitions by being loaded into, and unloaded from, trucks and planes. Since packages cannot drive or fly and vehicles cannot be loaded into or unloaded from anything, packages form a different partition from trucks and planes. This partitioning reflects functional differences between the package objects and the vehicle objects. We call these partitions *types*.

Given a partitioning of a finite domain according to the states objects can be in and the transforms that affect these states (called *transition rules*), it is possible to express the partitions as small finite state machines (FSMs) or automata. These automata tend to

be small because there is generally quite a high degree of partitioning in any realistic problem domain. They provide a way of focusing attention on the capabilities of the distinct functional types in the domain and they make explicit many invariant properties of these types. For example, if a type is associated with a two-state *finite-state machine* (FSM), it can be concluded that these states are mutually exclusive and that objects of this type must be in one state or the other, but never both, at all times during planning. This is an invariant property of the objects of this type that can be extracted from the FSMs and can provide very useful search control information to a planner (Bacchus and Kabanza 2000; Bacchus and Kabanza 1996; Fox and Long 2000; Gerevini and Schubert 1998; Kautz and Selman 1998; Kvarnstrom and Doherty 2000; Nau et al. 1999).

The TIM system constructs this FSM view of the domain using automatic analysis techniques. It begins by identifying all of the state transitions that can be made by all of the objects in the domain. This is done by considering all of the parameterized actions in the domain description. For each parameter, TIM constructs a collection of transformation rules on the basis of what *properties* objects that can instantiate each parameter exchange on application of the action. A property is a projection of a proposition onto one of its arguments, denoted by the predicate name subscripted with the argument position being considered.[1] The FSMs constructed by TIM are also termed *property spaces*, because the states traversed are collections of properties, rather than of propositions.

For example, consider the *drive-truck* operator in Figure 2. Objects that can instantiate the variable T exchange an at_1 property for another at_1 property. In other words, that object changes location. The locations involved are irrelevant (they will be different for different applications of the action). This exchange is denoted

$$at_1 \rightarrow at_1$$

Similarly, objects that can instantiate the variable Y acquire the property at_2 without giving up anything in exchange. This is denoted

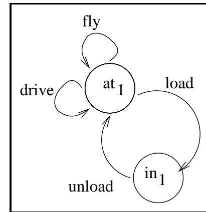
$$null \rightarrow at_2$$

and a rule of this form is called an *attribute-increasing rule*. Sometimes transitions are *enabled* by properties that are not part of the exchange. The object making the transition must have these enabling properties before the transition can be made. In the case of the drive operator, the transition on at_1, just shown, is enabled by $truck_1$. This is denoted

$$truck_1 \Rightarrow at_1 \rightarrow at_1$$

TIM groups together properties by forming closures of the properties on the left and right sides of the transition rules. The construction of these closures we call *uniting*. Each closure forms a disjoint set of properties governing which rules will be associated with those properties in a single property space. TIM then groups together all of the objects that can have any of the properties in each closure and builds an FSM indicating how these objects exchange these properties by means of the associated transitions. Property space construction is done in two stages: First, candidate property spaces are seeded from

[1] Where denoting an abstract property for which the argument position is not known, an unsubscripted identifier will be used.



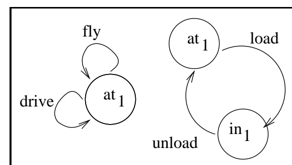


Figure 3 An FSM model of the behavior of vehicles and packages in Logistics, shown before (left) and after (right) splitting.

the objects, their initial states, and the inferred transition rules. Next, these seeds are grown into property spaces by a process called *extension*, in which the full set of states accessible to the objects, by applications of the transition rules, are inferred. Membership of the resulting property spaces is used to identify the partitions (types) to which each object in the domain belongs.

The initial phase of construction of property spaces can obscure the type distinctions that are present in the domain. In the Logistics domain, illustrated in Figure 2, it can be seen that packages can be at locations, just as vehicles can. If the domain description uses the same predicate to describe the location of packages as to describe the location of vehicles, packages and vehicles can appear to share the same behavior with respect to this property space. Figure 3 shows the property space that is constructed under this assignment. Of course, this is undesirable because packages and vehicles behave differently in this space, despite the fact that they share the at_1 property. This is because packages can exchange an at_1 property for an in_1 property, by means of a *load* transition, while vehicles cannot, and vehicles alone can perform $at_1 \rightarrow at_1$ transitions.

When a property space contains behaviors separated by a type distinction, as in Figure 3, the invariants that TIM infers, for the objects in that property space, are too weak to provide effective search control. For example, from the unsplit FSM in Figure 3, TIM can only infer that every object that can traverse this FSM must be either *at* somewhere or *in* something and this, while true, is weak because vehicles cannot have the in_1 property and therefore must always be *at* somewhere. The stronger invariant *is* available when the property space is separated into two—one for vehicles and one for packages. To separate these property spaces, TIM performs *subspace analysis* on the initial property space whenever it contains objects of different types, as in this case. Subspace analysis gives rise to the separated spaces shown on the right in Figure 3. In fact, the single-state property space can be split again, yielding two single-state automata, each with a single looping arc. One of these automata represents the type of truck, the other the type of plane. Trucks are distinct from planes because they can drive, but cannot fly.

As mentioned above, sometimes a property can be acquired without a concomitant exchange. In order to depict the acquisition and loss of such properties in a property space, it is necessary to know how many instances of the property are available in the world. For example, to know how many at_2 properties a location can acquire, it is necessary to know how many objects can instantiate the first argument of the *at* predicate. The size of the property space will be determined by that number, and the only useful invariants obtained

will correspond to cardinality constraints on the number of such properties available in any consistent state during planning. Properties that can be freely acquired and lost are deemed to be *attributes*.

When transition rules on properties and on attributes occur in the same candidate property space, the size of that space is dominated by the number of times the attribute can occur in any state. This makes the subsequent processing of the property space, including the inference of invariants, potentially expensive. We have observed that when attribute rules and transition rules occur in the same candidate property space, it is usually possible to split the property space by subspace analysis. Therefore, TIM does not extend candidate property spaces that contain attribute rules. Such spaces are termed *attribute spaces.* TIM identifies which objects in the domain can acquire the attributes within the space, by application of the attribute rules, but does not further process an attribute space unless subspace analysis is possible and leads to the construction of a new property space. This might lead to the loss of some invariants, in rare cases, but we believe this possible loss is justified by the efficiency gains obtained.

The basic analysis performed by TIM consists of the construction of a collection of property spaces and the inference, during this process, of type partitions and invariant facts. In order to clarify how the basic analysis provides a foundation for the identification of generic structure in a domain, we now begin the presentation of a simple worked example. The Bulldozer domain[2] describes a simple world in which a bulldozer can be driven between locations connected by roads and bridges. Jack can *board* and *disembark* from a bulldozer, and can *drive* along roads or *cross* bridges. The domain operators, written in PDDL, are given in full in Figure 4, and an example of an initial state is illustrated in that figure.

3.1 Analyzing the Bulldozer Domain

Given the domain as described in Figure 4, the basic analysis performed by TIM produces the following attribute spaces. Note that the domain encoding uses the predicate *mobile*. This is not to be confused with the generic type "mobile" discussed in Section 4. The way TIM constructs these rules from the parameterized actions supplied in the domain description are described in detail in (Fox and Long 1998).

Attribute Space 1
Properties:
driving2
Objects:

Rules:
vehicle1,at1 $\Rightarrow$ null $\rightarrow$ driving2
vehicle1,at1 $\Rightarrow$ driving2 $\rightarrow$ null

[2]This is a standard domain originally developed as part of the UCPOP release (Barret et al. 1996).

```
(:action Drive
     :parameters (?thing ?from ?to)
     :precondition (and (road ?from ?to)
                        (at ?thing ?from)
                        (mobile ?thing)
                        (not (= ?from ?to)))
     :effect (and       (at ?thing ?to)
                        (not (at ?thing ?from))))
```

```
(:action Cross
     :parameters (?thing ?from ?to)
     :precondition (and (bridge ?from ?to)
                        (at ?thing ?from)
                        (mobile ?thing)
                        (not (= ?from ?to)))
     :effect (and       (at ?thing ?to)
                        (not (at ?thing ?from))))
```

```
(:action Board
     :parameters (?person ?place ?vehicle)
     :precondition (and (at ?person ?place)
                        (mobile ?person)
                        (person ?person)
                        (vehicle ?vehicle)
                        (at ?vehicle ?place))
     :effect (and       (driving ?person ?vehicle)
                        (mobile ?vehicle)
                        (not (at ?person ?place))
                        (not (mobile ?person))))
```

```
(:action Disembark
     :parameters (?person ?place ?vehicle)
     :precondition (and (person ?person)
                        (vehicle ?vehicle)
                        (driving ?person ?vehicle)
                        (at ?vehicle ?place)
                        (mobile ?vehicle))
     :effect (and       (at ?person ?place)
                        (mobile ?person)
                        (not (driving ?person ?vehicle))
                        (not (mobile ?vehicle))))
```

Figure 4 The Bulldozer domain operators and a simple bulldozer world state.

Attribute Space 2	**Attribute Space 3**
Properties: at1, mobile1, driving1	**Properties:** at2
Objects: Jack, bulldozer	**Objects:** a, e
Rules:	**Rules:**
mobile1 ⇒ at1 → at1 [cross]	road2 ⇒ null → at2
mobile1 ⇒ at1 → at1 [drive]	road1 ⇒ at2 → null
person1 ⇒ at1,mobile1 → driving1	bridge1 ⇒ at2 → null
vehicle1,at1 ⇒ null → mobile1	bridge2 ⇒ null → at2
vehicle1,at1,driving2 ⇒ mobile1 → null	at2 ⇒ at2 → null
person1 ⇒ driving1 → at1,mobile1	at2 ⇒ null → at2

All three spaces are attribute spaces because they each contain at least one attribute rule. In the first space, the associated objects are those that have the at_1, $mobile_1$, or $driving_1$ properties in the initial state. Although the $person_1$ property is an enabling condition for some of the rules, that does not mean that it belongs to this space. Instead, it establishes a dependency between the (in this case, static) property and this attribute space. It is worth observing, at this point, that the first two rules in the space, which are apparently identical, are in fact distinguished by the operator that allows the $at_1 \rightarrow at_1$ transition. For one of the rules, *cross* is the operator, for the other, it is *drive*, as indicated in square brackets adjacent to the two rules. In the second space, the objects are those that have the at_2 property in the initial state. In the third space, the objects are those that start out with the $driving_2$ property, of which there are none, initially. Because all three spaces are attribute spaces, TIM does not perform extension to identify the states through which the associated objects can pass. However, TIM does go on to identify any further objects that belong with these spaces due to the ability to acquire the associated properties by application of the rules.

If a space contains attribute-increasing rules, any object that satisfies the enablers can acquire the property on the right side of the rule and should therefore be added to the space. In the first attribute space there is an increasing rule, $vehicle_1, at_1 \Rightarrow null \rightarrow mobile_1$, but no further objects satisfy the enabler in the initial state, so no further objects are added to the space. In the second attribute space there are two increasing rules: $road_2 \Rightarrow null \rightarrow at_2$ and $bridge_2 \Rightarrow null \rightarrow at_2$, and objects b, c, d, f, and g all satisfy either the $road_2$ or $bridge_2$ properties in the initial state. These objects could all, in principle, acquire the at_2 property, so they are added to the space. Similarly, the third attribute space starts off with an empty object collection, but the increasing rule $vehicle_1,\ at_1 \Rightarrow null \rightarrow driving_2$ reveals that the bulldozer can acquire the $driving_2$ property (because it has the properties at_1 and $vehicle_1$ in the initial state), so it is added to the space. This process yields the following attribute spaces by extension of Spaces 1 and 3 (Space 2 remains unchanged). The numbering of the spaces is intended to show their heritage (so Space $x.y$ is the yth space derived from Space x). The rules in the spaces shown below are the same as those in the spaces they are derived from, so are not repeated here.

Attribute Space 1.1	**Attribute Space 3.1**
Properties:	**Properties:**
driving2	at2
Objects:	**Objects:**
bulldozer	a, e, b, c, d, f, g
Rules:	**Rules:**
...	...

At this stage, the attribute spaces can be analyzed to reveal type distinctions in the domain. TIM can infer that Jack and the bulldozer are of different types because of their patterns of membership of the spaces. The locations form a third, distinct type.

3.2 Subspace Analysis

In order to gain access to domain structure that has been obscured by the attribute rules, TIM now refines the initial analysis to identify *subspaces*. The objective is to separate objects of different types into different spaces, in order to try to isolate the objects that are really affected by the attribute rules in the original space. Considering Attribute Space 2 above, it is easy to see that Jack cannot be affected by the attribute-increasing rule, since he does not satisfy the enablers, so it should be possible to isolate that rule with the bulldozer and gain a more precise picture of the state changes that Jack can make. The process by which this is done relies on the type inference having identified any type distinctions in the domain, because the attribute spaces are now subdivided around these type distinctions.

For example, Attribute Space 2 can be subdivided into two FSMs: one for the type including Jack, the other for the type including the bulldozer. Having split the space, TIM associates, with each subspace, the rules applicable to objects within them. The enablers of the rules are used to determine which subspace they are moved to. Any rule with no enablers appears in all subspaces obtained from the original attribute space. Initially this yields two spaces—a property space and an attribute space.

Property Space 2.1	**Attribute Space 2.2**
Properties:	**Properties:**
at1, mobile1, driving1	at1, mobile1, driving1
Objects:	**Objects:**
Jack	bulldozer
Rules:	**Rules:**
mobile1 $\Rightarrow$ at1 $\rightarrow$ at1	mobile1 $\Rightarrow$ at1 $\rightarrow$ at1
mobile1 $\Rightarrow$ at1 $\rightarrow$ at1	mobile1 $\Rightarrow$ at1 $\rightarrow$ at1
person1 $\Rightarrow$ at1,mobile1 $\rightarrow$ driving1	vehicle1,at1 $\Rightarrow$ null $\rightarrow$ mobile1
person1 $\Rightarrow$ driving1 $\rightarrow$ at1,mobile1	vehicle1,at1,driving2 $\Rightarrow$ mobile1 $\rightarrow$ null

The first of these is a property space, since it contains no attribute rules. For this space, TIM now identifies the legal states of the associated objects, first by finding the properties

of the objects in the initial state, and then by using the extension process described in (Fox and Long 1998). Extension is used to identify any states that can be entered by the associated objects following applications of the rules in the property space, starting with the states inhabited in the initial planning state by the objects in the property space. It computes a reachability analysis from the perspective of the objects in the space. Extension completes the new property space as follows:

Property Space 2.1.1	
Properties:	at1, mobile1, driving1
Objects:	Jack
Rules:	mobile1 ⇒ at1 → at1
	mobile1 ⇒ at1 → at1
	person1 ⇒ at1,mobile1 → driving1
	person1 ⇒ driving1 → at1,mobile1
States:	[at1,mobile1], [driving1]

The example shows that the same properties can appear in more than one subspace. In the original analysis, used for type inference, it is important that no property belongs to more than one FSM. However, subspaces are not used for type inference, only for invariant extraction, and the duplication of properties does not affect this.

Having found a property space, TIM can now infer some invariants about its associated objects. Assuming that Jack has been determined to be of type T_0, the locations are of type T_1 and the bulldozer to be of type T_2, the invariants are reported as follows:

$$\forall x : T_0 \cdot (\exists y : T_1 \cdot (at(x,y) \wedge mobile(x)) \vee \exists z : T_2 \cdot driving(x,z))$$
$$\forall x : T_0 \cdot \forall y \cdot \forall z \cdot (at(x,y) \wedge at(x,z) \rightarrow y = z)$$
$$\forall x : T_0 \cdot \forall y : T_2 \cdot \forall z : T_2 \cdot (driving(x,y) \wedge driving(x,z) \rightarrow y = z)$$
$$\forall x : T_0 \cdot \neg(\exists y \cdot (at(x,y) \wedge mobile(x)) \wedge \exists z \cdot driving(x,z))$$

In (Fox and Long 1998), we prove two important results, which can be restated as follows:

Theorem 1 *Let P be a planning domain, I be an initial state containing an object o and S be a second state, S, reachable by application of actions from P to the state I. Then, in any property space, PS, constructed by* TIM *in analyzing P and I and containing o, the intersection of the properties in PS and the projection of S for o is a state in PS.*

Theorem 2 *All invariants generated from property spaces constructed by* TIM *are valid.*

The first result shows that TIM finds all the legal states that can be visited by objects in the structure of property spaces, while the second demonstrates that TIM is sound.

The processes by which these invariants are obtained might be considered to be somewhat involved given their rather obvious nature. Other researchers have inferred similar invariants by means of alternative preprocessing strategies (Gerevini and Schubert 1998; Gerevini and Schubert 1996b; Kelleher and Cohn 1992; Morris and Feldman 1989). However, an important feature of the TIM approach is that the data structures constructed

during the basic analysis (the transition rules and property spaces) provide the foundations on which the recognition of generic types and behaviors is performed, as we will go on to demonstrate.

4 Generic Types and Behaviors

TIM is designed to construct an analysis of planning domains that is based on *behaviors*. The fundamental basis on which objects are collected into types is that of commonality of behavior. Behaviors are characterized by particular changes that objects can undergo as a consequence of being used in some role in an action. These changes can be changes of state, acquisition of new attributes, or the loss of existing attributes. Furthermore, we capture behaviors in which objects serve as catalysts of change in other objects. Where an object must be available, typically in some specific relation to objects undergoing change, to *enable* that change to take place, the object plays a role as a catalyst.

Although constructing types and associated behaviors within individual domains is fruitful—allowing us, for example, to infer invariants, symmetries, and other features of domains—it is potentially far more powerful to abstract from individual domains and explore common behaviors across multiple domains. In doing so, we find that certain common behaviors can be identified that correspond to intuitively familiar abstractions of well-understood problem structures. These abstractions of behaviors we refer to as *generic behaviors*, with the objects that play roles in the behaviors being of *generic types*. A generic type is therefore not simply a collection of objects that share a behavior within a single domain, but a collection of all possible collections of objects across all domains in which these collections share the same underlying behavior. Our types are hierarchical, so it is possible for a type in a specific domain to exhibit behavior that characterizes it as a particular generic type while, at the same time, exhibiting distinctive additional behaviors—perhaps even behaviors that characterize another generic type. The benefits that are offered from characterizing the behaviors of types with domains in terms of generic behaviors and generic types are the usual consequences of recognizing individuals as specializations of well-understood abstractions: all of the knowledge applying to the abstractions can be applied, in its specialized forms, to the individuals.

TIM identifies basic types in a domain by partitioning objects into equivalence classes according to functionality. Thus, basic types and their behaviors are defined and exist a posteriori. By contrast, generic types and behaviors are formed by abstraction of commonly occurring, and interdependent, functional equivalences across domains. Thus, generic types are defined in more general terms and exist a priori. The definition of a generic behavior is given in terms of a *fingerprint* that characterizes it in an abstract, domain-independent way. A fingerprint can be described in terms of required transition rules, FSM behaviors, predicate arities, relationships between variables appearing in operator schemas, invariants, and other features that can be identified in a domain description. Examples are given in Sections 4.1 and 5. Generic types are defined by the roles they play in generic behaviors—any basic type in a domain that fulfills a role described in a generic behavior can be ascribed the corresponding generic type. The presence of a generic type in a domain must be explicitly sought by identification of an appropriate fingerprint for a defining generic behavior. Any domain that features a fingerprint characterizing a specific generic behavior is deemed to contain that behavior and its associated generic

types. Where a generic behavior depends on the roles of several types, it will determine correspondingly many generic types. It is typical for a generic behavior to determine several generic types, as we will see in the following sections. Moreover, it is common for generic behaviors themselves to form collections that, together, define a common subproblem. We call these collections *clusters*, and these clusters form the foundation for identifying an appropriate exploitation strategy, as discussed in Section 7.

A strong relationship exists between our notion of *fingerprint* and the software-engineering notion of *design pattern* (Gamma et al. 1995). Defining a design pattern in his seminal work, Christopher Alexander states:

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

This generality is exactly what generic types are intended to capture, and the fingerprints define, in a planning-domain pattern language, the necessary components, and relationships between them, that must be identified in the domain for the presence of the generic type to be inferred.

The fingerprint of a generic type and its associated cluster is a characterization in terms of abstract structures that must be identified by pattern matching within the domain structure identified by TIM. These structures are founded on the FSMs generated by TIM, but can include the arities of predicates, the types of relationships defined by predicates, and the linkages between the FSMs, the predicates, and the operator schemas that define the behaviors of the objects in a domain. These structures can be seen as design patterns specific to planning domains, defined by a pattern language for planning domains. The development of such a pattern language is a topic of our future work. The characterizations provided in Figures 5 and 10 have a similar status to the design patterns of programs and can play a similar role: to support the construction of planning domain descriptions using well-understood structural components that capture common behaviors. Seen in this light, the automatic recognition of generic types is an attempt to recognize and capture design patterns in use within a planning domain. The exploitation of generic types as planning domain design patterns is at an early stage, and we perceive there to be opportunities in domain engineering as well as in supporting efficient planning.

In the remainder of this section, we outline the generic types and behaviors associated with transportation clusters. We describe their fingerprints and the processes by which TIM identifies them in domain models.

4.1 The Generic Type of Mobile Objects

The most fundamental of the generic types recognized by TIM is that identified with one of the simplest fingerprints: a single state and an associated transition starting and ending at that state (Figure 5). This signature structure, when it is part of a property space, indicates that objects that belong to the space can make transitions between different associated values that are linked to the objects by the properties forming the state. For example, if the state contains the single property at_1, then the objects in the property space make transitions between values that they are *at*. Provided that the state is denoted by a single property, P_i, derived from the n-ary predicate P, then the values of the vector

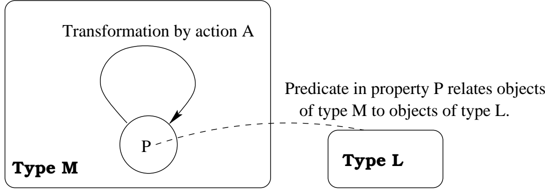


Figure 5 Simple structure characterizing a generic behavior.

x_i, which can occupy the $n-1$ arguments excluding the ith argument of P, define the points between which the objects in the property space can, in principle, make transitions.

Consider the simplest case: Suppose that a property space contains a single state that is defined by a single property, P_i, derived from a binary predicate P, and that a single transition starts and ends at this state. In this case, the space encodes the fact that objects can make transitions between unique single-valued relations with objects, which can satisfy the $(2-i)$th argument position of the predicate P. Execution of the operator from which the transition is derived causes an object in the property space to make a transition between an association with one of the related values to another. This can be seen as movement of the objects between locations. The operator causing the transitions is the "move" operation and the objects are *mobile*. The values with which the mobile objects are associated through P can be seen as *locations* and the predicate P as the *locatedness* condition linking the mobile objects and the locations.

The notion of a mobile object and the network of locations on which it moves is a powerful and common feature of planning problems. It is not surprising that mobile objects occur in problems that involve some sort of logistical element, where agents must be moved between the locations at which activities occur. It is less obvious that the same feature can arise in other domains. In particular, the locatedness condition can be a metaphor for any many-to-one relationship that must always hold between members of one collection of objects (the mobiles) and some subset of the members of a second collection (the locations). This metaphor is valuable, since the problem of navigating between locations in a map is equivalent to the problem of selecting an efficient sequence of values with which an object must be related in order to exchange a relationship with some initial value for a new relationship with some goal value. Thus, the same path-planning problem that solves the problem of efficiently crossing a map of locations will solve the metaphorically equivalent problem of efficiently traversing the range of a many-to-one relation, whatever that relation might actually be intended to represent in the planning problem. It is precisely this commonality of fundamental behavior that identifies these patterns as characteristic of generic types. In this case, the generic types involved are the mobile objects, or *mobiles*, and the *locations* between which they move.

In this basic case, identifying the signature structure and the consequent identification of the generic types and roles they play is relatively straightforward. It is a slightly more complex problem to identify the map that defines the legal transitions between locations available to mobiles. The process by which a map is derived is described in Section 4.3.

```
for each property space, P
     for each rule in P, r
          if r is of the form e ⇒ p → p
               and p corresponds to a binary predicate, pred
          then construct a new mobile collection, M;
               associate r with M;
               put the objects in P into M;
               mark the other argument of pred as a location type;
               record pred as the "at" relation of M;
          endif
     endfor
endfor

for each subspace, S
     for each rule in S, r
          if r is of the form e ⇒ p → p
               and p corresponds to a binary predicate, pred
          then if r is already associated with a mobile collection, M
               then if M is already refined
                    then   contruct a related mobile collection, M';
                           associate M' with M;
                           put the objects in S into M';
                    else   replace the objects in M with objects from S;
                           mark M as refined;
                    endif
               else  construct a new mobile collection, M;
                    put the objects in P into M;
                    mark the other argument of pred as a location type;
                    record pred as the "at" relation of M;
                    mark M as refined;
               endif
          endif
     endfor
endfor
```

Figure 6 Pseudo-code algorithm for detecting mobile objects in a planning domain description.

Figure 6 gives a pseudo-code description of the mobile-identification part of our analysis in property spaces and subspaces. The key observation regarding subspaces is that this analysis refines the analysis of the parent spaces, identifying subsets of mobile objects among a larger collection in the parent space. In situations in which TIM constructs a property space that contains a state transition rule, $p \rightarrow p$, implying the existence of a mobile, the objects in the space are proposed as members of a generic mobile type. However, it can happen that the members of the space cannot all exploit the rule that indicates movement. For example, if there is a collection of packages that can be either *at* a location or *in* a truck, while the trucks can only be *at* a location, the generic structure analysis carried out by TIM will place the *at* and *in* properties in the same property space, due to the $at \rightarrow in$ and $in \rightarrow at$ transition rules for packages, as we saw in Figure 3. This will cause the generic structure analysis to initially consider packages and trucks to both be part of the generic mobile collection, despite the fact that packages cannot exploit the $at \rightarrow at$ rule that makes the trucks mobile. However, as discussed, a more refined analysis performed by TIM will identify the separate types of trucks and packages and place them in separate subspaces. The analysis of these spaces will then allow TIM to determine that it is actually only trucks that are mobile. This more detailed analysis allows a *refinement* of the original mobile generic type and it replaces the collection of

mobile objects in the original generic type. In the case where subspace analysis identifies more than one refinement of the original space containing mobile types, these subtypes will be said to be *related* mobile types, since they will use the same movement operator and network of locations.

It can be seen, in Figure 6, that the analysis looks for binary predicates encoding locatedness. As mentioned earlier, this restriction can be lifted by considering locations as vectors of objects that can fill the nonmobile arguments of a possible locatedness predicate. In fact, it is also possible to restructure a domain automatically to convert higher arity predicates that encode locatedness conditions into binary predicates. We discuss this further in Section 6. A further special case that can arise is that in which a domain contains only one mobile object. In this case, it is not uncommon for the domain engineer to leave the object implicit—essentially encoding the object directly in the predicate rather than as a named object of the domain. TIM can identify implicit mobiles and successfully extract them from the domain.

Returning to the analysis of the Bulldozer domain, it can be observed, from Property Space 2.1.1, that Jack is a mobile object because of the presence of the two $at_1 \rightarrow at_1$ rules. In fact, Jack is mobile by two different actions (*cross* and *drive*). At first glance, it seems strange that Jack can achieve mobility by these actions independently of the bulldozer. The reason is that Jack uses the same *at*-relation as the bulldozer, and has the property of being *mobile* in the initial state, so the two rules are available to both Jack and the bulldozer during subspace analysis. Indeed, Jack can satisfy the preconditions of the *cross* and *drive* actions, because of using the same *at*-relation, even though this may not have been intended by the domain designer.[3] Indeed, information of this kind, presented in an appropriate way, can be helpful in debugging domain descriptions as they are constructed.

We would now like to be able to infer invariants for the bulldozer, but it is still hidden in an attribute space (Attribute Space 2.2). The basic analysis is unable to identify the missing invariants and does not yet have access to the mobility of the bulldozer. This leads to the need for a further subspace analysis step. We cannot split further on the basis of the number of types inhabiting the space, as there is only one in Attribute Space 2.2. Instead, we observe that the initial subspace analysis can lead to some subspaces containing properties that are not linked by the behaviors described by the rules in those subspaces. This can occur when the original space contained rules that linked these properties, but those rules apply to only a subset of the types represented in the space.

For example, considering Attribute Space 1.2 for bulldozer, we see that the two $at_1 \rightarrow at_1$ rules would appear in a different space from the two attribute rules if we were to perform the uniting step at this stage. We therefore *reunite* the rules in the new space, yielding as many new subspaces as closures constructed by this process.

4.2 Refined and Generic Type Level Analysis of the Bulldozer Domain

In order to identify the hidden structure described above, TIM uses attribute spaces formed during subspace analysis to seed a collection of new FSMs. In each such attribute space TIM performs the uniting process on the rules. Any new closures that are produced result in

[3]Note that we have used the standard Bulldozer domain encoding available as part of the Blackbox release (Kautz and Selman 1995).

the construction of a new attribute or property space. This involves adding the appropriate rules, initial states, and objects to each of the new subspaces.

The following new property spaces are constructed by applying this process to the bulldozer attribute space (Attribute Space 2.2) built during subspace analysis.

Property Space 2.2.1	**Attribute Space 1.2.2**
Properties:	**Properties:**
at1	mobile1
Objects:	**Objects:**
bulldozer	bulldozer
Rules:	**Rules:**
mobile1 ⇒ at1 → at1	vehicle1,at1 ⇒ null → mobile1
mobile1 ⇒ at1 → at1	vehicle1,at1,driving2 ⇒ mobile1 → null

We note that the first of these two spaces contains the two, apparently identical, $at_1 \rightarrow at_1$ transitions, and recall that these transitions are actually distinguished by the actions that cause them (*cross* and *drive*). Both new spaces inherit all of the objects of the original attribute space. Now we find the initial state properties relevant to each property space. This yields

Property Space 2.2.1.1	
Properties:	at1
Objects:	bulldozer
Rules:	mobile1 ⇒ at1 → at1
	mobile1 ⇒ at1 → at1
States:	[at1]

No additional states are accessible to the bulldozer via application of any of the rules in the property space. The process has successfully separated the attribute-valued properties of the bulldozer from its state-valued properties, allowing TIM access to the bulldozer invariants:

$$\forall x : T_2 \cdot \forall y \cdot \forall z \cdot (at(x,y) \wedge at(x,z) \rightarrow y = z)$$
$$\forall x : T_2 \cdot \exists y : T_1 \cdot at(x,y)$$

which tells us that no bulldozer can be in more than one place at a time and every bulldozer must be at some location at every point in time. Again, these facts are quite straightforward, but the property space that has been constructed during the analysis is critical for the identification and exploitation of the generic structure of the domain.

TIM is able to access the fact that the bulldozer is mobile by the *cross* and *drive* operators, so the current representation of mobility is extended to record the fact that the bulldozer is mobile. Because the mobility of the bulldozer is also given by the *cross* and *drive* actions, TIM records the mobility of the bulldozer as *related* to that of Jack. The next stage in the analysis of the Bulldozer domain is to determine the maps of locations that can be traversed by the mobile objects.

4.3 Inferring Maps for Mobiles

In order for mobile objects to move, there must be a network of locations through which they move. The definition of this network is found by identifying the parameters in the appropriate move action that correspond to the source and destination of the move. These are taken to be the nonmobile argument of the locatedness predicate affected by the move operator. The occurrence of the locatedness predicate in the preconditions of the move gives the source, and the occurrence in the effects of the move gives the destination. For example, in the Logistics domain, the action of driving from *city1-1* to *city1-2* specifies that the truck be at *city1-1*, and the effect specifies that the truck be at *city1-2* (see Figure 2). TIM is able to identify the nonmobile argument using the locatedness property of the mobile type, together with the assumption that the locatedness predicate is binary (possibly following transformation, as described in Section 6).

The move action might impose additional constraints on the circumstances under which a mobile might move between its source and destination. In the drive action described above, it is specified that *city1-1* and *city1-2* must be linked. Since links between cities in Logistics are static, this can be determined by examination of the initial state. The map that determines how the truck mobile type can move therefore connects two vertices only if they are linked in the domain description. In richer domains other constraints might be imposed—for example, it might be possible to travel to a specific location only with a pass, or visa. This would further constrain the connectedness of the network.

The preconditions that determine the constraints on access between the source and destination can be isolated and together form a predicate on the two parameters, with any additional parameters that are referred to in the appropriate precondition propositions being existentially quantified. This predicate is abstracted from the move operator and used to define the edge relationship between the location objects that form the vertices of the network on which the mobiles move. The process by which the predicate is abstracted from the preconditions of the movement action, and a network constructed governing the mobility of the corresponding mobile type, is described in detail in (Long and Fox 2000). For the Logistics DRIVE-TRUCK operator the edge predicate is

$$edge(from, to) \equiv \exists city \cdot in\text{-}city(city, to) \wedge in\text{-}city(city, from)$$

which demonstrates that locations in the same city will be linked by an edge in the inferred map. If the edge relationship includes dynamic propositions, then the map network, or map, is itself dynamic—perhaps having doors that can be used to grant or deny access to certain locations, or paths that close at certain points in the plan. The definition of the map structure can be used to plan routes for mobiles to move between locations. Where the map is static, this is a straightforward shortest-path problem, but when the map is dynamic, there is a more complex relationship between efficient paths and the achievement of access along the necessary edges of the paths.

There are many domains in which a mobile is able to use multiple forms of movement (referred to in Figure 13 as multimode movement). For example, amphibious vehicles can both drive and float. The presence of multiple-move actions indicates that in order to identify the most efficient route between two locations, it is necessary to combine the maps, produced by analysis of the move actions, into one (see Figure 7).

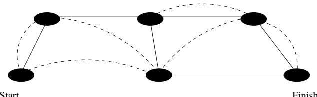
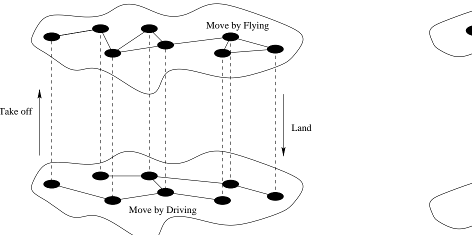


Figure 7 Moving on two maps: an example in which the route between two vertices is shorter on the combined map than on either map individually.



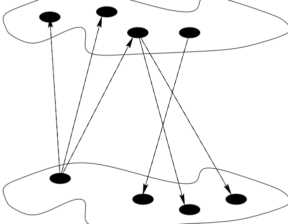


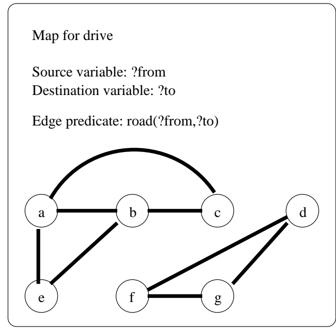
Figure 8 Moving between maps (left) and with multiple transit opportunities (right).

In the case where an object has more than one move action, as in the Bulldozer domain, the process is made more complex by the fact that the set of locations associated with each move action can appear disjoint even though all locations, in all of the sets, are accessible to the mobile object (because it can move by all of the move actions). In the Bulldozer domain, we see that the *drive* move action yields a map formed from all of the locations connected by the *road* map-link, while the *cross* mover yields a map formed from the locations connected by the *bridge* link. Since Jack and the bulldozer can use both of the move actions, they have access to all locations on both maps.

These two maps appear to be disjointed even though the bulldozer can actually travel between any two locations because it can traverse both road and bridge connections. No transit actions are required to gain access between the two maps because the locations in the two maps are shared. The situation can become more complex when mobiles can move between different maps, as discussed in Section 6. This situation is depicted in Figure 8. When multiple maps can be combined without *transit links* (special actions causing transition from one map to another), we call the mobile a *commuting* mobile.

5 Mobile-Related Generic Types: Portables, Passengers, and Drivers

Planning domains in which some collection of objects must be transported between locations are common, and, of course, such domains must include mobile objects to carry out the transportation tasks. The objects that are transported represent a further generic type—*portable* objects. We make the assumption that any carrier for portable objects must be mobile, so an object that is moved through space by the trajectory of a fixed robot arm would not currently be considered an example of the portable generic type. Objects can only be identified as portable if they participate in a specific kind of relationship with



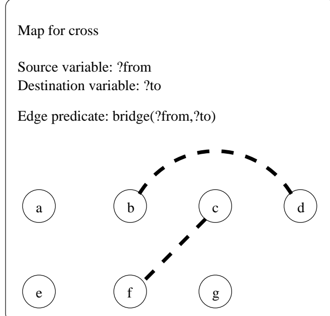


Figure 9 The two maps generated for a Bulldozer domain instance.

one or more mobile objects. Therefore, the presence of mobile objects must be inferred before TIM begins to look for portables.

Our algorithm for inferring the existence of portable objects begins by looking for FSM structures that indicate that any transition in the location of associated objects must pass through an intermediate state linking the potentially portable objects with a self-propelled (mobile) object. In Logistics, packages would be detected as portable because changes in their at_1 properties require them to pass through an *in-relation* with a mobile, indicating their transportation by a third party. Thus, the FSM for portable objects will always relate a *locatedness predicate,* used to describe the locatedness of portables that are not being carried, to a *portedness predicate*, used to describe the state of portables that are being carried. The locatedness predicate must place portable objects on locations within the map of some mobile that acts as carrier for that portable. The carrier will be the mobile referred to by the portedness predicate identified in the partner state in the FSM for the portable. Figure 10 shows the relevant FSM structure, inferred by TIM from the Logistics operator schemas, together with the relationships that must hold in order for the portable objects to be identified. Figure 11 shows the Logistics operator schemas that gave rise to this FSM: observe that the fact that the same variable is used to refer to the location at which the portable object and the carrier must be located for a load operation to be performed, and similarly for the location of the mobile for an unload operation and the final location of an unloaded portable.

It is important that the portable objects share the same location set as the carriers, since otherwise the movement of the supposed carrier will not be relevant to the location of the supposed portable. However, the portable objects might share the same location set indirectly. For example, if the locations that a carrier can visit all have pallets associated with them and some collection of portable objects moves from one pallet to another, carried by these carriers, then the portables will be linked to the locations used by the carrier, but indirectly through the relationship linking the pallets with the locations. An interesting variation on portable objects is the possibility for objects to be loaded into other objects that are themselves portable, such as loading fruit into boxes to be transported. We consider these *indirect* portable objects to be a specialization of portable objects.

The following components form the fingerprint for portability:

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

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

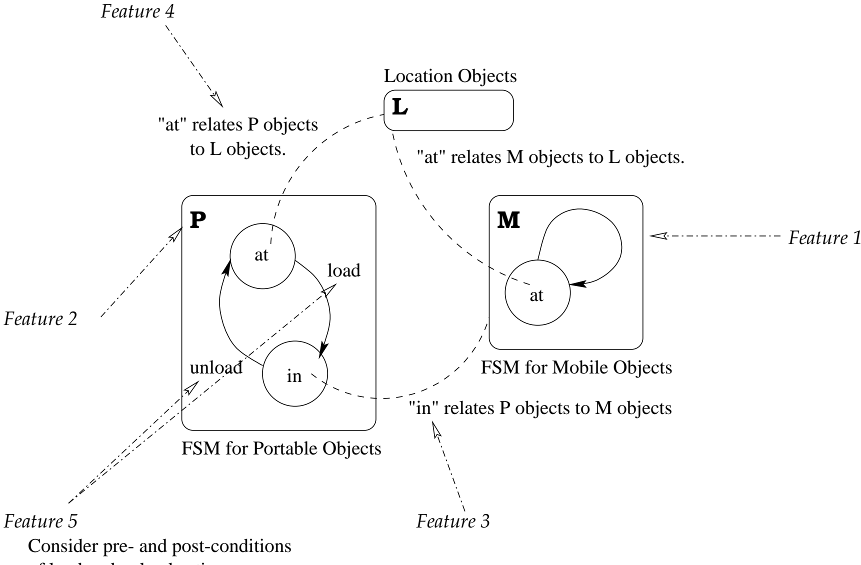


Figure 10 The features that identify portables and their associated behaviors.

```
(:action LOAD-TRUCK
    :parameters (?T ?P ?X)
    :precondition (and (truck ?T)
                       (at ?T ?X)
                       (package ?P)
                       (at ?P ?X))
    :effect (and (in ?P ?T)
                 (not (at ?P ?X))))
```

```
(:action UNLOAD-TRUCK
    :parameters (?T ?P ?X)
    :precondition (and (truck ?T)
                       (at ?T ?X)
                       (package ?P)
                       (in ?P ?T))
    :effect (and (at ?P ?X)
                 (not (in ?P ?T))))
```

Figure 11 Logistics operators revealing presence of portables.

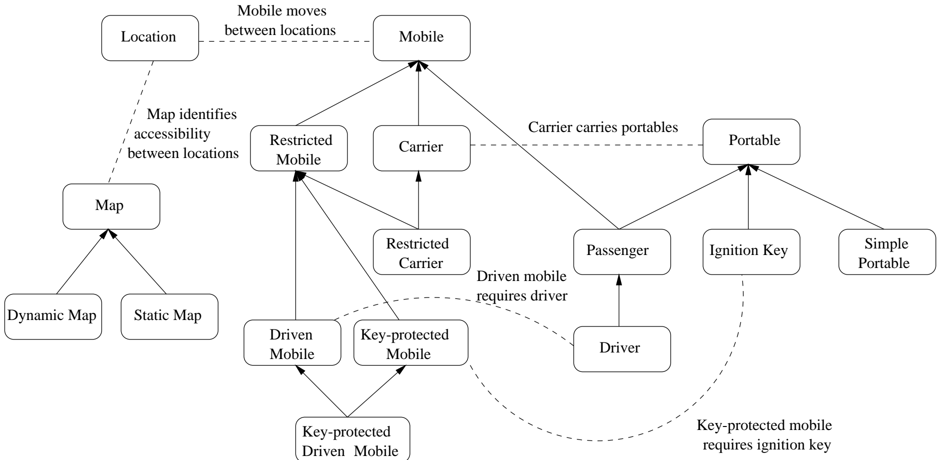


Figure 12 A hierarchy of generic types inferred by TIM.

The first task in finding portable objects is to identify the portedness predicate and establish the producing schemas for the $at \rightarrow in$ and $in \rightarrow at$ transitions. The producer for the $at \rightarrow in$ transition is inferred to be the *load schema* and the producer for the $in \rightarrow at$ transition is inferred to be the *unload schema*. The in-relation appears on the add-list of the *load* schema and on the delete-list of the *unload*, and it has both a portable argument and a carrier argument. We assume that portables are always explicit arguments, but our analysis can deal with the case where the carrier is implicit. Domain objects are made implicit by encoding them in predicates with corresponding dedicated operators. This is only feasible if there are few such objects, and while there is often only one mobile in a benchmark domain, there are generally many portables. Indirectly portable objects can be inferred using a similar process, but looking for a portedness predicate that links the indirectly portable objects with a portable object instead of with a mobile object. This indirection can, in principle, exist to multiple levels, although this is obviously rather rare. In the Bulldozer domain, Jack is a portable type, evidenced by Property Space 2.1 (Section 3) and the Board and Disembark operators, linking Jack to the carrier bulldozer.

A hierarchy of mobile-related types is recognized by TIM, as indicated in Figure 12. The figure illustrates some of the relationships among three hierarchies of generic types: location types, mobile types, and portable types. The figure also indicates a fourth structure, the map, formed from the relationships between locations. It can be seen that within the hierarchies of individual generic types, there can be rich and varied subtypes. For example, the mobile-type hierarchy includes carriers, which can carry portable objects. It also includes restricted mobile types, which are those that can only move when certain additional conditions are met, described in the preconditions of their move actions. Among the subtypes of restricted mobiles are those that require *drivers* and those that require *ignition keys*. The latter is a metaphor for any behavior in which a mobile is prevented from moving unless an object in a particular type (perhaps a type with only one object in it) is in the mobile. It can be seen that behaviors can be multiply inherited, so that, for

example, a mobile can be both restricted and a carrier. The hierarchies for portable and mobile objects intersect in the generic type of *passengers.* This is the collection of types that are both mobile and portable, meaning that they can be carried between locations or move without a carrier. The maps on which they move in these two different contexts can be different, although the locations must be shared, just as with portable objects and their carriers. Jack exhibits this behavior in the Bulldozer domain. If a type is portable and mobile on two different collections of locations, then this implies a different behavior, since it must then be possible for the objects to be at a location in each of the two collections simultaneously.

A refinement of passengers appears when an object plays a special role in the mobility of its carrier. Jack has this property in the Bulldozer domain, causing the bulldozer to acquire the $mobile_1$ attribute when he climbs into it and to lose it again as he disembarks. We call this subclass of passengers, on which their carriers depend for mobility, *drivers.* Similarly, *ignition keys* are immobile portable objects on which mobile objects depend for mobility. The relationships that allow TIM to recognize these additional generic types are present in the preconditions of the move operator for mobiles (e.g., a move operator might specify that some third party be in place in order for the mobile to move) and are recognized in the effects of the operators responsible for allowing drivers to board and disembark from the mobiles.

In the case of drivers, recognition is based on identification of an enabling condition for the movement of the (driven) mobile that is achieved by application of a load operator. That is, if the driven mobile requires an enabling condition in order to move that is only achieved by the load operator that boards the driver into the mobile and is subsequently undone by the disembarkation of the driver from the mobile, then it is clear that the driver must be on board the driven mobile for it to actually move. The driver object itself might not be referred to directly in the move operator for the driven mobile—if the loading of the driver causes a state change in the driven mobile that enables the movement, then this will be a sufficient indication of the dependency between the driver and the driven mobile. A similar enabling condition on the movement of mobile objects involving the loading of a portable, but otherwise immobile, object indicates the need for an object of the generic type we have called ignition keys. In the Bulldozer domain, the fact that the $at_1 \rightarrow at_1$ transitions for the bulldozer each has an enabler, $mobile_1$, indicates that the ability of the bulldozer to move between locations depends upon it achieving mobility through the state transitions made by some other object. Mobility is achieved when Jack is in the bulldozer—the bulldozer cannot move without a driver. TIM can detect that Jack is of the driver generic type because Jack is both mobile and portable, and it is the *loading* of Jack (by means of the *board* action) into the bulldozer that achieves the enabler for the bulldozer's mobility rule. The additional feature that characterizes drivers and ignition keys and is not shown in Figure 10 is the enabling condition on the movement transition for M. This condition can only be achieved by the load action for P, which loads a P-type object into an M-type object. It is possible for a mobile to have multiple enabling conditions, implying the need for multiple drivers, ignition keys, or some combination of these objects.

The hierarchy in Figure 12 distinguishes static and dynamic maps: *Static maps* are those in which the accessibility relationships between locations, recorded in the accessibility predicate extracted from the preconditions of the appropriate movement operator, are

entirely determined by the initial state. *Dynamic maps,* in contrast, are affected by actions, as indicated in Section 4.3. The figure also includes a subtype of portable objects that we call *simple portable objects.* This type includes those types of portables that play no role in a domain other than to be carried between locations. Objects of this kind are relatively common in benchmark planning domains (e.g., packages in the Logistics domain, balls in the Gripper domain (McDermott 1998), cars in the Ferry domain (McDermott 1998) and so on). It is worthwhile identifying this generic type since a useful heuristic can be applied to these objects: there is no point in putting these objects down in their starting locations or picking them up from their goal destinations. A further brief discussion of heuristics of this kind is included in Section 7.

Other dependencies between mobile types can exist, of course. For example, if a mobile can only move when an object, mobile on a different collection of locations, is present at an appropriate location, then, although we do not identify this as a driver-driven relationship, it clearly has much in common with it. This situation can arise when the enabling objects move directly between the mobiles they enable, so that the enabled mobiles are themselves locations for the movement of these enabling mobiles. We call this behavior *reallocation* of the enabling mobile between the dependent mobiles, the dependent mobiles being called *tasks* and the enabling mobiles *processors* (Long and Fox 2001).

It should be emphasized that although the names we use for the generic types are strongly suggestive of the roles they play, these roles might actually be metaphors for behaviors with quite different interpretations in the mind of the domain engineer. For example, if a domain allows walls to be painted with different-colored paints, perhaps specifying which paints can cover which other paints, then this can be interpreted, by TIM, as a situation in which the walls are mobile on a map of color locations. If there is a requirement for a decorator to be by the wall in order to paint it, and the decorators can move from wall to wall, then this is an example of the reallocation behavior of processors (decorators) between tasks (walls).

Generic types in the hierarchy of generic mobile and mobile-related types can occur in various combinations in a planning domain. These combinations form clusters (referred to in the introduction to this section), such as simple transportation clusters (carriers and simple portables moving on static maps), more complex logistical transportation clusters (driven carriers, drivers, and portables), driver scheduling clusters (driven mobiles and drivers), and so on. The cluster that involves the processors and tasks described above is called the *Multi-Processor Scheduling (MPS) cluster.* Strategies for exploitation of these clusters can apply to subsets of clusters or just to single clusters, as we will discuss in Section 7.

6 Generalizing Mobiles

The structure characterizing mobiles and their behavior can be generalized in several ways. These are summarized in Figure 13. In this figure, the entry in each row referring to n corresponds to the dimension of generalization. As discussed below, the fourth generalization can only occur in the context of the third generalization, so must contain more than one state. Otherwise the generalizations are broadly orthogonal to one another, and can appear in any combination in a domain.

Type of mobile	Arity	# S	# P	# M	FSM
Basic mobile	2	1	1	1	at
Generalization 1: Mobile on $n-1$ dimensions	$n > 2$	1	1	1	P
Generalization 2: Multi-mode moves	2	1	1	$n > 1$	at
Generalization 3: Flying and hovering mobiles	2	$n > 1$	1	1	at
Generalization 4: Complex locatedness	2	> 1	$n > 1$	1	at p

Figure 13 Table of generalizations of mobile structures. Arity is the number of arguments in the locatedness predicate. # S refers to the number of states in the state space containing the locatedness property; # P is the number of properties in the locatedness state; and # M is the number of movement actions.

In the first generalization, the predicate-defining locatedness can have more than two arguments. If the locatedness predicate is n-ary, then the mobiles move in space defined by $n-1$ coordinates. This does not imply that when n is 2 the mobiles move in 1D space—the connectedness of the locations can define the points of a (discretized) multidimensional space—but that the space is homogenous. Where n is greater than 2, the space in which the mobiles move is composed of the product of $n-1$ different homogenous spaces. These might represent, say, space, time, fuel availability, attitude, and so on. For example, a domain might be described using the predicate *configuration*, so that *configuration*(x, y, z) should be interpreted to mean that the object x is at location y, with current fuel-level z. In this case we might expect that movement would change the location and fuel-level values simultaneously. It is a straightforward transformation to convert higher-arity predicates into binary predicates, so this generalization can easily be converted into a standard mobile form and we can assume that all predicates are unary or binary for the purposes of the rest of this discussion. The transformation can be carried out in different ways, but the most effective converts mobility in the form of this first generalization into simultaneous mobility on a collection of maps (one for each dimension as indicated above).

The second possible generalization allows multiple transitions starting and ending at the locatedness state. This case arises when there are multiple ways that objects can move. These might correspond to different modes of movement. This case complicates the structure of the maps on which mobiles move, since they can move in different ways across the same map. The inference of maps and the composition of maps that arise from different movement modes is discussed in Section 4.3.

A third generalization is that the property space might include more than one state, with transitions linking the locatedness state to alternative states. This situation is more complex because it implies that the mobile objects can leave their locations and enter some different state. There are various interpretations of this behavior. It might be that the mobiles can be temporarily suspended from movement in order to play a role in some task so that, on conclusion of the task, the suspended mobile can return to movement from the location at which it was suspended. We call such mobiles *hovering mobiles*, since the mobile can be seen as hovering above its location before returning to it. A different possibility is that the mobiles can leave their original location and then make various transitions before returning to the first network at a different location to the one from which they left. Objects that exhibit an additional movement behavior in one or more of the states that they pass through in these transitions are called *flying mobiles* to reflect the idea that they can move on different networks ("flight paths" and "roads") using actions appropriate to each separate network ("flying" and "driving") and moving between the networks ("taking off" and "landing"—these are transit links referred to earlier in Section 4.3). These cases can all be identified by TIM and handled appropriately.

A final generalization is that the locatedness state might be characterized by more than one property. In fact, this situation cannot arise in the property spaces constructed by TIM unless there is at least one additional state in the property space, so that this generalization can only occur in the context of the third generalization described above. The reason for this is that TIM always *splits* properties that are common to the left and right sides of a rule from the rest of the rule, creating separate rules for the individual properties that appear on both sides of the original rule. Therefore, the only way that two properties can both appear in a single state with a rule of the appropriate form for movement is if the properties are forced into the same property space because of a transition to another state by another rule. The division of rules that refer to the same property on the left and right sides causes TIM to identify this generalization as a version of mobility without difficulty. In fact, the Bulldozer domain exhibits this generalization for Jack—the locatedness state for Jack contains both at_1 and $mobile_1$, although the transition rule responsible for moving Jack between locations is simply $mobile_1 \Rightarrow at_1 \rightarrow at_1$, indicating that the property $mobile_1$ is not affected by movement (either crossing or driving).

To summarize: Generalizations of the arity of the locatedness predicate and of the number of properties representing locatedness can both be transformed into the standard mobility case, but the other cases (Generalizations 2 and 3 in Figure 13) cannot be eliminated. These two generalizations can, of course, also appear together so that a *flying* mobile has multiple movement modes. However, TIM can recognize and manage these generalizations—the first form as more complex map structures and the second as various forms of generalized mobile types.

A final situation that should be noted arises when what might be considered as mobility of objects is encoded using a finer-grained encoding than that considered here. For example, consider the operators in Figure 14. In this case, the mobile objects make transitions between locations via intermediate states (a plane will be *at* a location, then take off to be *en route* on a flight path, then fly to arrive at the end of the path, where it will be *circling*, and finally land to be *at* its destination). To the domain designer it might appear obvious that the *at* predicate encodes locatedness, but syntactically the three states of the aircraft are equivalent. Our analysis will not currently identify this situation

```
(:action TAKE-OFF
        :parameters (?P ?X ?F)
        :precondition (and (pathFrom ?X ?F)
                           (at ?P ?X))
        :effect (and (enroute ?P ?F)
                     (not (at ?P ?X))))

(:action FLY
        :parameters (?P ?F ?Y)
        :precondition (and (pathTo ?F ?Y)
                           (enroute ?P ?F))
        :effect (and (circling ?P ?Y)
                     (not (enroute ?P ?F))))

(:action LAND
        :parameters (?P ?Y)
        :precondition (circling ?P ?Y)
        :effect (and (at ?P ?Y)
                     (not (circling ?P ?Y))))
```

Figure 14 Finer-grained encoding of movement.

as one encoding mobility. This observation raises an important point about generic types and behaviors: the use of terms such as "mobile" is, intentionally, strongly suggestive of an intuitive interpretation of the structures in a domain. However, it should not be interpreted as a suggestion that the analysis will recognize all behaviors that a domain engineer might interpret as examples of a particular generic behavior. Instead, the analysis identifies a precisely characterized set of behaviors. The precision of the characterization is the strength of the approach, since it offers a guarantee, during exploitation, that the behaviors being exploited are precisely those expected.

7 Architecture

The recognition of generic types and behaviors in a domain provides a way in which the performance of a planner can be improved, both in terms of the time it takes to solve problems in that domain, and the quality of the solutions found. The decomposition of a domain around its generic type structure enables the exploitation of specialized technology for solution of the subproblems that tend to be associated with the different generic types. We have observed that route-planning subproblems tend to arise when mobiles occur in a domain, including in cases where they co-occur with portables and other types in the transportation clusters. When driven mobiles arise, the problem of driver allocation complicates the route-planning problem. We have observed more general resource-allocation subproblems arising where types in the MPS cluster (see Section 5) can be detected. These problems, while interdependent with the planning context in which they arise, can be subjected to specialized treatment in a way that informs the search behavior of the planner.

As described in Section 1, we have developed a hybrid architecture based on the integration of planning with a selection of specialized techniques for addressing commonly occurring subproblems. This architecture, which comprises STAN5, uses TIM to identify the generic types associated with transportation (the mobile and mobile-related types) and with a range of other generic behaviors (Fox and Long 2001b; Long and Fox 2001; Long et al. 2000). Recognition of these types triggers the selection of appropriate subsolving technology from a library of tools that can be integrated with the domain-independent planning component of STAN5.

Figure 1, in Section 1, shows the integrated architecture of STAN5. The details of this integration, in terms of the relationships between the components of the architecture and

the operation of the constraint layer, are described in depth in (Fox and Long 2001b). This chapter does not address these issues, focusing instead on the techniques TIM employs to discover the presence of subproblems in a planning domain.

The key component of the integration is the TIM system, which analyzes a domain description to determine what generic types and behaviors are present. Having identified the presence of one or more generic structures, appropriate subsolvers are identified for the corresponding subproblems. These are then integrated with a domain-independent planner through a uniform interface, called a *contraint agenda*, allowing constraints to be passed between the planner and the subsolvers, and between the subsolvers themselves. This interface allows the subsolvers to influence the development of a plan while themselves being constrained by planned commitments. The way this integration is achieved in STAN4 is discussed in (Fox and Long 2001a)—its extension to STAN5 is discussed in (Fox and Long 2001b). The development of the interface between subsolvers and the generic planning engine, and the way it supports communication between the components of the integrated system, is an important current focus of our work.

In addition to exploitation of specialized subsolvers, we have also identified specific heuristics that can be applied to particular generic types to improve planning efficiency. In Section 5 we mentioned the fact that when a domain contains only *simple portables* it is possible to prune parts of the search space that will necessarily lead to long plans. Because this pruning heuristic is not safe to use in all transportation domains, it has to be recruited for use explicitly, as a consequence of the appropriate structures having been identified in the domain. Another heuristic is to eliminate ground action instances that refer to unreachable map locations. It is possible to observe a commonality between such heuristics and control rules of the sort exploited by other researchers (Bacchus and Kabanza 2000; Kautz and Selman 1998; Kvarnstrom and Doherty 2000; Nau et al. 1999) and we are exploring the possibility of automatic instantiation of general rules using the inferred structure of generic-type clusters.

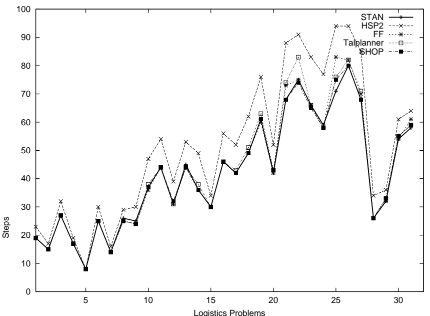
An interesting feature of STAN5 is its fail-safe behavior. If a domain features none of the known clusters, so that no specialized techniques can be recruited, the planner can be invoked to try generic search in the usual way. STAN5 is therefore not limited to solving problems in domains that fit the generic-type fingerprints we have so far defined. Where these can be recognized, STAN5 can benefit from the opportunity to exploit specialized approaches. In other cases, its performance is broadly comparable with that of other uninformed domain-independent strategies.

8 Results

In order to demonstrate the utility of our approach, we present data generated using two transportation domains featuring different transportation clusters. These domains are

1. *The Logistics benchmark domain.* This domain features a simple transportation cluster and can be handled using the STAN4 system (see below). We present a subset of the AIPS-2000 competition[4] data sets compiled by Fahiem Bacchus.

[4]The 2nd International Planning Competition held in Breckenridge, Colorado, during the 5th International Conference on AI Planning and Scheduling.



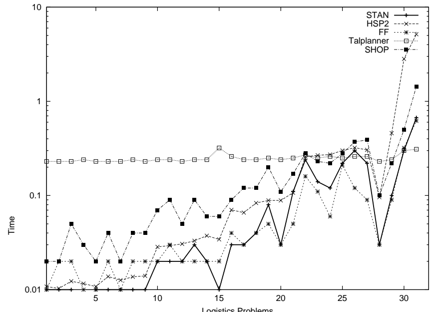


Figure 15 Quality of plans for, and time consumed to solve, Problems 0–30 in the first Logistics problem set. FF and STAN are fastest—TALplanner is slower than the fully automated planners on this problem set, and immune to variations in the problems. It overtakes the other planners at around Problem 30. Note that the thick line highlights STAN and that the graphs are log-scaled.

2. *The Bulldozer domain with multiple drivers.* The transportation cluster in this domain contains drivers and driven *commuting* mobiles. The use of this domain demonstrates the generality of the notion of mobility that TIM can identify, and that can be efficiently handled within STAN5, while completing the running example used in this chapter.

The data illustrates performance of STAN4 in the Logistics domain (because we present historical data, as mentioned above) and STAN5 in the Bulldozer domain. STAN4 (Fox and Long 2001a) is an early version of STAN5 able to cope with only the simplest transportation cluster containing mobiles, portables, and locations on simple maps containing no subparts or transit links. STAN5 extends STAN4 to handle more complex clusters. Nontrivial instances of the Bulldozer domain, which features an example of a more complex cluster, are beyond the capabilities of brute-force planning strategies. (We discuss this issue further in relation to Figure 17.) In the remainder of our experiments, we compare STAN5 with FF because FF is a planning system that is very similar to the general planning core of STAN5. The comparison therefore allows us to judge the extent to which the exploitation of the generic structure analysis presented in this chapter contributes to the power of the planner and the efficiency of the plans it generates. FF is one of the most successful of the current generation of domain-independent planners and, indeed, for the collection of bulldozer problems we consider, no Graphplan-based planning technology or SAT-planning-based technology is able to solve even the smallest of the problems.

The graphs in Figures 15 and 16 show how STAN4 performed in comparison with a diverse range of the best-performing planners in the AIPS-2000 competition, on problems from the Logistics data set. The planners used for comparison are FF (Hoffmann and Nebel 2000), HSP-2 (Bonet and Geffner 1997), TALplanner (Kvarnstrom and Doherty 2000), and SHOP (Nau et al. 1999). The Logistics domain was presented in two sets of problems. The problems increased in difficulty and the second set comprised larger (and hence harder) problems than the first.

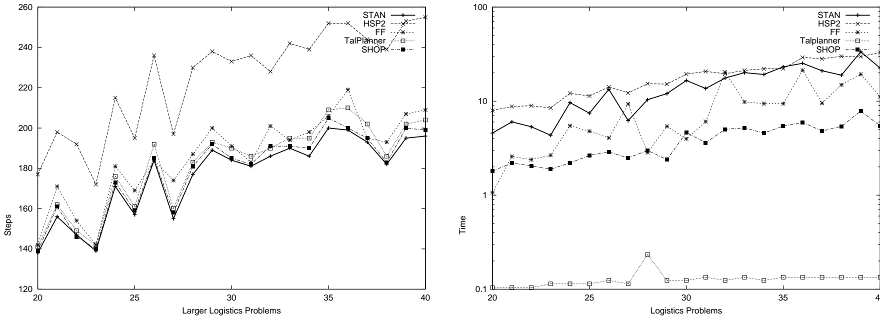


Figure 16 Quality of plans for, and time consumed to solve, Problems 20–40 in the second Logistics problem set. The data compares three fully automated planners, STAN4, HSP-2, and FF, with two hand-tailored planners. STAN4 is producing the best-quality plans because of its improved heuristic estimate.

The competition comprised a fully automated track and a hand-coded track in which planners were allowed to use hand-tailored domain knowledge. In the results presented here, STAN4, FF, and HSP-2 are all fully automated, while TALplanner and SHOP use hand-coded control knowledge. Despite the advantage of being supplied with hand-coded control knowledge, these planners did not consistently outperform the fully automated planners. For example, STAN4 and FF were both faster than TALplanner and SHOP on the first Logistics data set. More significantly, from the point of view of the exploitation of generic types, STAN4 produced the best-quality plans. The advantage STAN4 obtains in plan quality is particularly noticeable in the second Logistics data set (Figure 16), where plans are sufficiently complex for inefficiencies in the solution strategies to become clear. These results highlight the considerable benefits that can be gained by applying specialized technology to solving particular subproblems—including subproblems as apparently simple as the route planning in the Logistics domain—even compared with the exploitation of hand-coded control heuristics guiding general search procedures.

The second data collection (Figure 17) completes the running Bulldozer example, showing a data set for a randomly generated set of Bulldozer problems.[5] These problems include multiple bulldozers and drivers. One hundred problems were generated, with between 5 and 10 drivers, 10 to 40 bulldozers, and 10 to 50 locations. FF[6] failed to solve the 67 largest problems. On the 33 problems solved by both, the relative plan qualities are shown on the left in Figure 17, while on the right can be seen a plot showing how long it took each planner to produce plans of increasing length. As can be seen, STAN5 solved all the problems in the set in under 150 milliseconds, producing plans up to well over 1000 steps. IPP (Koehler et al. 1997), STAN in its Graphplan form, and Blackbox (version 3.9, using Chaff) (Kautz and Selman 1995) are all unable to solve even the smallest of these problems.

[5]Thanks to Jörg Hoffman for providing a generator for this collection.

[6]This data was generated using FF v1.0. We also tested version 2.2, which generally gives better quality plans (although still worse than STAN5 in all but one case), but solved only 17 problems, taking longer on each problem than version 1.0.

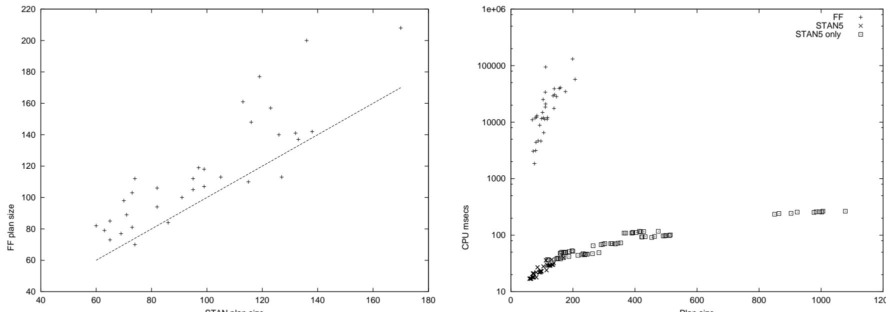


Figure 17 Relative plan quality and time performance of FF and STAN5 on Bulldozer problems. The line represents equal plan length in the first graph, with points above the line representing better STAN5 performance. On the right can be seen the time to produce plans comparing FF with STAN5 and also showing the problems that only STAN5 solved (marked by boxes).

The Bulldozer domain is interesting because, despite being a propositional domain with a simple structure, even moderately sized instances (5 drivers, 10 vehicles, and 20 to 30 locations) are beyond the scope of all other fully automated planning systems in the current literature. STAN5 can handle the domain because of its ability to decompose the route-planning components of the problem into a series of separate problems that can be solved without search. This decomposition is relatively simple in Bulldozer, because the route-planning aspects of the problem are not tightly integrated with its other features. STAN5 is capable of handling more sophisticated forms of integration as well, as discussed in (Fox and Long 2001b).

9 Conclusion

Planning domain descriptions represent the dynamics of specific problem domains in a way that facilitates the application of general problem-solving technology to the solution of problems in those domains. The advantage of exploiting general strategies is that they are reusable without extensive problem-solving effort on the part of the domain engineer. This has been an important argument in favor of knowledge-sparse, domain-independent planning since the inception of planning as a research field. However, because general problem-solving techniques are, by definition, uninformed about specific domain properties, they are impractical for solving all but the simplest planning instances. This clear disadvantage has led many researchers, motivated by application of planning to real problems, to adopt knowledge-rich approaches in which the domain description is engineered to constrain and direct the search behavior of the planner. This approach, while often effective in the domains to which they are tailored, places a heavy engineering burden on the domain designer and results in a planning system that cannot easily be generalized to apply to other domains.

The debate continues as to whether knowledge-sparse or knowledge-rich planning is the most fruitful direction for research in the planning field. A middle ground has been

explored, with a number of researchers considering ways in which a knowledge-sparse planner can be informed by control rules, provided by a domain engineer, that can be used to prune undesirable regions from the search space. If the domain engineer fails to provide all relevant control rules, the planner can default to full search. One reason why this approach is interesting is that the control rules can, in principle, be generalized to apply to families of related domains, so there is scope for the reuse of the control knowledge invested in the modeling of a specific domain.

The idea of reuse is very powerful, with many implications for planning, because planning domains often share many structural features and subproblems. The work we have described in this chapter is motivated by the need to reuse specialized problem-solving technologies, where appropriate, rather than to tackle well-understood, often computationally hard, subproblems using brute-force search. An effective approach to reuse requires that the specific features of domains be abstracted so that common elements can be recognized and exploited. We have developed an abstraction process based on the notion of identifying, using automatic techniques, the occurrence of generic types, and behaviors in a domain description. When the generic structure of a domain is identified, appropriate specialized techniques can be recruited to assist a planner in the efficient solution of problems arising in that domain.

This chapter has focused on the description of how generic types and behaviors, and their associated subproblems, are identified in planning domains by the automatic techniques of the TIM system. We have given a brief description of the architecture of STAN5, an integrated system that exploits the results of the TIM analysis to configure a planner suited to tackling problems in a given domain. This architecture is described in detail in our papers (Fox and Long 2001a; Fox and Long 2001b). We have presented results to demonstrate the power of the integrated approach and have indicated the future directions of this research.

References

Bacchus, F., and F. Kabanza (1996). Using temporal logic to control search in a forward-chaining planner. In M. Ghallab and A. Milani (Eds.), *New Directions in Planning.* IOS Press.

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

Barret, A., D. Christianson, M. Friedman, K. Golden, J. Penberthy, Y. Sun, and D. Weld (1996). UCPOP v4.0 user's manual. Technical Report TR 93-09-06d, Dept. of Computer Science and Engineering, University of Washington, Seattle.

Bonet, B., and H. Geffner (1997). Planning as heuristic search: New results. In *Proceedings of Fourth European Conference on Planning (ECP)*, Springer-Verlag.

Bylander, T. (1992). Complexity results for serial decomposability. In *Proceedings of the 10th National Conference on AI*, Cambridge, MA: AAAI/MIT Press.

Clark, M. (2001). Construction domains: A generic type solved. In *Proceedings of the 20th U.K. Planning and Scheduling Workshop*, Edinburgh.

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

Fikes, R., and N. Nilsson (1971). STRIPS: A new approach to the application of theorem-proving to problem-solving. *Artificial Intelligence 2*(3), 189–208.

Fox, M., and D. Long (1998). The automatic inference of state invariants in TIM. *Journal of AI Research 9*, 367–421.

Fox, M., and D. Long (1999). The detection and exploitation of symmetry in planning problems. In *Proceedings of 16th Internation Joint Conference on AI*, San Francisco, pp. 956–961. Morgan Kaufmann Publishers.

Fox, M., and D. Long (2000). Utilizing automatically inferred invariants in graph construction and search. In *Proceedings of the Fifth Conference on Artificial Intelligence Planning Systems (AIPS)*, Breckenridge, CO. AAAI Press.

Fox, M., and D. Long (2001a). Hybrid STAN: Identifying and managing combinatorial sub-problems in planning. In *Proceedings of 17th International Joint Conference on AI*, San Francisco, pp. 445–452. Morgan Kaufmann Publishers.

Fox, M., and D. Long (2001b). Integrating a general planning strategy with sub-solvers for common problems. Technical report, University of Durham.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). *Design Patterns: Elements of Reusable Software*. Boston: Addison Wesley.

Gerevini, A., and L. Schubert (1996a). Accelerating partial order planners: Some techniques for effective search control and pruning. *Journal of AI Research 5*, 95–137.

Gerevini, A., and L. Schubert (1996b). Computing parameter domains as an aid to planning. In *Proceedings of the Third Conference on AI Planning Systems*, pp. 94–101. AAAI Press.

Gerevini, A., and L. Schubert (1998). Inferring state constraints for domain-independent planning. In *Proceedings of the 16th National Conference on AI*, Cambridge, MA, pp. 905–912. AAAI/MIT Press.

Green, C. (1969). Theorem proving by resolution as a basis for question-answering systems. In B. Meltezer, D. Michie, and M. Swann (Eds.), *Machine Intelligence*, Volume 4. Edinburgh: Edinburgh University Press.

Hoffmann, J., and B. Nebel (2000). The FF planning system: Fast plan generation through heuristic search. *Journal of AI Research 14*, 253–302.

Jónsson, A., P. Morris, N. Muscettola, K. Rajan, and B. Smith (2000). Planning in interplanetary space: Theory and practice. In *Proceedings of the Fifth Conference on AI Planning Systems (AIPS)*, pp. 177–186. AAAI Press.

Kautz, H., and B. Selman (1995). Unifying SAT-based and graph-based planning. In *Proceedings of 14th Internation Joint Conference on AI*, San Francisco, pp. 318–325. Morgan Kaufmann Publishers.

Kautz, H. and B. Selman (1998). The role of domain-specific axioms in the planning as satisfiability framework. In *Proceedings of the Fourth Conference on AI Planning Systems*, Pittsburgh, PA, pp. 181–189. AAAI Press.

Kelleher, G., and A. Cohn (1992). Automatically synthesising domain constraints from operator descriptions. In *Proceedings of the 10th European Conference on AI*, pp. 653–655.

Koehler, J., B. Nebel, J. Hoffmann, and Y. Dimopoulos (1997). Extending planning graphs to an ADL subset. In *Proceedings of the Fourth European Conference on Planning*, Toulouse, pp. 273–285.

Kvarnstrom, J., and P. Doherty (2000). TALplanner: A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence 30(1-4)*, 119–169.

Long, D., and M. Fox (2000). Automatic synthesis and use of generic types in planning. In *Proceedings of the Fifth Conference on Artificial Intelligence Planning Systems (AIPS)*, Breckenridge, CO, pp. 196–205. AAAI Press.

Long, D., and M. Fox (2001). Multi-processor scheduling problems in planning. In *Proceedings of International Conference on AI (IC-AI)*, Las Vegas.

Long, D., M. Fox, L. Sebastia, and A. Coddington (2000). An examination of resources in planning. In *Proceedings of the 19th U.K. Planning and Scheduling Workshop*, Milton Keynes.

McCarthy, J. (1968). Programs with common sense. In M. Minsky (Ed.), *Semantic Information Processing*. Cambridge, MA: MIT Press.

McDermott, D. (1998). PDDL—the planning domain definition language. Technical report, Yale University. www.cs.yale.edu/users/mcdermott.html.

McDermott, D. (2000). The 1998 AI planning systems competition. *AI Magazine 21*(2), 35–56.

Morris, P., and R. Feldman (1989). Automatically derived heuristics for planning search. In *Proceedings of the Second Irish Conference on Artificial Intelligence and Cognitive Science*, School of Computer Applications, Dublin City University.

Muscettola, N. (1994). HSTS: Integrating planning and scheduling. In M. Zweben and M. Fox (Eds.), *Intelligent Scheduling*, pp. 169–212. San Mateo, CA: Morgan Kaufmann Publishers.

Nau, D., Y. Cao, A. Lotem, and H. Muñoz-Avila (1999). SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th International Joint Conference on AI*, San Francisco, pp. 968–975. Morgan Kaufmann Publishers.

Nau, D., S. Gupta, and W. Regli (1995). Artificial intelligence planning versus manufacturing-operation planning: a case study. In *Proceedings of the 14th International Joint Conference on AI*, San Francisco, pp. 1670–1676. Morgan Kaufmann Publishers.

Nebel, B., Y. Dimmopoulos, and J. Koehler (1997). Ignoring irrelevant facts and operators in plan generation. In *Proceedings of the Fourth European Conference on Planning*, Toulouse.

Newell, A., and H. Simon (1963). GPS, a program that simulates human thought. In E. Feigenbaum and J. Feldman (Eds.), *Computers and Thought*. New York: McGraw-Hill.

Pednault, E. (1989). ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of First International Conference on Principles of Knowledge Representation and Reasoning*, San Francisco, pp. 324–332. Morgan Kaufmann Publishers.

Wilkins, D., and M. desJardins (2000). A call for knowledge-based planning. In *Proceedings of the AI Planning and Scheduling (AIPS) Workshop on Analyzing and Exploiting Domain Knowledge for Efficient Planning*.