Towards An Algebraic Formulation of Domain Definitions using Parameterised Machines

T. L. McCluskey and R. M.Simpson
School of Computing and Engineering
The University of Huddersfield, Huddersfield HD1 3DH, UK
t.l.mccluskey,r.m.simpson@hud.ac.uk

Abstract

Abstractly specifying, rapidly creating, and re-using models of domain structure are desirable goals for knowledge engineering in AI Planning. However, while planning algorithms, and now knowledge engineering itself, are maturing areas of research, the descriptions used to formulate dynamic domain descriptions is under researched. In this paper we describe on-going work attempting to establish an algebraic approach to constructing domain descriptions.

We postulate that this formulation will serve several purposes: it will help descriptions of domains to be rapidly built up by composing and interfacing pre-defined abstract, primitive machines. It will help engineers re-use existing domains, and it will serve as a semantics for concrete description languages such as PDDL.

1 Introduction

In the area of knowledge engineering for AI Planning, recent research postulates that domain descriptions can be built up efficiently from a set of re-usable primitive components [6, 5]. Indeed, empirical study of planning domain descriptions suggests that they rely on commonly repeated structures. Knowledge engineers naturally would like to re-use domain descriptions when creating new ones, rather than creating descriptions from scratch. However, when attempting to re-use domain descriptions there are few if any abstractions used in AI Planning to help. In PDDL 1.1 [3] there is an ':extends' facility which implements a form of modularisation, but it is of limited help if an engineer has to change the re-used description in some way because this requires a deep understanding of the re-used description. It is generally the case that the re-used encodings' concrete syntax has to be studied in order to understand them and their interactions with the enhanced encoding. This is related to other problems with domain encoding:

– models of the *same* domain may differ in small details without rationale. For example, how many, and what parameters, should a predicate have? How is this determined? What predicates that are implicitly true should be stated explicitly in the conditions of an operator?

– if we would like to re-use domain descriptions in creating a new one, how are we to retrieve a similar description to work from? Have we to understand all the details of (someone else's) particular encoding before starting?

The problem is not unlike that of coding and re-using software, although the field of abstractions for software modules are far more developed. In AI Planning, an engineer can enter details of a new domain description from scratch, or they have to understand an old one and set about changing it. Previous research has introduced the idea of how objects and object transitions can be used as

the primitives for domain modelling in planning [1, 9, 5]. This takes place during the formulation stage within knowledge acquisition and covers applications where we can assume planning domain descriptions describe the changes of state of objects within the domain as a result of the application of the domain operators.

In this paper we introduce an algebraic method to capture the dynamic structure of AI Planning domains. These are built up by composing and interfacing pre-defined abstract, primitive machines. Changes of state of objects are represented in parameterised state machines and domain descriptions can be synthesised from the combination of those machines. Using state machines as primitives may not be universally applicable to all application domains; indeed we would argue that trying to find a set of primitives that are universally applicable is a bad idea. By necessity, there should be more than one foundation for dynamic representation languages - the state-machine view is just one of them.

PDDL was designed within the area of AI Planning to capture 'dynamics and nothing else'. We follow this maxim in this paper, but seek to find tools to compose and analyse domain descriptions captured this way. In this paper we discuss foundational work which aims to capture domain descriptions using *heterogeneous algebras*. A particular domain description such as the blocks world or the dock workers world can then be captured as a *value* of that algebra. This value we call a *domain definition* to distinguish it from the concrete description.

2 Domain Definitions as Values of Heterogeneous Algebras

A *homogeneous* algebra is a set of values (called the 'carrier' set) together with a set of totally defined, closed operators. 'Closed' here means that the operator's return values are always in the carrier set. For example, the set of numbers known as the 'Natural Numbers', together with the operators '+' and '*', form a homogeneous algebra. Reader interested in this topic may consult reference [8]. A *heterogeneous* algebra is one where there are a *set* of carrier sets, with one particular set representing the values of the algebra being defined. This is called the type of interest (TOI) in the algebraic formulation of data types. The carrier sets that are not the type of interest can be thought of as values from 'imported' algebras. The behaviour of the stack data structure can be readily captured by a heterogeneous algebra, with operators such as 'pop', 'push', 'initialise', and 'top', and the values within the stack taken from one of the imported algebras. Heterogeneous algebras have 'constructors': these are operators that produce new values of the TOI eg stack's 'push', and 'initialise'. 'destructors' break down old values to create new ones, eg 'pop'. 'selectors' are operators that take values of the type of interest and return a value from an imported algebra, eg 'top'.

An important distinction made in the literature is between *abstract* algebras and *concrete* algebras. Abstract algebras are the ones where operators and values are defined abstractly. For example, an algebraic specification may just contain the signature of the operators and a set of equational axioms relating the operators to each other. Concrete algebras are more detailed, and and conform to abstract algebras. For example, a computer implementation of a stack can be thought of as a concrete algebra, whereas the signature and properties of the stack form the abstract algebra. These are related in the following way: A concrete algebra is a *model* of an abstract algebra if (roughly) the values of the concrete algebra satisfy the appropriate properties or axioms in the abstract algebra.

We plan to capture planning domain description languages with abstract, heterogeneous algebras. A domain description will then be a model of some heterogeneous algebra, or in other words: *Each value of the 'type of interest' of the abstract algebra will correspond to a concrete domain description.* The algebra will be built up from various imported algebras, as discussed below. By capturing domains this way, the idea is that we will have well defined, powerful ways of composing domains

descriptions, and machinery to help analyse domains descriptions.

We will exploit the 'object centric view' of domain description to explore domain definition using algebras. The 'object centric view' is where a domain is considered as a set of objects which interact in a well defined dynamic system. Objects, object states and values of object attributes are collected into 'sorts'. These sorts bear a similarity to the idea of object classes in object-oriented design and programming (although sorts are more general and less implementation-oriented than classes). Objects go through transitions from one state to another, and the full set of transitions of an object forms a 'machine'. Thus objects have 'Life Histories" - simply the sequences of values that an object variable can take. World states in planning are made up of sets of individual object states. The idea of forcing conceptualisations of planning domain descriptions into this kind of orientation is used as the basis of tools such as itSIMPLE [7] and GIPO [5]).

3 Domain definition on single sorts - Machines

We start by defining algebras representing state machines giving a limited form of domain definition. These give the basic building blocks of the formulation. Each machine is parameterised by a single object *sort* and hence are formed such that all states and transitions within the machine relate to the same object sort.

3.1 Machine constructors

We introduce the linear machine as the constructor:

lm_{sort} : non-empty list of state names $\Rightarrow Mac_{sort}$

If the list of states contains two adjacent states with the same name that transition forms a self loop and there is no distinction between the transition that takes the machine from the first to the second from the transition that takes the machine from the second to the first. With the exception of self loops all the names constituting the state list must be distinct. Adjacent states in the list will be connected by two transitions, from the first to the second and from the second to the first. The definition below produces the state machine shown in figure 1. The PDDL equivalent of this (produced by the GIPO tool) is shown in figure 2.

$lm_s : [s1, s2, s2] = Machine1$

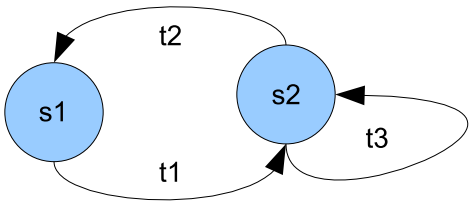

Figure 1: $Machine1$

A machine may be constructed from another machine by a number of operators:

- $rem : Mac_{sort} \times T \Rightarrow Mac_{sort}$

  rem rem takes a machine and a transition belonging to that machine and produces a copy minus the identified transition. The expression

```
(define (domain examples)
  (:requirements :strips :equality :typing)
  (:types s)
  (:predicates (s1 ?s1 - s) (s2 ?s1 - s))

  (:action t2                    (:action t1
    :parameters ( ?S - s)          :parameters ( ?S - s)
    :precondition (s2 ?S)          :precondition (s1 ?S)
    :effect (and                   :effect (and
        (not (s2 ?S))                  (not (s1 ?S))
        (s1 ?S)))                      (s2 ?S)))

  (:action t3
    :parameters ( ?S - s)
    :precondition (s2 ?S)
    :effect)))
```

Figure 2: $Machine1$ - PDDL

$rem : (lm_x : [a, b]), (b \rightarrow a)$

returns machine Machine1 with the single transition from state b to state a removed. If the transition identified does not exist in the machine the operator has no effect (and hence the operator is $total$).

- $prop : Mac_x \times N \times sort_y \Rightarrow Mac_x$

  prop takes a machine of sort x, a property name N and a sort y, and adds a property such that at every state in the given machine, individuals will be related to some value of sort y that constitutes the value for the property N of the object in this state. An axiom on this constructor is that x must not equal y. For example

  $prop : Machine1, location, loc$

  creates a machine as in figure 1 but where $location(s1)$ and $location(s2)$ are defined functions, returning a value of sort loc.

- $merge : Mac_x \times Mac_x \times [(a_1, a_2)/a_3, ...] \Rightarrow Mac_x$

  merge takes two machines of the same sort and a list of pairs of state names where the first name is a state in the first machine and the second is a state in the second machine. $merge$ makes these states identical and renames them with the provided name in the combined machine. Merge preserves the transitions of both machines. For example

  $merge : Machine1, (lm_s : [a1, a1]), [(s1, a1)/b1] = Machine2$

  A diagram representing $Machine2$ is shown in figure 3. We have added a self-loop transition to state $s1$ in $Machine1$ and renamed it to state $b1$.

- $chg : Mac_x \times T \times P \times PC \Rightarrow Max_x$

  Chg takes a machine, a transition identifier, a state property name $p \in P$ and a propositional constraint on p and produces a new machine of the same sort where the property p changes value in accordance with the propositional constraint whenever the machine makes the identified transition. For example:

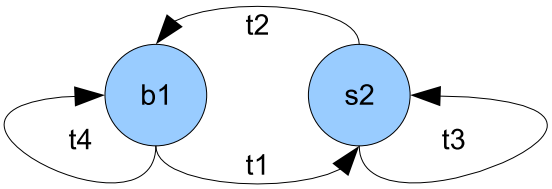

Figure 3: Machine2

$chg : (prop : (lm_s[s1, s1]), location, loc), s1 \rightarrow s1, location, next$

produces a machine of sort s with the single state $s1$ and a single self loop. The machine has a single property $location$ that is of value type loc and the transition forming the self loop transforms the value of the property location according to instances of the predicate $next$ that links two values of the type loc. Consider an example with a single next predicate $next(a, b)$. If the machine had an initial value of the location property of a, it could only make one transition of the self loop with the result that the location property of the object instance would have location b. A trace of the example is shown in figure 4

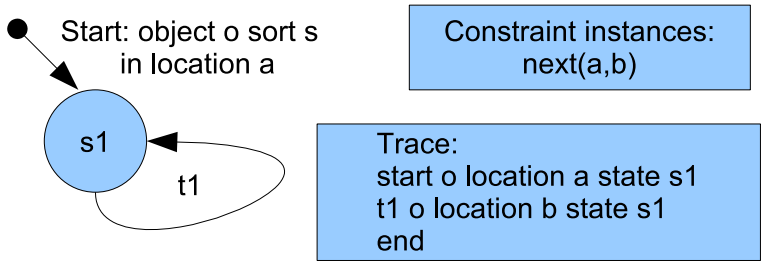

Figure 4: Trace Example

The PDDL code auto-generated from the Trace Example Machine is shown in figure 5.

3.2 Domain definitions - Multiple Sorts

Domain definitions are formed from machines/domain definitions and operators on domain definitions. Any machine specification is a domain definition. There is an invisible operator *promote* that can be applied to any machine.

$promote : Mac \rightarrow DD$

Values of DD are abstract algebra expressions, each representing a concrete domain description (in for example PDDL). Combining Domain definitions uses a *Union* operator:

$\cup : DD \times DD \rightarrow DD$

This combines two domain definitions. It renames identifiers in the second domain definition to ensure that there is no overlap in the name spaces of the two domains.

```
(define (domain examples)
  (:requirements :strips :equality :typing)
  (:types s loc)
  (:predicates (s1 ?s1 - s)
    (location ?s1 - s ?loc1 - loc)
    (next ?loc1 - loc ?loc2 - loc))

  (:action t1
    :parameters ( ?S - s ?LocA - loc ?LocB - loc)
    :precondition (and (s1 ?S)
        (location ?S ?LocA)
        (next ?LocA ?LocB))
    :effect (and
        (not (location ?S ?LocA))
        (location ?S ?LocB))))
```

Figure 5: Trace Example Machine - PDDL

Simulating actions that are formed from a collection of object transitions is performed by variations on three main operators. These link up transitions between machines to simulate classical planning domain operators.

- $prevail : DD \times T \times S \rightarrow DD$

  Prevail takes a domain definition, a transition t_{s1} of sort $s1$, and a state name s_{s2} of sort $s2$ from the domain definition. It produces a domain definition where t_{s1} can only be made by an object of sort $s1$ when another object of sort $s2$ is in the identified state and that object remains in that state when the identified transition is made.

- $necessary : DD \times T \times T \rightarrow DD$

  Necessary takes a domain definition and two transitions drawn from two distinct machines, and requires that both transitions occur simultaneously.

- $conditional : DD \times T \times T \rightarrow DD$

  Conditional takes a domain definition and two transitions and requires that the second transition can only occur simultaneously with the first, but no such restriction is placed on the first.

At this stage we have an abstract algebra powerful enough to construct some values which have as models standard planning domains. For example consider the Tyre Change domain, which is set up to allow plans to be generated to change a flat tyre on a car. This has 6 machines representing the transitions of one of the car's wheel hubs, its wheels, its boot, the fixing nuts on a wheel, and a wrench and car jack.
$lm_{hub} : [onground, raisedup, upnonuts, upnowheel] = hub\text{-}m$
$lm_{jack} : [j\text{-}incar, j\text{-}carried, inuse] = jack\text{-}m$
$lm_{wheel} : [w\text{-}incar, w\text{-}carried, onhub] = wheel\text{-}m$
$lm_{wrench} : [wr\text{-}incar, wr\text{-}carried] = wrench\text{-}m$
$lm_{carboot} : [open, locked] = carboot\text{-}m$
$lm_{nuts} : [fastened, loose, free] = nuts\text{-}m$

$$(\cup : nuts, (\cup : hub\text{-}m, (\cup : jack\text{-}m, (\cup : wheel\text{-}m, (\cup : wrench\text{-}m, carboot\text{-}m))))) = m1$$

Value $m1$ represents a domain with 6 machines specified, where objects can change state any linear direction, but with no machine interaction specified. We use the $prevail$, $necessary$ and $conditional$ operators above to specify how the machines interact.

$$necessary : m1, j\text{-}carried \rightarrow inuse, onground \rightarrow raisedup = m2$$

This creates $m2$, representing a new domain description where the transition of a car hub (from being on the ground to being raised up) must happen at the same time as the transition of a jack (from being carried to being used).

$$prevail : (prevail : m2, fastened \rightarrow loose, wr\text{-}carried), fastened \rightarrow loose, onground = m3$$

In $m3$, to loosen the nuts on a hub, the hub must be on the ground and the wrench must be carried.
$necessary : m3, loose \rightarrow free, raisedup \rightarrow upnonuts = m4$
$prevail : (prevail : m4, loose \rightarrow free, inuse), loose \rightarrow free, wr\text{-}carried = m5$

In $m5$, freeing the nuts on a hub occurs at the same time as the nuts leaving the hub. This can only be done if the jack is in use and the wrench is being carried.

At this point we might carry on to specify the final handful of interactions within the usual version of the domain description. However, it is appropriate to point out a deficiency in the formulation up to now: *Certain relationships between objects that are formed at an interaction between machines must persist through a number of machine states.* For example, the jack that is used to raise a hub is the same jack that is removed once the hub is lowered. And a wheel that is put on a hub is the same as one that is removed. So in general inter-action between machines forms relationships between objects that must persist - we call these object $associations$. States need to have $memories$, and the association is used for this purpose. In predicate-based languages like PDDL associations are recorded in domain descriptions using predicates, or by introducing extra arguments to existing predicates. These predicates are added or deleted from the states by hand crafted action representations. Whereas the rational for predicate argument formation in logic-based languages appears vague, in the algebraic formulation it it explicit as shown in the next section.

3.2.1 Domain Operator Modifiers

The above three operators **prevail, necessary** and **conditional** can be modified in their requirements in the following ways. **add association (+)** or **remove association (-)** or **constrain properties (c)**

- Add association

  $prevail+ : DD \times T \times S \rightarrow DD$
  $necessary+ : DD \times T \times T \rightarrow DD$
  $conditional+ : DD \times T \times T \rightarrow DD$

  $(prevail+ : d, s1 \rightarrow s2, s)$ is a new domain description where the object in state $s2$ becomes associated with the object in state s (the prevail state). This association is transmitted to all states in the machine containing $s1$ and $s2$ that can be reached from transitions through $s2$. The association can be terminated in the state sub graph when a transition identified in a **remove association** is traversed (see below).

  $(necessary+ : d, s1 \rightarrow s2, t1 \rightarrow t2)$ is similar: here the object making the transition into $s2$ becomes associated with the object making the transition into $t2$. This association is transmitted

to all reachable states from the state $s2$. The association is only terminated in the state sub graph when a transition identified in a **remove association** is traversed. The associated sub graph so formed is bounded by $+$ and $-$ operators linking the same associated sort.

$conditional+$ is similar to $necessary+$.

- Remove Association

  $prevail-: DD \times T \times S \rightarrow DD$
  $necessary-: DD \times T \times T \rightarrow DD$
  $conditional-: DD \times T \times T \rightarrow DD$

  Remove association (-) requires that their is already an association between the objects. In this case the association between them is terminated.

- Constrain Properties

  Constrain properties (c) adds an extra clause to the prevail, necessary or conditional operators.

  $prevail\ c: DD \times T \times S \times P_1 \times P_2 \times PC \rightarrow DD$
  $necessary\ c: DD \times T \times T \times P_1 \times P_2 \times PC \rightarrow DD$
  $conditional\ c: DD \times T \times T \times P_1 \times P_2 \times PC \rightarrow DD$

  Where P_1 refers to a property of the object making the first named transition from the domain DD. P_2 refers to a property of the object in the state (prevail) or object making the second named transition (necessary or conditional). PC refers to a propositional constraint that must relate the two properties P_1 and P_2 for the transitions to be made. A simple example of the $prevailc$ operator occurs in transport domains where it is used to require that the presence of an available truck should be at the same location as a package that is loaded into the truck. This is also a case where a persistent association should be formed to remember which truck the package is loaded into. We allow the modifiers "+" or "-" to be combined with the "c" modifier in the same operation. This is illustrated in a simple "mobile" example below (loc = location, and lded = loaded).
  $prop: (lm_{truck}: [available, available]), loc, loc = Truck$
  $prop: (lm_{package}: [unlded, lded, lded]), loc, loc = Package$
  $prevail + c: (\cup: Truck, Package), (unlded \rightarrow lded), available, loc, loc, eq = Mobile_1$
  Where eq is simply an equality predicate. To complete this "mini" transport domain we need the operator:
  $prevail - c: Mobile_1, (lded \rightarrow unlded), available, loc, loc, eq = Mobile_2$
  To ensure that the package is unloaded at the same location as the truck it is in. At which point the association is forgotten. Finally to require the package to change location as the truck changes location we need:
  $conditionalc: Mobile_2, (available \rightarrow available), (lded \rightarrow lded), loc, loc, eq = Transport$
  $Transport$ defines a domain in which trucks can move packages from location to location.

4 Dock Workers Robots Example

To test out the formulation with respect to expressiveness and power of abstraction, we attempt to capture the Dock Workers Robots domain. This is a non-trivial example developed for Ghallab, Nau and Traverso's recent textbook [4]. In the dock workers robots example there are automated robots that can move containers, one at a time, around a port area. The robots are loaded and unloaded by

static cranes. The containers that are moved around are stacked in piles at the same locations as the static cranes. This specification is built from three machines one for each of the *robots, containers* and *cranes.*

Robot

$prop : (lm_{robot} : [free, free, busy, busy]), location, loc = Robot_1$
$chg : Robot_1, free \rightarrow free, location, next = Robot_2$
$chg : Robot_2, busy \rightarrow busy, location, next = Robot_3$

These three constructors produces the value $Robot_3$, corresponding to the machine representing the robot's dynamic behaviour. The resulting machine is shown in figure 6 where the transitions have been given meaningful names, for example the transition from *free* to *busy* is labeled *load.* What we are capturing is that the robots have two primary states *free, busy* and that in either state robots have a location property. Additionally the robot may change its location, either when free or when busy by moving *move,moveContainer.*

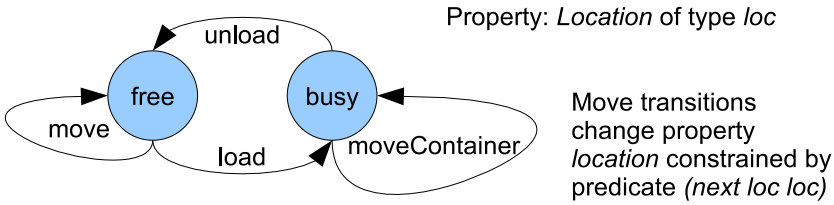

Figure 6: Robot Life History Diagram

Container

The stacking of containers form a version of the classic 'Blocks World' problem where the containers are stacked in towers at named locations. To model the containers we need states to identify containers loaded on a robot, temporarily held by a crane, and on a stack of containers. However to keep track of containers in on a stack we need to distinguish between three possible states, 1. forming the top of the stack with no containers below, 2. forming the top of the stack with a single container below and 3. forming the top of a stack with two or more containers below. Much of the complexity of the problem comes from tracking how containers can change state as they and other containers are placed on and removed from the stacks. All of these states are of course parameterised by the location of the containers.

$prop : (lm_{container} : [onRobot_1, onRobot_1, held_1]), location, loc = Cont_1$
$prop : (lm_{container} : [held_2, onBaseTop_2]), location, loc = Cont_2$
$prop : (lm_{container} : [onBaseTop_3, onBase_3]), location, loc = Cont_3$
$prop : (lm_{container} : [held_4, onTop_4]), location, loc = Cont_4$
$prop : (lm_{container} : [onTop_5, on_5]), location, loc = Cont_5$
$prop : (lm_{container} : [held_6, onTopBase_6]), location, loc = Cont_6$

These six equations define the state changes a container can make and ensure that every state can be qualified by a *location* property.

$necessary + c : (\cup : Cont_6, Cont_3), held_6 \rightarrow onTopBase_6, onBaseTop_3 \rightarrow onBase_3,$
$location, location, eq = Cont_7$

$necessary-c:(\cup:Cont_6,Cont_3),onTopBase_6 \rightarrow held_6,onBase_3 \rightarrow onBaseTop_3,$
$location,location,eq = Cont_8$
$necessary+c:(\cup:Cont_4,Cont_5),held_4 \rightarrow onTop_4,onTop_5 \rightarrow on_5,$
$location,location,eq = Cont_9$
$necessary-c:(\cup:Cont_4,Cont_5),onTop_4 \rightarrow held_4,on_5 \rightarrow onTop_5,$
$location,location,eq = Cont_{10}$

The next four equations require that certain state changes *necessarily* occur together. The first of these requires that when one container changes state from $held_6 \rightarrow onTopBase_6$ another container must simultaneously change state from $onBaseTop_3 \rightarrow onBase_3$. That is when a new container is placed on a single container stack the new container is $onTopBase$ and the container now below is in state $onBase$. The $+c$ annotation requires that the top, first referenced, container establishes a memory/association of which container it is placed on and the constraint is placed such that the two containers are in the same location. The other three *necessary* equations can similarly be understood, noting that where we have the $-$ annotation the association memory is removed/forgotten.

$merge:Cont_7,Cont_8,[(held_6,held_6)/held] = Cont_{11}$
$merge:Cont_9,Cont_{10},[(held_4,held_4)/held] = Cont_{12}$
$merge:Cont_{11},Cont_{12},[(held,held)/held] = Cont_{13}$
$merge:Cont_{13},Cont_2,[(held,held_2)/held] = Cont_{14}$
$merge:Cont_{14},Cont_1,[(held,held_1)/held] = Cont_{15}$

The five *merge* equations above essentially assert that the variously indexed $held$ states all refer to the same state with the canonical name $held$.

$chg:Cont_{15},onRobot_1 \rightarrow onRobot_1,location,next = Cont_{16}$

The final equation defining the *container*'s life history indicates that the transition $onRobot_1 \rightarrow onRobot_1$ involves changing the location property of the container, where the change obeys a *(next loc,loc)* constraint.

Crane

$prop:(lm_{crane}:[available_1,inUse_1]),location,loc = Crane_1$
$prop:(lm_{crane}:[available_2,inUse_2]),location,loc = Crane_2$
$prop:(lm_{crane}:[available_3,inUse_3]),location,loc = Crane_3$
$prop:(lm_{crane}:[available_4,inUse_4]),location,loc = Crane_4$
$merge:Crane_1,Crane_2,[(available_1,available_2)available,(inUse_1,inUse_2)/inUse] = Crane_5$
$merge:Crane_5,Crane_3,[(available,available_3)available,(inUse,inUse_3)/inUse] = Crane_6$
$merge:Crane_6,Crane_4,[(available,available_4)available,(inUse,inUse_4)/inUse] = Crane_7$

The seven equations for the Crane define a machine with two states *available,inUse* which are connected by four transitions from *available* $\rightarrow$ *inUse* and four from *inUse* $\rightarrow$ *available.* The Crane also has a property of *location* with type *loc.* The first four equations set up the machines each with a pair of transitions. The states of those four machines are then merged in the following three equations. The multiple transitions are required to correspond to the different actions of picking up (or placing on) a container from (1) a robot (2) the container in each of: a single container stack, the container on top of the base container, the container on top of any stack of size greater than two.

The three separate machines for robots, containers and cranes have been set up. We need to compose them together and model their interactions.

Compound Domain definitions

Link loading Container to Robot when loading and unloading.
$necessary + c : (\cup : Robot_3, Cont_{16}), free \rightarrow busy, held \rightarrow onRobot_1, location, location, eq = DD_1$
$necessary + c : DD_1, busy \rightarrow free, onRobot_1 \rightarrow held, location, location, eq = DD_2$
$necessary : DD_2, onRobot_1 \rightarrow onRobot_1, busy \rightarrow busy = DD_3$

These equations all require that transitions from the disparate machines accompany one another and may establish or remove a remembered association. The first for example requires that when a robot changes from states $free$ to $busy$ a container must simultaneously change state from being $held$ to being $onRobot_1$ additionally the constraint is placed that the container and robot must be in the same location and that the association between robot and container is remembered.

Link crane to container loaded onto robot
$necessary+c : (\cup : Crane_7, DD_2), inUse_1 \rightarrow available_1, held \rightarrow onRobot_1, location, location, eq = DD_3$
$necessary - c : DD_3, available_1 \rightarrow inUse_1, onRobot_1 \rightarrow held, location, location, eq = DD_4$

Link crane to lifting from and placing containers on stacks.First placing on and removing from empty base.
$necessary+c : DD_4, inUse_2 \rightarrow available_2, held \rightarrow onBaseTop_2, location, location, eq = DD_5$
$necessary-c : DD_4, available_2 \rightarrow inUse_2, onBaseTop_2 \rightarrow held, location, location, eq = DD_6$

Next placing on and removing from container directly on base.
$necessary+c : DD_6, inUse_3 \rightarrow available_3, held \rightarrow onTopBase_6, location, location, eq = DD_7$
$necessary+c : DD_7, available_3 \rightarrow inUse_3, onTopBase_6 \rightarrow held, location, location, eq = DD_8$

Finally placing on top of stack and removing
$necessary + c : DD_8, inUse_4 \rightarrow available_4, held \rightarrow onTop_5, location, location, eq = DD_9$
$necessary - c : DD_8, available_4 \rightarrow inUse_4, onTop_5 \rightarrow held, location, location, eq = DD_9$

The domain definition DD_9 is an accumulated domain definition for the dock workers robots world. The 'size' of the formalism indicates that it is an abstraction of particular domain encodings (such as the source encoding from the text book's website).

5 Conclusions

We have described an initial formulation of planning domain definitions as values of an abstract, heterogeneous algebra, and described an extended example in an attempt to evaluate it. This work parallels the visual interface of the GIPO III tool[1]. In this tool's interface, the user builds up a domain in terms of primitive and composed machines *graphically.* The tool then translates the graphical representation to domain description languages (currently OCL and PDDL). Here, we have built up the definition algebraically. From our initial investigation, we postulate that an algebraic approach using state machines gives the following benefits:

[1] http://scom.hud.ac.uk/planform/gipo

1. abstraction: the domain encoder has less decisions to make about details of the encoding. In particular the domain encoder need not think in terms of predicates and parameters, but rather in terms of state transitions and machines. The latter has a more engaging visual metaphor as getting an expert to describe life histories of objects is easier that getting them to describe predicates, parameters and parameter co-designations.
2. re-use via rapid generation and extension of domain descriptions : new domain descriptions can be efficiently composed from existing high level primitives using composite algebraic operators. Old domain descriptions described in terms of abstract machines can be efficiently enhanced or changed.
3. analysis: the approach means domain descriptions are broken down into connected, known primitive machines. The kinds of primitives and the way they are combined will give a kind of classification to the domain.
This initial work has provided some evidence for 1. and 3. We have implemented and tested a translator which inputs algebraic formulations of machines and outputs concrete domain descriptions. This provides some support for 2. However, many questions remain:
- will the expressive power of the algebraic formulation be able to match that required of domain description languages such as PDDL 2.1 etc?
- to what degree will the suppression of domain description detail help engineers understand domain definitions?
- what relation does our formulation have to other to other formal languages used in computing, and previous semantic descriptions of domain description languages[2]?
Part of our future work will be to investigate a range of PDDL domains eg the competition domains and investigate the degree to which they can be captured and analysed using algebra.

References

[1] A. Cesta and A. Oddi. DDL.1: A Formal Description of a Constraint Representation Language for Physical Domains. In M. Ghallab and A. Milani, editors, *New Directions in AI Planning*, pages 341–352. IOS Press, 1996.

[2] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains . In *Technical Report, Dept of Computer Science, University of Durham*, 2001.

[3] M. Ghallab, A. Howe, C. Knoblock, D. McDermott, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl - the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[4] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann ISBN 1-55860-856-7, 2004.

[5] T. L. McCluskey and R. M. Simpson. Knowledge Formulation for AI Planning. Proceedings of 4th International Conference on Knowledge Engineering and Knowledge Management (EKAW 2004) Whittlebury Hall, Northamptonshire, UK, 2004. Published by Springer in the LNAI series., 2004.

[6] R. M. Simpson, T. L. McCluskey, Maria Fox, and Derek Long. Generic Types as Design Patterns for Planning Domain specifications. In *Proceedings of the AIPS'02 Workshop on Knowledge Engineering Tools and Techniques for AI Planning*, 2002.

[7] T.S.Vacquero and F.Tonidanel and J.R.Silva. The itSIMPLE tool for Modelling Planning Domains. In *Proceedings of the First International Competition on Knowledge Engineering for AI Planning, Monterey, Califormia, USA*, 2005.

[8] J. G. Turner and T. L. McCluskey. *The Construction of Formal Specifications: an Introduction to the Model-Based and Algebraic Approaches*. McGraw-Hill Software Engineering Series, London. ISBN 0-07-707735-0, 1994.

[9] D. Wilkins. Using the SIPE-2 Planning System: A Manual for SIPE-2, Version5.0. SRI International, Artificial Intelligence Center, 1999.