Technical Paper

Conversion of SIPE2 Domain Descriptions to O-Plan2 TF

Brian Drabble

Artificial Intelligence Applications Institute
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
United Kingdom

January 6, 1997

ARPA-RL/O-Plan/TR/1 Version 1

Contents

Abstract

The aim of this document is to describe the steps required to convert a domain description specified in the SIPE-2 Domain Description Language (DDL) to the Task Formalism language (TF) used in O-Plan2. The document will outline the differences in syntax for common items such as expansions, conditions, effects etc. and where possible the required substitutions will be described. Differences in semantics and methods of search control however, cannot be handled by simple substitutions and as such guidelines will be provided as to possible reformulations of specified items. At present the descriptions will be limited to the SIPE-2 for Crisis Action Planning (SOCAP) domain. However, it is hoped in later versions of the document to cover the complete range of SIPE-2 DDL constructs. The papers concludes with a simple example of the conversion of a SIPE-2 operator schema to its equivalent in TF and a series of discussion points raised during the study. The discussion points describe the two major differences or *gaps* between SIPE-2 DDL and TF and suggest possible ways in which a mapping could be made from one to the other.

1 Introduction

The aim of this document is to describe the steps required to convert a domain description specified in the SIPE-2 Domain Description Language (DDL) to the Task Formalism Language (TF) used in O-Plan2. These two languages carry out the same *function* but have very different syntax and underlying philosophies. The languages allow the user to represent a number of different aspects of a chosen application domain. These are as follows:

1. **Domain Objects**
   These describes the different objects in the domain, their attributes and the classes to which they belong. For example, a block may have attributes mass, colour, position, etc and may belong to the class objects, sub-class movable objects.

2. **Tasks**
   These describe the particular tasks which can be performed in the given domain. This may or may not include the description of the initial state of the world prior to the start of the task.

3. **Operators**
   These describe the actions and operator schemas which can be used by the planning system in the generation of a plan for a specified task.

4. **Domain Rules**
   These specifies the "rules" which can be used to deduce additional statements about the domain. For example, `''if no block is on top of block1 then block1 is clear''`.

Each of these different aspects is dealt with in the following sections. The report concludes with:

1. an example of the conversion of a SIPE-2 DDL specified operator to TF. It shows that such a conversion is possible to mechanise but the resulting TF is far from satisfactory.

2. two discussion examples where the differences between the SIPE-2 DDL and TF are such that further thought is required. The aim of these examples is to start such as discussion.

A list of support materials which were used during the preparation of this report is described in Appendix 1.

2 Review of the Major Structures in SIPE-2 DDL

The aim of this section is to describe the major structural components of the SIPE-2 DDL. Where direct equivalents with current TF structures are available they will be shown along side the SIPE-2 instructions. Major differences between SIPE-2 DDL will be described in Sections 4 and 5 together with a discussion concerning the possible ways in which these *gaps* may be bridged.

2.1 Representation of Predicates

One of the major differences between the two languages is the representation of `patterns`. SIPE-2 uses a scheme in which the pattern is delimited by brackets of the form `()`. The value that pattern is assigned is assumed to be `true` as in predicate logic. For example, `(distance POPE.AFB.AFB FT.BRAG 100.00)`, `(located 63rd-SFB FT.BRAGG-LAND)`.

In O-Plan2 all patterns are delimited by brackets of the form {}. The value of the pattern is expressed functionally and may not just be a truth value. For example `{colour ball} = green`, `{on a b} = true`, etc. When converting from SIPE-2 DDL to O-Plan2 TF it is very important that as many of these functional relationships can be identified and used. By using such functional relationships is it possible to significantly reduce the size of search spaces and in doing so make planning tractable. For example suppose, we have a switch with two positions `on` and `off` and two operators `turn-on` and `turn-off`. By modelling this as a set of truth values the operator `turn-on` would need to assert the effects, `{switch1 on} = true` and `{switch1 off} = false`. The operator `turn-off` would need to assert the opposite values. However, by modelling this as a functional relation between the switch and its position it can be modelled in a single statement as follows, `{status switch1} = on` and there is no need to keep track of any conflicting states.

2.2 Object Hierarchy

The object hierarchy within SIPE-2 is defined by means of classes and subclasses. These classes define particular *instances* which define objects which belong to the named class. For example:

```
CLASS: territory
SUBCLASSES: land,sea,airspace,location;
INSTANCES: ntunisia-t,stunisia-t,tunisia,spain,sicily,italy,france,nalgeria;
END CLASS

CLASS: land
PARENT-CLASS: territory
SUBCLASSES: region,route;
INSTANCES: ntunisia-l,stunisia-l,nalgeria,salgeria;
END CLASS
```

```
CLASS: sea
PARENT-CLASS: territory
SUBCLASSES: sea-sector,sea-loc;
INSTANCES: medsea,atlantic;
END CLASS

CLASS: region
PARENT-CLASS: land
INSTANCES: nwtunisian-border,ntunisian-border,stunisian-border,
           swtunisian-border,wtunisian-border,
           ntunisian-coastal,nwtunisian-coastal,netunisian-coastal;
END CLASS
```

Each of the given *instances* is later defined by an *object* which defines the particular *properties* of the instance. For example, the forces available in a given domain could be described as follows:

```
                          Forces
                            |
          -----------------+------------------------
          |                 |         |              |
      cargo-by-air         air      navy           army
                            |
      ---------------------+------------------
      |          |       |        |          |
   tattack    sattack  sairlift  tairlift  t-fighter
                                             |
                                       ------+------
                                       |           |
                                   Mig23-sq    Mig25-sq
```

The definition of the class `Mig23-sq` (squadrons of Mig23 jets) and particular instances of these squadrons i.e. `alg-1stmig23 alg-2ndmig23` would be as follows:

```
CLASS: MIG23-sq
PARENT-CLASS: tfighter
INSTANCES: alg-1stmig23,alg-2ndmig23;
END CLASS
```

The properties of these squadrons would are associated with and *object* as follows:

```
OBJECT: alg-1stmig23
PARENT-CLASS: MIG23-sq
PROPERTIES:
```

```
 F-RATIO = 10,
 AIRCRAFT = 25,
 AIR-FIREPOWER = 250,
 AIR-MOBILITY = 3,
 SORTIES = 50,
 A-RANGE = 400;
END OBJECT
```

These properties are **static** facts about this particular object and cannot be changed during the course of the plan. Properties can also be defined for whole classes of objects which all of their instances will inherit.

The current SOCAP domain has been broken down into a set of files as follows:

- **socap-classes.sipe**: defines the classes of objects to be found in most military planning domains
- **socap-oprsonly.sipe**: contains the operators avialable in the current domain
- **socap-preds.sipe**: defines the initial *dynamic* information of the domain
- **socap-probs.sipe**: defines the different problem formulations for the domain

These files are sufficient to run the particular problems defined for the domain. However, three extra files have been provided which provide information specific to this particular domain i.e. "the Tunisian Scenario".

- **socap-newpreds.sipe**: problem specific predicates
- **socap-newclasses.sipe**: problem specific classes
- **apportion.lisp**: problem specific instances and objects of the class defined in the socap-newclasses.sipe

The information provided in these files contains details of specific units and their requirements e.g. number of personal, tonnage of support materials, status, etc. The following example shows how the information in the files is integrated:

1. The file `socap-classes.sipe` defines the structure that an armoured division is a subclass of army which in turn is a subclass of forces.
2. This structure is then *augmented* by the entries in `socap-newclasses.sipe` by the fact that an armoured division may itself be subclassified into an armoured cavalry regiment (ACR)
3. The particular instances of an ACR which are available in the domain are stored in the file `apportion.lisp` i.e. the `3rd-ACR`, the `107th-ACR`, the `116th-ACR` and the `278th-ACR`.

However, the problem is complicated by the fact that the information held in the files `socap-newclasses.sipe` and `apportion.lisp` is not stored in the same format as that in the four "base" files! Instead a series of SIPE-2 specific input/reformatting routines are used to convert the information into a form which SIPE-2 can handle. These three files are the ones which will need to be converted when we consider a new problem domain such as "the Sri Lankan Scenario". A description of the proposed system of class representation to be used in the O-Plan2 systems in described in Section 1.5.

2.3 Operator Schemas

The operator schemas in the two different languages need to represent the same basic set of concepts and ideas. The SIPE-2 DDL classifies **four** different types of operator:

1. **operator**
   The definition of an action which can change the state of the world.
2. **causal rule**
   A rule which can be used to infer new statements about the state of the world at a point in the plan. For example, `''if no block is on top of block1 then block1 is clear''`.
3. **state rule**
   A rule which can be used to infer the state of a given object at a point in the plan. For example, `if object1 is on support1 and support1 is in room1 then object1 is also in room1`
4. **init.operator**
   A rule which can be used to infer new statements about the world (i.e. like a causal rule) but the rule is only applied to the initial state specification.

The idea behind dividing the rules into two sets is that rules about causal events should be matched before constraints about the domain. In all domains implemented in SIPE-2 state rules never have a precondition (in O-Plan2 this is an `only_use_if` condition) only a condition (in O-Plan2 this is an `achieve` condition) while causal rules always have a precondition and perhaps a condition. Thus the causal rules are reacting to changes between states, while deductive rules are enforcing the constraints within a state. Causal rules are **always** applied **before** the state rules. (**NB** there is no difference between causal rules and deductive rules other than their order of applicability - they have identical expressive powers. Both types of rule may have both conditions and preconditions (or neither)).

In the case of the `trigger` and `conditions` they are **always** matched in the **current** world state while preconditions are matched in the **previous** world state. All deductions that can be made are performed when a new node is inserted into the plan. IN SIPE-2 the deduced effects are recorded as if they were brought in as a direct result of the node. Deductions are **not** attempted at other points in the planning process.

Initially all causal rules whose trigger matches a node specified effect are applied, thereby producing an initial set of deduced effects for that node. After all such rules have been applied, the planner determines which newly deduced effects were not already true in the given situation and permits the causal rules to trigger on them recursively. This process continues until no effects are deduced that were not already true, thus computing the deductive closure of the causal rules. This process is then repeated for the state rules.

The trigger pattern itself can be present in both the `effects` field **and** the `goal` field of a schema. From the description provided in the SIPE-2 DDL manual the rules are only applied to the effects field and hence the `goal` fields do not cause a deductive process to begin.

In the given SOCAP domain there only two domain rules which define the state of a given `force` and as follows:

```
STATE-RULE: arrived1
ARGUMENTS: force1,territory1,territory2;
TRIGGER: (ground-moved force1 territory1 territory2);
EFFECTS: (located force1 territory2);
END STATE-RULE

STATE-RULE: arrived2
ARGUMENTS: force1,territory1,territory2;
TRIGGER: (moved force1 territory1 territory2);
EFFECTS: (located force1 territory2);
END STATE-RULE
```

The trigger fields of these rules can be *set* by a large number of plan schemas. The following table describes those schemas which assert the trigger pattern and the particular field of the schema in which the pattern can be found. As described earlier the trigger pattern of the rule may be part of a schema `goal` field **or** its `effect` field. In the case of `goal` field the assertion of the pattern as a `goal` will **not** cause the rule to trigger. The `goal` pattern is used to match against the `purpose` field of a schema designed to achieve the specified goal.

Rule	Schema Name	Trigger Type
`arrived1`	Deter-border-incursion-by-ground-patrol	`effect`
	Deter-border-incursion-by-key-terrain	`effects`
	Provide-defence-by-key-terrain	`effects`
	Provide-defence-by-protect-loc	`effects`
	Deploy-sof	`goal`
	Deploy-army	`goal`
	Deploy-airforce	`goal`
	Deploy-navy1	`goal`
	Deploy-navy2	`goal`
	move-ground1	`effects`
`arrived2`	Deter-border-incursion-by-admin-landing	`effects`
	Deter-border-incursion-by-amphib-landing	`effects`
	Deter-naval-zone-incursion-by-naval-patrol	`effects`
	Provide-defense-by-naval-patrol	`effects`
	Deploy-sof	`goal`
	Deploy-army	`goal`
	Deploy-airforce	`goal`
	Deploy-navy1	`goal`
	Deploy-navy2	`goal`
	move-by-sairlift1	`effects`
	move-by-airlift+sealift	`effects`
	move-by-tairlift1	`effects`
	move-by-sealift1	`effects`
	move-by-sealift2	`effects`
	move-by-sealift3	`effects`
	move-by-sealift4	`effects`
	move-submarine1	`effects`
	move-submarine2	`effects`
	locate-tairlift	`effects`
	locate-sairlift	`effects`
	locate-ssealift	`effects`

In the currently implemented TF however, there is only a single operator type and at present no way of using state or causal rules. The next section of this report will only consider the conversion of plan operator descriptions and does not consider `causal rules`, `state rules` or `init.operators`. The following section will describe in outline the common areas between the SIPE-2 DDL and TF by showing equivalent statements in each language.

2.3.1 Schema Names

SIPE-2	`OPERATOR <name>`
O-Plan2	`schema <name>;`

This declares the name of the particular operator and must be a unique identifier.

```
SIPE-2: OPERATOR: puton

O-Plan2: schema puton;
```

2.3.2 Variable Declarations

SIPE-2	`ARGUMENTS: <argument_list>`
O-Plan2	`vars <vars_list>;`

This declares the variables which are used local to the schema.

NB the problems which seem to arise here are with the binding of variables in particular when? and how?, and the passing of variables into a schema. Specification of Actors in particular `instantiate` need to be examined.

O-Plan2
The `<vars_list>` of O-Plan2 contains the variables being declared and their type definitions which allow the planner to associate the variable specified to a given class of objects and to give any restrictions on its value. Further information concerning the relationship between variables is declared in a separate `vars_relations` field.

SIPE-2
The `<argument_list>` of SIPE-2 contains the variables being declared, a set of restrictions on the values which the variables can take and some information concerning their types.

For a full description of object types and the definition of the classes refer to Section 2.5. In the following example the same set of variable declarations and their restrictions are given using the two different languages.

```
SIPE-2: ARGUMENTS: block1, object1 is not block1

O-Plan2: vars ?block1  = ?{type objects},
              ?object1 = ?{type objects};

         vars_relations ?block1 /= ?object1;
```

2.3.3 Schema Trigger Conditions

SIPE-2	`PURPOSE: <action or goal pattern>`
O-Plan2	`only_use_for_effects <pattern> = <value>, ...  ;`
	`expands <action pattern>;`

This describes the reasons under which the operator may be used. In the case of SIPE-2 there can only be a single `PURPOSE` and the pattern `PURPOSE` serves as a trigger for action expansion as well as for condition satisfaction. In the case of O-Plan2 a distinction is made between the use of a schema for expansion **expansion** and its use for condition satisfaction **only_use_for_effects**. An example of its use is as follows:

```
 SIPE-2: PURPOSE: (border-incursion-deterred coa1)

O-Plan2: only_use_for_effects {border-incursion-deterred ?coa1} = true;
```

2.3.4 Schema Decomposition

```
SIPE2    PLOT: <expansion>
O-Plan2  nodes <expansion item>;
         orderings <orderings>;
         conditions <conditions>;
         resources <resource usage statements>;
         time_windows <time window specifications>;
```

This declares the items which make up the expansion of an operator. These items include the actions of the expansion, the conditions which must be satisfied for the actions to be executed, ordering information between actions, resource usage/specification information and time windows during which specified actions must be executed.

Again the information specified within these fields differs significantly between the two languages.

O-Plan2
O-Plan2 assumes actions can occur in parallel unless instructed differently by the schema writer. Separate fields are used to describe the major components of the decomposition.

1. **nodes** field specifies the decomposition of the schema in terms of its actions and any dummy nodes for convenience.

2. **ordering** field specifies the explicit ordering information known by the user between the actions of the decomposition.

3. **conditions** field specifies the conditions which need to satisfied at points in the operator decomposition and their type. The type allows the planner to identify the tactics open to it when trying to satisfy the condition. A full list of condition types in given in Section 10.4 of the Implementation Manual.

4. **resources** field specifies the resource specification and usage which takes place at points within the decomposition

5. **time_windows** field specifies the time windows during which certain actions of the decomposition must be executed.

SIPE-2

Unlike O-Plan2 which uses six separate slots to specify a schemas decomposition, SIPE-2 uses a single `PLOT` field which breaks down into a series of subfields which specify the action decomposition, the goals which need to be achieved, the ordering information between actions, resource usage and time window information. SIPE-2 requires the schema writer to explicitly declare which parts of the schema can be executed in parallel **and** which must be executed in sequence. The syntax does not allow any arbitrary ordering to be represented. However, any SIPE-2 ordering can be represented in O-Plan2. In the following example, two actions `deter-border-incursion` and `provide-territorial-defence` are declared as separate branches of a parallel decomposition.

```
PLOT:
  PARALLEL
  BRANCH 1:
    PROCESS
      ACTION:    deter-border-incursion;
      ARGUMENTS: coa1;
      EFFECTS:   (border-incursion-deterred coa1);
  BRANCH 2:
    PROCESS
      ACTION:    provide-territorial-defence;
      ARGUMENTS: coa1;
      EFFECTS:   (territory-defence-provided coa1);
 END PARALLEL
END PLOT
```

The SIPE-2 schemas also introduce `GOALS` into the network which act as place holders for future work. The semantics of a goal are that a condition needs to be established and it is then protected from the point at which it is established to the start of the first action of the decomposition. In the example below the condition `(deployed army1 urban1 end_time1)` needs to be asserted by some previous action and must hold true until the start of `traverse-terrain`.

In the following example the same information is declared using the two different languages.

```
SIPE-2: PLOT:
         GOAL: (deployed army1 urban1 end_time1)
          PROCESS:
            ACTION:    traverse-terrain;
            ARGUMENTS: army1, urban1, urban2;
            EFFECTS:   (ground-moved army1 urban1 urban2)
          PROCESS:
            ACTION:    occupy-key-terrain;
            ARGUMENTS: army1, coa1, urban2;
            EFFECTS:   (key-terrain-occupied army1 urban2 coa1);
         END-PLOT
```

```
O-Plan2: nodes  1 action {traverse-terrain ?army1 ?urban1 ?urban2},
                2 action {occupy-key-terrain ?army1 ?urban2 ?coa1};

         orderings 1 --> 2;

         effects {ground-moved ?army1 ?urban1 ?urban2} = true at 1,
                {key-terrain-occupied ?army1 ?urban2 ?coa1} = true at 2;

         conditions achieve {deployed ?army1 ?urban1 ?end_time1} at begin_of 1;

         time and resource statements can also be given in O-Plan2
```

2.4 Domain Rules

Domain rules allow a planning system to identify new contextual information about a domain which is usually impossible to represent within single schemas. For example, in a block stacking domain, the operator `move` can only assert a block is clear after it has been executed iff a block can support [1] at most **one** other block. If the blocks are of differing sizes then asserting the block is clear after the move is incorrect. This could be achieved by having several different schemas which are tailored to specific situations. For example, `move_1_leaving2`, `move_1_leaving3`, etc but that would be highly inefficient.

An alternative way of dealing with contextual effects is to have rules which infer possible effects of actions. From the example above, if there are no patterns of the form {`on ?x b`} (where `?x` is a free variable) then the planner can assert {`clear b`} This is achieved in SIPE-2 by means of a `state rule` as follows:

```
STATE-RULE: dclear
ARGUMENTS: object1, object2, object3 CLASS EXISTENTIAL
TRIGGER: ~(on object1 object2)
CONDITION: ~(on object3 object2)
EFFECTS: (clear object2)
END STATE-RULE
```

An alternative type of domain rule is a `CAUSAL-RULE` which allows the planner to deduce causal effects. For example, if a block is moved then it is no longer where is was!!! This can be achieved in SIPE-2 by means of a causal rule as follows:

```
CAUSAL-RULE: noton
ARGUMENTS: object1, object2, object3
TRIGGER: (on object1 object2)
```

[1] Support in the sense its top surface area is equal to one block. This still allows towers of blocks to be built!!!

```
PRECONDITION: (on object1 object3)
EFFECTS: ~(object1 object3)
END CAUSAL-RULE
```

In the current O-Plan2 TF language there is no implemented mechanism to represent either `STATE-RULES` or `CAUSAL-RULES`, although the `domain_rules` TF statement has been defined to allow the exploration of such issues within the O-Plan2 framework. The main problem which will have to be considered when creating such a mechanism for O-Plan2 is that the trigger conditions might need to be integrated from possibly disparate parts of the plan state. SIPE-2 considers only a single trigger condition on each rule and as such does not run up against this integration problem. A simple example of where more than one trigger may be required is as follows. Consider a door with two locks both of which must be turned together for it to open. By turning one lock and then the other the door will **not** open. SIPE-2 would find this impossible to model.

2.5 Type and Object Hierarchy

The type and object hierarchy allows objects within the target domain to be declared together with their attributes and requirements. To describe the different mechanisms employed in the SIPE-2 DDL and TF a specific example will be used. Suppose the target domain includes a set of different vegetables: artichokes, corn, sprouts and mushrooms.

SIPE-2
The vegetables would be described as follows:

```
CLASS: vegetables
PARENT-CLASS: FOOD
SUBCLASSES: artichokes, corn, sprouts, mushroom
END CLASS

CLASS: artichokes
PARENT-CLASS: vegetables
PROPERTIES:
  STEAM-TIME: 45;
INSTANCES: artichoke1, artichoke2
END CLASS

OBJECT: artichoke1
PARENT-CLASS: artichokes
PROPERTIES:
  COLOUR: green
  SIZE: 5
END OBJECT
```

O-Plan2

At present the TF language does not support such a hierarchy but instead relies in a "flat grouping" of objects, a series of `always` statements[2] and a set of matching functions referred to as Actors. The vegetables could be described as follows:

```
types     vegetables = {artichokes corn sprouts mushroom},
          artichokes = {artichoke1 artichoke2};

always    {steam_time artichokes} = 45,
          {colour artichoke1} = green,
          {size artichoke1} = 5;
```

The association of `artichoke1` to `artichokes` and `artichokes` to `vegetables` is made at the schema level by means of a `type` actor. For example, a schema could describe a variable as being `green` and of type `artichoke` as follows:

```
  vars ?vegetable = ?{and ?{type artichokes}
                          ?{hassupitem 2 {colour ??} green}};
```

It is therefore possible to build a hierarchy of types as in SIPE-2 from existing actor functions. However, its can get extremely convoluted and as such may only be useful in simple examples. The SOCAP domain could be coded in this way but the efficiency of such a representation will need to be investigated. This may result in a requirement for a proper hierarchical object structured store with full inheritance and this should be investigated as an alternative to the use of the scheme outlined above.

The SOCAP domain does not include any resource reasoning although several `resource` statements are present but have been commented out. They appear to be *fixed unit resources*, i.e. they are used within a certain action and then returned to a central "pool". All of the named resources were either armies, navies or airforces. The aim of declaring such resources usages was to stop the resources being allocated to other actions during the time period of allocated action.

To show the possible advantages of O-Plan2 over SIPE-2 in the area of resource reasoning we would need to have a problem which contained realistic resource usage.

2.6 Task and Problem Descriptions

The function of the task description is to introduce the task or goals which the planner is required to solve. The conversion is relatively easy to automate and involves combining the two SIPE-2 DDL entries `PREDICATES:` and `PROBLEM` into the single `task_schema` structure as used in TF. However, this scheme does not differentiate between always facts which can never be refuted by plan actions (referred to in TF as `always` statements) and the rest of the domain

[2] An `always` statement is one which cannot be refuted by the effects of any actions within the domain.

facts. For example, in a block stacking domain the `table` is always assumed to be clear and is thus always available as the destination of a `move` action.

The following example shows the conversion of a simple block stacking task description from SIPE-2 DDL to TF.

```
  PREDICATES
  (on c a) (on a table) (on b table) (clear b) (clear c) (clear table)
  END PREDICATES

  PROBLEM: prob4
  PARALLEL:
   BRANCH: 1
     GOAL: (on a b)
   BRANCH: 2
     GOAL: (on b c)
  END PARALLEL END PROBLEM
```

This should be translated to:

```
always {clear table};

schema task_stack_ABC;
  nodes 1 start,
        2 finish;
  orderings 1 ---> 2;
  conditions achieve {on a b} = true at 2,
             achieve {on b c} = true at 2;
  effects {on c a} at 1,
          {on a table} = true at 1,
          {on b table} = true at 1,
          {cleartop c} = true at 1,
          {cleartop b} = true at 1;
end_schema;
```

3 Simple Conversion

The aim of this section is to describe the conversion of a plan operator schema written in the SIPE-2 DDL to the O-Plan2 TF language. Two examples are given and these are as follows:

1. **Example Description**
   A simple `cleartop` operator taken from a block stacking domain. This operator clears the top of block `?x` by moving the object `?y` to a new place `?z`. The preconditions required are that both `?y` and `?z` are clear, i.e. they have no other block on top of them before the move is executed.

3.1 Description of the Block Stacking Example

The aim of this section is to present a simple example of the conversion of the `cleartop` operator from the SIPE-2 DDL to O-Plan2 TF. The schema to be converted in shown in Figure 1.

```
OPERATOR: cleartop
 ARGUMENTS: object1, object2 IS NOT object1, block1
 PURPOSE: (clear object1)
 PRECONDITIONS: (on block1 object1)
 PLOT:
  PROCESS
   ACTION: puton
   ARGUMENTS: block1, object2
   EFFECTS: (on block1 object2)
   PROTECT-UNTIL: (clear object1)
   GOAL: (clear object1)
 END PLOT
END OPERATOR
```

Figure 1: SIPE-2 cleartop operator

The description of the conversion will now be described in a series of steps:

1. **Operator name**
   The name of the operator can be simply copied across into the <name_field>> of the schema and a corresponding end_schema added to define its end.

```
schema cleartop;
end_schema;
```

2. **Variable Declarations**
   The three variables introduced in the schema namely object1, block1 and object2 need to be declared together with their type. A type is only strictly needed by O-Plan2 if the variable will not always be bound after schema selection. As a result it is always better to provide a value for its type. The assumption in this example is that all blocks are members of the class objects. A constraint is posted in the declaration statement that two of the objects must be different. This is dealt with means of a vars_relations statement.

```
types objects = (a b c table);

schema cleartop;
  vars  ?block1 = ?{type objects},
        ?object1 = ?{type objects},
        ?object2 = ?{type objects};
  vars_relations ?object1 /= ?object2;
end_schema;
```

3. **Reasoning for using the schema**
   The reason for using this schema is PURPOSE: clear object1 and this can be mapped directly to an only_use_for_effects statement in the new schema.

```
types objects = (a b c table);

schema cleartop;
  vars  ?block1 = ?{type objects},
        ?object1 = ?{type objects},
        ?object2 = ?{type objects};
  vars_relations ?object1 /= ?object2;
  only_use_for_effects {cleartop ?object1}
end_schema;
```

4. **Action Decomposition**
   The plot structure introduces the decomposition of the operator and is equivalent the the nodes structure in TF.

```
types objects = (a b c table);

schema cleartop;
  vars  ?block1 = ?{type objects},
        ?object1 = ?{type objects},
        ?object2 = ?{type objects};
  vars_relations ?object1 /= ?object2;
  only_use_for_effects {cleartop ?object1};
  nodes 1 action {puton ?block1 ?object2};
end_schema;
```

5. **Conditions**
   The conditions are the statements which need to be satisfied for the actions to be executed. In the SIPE-2 schema conditions are introduced in two different ways, through the `GOALS:` and `PRECONDITIONS:` statements.

   (a) The `PRECONDITIONS:` statement describes conditions which need to be `true` before the action can be considered. They filter choices of schema at selection time. These map directly to the `only_use_if` statement in TF.

   (b) The `GOALS:` statement describes conditions which need to be satisfied before the action can execute. The type of this condition may vary from application to application. In this example the type chosen is `achieve` which allows the greatest flexibility in satisfying the condition. The node number is used in the condition instruction to specify the point at which the condition is required.

```
types objects = (a b c table);

schema cleartop;
  vars  ?block1 = ?{type objects},
        ?object1 = ?{type objects},
        ?object2 = ?{type objects};
  vars_relations ?object1 /= ?object2;
  only_use_for_effects {cleartop ?object1};
  nodes 1 action {puton ?block1 ?object2};
  conditions only_use_if {on ?block1 ?object1} at 1,
             achieve {cleartop ?block1} at 1;
end_schema;
```

3.2 Review of the Example Conversion

The conversion is quite easily achieved by a simple line by line substitution form SIPE-2 DDL to TF. However, this does not result in efficient TF and in many cases the resulting TF is poorly written. In this example, a better schema would be as in Figure 2.

In this schema only two variables are introduced `?x` and `?y` denoting the block and the one it supports respectively. Their type is further restricted to be of `movable` objects which stops the

```
schema makeclear;
  vars ?x = ?{type movable_objects},
       ?y = ?{type movable_objects};
  vars_relations ?x /= ?y;
  only_use_for_effects {cleartop ?x} = true;
  nodes 1 action {puton ?y table};
  conditions only_use_if {on ?y ?x} at 1,
             achieve  {cleartop ?y} at 1;
end_schema;
```

Figure 2: Improved TF schema for cleartop

schema being used to clear the table! By ensuring that a block which is moved is always placed on the table rather than another block a "free" space can always be found at no cost in terms of planning. Of course, there could be other considerations of having side effects of achieving other required goals at the same time.

4 Discussion 1: Conditional Iterations

This example is taken from the SIPE-2 for Crisis Action Planning (SOCAP) demonstration and shows an operator which was designed to deter incursions across a national border by hostile land, sea or air units. It includes a parallel loop construct which has no direct correspondence in O-Plan2 TF. The schema is described in Figure 3.

```
OPERATOR: deter-border-incursion
ARGUMENTS: coa1,
           end-time1,army1,route1,
           end-time2,navy1,sea-loc1,
           end-time3,air1,air-loc1;
PRECONDITION: (immed-threat-enemy army1 route1 coa1 end-time1)
              (immed-threat-enemy navy1 sea-loc1 coa1 end-time2)
              (immed-threat-enemy air1 air-loc1 coa1 end-time3);
PURPOSE: (border-incursion-deterred coa1);
PLOT:
PARALLEL
 BRANCH 1:
  PROCESS
   ACTION: PARALLEL-LOOP;
   GOALS: (deter-threat army1 coa1 end-time1);
   PATTERN: (immed-threat-enemy army1 route1 coa1 end-time1);
 BRANCH 2:
  PROCESS
   ACTION: PARALLEL-LOOP;
   GOALS: (deter-threat navy1 coa1 end-time2);
   PATTERN: (immed-threat-enemy navy1 sea-loc1 coa1 end-time2);
 BRANCH 3:
  PROCESS
   ACTION: PARALLEL-LOOP;
   GOALS: (deter-threat air1 coa1 end-time3);
   PATTERN: (immed-threat-enemy air1 air-loc1 coa1 end-time3);
END PARALLEL
END PLOT END OPERATOR
```

Figure 3: An example SOCAP schema

The steps of the conversion are similar to those given in Example 1 and thus only the salient points will be given.

1. Assuming we have constructed a schema to the following point and are considering the conversion of the `PROCESS` statements.

```
schema deter-border-incursion;
  vars ?coa1 = ?(type course_of_action},
       ?army1 = ?{type army},
       ?navy1 = ?{type navy},
       ?air1  = ?{type airforce},
       ?route1 = ?{type route}
       ?sea-loc1 = ?{type sea-locations},
       ?air-loc1 = ?{type air-locations};
  nodes;
  conditions only_use_if (immed-threat-enemy ?army1 ?route1 ?coa1 ?end-time1),
            only_use_if (immed-threat-enemy ?navy1 ?sea-loc1 ?coa1 ?end-time2),
             only_use_if (immed-threat-enemy ?air1 ?air-loc1 ?coa1 ?end-time3);
  end_schema;
```

Each of the PROCESS statements can be executed in parallel and the PARALLEL-LOOP construct allows the schema to retrieve from the current plan state all items which match PATTERN and to assert them as GOALS to be achieved. In this case to make sure any hostile air, sea or land units are deterred.

Review of Discussion 1

At this point do we need a new TF construct to handle the PARALLEL-LOOP construct or should we have a schema which deals with a single aggressor and then posts its self back?

Option 1: New Loop Construct
The loop construct would require the planner to iterate a number of times across a `set` defined by the TF writer. The set would be built up from a set of system defined set manipulator functions and the basic set would be as follows: `add`, `remove`, `union intersection`, `is_member_of`. An example use of such a construct would be as follows:

`iterate refuel over {?intersection {armoured-division} {armoured-cavalry}};`

The advantage of this option would be that it would provide a more flexible representation schema and secondly the need for set manipulation functions has already been identified as a long term goal within the development of the TF language. The disadvantage is that further discussions will be required to identify the exact syntax and semantics of such as construct which will then need integrating into the parser table of the TF compiler as well as new code being written to handle it within the planner itself.

Option 2: Single Schema Re-posting
This could be dealt with by using a schema with an `only_use_if` to identify the aggressors. When all patterns have been dealt with the schema will no longer be applicable and will thus terminate. The advantage of this scheme is that no new TF will need to be defined and a similar approach has been used before on a Naval Replenishment at Sea problem to identify the number of ships requiring refueling and reprovisioning.

The problem with either of these schemes would be that trying to automate the conversion would be a nightmare and as such the best solution may be to leave the `PARALLEL-LOOP:` in place and convert it by hand with one of the above methods.

5 Discussion 2: Use of the Object Hierarchy within Schemas

This operator is again taken from the SOCAP data and shows the use of lookup information from the object hierarchy and its use in restricting the instantiation of variables. The schema which describes the steps to deter an incursion by means of a ground patrol is described in Figure 4.

```
OPERATOR: Deter-border-incursion-by-ground-patrol
ARGUMENTS: army2,coa1,end-time1,
         urban1,urban2 is not urban1,region1,region2,route1,end-time3,
         numerical1 is (thirdsize-firepower army2),
         numerical2 is (mobility army2),
         numerical3 is (size route1),
         numerical4 is (terrain route1),
         army1 with firepower greater than numerical1,
         army1 with mobility greater than numerical2,
         army1 with type-size less than numerical3,
         army1 with terrain-type greater than numerical4;
INSTANTIATE: army1,urban1;
PURPOSE: (deter-threat army2 coa1 end-time1);
PRECONDITION:
         (d-day end-time3)
         (immed-threat-enemy army2 route1 coa1 end-time1)
         (apportioned-forces army1)
         (base-approval urban1 coa1)
         (near-territory urban2 route1)
         (located-within urban2 region1)
         (located army2 region2 coa1)
         (adjacent-territory region2 region1);
PLOT:
  GOAL: (deployed army1 urban1 end-time1);
  PROCESS
        ACTION: traverse-terrain;

        ARGUMENTS: army1,urban1,urban2;
        EFFECTS: (ground-moved army1 urban1 urban2);
  PROCESS
        ACTION: ground-patrol;
        ARGUMENTS: army1,region1,coa1,end-time1;
        EFFECTS: (ground-patrols army1 region1 coa1);
END PLOT END OPERATOR
```

Figure 4: SIPE-2 Schema for Deterring Border Incursion

Review of the Discussion 2

In this schema extensive use of the object hierarchy is made to help restrict variable bindings and to provide domain facts to aid in search control. Some of the facts are static in the sense that they cannot be changed over the course of the plan and others are dynamic. A conversion mapping could be made to the TF statements `compute`, `only_use_if` and `only_use_for_query` statements but there is very little opportunity for this to be carried out automatically.

The main differences which can be picked out are those between facts asserted as, (<variable_name> `is` <value>) and those asserted as (<variable_name> `with` <attribute_value>). For example,

1. `numerical1 is thirdsize-firepower army2`

   states that for the object bound to the variable `army2`, search for its attribute `(thirdsize-firepower army2)` and assign it to `numerical1`.

2. `army1 with firepower greater than numerical1`

   states that the object bound to the variable `army1` must have `firepower greater than numerical1`.

This schema makes the problem slightly easier than would be in the general case by forcing the schema to instantiate the value of `army1`. However, this will not always be the case and as such the dependencies which may build up on object selection could prove to be extremely complicated. For example, in the case where `army1` is not bound then the planner would have to set up dependencies between all possible values given it could be either of `armya`, `armyb`, etc.

The `instantiate` slot of the schema allows the variable(s) specified to be instantiated immediately after the operator is applied. The instantiation is carried out after the `PRECONDITIONS:` are matched and after the `PLOT:` has been inserted, **but** before any deductions are made. This would NOT map directly to the actor **bound** match specification in the current TF actor library (see below).

Using current ideas, the following schema might be applicable.

```
always {size route_A} = 2,
       {terrain route_A} = 4;

type region = {region_a region_b},
     urban  = {region_a_urban_1 region_a_urban_2 region_b_urban_1},
     army   = {army_north army_south army_east army_west},
     route  = {route_A route_B route_C};

The following information could be asserted as part of the initial
facts of the task schema.
```

```
...
effects {firepower army_north} = 5,
        {mobility army_north} = 3,
```

The `vars` and `vars_relations` fields of the schema together within the actions of the schema expansion could be defined as follows:

```
 schema deter_border_incursion_by_ground_patrol;
 vars ?army1 = ?{bound},
      ?army2 = ?{type army},
      ?coa1 = ?{type course_of_action},
      ?urban1 = ?{bound},
      ?urban2 = ?{type},
      ?region1 = ?{type region},
      ?region2 = ?{type region},
      ?route1 = ?{type route},
      ?end_time1,
      ?end_time3;

 vars_relations ?urban1 /= ?urban2;

 nodes action 1 {traverse_terrain ?army1 ?urban1 ?urban2},
       action 2 {ground_patrol ?army1 ?region1 ?coa1};

 orderings  1 --> 2;

 conditions ;

 effects {ground_moved ?army1 ?urban1 ?urban2} at 1,
         {ground_patrols ?army1 ?region1 ?coa1} at 2;

end_schema;
```

The conditions which need to be satisfied together with the variable restrictions could then be defined as follows:

```
conditions compute multiple_answer numerical1 = {thirdsize-firepower ?army2},
                   multiple_answer numerical2 = {mobility ?army2},
                   multiple_answer numerical3 = {size ?route1},
                   multiple_answer numerical4 = {terrain ?route1};
           only_use_if (d-day ?end-time3),
           only_use_if (immed-threat-enemy ?army2 ?route1 ?coa1 ?end-time1),
           only_use_if (apportioned-forces ?army1),
```

```
            only_use_if (base-approval ?urban1 ?coa1),
            only_use_if (near-territory ?urban2 ?route1),
            only_use_if (located-within ?urban2 ?region1),
            only_use_if (located army2 ?region2 ?coa1),
            only_use_if (adjacent-territory ?region2 ?region1),
```

The `compute` function returns the value of the given pattern in the database. In each of the examples above there is a single value for each pattern at a specified point in the plan. The function `thirdsize-firepower` is defined in the SOCAP domain as follows:

```
(defun thirdsize-firepower (x) (* 0.333 (car (get.attr x 'firepower))))

(defun halfsize-air-firepower (x) (* 0.5 (car (get.attr x 'air-firepower))))
```

This is rather scruffy but "it gets the job done". TF does allow external functions to be defined and executed and this could be used for such procedures.

Within the ARGUMENTS: field of the schema are a set of comparisons between the threatening force `army2` and the threatened force `army1`. These could be mapped to a set of actor restrictions on the variable `?army1` as follows:

```
vars ?army1 = ?{and ?{ > ?{property {fire-power ?army1}} ?numerical1}
                    ?{ > ?{property {mobility ?army1}} ?numerical2}
                    ?{ < ?{property {type-size ?army1}} ?numerical3}
                    ?{ > ?{property {terrain-type ?army1}} ?numerical4}
               }
```

A problem which could arise is that of the order of processing within schema selection, i.e. are the values of the variables in the `compute` function i.e. numerical1, numerical2, etc assigned before the check is made against the vars restrictions? The `INSTANTIATE:` merely checks that the variables have a value on the exit of the schema and not on entry to the schema.

The `PRECONDITIONS:` slot can usually be mapped directly to an equivalent `only_use_if` field of TF. The problem is however, that this may be too strong in terms of the tactics allowed to re-achieve a failed precondition. At present if a `only_use_if` is broken then the plan state is poisoned and new plan states searched. In many cases it would be more desirable to use an `only_use_for_query` which would allow the condition to be resatisfied should it become broken.

Some of the facts which are used in the `PRECONDITIONS:` field are used to find a binding and some are used as true schema filters. For example, the conditions `(d-day end_time3)` and `adjacent_territory region1 region2` are used to find bindings for the variables `end_time3` `region1` and `region2` respectively. However, during the course of plan generation and/or execution it would be possible to change the time of `d-day` but unless someone was extremely careless with a nuclear device it would seem highly unlikely that `region1` would ever stop being adjacent to `region2`!! It may be possible to define the `always` and initial effects of task schema first and then try to ascertain which condition type is required by matching against these sets of facts. However, which ever way we decide to tackle this problem it will not be an easy one to deal with automatically.

The condition types `supervised` and `unsupervised` can only be inserted by manual inspection of the condition type and at present I see know way of automating this process.

The `PLOT:` information contains two actions `traverse-terrain` and `ground-patrol` and each of these has an asserted effect. If we chose to make both of the actions primitive (in a TF sense) then it will mean moving both the actions and their effects out of the schema. This might prove a little difficult to achieve automatically in all cases.

The complete schema conversion is described in the following Figure 5.

```
schema deter_border_incursion_by_ground_patrol;
 vars ?army1 = ?{and ?{ > ?{property {fire-power ?army1}} ?numerical1}
                     ?{ > ?{property {mobility ?army1}} ?numerical2}
                     ?{ < ?{property {type-size ?army1}} ?numerical3}
                     ?{ > ?{property {terrain-type ?army1}} ?numerical4}
                },
      ?army2 = ?{type army},
      ?coa1 = ?{type course_of_action},
      ?urban1 = ?{bound},
      ?urban2 = ?{type},
      ?region1 = ?{type region},
      ?region2 = ?{type region},
      ?route1 = ?{type route},
      ?end_time1,
      ?end_time3;

 vars_relations ?urban1 /= ?urban2;

 nodes action 1 {traverse_terrain ?army1 ?urban1 ?urban2},
       action 2 {ground_patrol ?army1 ?region1 ?coa1};

 orderings  1 --> 2;

 conditions compute multiple_answer numerical1 = {thirdsize-firepower ?army2},
                    multiple_answer numerical2 = {mobility ?army2},
                    multiple_answer numerical3 = {size ?route1},
                    multiple_answer numerical4 = {terrain ?route1};
            only_use_if (d-day ?end-time3),
            only_use_if (immed-threat-enemy ?army2 ?route1 ?coa1 ?end-time1),
            only_use_if (apportioned-forces ?army1),
            only_use_if (base-approval ?urban1 ?coa1),
            only_use_if (near-territory ?urban2 ?route1),
            only_use_if (located-within ?urban2 ?region1),
            only_use_if (located army2 ?region2 ?coa1),
            only_use_if (adjacent-territory ?region2 ?region1);

 effects {ground_moved ?army1 ?urban1 ?urban2} at 1,
         {ground_patrols ?army1 ?region1 ?coa1} at 2;

end_schema;
```

Figure 5: Completed Schema Conversion for Deterring Border Incursion

6 Appendices

The following appendices contain information concerning:

- **Appendix 1**:
  provides a list of materials used in the preparation of this report
- **Appendix 2**:
  provides a list of points raised during the conversion of SIPE-2 DDL to TF which require further consideration

6.1 Appendix 1: Support Materials

This appendix provides a list of materials used in the preparation of this report together with extra material which has been obtained from the DARPA/Rome Laboratory Knowledge-based Planning and Scheduling Initiative.

1. *Using the SIPE-2 Planning System*, Version 3, 4th June 1992
2. *SIPE-2 Language Reference Manual*, 19th March 1992
3. On-line SOCAP domain description, Imported 19th February 1992, all files dated 31st August 1992
   - README-index of files available in the on-line copy
   - `classes`: SOCAP classes of objects
   - `newclasses`: extra SOCAP classes and objects
   - `oprs`: SOCAP operators
   - `preds`: SOCAP predicates, includes the capabilities of forces and force descriptions
   - `newpreds`: extra SOCAP predicates
4. Hardcopy SOCAP domain description, Received 24th August 1992. This material covers two main areas:
   - SOCAP diagrams for the IFD-92 run through
   - Sample screen images for the IFD-92 run through

6.2 Appendix 2: Future Requirements for SOCAP

This appendix provides a listing of the major differences between SIPE-2 DDL and TF. A more detailed description of the differences and a discussion as to how these *gaps* may be bridged is given in Sections 4 and 5.

The main differences can be categorised as follows:

1. Conditional Iteration
   SIPE-2 DDL has a `PARALLEL-LOOP` construct which allows the schema writer to **iterate** around a loop where the number of iterations is controlled by the presence of a `pattern` in the plan state. For example, in deterring a border incursion the planner should dispatch a number of units equal to the number of aggressor units. At present there is no support in O-Plan2 for this type of construct.

2. Object Hierarchy and Schema
   SIPE-2 DDL makes extensive use of its class and instance hierarchy to provide information for schema variable matching and schema filtering. A class and instance hierarchy could be constructed using the current TF supported by O-Plan2 and by using the currently defined condition types supported by O-Plan2. These could be used only as a *stop gap* measure until a full CLOS type module has been defined and implemented for O-Plan2. At present there is no support in O-Plan2 for a CLOS type data structure.

3. Domain Rules
   At present there is no support within the current O-Plan2 system for the types of domain rule which can be supported by the SIPE-2 DDL. The SOCAP domain contains only **two** rule of the type `causal` which have only one trigger pattern.

4. Compute Conditions and Actor Support
   The `compute_condition` statement and the actor `property` will be required to implement part of the schema variable matching capability which is available in the SIPE-2 DDL. As present there is no support in O-Plan2 for either of the items.

5. OR Condition Placed on Operator Preconditions
   At present SIPE-2 allows an OR condition to be placed on two or more preconditions of a operator schema. For example:

   ```
   (or (unit-deters-immed-threat army1 army2)
       (apportioned-forces army1))
   ```

   If **either** of the above conditions are satisfied then the schema can be used. At present TF does not allow for such a construct but the same functionality can be achieved by having **two** separate schemas which are identical except that one contains the first precondition and the second contains the other precondition. The drawback to this approach is that the number of schemas becomes excessive and difficult to maintain consistency accross what are essentially the same schema.