O-Plan

Task Formalism Manual

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

January 31, 1997

Version 3.1

Acknowledgements
The O-Plan project began in 1984. Since that time the following people have participated: Colin Bell, Ken Currie, Jeff Dalton, Roberto Desimone, Brian Drabble, Mark Drummond, Anja Haman, Ken Johnson, Richard Kirby, Glen Reece, Arthur Seaton, Judith Secker, Austin Tate and Richard Tobin.

Prior to 1984, work on Interplan (1972–4) and Nonlin (1975–6) was funded by the UK Science and Engineering Research Council and provided technical input to the design of O-Plan.

From 1984 to 1988, the O-Plan project was funded by the UK Science and Engineering Research Council on grant numbers GR/C/59178 and GR/D/58987 (UK Alvey Programme project number IKBS/151). The work was also supported by a fellowship from SD-Scicon for Austin Tate from 1984 to 1985.

From 1989 to 1992, the O-Plan project was supported by the US Air Force Rome Laboratory through the Air Force Office of Scientific Research (AFOSR) and their European Office of Aerospace Research and Development by contract number F49620-89-C-0081 (EOARD/88-0044) monitored by Northrup Fowler III at the USAF Rome Laboratory.

From 1992 to 1995, the O-Plan project was supported by the ARPA/Rome Laboratory Knowledge Based Planning and Scheduling Initiative through the US Air Force Rome Laboratory through the Air Force Office of Scientific Research (AFOSR) and their European Office of Aerospace Research and Development by contract number F49620-92-C-0042 (EOARD/92-0001) monitored by Northrup Fowler III at the USAF Rome Laboratory.

From 1995 to 1998, the O-Plan project was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under grant number F30602-95-1-0022.

Additional resources for the O-Plan and O-Plan projects have been provided by the Artificial Intelligence Applications Institute through the EUROPA (Edinburgh University Research on Planning Architectures) institute development project.

From 1989 to 1993, research on scheduling applications of the O-Plan architecture was funded by Hitachi Europe Ltd. From 1989 to 1992, the UK Science and Engineering Research Council (grant number GR/F36545 – UK Information Engineering Directorate project number IED 4/1/1320) funded a collaborative project with ICL, Imperial College and other partners in which the O-Plan architecture was used to guide the design and development of a planner with a flexible temporal logic representation of the plan state. A number of other research and development contracts placed with AIAI have led to research progress on the O-Plan prototype.

Contact Information
The O-Plan project team can be contacted as follows:

O-Plan Team
Artificial Intelligence Applications Institute
The University of Edinburgh
80, South Bridge
Edinburgh EH1 1HN
United Kingdom

Tel: (+44) 131 650 2732
Fax: (+44) 131 650 6513
Email: oplan@ed.ac.uk
WWW: http://www.aiai.ed.ac.uk/ oplan/

Created:	January 16, 1992 by Austin Tate
Last Modified:	January 6, 1997 (10:00) by Jeff Dalton
Printed:	January 31, 1997

IMPORTANT NOTE

This Task Formalism Manual presents a comprehensive view of a language for describing planning domains, activities, processes, tasks and plans. It acts as a design focus for work on the O-Plan project and research into planning domain modelling.

The current implementation of the O-Plan planner does not accept all of the Task Formalism constructs documented here. A list of unsupported features and features under investigation is given in section 11.6.

Contents

1 Introduction to Task Formalism

Domain representation in planning attempts to capture the detailed description of permissible actions or operations within an application area, including information about the effects of actions, conditions on the use of actions, and how such conditions should be satisfied. The need to describe such a wide range of information has led to the specification and development of a high level action description language called Task Formalism, or more conveniently TF. TF originated in the Nonlin planning system but has been refined and extended for action descriptions within the O-Plan Planning System being developed at AIAI.

TF is *not* intended as the normal mode of interaction with the user describing a domain. It is intended as an *intermediate language* which fits between a supportive user interface and the planner. TF can be considered to be the target language for a helpful domain writer's support tool. However, in the present release of the O-Plan system, it is necessary to use the TF language directly. TF has also been designed to allow a useful level of compile time checking to be performed.

TF is used to give an overall hierarchical description of an application area by specifying the possible activities within the application domain and describing how those activities can be "expanded" into sets of sub-activities with ordering contraints imposed. Plans are generated by choosing suitable expansions for activities in the plan (i.e. refining those activities) and including the sets of more detailed sub-activities described by the chosen expansions. Ordering constraints are then satisfied to ensure that asserted effects of some actions satisfy, and continue to satisfy, conditions on the use of other actions. Other constraints, such as a time window for the action and restrictions on resource usage, are also included in the descriptions. These descriptions of actions form the main structure within TF—the **schema**. Schemas are also used in a completely uniform manner to describe **tasks** set to the planning system, in the same formalism. Other TF structures hold global information of various sorts and heuristic information about preferences for choices to be made during planning.

The purpose of this manual is to introduce the potential TF writer to the constructs of the language in order to describe application domains for the O-Plan planning system. TF, like high level computer languages, is compiled by the TF Compiler into an internal data structure representing the Domain Information. The Compiler does some error checking, but it is advisable to have a thorough understanding of TF before starting out.

The TF descriptions included below are presented in a mixed reference/rationale form where the specification of TF structures are associated with explanatory notes.

TF is still the subject of research and development. There is no guarantee that forward compatibility of any TF form or component will be preserved across new releases of O-Plan. This manual includes an extensive domain description language which goes beyond the specific features that can be supported by the current implementation of O-Plan. Section 11.6 describes the current implementation in terms of unsupported features and features which are anticipated for the future.

2 Hierarchical AI Planning Systems

Many knowledge-based planners—including the O-Plan Planning Agent—perform a similar procedure to develop a plan. The process is summarised in Figure 1.

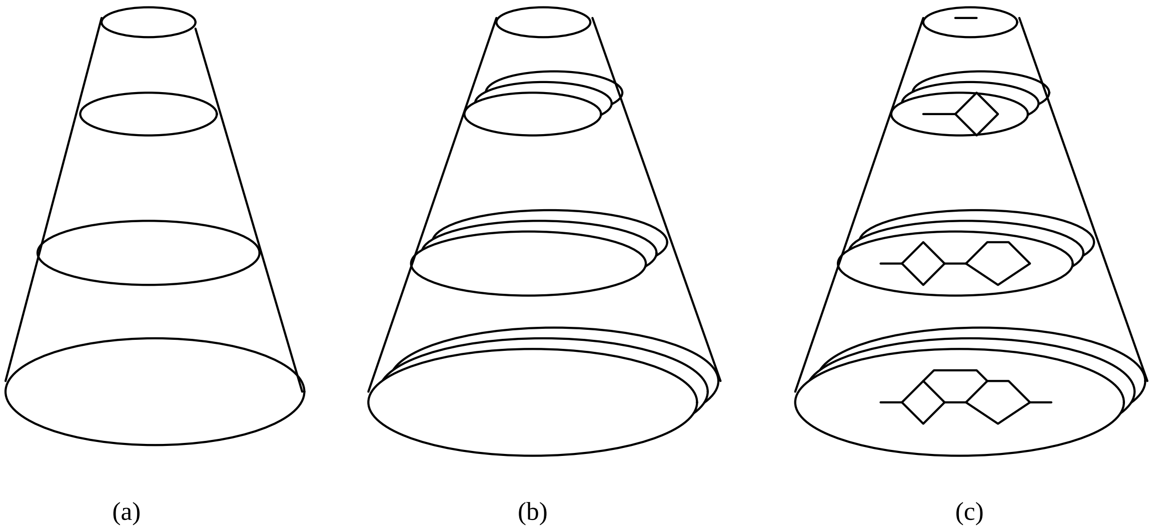


Figure 1: Outline of the operation of an AI Planner

Figure 1 is a simplified overview of a "Hierarchical non-linear AI planner". The three parts show:

a) Based on a hierarchical representation of the plan, a task in the form of a skeleton plan or a set of requirements can be given and expanded out to greater levels of detail.

b) The planner searches through alternative methods of expanding high level plans to lower level ones (and alternative means of satisfying conditions, choosing objects, etc). Interactions between solutions to different parts of the plan are detected and corrected.

c) At each level, the plan is represented as a network of nodes in a form that allows the use of knowledge about the problem (such as time and resource constraints) to restrict the search for a solution.

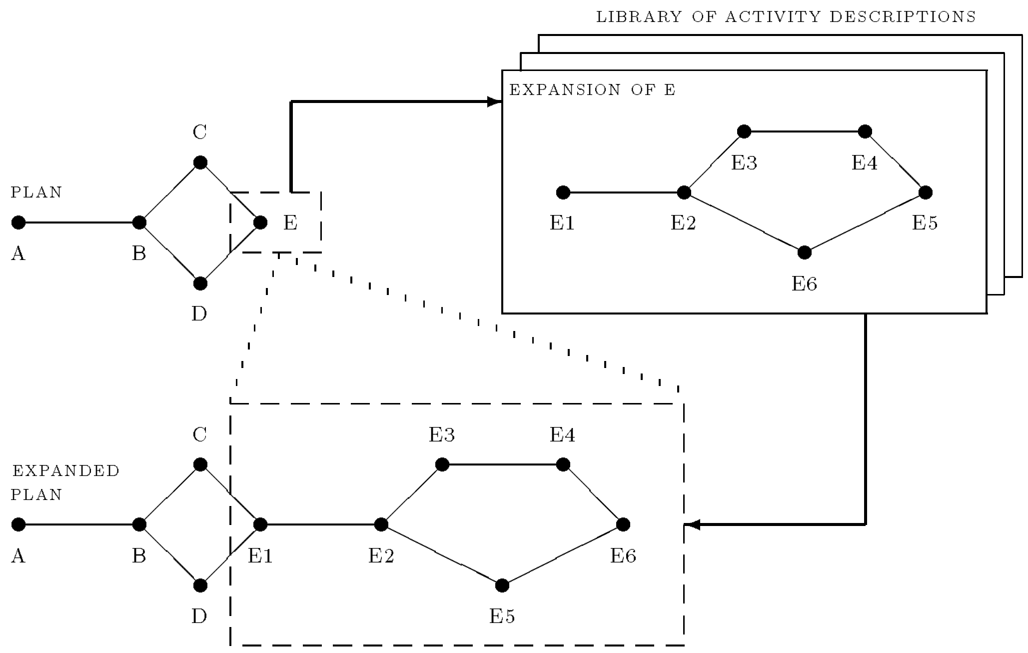


Figure 2: Expanding a step of the plan

Part (b) is often called "expanding" the plan. As shown in figure 2, some high level representation of a step in the plan is selected and a more detailed method of carrying it out is found in a "library" of such activity breakdowns (these are termed *schemas*). The expansion is then inserted into the plan and any unresolved problems (such as unsatisfied conditions, actions needing further expansion, resource requirements, etc) are noted.

3 Some Key Concepts

3.1 Actions and Events

Planning takes place in an environment where certain things are under the explicit control of one or more *agents* responsible for completing one or more *tasks* in the domain.

Actions are the activities which can be performed under the control of the agents and which can alter the environment in which the agents perform their tasks.

Events are those activities outwith the explicit control of the agents.

Actions and events can be described at a number of levels of detail in a hierarchical fashion by being shown to be composed of a number of sub-actions or sub-events ordered in some given way. *Primitive* actions and events cannot be further decomposed.

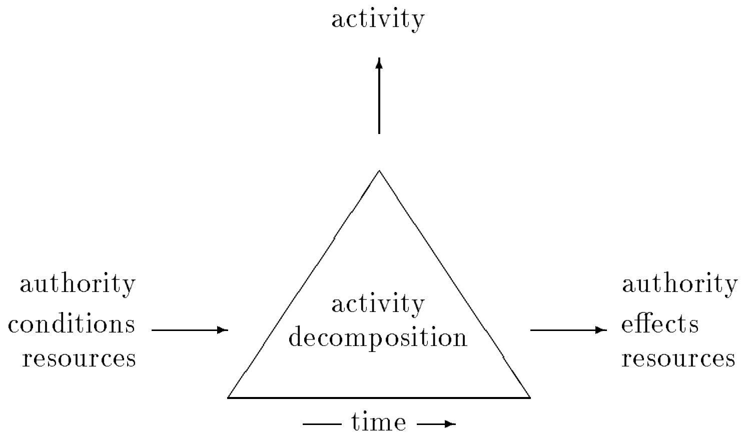
3.2 Schemas

Schemas are used to describe actions and events and where applicable their decomposition to a more detailed level of description. There are a number of different types of schema: normal (or action) schemas, process schemas, task schemas and meta-schemas.

A normal (action) schema describes an action under the deliberate control of the plan. A process schema describes an event or process comprising several linked events which is outwith the direct control of the agent (though actions performed by the agent may result in such events or processes being triggered). A task schema describes a task which the planner is being asked to perform and may describe the environment in which the task is to take place (in terms of the world model and the known events that will occur in it).

A meta-schema is a simple means to allow for families of similar schemas of any of the three types above to be described. Meta-schemas are used at TF compile time.

A schema can be pictured of as a triangle showing two ways in which the schema can be utilised. One, pointing from the top of the triangle, relates to its use as a means of refining, decomposing or expanding an action or event to a lower level of detail. The right hand (output) side relates to its possible use to provide an effect to satisfy a condition (or to provide additional resources or authorities required for use in a plan). The third left hand (input) side of this triangle is the applicability conditions for the schema (or the resources or authority it requires).



3.3 Authority

The O-Plan planner is intended to operate in a distributed command, planning and execution environment. In such an environment, the authority provided to a planner to create a plan or to modify it, and to an execution agent to allow execution or adaptation of a specific chosen plan, needs to be made clear.

O-Plan currently supports only a simple implementation of authority management. However, the basis of a more sophisticated scheme is allowed for with the following:

- the notion of separate *plan options* which are individually specified task requirements, plan environments and plan elaborations. The Task Assignment agent can create as many as required. The plan options may contain the same task[1] with different search options or may contain a different task and environmental assumptions. It is possible to have only one plan option as the minimum[2].
- the notion of *plan phases*. These are individually provided actions or events stated explicitly in the top level task description given by the Task Assignment agent. Greater precision of authority management is possible by specifying more explicit phases at the task level. It is possible to have only one "phase" in the task as the minimum[3].
- the notion of *plan levels*. Greater precision of authority management is possible by specifying more explicit levels in the domain Task Formalism (TF). It is possible to have only one "level" in the domain as the minimum.

[1] Multiple conjunctive tasks specified together is also possible.

[2] Plan options may be established and explicitly switched between by the Task Assignment agent.

[3] In fact any sub-component of any task schema or other schema included by task expansion in a plan can be referred to as a "phase" within the O-Plan planner agent. This can be done by referring to its node number.

- for each "phase", planning will only be done down to an authorised "level" at which point planning will suspend leaving appropriate agenda entries until deeper planning authorisation is given.
- execution will be separately authorised for each "phase".

The planner agent will only need to be able to refer to *numbers* for plan options, phases and levels. Domain related names that are meaningful to the user may be associated with these numbers through the Task Assignment agent.

3.4 Effects and Conditions

At the heart of the O-Plan plan representation for effects and conditions are the TOME (Table Of Multiple Effects) and GOST (GOal STructure)—tables to record all effects generated by actions in the plan, the conditions satisfied at points in the plan, and the intentions behind the ways in which conditions have been achieved. The GOST is the means by which the scope of an effect (there could be several alternative such effects) which satisfies a condition can be recorded and used to protect against the introduction of interacting effects. In a final valid plan all conditions introduced during planning are required to have been satisfied and maintained over the required period, hence there will be a valid GOST entry for each condition encountered.

Conditions are one of the most elaborate of all TF statements due to the variety of condition types identified as being useful in O-Plan. The main types are:

only_use_if A filter on the relevance of the schema based on a statement in the environment which it is not anticipated will be altered during the required range.

Normally used to filter out non-applicable schemas.

In the Nonlin planner, these were called **usewhen** or **holds** conditions.

only_use_for_query A condition anticipated as being satisfied in the environment.

Normally used to bind variables appearing in the condition.

supervised A condition established by (one or more alternative nominated) substep(s) of the schema's decomposition.

Normally used to protect conditions across time intervals within a schema.

unsupervised A condition which is anticipated as being established elsewhere in a plan in which this schema is used.

Normally used to order steps in a plan to meet sequencing requirements.

achieve A condition which may be satisfied by any means available to the planner (including adding new plan structure).

compute These conditions provide the *O-Plan External System Interface*. They are not conditions satisfiable directly from effects within a plan. A **compute** condition describes a requirement which can be satisfied using information from an external system (or database or user).

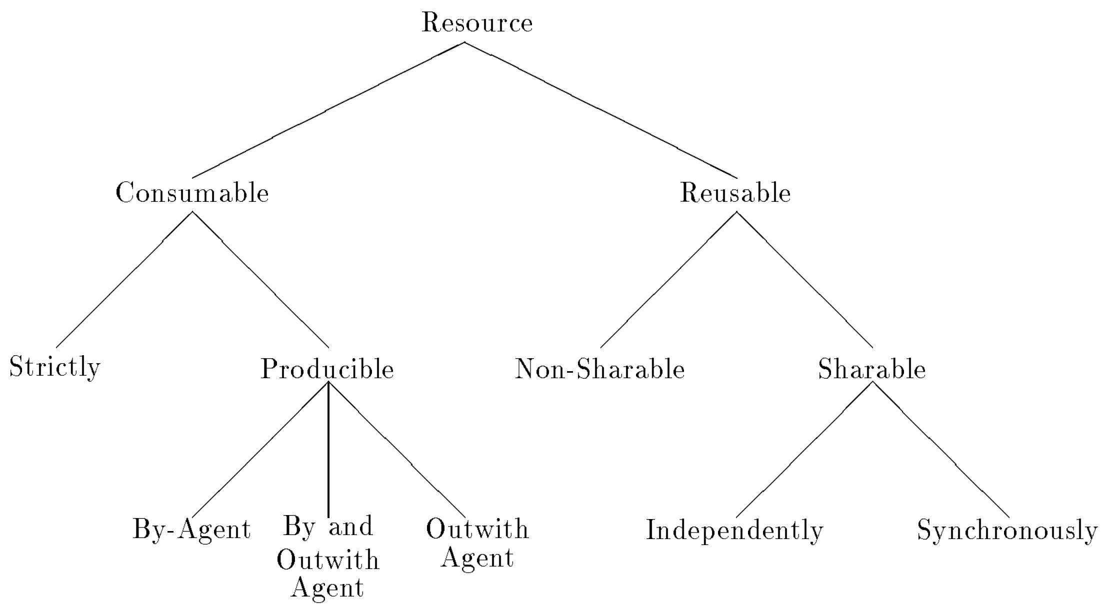
Other condition types can be identified but the ones above have been found to be useful ways to extract knowledge from a domain writer in a form that can be used to restrict search in an AI planner. The control of planner search via condition types is worthy of a serious study in its own right, and could form an ideal Ph.D. topic.

Condition typing allows information to be kept about when, how and why a condition present in the plan has been satisfied and the way it is to be treated if the condition cannot be maintained. However, use of this information itself will almost certainly commit the planner to prune some of the potential search space thereby losing completeness of search if the TF writer uses an inappropriate condition type. Unfortunately this puts a burden on the domain writer and can make domain writing a difficult job.

Condition typing helps direct the planning process, but it also requires the domain TF writer to structure the hierarchy of the tasks or actions more clearly (usually using separate *levels*). It forces checks to be made on processes or actions which should communicate with others to ensure that they actually do advertise their results through a common vocabulary.

3.5 Resources

The types of resource used in a plan can be classified as shown here.



consumable_strictly: A set amount of the resource is available and cannot be topped up.

consumable_producible_by_agent: The resource can be topped up from actions for agents under the control of the plan.

consumable_producible_outwith_agent: This resource type is similar to consumable_producible_by_agent except that extra resource is only obtained via an off-line process such as a delivery rather than via actions of agents under the control of the plan.

consumable_producible_by_and_outwith_agent: This resource is a combination of the two above. Resources can be produced both by agent actions and by off-line processes.

reusable_non_sharable: The resource is allocated from a "central pool" in unit amounts and when the resource is finished being used it is then deallocated back to the pool. For example workman, robots, lorries, etc.

reusable_sharable_independently: The resource can be shared without coordination to specific time periods, e.g. spaces in a car park.

sharable_synchronously: The resource is shared for a specific time, e.g. capacity in a particular journey of a ship or cargo plane.

3.6 Patterns

A ⟨pattern⟩ is a TF component used throughout the Task Formalism. A ⟨pattern⟩ can be a constant name beginning with a letter and followed by alphanumerical characters and one or two other permitted characters (e.g. abc, part_32), a number (e.g. 3, 5678, 3.142), or several of these things surrounded by the braces { and } to any depth (e.g. {on a b}, {30 10 60}, {colour {camera_1 filter_3}}), or set brackets (and) to any depth (e.g. (a (b c) d)). These are all *fully instantiated* ⟨pattern⟩s.

A ⟨pattern⟩ may also contain variables which have the form ?⟨name⟩, e.g. ?colour. Such patterns containing variables are often used to state the conditions and effects of actions in the domain.

A ⟨pattern⟩ can also represent a *pattern specification* by giving a template of pattern forms that can match it. All match constraint specifications start with the character "?". The general "match anything" specification is ??. Other match constraints include ?{**not** green}, ?{**bound**}, etc. For example, {on ?? ?{**not** table}} is a valid specification for {on a b} but not for {on a table}.

3.7 ⟨pattern⟩ = ⟨value⟩

A basic building block within O-Plan is the ⟨pattern⟩ = ⟨value⟩ form which is used throughout O-Plan to record information of the form:

```
function(param, ...) = value
```

such as in effects and conditions. An example is:

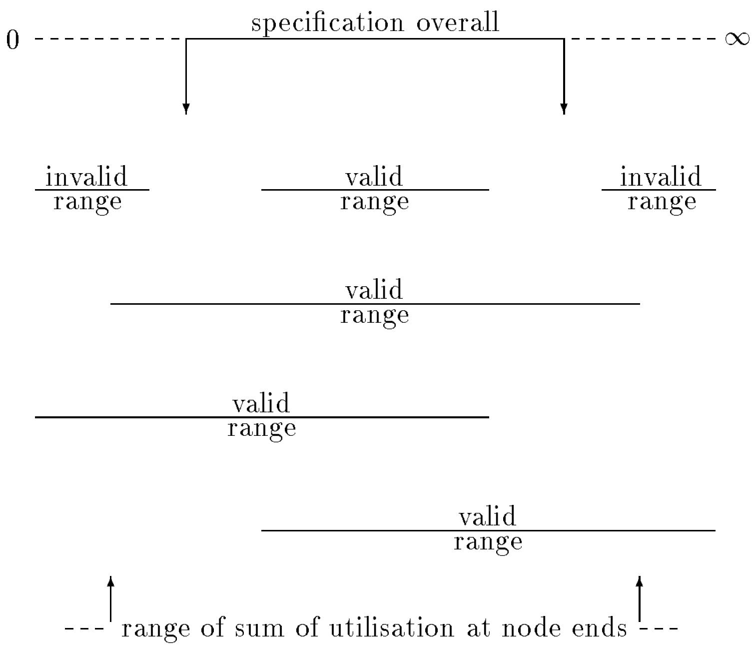
```
{colour camera_1 filter_3} = red
```

where the first *fixed* word on the left side of the = is the function and subsequent entries on the left side are the parameters. The value is usually a single name or number, but can also be a general pattern where appropriate (e.g. {20 30 15} to represent a 3-dimensional spatial coordinate). The ⟨pattern⟩ = ⟨value⟩ form can be fully instantiated as in the example above or may contain match constraints as mentioned earlier.

3.8 Min/Max Pairs for Time Windows and Resource Usage

All time windows and resource usage specifications in O-Plan are maintained as **min/max** pairs, specifying the upper and lower bounds known at any time. Such bounds may be defined by symbolic expressions which can depend on variables, but O-Plan maintains a numerical pair of bounds for all such numerical values. In fact, a third entry is associated with such numerical bounds. This third entry is a *projected* value (which could be a simple number or a more complex function, data structure, etc.) used by the planner for heuristic estimation, search control and other purposes.

Time windows play an important part in O-Plan in two ways. Firstly, as a means of recording time limits for the start and finish of an action, for its duration and for delays between actions. Secondly, during the planning phase itself as a means of pruning the potential search space if temporal validity is threatened.



Similarly, resource usage specifications are used to ensure that resource usage stays within the bounds indicated. There are two types of resource usage statements. One gives a *specification* of the **overall** limitation on resource usage for a schema (over the total time that the schema's expansion can span). The other type describes actual resource *utilisation* **at** points in the expansion of a schema. It must be possible (within the restriction on the ranges in the actual resource usage statements) for a point in the range of the sum of the resource usage statements to be within the overall specification given.

3.9 Notepad

A *Notepad* is associated with a plan in order that *global* information about the plan can be recorded on it. Conditions can also be stated with respect to the contents of the Notepad and schemas can be selected to achieve entries on the Notepad. Notepad effects are known as *Notes*. Normally, effects are asserted at a specific point in the plan network and conditions are stated as needing to be satisfied at specific points. The Notepad facility provides a means to record plan information which is not specific to any component of the plan.

For example, notes can be used to record information about the strategy or approach being adopted in the search for a solution, or it may be used to record information about the choices being made. A user may also use the Notepad directly during plan generation when acting in the *Planner User Role* – the Knowledge Source KS_USER normally allows Notes to be recorded and the Notepad to be viewed.

O-Plan1 used the term *jotter* for the Notepad.

3.10 O-Plan External System Interface

The O-Plan External System Interface is provided via **compute** conditions in O-Plan schemas. The external system could be a data base system, a modelling package for the application, a CAD system, a table look-up system or a special interface to the user, for example. The external system is called by a function name and is passed parameters which are instantiated versions of the schema variables referred to in the call expression. The external system can also be an O-Plan support routines. One gives access to the question answering routine which can query conditions at the point in the plan where the compute condition is evaluated. Another allows Notepad entries to be changed.

A protocol for the External System Interface allows the results to be returned to the planner in a form that can be understood. This allows none, one or several alternative results to be returned along with optional dependency statements on the continuing validity of the various results (in a form which the planner can take responsibility to maintain).

New user-provided compute functions can be declared to the planner via a **compute_conditions** statement in the TF for a domain. This ensures that the planner knows how to call a compute condition and how to deal with its results.

4 Commented Examples

Before covering the detail of TF forms and syntax in this manual, two examples will be presented here to give a flavour of the language.

4.1 Blocks World

In this first example the single schema required to describe operations in a simple blocks world is outlined. The world is simple because there is no consideration given to block dimensions, alignment of blocks or to the lifting mechanism. The only prerequisites are that a block has to be clear before it can be moved and that its destination block is also clear (the table is assumed to have clear space always). The single schema is called **puton** to reflect the lift-and-stack nature of the application.

In this particular application the schema is only used to satisfy **achieve** conditions through the use of (only_use_for_)effects, hence the expands pattern merely serves to describe the (primitive) action. This schema also illustrates the use of local schema variables. These variables can be instantiated by a number of means:

- By use of the **expands** or (**only_use_for_**)**effects** with fully instantiated patterns.
- By use of the **only_use_for_query** condition type.
- If the **schema variables** are not fully instantiated during the period of use of the schema then such variables are converted to plan state (i.e. global) variables, to be handled later by an appropriate knowledge source.

```
always {cleartop table};

types objects = (a b c table),
      movable_objects = (a b c);

schema puton;
  vars ?x = ?{type movable_objects},
       ?y = ?{type objects},
       ?z = ?{type objects};
  vars_relations ?x /= ?y, ?y /= ?z, ?x /= ?z;
  expands {puton ?x ?y};                      ;;; the actual action name
  only_use_for_effects
            {on ?x ?y}    = true,
            {cleartop ?y} = false,            ;;; satisfy conditions in plan
            {on ?x ?z}    = false,
            {cleartop ?z} = true;
  conditions  only_use_for_query {on ?x ?z},
```

```
                  ;;; only_use_for_query is used to bind one or more
                  ;;; free variables conditions have value true
              achieve {cleartop ?y},
              achieve {cleartop ?x};
end_schema;
```

The above representation of the Blocks World domain can be used to generate plans from tasks set within appropriate **task** schemas. Here is an example which describes an initial world (as shown in the diagram) and a final world in which Block A is on Block B which is on Block C.

```
task stack_ABC;
  nodes 1 start,
        2 finish;
  orderings 1 ---> 2;
  conditions achieve {on a b} at 2,
             achieve {on b c} at 2;
  effects {on c a} at 1,
          {on a table} at 1,
          {on b table} at 1,
          {cleartop c} at 1,
          {cleartop b} at 1;
end_task;

;;;                                          +---+
;;;                                          | A |
;;;      +---+                               +---+
;;;      | C |            --------->         | B |
;;;      +---+  +---+                        +---+
;;;      | A |  | B |                        | C |
;;; -----+---+--+---+-----                   +---+
;;;         TABLE
```

4.2 House Building

The second example shows more of the features of TF, in particular the use of *typed conditions* to reflect the "sub-contractor" nature of individual schemas. In particular, the schema actions correspond to the activities performed by a possible contractor) and the schema conditions make a statement about the level of commitment of that contractor to that particular condition. For example a supervised condition states categorically that the same contractor will take responsibility for satisfying required conditions on actions within the sub-plan, whereas unsupervised conditions state that the responsibility lies elsewhere but is required to hold at the appropriate time.

This example TF also shows a couple of *time windows* statements and *primitive* schemas, it introduces alternative schema expansion methods, and it deliberately includes an alternative which fails to lead to a valid plan.

```
task build_house;              ;;; top level task schema to initiate planning
  nodes 1 start,
        2 finish,
        3 action {build house}; ;;; this action is refined by the schema below
  orderings 1 ---> 3, 3 ---> 2;
end_task;

schema build;
  expands {build house};        ;;; this expands the top level action
  nodes     1 action {excavate and pour footers    },  ;;; some are primitive
            2 action {pour concrete foundations    },
            3 action {erect frame and roof         },
            4 action {lay brickwork                },
            5 action {finish roofing and flashing  },
            6 action {fasten gutters and downspouts},
            7 action {finish grading               },
            8 action {pour walks and landscape     },
            9 action {install services             },  ;;; some are not.
           10 action {decorate                     };
  orderings 1 ---> 2,  2 ---> 3,  3 ---> 4,  4 ---> 5,
            5 ---> 6,  6 ---> 7,  7 ---> 8;
  ;;; actions 9 & 10 are not ordered wrt other actions - they are in parallel
  conditions   supervised {footers poured       } at 2 from [1],
               supervised {foundations laid     } at 3 from [2],
               supervised {frame and roof erected} at 4 from [3],
               supervised {brickwork done       } at 5 from [4],
               supervised {roofing finished     } at 6 from [5],
               supervised {gutters etc fastened } at 7 from [6],
             unsupervised {storm drains laid    } at 7,
               supervised {grading done         } at 8 from [7];
  ;;; note the unsupervised condition - its satisfaction is outwith
  ;;; the control of this schema but must still be satisfied
  time_windows between 1~11:30:00 and 1~14:30:00 at 2,
               between 1~12:00:00 and 1~14:00:00 at 3;
  ;;; time window examples for start times of actions 2 & 3
end_schema;

schema service_1;
  expands {install services};   ;;; one way of expanding {install services}
  only_use_for_effects {installed services 1};
  nodes     1 action {install drains           },
            2 action {lay storm drains         },
            3 action {install rough plumbing   },
```

```
            4 action {install finished plumbing},
            5 action {install rough wiring     },
            6 action {finish electrical work   },
            7 action {install kitchen equipment},
            8 action {install air conditioning };
  orderings 1 ---> 3,  3 ---> 4,  5 ---> 6,  3 ---> 7,  5 ---> 7;
  conditions   supervised {drains installed       } at 3 from [1],
               supervised {rough plumbing installed} at 4 from [3],
               supervised {rough wiring installed  } at 6 from [5],
               supervised {rough plumbing installed} at 7 from [3],
               supervised {rough wiring installed  } at 7 from [5],
             unsupervised {foundations laid       } at 1,
             unsupervised {foundations laid       } at 2,
             unsupervised {frame and roof erected  } at 5,
             unsupervised {frame and roof erected  } at 8,
             unsupervised {basement floor laid    } at 8,
             unsupervised {flooring finished      } at 4,
             unsupervised {flooring finished      } at 7,
             unsupervised {painted                } at 6;
    ;;; As in the real world this sub-contractor relies heavily on others
    ;;; to prepare things beforehand - see the unsupervised conditions.
end_schema;

schema service_2;
  expands {install services};   ;;; another possible expansion
  only_use_for_effects {installed services 2};
  nodes     1 action {install drains           },
            2 action {install rough plumbing   },
            3 action {install finished plumbing},
            4 action {install rough wiring     },
            5 action {finish electrical work   },
            6 action {install kitchen equipment},
            7 action {install air conditioning };
  ;;; This sub-contractor fails to {lay storm drains}.
  ;;; This will lead to plan failure when this schema is used
  orderings 1 ---> 2,  2 ---> 3,  4 ---> 5,  2 ---> 6,  4 ---> 6;
  conditions   supervised {drains installed       } at 2 from [1],
               supervised {rough plumbing installed} at 3 from [2],
               supervised {rough wiring installed  } at 5 from [4],
               supervised {rough plumbing installed} at 6 from [2],
               supervised {rough wiring installed  } at 6 from [4],
             unsupervised {foundations laid       } at 1,
             unsupervised {frame and roof erected  } at 4,
             unsupervised {frame and roof erected  } at 7,
             unsupervised {basement floor laid    } at 7,
```

```
             unsupervised {flooring finished       } at 3,
             unsupervised {flooring finished       } at 6,
             unsupervised {painted                 } at 5;
  effects {wallpaper on} = false at 5;  ;;; an interaction check
  ;;; Effects can be asserted - this one strips wallpaper.
  ;;; Effects of this form can cause interactions to occur and plan steps
  ;;; to be linearised if a condition {wallpaper on} = true appears in, say,
  ;;; the decorate schema.
end_schema;

schema decor;
  expands {decorate};
  nodes      1 action {fasten plaster and plaster board},
            2 action {pour basement floor             },
            3 action {lay finished flooring           },
            4 action {finish carpentry                },
            5 action {sand and varnish floors         },
            6 action {paint                           };
  orderings  2 ---> 3,  3 ---> 4,  4 ---> 5,  1 ---> 3,  6 ---> 5;
  conditions unsupervised {rough plumbing installed   } at 1,
             unsupervised {rough wiring installed     } at 1,
             unsupervised {air conditioning installed } at 1,
             unsupervised {drains installed           } at 2,
             unsupervised {plumbing finished          } at 6,
             unsupervised {kitchen equipment installed} at 6,
               supervised {plastering finished        } at 3 from [1],
               supervised {basement floor laid        } at 3 from [2],
               supervised {flooring finished          } at 4 from [3],
               supervised {carpentry finished         } at 5 from [4],
               supervised {painted                    } at 5 from [6];
  time_windows between 1~11:30:00 and 1~14:30:00 at 2,
               between 1~12:00:00 and 1~14:00:00 at 3;
end_schema;

;;; Now for completeness a list of primitive actions. Primitives are
;;; defined as having no nodes list and must have an expands pattern.

schema excavate;
  expands {excavate and pour footers};
  only_use_for_effects {footers poured} = true;
end_schema;

schema pour_concrete;
  expands {pour concrete foundations};
  only_use_for_effects {foundations laid} = true;
```

```
end_schema;

schema erect_frame;
  expands {erect frame and roof};
  only_use_for_effects {frame and roof erected} = true;
end_schema;

schema brickwork;
  expands {lay brickwork};
  only_use_for_effects {brickwork done} = true;
end_schema;

schema finish_roofing;
  expands {finish roofing and flashing};
  only_use_for_effects {roofing finished} = true;
end_schema;

schema fasten_gutters;
  expands {fasten gutters and downspouts};
  only_use_for_effects {gutters etc fastened} = true;
end_schema;

schema finish_grading;
  expands {finish grading};
  only_use_for_effects {grading done} = true;
end_schema;

schema pour_walks;
  expands {pour walks and landscape};
  only_use_for_effects {landscaping done} = true;
end_schema;

schema install_drains;
  expands {install drains};
  only_use_for_effects {drains installed} = true;
end_schema;

schema lay_storm;
  expands {lay storm drains};
  only_use_for_effects {storm drains laid} = true;
end_schema;

schema rough_plumbing;
  expands {install rough plumbing};
  only_use_for_effects {rough plumbing installed} = true;
```

```
end_schema;

schema install_finished;
  expands {install finished plumbing};
  only_use_for_effects {plumbing finished} = true;
end_schema;

schema rough_wiring;
  expands {install rough wiring};
  only_use_for_effects {rough wiring installed} = true;
end_schema;

schema finish_electrical;
  expands {finish electrical work};
  only_use_for_effects {electrical work finished} = true;
end_schema;

schema install_kitchen;
  expands {install kitchen equipment};
  only_use_for_effects {kitchen equipment installed} = true;
end_schema;

schema install_air;
  expands {install air conditioning};
  only_use_for_effects {air conditioning installed} = true;
end_schema;

schema fasten_plaster;
  expands {fasten plaster and plaster board};
  only_use_for_effects {plastering finished } = true;
end_schema;

schema pour_basement;
  expands {pour basement floor};
  only_use_for_effects {basement floor laid } = true;
end_schema;

schema lay_flooring;
  expands {lay finished flooring};
  only_use_for_effects {flooring finished} = true;
end_schema;

schema finish_garden;
  expands {finish garden};
  only_use_for_effects {garden finished};
```

```
end_schema;

schema finish_carpentry;
  expands {finish carpentry};
  only_use_for_effects {carpentry finished} = true;
end_schema;

schema sand;
  expands {sand and varnish floors};
  only_use_for_effects {floors finished} = true;
end_schema;

schema paint;
  expands {paint};
  only_use_for_effects {painted} = true;
end_schema;
```

5 Conventions Used in the TF Description

Some simple conventions used throughout deserve explanation before the TF language is defined:

Keywords Keywords used in TF are written in **bold** lettering for highlighting purposes in the descriptions of statements.

Components TF components are surrounded by angle brackets “$\langle\ \rangle$”.

Options Optional words or phrases are surrounded by square brackets “[ ]”.

Choice If there is more than one possible representation for an expression then the alternatives are separated by the vertical bar character “|”.

Repetition If a structure can be repeated indefinitely then this is indicated by three dots “...” occurring directly under (i.e. aligned with) the beginning of the structure to be repeated, or directly after the structure (on the same line) when the meaning should be clear.[4]

Component Definition Following the definition of a TF statement, a number of components may be further defined. This is indicated by following the name of the component by “::=” and its definition.

Punctuation Use is made of two punctuation marks in TF statement definitions, namely

1. “;” indicates the end of a statement (for example a condition list).
2. “,” is used as a list separator within statements.

Neither is optional and their omission will cause an error when the TF is compiled.

The TF Compiler requires “white space” to separate operators or potential operators. E.g. you can write “3+4” but not “3+−4”. For the latter, you have to write “3+ −4”. However, single character parentheses, braces and punctuation does not require surrounding white space. When in doubt add parentheses or white space.

Comments Comments may be included anywhere in TF descriptions and are introduced by three semicolons “;;;” Everything following these in the line is treated as comment and ignored by the TF Compiler. .

[4]Note that if the structure ends with a comma, the comma should still be treated as a *separator*. That is, the comma should be written only to spearate instances of the structure and should not be written after the last instance in a sequence.

6 Component Definitions

TF forms are the basis for the TF language and are defined in the next section. A number of commonly used *components* occur in several of these forms. For convenience, these are defined separately in this section.

6.1 Basics

The following defines some basic components used in many TF forms.

```
<atom> ::= <name> | <number>

<name> ::= [<digit> ...] <letter_or_special> <name_body>

<name_body> ::= [ <letter_or_special> | <digit> ]
                ...

<letter> ::= A | B | ... | Z | a | b | ... | z

<letter_or_special> ::= <letter> | _ | %

<text_string> ::= " [ <character> ... ] "
```

Although a ⟨name⟩ can start with a digit, it must not be a ⟨number⟩.

```
<number> ::= <integer> | <float> | inf | infinity

<integer> ::= [ <sign> ] <digits>

<float> ::= [ <sign> ] <digits> [. <digits> ] [ <exponent> ]

<digits> ::= <digit> ...

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<exponent> ::= <exponent_marker> [<sign>] <digits>

<exponent_marker> ::= e | E

<sign> ::= + | -
```

`inf` = `infinity` = ∞ is a number larger than any other.

6.2 Patterns, Values, Variables and Match Constraints

```
<pattern_component> ::= <atom> | <match_constraint> |
                        { <pattern_component> ... } |
                        ( <pattern_component> ... )

<general_pattern> ::= { <pattern_component> ... }

<pattern> ::= { <name> <pattern_component> ... }

<value> ::= <pattern_component>

<variable_name> ::= ?<name>

<variable_restriction> ::= <match_constraint> | undef

<match_constraint> ::=     ??
                         | ?<name>
                         | ?{bound}
                         | ?{type <type_name>}
                         | ?{not <pattern_component>}
                         | ?{or <pattern_component> ... }
                         | ?{and <pattern_component>  ... }
                         | ?{contains <pattern_component>}
                         | ?{has <function_name>
                               [ <other_function_argument> ... ]
                                 <function_result> }
                         | ?{satisfies <predicate_name>
                               [ <other_predicate_argument> ... ] }

<function_name> | <predicate_name> ::= <name>

<other_function_argument> ::= <parameter appropriate to function>

<other_predicate_argument> ::= <parameter appropriate to predicate>

<function_result> ::= <result appropriate to function>

<sup_position> ::= <integer 1 to length of <pattern> given>
```

The special match restriction **undef** is equivalent to **??** and means that there is no restriction on the item being matched.

The other match restrictions have the following meaning:

- **?{bound}** matches items where any variables referred to are fully instantiated.

- **?{type ⟨type_name⟩}** matches items which are included in the set of names declared by ⟨type_name⟩.
- **?{not ⟨parameter⟩** } matches anything that does not match the parameter.
- **?{or ⟨parameter⟩ ...** } matches anything which matches one of the parameters.
- **?{and ⟨parameter⟩ ...** } matches anything which matches all of the parameters.
- **?{contains ⟨set⟩** } matches anything which matches one member of the set.
- **?{has ⟨function_name⟩ [⟨other_function_argument⟩ ...] ⟨function_result⟩}** matches anything (the ⟨matched_item⟩) for which lisp can evaluate

  (⟨function_name⟩ ⟨matched_item⟩ [⟨other_function_argument⟩ ...])

  and where the result of this call matches the given ⟨function_result⟩. If Lisp cannot evaluate the function on the given arguments, then the result is no match.

  An example of the use of **has** is **?{has** length 3} which can be used to check that a matching item is a list with three elements at its top level.
- **?{has ⟨predicate_name⟩ [⟨other_predicate_argument⟩ ...]}** matches anything (the ⟨matched_item⟩) for which lisp can evaluate

  (⟨predicate_name⟩ ⟨matched_item⟩ [⟨other_predicate_argument⟩ ...])

  and return **t** (true). If Lisp cannot evaluate the predicate on the given arguments, then the result is no match.

  satisfies is an easier way to specify boolean match requirements than **has**, but otherwise is very similar.

  An example of the use of **satisfies** is **?{satisfies** ¿ 50} which can be used to check that a matching item is greater than 50.

6.3 Expressions

Expressions (possibly containing variables whose values will only become apparent when a form is used within the planner) may be given wherever a number appears in suitable TF forms. This facility is not currently supported.

```
<expression> ::=    [ <plus_or_minus> ] <operand>
                  | <expression> <operator> <expression>
                  | ( <expression> )

<plus_or_minus> ::= + | -

<operator> ::= <plus_or_minus> | * | /

<operand> ::= <number> | <variable_name>
```

Operator precedence is $*$ and / take precedence over $+$ and $-$. No other precedence should be assumed. It is recommended that parentheses are used to make the meaning of the expression clear where necessary.

6.4 Sets

Sets are surrounded by parentheses "(" and ")". They may be sets of names, or may be sets of more general items which includes numbers, patterns, etc.

```
<set> ::= <name_set> | <general_set>

<name_set> ::= ( <name> ... )

<general_set> ::= ( <general_pattern> ... )
```

It is possible to do matching on a member of a set with the ?{**contains** $\langle$set$\rangle$ } match constraint. It is anticipated that more comprehensive handling and matching for sets will be added in future.

6.5 Nodes

Nodes are introduced in schemas to show a decomposition of a higher level task, action or process. Nodes have a type.

```
<node_type> ::= <action_or_event> | <dummy_node_type>

<action_or_event> ::= action | event

<dummy_node_type> ::= dummy | start | finish
```

6.6 Specifications of Node Numbers, Node Ends and Time Points

```
<node_number> ::= <integer>

<node_end> ::=  [ <end> ] <node_number>  |  [ <end> ] <self>

<end> ::= begin_of | end_of

<at_spec> ::= at <node_end>
```

A $\langle$node_end$\rangle$ can be given for any time point referred to in TF. The default "end" depends on the context of use – see the individual notes on TF forms for the default in each case. **at self** (possibly with an optional $\langle$`end`$\rangle$) may be used to specify that the time point is the whole schema expansion or the whole primitive being defined.

6.7 Numerical Bounds

```
<min_max_spec> ::=     <expression> .. <expression>
                     | min <expression> .. max <expression>
                     | min <expression>
                     | >= <expression>
                     | max <expression>
                     | <= <expression>
                     | <expression>
                     | no[ne]
                     | some
                     | ( <min_max_spec> )
```

All numerical ranges map to a minimum and maximum pair as follows:

	min	max
x	x	x
x..y	x	y
min x .. **max** y	x	y
min x	x	infinity
>=x	x	infinity
max y	0	y
<=y	0	y
no or **none**	0	0
some	0	infinity

Depending on the context, the default is usually **some** (e.g. for resource specifications given with the **overall** descriptor), or **none** (e.g. where resource usage specifications are given **at** node ends).

6.8 Time Specifications

```
<time_spec> ::=    <expression>  [ <time_units> ]
                 | <hours> : <minutes> [ : <seconds> ]
                 | <days> ~ <hours> : <minutes> [ : <seconds>]

<days> | <hours> | <minutes> | <seconds> ::= <integer>

<time_units> ::=    seconds
                 |  minutes
                 |  hours
                 |  days

<time_bounds_spec> ::=  <time_bounds_pair> [ with <time_preference> ]
```

```
                    |  ( <time_bounds_pair> [ with <time_preference> ] )

<time_bounds_pair> ::=  [ occurs_at ] <time_spec>
                    |  [ et = ]       <time_spec> [ .. ]
                       [ lt = ]       <time_spec>
                    |  [ between ] [ et = ] <time_spec>
                       [ and | .. ] [ lt = ] <time_spec>
                    |  after <time_spec>
                    |  > <time_spec>
                    |  before <time_spec>
                    |  < <time_spec>

<time_preference> ::=  ideal = <time_spec>
                               other preferences are being considered
```

The specifications of ⟨days⟩, ⟨hours⟩, etc. are integer. All time specifications map to a number of time units relative to some absolute zero time: 0~0:0:0. In the absence of an explicit specification of the ⟨time_units⟩ seconds is the default. The special symbols **inf** and **infinity** may be used and map to a number larger than any other number.

All time window specifications map onto the same form of a minimum/maximum pair as shown in the table below.

	est/min	lst/max
occurs_at t	t	t
t	t	t
et=t1..lt=t2	t1	t2
t1..t2	t1	t2
et=t1..lt=t2	t1	t2
between t1 **and** t2	t1	t2
after t	t	infinity
$>$ t	t	infinity
before t	0	t
$<$ t	0	t
default time specification	0	infinity

The **initial_time** specification for the plan will serve to assist the planner to improve the lower time bound specifications when the planner operates.

6.9 Resource Specifications

```
<resource_usage_spec> ::= <resource_usage_keyword>
                          { resource <resource_name>
                               [ <resource_qualifier> ... ] }
                          = <resource_range>
```

```
                            [ <resource_unit> ]
                            [ <resource_scope_spec> ]

<resource_name> ::= <name>

<resource_qualifier> ::= <name>

<resource_range> ::= unlimited | <min_max_spec>


<resource_unit> ::= <resource_unit_name> | <resource_unit_synonym>

<resource_unit_name> ::= <name>

<resource_unit_synonym> ::= <name>
```

May be used for plural form, etc

```
<resource_scope_spec> ::=  overall | <at_spec>
```

Default is **overall**
If the **at** option is given without an ⟨end⟩ being specified for the given node, then the default is that specified by the ⟨resource_usage_node_end⟩ in the **defaults** TF statement.

```
<resource_usage_keyword> ::=   sets        | * |
                               allocates   | - |
                               deallocates | + |
                               produces    | ^ |
                               consumes    | v
```

The * symbol is used as shorthand for **sets** rather than the more obvious = since a resource usage statement containing it will already include a = symbol.

The keyword **overall** when stated as a ⟨resource_scope_spec⟩ allows *resource specification* for the schema. This sets a bound on the resource allowed to be used in any expansion of the schema as given in *resource utilisation* statements which always have an **at** ⟨node_end⟩. The single character ⟨resource_usage_keyword⟩s given above are synonyms for the longer forms shown on each line. These short forms normally are used in printouts of resource usage statements in O-Plan.

```
consumes {resource money} = 0..1000 dollars overall
```

This sets a (min, max) pair on the whole action, i.e. it sets the limits within which resources used in schema and any expansion of it will be confined. In this case it would limit the amount of money spent to 1000 dollars. The **overall** default limits are (0, infinity).

There are limitations on the ⟨resource_usage_keywords⟩ that may be given in a ⟨resource_usage_spec⟩ for a given ⟨resource_name⟩ depending on the ⟨resource_class⟩ it is declared to belong to. These are shown in the table below.

⟨resource_class⟩	initial_resources	usage in a schema
consumable_strictly	sets/produces	consumes
consumable_producible_by_agent	sets/produces	consumes/produces†
consumable_producible_outwith_agent	sets/produces	consumes/produces†
consumable_producible_by_and_outwith_agent	sets/produces	consumes/produces
reusable_non_sharable	sets/produces	allocates/deallocates (paired)
reusable_sharable_independently	sets/produces	ditto
reusable_sharable_synchronously	sets/produces	ditto

†`produces` only allowed in a process schema (not a normal schema describing an agent's deliberate actions) for consumable_producible_outwith_agent and only allowed in a normal schema (not a process schema describing activities outside of the agent's direct control) for consumable_producible_by_agent.

6.10 Authority Statements

Authority to plan to a given level or to execute a plan is provided via the Task Assignment agent in O-Plan. TF provides support to ensure that relevant information can be communicated to the planner.

```
<authority_statement> ::=  <provides_or_requires> <individual_authority>

<provides_or_requires> ::= provides | requires

<individual_authority> ::= {authority plan <phase_number>} = <level_number>
                         | {authority execute <phase_number>} = <yes_or_no>

<phase_number> := <node_number> | all
```

⟨node_number⟩ will refer to a node within the nodes component of a task. Note that this is only suitable for one task.

```
<level_number> ::= 0 | 1 | ... | inf | infinity
```

A ⟨level_number⟩ can range up the maximum number of plan levels (and the special "level" **inf** or **infinity**).

```
<yes_or_no> := yes | no
```

7 TF Forms

Domain information is provided to the O-Plan Planner via a TF input file which is translated by the TF Compiler. A TF file is made up of TF forms. The TF Compiler operates in a single pass over the input file. The Task Formalism language syntax is designed to allow this.

```
<tf_file> ::= [ <tf_form> ]
              ...
```

There are a limited number of TF forms. They can be given in any order, and more than one particular form can appear in any separate ⟨tf_file⟩. Later forms add to the information extracted from earlier forms. The only requirement on the order in which forms are given is that information used in later forms be available before use.

```
<tf_form_keyword> ::= defaults | include | plan_viewer | world_viewer |
                      resource_units | resource_types | default_resources |
                      domain_rules | compute_condition | always | types |
                      initially | initial_resources | initial_authority |
                      initial_time

<tf_form_major_keyword> ::= tf_info | plan_levels |
                            preferences | language |
                            [meta_][process_]schema | task
```

TF forms have a regular structure. Each is introduced by a keyword and ends with a semi-colon. Internal terms are separated by ",". Where there is a compound TF form which has several internal keyword forms (e.g. for a schema definition), then the outer level form is introduced by a keyword and ended by **end_⟨keyword⟩**. Thus the general structure of TF is:

```
<tf_form> ::=   <tf_form_keyword> <component> ,
                                  ...

              | <tf_form_major_keyword> <component> ,
                                        ... ;
                    <minor_keyword> <component> ,
                                    ... ;
                    ...
                end_<tf_form_major_keyword>

<minor_keyword> ::= see specific <tf_form> definition

<component> ::= see specific <tf_form> definition
```

7.1 Documentary Information

```
tf_info <info_word> <text_string> ;
        ...
end_tf_info;

<info_word> ::= <name>
```

For example the ⟨info_word⟩ could be title, author, date, history or description, etc.

7.2 TF Compiler Defaults

The TF Compiler provides defaults for a number of components of TF forms if they are not given. For example, for ⟨pattern⟩ [= ⟨value⟩], the ⟨value⟩ is optional. If not provided, the default (normally **true**) is used instead. The TF **defaults** statement allows certain defaults to be altered. The TF complier will use the relevant defaults for all TF forms entered after the end of the **defaults** statement and will use these defaults until a new domain is specified to O-Plan.

```
defaults <default_assignment> ,
         ... ;

<default_assignment> ::= value = <value>
                       | variable_restriction = <variable_restriction>
                       | condition_at_node_end = <end>
                       | condition_contributor_node_end = <end>
                       | achieve_after_point = <default_achieve_after_point>
                       | effect_at_node_end = <end>
                       | resource_usage_node_end = <end>
                       | time_window_node_end = <end>
                       | link_from_node_end = <end>
                       | link_to_node_end = <end>
                       | resource_overall = <min_max_spec>
                       | resource_at_node_end = <min_max_spec>

<default_achieve_after_point> ::= [begin_of] self | [end_of] start
```

The following default are used by the TF Compiler if a defaults statement has not been given for any defaulted component.

```
defaults value = true,
         variable_restriction = undef,
         condition_at_node_end = begin_of,
         condition_contributor_node_end = end_of,
```

```
achieve_after_point = begin_of self,
effect_at_node_end = end_of,
resource_usage_node_end = begin_of,
time_window_node_end = begin_of,
link_from_node_end = end_of,
link_to_node_end = begin_of,
resource_overall = (0..infinity),      i.e. some
resource_at_node_end = (0..0);         i.e. none
```

7.3 Including files

```
include <file_name>;

<file_name> ::= <text_string>
```

The **include** statement causes the contents of the indicated file to be processed by the TF compiler. It can be used, for instance, when it is convenient to break up a large file into sections, or when some TF definitions should be part of several several different domain descriptions.

7.4 User Interface Specification

The user interface for O-Plan is supported through two *viewers* – a *Plan View* and a *World View* of the plan. O-Plan provides default viewers for each of these in a domain independent way. However, it is anticipated that replacement or domain specific viewers will be provided for realistic applications. This can be done with the **plan_viewer** and **world_viewer** TF forms.

7.4.1 Plan View

```
plan_viewer program = <plan_viewer_program_name>
            [ , information = <plan_viewer_parameter_string> ]
            [ , <plan_viewer_feature> = <availability> ]
            ... ;

<plan_viewer_program_name> ::= <text_string>

<plan_viewer_feature> ::= plan_output | levels_output | resource_output |
                          node_selection | link_selection | entity_detail |
                          tf_input

<availability> ::= yes | no
```

The default is **yes** for plan viewer features listed in the TF form, otherwise **no**.

```
<plan_viewer_parameter_string> ::= <text_string>
```

The plan viewer is initiated as a separate shell process and is called with the command ⟨`plan_viewer_program_name`⟩ ⟨`plan_viewer_parameter_string`⟩ `&` . The process is connected via a two way communications channel (on UNIX a *pipe*) to and from an O-Plan process. The ⟨plan_viewer_parameter_string⟩ may hold the detail of a domain specific or plan viewer specific file of information to be used in the plan viewer (such as icons for nodes in the plan, etc) as well as giving any necessary shell command modes and flags. A recommended format for a file holding details for a plan viewer program is provided, but variations for specific plan viewers are possible.

The features of a plan viewer are as follows:

plan_output indicates that the plan viewer can accept output from the planner in the *O-Plan plan output format*. A simple textual presentation of this information is possible. Note that it is assumed that all plan viewers should have the **plan_output** feature available – it would be unhelpful of a plan viewer not to provide this feature at least in a simple form!

levels_output indicates that the plan viewer can show information about levels of a plan in a useful form.

resource_output indicates that the plan viewer can show information about resource usage perhaps in the form of gantt charts, capacity profiles, etc.

node_selection indicates that the plan viewer is able to give input to O-Plan showing nodes being pointed at in the last plan which was output. The node numbers given in that output will be passed for any node selected in the plan viewer by the user.

link_selection indicates that the plan viewer is able to give input to O-Plan showing links being pointed at in the last plan which was output. A pair of node numbers is produced by the plan viewer (relative to node numbers given in the last plan output) representing the end nodes of any link selected in the plan viewer by the user.

entity_detail indicates that the plan viewer can display detail of nominated entities.

tf_input indicates that the plan viewer can produce TF input in a legitimate format (for example, if tasks can be specified in the plan viewer by some means, or if actions, resource profiles, etc can be "drawn" and converted to legitimate TF). One way in which this can be done is by the provision of drawing aids for actions, links, conditions, effects, etc.

The *O-Plan plan output format* is introduced by the single word **plan** on one line followed by statements describing nodes. Nodes are introduced with the single word **node** on one line followed by a fixed number of lines as described below. A node statement is terminated with the single word **end_node** on a separate line. The plan output format is terminated by the single word **end_plan** on a separate line. Leading spaces and tab characters on any line may be ignored. Blank lines in the output may be ignored.

```
plan
   node
      <node_reference>
      ( [ <predecessor of begin_end> ... ] )
      ( [ <successor of begin_end> ... ] )
      ( [ <predecessor of end_end> ... ] )
      ( [ <successor of end_end> ... ] )
      <node_time_information>
      <node_type>
      <node_label>
   end_node
   ...
end_plan

<predecessor of begin_end> | <successor of begin_end> |
  <predecessor of end_end> | <successor of end_end>
     ::= <end> <node_reference>

<node_reference> ::= node-<integer>[-<integer> ...]

<node_label> ::= " [ <character> ... ] "

<node_time_information> ::= ( <earliest_begin_time>
                              <latest begin_time>
                              <earliest_end_time>
                              <latest_end_time>
                              <minimum_duration>
                              <maximum_duration> )

<earliest_begin_time> | <latest begin_time> |
  <earliest_end_time> | <latest_end_time>   |
  <minimum_duration>  | <maximum_duration>  ::= <integer>
```

It is useful to know that ⟨node_reference⟩s easily show the expansion level at which a node was introduced into a plan. An example node number for a top level node such as the **finish** node of a plan is "node-2". A node which is at the third level might have a ⟨node_reference⟩ of "node-15-2-4".

If the plan viewer can call on a file of information to tailor its output, it is recommended that it contain entries in the following format (where this is possible).

```
<drawing_object_name> -> <associated_instructions_or_data>

<drawing_object_name> ::=   <action_or_event> <drawing pattern>
                          | <dummy_node_type>
```

```
<drawing_pattern> ::= <fully_instantiated_pattern> | <pattern_with_??>
```

⟨fully_instantiated_pattern⟩ and ⟨pattern_with_??⟩ are patterns not containing match restrictions or variables.

The ⟨associated_instructions_or_data⟩ could hold icon filenames or drawing instructions, etc.

7.4.2 World View

```
world_viewer program = <world_viewer_program_name>
             [ , information = <world_viewer_parameter_string> ]
             [ , <world_viewer_feature> = <availability> ]
             ... ;

<world_viewer_program_name> ::= <text_string>

<world_viewer_feature> ::= snapshot | incremental | tf_input

<availability> ::= yes | no
```

The default is **yes** for world viewer features listed in the TF form, otherwise **no**.

```
<world_viewer_parameter_string> ::= <text_string>
```

The world viewer is initiated as a separate shell process and is called with the command `⟨world_viewer_program_name⟩ ⟨world_viewer_parameter_string⟩ &` . The process is connected via a two way communications channel (on UNIX a *Pipe*) to and from an O-Plan process. The ⟨world_viewer_parameter_string⟩ may hold the detail of any domain specific drawing and presentation information necessary to specialise the world viewer program as well as giving any necessary shell command modes and flags. A recommended format for a file holding details for a world viewer program is made, but variations for specific world viewer programs is possible.

The features of the world viewer program are as follows.

snapshot indicates that the world viewer program can accept a sets of facts and statements about the world state in the form of the *O-Plan world output format* and can present this to the user. A simple textual presentation of this information is possible.

incremental indicates that it is possible to follow the initial startup of the program or any snapshot output (if that feature is available) with *changes* in the world state which the planner wishes to display. These are in the same format as the full snapshot *O-Plan world output format* but present only a partial description of a context in the plan.

tf_input indicates that the world viewer program can produce TF input in a legitimate format (for example, if tasks can be specified in the world viewer program by some means, or if initial information can be provided (e.g. an initial world state) and these can be converted to legitimate TF). One mechanism is to allow the drawing of objects directly in the domain (such as the features of a building or structure, or the placing of objects on a map) and to convert these to **initially** or **always** TF statements.

The proposed user interface for O-Plan allows for facilities for context snapshot image saving (in a *pic*) and recording and playback of a series of such images (in *flicks*) to be provided. However, these will be provided and managed by the world viewer program and are thus not part of the definition of the world viewer system in TF.

The *O-Plan world output format* is introduced by the word **world** followed by a keyword **snapshot** or **increment** on one line followed by statements of the form shown on a single line with a line **end_world** being used to terminate the output.

```
world <world_view_type>
   <pattern> = <value>
   ...
end_world

<world_view_type> ::= snapshot | increment
```

If the world viewer program can call on a file of information to tailor its output, it is recommended that it contains entries in the following format (where this is possible).

```
<domain_statement> = <domain_value> -> <associated_instructions_or_data>

<domain_statement> | <domain_value> ::=  <fully_instantiated_pattern>
                                       | <pattern_with_?? >
```

The ⟨associated_instructions_or_data⟩ could hold drawing instructions, etc.

7.5 Plan Levels

The **plan_levels** TF form allows a description of the names of actions, events, effects and resources introduced at each distinct level.

```
plan_levels <number> [ actions = ( <pattern> ... ), ]
                     [ events = ( <pattern> ... ), ]
                     [ effects = ( <pattern> ... ), ]
                     [ resources = ( <pattern> ... ) ] ;
            ...
end_plan_levels;
```

7.6 Preferences and Heuristic Information

Preferences can be stated to help the planner choose between valid options it encounters.

```
preferences <preference_statement> ;
            ...
end_preferences;

<preference_statement> :=  prefer_plans_with <number> <preference_word> ,
                                                ... ;
                          | prefer_schemas <pattern> [ = <value> ]
                                                  use (<schema_name> ... ) ,
                                             ... ;

<preference_word> ::= { resource <resource_name>
                                 [ <resource_qualifier> ... ] }
                      | <plan_feature>

<plan_feature> ::= earliest_finish_of_plan |
                   latest_finish_of_plan |
                   number_of_nodes
```

The preference information in **prefer_plans_with** is used to construct the heuristic evaluator to be used by the planner to compare different plans for continued development. Including a resource statement in the preference description means that lower utilisation of the resource indicated is treated as preferential to higher resource utilisation levels.

The preference ⟨number⟩ should be 1 for those resources or plan features which are most important in a plan. Progressively higher numbers may be used to indicate other resources or plan features which affect the value of a plan in a progressively less important way. It is possible to give the same number two or more times to indicate that two resources or plan features are of equal importance in rating alternative plans. For example:

```
preferences
   prefer_plans_with 1 latest_finish_of_plan,
                     2 {resource fuel Port_1},
                     2 {resource fuel Port_2},
                     3 {resource money};
end_preferences;
```

earliest_finish_of_plan is a measure of the earliest time at which the plan may finish. **latest_finish_of_plan** is a measure of the latest time at which the plan may finish. **number_of_nodes** counts the "complexity" of the plan.

The **prefer_schemas** statement gives the order in which the various schemas that can be used to expand an action pattern or used to achieve an **achieve** condition should be tried after ruling out any non-applicable schemas with the filter information provided. Any schemas not mentioned in a **prefer_schemas** statement but which can be used to expand an action pattern or to achieve a condition are used in the order they are presented to the planner, but after the ones declared as preferred.

7.7 Resource Information

Resources statements are of the form:

`resource` *name qualifier* ... = *number* [*units*]

Examples are:

```
{resource money} = 145 dollars
{resource fuel port_A tank_C} = 15000 gallons
{resource bricks site_567} = 10000
```

That no units are specified means that individual brick units are used.

The following unit and resource type declaration statements give the information which will allow the O-Plan TF Compiler and planner to correctly handle resource declarations in later TF.

```
resource_units <resource_unit_name> [ / <resource_unit_synonym> ]
                                          ...
                        [ = <resource_unit_class> ] ,
                ... ;

resource_types <resource_class>
                       { resource <resource_name>
                                  [ <resource_qualifier_spec> ] ... }
                       [ = <resource_unit> ] ,
                ... ;

resource_conversions <to be defined> ;
```

resource_conversions allows resource unit or resource type conversions in one or two directions. Schemas can be used to define the production of one resource from another. However, the **resource_conversion** statement may provide a convenient shorthand for some conversions.

```
<resource_unit_class> ::= count | size | weight |
                            set ( <name> , ...)            default is count

<resource_class> ::= consumable_strictly |
```

```
                        consumable_producible_by_agent |
                        consumable_producible_outwith_agent |
                        consumable_producible_by_and_outwith_agent |
                        reusable_non_sharable |
                        reusable_sharable_independently |
                        reusable_sharable_synchronously

  <resource_qualifier_spec> ::= <pattern_component>
```

Examples are:

```
resource_units  person/people = count,
                gallons = count;

resource_types consumable_strictly {resource money} = dollars,
               consumable_producible_by_agent
                   {resource fuel ?{type port} ?{type tank} } = gallons;
```

The following notes apply to resource definitions:

- Names specified in the **resource_units** statement are used to define unit types such as person/people, gallons, kilograms, etc. These are later used in **resource_types** declarations.
- The unit type needs to be one of the following **count size**, **weight** or **set**. If a set is specified then the full enumeration of the set should be defined. If not specified the unit type is assumed to be **count**. Only the **count** resource unit class is supported by O-Plan at present.

Not all of the ⟨resource_class⟩es are provided at present, and a more thorough consideration of their individual utility and necessity is still to be conducted.

As described earlier, the class of a resource will limit the types of resource usage statement that can be provided in an **initial_resources** statement or in a **only_use_for_resources** or **resources** clause of a schema. For example, a **consumable_strictly** resource cannot be **produced** in a schema.

7.8 Default Resource Information

This statement can be used to indicate default resource information about certain actions. Via this statement it is possible to give bounds on resource usage for higher level actions, prior to specific resource usage statements at the lower levels of detail. This can be helpful in avoiding wasted search. The default information is overridden by any specific statement about a specific resource for the same end of any action. It is used at compile time by the TF Compiler.

```
default_resources <pattern>
                        { resource <resource_name>
                                   [ <resource_qualifier> ... ] } =
                               [ <resource_usage_keyword> ]
                               <resource_range>
                               [ <resource_unit> ]
                               [ overall | at begin_of | at end_of ] ,
                        ... ;
                   ...  ;
```

The **overall**, **at begin_of** or **at end_of** term is given after the resource usage declaration in a position compatible with resource usage specifications given in schemas. The default if this is not specified is **overall**.

The indicated number of units of the given resource will be assumed to be *set, allocated, deallocated, produced* or *consumed* whenever the ⟨pattern⟩ appears as an action in the plan. However this is only the default situation as this can be locally overridden for any specific resource used by the action with name ⟨pattern⟩ used in a particular schema expansion by explicitly declaring a resources clause for the node which specified that action.

7.9 Calendar and Time Information

All time specifications map to a number of *time units* kept as integers in O-Plan. By default these time units are assumed to refer to seconds. It is intended that calendar information and more flexible references to time points will be possible in future releases.

7.10 Domain Constraints

```
domain_rules [ forall <variable_name> [ = <variable_restriction> ] ,
                       ... ]
                 <pattern> [ = <value> ] [ & [ <pattern> [ = <value> ] ]
                                            ...
                                     => <pattern> [=<value>] ,
             ... ;
```

For example:

```
domain_rules forall ?a=?{type block}, ?b=?{type block},
                           {on ?a ?b}=true => {cleartop ?b}=false;
```

The **domain_rules** TF form is used to state implied relationships between statements about the domain. O-Plan1 was able to use information about sets of inconsistent conditions to guide preferences about ordering of choices using a heuristic called *Temporal Coherence*. Domain rules can also provide input to this heuristic since if **A=v1 & B=v2** $\Rightarrow$ **C=v3** then a requirement for the set of conditions **(A=v1, B=v2, C=not(v3))** is obviously inconsistent. Such an inconsistency can sometimes be avoided by temporal displacement of the establishment of one or more members of the set, but this will usually require more work than attempting other search paths. Although the use of Temporal Coherence to exploit this heuristic was used within O-Plan1, it is not currently used in O-Plan.

7.11 Compute Conditions

Domain specific compute functions which may be used in the **compute** condition clause within a schema must be declared to the planner via the **compute_condition** statement.

```
compute_condition
   [multiple_answer] { <compute_function_name> [ <pattern_component> ... ]}
                                          = <value>
                     [ depends_on <compute_dependency>
                                     ...  ] ,
   ... ;

<compute_function_name> ::= <name>
```

The keyword **multiple_answer** is stated if the compute condition can give alternative answers. Otherwise a single answer is assumed. The optional **depends_on** clause must be given if the compute condition requires specific conditions to be maintained by the planner for the answers to be valid. The ⟨pattern⟩s in the declaration give restrictions on bindings for the various parameters and the answer pattern format.

A **compute** condition may have a **depends_on** phrase. The format is as follows:

```
<compute_dependency> ::= <pattern> [ = <value> ] [ from <variable_name> ]
```

A set of pre-defined compute conditions with names starting with **fn_** are provided in the O-Plan system. See the relevant section of this manual for those currently available.

7.12 Language Specific Code

It is possible to refer to specific language code relating to a domain description by using the **language** statement. This can be useful to define ⟨compute_function_name⟩ routines in particular.

```
language <language_name>;
   <language_statement>
   ...
end_language ;

<language_name> ::= lisp
```

Only Lisp is supported at present

An example is:

```
language lisp;
  (set-paramater :psgraph-all-nodes t)
  (load "some-compute-functions")
end_language;
```

7.13 Object Types

```
types <type_name> = <name_set> | <integer_range> ,
      ... ;

<type_name> ::= <name>

<integer_range> ::= ( <integer> .. <integer> )
```

The **types** statement specifies and names a class of objects within the domain. One or more of these types can then be specified within a variable declaration to describe the possible set of bindings for the variable. For example:

```
types objects = (a b c d table),
      movable-objects = (a b c d),
      count = (1 .. 10);
```

It is then possible to give a variable restriction such as ?{type movable-objects}.

7.14 Global Data

Statements which always hold in the domain may be given with the **always** TF form.

```
always  <pattern> [ = <value> ] ,
        ...  ;
```

Any **always** entry overrides any schema effect (for example, this may occur if a schema has asserted the effect {`clear ?x`} `= false` where ?x has been bound to the "table", but the "table" has been declared **always** {`clear table`} `= true`).

No match restrictions (e.g. `??` or `?`{`not table`}) are allowed in a ⟨pattern⟩ = ⟨value⟩ of an **always** statement.

It is anticipated that facilities will be added to the **always** statement to set up a class/sub-class instance hierarchy of objects with attributes which have single values inherited from their super-class(es). Patterns of form {⟨`attribute`⟩ ⟨`object`⟩} `=` ⟨`value`⟩ will be able to match against

always object/attribute/value definitions given in the anticipated form. Hence, for the moment, it is suggested that global data that could be more clearly written in a class/sub-class/instance format be represented in a number of statements of form {⟨`attribute`⟩ ⟨`object`⟩} `=` ⟨`value`⟩.

7.15 Actions and Schemas

The schema is the main form in Task Formalism.

Except for the keywords **schema** and **end_schema** and the ⟨schema_name⟩ and subject to the restriction that a usable schema will have at least one of the **expands**, **only_use_for_effects**, **only_use_for_resources** or **only_use_for_authority** statements, all other sub-clauses of a schema are optional and their ordering can be quite flexible (in general introduce types, variables or node numbers before use elsewhere).

To improve readability in the description below the [ ] brackets which should indicate that each sub-clause is optional at the top level of a schema are omitted.

```
[meta_][process_]schema <schema_name>;

instance_of <meta_schema_name> ;

;;; public information

info <info_word> <text_string> ,
     ... ;

vars    <variable_name>  [ =  <variable_restriction> ] ,
        ... ;

expands <pattern> ;

only_use_for_effects    <pattern>  [ = <value> ]  [ at <effect_point> ] ,
                        ... ;

only_use_for_resources  <resource_usage_spec> ,
                        ... ;

only_use_for_authority  <authority_statement> ,
                        ... ;

;;; private information

local_vars   <variable_name>  [ = <variable_restriction> ] ,
             ... ;

vars_relations <variable_name> <relationship> <variable_name> ,
```

```
                 ... ;

   nodes   <node_spec> ,
           ... ;

   orderings    <node_end>  [ --- <delay_spec> ]  ---> <node_end> ,
                ... ;

   conditions   <condition_statement> ,
                ... ;

   effects         <pattern>  [ = <value> ]  [ at <effect_point> ] ,
                   ... ;

   resources       <resource_usage_spec> ,
                   ... ;

   authority       <authority_statement> ,
                   ... ;

   time_windows    <time_window_spec> ,
                   ...  ;

   <other_constraint_clause> ;
   ...

   end_schema ;
```

The sub-clauses in the various forms of **schema** contain the following components.

```
   <node_spec> ::= <node_number> <node_form>
                |  <ordering_block>

   <node_form> ::= dummy | start | finish
                |  action <pattern>
                |  event <pattern>
                |  iterate <iterated_node_form>
                |  foreach <iterated_node_form>

   <iterated_node_form> ::= <iterated_node_type> <pattern> for <iterators>

   <iterated_node_type> ::= action | event

   <iterators> ::= <variable_name> over <iteration_set>
                   [and <iterators>]
```

```
<iteration_set> ::= <pattern_component>

<ordering_block> ::= sequential <node_spec> ... end_sequential
                  |  parallel <node_spec> ... end_parallel

<effect_point> | <condition_point> ::= <node_end> | notepad

<delay_spec> ::= <time_bounds_spec>

<time_window_spec> ::= <time_bounds_spec>  [ <at_spec> ] |
                       duration <node_number> = <time_bounds_spec> |
                       duration self = <time_bounds_spec> |
                       delay_between <node_end> [ and | : ] <node_end>
                                                = <delay_spec>
```

Initially, only a framework for other constraints is provided to allow for experimentation. One possible experiment might be the inclusion of a *spatial* constraint manager.

The **conditions** sub-form is one of the most complex statement in TF and has a number of options.

```
<condition_statement> ::=
            supervised <pattern> [ = <value> ]
                          at <condition_point> from <contributor_entry> |

            achieve <pattern> [ = <value> ]
                          [ at <condition_point> ]
                          [ after <achieve_after_point> ] |

            <limited_condition_type> <pattern> [ = <value> ]
                          [ at <condition_point> ] |

            compute { <compute_function_name> [ <pattern>
                                                ... ] } = <general_pattern>
                                        [ at <condition_point> ]
                                        [ depends_on <compute_dependency>
                                                     ...  ]

<limited_condition_type> ::= unsupervised |
                             only_use_if | only_use_for_query

<contributor_entry> ::= <node_end> | ( <node_end> ... )
                      | [ <node_end> ... ]
```

The [⟨node_end⟩ ...] square bracketed alternative syntax above is a valid option. The [ ] brackets here do not indicate optional inclusion as normal.

```
<achieve_after_point> := <end> <node_number>
                       | [ begin_of] self | [end_of] start
```

The following sections describe some of the parts of the **schema** statement in more detail.

7.15.1 General Notes

In all cases where a value is allowed and no [= ⟨value⟩] term is given to accompany a ⟨pattern⟩ (e.g. in effects and conditions) then the ⟨value⟩ defaults to that set in the TF Compiler **defaults value** statement.

Similarly if there is no explicit **at** ⟨node_end⟩ mentioned then the relevant information (effect, condition, time window, resources, etc.) is given with respect to the overall schema network (i.e. the equivalent of **at self** with a **begin_of** or **end_of** keyword). With normal TF Compiler defaults for the end to use for effects and conditions, this implies that the effect is asserted at the **end_of** of the *last* node of the expansion, whereas **conditions**, **time_windows** and **resources** are assumed to take effect from the **begin_of** of the *first* node of the expansion. This is the case for single and multiple node networks.

7.15.2 Schema

meta_schemas, **process_schema**s and **meta_process_schema**s are not currently supported by the O-Plan TF Compiler, but our intention is to provide these in future.

A **[meta_][process_]schema** may be an instance of an existing **meta_schema**. The effect is as if all the components of the meta_schema were already available to the new schema being defined. A meta_schema can have any components that a schema can, except for nodes and orderings on these nodes. The **meta_** capability is entirely implemented by the TF Compiler and involves the equivalent of a textual addition of the components provided in a **meta_schema** into the schema which is declared to be an **instance_of** the meta-schema.

7.15.3 Vars, Local_vars and Vars_relations

Schemas may introduce variables via the **vars** statement, and can appear in any ⟨`pattern`⟩ or ⟨`pattern>=<value`⟩ component within the rest of the schema.

When declared, variables may be unrestricted (the value **undef** – the default) or given some restriction (e.g. ?{**not** table} or ?{**type** block}). A type is required for any variable which will not be bound after schema selection by the planner. A schema may insist on a full binding for a variable before being used for an expansion (using the **?{bound}** match specification).

local_vars (like **vars**) Variables which are declared in the vars and **local_vars** sections are handled similarly and indeed there is nothing to stop the TF writer declaring all variables in the **vars** section. However, by splitting the declarations a clearer distinction is made within the schema between the *public* and *private* parts of the schema.

The **vars_relations** statement specifes that two variables within the schema have a specified relationship. At present this can be = or / =.

7.15.4 Expands and Only_Use_For_ ...

The **expands** ⟨pattern⟩ phrase is only present if the schema or process schema refines an action or event (respectively) to a lower level of detail – this is essential in the case of a primitive schema description. If there are no **only_use_for_...**s mentioned then it is assumed that the schema can only be used for expansion. This implies then that at least one of the **expands** or **only_use_for_...** statements must be present in any schema.

only_use_for_effects allows selection of a schema to achieve a required condition in the plan. The **only_use_for_effects** are asserted into the TOME like normal **effects**. The normal **effects** in the schema are not used in the lookup of suitable schema expansions and they should not repeat or contradict **only_use_for_effects** statements used earlier in the same schema.

only_use_for_resources allows selection of a schema to change resource availability in a plan – normally through the production of resources.

only_use_for_authority allows selection of a schema to change authority levels for planning or execution.

7.15.5 Nodes, Orderings – Expansions or Decompositions

The *expansion* or *decomposition* of a schema is defined by the **nodes** list and the associated **orderings**.

Nodes may have type **action** or **dummy**. ⟨node_type⟩s **start** and entry for each member of the set and for iterate will establish a**finish** are simply special forms of the **dummy** node type and are only found in task_schemas. They may be introduced automatically by the TF Compiler or User Interface rather than explicitly included by the user. All **dummy** node types are defined to have a duration of 0 time units, i.e., their **begin_of** time is identical to their **end_of** time.

The **orderings** on the nodes specified in the schema network may include a ⟨delay_spec⟩ between the two ⟨node_end⟩s. This may be set to 0 for *consecutive* actions or any numerical expression specifying a delay >= 0. This expression may be an evaluable expression which may or may not contain variables.

An ⟨order_block⟩ allows ordering constraints to be specified along with the nodes. A **sequential** block specifies that the nodes and blocks it contains must be linked to enforce the order in which they are listed. A **parallel** block adds no restrictions of its own but allows nodes to be grouped within a **sequential** block.

The **foreach** or **iterate** option may be given for node types **action** and **event** along with the associated **from** ⟨variable_name⟩ **over** ⟨iteration_set⟩ phrase (which is only allowed in this context). This indicates that the node information should be replicated for each member of the set and the resulting nodes placed in parallel (**foreach**) or in a sequence (**iterate**). If there is more than one iterator, connected by **and**, nodes are generated from the Cartesian product of the sets.

Although the ⟨iteration_set⟩ is defined as a ⟨pattern_component⟩, which might be a single variable, it should be instantiated to a list of items, as if the syntax had been ⟨general_set⟩.

For both **foreach** and **iterate**, expansion will generate a new node entry for each member of the set (or of the Cartesian product). For **iterate**, there will additionaly be a sequential ordering of the nodes. An example is:

```
N iterate action {fly_to ?way_point}
                 for ?way_point over ({100 50} {200 60} {150 40})
```

on expansion this would be equivalent to:

```
nodes N action {fly_to {100 50}},
      X action {fly_to {200 60}},
      Y action {fly_to {150 40}};
orderings N ---> X,  X ---> Y;
```

Any orderings on the original node number N are applied to the **begin_of** the first node in the sequence and the **end_of** the last node in the sequence.

A **foreach** is identical except that no orderings are introduced. This means that any orderings given on the **foreach** node number may cause **dummy** nodes to be inserted to preserve the intended orderings. For example,

```
N foreach action {counter_problem ?problem}
                 for ?problem over (issue_1 issue_2)
```

on expansion this would be equivalent to:

```
nodes N dummy,
      X dummy,
      Y action {counter_problem issue_1},
      Z action {counter_problem issue_2},
orderings N ---> Y, N ---> Z,
          Y ---> X, Z ---> X;
```

Note that the expansion for **iterate** and **foreach** is done while palnning and (*not* at TF compile-time). Hence, it is possible for the iteration set to be something that could not be determined at compile-time. This llows, as examples, the set to be read from an external source of information (such as an external trajectory computation routine) or instantiated through variables set elsewhere in planning.

7.15.6 Conditions

Typed conditions are used within O-Plan to aid the planning process.

A **supervised** condition is satisfied from an earlier point in that schema and must be further qualified by the [**from** ⟨contributor_entry⟩] which identifies the point(s) at which the contributing effects are made available.

7.15.7 Conditions - Compute

compute conditions provide the external systems interface to O-Plan. The left hand side of a **compute** condition has the form:

```
{ <function_name> [ <parameter> ... ] }
```

where each parameter can have the same recursive form. The parameters will typically be (or at least include) schema variables which will normally be instantiated before use.

The ⟨general_pattern⟩ must match against the result for the **compute** condition evaluation to succeed. Further variable binding can occur during this match.

The optional **depends_on** phrase, which is only given for a compute condition, has one or more ⟨compute_dependency⟩ clauses of the form:

```
<pattern> [ = <value> ] [from <variable_name> ]
```

and these are returned to record the dependencies that must continue to be maintained for the results to be valid. If the **from** phrase is specified, the particular statement must be maintained from (one of the list of) contributor(s) returned to the ⟨variable⟩.

For example, the following **compute** condition would check if one block was *over* another block by (recursively) checking individual statements of form {on _ _}.

```
compute {over ?x ?z}=true at end_of 3
           depends_on {on ?x ?y}=true from ?contrib-1
                      {over ?y ?z}=true from ?contrib-2
```

If there is no **from** phrase then it is assumed that the contributor is the initial node of the plan, that is from the initial state. This also works for the case when the contributor is an **always** fact, however, it is better not to give a **depends_on** term if a dependency is related to a fact which can never be invalidated. Dependencies must be maintained *to* the ⟨node_end⟩ specified in the **at** phrase for the condition. As with **only_use_for_query** type conditions, dependencies are maintained but are considered "re-establishable" by recomputing the **compute** function. This will result in the retraction of the original GOST entry and the establishment of a new one.

compute type conditions containing variables are evaluated as and when the appropriate variable bindings are found.

7.15.8 Notepad

The **notepad** can be used as an **at** qualifier to keep effects which are related to the overall plan but are not associated with any specific plan item (such as an action) in the plan. **notepad** effects are known as *notes*. Conditions can be stated with respect to the **notepad** effects using **at notepad**.

7.15.9 Authority

Not in use in the current version of O-Plan.

7.15.10 Time Windows

It is possible to specify three types of time windows in schemas. One is normally used to give a metric time value (e.g. a specific time or date) or a time relationship to a specific time point. This is done with the **at** option. The second is used to specify the **duration** of a node (normally an action node type) in the expansion. The third allows time distances between two points to be specified (normally for a **delay_between** between the ⟨end⟩s of one node and another. This **delay_between** form is exactly equivalent to the specification of a ⟨node_end⟩ — ⟨delay_spec⟩ —> ⟨node_end⟩ in the orderings statement. The TF writer is free to use whichever form is most convenient. If *both* forms are used the specifications must be compatible (but not necessarily equal).

7.15.11 Other Constraints

The O-Plan architecture allows for the replacement of the standard constraint managers for time and resource management with more capable constraint managers from other sources, and also for the addition of constraint managers for new constraint types.

If a standard manager is replaced, then the more capable manager may be able to handle richer constraints as well as managing the standard time and resource information given in TF. Since these richer constraints may not fit the standard time and resource syntax, they can be specified in an ⟨other_constraints_clause⟩. New types of constraints may also be specified in that way. The TF compiler provides a way for the syntax of ⟨other_constraints_clause⟩ to be extended to include these new syntaxes.

O-Plan provides this feature as a way to allow for extension of the TF Compiler by a systems integrator and to act as a demonstration of how to pass information between O-Plan and its constraint managers. Any **other_constraints** which are rejected by the relevant constraint manager which is installed in O-Plan will be treated as a **notepad** effect and will thus appear there as a **note**.

7.16 Primitive Actions

A definition of a *Primitive Action* can be given by providing a schema with an **expands** statement and no *expansion* (i.e. no **nodes** and associated **orderings**). An action pattern introduced by a higher level schema is then considered as *primitive* and not expandable further if its action ⟨pattern⟩ matches such an **expands** entry. A *Primitive Action* schema can still have other schema related information such as **vars**, **conditions**, **(only_use_for_)effects**, **time_windows** and **(only_use_for_)resource** statements.

7.17 Initial Information for Plan Generation

An initial world description and other initialisation for tasks may be provided through the following TF forms:

```
initially <pattern> [ = <value> ] ,
          ...  ;

initial_resources  <resource_usage_spec> ,
                   ...  ;

initial_authority <authority_statement> ,
                  ... ;

initial_time  <time_spec> ;
```

Unlike all other TF forms, these initial statements are not additive or incremental to the information from previous forms. They represent the total information to be used in the specification of any following task. It is possible to select an empty set of initial information by use of the following:

```
initially;             ;;; no initial world model statements
initial_resources;     ;;; no initial resource availability
initial_authority;     ;;; no initial authority (see note)
initial_time;          ;;; reset initial time to "zero" time
```

The initial information given is used for any following task that is provided. A change of the initial information does not alter the initial information which may have been used for earlier provided task statements.

Giving an empty **initial_authority** would not be very useful as it would not allow any planning to take place. The TF Compiler will issue the warning "no authority to plan being given – check that this is intended". A more useful "baseline" initial authority would be to allow planning to the most primitive level of activity available, but not to go on and immediately allow execution without further authorisation. That is:

```
initial_authority provides {authority plan all} = inf,
                  provides {authority execute all} = no;
```

This is the default on O-Plan initiatisation if no explicit **initial_authority** statement has been given.

7.18 Task Schemas

Task schemas are the means of (uniformly) specifying particular tasks for plan generation. Within these schemas it is possible to fully specify separate resource limits, initial time windows and initial world model states for each of the possibly many separate alternative task schemas at the top level.

Schemas whose name starts with the keyword **task_** are selected by the O-Plan Task Assignment user menu system to present to the user as alternative tasks to select from. Once one is selected, planning is initiated by posting an initial plan based on it.

A task schema uses the information from the last provided **initially**, **initial_resources**, **initial_authority** and **initial_time** TF forms as a basis. Additional effects, resources and authorities or a different time specification can then be provided and will add to those provided previously. Task schemas inherit these initial statements in a manner similar to the schema/meta_schema relationships. Note that in the case of initial authorities, if none are given in an **initial_authorities** statement, then the default is to allow planning to any level, but not to allow execution (see **initial_authority** definition section).

A task schema has the following general form:

```
schema task_<name>;
   only_use_for_effects {task_achieved} = true at end_of 2;
   nodes   1   start,
           2   finish;
   orderings end_of 1 ---> begin_of 2;
   ;;; the user may provide additional nodes, orderings, conditions, etc
   time_windows  <initial_time> at 1;
   resources     <initial_resources> at 1;
   authority     <initial_authorities> at 1;
   effects       <effects from initially statement> at 1;
end_schema ;
```

The current relevant TF Compiler **defaults** node ends are used for the **at** 1 (**start** node) position for initial effects, authority, resources and time. Recall that a **start** node is a **dummy** node and hence has zero duration. This means that all initial information will apply at the same plan-relative time whichever end is the current default.

For convenience, a TF form is provided to ease the specification of tasks. The basic form is:

```
task <name>;
   ;;; the user may provide nodes, orderings, conditions, etc.
end_task;
```

The TF Compiler adds the following to the entries provided by a TF writer in such a **task**:

```
time_windows  <initial_time> at 1;
resources     <initial_resources> at 1;
authority     <initial_authority> at 1;
effects       <effects from initially statement> at 1;
```

In all task schemas, the user must ensure that the **start** node is node number 1 and that the **finish** node is node number 2. These must be ordered with respect to each other and any other user provided nodes such that provide for a unique start and finish node for the task expansion.

8 TF Compiler

The O-Plan TF Compiler converts the Task Formalism language (coming from a file or from typed input from a user) into the internal Domain Information used by the O-Plan planner. The compiler can be run incrementally and will add to or modify the existing Domain Information available to the planner.

Where a TF form is specified which has the same name as a form which already exists (say a schema or a type with the same name) or where a statement which is not additive is given (i.e. the **initially**, **initial_resources** and **initial_time** statements) then these override any previous entry in the domain information.

It is anticipated that facilities to change previously specified TF forms will be provided, as well as the current facility to completely replace an old form or add to the forms already present.

The **defaults** statement in TF is used to inform the planner of the defaults it should use in its operation. The compiler uses the given defaults or those provided in the last **defaults** statement until such time as a new domain is selected (normally by re-initialising the planner or calling the **new_domain** O-Plan command).

The TF Compiler performs a number of TF *form expansion* roles. Two examples are for the incorporation of the current **initially**, **initial_resources** and **initial_time** statements into a **task_schema** and the incorporation of the text of a **meta_[process_]schema** into a schema shown as an **instance_of** the meta-schema.

9 O-Plan Commands

The following are commands which can be used to control the top level of O-Plan:

oplan This command is the means of entry into the menu-driven O-Plan task assignment interface which will be the normal method of interacting with and controlling O-Plan. It provides access to each of the commands below in a convenient form.

new_domain This instructs the Planner to clear all knowledge of previous problems worked on during this current session.

tf ⟨domain_name⟩ This action loads in the TF description from the file associated with ⟨domain_name⟩. This file name is derived directly from the ⟨domain_name⟩. It will incrementally add to any previous TF descriptions given since the last **new_domain** command.

plan ⟨task_schema_name⟩ Planning starts with the loading onto the pending task agenda of the task schema with name **task_⟨task_schema_name⟩** describing the action(s) to be expanded, or condition(s) to be achieved. An error will be signaled if a task schema with the given name has not been provided previously.

plan_view This command provides a menu of plan state browsing facilities to the user. Plan browsing is possible at any time, whether or not a plan has already been successfully generated.

world_view This command provides a menu to a facility which can provide descriptions of the state of the world model at nominated points in the plan. Plan simulation is possible at any time, whether or not a plan has already been successfully generated.

replan Having already generated a successful solution, **replan** looks for a further solution amongst the alternatives remaining.

execute This command instructs the planner to pass the plan for execution to the execution system. An error will be signaled if the execution system is not available in the O-Plan system running. In the current implementation. execute may only be used on a valid fully generated plan.

quit To exit from the O-Plan system.

10 Predefined Compute ⟨function names⟩

O-Plan provides a number of predefined compute functions. Each has a name starting with **fn_**. The predefined boolean predicates return values "**true**" or "**false**". This simplifies the writing of compute conditions in schemas. E.g.

```
condition compute true = {fn_neq ?filter green}
```

- **fn_ask** (⟨prompt⟩ [, ⟨response_list⟩])
  ⟨response_list⟩ may be **undef** for any answer. This is the default if no ⟨response_list⟩ is given. The first answer in any list is the default if the user types return. If the user types return and the response list is undef, then the user is asked the question again after a suitable message is given about their being no defaults.
- **fn_cond** (⟨boolean⟩ , ⟨true_result⟩ , ⟨false_result⟩)
  Provides a conditional answer which depends on the boolean parameter.
- **fn_or** (⟨boolean⟩ , ⟨boolean⟩)
  This is a function to compute the "or" of the arguments.
- **fn_and** (⟨boolean⟩ , ⟨boolean⟩)
  This is a function to compute the "and" of the arguments.
- **fn_eq** (⟨parameter⟩ , ⟨parameter⟩)
  Checks if the parameters are equal.
- **fn_neq** (⟨parameter⟩ , ⟨parameter⟩)
  Checks that the parameters are not equal.
- **fn_leq** (⟨parameter⟩ , ⟨parameter⟩)
  Computes whether parameter 1 is less than or equal to parameter 2.
- **fn_geq** (⟨parameter⟩ , ⟨parameter⟩)
  Computes whether parameter 1 is greater than or equal to parameter 2.

These are available as if the following **compute_conditions** statement had been provided.

```
compute_condition {fn_ask ?? ?{or undef ?{type list}}} = ??,
     {fn_cond ?{or true false} ?? ??} = ??,
     {fn_or ?{or true false} ?{or true false}} = ?{or true false},
     {fn_and ?{or true false} ?{or true false}} = ?{or true false},
     {fn_eq ?? ??} = ?{or true false},
     {fn_neq ?? ??} = ?{or true false},
     {fn_leq ?{type number} ?{type number}} = ?{or true false},
     {fn_geq ?{type number} ?{type number}} = ?{or true false};
```

Currently, all predefined compute functions are defined to require fully instantiated parameters (i.e., have an additional constraint of ?{**bound**}) and will return a single result without any dependency information.

11 Guidelines for Writing TF

It is intended that a guide to how to approach the modelling of a domain in TF will be provided in due course. For the moment, this section will collect together detailed advice on the use of various TF forms and experience gained, or common pitfalls encountered, in coding specific domains.

We suggest below an ordered set of steps that a TF domain writer may go through to ensure a good result. We rather grandly call this the Task Formalism Method (TFM) to reflect our desire to gather experience of writing TF to improve the method itself and to provide future TF Compiler intelligent support and user guidance.

11.1 Scope the Domain and Initial Analysis

Like any data analysis task, it is important to plan carefully how a domain description is to be provided in TF to O-Plan. It is all too easy to let a domain description grow in a haphazard and inconsistent way. The present TF compiler and user interface support aids do not offer the TF writer much support apart from error-checking.

It is useful to view one user role in writing a domain description in TF as being that of *Domain Expert*. This user will decide on the scope of the domain and introduce the top level of description. It is then possible to "fill-in" the details by considering other information given to describe a domain in TF as being provided by one or more *Domain Specialists*.

11.2 Action Expansion or "Goal" Achievement?

Two different approaches are possible to model domains. A hierarchical *action expansion* approach is primarily supported by O-Plan. However, it is also possible to state required conditions on the state of the world at certain points – a *goal achievement* approach. This is also supported by O-Plan. The approaches can be mixed in any way convenient to model the domain. However, it is useful to consider which is to be the main approach during the initial domain modelling exercise.

11.3 Levels of Modelling

It is all too easy to introduce actions, events, effects and resources and state conditions or use resources at different levels, making the modelling awkward and unnatural. This is sometimes referred to as "hierarchical promiscuity" or "level promiscuity". This will almost certainly lead to the inability to make effective use of search restriction domain information such as condition and resource types.

Actions and the effects they introduce are at a particular domain modelling level. Higher levels are more abstract, lower levels are more detailed. In some cases, certain (external) types of conditions can only be stated on effects introduced at a domain modelling level which is at a higher or the same modelling level as the condition. In other cases, certain (internal) types of conditions can only be stated on effects introduced at a domain modelling level which is at the same or a lower modelling level as the condition.

In anything other than trivial domains, it is essential to have a plan based on an initial analysis of the structure of the problem to decide on what actions, events, effects and resources will be modelled at progressively more detailed levels.

Assuming an action expansion approach, we suggest the following method as being suitable for O-Plan; it provides most effectively for hierarchical expansion of activity based plans with associated condition/effect, resource and time modelling. The steps may not be able to be followed in a total order, but the items below may act as a useful checklist. If a goal achievement approach is being used primarily, then the ordering of the steps may place an earlier emphasis on those steps related to condition (especially the **achieve** type) and effect modelling.

1. Identify the main actions (and events) that will appear at the top level of a task or plan. This is the *task* or top level.

2. Gradually work down through progressively lower levels of detail and try to identify the more detailed actions (and events) to be introduced. It is best if each level introduced has some real meaning to those involved in planning in the real world. Giving a "name" to each level is a good discipline to ensure that the modelling levels will be useful.

3. It is then useful to decide on what statements about the world (in the form of effects) will be introduced and manipulated at the various levels by the actions (and events) at each level.

4. It is only after these steps have been taken that the conditions required for each action (or event) need to be considered. It is then possible to ensure that these are introduced at levels at or below the level in which the relevant effects are introduced.

   Type information to restrict the usage of conditions to those that are meaningful in the domain can now be added reliably.

5. This can then be refined by considering the resources that are manipulated at each level.

6. Time restrictions and information can then be considered.

11.4 Writing a Schema – the Schema Envelope

When writing a schema in TF it is useful to view the schema as "owned" by some individual responsible for the activity being described. That individual describes one way in which the higher level activity can be performed (or one way in which an indicated effect can be produced) in a plan.

Consider that there is a bounding box or *envelope* which at its outside edge performs the activity or produces the desired effect(s). It is possible to state overall requirements for the schema on the envelope itself – in terms of required conditions, outer bounds on resource usage, outer limits of the time the activity will take, etc. This can be done by using the keywords **overall** or **at self** in appropriate specifications of resources, time windows and conditions. Then, internally within the envelope, the details of the way in which this is done can be described.

11.5 Help for the TF Writer

These notes provide more detailed advice on the use of specific TF forms.

1. O-Plan (as its predecessors Nonlin and O-Plan1) is able to effectively exploit effects and conditions expressed in a functional way. Where non-boolean functional relationships exist in the domain, modelling them directly can give great advantages in terms of search space restriction and clarity of description over a more traditional modelling of facts in a predicate logic form, as is common in other AI planners. For example, rather than use: `{switch 1 on}=true` and `{switch 1 off}=false` as two separate statements it is better to use a functional form `{switch_mode 1}=on` and limit the value to be one of `on` or `off`. This is even more effective when the value can take a set of bindings. E.g. `{filter camera_1}=cyan` where the colour can be one of a set of filter colours available. Similarly some numerical functional relationships are easily and efficiently expressed to the planner by this mechanism such as `{age person_2}=37`.

2. To model the inclusion of conditional actions, it is necessary to provide two schemas for an effect which represents that a condition is satisfied or the action is included. For example to conditionally include a painting action only if the walls are not already painted, it is possible to provide two schemas with **only_use_for_effect** (walls painted) say. One schema would have a set of conditions that checked if the walls were already painted but would not introduce more actions. The other would check that walls were *not* already painted (via **only_use_if** conditions) and would introduce the action to paint the walls. The conditional action can then be introduced at any point in another schema by including an **achieve** condition for the **only_use_for_effect** (walls painted say).

3. To model the conditional or case-based inclusion of effects it is necessary to provide several schemas with mutually exclusive condition or variable restriction sets that differentiate when the effect should be introduced and when it should not be.

4. Variables can be used in schemas, but it is a requirement that the planner know the type of any partially instantiated variables left after all selection conditions and constraints on the schema have been applied. It is an additional constraint that this type must be enumerable (such as a set type – like a Pascal scalar, or an integer with a bounded range). If you write a schema in such a way that there is any possibility of a variable within the schema being left not fully bound, then it must have such a type added to its restrictions list.

11.6 Modelling Reusable Non-sharable Resources with Effects/Conditions

Until such time as O-Plan supports a wider range of resource types, reusable non-sharable resources can be modelled with conditions and effects. Reusable non-sharable resources are fixed items such as keys or a specific transport vehicle.

For Action Expansion In an action expansion modelling approach is being used, resource allocation/deallocation can be done with effects, supervised and unsupervised conditions at the outer envelope of the action. The resources and their status would usually be declared in the initial state for a task as follows:

```
effects {status workman_1} = unallocated at 1,
        {status workman_2} = unallocated at 1;
```

An action which required such a resource would have conditions and effects such as:

```
condition unsupervised {status ?workman} = unallocated at begin_of self,
            supervised {status ?workman} = allocated_to_XX
                                   at end_of self from begin_of self;
  effects   {status ?workman} = allocated_to_XX at begin_of self,
            {status ?workman} = unallocated at end_of self;
```

The `allocated_to_XX` is qualified by the purpose of the allocation (`XX`) to ensure that other potential parallel actions do not mistake the allocation of the resource as being related to their different purpose.

The allocation of the resource can be more precise than the outer envelope of the whole action (**at self**) if the range required is known to be less. The interval between the resource becoming available and it being released will be protected by the **supervised** GOST entry.

The use of a **unsupervised** condition type as opposed to an **only_use_for_query** will greatly reduce the size of the search space since it leaves scheduling of the resources until late in planning (at the expense of possible failure to resource a given plan at this late stage). For example, in a problem of allocating 4 resources to two activities using **unsupervised** as opposed to **only_use_for_query** the number of planning cycles was reduced from 839 to 38 in a specific domain.

For Goal Achievement If using a goal achievement approach (i.e. **achieve** condition satisfaction) most of the resource allocation can be done by the use of `achieve` conditions. This leads to larger search spaces.

12 Current Implementation

12.1 Unsupported Features

The following features are not supported by the current implementation of O-Plan. They have been shown in the TF Manual as an indication of how O-Plan is intended to develop.

1. ⟨expression⟩ is not supported in ⟨time_spec⟩ and ⟨min_max_spec⟩. Only a ⟨number⟩ may be given wherever ⟨expression⟩ appears in the syntax.

2. **plan_viewer** and **world_viewer** TF forms are supported only as documentation at present. Any specific plan viewer and world viewer programs other than the default must be activated explicitly by the user.

   Only the world viewer snapshot output facility is supported.

3. **preferences** are not supported. Schema preference ordering is the same as the order of processing by the TF compiler, modified by a cost estimate.

4. Only the **consumable_strictly** resource class is supported. Consideration of the value of the different resource classes is still necessary.

5. Only the **count** resource type is supported. Consideration of the value of the different types is still necessary.

6. The **default_resources** statement is not supported. The value of this statement is under consideration.

7. The **domain_rule** statement is not supported. Also, O-Plan does not use *Temporal Coherence* information to inform its search choices.

8. The **plan_levels** statement is not supported.

9. Restricted **compute** condition support is available. The **depends_on** clause cannot be given, none of the predefined compute functions (see §7.18) are available, and the **compute_condition** statement is largely ignored. However, any Common Lisp function whose name would count as a ⟨name⟩ can be called, and **multiple_answer** is supported. (At present, the **compute_condition** statement is consulted only to see whether function are **multiple_answer** or not.) Moreover, provided that all the variables in the function arguments are bound by the time the condition is evaluated during schema selection, numeric computations may be freely performed. (After that, numeric values of variables must be members of the variable's type, and the only numeric types available are integer ranges.)

10. **meta_schemas** and **process_schemas** are not supported.

11. The **event** node type is not supported.

12. The **foreach** and **iterate** node iteration is supported, but the iteration set(s) must be fully instantiated by the time the schema is selected. This can usually be accomplished by using **only_use_if** or **compute** conditions to bind the variables in the ⟨iteration_set⟩(s).

    Note that the expansion of **foreach** and **iterate** is not exactly as described in section 7.15.5. Instead a dummy node is always created and all the generated nodes are put *between* the two ends of the dummy. This means that references to the node number of the iteration can work unmodified (while in the documented implementation some renumbering of references would be needed), which significantly simplifies the implementation. However, it leads us to violate the rule that a dummy node always has zero duration.

13. The **notepad** is not supported.

14. The ⟨other_constraint_clause⟩, which provides a place for new types of contraints managed by "plug-in" constraint managers (including constraint managers external to O-Plan), is supported, but the details are subject to revision and are not yet described in this document.

15. The **initial_authority** statement is not active at present. Give the information directly in a **task** schema. However, **initially**, **initial_resources**, and **initial_time** will work.

12.2 Features Anticipated

There are a number of areas where extensions to TF are anticipated in future, but where detailed discussion of the form of the TF statement has not been decided.

1. TF has been designed to allow many compile time error messages and warnings to be given to the TF provider. The current system should detect most syntax errors and performs a range of semantic checks. However, more extensive checking is planned.

2. It is anticipated that facilities to change previously specified TF forms will be provided, as well as the current facility to completely replace an old form or add to the forms already present.

3. It is anticipated that considerable development of the **plan_viewer** and **world_viewer** features will be undertaken. In particular the development of a query language to allow the viewers to selectively interrogate the agent and plan state in O-Plan may be a focus for future work.

4. It is anticipated that the **type** of a variable will be extended to allow further numeric types.

5. It may be useful to require that a unique start and end node be provided for any schema expansion given. This would allow regular handling of plans and schema expansions in the O-Plan system and in user interfaces. If necessary, the TF Compiler could introduce new **dummy** nodes to ensure that a unique initial and final node in an expansion was available. The TF Compiler could renumber the nodes in the **nodes** statement where

necessary to ensure that the unique initial node is the ⟨node_number⟩ 1 and the unique final node is the ⟨node_number⟩ 2. However, any such renumbering may spoil the user's view of node numbering and naming for authority-related matters and for diagnosis of problems.

6. It is anticipated that an iteration facility will be allowed in a ⟨resource_usage_spec⟩. This will allow an **initial_resources** declaration or a schema **only_use_for_resources** or **resources** clause to give repetitive delivery conditions for **produces** statements, for example. The type of statement may have the form:

   ```
   <resource_usage_spec> [ per <time_unit> over_period <time_bounds_spec> ]
   ```

   The **per/over_period** qualifier will only be allowed for resources belonging to appropriate resource classes.

7. A means to associate *calendars* with the time units in a flexible way should be provided.

8. A way to specify numerical comparisons rather more directly than through **compute** conditions is envisaged.

9. A way to give conditional conditions relating to **compute** conditions is envisaged.

10. A method of specifying an open-ended set of effects instantiated from a set of matching objects is being considered. E.g.

    ```
    forall <variable_name> in <set_specification> <pattern> = <value>
    ```

    For example:

    ```
    forall ?sw in (switch1 switch2 switch3) {status ?sw} = off
    forall ?x in ?y {mode ?x} = nominal
    ```

    Where `?y` is set elsewhere

    There may be a need to have similar set-based conditions, or to properly manage "for all" conditions rather than "there exists" conditions as currently supported.

11. It is anticipated that more comprehensive handling and matching for sets will be added in future.

12. Consideration may be given to allow restriction of the schemas which may be tried to expand an **action** or **event** node or to satisfy an **achieve** condition. This would be done by allowing ⟨schema⟩s to be provided as *advice* which would be used instead of a general lookup for matching schemas.

13. Consideration may be given to an ability to specify ordering links with respect to a sub-node of a nominated schema which expands an **action** or **event** type node, i.e., **orderings** K.L —> M.N. This could be useful where schema expansion nodes at the lower level represent for example *phases* of some activity which are to be referred to from levels above.

14. Consideration will be given to the inclusion of a new node type which would allow an action to be considered primitive for plan generation purposes, but which could be marked as subsequently expandable in the normal way. Such a node type was included in later versions on Nonlin (called a **query** node type) to allow for actions which depended on inspections or tests. The ideas was that if the test failed, that normal expansion through the use of suitable recovery or repair actions was then possible.

15. It may be useful to provide a facility to insist that a variable be fully instantiated during the course of making a schema expansion (this may force the binding of the variable during the expansion process if this does not occur due to other conditions restricting the variable to one object). This is different to the use of the ?{**bound**} match restriction, which insists that the variable is bound when a schema is first selected from the pattern being expanded or the effect, resource or authority required.

12.3 Features Under Review

O-Plan is a research prototype and the Task Formalism (TF) language is itself undergoing changes as research ideas are clarified. Any user of O-Plan and TF should note that the O-Plan team, AIAI and the University of Edinburgh make no warranty that any TF statement form will be supported in future releases.

There are a number of areas where extensions to TF are anticipated in future, but where detailed discussion of the form of the TF statement has not been decided.

1. At present it is anticipated that normal (action) schema node lists will not contain events, and that process schema node lists will not contain actions. Task schema node lists will be allowed to contain both. However, further consideration and trial coding of domains is required to clarify this proposed restriction.

2. The **default_resources** statement is under consideration for removal and is not supported by the current release of O-Plan. Another possibility is the removal of the **begin_of** and the **end_of** optional qualifiers in the **default_resources** statement and only to let the statement define the resource specification limits between the beginning and end of the action (i.e. the equivalent to stating a ⟨resource_usage_spec⟩ **at overall**).

Index