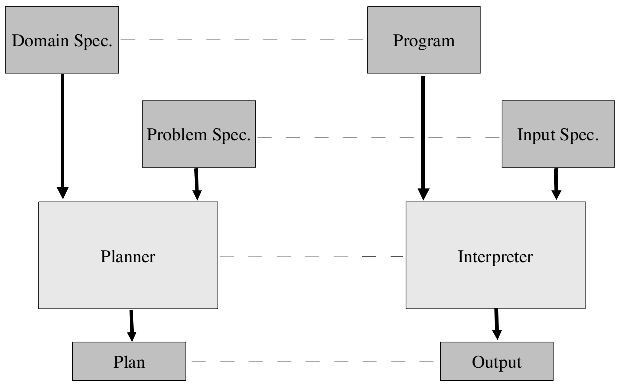
Domain Knowledge

- Domain knowledge consists of
  - declarative knowledge of the behaviour and structure of the domain
    (what can be done in the domain?);
  - description of initial and goal conditions, including problem objects
    (what must be done in this problem, and to what?);
  - control knowledge
    (why should one choose to do a particular thing?);
  - quality metric knowledge
    (what makes a good plan?).

So what is this all about?

- Key issues covered in this track:
  - How do we give a planner access to domain knowledge?
  - How does a planner determine when and which knowledge can be exploited?
  - How do we confirm that the knowledge supplied to a planner is consistent and correct?

Planning and Programming



Domain Engineering vs. Software Engineering

Some comments on domain engineering:

J. Hertzberg, 1995 , Proceedings of EWSP95, invited talk: *On building a planning toolbox*:

"While there is a large corpus of literature on the planning process and its details, there is not enough work about knowledge acquisition for planning: How to get domain knowledge into a planner?"

K. Erol, 1995, PhD Thesis: *HTN Planning: Formalisation, Analysis and Implementation, University of Maryland*):

"...it is the most neglected aspect of planning, and there is not an established software engineering methodology to guide this job".

R. Aylett, G. Petley, P. Chung, J. Soutter and A. Rushton, 1997, *Planning and Chemical Plant Operating Procedure Synthesis: a Case Study*, ECP'97:

"...minimal guidance in the literature".

Domain Engineering vs. Software Engineering

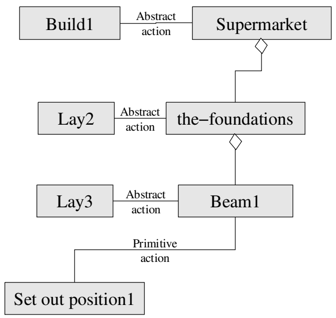
- Programming and software engineering is well supported by development environments.
- Design methodologies.
- Some powerful tools in programming:
  - Type checking (and type inference in some languages);
  - Abstraction (procedural−abstraction, object−abstraction and data−abstraction);
  - Organisational (functions, objects, records, modules);
  - Libraries for reuse;
  - Design tools with automatic code generation.
- Task Formalism method and some support tools (for O−Plan and HTN style planning). [*A. Tate, P. Jarvis and S. Polyak*]
- Object−centred language: an object−based domain encoding language. [*L. McCluskey, and J. Porteous*]
- ASPEN (Spacecraft mission planning language) [*B. Smith, S. Chien at al.*]

Examples: TF

```
task build−supermarket;
    nodes 1 action {build1 supermarket};
end_task;

schema build−supermarket;
    expands {build1 supermarket };
    nodes  1 actions {lay2 the−foundations},
              2 ....;
end_schema;

schema lay3−beam1
    expands {lay3 beam1}
    nodes 1 primitive {set−out−pos beam1};
end_schema;
```



Examples: OCLh − an HTN variant of OCL

```
substate_classes(truck, TU,
      [
      [moveable(TU), available(TU)],
      [moveable(TU), busy(TU) ]
      ]).
operator( move(V, O, L, R),
       [],
       [sc(truck,V,[at(V,O),
          is_of_sort(R,road_route),
          moveable(V),
          in_city(O,City), in_city(L,City)] =>[at(V,L)]  )],
       [ sc(package,X,[loaded(X,V),at(X,O)]
          =>[loaded(X,V),at(X,L)])   ]
).
```

Focus is on objects, with operators describing effects in terms of transitions of objects between legal states.

Examples: ASPEN

```
State_variable camera_mode {
    states = {"on","off","warming−up"};
    default_state = "off";
    transitions = ("off"<−>"warming−up",
        "warming−up"−>"on", "on"−>"off");
    };
Activity instr_on {
    constraints =
        ends_before start_of instr_off by [10,3600];
    reservation = instr_sv change_to "on";
};
```

Richly expressive language also allowing C functions to be hooked in to additional information.

Allows resources, temporal constraints, quality metrics and procedural abstraction (HTN−style).

"Control" knowledge in programs

- Separation between declarative and control knowledge in some programming languages: Prolog and pure functional languages.
- Is the price for pure declarative specification paid in performance?
- Control knowledge "smuggled in"
  - clause ordering,
  - careful management of laziness in functional languages.
- Explicit guidance of computation by procedural flow−of−control structure.
- Coding other control knowledge typically requires specialist language knowledge (including compiler/interpreter specific knowledge).
- Compiler "hints" such as *inline* and *register* require some specialist knowledge and the compiler can choose to ignore them.

Control knowledge in planner domain encodings

- Precondition and operator ordering (implicit control knowledge).
- Specification of preferred choices for achieving certain goals (distinguished *side−effects* and *primary−effects*).
- Heuristic evaluation functions.
- Focus: abstraction (ABSTRIPS and ALPINE).
- Direct guidance (HTNs).
- Control rules governing temporal relations in a plan (TLPlan [*Bacchus*] and TalPlanner [*Doherty et al*]): "if X is true of the state at time t then Y must be true at time t+1".

Most of these require planner−specific knowledge to ensure effective control.

Automatic tool support for domain engineering

- Grant has considered generating operators from entity−relation diagrams.
- Polyak has developed tool support for the construction of HTN−style domains (O−Plan).
- Gaines, Hayes and Castano have examined operator generation from a database of effector control knowledge.
- Basic generic tool support is rare or non−existent:
  - type checking (or type inference);
  - syntax−directed domain editors;
  - modular domain construction;
  - support for reuse.

Type analysis of Planning Domains

- Programs are built from predefined primitives with types defined *a priori*. The rules for types in function application allow type checking in expressions.
- In programs types define what can be done to values.
- In planning domains behaviours are (typically) built without any primitive components and there are no *a priori* types.
- In planning domains types are defined by what can be done with values.

- We *can* infer type groupings for objects in planning domains and confirm that these are consistent with user−defined groupings.
- Type inference is performed by examining which objects in the domain could instantiate operators granting particular behaviours.
- Planning domain types are defined by groups of objects that share possible behaviours.

Domain consistency checking

- Planning domains attempt to model some aspect of an external world.
- The behaviour of the model must be consistent with the (possibly abstracted) behaviour of the external world.
- Automatic consistency and validity checking would play a role similar to many compiler/interpreter error checking techniques in programming.

- To automatically check consistency requires the user to supply the same information in two different ways.
- Partially automatic checking can be done by converting a single input form into a different form and leaving the domain engineer to confirm that this is consistent with expectations.

- A help in this approach is the automatic construction of domain invariants [*Gerevini and Schubert, Fox and Long, Rintanen, Scholz*].

Where have we been?

- Domain analysis has two elements:
  - the (necessary) human domain engineer's analysis leading to a domain encoding;
  - the (possible) automatic pre−planning analysis leading to a potential improvement in planner performance.

- This talk was focussed on the first of these elements and the few support tools and methods that currently exist to make the job slightly easier.

- Several issues have been raised that motivate the second element:
  - human domain engineers are sometimes wrong about control knowledge;
  - human domain engineers don't always see a problem in a way that allows some control strategies to be exploited;
  - human domain engineers should not have to be planning (and *planner*) experts.

Where are we going?

- How can automatic domain analysis reduce the burden on the domain engineer?
  - Eliminate the need to state the obvious;
  - Grant access to unobvious control knowledge;
  - Ensure control knowledge is consistent.

- What can be inferred and how can it be used?

- What techniques and systems support the analysis?

- What does it all cost?
  - Domain analysis on−line and off−line.

No free lunch

- Compiler optimisations take time. Full optimisation could take a lot of time!
  - Eg: Register allocation problem is NP−hard.

- Sometimes make do with heuristic solutions to hard analysis problems.

- Is a heuristic analysis capable of reporting useful results?
  - Need "fail−safe" behaviour, or else the analysis is a gamble;
  - Analysis is useful if the benefits outweigh the costs;
  - But, should we amortize the costs?

Tomorrow: What domain analyses can be done and are they worth it?