Building Concepts from Components: Working Note 1

Peter Clark and Bruce Porter
Dept. Computer Science
Univ. Texas at Austin
Austin, TX 78712, USA
{pclark,porter}@cs.utexas.edu

1 Introduction

Our goal is to construct and modify concept descriptions through the manipulation and composition of *components* – fragments of representation at an intermediate level of generality.

Consider a frame (in our frame-language KM) which (partially) represents `Bus`, shown below:

```
Bus
---
   engine: PetrolEngine
           ------------
                force: Average
                fuel: Gasoline *G
                      --------
                           state: Volatile
   fuel-tank: GasTank
              -------
                contents: Gasoline *G
                material: Steel
                strength: Average
   mass: Average
   top-speed: Average
```

(`*G` indicates the slot-values are coreferential). This frame can alternatively be drawn as a conceptual graph, and this frame language viewed as a linear notation for conceptual graphs. Sowa offers a similar graph for `Bus` in his book Conceptual Structures (p129).

To build a knowledge base containing units like this, one method is simply to manually enumerate them. However, such an approach is slow, error-prone, doesn't allow us to generate abstractions of concepts, and doesn't allow us to handle novel concepts which might arise later. A preferable approach is to view descriptions like these as *compositions* of more general relationships. Individual relationships implicit in the above frame are represented as general *components* within a knowledge base. As we argue below, a compositional approach helps overcome some of the limitations of manually enumerating concepts.

The model of composition we are working with is of *unification* of these graph-structured components. Components are identified by (multiple) inheritance, by 'demons' firing (which watch for certain patterns in the composition), and by the user manually adding components. To compose two components, ie. unify two graphs, where a node in each graph are known to correspond to each other:

1. Those nodes are unified by taking their (most general) common specialization. (eg. `bus` and `vehicle` unify to `bus`).
2. Outgoing arcs with the same labels are considered equivalent, and merged into one.
3. Steps 1 and 2 are repeated as other nodes come into correspondence as a result of merging arcs.

As an example of handling novel concepts, suppose we were asked about a `RocketBus` (ie. a bus with a rocket engine). As well as replacing `PetrolEngine` with `RocketEngine` in the above frame, we'd also like to conclude some other things, eg.

- The `top-speed` is now `high` rather than `average`
- The `fuel-tank` doesn't contain `Gasoline` any more, but instead contains rocket fuel (`oxy+hyd`).
- The `fuel-tank` must be `strong` to contain explosive rocket fuel, and so must be made of something stronger than `Steel` (eg. `Titanium`, `Kryptonitemize`).
- ...

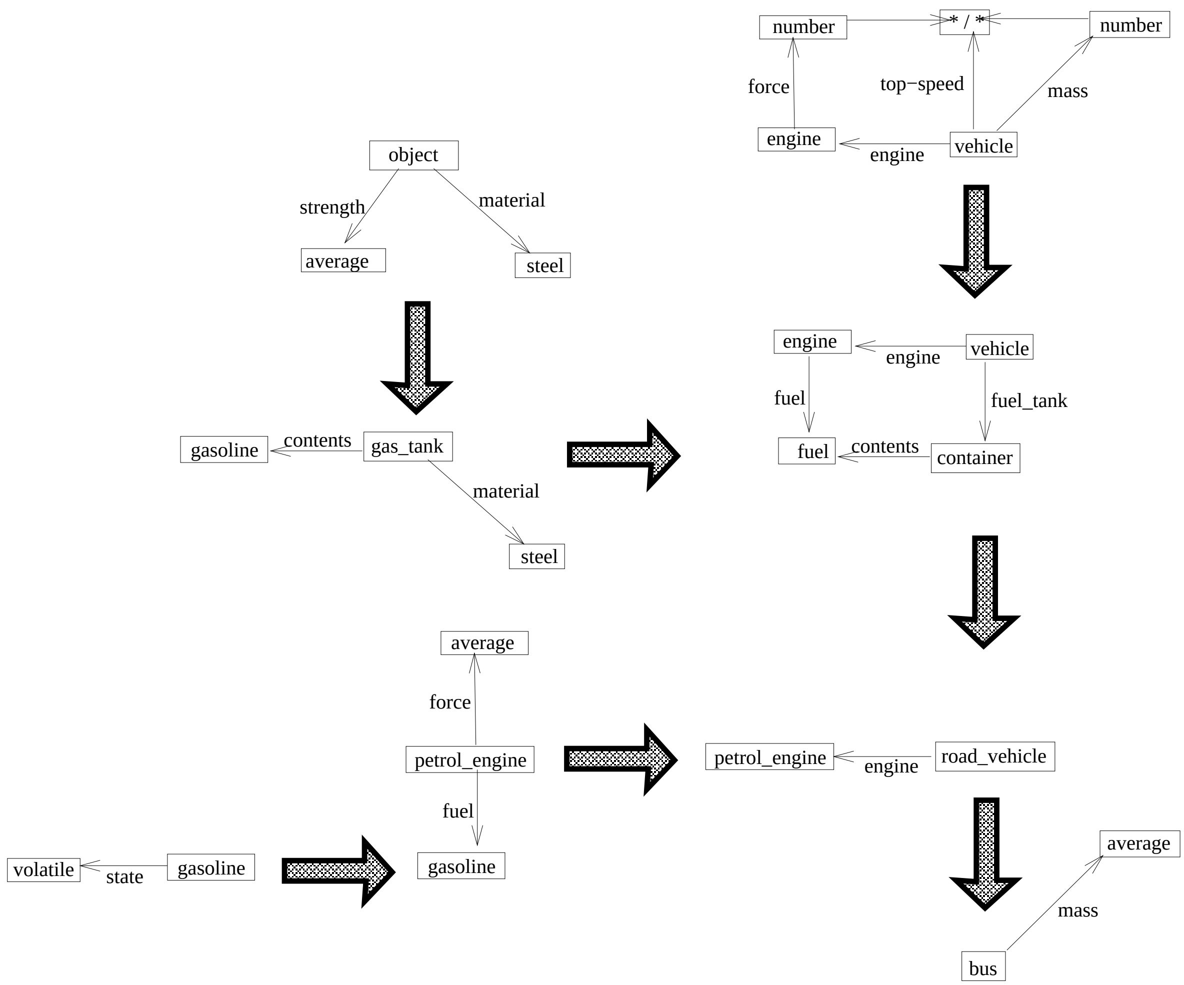
To achieve these consequences, we

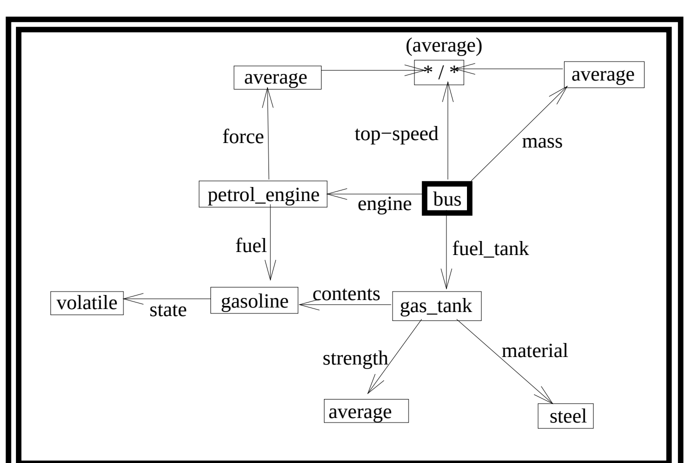
- View `Bus` as a composition of more general components
- To compose `RocketEngine` with `Bus`, we first *reassemble* `Bus` from its components, but using `RocketEngine` rather than `PetrolEngine`.
- During reassembly, some of the old components no longer 'fit' (ie. fail to unify with the new `Bus'` concept being constructed). Similarly, other new components are introduced, either automatically (through 'demons' watching for when they can apply) or manually (prompted by queries from the user).

The following pages illustrate how `Bus` and `RocketBus` are constructed from components, and then a brief six-step walk-through of how the novel concept `RocketBus` is constructed from `Bus` and `RocketEngine` is given.

Creation of the concept ‘Bus’ through composition of components.

Graphs representing different aspects and constraints relevant to bus are combined (through unification) to produce the final composite representation.



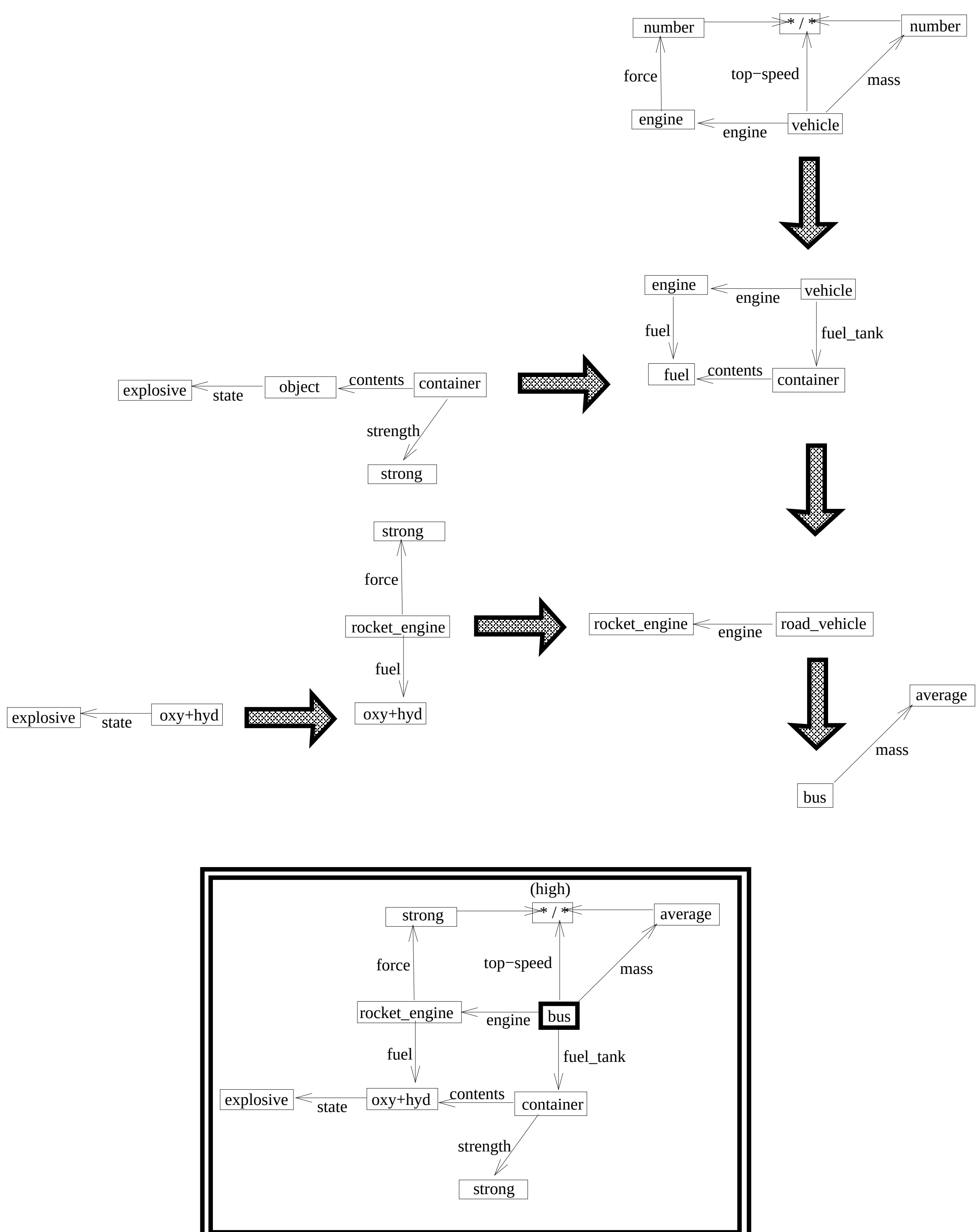


Creation of the novel concept ‘Rocket Bus’, as a modification of ‘Bus’

To make this new concept, petrol_engine has been replaced with rocket_engine. As a result:

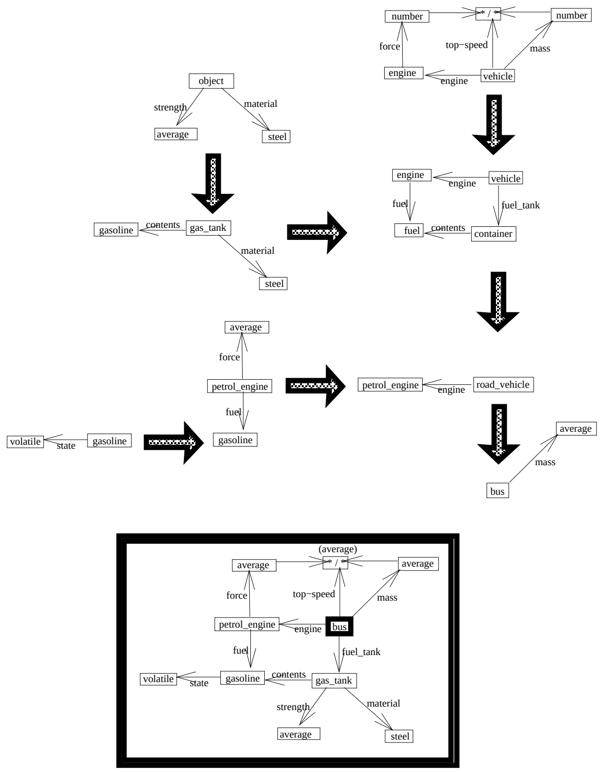
(i) some of the old components no longer ‘fit’ (and hence are dropped)

(ii) some new components are brought in (eg. that the fuel tank must be strong)

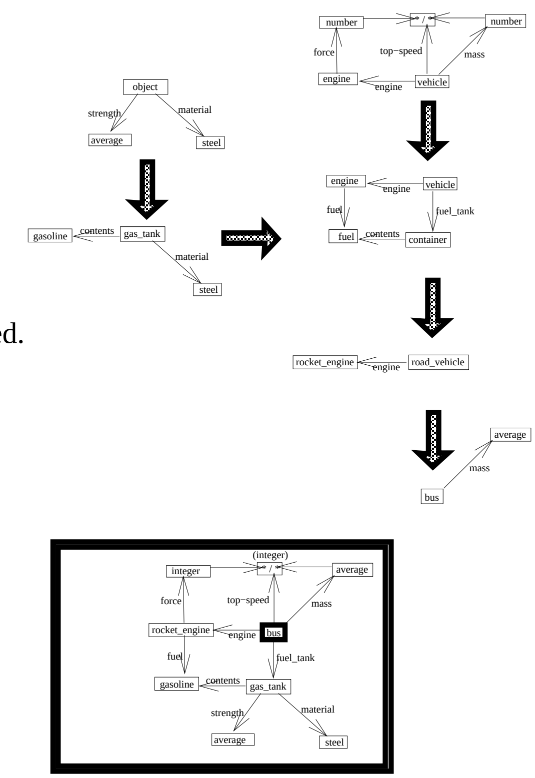


Creation of novel concept ‘Rocket Bus’ by modifying ‘Bus’

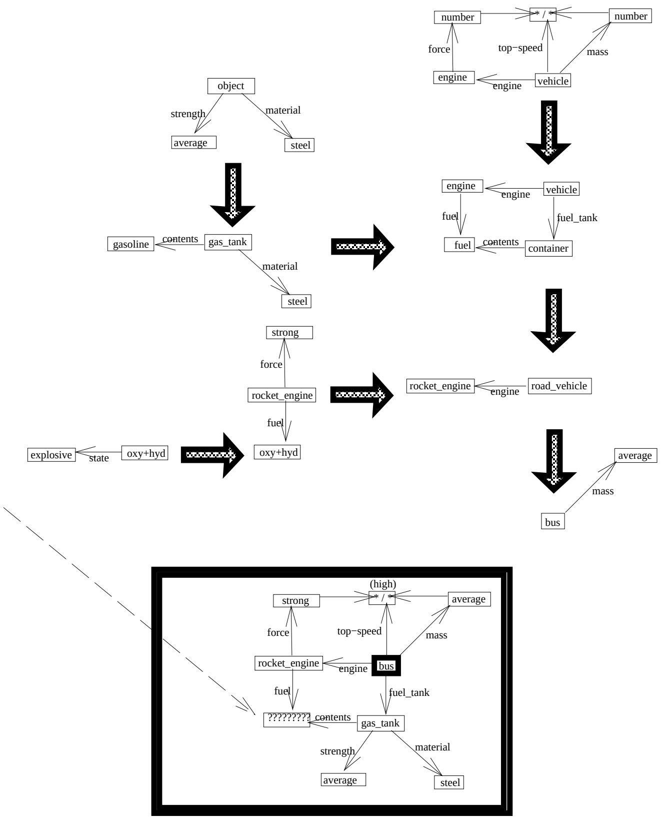
1. Starting point –– the representation of a normal bus, as constructed from components



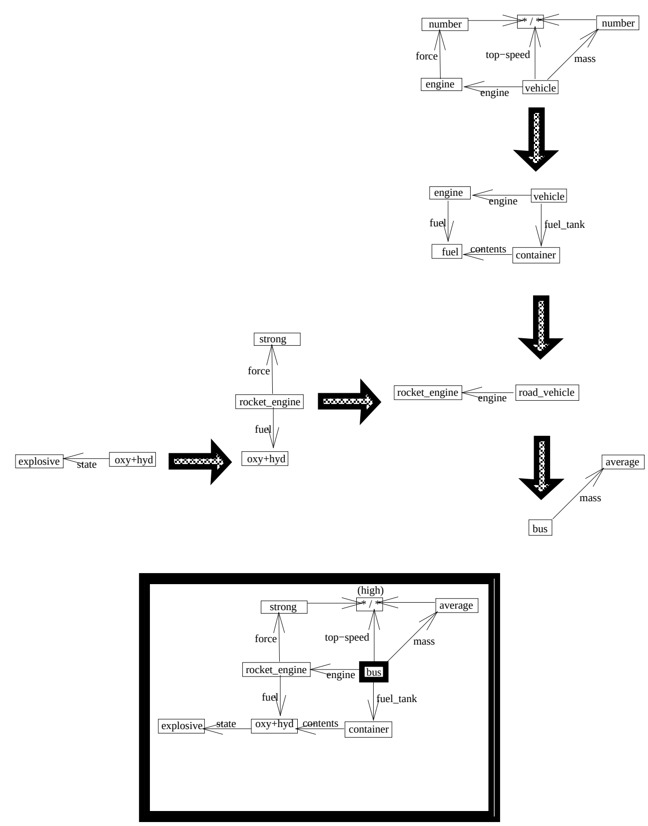
2. Replace petrol_engine with rocket_engine.
The components which "petrol_engine" pulled in are dropped.

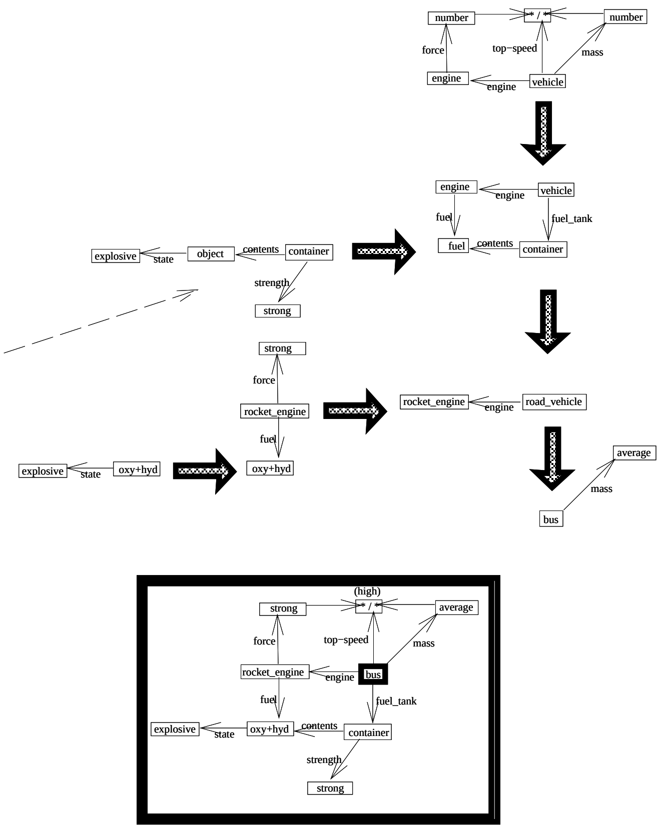


3. rocket_engine pulls in its own components.
   This causes a clash as to the fuel of the engine.

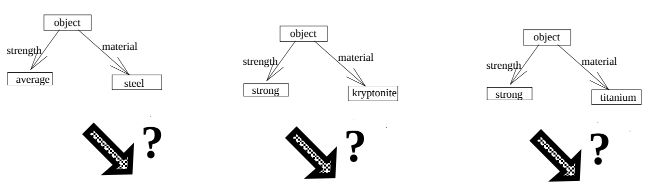


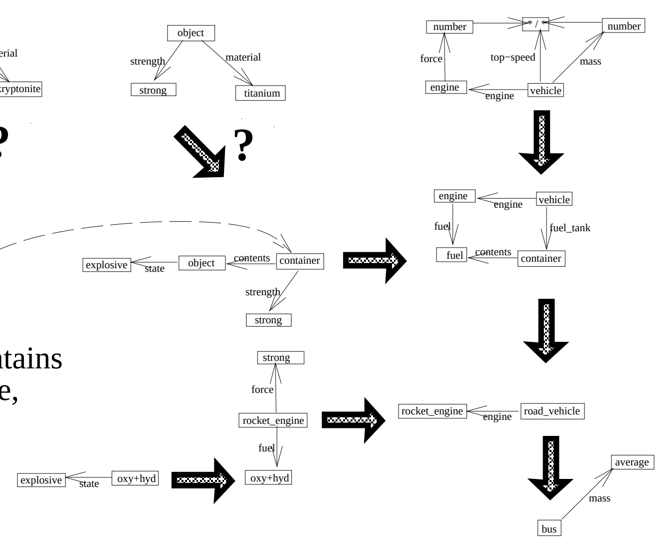
4. Under user−guidance, oxy+hyd wins the conflict,
   and so the gas−tank component (and the components
   it pulls in) is forced out.



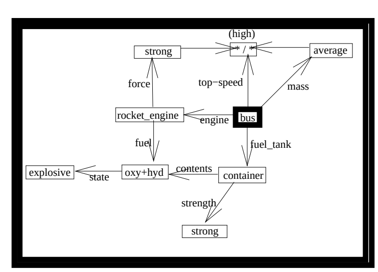


5. Now the container is known to contain an explosive liquid. This itself pulls in another component, stating that the container must be strong.





6. If the user asks for the material of the fuel container (say), the system tries to find a component which contains an answer and which fits the current description. Here, two possibilities fit, stating the material = kryptonite or material = titanium respectively.



Appendix

```
% APPENDIX: Representation of the previously described components for
% Bus in the programming language LIFE.

% "Petrol engines use gasoline, and produce average force."
:: petrol_engine(fuel=>gasoline, force=>av).

% "A vehicle's top speed is it's engine's force / vehicle mass."
:: vehicle(top_speed=>qdiv(F,M), engine=>engine(force=>F), mass=>M).

% "A bus has an average mass."
:: bus(mass=>av).

% "Road vehicles have petrol engines."
:: road_vehicle(engine=>petrol_engine).

% "A vehicle's fuel-tank contains the engine's fuel."
:: vehicle(engine=>engine(fuel=>F),
   fuel_tank=>container(contents=>F)).

% "Gas tanks are made of steel."
:: gas_tank(material=>steel).

% "Gasoline is volatile."
:: gasoline(state=>volatile).

% 3 daemons:
% "Kryptonite things are strong and heavy."
% "Titanium things are strong and light."
% "Steel things are average strength and average weight."
:: O:object | constraint(O).
constraint(O:object(material=>kryptonite)) -> O = @(strength=>hi,weight=hi).
constraint(O:object(material=>titanium)) -> O = @(strength=>hi,weight=lo).
constraint(O:object(material=>steel)) -> O = @(strength=>av,weight=>av).
constraint -> succeed.

% Inheritance hierarchy
petrol_engine <| engine.
bus <| road_vehicle.
road_vehicle <| vehicle.
gas_tank <| container.
container <| object.
gasoline <| liquid.

% Rules for qualitative division:
qdiv(hi,hi) -> av. qdiv(av,hi) -> lo. qdiv(lo,hi) -> lo.
qdiv(hi,av) -> hi. qdiv(av,av) -> av. qdiv(lo,av) -> lo.
qdiv(hi,lo) -> hi. qdiv(av,lo) -> hi. qdiv(lo,lo) -> av.

demo :- X = bus, X.fuel_tank = gas_tank, pretty_write(X).

% > demo?
% bus(engine => petrol_engine(force => av,
%                             fuel => A: gasoline(state => volatile)),
```

```
%     fuel_tank => gas_tank(contents => A,
%                           material => steel,
%                           strength => av,
%                           weight => av),
%     mass => av,
%     top_speed => av).
```