Modelling a Framework for Plugins

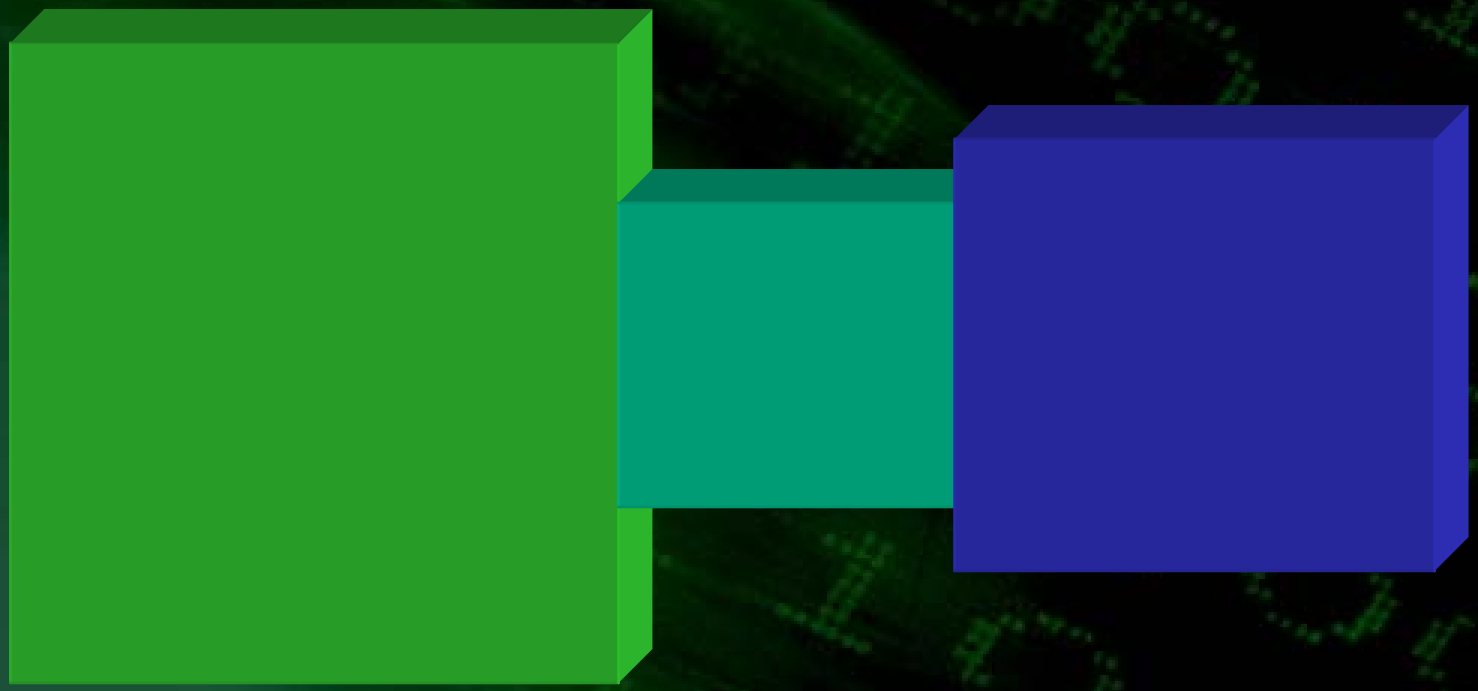
Robert Chatley, Susan Eisenbach and Jeff Magee

Imperial College London

Extensible Applications

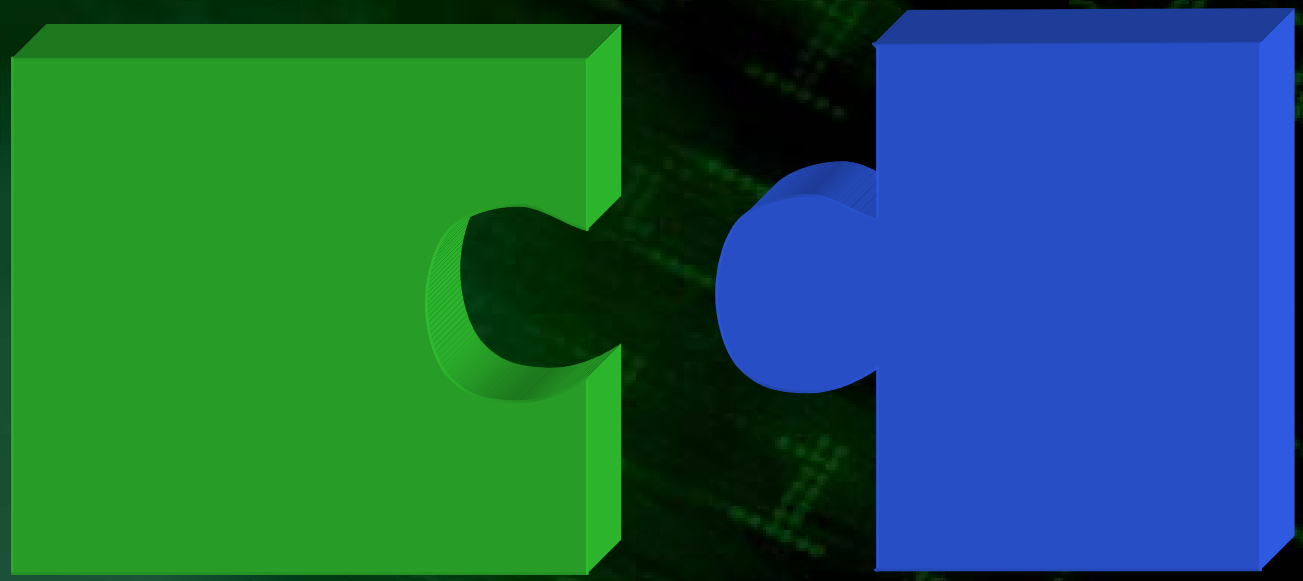
- Extend system functionality
- Decompose large systems
  - Configure for different situations
  - Software product lines [Bosch]
- Third party extensions
- Unanticipated Software Evolution
  - Requirements change over time
  - Fixes and upgrades required
  - May require system is not restarted

Extending Applications



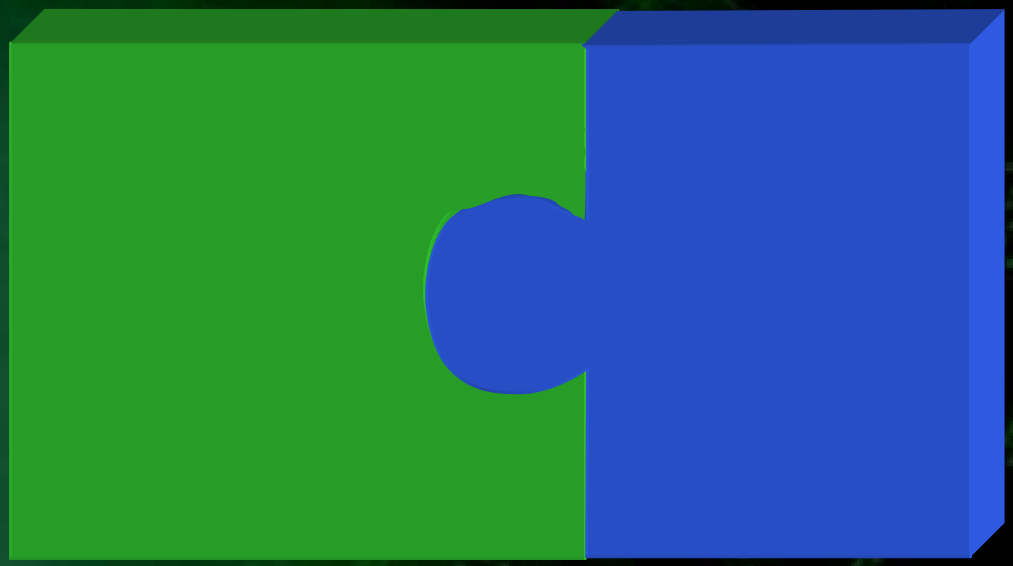
- Glue code needed between each pair of components

Plugin Components

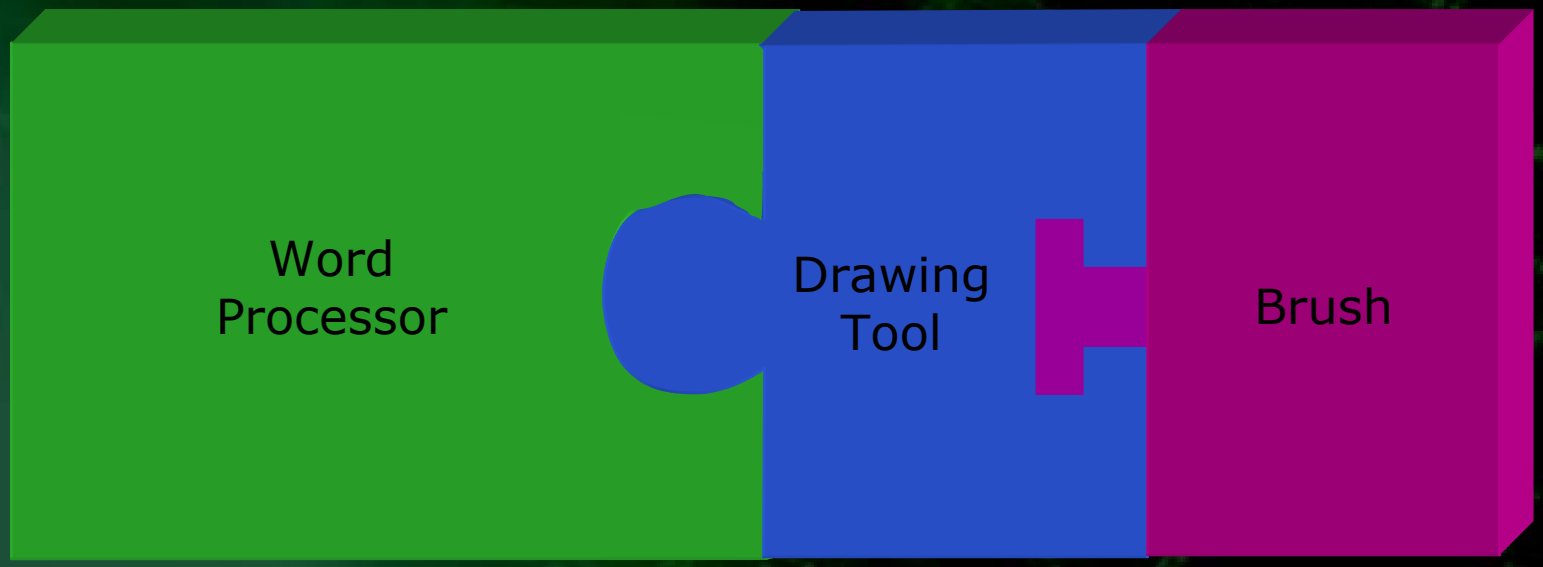


c.f. Required and Provided interfaces

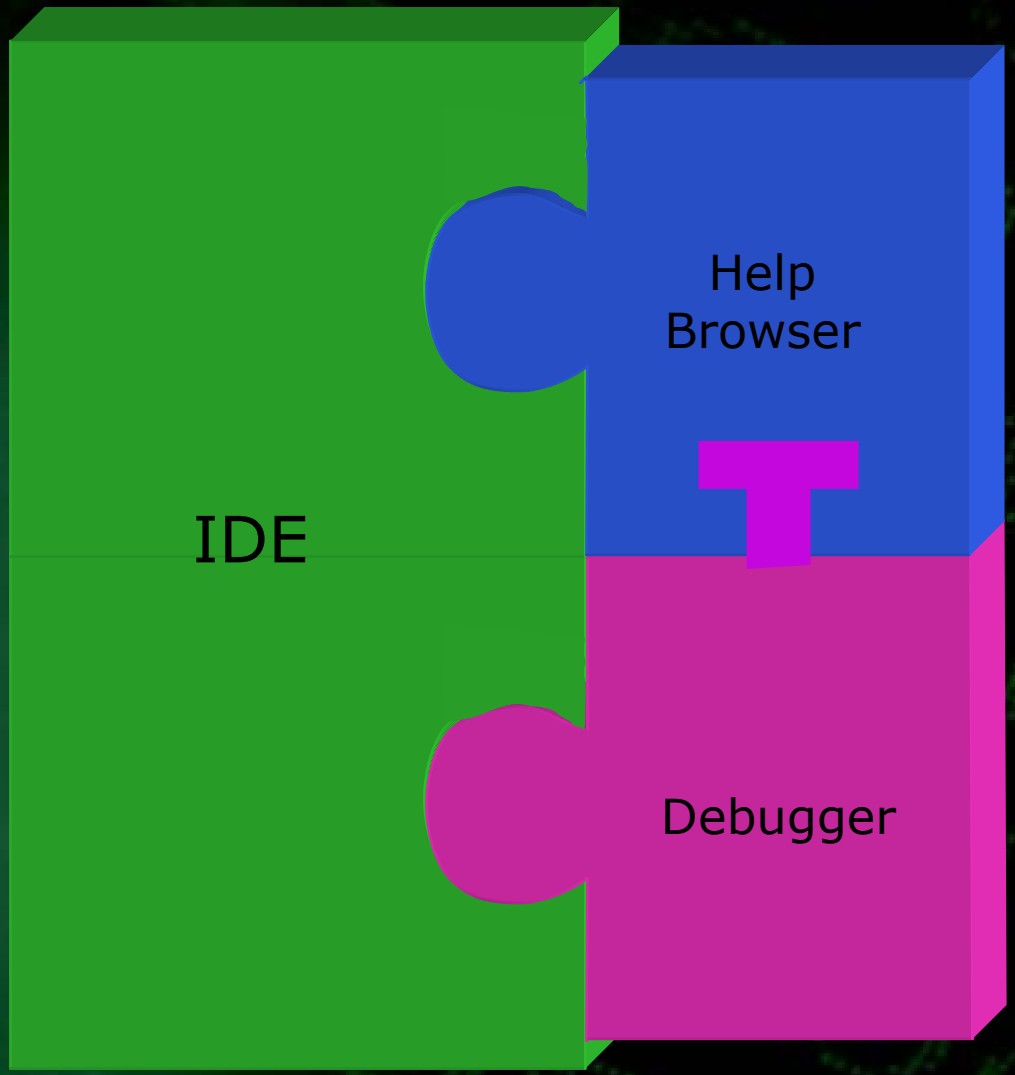
Plugin Components



Jigsaw analogy



Another example configuration



Components comprise…

- Holes are interfaces
- Pegs are concrete classes implementing interfaces
- May contain other classes, graphics…
- Nothing else – no manifest or ADL

Fitting Plugins Together

- Plugins fit together if they have a peg and hole that match
  - One has a required interface (hole)
  - The other provides a class (peg) whose interface is a subtype of the required interface
- When a match is found a binding can be created between the two components
- A system comprises a set of connected plugins, assembled at runtime

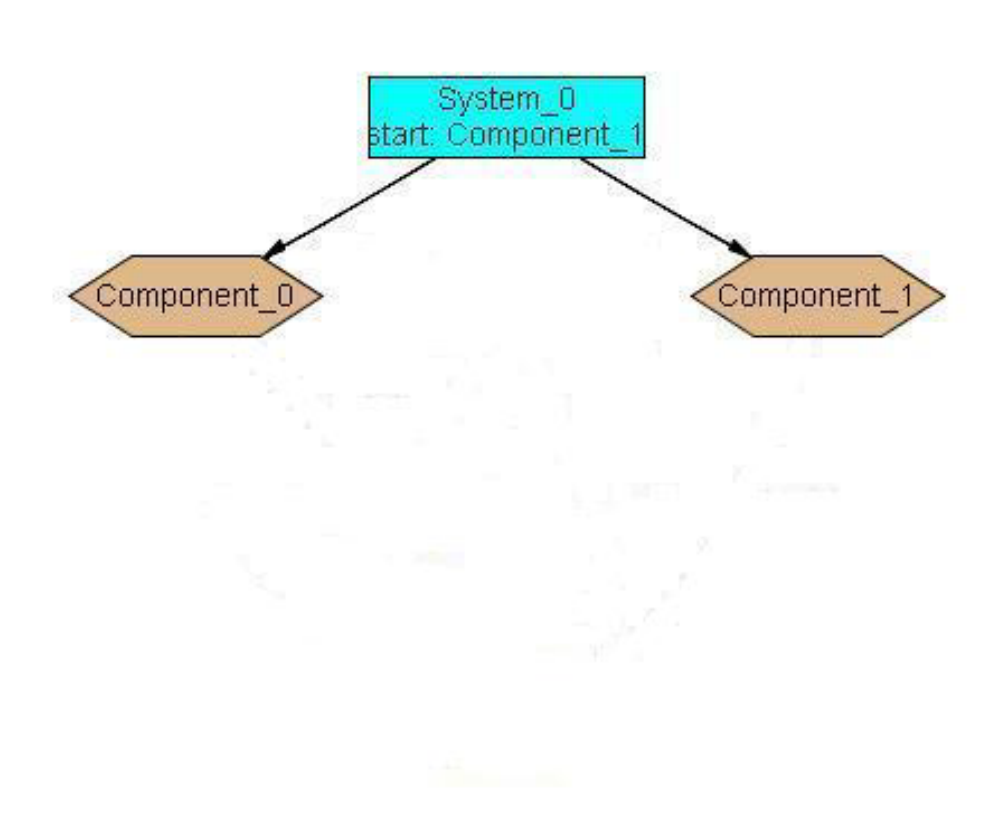
Modelling

- Formal model in Alloy [D. Jackson]
  - Language for describing relational structures
  - Tool supported analysis
  - Generates example situations
- Allows precise specification of structural rules
- Analysis throws up design questions

Alloy Example

```
sig System {
   components : set Component,
   start      : Component,

}
```



(this is just a fragment of the full model)

Alloy Example

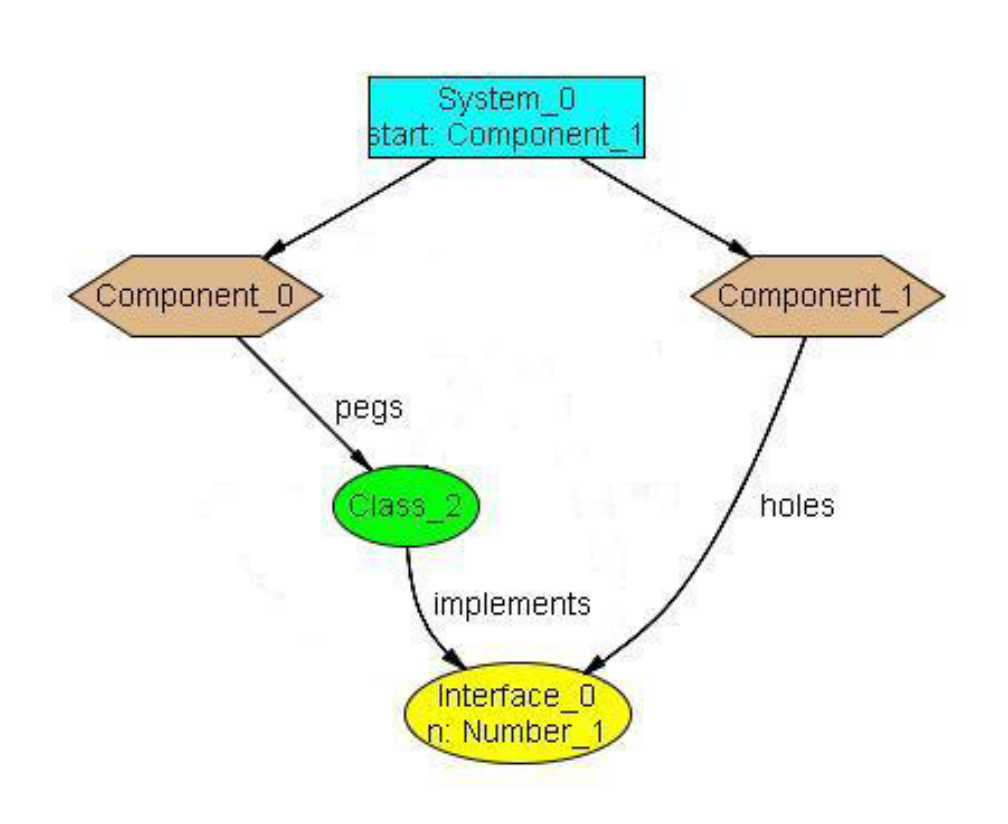
```
sig System {

   components : set Component,
   start      : Component,


}

sig Component {

   pegs  : set Class,
   holes : set Interface
}
```



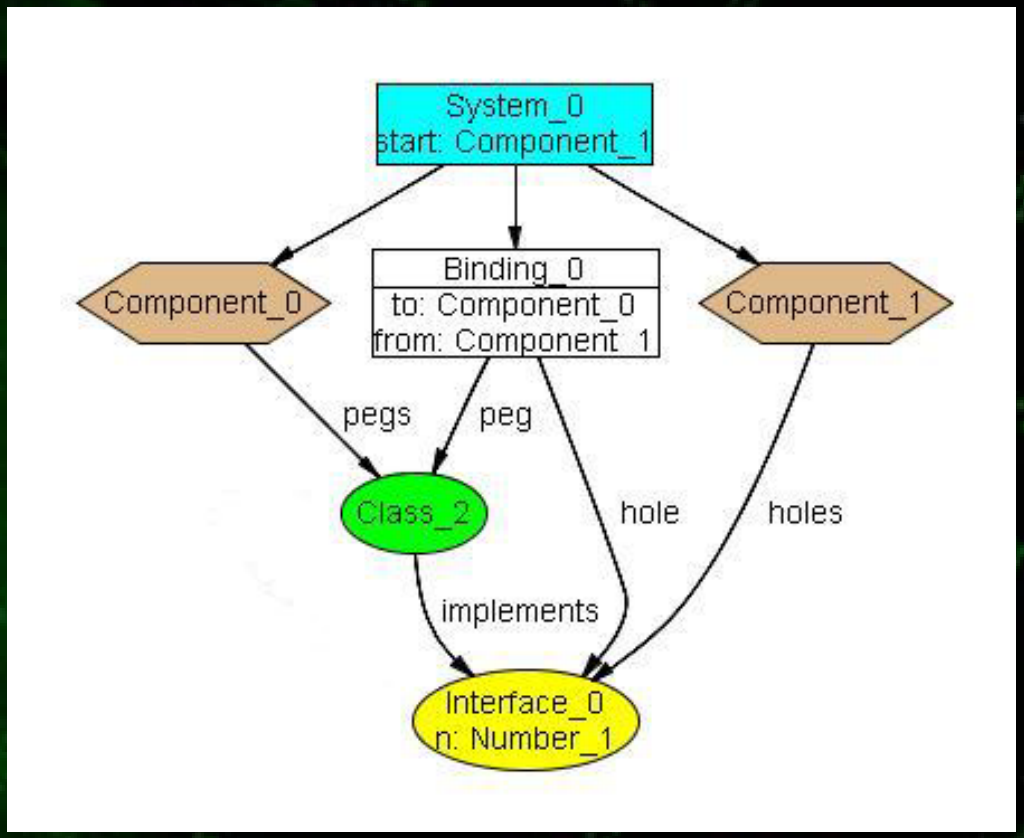
(this is just a fragment of the full model)

Alloy Example

```
sig System {
   components : set Component,
   start      : Component,
   bindings   : set Binding
}
sig Component {

   pegs  : set Class,
   holes : set Interface
}
sig Binding {

   hole : Interface,
   from : Component,
   peg  : Class,
   to   : Component
} {
   to != from
   hole in from.holes
   peg in to.pegs
}
```

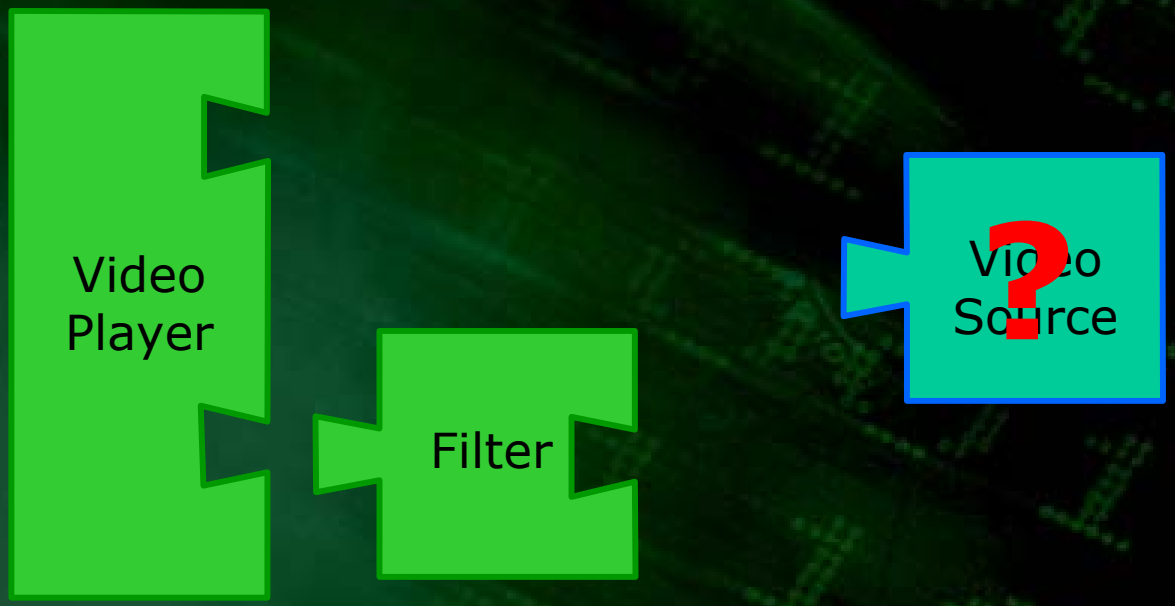


(this is just a fragment of the full model)

Cardinality Constraints

- May wish to constrain how many plugins can be connected to certain components
  - When resources are limited
  - To constrain structure
- Add a number to an interface that represents the number of plugins that can still be added to that interface
- In the addition function:
  - Check the number is greater that zero
  - Decrement number when addition made

Non-determinism



- For a deterministic system
  - Need to be able to specify binding policy
  - Choose between binding candidates

Binding Strategies

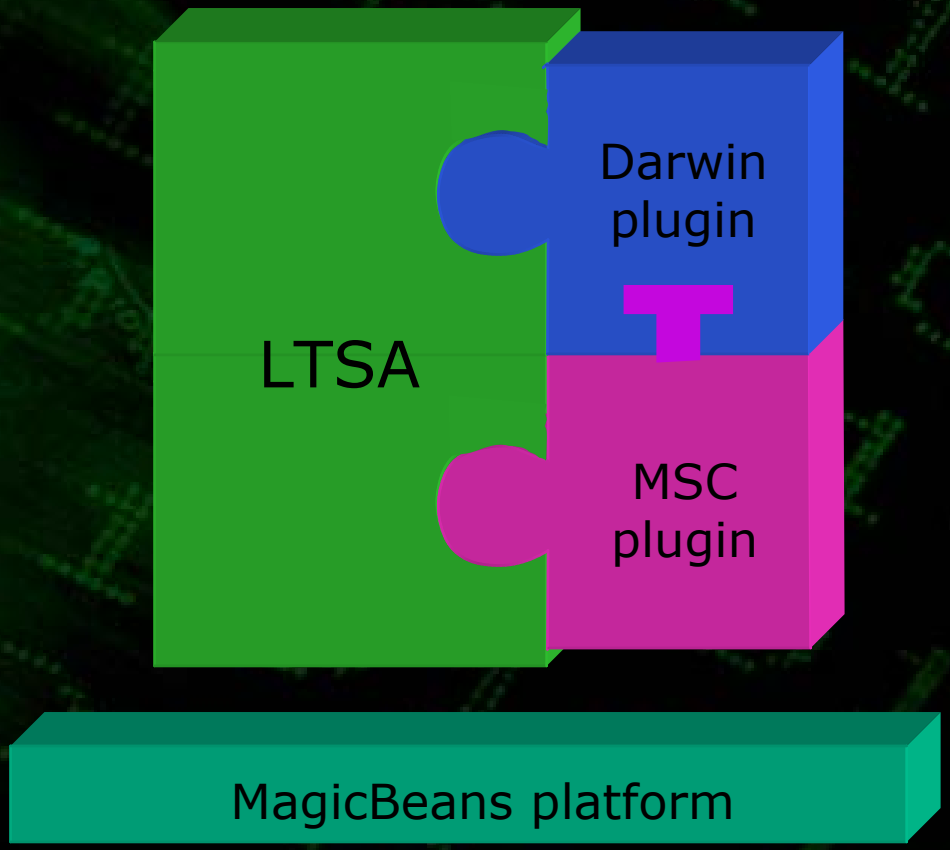
- Given multiple possible components to bind to, which is best?
  - No general answer
- Specify binding policy, e.g. using preference functions
  - Plugin developer defines pairwise function(s)
  - Plug functions in to framework

```
C prefer( C bindThis, C toThis, C orThis )
```

- Add or remove functions from set to change or refine policy

Implementation - the MagicBeans platform

- Codifies the model presented
- Written in Java to manage plugin applications written in Java
- Used to implement experimental programs and new version of LTSA tool
  - MSC plugin
  - Darwin plugin
  - Web animation plugin
  - BPEL plugin
  - A/G Reasoning plugin (NASA)



Summary and Future Work

- Model of connecting plugin components at runtime based on matching interfaces
  - Model refined using Alloy
  - Runtime implemented
- Future work
  - Removing and replacing plugins
  - Modelling specific configurations
  - Behaviour models