Alloy Specification of Backbone

Organisation of Alloy Modules

Figure 1 shows the dependencies of the individual Alloy modules in the specification.

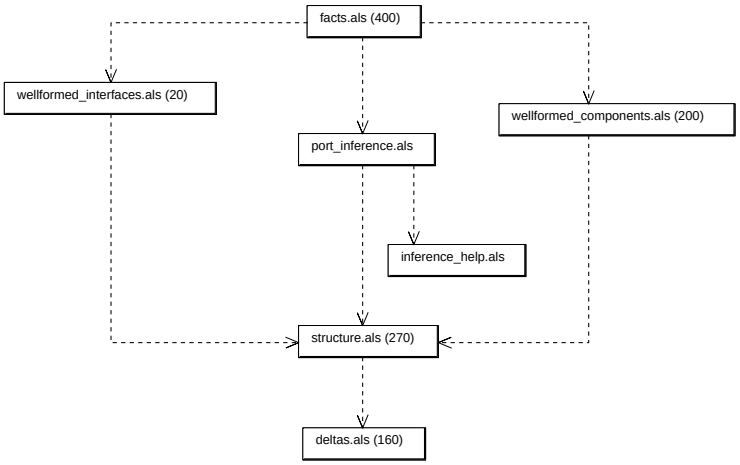

Figure 1: Module structure of the specification

Main Signatures

A UML class diagram of the main signatures is shown in figure 2.

Constituent Signatures

The consituent signatures are shown in figure 3.

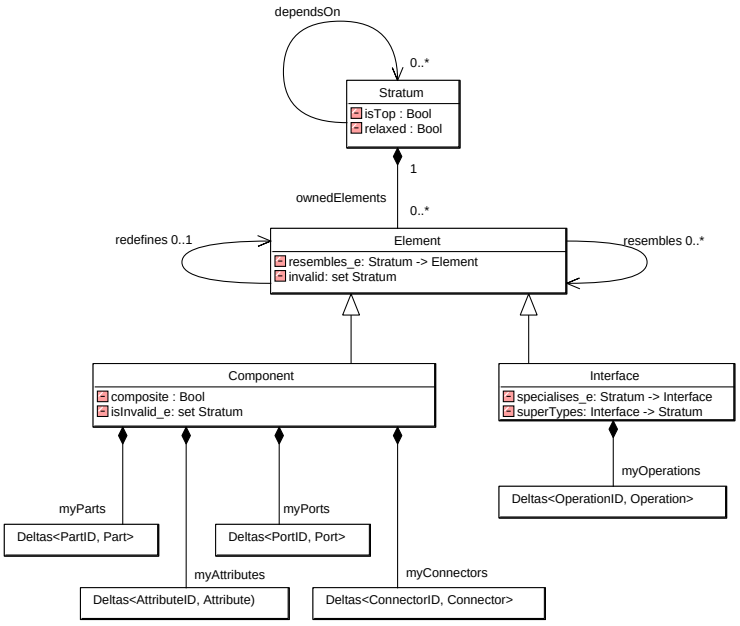

Figure 2: The main signatures of the Backbone specification

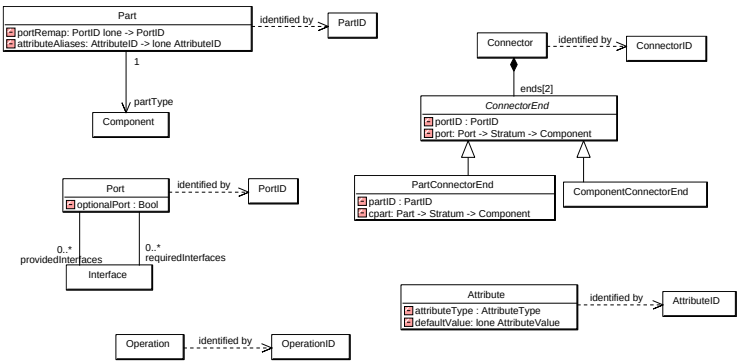

Figure 3: The constituent figures for the Backbone specification

Structure.als

Listing 1: structure.als

```
1   module structure
2
3   open util/boolean as boolean
4   open util/relation as relation
5   open deltas[Stratum, Component, PartID, Part] as Parts
6   open deltas[Stratum, Component, PortID, Port] as Ports
7   open deltas[Stratum, Component, ConnectorID, Connector] as Connectors
8   open deltas[Stratum, Component, AttributeID, Attribute] as Attributes
9   open deltas[Stratum, Interface, OperationID, Operation] as Operations
10  open deltas[Stratum, Interface, InterfaceImplementationID, InterfaceImplementation] as
        InterfaceImplementation
11
12
13  one sig Model
14  {
15    -- normally this should be set to none
16    errorsAllowed: set Stratum,
17    providesIsOptional: Bool
18  }
19
20  sig Stratum
21  {
22    -- strata that this directly depends on
23    dependsOn: set Stratum,
24    -- does this expose stratum it depends on?
25    isRelaxed: Bool,
26
27    -- derived state -- exposes strata includes this and canSee
28    exposesStrata: set Stratum,
29    canSee: set Stratum,
30    canSeePlusMe: set Stratum,
31
32    -- simple is all that we directly depend on
33    -- taking away what any children depend on
34    simpleDependsOn: set Stratum,
35
36    -- a single top exists which binds directly
37    -- any independent stratum
38    isTop: Bool,
39    -- this is every stratum that can be seen from here down
40    transitive: set Stratum,
41    transitivePlusMe: set Stratum,
42    ownedElements: set Element,
43    -- each attribute type is owned by a single stratum
44    attributeTypes: set AttributeType,
45
46    -- elements that redefine another
47    redefining: set Element,
48    -- components that are new definitions
49    defining: set Element
50  }
51  {
52    defining = ownedElements - redefining
53    canSeePlusMe = canSee + this
54    transitivePlusMe = transitive + this
55  }
56
57  abstract sig Element
58  {
59    home: Stratum,
60    redefines: lone Element,
61    resembles: set Element,
62
63    -- for a given stratum, a component resembles other components in a given stratum view
64    resembles_e: Element -> Stratum,
65    -- does this act as a non-primed for a particular stratum
66    actsAs_e: Element -> Stratum,
67    -- is this element valid for a given stratum?
68    isInvalid_e: set Stratum
69  }
70  {
71    -- owned by a single stratum
72    home = ownedElements.this
73  }
74
75  sig Component extends Element
76  {
77    -- composite or leaf?
78    -- all components apart from direct impl components are always composite
79    isComposite: Bool,
80    implementation: lone ComponentImplementation,
81
82    -- the final result, after taking redef + resemblance into account
```

```
    iDParts: PartID -> Part -> Stratum,
    parts: Part -> Stratum,
    ports: Port -> Stratum,
    connectors: Connector -> Stratum,
    attributes: Attribute -> Stratum,

    myParts: lone Parts/Deltas,
    myPorts: lone Ports/Deltas,
    myConnectors: lone Connectors/Deltas,
    myAttributes: lone Attributes/Deltas,

    -- the internal links, used for port type inferencing
    links: PortID -> PortID,
    inferredLinks: Port -> Port -> Stratum
  }
  {
    -- for components
    redefines + resembles in Component
    -- propagate up the objects from the delta into the sig, to make it more convenient
    parts = myParts.objects
    ports = myPorts.objects
    connectors = myConnectors.objects
    attributes = myAttributes.objects

    -- a leaf has an implementation
    isFalse[isComposite] <=> one implementation

    -- form idParts
    iDParts = {n: PartID, p: Part, s: Stratum | s -> n -> p in myParts.objects_e}

    -- make sure links has no duplication
    no ~links & links

    -- we only have links for leaves, so all ports must be new
    dom[links] + ran[links] in myPorts.newIDs
  }

  -- ensure each delta is composed by only one component
  fact
  {
    all p: Parts/Deltas | one myParts.p
    all p: Ports/Deltas | one myPorts.p
    all c: Connectors/Deltas | one myConnectors.c
    all a: Attributes/Deltas | one myAttributes.a
  }

  sig Interface extends Element
  {
    -- the expanded elements
    operations: Operation -> Stratum,
    implementation: InterfaceImplementation -> Stratum,
    superTypes: Interface -> Stratum,

    -- the deltas
    myOperations: Operations/Deltas,
    myImplementation: InterfaceImplementation/Deltas
  }
  {
    -- for interfaces
    redefines + resembles in Interface
    -- propagate up the objects from the delta into the sig, to make it more convenient
    operations = myOperations.objects
    implementation = myImplementation.objects
  }
  -- ensure each delta is composed by only one interface
  fact
  {
    all p: Operations/Deltas | one myOperations.p
    all i: InterfaceImplementation/Deltas | one myImplementation.i
  }


  -- each artifact must have a id, so it can be replaced or deleted
  sig PartID, PortID, ConnectorID, AttributeID, OperationID, InterfaceImplementationID {}

  sig Part
  {
    partType: Component,
    -- remap a port from this part onto the port of a part that we are replacing
    -- (new port -> old, replaced port)
    portRemap: PortID lone -> lone PortID,
    portMap: Stratum -> PortID lone -> lone Port,

    -- the values of the attributes are set in the part   (child id -> parent id)
    -- although they don't have to be set if we want to take the default
    attributeValues: AttributeID -> lone AttributeValue,
    -- do we alias a parent attribute?
    attributeAliases: AttributeID -> lone AttributeID,
    -- or do we simply copy a parent attribute, but retain our own state?
    attributeCopyValues: AttributeID -> lone AttributeID,

```

```
174     -- derived state -- the parts that the connectors link to
175     linkedToParts: Part -> Stratum -> Component,
176     -- derived state -- any compons that the connectors link to
177     linkedToOutside: Stratum -> Component
178   }
179
180   abstract sig Index {}
181   one sig Zero, One, Two, Three extends Index {}
182
183   pred isContiguousFromZero(indices: set Index)
184   {
185     indices = indices.*(Three->Two + Two->One + One->Zero)
186   }
187
188   sig Port
189   {
190     -- set values are what the user has explicitly set
191     setProvided, setRequired: set Interface,
192     -- provided and required are inferred
193     provided, required: Interface -> Stratum -> Component,
194     mandatory, optional: set Index
195   }
196   {
197     -- mandatory indices start at 0, optional start from mandatory end, no overlap
198     -- all contiguous and must have some indices
199     isContiguousFromZero[mandatory] and
200       isContiguousFromZero[mandatory + optional]
201     no mandatory & optional       -- no overlap
202     some mandatory + optional     -- but must have some indices
203   }
204
205
206   sig Connector
207   {
208       -- require 2 ends
209       ends: set ConnectorEnd
210   }
211   {
212     -- ensure 2 connector ends using a trick felix taught me
213     some disj end1, end2: ConnectorEnd | ends = end1 + end2
214       all end: ends |
215           end.otherEnd = ends - end
216   }
217
218   abstract sig ConnectorEnd
219   {
220       portID: PortID,
221       port: Port -> Stratum -> Component,
222       index: Index,
223       otherEnd: ConnectorEnd
224   }
225   {
226       -- an end is owned by one connector
227       one ends.this
228   }
229
230   sig ComponentConnectorEnd extends ConnectorEnd
231   {
232   }
233
234   sig PartConnectorEnd extends ConnectorEnd
235   {
236       partID: PartID,
237       cpart: Part -> Stratum -> Component
238   }
239
240
241   sig Attribute
242   {
243     attributeType: AttributeType,
244     defaultValue: lone AttributeValue
245   }
246   {
247     some defaultValue =>
248       defaultValue.valueType = attributeType
249   }
250
251   sig AttributeValue
252   {
253     valueType: AttributeType
254   }
255
256   sig AttributeType
257   {
258   }
259   {
260     -- owned by one stratum
261     one attributeTypes.this
262   }
263
264   sig Operation
```

```
{
  -- this identifies the impelementation id and signature
}

sig InterfaceImplementation
{
  -- this identifies the interface implementation clas or no s.dependsOns...
}

sig ComponentImplementation
{
  -- this identifies the component implementation class...
}

-- used for port inference
-- a bit like a connector, but multiplicity and optionality don't count
abstract sig LinkEnd
{
  linkPortID: PortID,
  linkError: Stratum -> Component
  -- the internal interfaces are the interfaces presented inside the component content area
  --   for a port, it is the interfaces seen internally (opposite)
  --   for a port instance, it is the interfaces seen externally (same)
}

sig ComponentLinkEnd extends LinkEnd
{
}

sig PartLinkEnd extends LinkEnd
{
  linkPartID: PartID
}

-- translate from port id to component link end -- guaranteed to be 1 per id
fun getComponentLinkEnd(id: PortID): one ComponentLinkEnd
{
  { end: ComponentLinkEnd | end.linkPortID = id }
}

-- translate from a port/part to a part link end -- guaranteed to be 1 per pair
fun getPartLinkEnd(portID: PortID, partID: PartID): PartLinkEnd
{
  { end: PartLinkEnd | end.linkPortID = portID and end.linkPartID = partID }
}

fun ComponentLinkEnd::getPort(s: Stratum, c: Component): one Port
{
  c.myPorts.objects_e[s][this.linkPortID]
}

fun PartLinkEnd::getPortInstance(s: Stratum, c: Component): Port -> Part
{
  let
    cpart = c.myParts.objects_e[s][this.linkPartID],
    cport = cpart.partType.myPorts.objects_e[s][this.linkPortID] |
  cport -> cpart
}

-- get the port of a component connector
fun ComponentConnectorEnd::getPort(s: Stratum, c: Component): lone Port
{
    c.myPorts.objects_e[s][this.portID]
}

-- should return only 1 Port, unless the component is invalid. NOTE: the component owns the part
fun PartConnectorEnd::getPortInstance(s: Stratum, c: Component): Port -> Part
{
    let
        ppart = c.myParts.objects_e[s][this.partID],
        port = ppart.portMap[s][this.portID] |
    port -> ppart
}

fun PartLinkEnd::getPortInstanceRequired(s: Stratum, c: Component): set Interface
{
  let portPart = this.getPortInstance[s, c],
    pport = dom[portPart],
    ppartType = ran[portPart].partType
  {
    pport.required.ppartType.s
  }
}

fun PartLinkEnd::getPortInstanceProvided(s: Stratum, c: Component): set Interface
{
  let portPart = this.getPortInstance[s, c],
    pport = dom[portPart],
    ppartType = ran[portPart].partType
  {
    pport.provided.ppartType.s
```

```
356     }
357   }
```

Facts.als

Listing 2: facts.als

```
module facts

open util/boolean as boolean
open util/relation as relation
open util/ternary as ternary
open structure
open port_inference
open wellformed_components
open wellformed_interfaces
open stratum_help
open redefinition_types

-----------------------------------------------------------------------

fact StratumFacts
{
  invalidateUnseenPorts[]
  one isTop.True
  all s: Stratum
  {
    -- for relaxed, expose what it depends on and their exposures
    isTrue[s.isRelaxed] =>
      s.exposesStrata = s + s.dependsOn.exposesStrata
    else
      s.exposesStrata = s

    -- used for partial ordering
    -- contains nothing the children already depend on
    s.simpleDependsOn = s.dependsOn -
      s.dependsOn.transitive

    -- no cycles
    s not in s.^dependsOn

    -- can only see what others expose
    s.canSee = s.dependsOn.exposesStrata

    -- the strata we can see using the dependency graph
    s.transitive = s.^dependsOn

    -- a stratum is called top if no stratum depends on it
    isTrue[s.isTop] <=> no simpleDependsOn.s

    -- have max of one redefinition of an element per stratum
    all e: Element |
      lone s.ownedElements & redefines.e
    -- ties up redefining and redefines
    s.redefining = s.ownedElements & dom[redefines]
  }
}

pred independent[stratum1, stratum2: Stratum]
{
  stratum2 not in stratum1.*dependsOn
}

pred mutuallyIndependent[a, b: Stratum]
{
  independent[a, b] and independent[b, a]
}

fun stratumPerspective[stratum: Stratum]: set Stratum
{
  stratum.*dependsOn
}

-----------------------------------------------------------------------
-- handle the basics of resemblance(specialisation) and redefinition
-----------------------------------------------------------------------

fact ElementFacts
{
  -- nothing can resemble a redefinition -- check to see that the things we resemble don't redefine
  no resembles.redefines

  all
    e: Element |
  let
    owner = e.home,
    -- strata that can see the component
    resemblingOwningStratum = e.resembles.home,
    redefiningOwningStratum = e.redefines.home
  {
```

```
84        -- no circularities in resemblance or redefinition, and must be visible
85        e not in (e.^resembles + e.redefines)
86        resemblingOwningStratum in owner.canSeePlusMe
87        redefiningOwningStratum in owner.canSee
88
89        -- tie up the owning stratum and the elements owned by that stratum
90        e.home = ownedElements.e
91
92        -- we only need to form a definition for stratum that can see us
93        all s: Stratum |
94        let
95          -- who should I resemble
96          -- (taking redefinition into account)
97          iResemble = e.resembles_e.s,
98          -- if we resemble what we are redefining,
99          -- look for the original under here
100         topmostOfRedefined = getTopmost[
101           owner.simpleDependsOn,
102           e.redefines & e.resembles],
103         -- look for any other resembled components
104         -- from here down
105         topmostOfResemblances = getTopmost[
106           s,
107           e.resembles - e.redefines]
108       {
109         -- who do we act as in this stratum?
110         e.actsAs_e.s =
111           { real: Element | no real.redefines and e in getTopmost[s, real] }
112         -- rewrite the resemblance graph to handle redefinition
113         owner not in s.transitivePlusMe =>
114           no iResemble
115         else
116           iResemble = topmostOfRedefined + topmostOfResemblances
117       }
118     }
119   }
120
121   fun getTopmost(s: set Stratum, e: Element): set Element
122   {
123     let redefined = redefines.e & s.transitivePlusMe.redefining,
124       topmostRedefined = redefined - redefined.resembles_e.s
125         { some topmostRedefined => topmostRedefined else e }
126   }
127
128   ------------------------------------------------------------------------
129   -- handle any extra rules for interfaces
130   ------------------------------------------------------------------------
131
132   fact InterfaceFacts
133   {
134     all i: Interface |
135     let
136       owner = i.home
137     {
138       -- we only need to form a definition for stratum that can see us
139       all s: Stratum |
140       let
141         invalid = s in i.isInvalid_e,
142         visible = owner in s.transitivePlusMe
143       {
144         -- if we can see this component, test to see if it is valid in this stratum
145         not visible =>
146           invalidateUnseenInterface[s, i]
147         else
148         {
149           -- ensure that the subtypes are set up correctly
150           -- this is a subtype of an interface if we can reach it transitively and
151           -- our set of operations are a super-set of the super type's
152           -- of operationID -> Operation.  i.e. if you replace an operationID you are breaking subtype
153           -- so the operationID is the name, and the Operation is the full spec which is assumed to have
154           -- changed in a redef...
155           -- NOTE: the super types are direct -- to follow use closure
156           --       also note that we only need/want supertypes for non-primes
157           no i.redefines =>
158           {
159             i.superTypes.s =
160               { super: i.resembles |
161                 super.myOperations.objects_e[s] in
162                   i.myOperations.objects_e[s] }
163           }
164           else
165             no i.superTypes.s
166
167           -- merge any parts and apply changes
168           let topmost = getTopmost[s, i] & Interface
169           {
170             i.myOperations::mergeAndApplyChangesForResemblance[
171               s, i, i.resembles_e.s.myOperations]
172             i.myOperations::mergeAndApplyChangesForRedefinition[
173               s, i, topmost, topmost.myOperations]
174             i.myImplementation::mergeAndApplyChangesForResemblance[
```

```
175               s, i, i.resembles_e.s.myImplementation]
176             i.myImplementation::mergeAndApplyChangesForRedefinition[
177               s, i, topmost, topmost.myImplementation]
178           }
179
180           -- the component must be valid in the place it was defined
181           -- also, in any stratum (apart from the top one), a visible component must be valid
182           (s = owner or s not in Model.errorsAllowed) => !invalid
183           s = owner =>
184           {
185             i.myOperations.deltasIsWellFormed[s]
186             i.myImplementation.deltasIsWellFormed[s]
187           }
188
189           -- a component is valid if it is well formed...
190           !invalid <=> interfaceIsWellFormed[s, i]
191         }
192       }
193     }
194   }
195
196
197   ------------------------------------------------------------------------
198   -- handle any extra rules for components
199   ------------------------------------------------------------------------
200
201   fact ComponentFacts
202   {
203     -- no part or port can refer explicitly to a redefined component or interface
204     no partType.redefines
205     no (setProvided + setRequired).redefines
206
207     all c: Component |
208     let
209       owner = c.home,
210       -- strata that the component can see
211       iCanSeePlusMe = owner.canSeePlusMe,
212       types = c.myParts.newObjects.partType,
213       attrTypes = c.myAttributes.newObjects.attributeType,
214       interfaces = c.myPorts.addedObjects.(setRequired + setProvided)
215     {
216       -- resemblance has no redundancy
217       c.resembles = c.resembles - c.resembles.^resembles
218
219
220       -- component implementations stay the same...
221       isFalse[c.isComposite] =>
222       {
223         -- no resemblance, redef, parts + noone resembles or redefines it
224         no c.myParts
225         no c.myConnectors
226         no resembles.c
227         no c.resembles
228       }
229       else
230         no c.links
231
232       -- parts can only have types from here down, excluding myself...
233       -- attribute types can only be from here down
234       -- ports can only refer to interfaces from here down
235       types.home in iCanSeePlusMe
236       attributeTypes.attrTypes in iCanSeePlusMe
237       interfaces.home in iCanSeePlusMe
238
239       -- ensure that the port remaps are correctly formed for the stratum they are owned by
240       let delta = c.myParts |
241       all p: delta.newObjects | -- parts of the delta
242       let
243         partID = delta.replaceObjects.p,
244         oldPart = delta.oldObjects_e[owner][partID],
245         remap = p.portRemap,
246         newPortIDs = dom[remap],
247         oldPortIDs = ran[remap]
248       {
249         -- we can only alias ports that we actually have
250         newPortIDs in dom[p.partType.myPorts.objects_e[owner]]
251         -- we can only use port ids of the component we are replacing
252         oldPortIDs in dom[oldPart.partType.myPorts.objects_e[owner]]
253
254         -- each port we remap should have a different id, or there's no point
255         -- this is not strictly needed, but ensures nice witnesses
256         bijection[remap, newPortIDs, oldPortIDs]
257
258         -- can't map a port id onto the same id
259         no remap & iden
260       }
261
262       -- we only need to form a definition for stratum that can see us
263       all s: Stratum |
264       let
265         invalid = s in c.isInvalid_e,
```

```
266         visible = owner in s.transitivePlusMe
267       {
268         -- if we can see this component, test to see if it is valid in this stratum
269         not visible =>
270           invalidateUnseenComponent[s, c]
271         else
272         let topmost = getTopmost[s, c] & Component
273         {
274           -- merge any parts and apply changes
275           c.myParts::mergeAndApplyChangesForResemblance[
276             s, c, c.resembles_e.s.myParts]
277           c.myParts::mergeAndApplyChangesForRedefinition[
278             s, c, topmost, topmost.myParts]
279
280           -- merge any ports and apply changes
281           c.myPorts::mergeAndApplyChangesForResemblance[
282             s, c, c.resembles_e.s.myPorts]
283           c.myPorts::mergeAndApplyChangesForRedefinition[
284             s, c, topmost, topmost.myPorts]
285
286           -- merge any connectors and apply changes
287           c.myConnectors::mergeAndApplyChangesForResemblance[
288             s, c, c.resembles_e.s.myConnectors]
289           c.myConnectors::mergeAndApplyChangesForRedefinition[
290             s, c, topmost, topmost.myConnectors]
291
292           -- merge any attributes and apply changes
293           c.myAttributes::mergeAndApplyChangesForResemblance[
294             s, c, c.resembles_e.s.myAttributes]
295           c.myAttributes::mergeAndApplyChangesForRedefinition[
296             s, c, topmost, topmost.myAttributes]
297
298           -- if we are "home", all the deltas must be well formed...
299           -- this is not necessarily the case if we are not home
300           s = owner =>
301           {
302             c.myParts.deltasIsWellFormed[s]
303             c.myPorts.deltasIsWellFormed[s]
304             c.myConnectors.deltasIsWellFormed[s]
305             c.myAttributes.deltasIsWellFormed[s]
306           }
307
308           setupParts[s, c]
309           setupConnectors[s, c]
310           isTrue[c.isComposite] =>
311             setupCompositeLinks[s, c]
312           else
313             setupLeafLinks[s, c]
314
315           -- the component must be valid in the place it was defined
316           -- also, in any stratum (apart from the top one), a visible
317           -- component must be valid
318           (s = owner or s not in Model.errorsAllowed) => !invalid
319
320           -- a component is invalid iff it is not well formed
321           invalid <=>
322             (!componentIsWellFormed[s, c] or !linksAreWellFormed[s, c])
323         }
324       }
325     }
326   }
327
328   pred setupParts(s: Stratum, c: Component)
329   {
330     -- reference the parts we are linked to and link to the outside if true
331     let allParts = c.parts.s
332     {
333       no (Part - allParts).linkedToParts.c.s
334       all pPart: allParts |
335       {
336         pPart.linkedToParts.c.s =
337         { p: allParts - pPart |
338           some end: c.connectors.s.ends |
339             end.cpart.c.s = pPart and end.otherEnd.cpart.c.s = p }
340
341         -- reference if we are linked to the outside of the component
342         s -> c in pPart.linkedToOutside <=>
343         {
344           some end: c.connectors.s.ends |
345             end.cpart.c.s = pPart and end.otherEnd in ComponentConnectorEnd
346         }
347       }
348     }
349
350     -- form the full port map for this stratum, taking remap into account
351     all p: c.myParts.newObjects |
352     let
353       -- turn the remap from id -> id to id -> port
354       idToPort = p.partType.myPorts.objects_e[s],
355       newPorts = idToPort[PortID],
356       remap =
```

```
357         { newPort: newPorts, oldID: PortID |
358           idToPort.newPort -> oldID in p.portRemap }
359     {
360       -- remove the existing ID of the port before adding the new one
361       ~(p.portMap[s]) =
362         ~(p.partType.myPorts.objects_e[s]) ++ remap
363     }
364   }
365
366
367   -- if the connector is not visibile to a component in a stratum, it should be zeroed out to
368   -- make it easier to interpret the results and and zeroOutUnseenElement[s, e] cut back on the state space
          for performance reasons
369   fact ZeroOutUnseenConnectorsFact
370   {
371     all conn: Connector, c: Component, s: Stratum |
372     conn not in c.connectors.s =>
373     {
374       all end: conn.ends
375       {
376         no end.port.c.s
377         no end.cpart.c.s
378       }
379     }
380   }
381
382   -- if the part is not visible to a component in a stratum, it should also be zeroed out
383   -- for understandability and performance reasons
384   fact ZeroOutUnseenPartsFact
385   {
386     all p: Part, c: Component, s: Stratum |
387     p not in c.parts.s =>
388     {
389       no p.linkedToParts.c.s
390       s-> c not in p.linkedToOutside
391     }
392   }
393
394   -- if the part is not visible to a component in a stratum, it should also be zeroed out
395   -- for understandability and performance reasons
396   -- NOTE: if you move the valid setting up into the main body, it gets slow for some bizarre reason
397   pred invalidateUnseenComponent(s: Stratum, c: Component)
398   {
399     s not in c.isInvalid_e -- it isn't valid here
400     no c.parts.s
401     no c.iDParts.s
402     no c.ports.s
403     no c.connectors.s
404     no c.attributes.s
405     no c.inferredLinks.s
406     c.myParts::nothing[s]
407     c.myAttributes::nothing[s]
408     c.myPorts::nothing[s]
409     c.myConnectors::nothing[s]
410   }
411
412   pred invalidateUnseenInterface(s: Stratum, i: Interface)
413   {
414     s not in i.isInvalid_e -- it isn't valid here
415     no i.operations.s
416     no i.implementation.s
417     i.myOperations::nothing[s]
418     i.myImplementation::nothing[s]
419     no i.superTypes.s
420   }
421
422   pred invalidateUnseenPorts()
423   {
424     all s: Stratum, c: Component
425     {
426       all p: Port |
427         p not in c.ports.s =>
428           no p.(provided + required).c.s
429     }
430   }
```

Deltas.als

Listing 3: deltas.als

```
module deltas[Stratum, Element, ID, Object]

open util/boolean as boolean
open util/relation as relation

sig Deltas
{
  -- the expanded objects for this stratum.  these 2 fields are the output of the merge!
  objects:              Object -> Stratum,

  -- old objects is what was what was there before any replacing was done
  oldObjects_e:        Stratum -> ID -> Object,
  originalOldObjects_e:    Stratum -> ID -> Object,

  -- working variables to track the expansion of objects, and allow it to happen cumulatively
  -- note: we need to keep track of what has been deleted and replaced to handle the cumulative effects
  -- e.g. delete in one stream, not in the other.
  -- NOTE: original is taking only resemblance_e into account, non-original is the full definition
  --       taking redefinition into account also
  objects_e: Stratum -> ID -> Object,
  deletedObjects_e:            Stratum -> ID,
  replacedObjects_e:           Stratum -> ID -> Object,
  originalObjects_e:           Stratum -> ID -> Object,
  originalDeletedObjects_e:    Stratum -> ID,
  originalReplacedObjects_e:   Stratum -> ID -> Object,

  -- newObjects is any objects added or replaced.  these fields allow new object creation to be controlled
  newObjects:             set Object,
  addedObjects:           set newObjects,
  replacedObjects:        set Object,
  -- newIDs are any new IDs added
  newIDs:                 set ID,
  -- the deltas that are to be applied.  these 3 fields are the input to the merge
  deleteObjects:         set ID,
  addObjects:            newIDs one -> one addedObjects,
  replaceObjects:        ID one -> lone replacedObjects
}
{
  -- cannot delete and replace
  no dom[replaceObjects] & deleteObjects
  replacedObjects = newObjects - addedObjects
}

-- indicate that any new part/ID can only be introduced by one component
fact Owned
{
  all o: Object |
    one newObjects.o

  -- do we also test for IDs?
  all n: ID |
    one newIDs.n
}


pred Deltas::oneObjectPerID(s: Stratum)
{
  let objects = this.objects_e[s] |
    function[objects, dom[objects]]
}

pred Deltas::nothing(s: Stratum)
{
  no this.objects.s
  no this.objects_e[s]
  no this.oldObjects_e[s]
  no this.deletedObjects_e[s]
  no this.replacedObjects_e[s]
  no this.originalObjects_e[s]
  no this.originalDeletedObjects_e[s]
  no this.originalReplacedObjects_e[s]
}

-- ensure that deletes and replaces makes sense from the perspective of the original stratum
pred Deltas::deltasIsWellFormed(owner: Stratum)
{
  -- no overlap between deleted and replaced IDs
  let
    deleteIDs = this.deleteObjects,
    replaceIDs = dom[this.replaceObjects]
  {
    -- no overlap between deleted and replaced
    no deleteIDs & replaceIDs
```

```
84        -- anything we delete or replace must be there already
85        deleteIDs + replaceIDs in dom[this.originalOldObjects_e[owner]]
86      }
87    }
88
89    -- ensures that this delta removes everything
90    pred Deltas::cleanSlate(owner: Stratum)
91    {
92      this.deleteObjects = dom[this.oldObjects_e[owner]]
93    }
94
95    -- ensures that we only have adds, no deletes or replaces
96    pred Deltas::onlyAdds(owner: Stratum)
97    {
98      no this.deleteObjects
99      no this.replaceObjects
100   }
101
102
103   -- the predicate to merge any underlying resembled entities and apply current changes
104   -- this is driven off the newly computed resemblance graph for each component in each stratum
105   pred Deltas::mergeAndApplyChangesForResemblance(
106     s: Stratum,
107     c: Element,
108     -- who should I resemble, taking redefinition into account
109     iResembleDeltas_e: set Deltas)
110   {
111     -- handle add, delete etc as if we are only taking resemblance into account
112     -- nothing will ever resemble itself
113     this.originalOldObjects_e[s] =
114       (iResembleDeltas_e.originalObjects_e[s]
115         - iResembleDeltas_e.originalDeletedObjects_e[s]->Object)
116           ++ iResembleDeltas_e.originalReplacedObjects_e[s]
117
118     this.originalDeletedObjects_e[s] =
119       iResembleDeltas_e.originalDeletedObjects_e[s]
120         - dom[iResembleDeltas_e.originalReplacedObjects_e[s]] + this.deleteObjects
121
122     this.originalReplacedObjects_e[s] =
123       (iResembleDeltas_e.originalReplacedObjects_e[s] - this.deleteObjects->Object)
124         ++ this.replaceObjects
125
126     this.originalObjects_e[s] =
127       (((iResembleDeltas_e.originalObjects_e[s] - this.originalDeletedObjects_e[s]->Object)
128           ++ this.originalReplacedObjects_e[s]) + this.addObjects)
129             ++ this.replaceObjects
130   }
131
132   pred Deltas::mergeAndApplyChangesForRedefinition(
133     s: Stratum,
134     c: Element,
135     topmost: set Element,
136     -- who should I resemble, taking redefinition into account
137     iResembleDeltas_e: set Deltas)
138   {
139     -- expand out into a easier form for expressing well-formedness rule, where IDs don't count
140     this.objects = {p: Object, s: Stratum |
141       some n: ID | s->n->p in this.objects_e}
142
143     -- handle add, delete etc as if we are only taking resemblance into account
144     topmost = c =>
145       this.oldObjects_e[s] = iResembleDeltas_e.originalOldObjects_e[s]
146     else
147       this.oldObjects_e[s] =
148         (iResembleDeltas_e.originalObjects_e[s]
149           - iResembleDeltas_e.originalDeletedObjects_e[s]->Object)
150             ++ iResembleDeltas_e.originalReplacedObjects_e[s]
151
152     this.deletedObjects_e[s] =
153       iResembleDeltas_e.originalDeletedObjects_e[s]
154         - dom[iResembleDeltas_e.originalReplacedObjects_e[s]]
155
156     this.replacedObjects_e[s] = iResembleDeltas_e.originalReplacedObjects_e[s]
157
158     this.objects_e[s] =
159       (iResembleDeltas_e.originalObjects_e[s] - this.deletedObjects_e[s]->Object)
160         ++ this.replacedObjects_e[s]
161   }
```

Wellformed_components.als

Listing 4: wellformed_components.als

```
module wellformed_components

open util/boolean as boolean
open util/relation as relation
open util/ternary as ternary
open structure


-- check that the component is well formed
pred componentIsWellFormed(s: Stratum, c: Component)
{
  -- ***RULE W4: the type of the part must not be in a cyclic relationship with itself through containment
  -- it also cannot be cyclical with respect to resemblance
  c not in c.^(resembles_e.s)
  -- the original (either the thing being redefined or the original)
  -- is not in the composition hierarchy taking resemblance into account
  -- NOTE: if c cannot be composed if it is a redefinition
  -- NOTE: a further constraint is that we cannot be composed of the thing we
  --       are redefining
  let
    resembling = resembles_e.s, partTypes = parts.s.partType,
    original = no c.redefines => c else c.redefines
  {
    original not in c.*(resembling + partTypes).partTypes
  }
  -- ***RULE W6: a component must have some ports
  some c.ports.s

  -- to be well formed, we must have one element per ID
  c.myPorts.oneObjectPerID[s]
  c.myAttributes.oneObjectPerID[s]

  // if this is composite, ensure the ports, parts and connectors are well formed
  isTrue[c.isComposite] =>
  {
    -- must always have some parts
    some c.parts.s

    -- to be well formed, we must have one element per ID
    c.myParts.oneObjectPerID[s]
    c.myConnectors.oneObjectPerID[s]

    -- don't require any parts -- e.g. junction components for altering connection interfaces
    partsAreWellFormed[s, c]
    connectorsAreWellFormed[s, c]
    portAndPortInstancesAreConnected[s, c]
  }
}

pred portAndPortInstancesAreConnected(s: Stratum, c: Component)
{
  all port: c.ports.s |
    portIsConnected[s, c, port]

  all cpart: c.parts.s |
    all port: cpart.partType.ports.s |
      portInstanceIsConnected[s, c, port, cpart]
}


pred partsAreWellFormed(s: Stratum, c: Component)
{
  all pPart: c.parts.s
  {
    -- ***RULE C8: it must be possible to reach this part from a series of connections from the owning
          component
    -- otherwise, this part will be completely internally connected -- an island
    s -> c in pPart.*(linkedToParts.c.s).linkedToOutside
    -- check the attributes
    let
      valueIDs = dom[pPart.attributeValues],
      aliasIDs = dom[pPart.attributeAliases],
      copyIDs = dom[pPart.attributeCopyValues],
      parentAttrs = c.myAttributes.objects_e[s],
      partAttrs = pPart.partType.myAttributes.objects_e[s],
      partAttrIDs = dom[partAttrs]
    {
      -- should have no overlap between the different types of possibilities
      disj[valueIDs, aliasIDs, copyIDs]
      -- all the IDs must exist in the list of attributes
      (valueIDs + aliasIDs + copyIDs) in partAttrIDs

      -- any new values must have the correct type
```

```
        all ID: valueIDs |
          pPart.attributeValues[ID].valueType = partAttrs[ID].attributeType

        -- any aliased or copied attributes must exist and have the correct type
        all ID: aliasIDs |
          partAttrs[ID].attributeType =
            parentAttrs[pPart.attributeAliases[ID]].attributeType
        all ID: copyIDs |
          partAttrs[ID].attributeType =
            parentAttrs[pPart.attributeCopyValues[ID]].attributeType

        -- anything left over must have a default value or else the parts attribute is unspecified
        all ID: partAttrIDs - (valueIDs + aliasIDs + copyIDs) |
          one partAttrs[ID].defaultValue
      }
    }
  }


  pred setupConnectors(s: Stratum, c: Component)
  {
    all end: c.connectors.s.ends |
    let other = end.otherEnd, aport = end.port.c.s, otherPort = other.port.c.s |
    {
      -- if just one end of the connector goes to the component, it must be mandatory
      -- if the part end is mandatory
      end in ComponentConnectorEnd =>
      {
        end.port.c.s = (end & ComponentConnectorEnd)::getPort[s, c]
      }
      else
      {
        -- this is a part connector end, make sure we connect to a single port instance
        let
          portAndPart = (end & PartConnectorEnd)::getPortInstance[s, c],
          resolvedPort = dom[portAndPart]
          {
            end.port.c.s = resolvedPort
            end.cpart.c.s = ran[portAndPart]
          }
      }
    }
  }

  pred connectorsAreWellFormed(s: Stratum, c: Component)
  {
    all end: c.connectors.s.ends |
    let other = end.otherEnd, aport = end.port.c.s, otherPort = other.port.c.s |
    {
      end.index in end.port.c.s.(mandatory + optional)
      one end.port.c.s

      -- if just one end of the connector goes to the component, it must be mandatory
      -- if the part end is mandatory
      end in ComponentConnectorEnd =>
      {
        -- note: other end must be a part connector end, as no component to component connectors are allowed
        -- if the outside is optional, the inside cannot be mandatory...
        end.index in aport.optional => other.index in otherPort.optional
      }
      else
      {
        one end.cpart.c.s
        end.index in aport.optional <=> other.index in otherPort.optional
      }
    }
  }



  ----------- support predicates

  pred portIsConnected(s: Stratum, c: Component, o: Port)
  {
    -- ports on the component must always be connected internally
    all idx: o.mandatory + o.optional |
    one end: c.connectors.s.ends & ComponentConnectorEnd |
    {
      end.port.c.s = o
      idx = end.index
    }
  }

  pred portInstanceIsConnected(s: Stratum, c: Component, o: Port, p: Part)
  {
    -- don't need to check any provided interfaces -- as these are always optional
    -- match up any mandatory required interfaces on the port with a single connector
    all idx: o.mandatory + o.optional |
    let ends =
    { end: c.connectors.s.ends & PartConnectorEnd |
      end.port.c.s = o and end.cpart.c.s = p and idx = end.index
```

```
174     }
175     {
176       (some o.required.c.s or isFalse[Model::providesIsOptional]) =>
177       {
178         idx in o.mandatory =>
179           -- must have a connection
180           one ends
181         else
182           -- can only have at most one connection
183           lone ends
184       }
185     }
186   }
```

Wellformed_interfaces.als

Listing 5: wellformed_interfaces.als

```
1  module wellformed_interfaces
2
3  open util/boolean as boolean
4  open util/relation as relation
5  open util/ternary as ternary
6  open structure
7
8  -- check that the interface is well formed
9  pred interfaceIsWellFormed(s: Stratum, i: Interface)
10 {
11   -- no circular resemblance for the interface, from the perspective of this stratum
12   i not in i.^(resembles_e.s)
13   -- must have some operations
14   some i.myOperations.objects.s
15   -- should have only 1 operation definition per id
16   i.myOperations.oneObjectPerID[s]
17   -- don't necessarily have to introduce a new implementation
18   lone i.myImplementation.newObjects
19   -- we should only have one implementation, so if we resemble something
20   -- we must replace the implementation
21   one i.implementation.s
22 }
```

Port_inference.als

Listing 6: port_inference.als

```
module port_inference

open util/boolean as boolean
open util/relation as relation
open structure
open inference_help

pred setupLeafLinks(s: Stratum, c: Component)
{
  -- make sure we have enough link ends
  ensureLinkEndsExist[s, c]

  let
    idToPorts = c.myPorts.objects_e[s],
    inferred =
      { p1, p2: ran[idToPorts] |
        idToPorts.p1 -> idToPorts.p2 in c.links }
  {
    -- copy over the links
    c.inferredLinks.s = inferred

    -- copy over the sets
    all cport: c.ports.s |
    let
      end = getComponentLinkEnd[idToPorts.cport],
      errors =
        some other: c.ports.s |
          cport -> other in c.inferredLinks.s and
            (cport.required.c.s != other.provided.c.s or
             cport.provided.c.s != other.required.c.s)
    {
      -- propagate the set value into the inferred value
      cport.required.c.s = cport.setRequired
      cport.provided.c.s = cport.setProvided

      -- we have no errors if all linked match up exactly
      s -> c in end.linkError <=> errors
    }
  }
}

pred setupCompositeLinks(s: Stratum, c: Component)
{
  ensureLinkEndsExist[s, c]

  let
    allPorts = c.ports.s,
    allParts = c.parts.s,
    idToPorts = c.myPorts.objects_e[s],
    idToParts = c.myParts.objects_e[s],
    -- flatten everything into a LinkEnd->LinkEnd structure so we can
    -- use transitive closure to navigate
    portToPort =   makePortToPort[s, c],
    partInternal = makePartInternal[s, c],
    partToPart =   makePartToPart[s, c],
    portToPart =   makePortToPart[s, c],
    partToPort =   ~portToPart,
    -- we connect by going from a port to a port,
    -- or from a port to part to possibly the other side of the part
    -- and then onto another part etc, until we get to a final part,
    -- or to a final port
    fromPortToPart = portToPart.*(partInternal.partToPart),
    fromPartToPort = ~fromPortToPart,
    fromPartToPart = partToPart.*(partInternal.partToPart),
    -- harsh allows us to bounce around looking for any possibly connected other elements.
    -- used to disallow inferredLinks via tainting
    harshFromPortToAny =
      portToPart.*(portToPart + partToPort + portToPort + partInternal + partToPart),
    fromPortToPort = portToPort + fromPortToPart.partInternal.partToPort
  {
    -- set up the inferred links, propagating the constraints to the next level
    propagateInferredCompositeLinks[
      s, c, harshFromPortToAny, fromPortToPort,
      partInternal, portToPort]

    -- get the provided and required interfaces of ports
    all cport: allPorts |
    let
      end = getComponentLinkEnd[idToPorts.cport],

      infReq = cport.required.c.s,
      reqEnds = end.fromPortToPart,
      requiresFromEnds =
```

```
84            { r: Interface  |
85              some ce: reqEnds |
86                r in ce.getPortInstanceRequired[s, c] },
87        matchingRequires = extractLowestCommonSubtypes[s, requiresFromEnds],
88
89          infProv = cport.provided.c.s,
90          provEnds = end.fromPortToPort - end +
91            { e: PartLinkEnd |
92              e in end.fromPortToPart and no e.partInternal
93            },
94          providesFromEnds =
95            { p: Interface |
96              {
97                (some e: provEnds & ComponentLinkEnd |
98                  p in e.getPort[s, c].required.c.s)
99                or
100               (some e: provEnds & PartLinkEnd |
101                 p in e.getPortInstanceProvided[s, c])
102             }
103           },
104       matchingProvides = extractHighestCommonSupertypes[s, providesFromEnds]
105       {
106         infReq = matchingRequires
107         infProv = matchingProvides
108
109         s -> c not in end.linkError <=>
110           (oneToOneProvidedMappingExists[s, c, infProv, reqEnds] and
111            oneToOneRequiredMappingExists[s, c, infReq, reqEnds])
112       }
113
114       -- enforce the constraints for each port instance
115       all cpart: allParts,
116         cport: cpart.partType.ports.s |
117       let
118         type = cpart.partType,
119         end = getPartLinkEnd[cpart.portMap[s].cport, idToParts.cpart]
120       {
121
122         {
123           let
124             infReq = end.getPortInstanceRequired[s, c],
125             infProv = end.getPortInstanceProvided[s, c],
126             terminalEnds = end.fromPartToPort +
127               { e: PartLinkEnd |
128                 e in end.fromPartToPart and no e.partInternal
129               },
130             provFromTerminalEnds =
131               { p: Interface |
132                 {
133                   (some e: terminalEnds & ComponentLinkEnd |
134                     p in e.getPort[s, c].provided.c.s)
135                   or
136                   (some e: terminalEnds & PartLinkEnd |
137                     p in e.getPortInstanceRequired[s, c])
138                 }
139               },
140             allEnds = end.fromPartToPort +
141               { e: PartLinkEnd |
142                 e in end.fromPartToPart
143               },
144             matchingTerminalProvides =
145               extractLowestCommonSubtypes[s, provFromTerminalEnds]
146           {
147             s -> c not in end.linkError <=>
148             {
149               no end.partInternal =>
150                 providesEnough[s, infProv, matchingTerminalProvides]
151               oneToOneRequiredMappingExists[s, c, infProv, allEnds]
152               oneToOneProvidedMappingExists[s, c, infReq, allEnds]
153             }
154           }
155         }
156       }
157     }
158   }
159
160   pred linksAreWellFormed(s: Stratum, c: Component)
161   {
162     let
163       allPorts = c.ports.s,
164       allParts = c.parts.s,
165       idToParts = c.myParts.objects_e[s],
166       portToPort =   makePortToPort[s, c]
167     {
168       -- enforce that no ports connect directly to each other
169       -- as this can lead to indeterministic interface assigment
170       isTrue[c.isComposite] =>
171         no portToPort
172
173       -- enforce the constraints for each port
174       all cport: allPorts |
```

```
175       let
176         infProv = cport.provided.c.s,
177         infReq = cport.required.c.s,
178         idToPorts = c.myPorts.objects_e[s],
179         end = getComponentLinkEnd[idToPorts.cport],
180         amHome = c.home = s,
181         setInterfaces = cport.(setProvided + setRequired)
182       {
183         -- check any set values only if we are "home"
184         (amHome and some setInterfaces) =>
185         {
186           infProv = cport.setProvided
187           infReq = cport.setRequired
188         }
189
190         -- must have some interfaces
191         some infProv + infReq
192         s -> c not in end.linkError
193       }
194
195       -- enforce the constraints for each port instance
196       all cpart: allParts,
197         cport: cpart.partType.ports.s |
198       let
199         end = getPartLinkEnd[cpart.portMap[s].cport, idToParts.cpart],
200         infReq = end.getPortInstanceRequired[s, c],
201         infProv = end.getPortInstanceProvided[s, c]
202       {
203         -- must have some interfaces
204         some infProv + infReq
205         s -> c not in end.linkError
206       }
207     }
208   }
209
210
211   -- set up the inferred links for this component for a leaf, just use the links
212   -- for a composite, trace through from port to port, but only infer a link if there
213   -- is no terminal part involve anywhere
214   pred propagateInferredCompositeLinks(
215     s: Stratum,
216     c: Component,
217     harshFromPortToAny:  ComponentLinkEnd -> LinkEnd,
218     fromPortToPort:      ComponentLinkEnd -> ComponentLinkEnd,
219     partInternal:        PartLinkEnd -> PartLinkEnd,
220     portToPort:          ComponentLinkEnd -> ComponentLinkEnd)
221   {
222     let
223       idToPorts = c.myPorts.objects_e[s],
224       allPorts = c.ports.s,
225       terminateInternallyIDs =
226       { id: dom[idToPorts] & PortID |
227         some end: PartLinkEnd |
228         let instance = end.getPortInstance[s, c],
229           cport = dom[instance],
230           cpart = ran[instance]
231         {
232           getComponentLinkEnd[id] in harshFromPortToAny.end
233           no end.partInternal
234           -- only provided terminals break linking
235           some cport.provided.(cpart.partType).s
236         }
237       }
238     {
239       -- find all port->port combinations that go through a leaf part and link up
240       -- but which don't have a termination on a provided port instance interface
241       let inferred =
242       { p1, p2: Port |
243         some end: dom[fromPortToPort] | let other = end.fromPortToPort
244         {
245           disj[end, other]
246           p1 = idToPorts[end.linkPortID]
247           p2 = idToPorts[other.linkPortID]
248           -- if we can reach a port which links internally,
249           -- do not create an alias
250           no (end + other).*portToPort.linkPortID & terminateInternallyIDs
251         }
252       } |
253       c.inferredLinks.s = inferred
254     }
255   }
```

Inference_help.als

Listing 7: inference_help.als

```
module inference_help

open structure
open util/boolean as boolean


pred providesEnough(s: Stratum, provided: set Interface, required: set Interface)
{
  all prov: provided |
    one req: required |
      req in prov.*(superTypes.s)

  -- ensure that it works the other way around
  all req: required |
    one prov: provided |
      req in prov.*(superTypes.s)
}

pred oneToOneProvidedMappingExists[s: Stratum, c: Component, provided: set Interface, ends: LinkEnd]
{
  all end: ends & ComponentLinkEnd |
    oneToOneMappingExists[s, provided, end.getPort[s, c].required.c.s]

  all end: ends & PartLinkEnd |
    oneToOneMappingExists[s, provided, end.getPortInstanceProvided[s, c]]
}

pred oneToOneRequiredMappingExists[s: Stratum, c: Component, required: set Interface, ends: LinkEnd]
{
  all end: ends & ComponentLinkEnd |
    oneToOneMappingExists[s, required, end.getPort[s, c].provided.c.s]

  all end: ends & PartLinkEnd |
    oneToOneMappingExists[s, required, end.getPortInstanceRequired[s, c]]
}

pred oneToOneMappingExists[s: Stratum, a: set Interface, b: set Interface]
{
  all aa: a |
    one bb: b |
      bb in expand[s, aa]

  -- ensure that it works the other way around
  all bb: b |
    one aa: a |
      bb in expand[s, aa]
}

fun extractHighestCommonSupertypes(s: Stratum, require: Interface): set Interface
{
  let
    -- map is an interface in require (i) with all matching interfaces in require (e)
    map =
    { i: require, e: require |
      some expand[s, i] & expand[s, e] },
    highestCommonSupertypes =
    {
      super: Interface |
        some i: require |
          super = highestCommonSupertype[s, map[i]]
    }
  {
    highestCommonSupertypes
  }
}

fun highestCommonSupertype(s: Stratum, required: set Interface): lone Interface
{
  { i: Interface |
    {
      required in *(superTypes.s).i
      no sub: superTypes.s.i |
        required in *(superTypes.s).sub
    }
  }
}

fun extractLowestCommonSubtypes(s: Stratum, require: Interface): set Interface
{
  let
    -- map is an interface in require (i) with all matching interfaces in require (e)
    map =
    { i: require, e: require |
```

```
84          some expand[s, i] & expand[s, e] },
85        lowestCommonSubtypes =
86        {
87          sub: Interface |
88            some i: require |
89              sub = lowestCommonSubtype[s, map[i]]
90        }
91      {
92        lowestCommonSubtypes
93      }
94    }
95
96    fun lowestCommonSubtype(s: Stratum, required: set Interface): lone Interface
97    {
98      { i: Interface |
99        {
100         required in i.*(superTypes.s)
101         no super: i.superTypes.s |
102           required in super.*(superTypes.s)
103       }
104     }
105   }
106
107   fun expand(s: Stratum, i: Interface): set Interface
108   {
109     -- expand forms the full expanded supertype and subtype hierarchy
110     i.*(superTypes.s) + ~(superTypes.s).i
111   }
112
113
114   -- a generator axiom to ensure that we have a unique link end per port, or port instance
115   pred ensureLinkEndsExist(s: Stratum, c: Component)
116   {
117     let
118       idToPorts = c.myPorts.objects_e[s],
119       idToParts = c.myParts.objects_e[s]
120     {
121       -- set up the linkends
122       -- ensure all ports have a link end
123       all portID: dom[idToPorts] |
124         one l: ComponentLinkEnd |
125           l.linkPortID = portID
126
127       -- ensure all part/ports have a link end
128       all ppart: c.parts.s |
129         let partID = idToParts.ppart |
130           all portID: ppart.portMap[s].Port |
131             one l: PartLinkEnd |
132               l.linkPortID = portID and l.linkPartID = partID
133     }
134   }
135
136   fun makePortToPort(s: Stratum, c: Component): ComponentLinkEnd -> ComponentLinkEnd
137   {
138     isTrue[c.isComposite] =>
139     { p1, p2: ComponentLinkEnd |
140       some end: c.connectors.s.ends | let other = end.otherEnd
141       {
142         disj[end, other]
143         end + other in ComponentConnectorEnd
144         end.portID = p1.linkPortID
145         other.portID = p2.linkPortID
146       }
147     }
148     else
149     { p1, p2: ComponentLinkEnd |
150       some end: dom[c.links], other: c.links[end]
151       {
152         -- these are disjoint because of a clause in structure.als
153         end = p1.linkPortID
154         other = p2.linkPortID
155       }
156     }
157   }
158
159   fun makePartInternal(s: Stratum, c: Component): PartLinkEnd -> PartLinkEnd
160   {
161     let
162       idToParts = c.myParts.objects_e[s] |
163     { p1, p2: PartLinkEnd |
164       some partID: dom[idToParts] |
165       let
166         realPart = idToParts[partID],
167         realType = realPart.partType,
168         inferredOneWay = realType.inferredLinks.s,
169         inferred = inferredOneWay + ~inferredOneWay,
170         idToPorts = realPart.portMap[s],
171         realPort = idToPorts[p1.linkPortID]
172       {
173         disj[p1, p2]
174         p1.linkPartID = partID
```

```
175         p2.linkPartID = partID
176         realPort in dom[inferred]
177         idToPorts[p2.linkPortID] = inferred[realPort]
178       }
179     }
180   }
181
182   fun makePartToPart(s: Stratum, c: Component): PartLinkEnd -> PartLinkEnd
183   {
184     { p1, p2: PartLinkEnd |
185       some end: c.connectors.s.ends | let other = end.otherEnd
186       {
187         disj[end, other]
188         end.portID = p1.linkPortID
189         end.partID = p1.linkPartID
190         other.portID = p2.linkPortID
191         other.partID = p2.linkPartID
192       }
193     }
194   }
195
196
197   fun makePortToPart(s: Stratum, c: Component): ComponentLinkEnd -> PartLinkEnd
198   {
199     { p1: ComponentLinkEnd, p2: PartLinkEnd |
200       some end: c.connectors.s.ends | let other = end.otherEnd
201       {
202         end in ComponentConnectorEnd
203         end.portID = p1.linkPortID
204         other.portID = p2.linkPortID
205         other.partID = p2.linkPartID
206       }
207     }
208   }
```