1 Layers

Any architectural description needs to include a declaration of all the modules inside the architecture. The set of all modules inside the architecture is called *components*. We'll say $components = \{IO, File, Reader, Writer, Database, Socket, Server\}$, using the example from the other document.

We want to further specify that modules belong to a particular layer. The easiest way to do this is to represent a layer as a set of modules. The example in the other document might look like this:

$l_1 = \{IO, File\}$

$l_2 = \{Reader, Writer\}$

$l_3 = \{Database\}$

$l_4 = \{Socket, Sever\}$

This declares three sets corresponding to the layers. To limit how layers access each other we refer to the notion of authority as described in the ECOOP 16 paper. $auth(m)$ returns the set of objects for which m has a capability. $auth(l_3)$ should only contain objects in l_2 and same for l_2 and l_1. Invariants for those are as follows:

$\forall m \in l_3 \cdot auth(l_3) \subseteq l_2$

$\forall m \in l_2 \cdot auth(l_2) \subseteq l_1$

We might want a strict enforcement of layers: at the moment if a module is inside the architecture but not inside one of the sets then it has unrestricted access to everything. An invariant to ensure that every module in the architecture belongs to a layer is the following:

$\forall m \in components \cdot m \in l_1 \cup l_2 \cup l_3 \cup l_4$

At the moment modules external to the architecture have unrestricted access over all the layers. To limit use of the system by the outside world to the topmost layer (layer 4) we can write:

$\forall m \notin components \cdot \forall n \in auth(m) \cdot n \in components \rightarrow n \in l_3$

This says that for any external m in *components*, if n is a component in the authority of m, then n is in layer 3.

2 Pipeline

Because pipeline architecture is all about sequencing operations it might be useful to use temporal predicates. First we'll describe a simplified Wyvern pipeline as being $components = \{lexer, parser, checker, interpreter, main\}$.

We'll think about the flow of time in a program as a computation path. So each time quantum refers to a single program state. The time in the program updates when the next instruction is executed. With that in mind, here are some modalities from linear-temporal logic:

- *Next.* This is a unary modality written as $\boldsymbol{N}x$. It says that x holds for the next time step.
- *Until.* This is a binary modality written as $x\boldsymbol{U}y$. It says that at some timestep in the future y becomes true and for every time step until then, x is true.
- *Release.* Written as $x\boldsymbol{R}y$. This says that y is true until x becomes true. If x never becomes true, then y is true forever.
- *Future/Finally.* Written as $\boldsymbol{F}x$. This says that x is true at some point in the future.
- *Globally.* Written as $\boldsymbol{G}x$. This says that x is always true.

We want to say that typechecking must happen after parsing and that the output of parsing must not be modified until typechecking occurs. This could be written as:

$(\mathit{output} = \mathit{parser.parse}(\_\_)) \rightarrow (\boldsymbol{F}\ \mathit{checker.check}(\mathit{output}))$

This says that if *output* is the result of calling parser.parse on an argument then at some point *output* is passed as an argument to an invocation of *checker.check*. Note that some pattern-matching is going on: the argument to *parser.parse* is being wildcarded because we don't need to refer to it. The output of *parser.parse* is being captured in the variable *output*.

Furthermore, note that $\mathit{output} = \mathit{parser.parse}(\_\_)$ and $\boldsymbol{F}\ \mathit{checker.check}(\mathit{output})$ are not formulae in temporal logic. They are really a shorthand to say that if there exists a timestep at which $\mathit{output} = \mathit{parser.parse}(\_\_)$ happens then at a later timestep $\mathit{checker.check}(\mathit{output})$ also happens (the moment at which it happens being determined by the $\boldsymbol{F}$ modality).

Nothing prevents the value of the captured output from being modified in between the call to *parse* and the call to check. First let's say $modifies(l, e)$ if and only if expression e has side effects on l.

To say that the output of the parser is not modified until it gets shoved into the typechecker we add an extra clause in the predicate, using our new definition of $modifies$.

$$\begin{aligned}&(output = parser.parse(\_\_)) \rightarrow (\boldsymbol{F}\ checker.check(output))\\ \wedge\ &(\neg modifies(l, e)\ \boldsymbol{U}\ checker.check(output))\end{aligned}$$

Where e is the execution path from *parser.parse* to *checker.check*. Note there can be more than one and we want it to be true for all of them so to make that true we'd need e to be a bound variable and possible add in some temporal quantifiers or something of the sort. Another approach would be to give annotations restricting the execution path (e.g. to say that there is a certain method *meth* which is the glue between *parser.parse* and *checker.check* and that the only execution path happens in *meth*).

The idea can be used to describe the other operation sequences as well.

3 Service Mediators

In the example of *ResourceLocater* being a mediator for $WebBrowser$ an issue is that there is no guarantee on who *ResourceLocater* is actually talking to. *ResourceLocater* could be talking to unknown external modules which do malicious things.

Since *ResourceLocater* only abstracts an information flow from $WebBrowser$ to $WebServer$ a useful constraint might be to say that *ResourceLocater* does not contain a capability for anything which $WebBrowser$ does not. From the perspective of $WebBrowser$ this prevents *ResourceLocater* from looking up anything we don't already know about. An invariant for this would be:

$$auth(ResourceLocater) \subseteq auth(WebBrowser)$$

Issue: $WebBrowser$ may be delegating to *ResourceLocater* because $WebBrowser$ doesn't have permission to talk to $WebServer$ but *ResourceLocater* does. Can we write an invariant in this situation?

4 Conversation Mediators

One Mediator

Let A, B, and C be participants in a conversation with two flows of information between them: $I_{A\rightarrow B}$ and $I_{A\rightarrow C}$. We may wish to specify their communication logic in one central area, a mediator *med*.

One way to do this is to explicitly specify the methods in which the information flow happens. To illustrate, it may be that *med* takes input from A and passes it to B in the method *med.msgAtoB*. Then in *med.msgAtoB* we shouldn't see any references to C. An invariant to this effect is:

$C \notin auth(med, med.msgAtoB)$

Similarly if *med* had a method *msgAtoC* implementing the information flow $I_{A \to C}$ then B should not be referenced in that method. To wit:

$B \notin auth(med, med.msgAtoC)$

For a more complex example imagine we have the same two information flows but in a bigger conversation with more participants. To make it more general we might provide the following invariants:

- $conversation = \{A, B, C, D, E, F\}$
- $\forall m \in conversation \cdot m \in auth(med, med.msgAtoB) \to m = B$
- $\forall m \in conversation \cdot m \in auth(med, med.msgAtoC) \to m = C$

Many Mediators

Another solution might be to have a mediator per information flow. So you would have a single object for $I_{A \to B}$ and another for $I_{A \to C}$. Invariants to describe this situation:

- $conversation = \{A, B, C\}$
- $mediators = \{medAtoB, medAtoC\}$
- $\forall med \in mediators \cdot \#(mediators \cap auth(med)) = 2$
- $mediatedBy(x, y) = \{\ med \in mediators \cdot conversation \cap auth(med) = \{x, y\}\ \}$
- $\forall x, y \in conversation \cdot \#(mediatedBy(x, y)) \in \{0, 1\}$

The first two invariants describe sets of participants in the conversation and their mediators. The third invariant says that every mediator only mediates between two of the participants (so for example there is no mediator talking between three of the participants). The fourth is a function on pairs of modules: it returns the set of all mediators which mediate between the two modules. The fifth invariant says that for every pair of participants there is at most one mediator.

Note that this architectural definition would exclude the previous situation of having one mediator because a single mediator would have a capability to all of the participants. However we could do a similar quantification over the methods of the mediator.