Rethinking Software Connectors

or

How I Learned To Start Worrying And Love Terminology

Stephen Kell

Stephen.Kell@cl.cam.ac.uk

Computer Laboratory

University of Cambridge

Connectors are cool

Whence “connectors”?

- computation is old hat
- communication: the not-so-old frontier
  - high-level design / architecture
  - re-use
  - distribution / parallelisation

Connectors as a concept

Three oft-cited works conceptualising connectors:

- Shaw 1993 (intuitions and motivations)
- Allen 1997 (formalisation)
- Mehta 2000 (taxonomy)

Outstanding question: what *are* connectors?

Not yet a mature concept.

Connectors are confusing

- intuitive extensional definitions only
  - “mediate interactions” (Shaw)
  - “manifest themselves as [*examples*]” (Mehta, Shaw)
  - “describe the interactions” (Allen)
- too many kinds of thing
  - servers? data encodings? protocol specifications?
- distinctions: type, instance, “image”, state, invocation
- disagreement on relations
  - connectors $\cap$ components $= \emptyset$?
  - coordinators $\supseteq$ connectors?
  - adaptors $\subseteq$ connectors?

Connectors characterised

Let's characterise connectors more simply:

- communication (mechanism, information)
- coupling (agreement, meaning)

To be a connector, it takes mechanism.

To *use* a connector meaningfully, it takes agreement.

This *could* be straight out of a communication theory text. . .

- . . . but I haven't yet found which. If you know, tell me!

Will return to this later....

Corollaries

What *doesn't* define connectors? Popular red herrings:

- protocol
- polymorphism / typelessness
- lack of state
- dynamic creation

I contend that the following are all connectors:

- an operating system [kernel]
- the Internet (or any network)
- the air in this room
- the media

What defines a *component*? Good question.

The connector-component “continuum”

Are these components or connectors?

- filter in pipe-and-filter style
- “façade” layers in a layered web app
- servers / multiplexers
- protocol adapters
- marshallers
- any stateful shared component (covert channels...)

Assertion: “connector or component” isn’t intrinsic...

- . . . it depends on a chosen level of abstraction.
- It’s valid to consider connectors $\supseteq$ components. . .
- . . . despite the dogma which says otherwise.

It's only terminology, but I like it

Gauging reaction: please pick one (and save for questions):

- trivially, you are correct
- no, you are incredibly mistaken
- I don't care either way!

Aside: why should we care? My tentative answer:

- credibility
- communication is one of the hardest parts of research
  - just like software

Coupling

Coupling is a familiar term. What does it mean?

- draw a surface around an arbitrary part of a system;
- evaluate
  - *how much* the inside needs to *know* about outside...
  - ... and vice-versa...
  - in order for the system to “work” (in whatever sense)

Information theory refines “how much” and “know”.

Shannon calls these “knowledges” the “code”.

I call them “agreements”. . .

- . . . because *dis*agreement causes the problems

Coupling and connectors

Coupling occurs across connectors (and nowhere else).

Dealing with (static) coupling:

- have less of it (minimisation)
  - information hiding
  - late binding, negotiation, discovery
  - only goes so far (no coupling $\leftrightarrow$ no communication)
- make it less of a problem (mitigation)
  - localisation of definitions
  - standardisation
  - adaptation

Theorem: black-box composition of independently developed code *requires* adaptation.

Coupling by “layered” agreements

Choice of mechanism is one agreement. We refine it by

- shared message (content) coding rules
- shared timing and sequencing (context) coding rules

Disagreements especially problematic:

- often not stated explicitly
  - can’t check statically (or adapt)
- often not stated in a single place (not localised)
  - poor mitigation of coupling; drift
- ambiguity: mismatches subtle and/or undetected
  - can’t check dynamically (or use negotiation)

Solutions: self-description, {localised, static} specification

Coupling and “coupling”

Recap: whence connectors? *Re-use* and also *concurrency*.

“Coupling” can mean more than one thing: consider Linda.

- “loose coupling” oft-cited advantage of send-receive
- usually means simple, unconstrained ($\rightarrow$ parallelism!)
- but these mean more layered agreements. . .
- . . . so can *worsen* coupling!

In all cases, simpler mechanisms mean that

- more layered agreements are required. . .
- . . . in order to convey a given meaning
- so more opportunity for disagreement (mismatch)

Complexity trade-offs

We can trade off complexity between

- communication abstractions. . .
- . . . and the computation (and state)
- (of the components which use them)

Canonical example: smart network vs dumb network

- smarter $\rightarrow$ simpler clients, fewer layered agreements
- dumber $\rightarrow$ less constraint, more flexibility

Optima are inherently application-dependent.

- dumber doesn't always gain *useful* flexibility
- smarter doesn't always enable simplicity (end-to-end)

Configuration

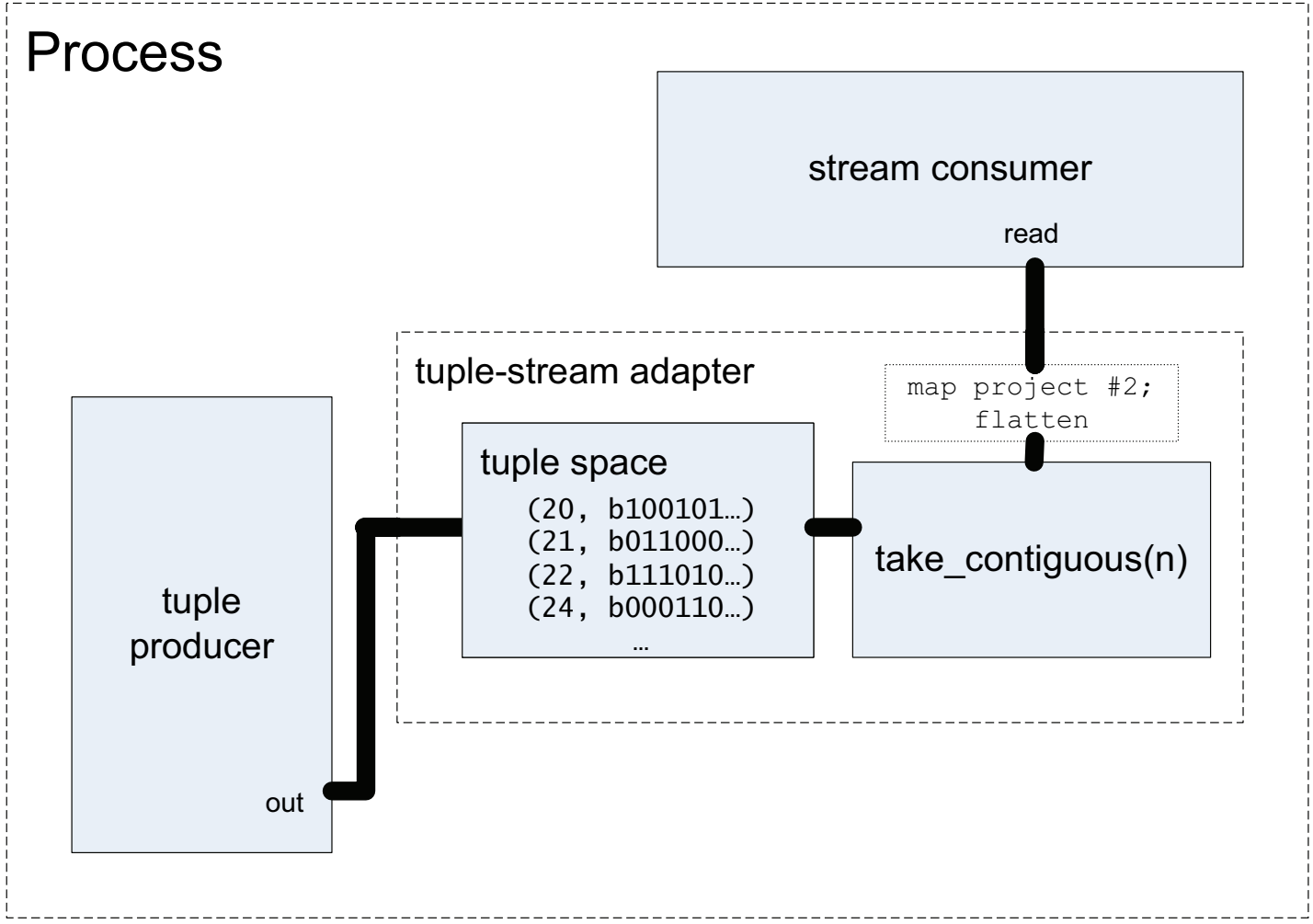
Any complex system is a *configuration* of simpler ones (recursively).

A configuration has defining properties:

- there's more than one piece to it
- pieces are joined together in some arrangement
- pieces may be atomic or recursive
- pieces may be re-used or novel

Configuration concretion

Here's a picture of a fictional configuration.



Configuration explaining connectors

Some re-statements of the observations which begat “connectors”:

- programming languages aren’t great for expressing architecture
- programming languages aren’t great for expressing configuration
- communication abstractions are comparatively neglected
- tool support across configurations is weaker than within them

Configuration, connectors, components

Any programming language is also a configuration language

- but some configuration languages are not programming languages
- e.g. symbol bindings in a linkage language
- configuration need not be Turing-powerful

Configuration and programming unified by *primitive connectors*.

- i.e. must unify
  - mechanisms denoted in programming language
  - mechanisms denoted in configuration language
- they don't have to be *the same*. . . why shouldn't they?

Configuration coming soon. . .

Some ideas:

- operating system + adaptation
- pluggable checking of configurations
- adaptation as the default
- semi-automatic refactoring (to separate communication)
- extensibility by interposition

Conclusions

- Lack of clarity or of explicit definitions can make much good work appear conceptually confused.
- Mechanism and agreement may be a useful way to think of connectors and coupling.
- There are many trade-offs surrounding coupling
- “Explicit configuration” might be better than “first-class connectors”.
- Maybe we’re already doing “explicit configuration” research. . .
  - . . . but just not expressing it right.

Questions are welcome.