Process Calculi for web services

Lucian Wischik, University of Bologna
http://ww.wischik.com/lu/research/choreography.html

W3C choreography group, March 2003, CA

These slides are available on this web site. And the web site also has specific links for the work that I mention.

My name's Lucian Wischik. I'm a researcher in process calculi, specifically the pi calculus.

I'm talking to you here now because maybe web services can benefit from what's been done with process calculi. I hope today to give a quick primer on them, something that's easier to digest than the textbooks and research papers on the subject.

"process calculus" means two things:

- a simple ***language*** for writing/describing/specifying interactive message-passing programs.
  - *more concise than automata*
  - *better than automata for 'reconfigurable' systems*
  - it's trivial, easy, but in this talk I'll use diagrams instead.
- a notion called `***bisimulation***' to say when two programs have the same interactive behaviour.
  - *difficult! subtle! this is the topic of the talk*

What does it mean, "process calculus"? Two things. First, a language for writing interactive programs - ones that work by message-passing. Really, you can work with automata diagrams like in Biztalk, or write in a process language, and I'll use diagrams in this talk. The language is so simple that you won't benefit much from a primer by me. Just read the first couple of pages of Milner's book.

The second thing in a process calculus is a notion called "bisimulation". It's for saying whether two programs have the same interactive behaviour. The thing is, in the olden days, when we wrote functions, it was easy to say whether two functions are equivalent – if two functions give identical outputs for identical inputs, then they're equivalent. For interactive programs it's harder. That's what I want to talk about.

what have process calculi ever done for us?

Bisimulation:

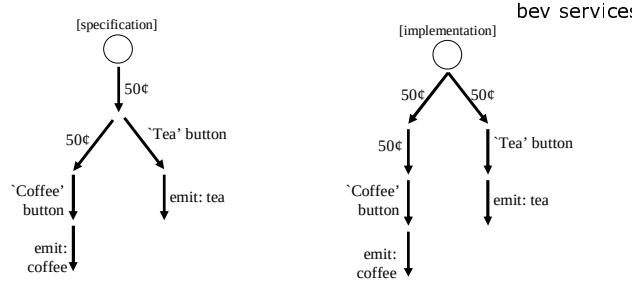
- we need to ***specify*** the behaviour of interacting services
- most researchers settled upon **`*bisimulation*'** for this job
- but it can be a bit subtle.

- Plan: to explain the idea behind it,
  outline state of the art, avoid the maths.

I reckon that this is relevant to web services, because you'll need to specify the behaviour of a web service, and to be able to judge whether an implementation is equivalent to the specification. And that's what bisimulation is all about, and there's been lots of work in it, and maybe web services can benefit from the work.

This "lots of work" culminated in a book last year by Davide Sangiorgi. which is an inch and a half thick of serious heavy-duty maths. I hope to explain the ideas behind it, without any of the maths. When Davide explained his book to his wife, she just said "that's all trivial", so maybe you'll feel the same!

bev services



- These two bev-services accept the same message-sequences {50c,'T',tea} and {50c,50c,'C',coffee}.
  But they are not equivalent services! *message-sequences are inadequate*.
- ***Bisimulation*** is needed: we must pay attention to the states, to the possibilities still available at each state.

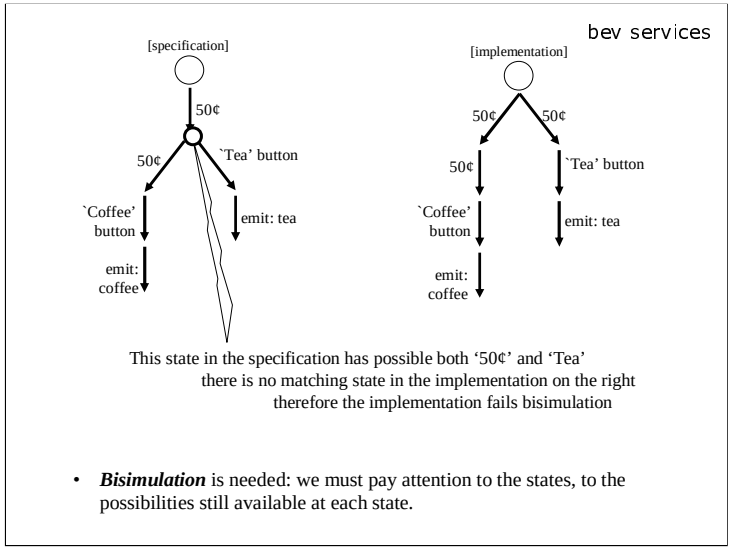
I'm still learning about WEB services, so instead of them, I thought I'd talk about something I'm more familiar with: BEV services.

This is a classic example from the 80s. (from a calculus called CCS). From Milner. We see two bev services here. The one on the left is the specification, on the right is the implementation. We wish to establish whether they are the same.

There's something wrong with the implementation. Can anyone see what it is? [answer: the machine itself decides whether you get tea or coffee, not you.] We hope for an automated test that will tell us: this implementation fails.

What kind of automated test? Well, the diagrams are finite-state automata, and the classic test of automata-equivalence is whether they accept the same sequence of messages. In this case they do: {50c,T,tea} or {50c,50c,C,coffee}. So, message-sequence-testing isn't good enough.

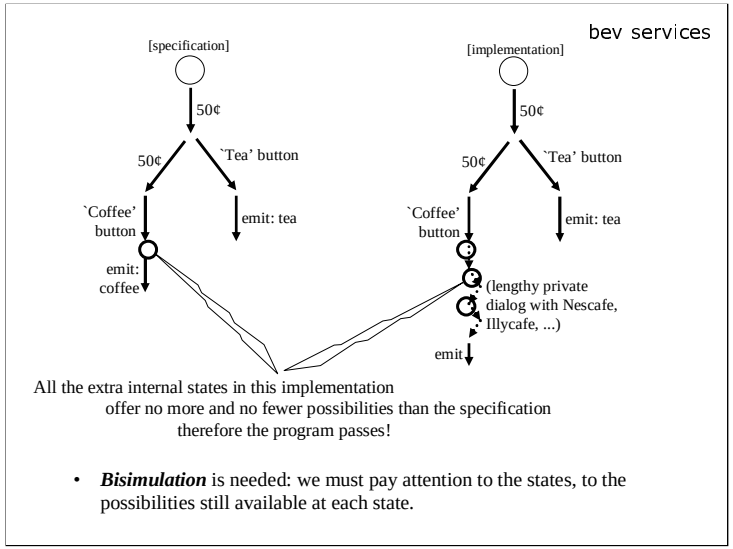
Instead...



Instead we need to pay attention to the states, to the possibilities still available at each state. This is the essence of bisimulation.

Let's see why this example fails the bisimulation test.

In the specification on the left, there is this state which can accept either a press on the 'tea' button, or a further 50cents. But there is no matching state on the right. Therefore, it fails.

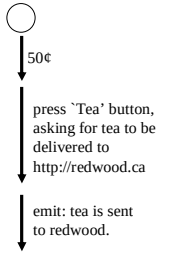


Here's an example which succeeds the bisimulation test.

This implementation on the right doesn't emit the coffee straight away. Instead, it engages in a lengthy private dialog with various Coffee-bean services. Only at the end does it emit the coffee.

Well, all of these internal states of the private dialog, they all have the same external possibilities open to them as the specification. Therefore this succeeds.

- ***Reconfigurability*** makes the topic harder:
  if messages can include the names of other channels, then...



[specification]
"...the state that receives the message **tea_please(redwood)** must be followed by a state that sends the message **tea** to redwood."

As well as observing messages,
we can observe a message's arguments,
and we must parameterise the rest of the specification upon them.
(obvious, really!)

"output capability"
MS BizTalk

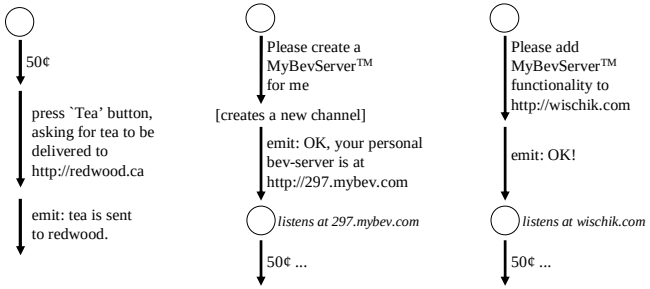
BUT. The examples on the previous slide were easy. They're for a simple service without reconfiguration.

What exactly is "reconfiguration"? It means that messages can include channel-names. So I could tell you an address, and you send back your answer to that address. Here, I ask for tea to be delivered to Redwood.

If you've used Biztalk, this reconfiguration is called "dynamic ports". (actually, if you've used dynamic ports in Biztalk, you'll understand what I said earlier: automata diagrams aren't good at expressing reconfigurability, a proper language is better).

Anyway, obviously, the specification needs to be parameterised on the contents of messages.

- ***Reconfigurability*** makes the topic harder:
  if messages can include the names of other channels, then...



"output capability"
MS BizTalk

"new-channel creation"
Join Calculus,
Localised Pi Calculus.

"input capability"
Pi calculus. MS Highwire.
Implement it with fusions.

There are other forms of reconfigurability as well.

Like in the Join calculus and Localised Pi Calculus you can create a new channel at runtime, which then receives messages. This example is a kind of meta-bev-server, which can create new bev servers.

What sort of thing is this good for describing? It's like when a server gets a request at the start of the dialog, then it creates some separate channels to handle the rest of this particular dialog instance.

(You can do this in other ways like with cookies, or whatever, like people do at the moment. But there's a good reason to put the capability into the language instead. It means that it becomes something you can specify and verify.)

Another kind of reconfigurability is found in the full pi calculus. This is my research area. It's called "input capability".

It means that I can tell the meta-bev-server to add functionality to a pre-existing channel, like wischik.com here. (the difference, in the previous example, it could only add functionality to its own newly-defined channel).

Well, we don't know if this full pi-calculus capability will be practically useful or not. It's implemented with something called fusions, and is present in MS Highwire. Once it's been available, and once people have written real programs with it, we'll see whether it's useful.

conclusions

- ***Reconfigurability*** makes the topic harder:
  if a message includes the name of other channels, then...
  - You have to take this into account for bisimulation
    i.e. parameterise the specification on the data received in messages
  - But now, it's easy to end up with an ***unsafe bisimulation***
    i.e. clients can spot differences, even though the implementation 'passed'
  - The ideal: "a program passes the specification if no client can ever distinguish the two." (called ***congruence***). But not computable.
  - So researchers find ***safe approximations*** for their bisimulations, stricter than necessary, but ***easy to model-check.***
  - We dream of ***behavioural type systems*** – where the type-checker checks that interaction obeys a given protocol, as well as data obeying its type.

The upshot is that, depending on what capabilities we think web services have, the specification language has to reflect that.

And this makes it more fiddly to get the right form of bisimulation for the language.

I also think that transactions will also make it more fiddly, but they've hardly been addressed yet by process calculus researchers.

This is where you begin to get fed up with bisimulation, to think it's too complicated.

So researchers like to cut to the chase and come up with a simple, clear definition. This philosophical ideal is this: you say: two systems, the specification and the implementation, the *two systems have the same interactive behaviour if no client can ever distinguish the two.*

But this is just an ideal. It's not practical, you can't test it, you can't prove it.

So researchers work to find safe, conservative approximations that can be tested.

This is a bit heavy-going. Really, in a year or two's time, we hope to see it boil down into something quite simple: a compiler where it's type system checks behaviour – not just data-types matching, but it also checks that for a procedure which uses some channels, which sends or receives data over the channels – it checks that the procedure is obeying a given protocol with these messages, with these channels. That'd be a lovely way to write programs that use web-services.