Making system composition flexible, automatic, safe, practical…

(but not always at the same time)

Stephen Kell

Stephen.Kell@cl.cam.ac.uk

Introduction and running order

An informal and incomplete survey of work by DJG and co.

- corrections, heckles, interruptions all welcome

Priority was assigned based on responsiveness factor. So hopefully

- 15 minutes on my work
- 10 minutes on David's
- 7.5 minutes on Atif's
- 5 minutes on Jin's
- 2 minutes each on Yiannis, Henry, Tope, Aisha, Behzad

Unifying themes

Some partially-unifying themes:

- communication within “open”, modular and/or distributed systems
- theory meets practice
- weak distinction between hardware and software

My work: problem statement

Writing software from scratch is expensive. Avoid it: re-use.

- re-use is hard too, because of various flavours of mismatch:
  - interface (operations, signatures, protocol, encoding)
  - architectural (structural assumptions)
  - “packaging” (choices of binary concretion, communication abstractions)
- kinds of re-use: white-, grey-, black-box (decreasing invasiveness)
- (Theorem:) re-use inevitably involves *adaptation*

Motivating tasks

Here are some tasks which are difficult using existing tools:

- porting an application to use comparable but different supporting software
  - e.g. GTK+ app to Qt; IE plugin to Firefox;
- composing heterogeneous components within the same application
  - linking a web form with a C program
  - linking a Perl script with a spreadsheet function

What I propose:

- practical *systems-level black-box* adaptation. . .
- . . . making these tasks measurably less complex

Approach

What does *systems-level* mean? It means we do adaptation

- close to run-time (typically after deployment)
- at the linkage level (i.e. typically on binary representations)

Why are these good things?

- important class of mismatches only show up close to deployment e.g. library versioning problems
- others are easier to deal with later, when more context is known
  - consider adapting a JVM

Goal: separate *functionality* from all details of *integration*.

Inspirations and observations

Linking languages, esp. Knit (Reid et al, OSDI '00):

- simple, flexbile, declarative description of linkage graph
- adaptation by symbol renaming (only)
- a convenient place to add further adaptation features...

Scripting languages ("glue"):

- brevity ($\rightarrow$ easy invasive changes)
- expressivity ($\leftarrow$ many convenience features)
- support many/most OS communication mechanisms
- explicit adaptation features (regex-based rewriting)
- complex and error-prone!

How can we get the best of both worlds?

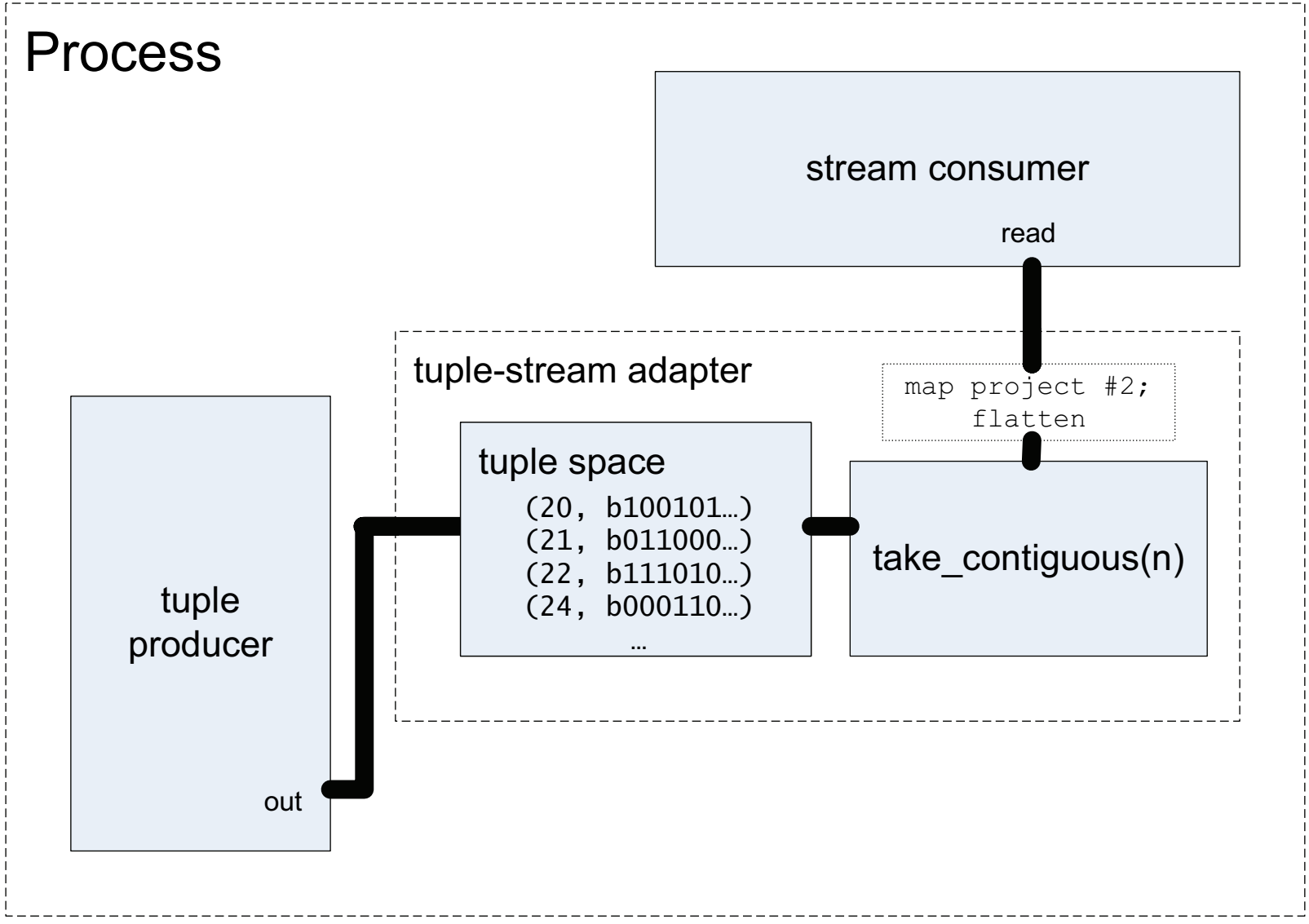
Early decisions and concepts

"Configuration languages": languages expressing structure

- examples: linking, module interconnection, architecture description, . . .
- form a spectrum parallel to programming languages
- benefit: make structure explicit
- benefit: separate static from dynamic
- may or may not have explicit hierarchy
  - hierarchy helps exploit recursiveness of re-use?

Idea: build a linkage-level configuration language supporting adaptation.

An example configuration



Knit-like code for the example (1)

```
// simplified Knit-esque syntax with added adaptation features

unit myTupSpc {
  exports [ prod { (int, bit list) in_ordered() },
            cons { void      out(int, bit list) } ];
  files { obj_elf("C", tuplespace.o) }
}

unit myTupleProducer {
  imports [ dest { void output(bit list, int) } ];
  exports [ /* ... */ ];
  files { obj_elf("C", tupleprod.o) }
}

unit myStreamConsumer {
  imports [ streamProvider { int read(byte addr, int) } ];
  exports [ /* ... */ ];
  files { obj_elf("C", streamcons.o) }
}
```

Knit-like code for the example (2)

```
unit takeContiguous {
  imports [ source { (int, bit list) in_monotonic() } ];
  exports [ listProvider { (int, bit list) list get() } ];
  files { obj_elf("C", take_contig.o) }
}

unit Process {
  exports [ /* ... */ ];
  link exec_elf("process") {
    myTupleProducer.dest  <- myTupSpc.cons { output(a, b) <- out(b, a) }
    takeContiguous.source <- myTupSpc.prod { in_monotonic <- in_ordered }
    myStreamConsumer.streamProvider <- {
      read <- flatten(map (project #2),
        takeContiguous.listProvider.get)
    }
  }
}
```

Summary of the design

Key points:

- usual languages are for implementing *functionality*
- tackle *integration* in the linkage domain

The following features to be important:

- explicit hierarchy
- ad-hoc adaptation in the configuration language (convenience)
- set of adaptations is open; may define externally
  - allows for generative adaptation

What can you do after that?

So far, so good. What cool things come next?

- apply adaptation algorithms (e.g Yellin & Strom)
- case studies (re-use features from Firefox; translate plug-ins)
- demonstrate reduced coupling measurement
  - might need new coupling measure based on interface complexity
- re-implement as dynamic loading
- refactor existing code
- pluggable checking

Time I got coding on all of that. Enough about my work!

David (1): Orangepath compiler

Orangepath: synthesis of hardware/software systems

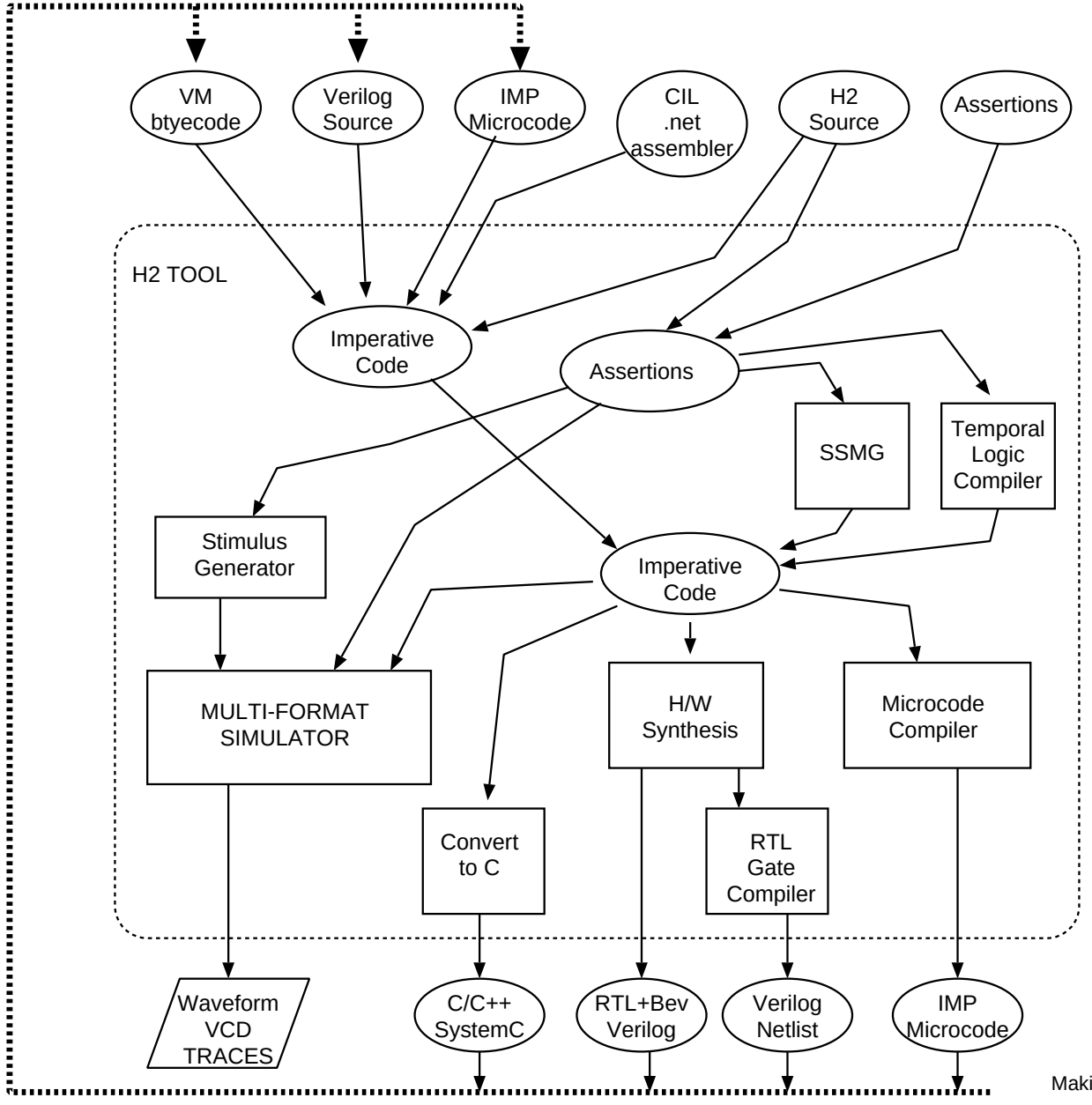
Different deployments mean different requirements:

- different object-level representations
- different partitionings between hardware and software

Idea: use the hourglass approach.

- many input formats (some high-level, nondeterministic)
- goal-based refinement engine (various algorithms)
- many output formats
  - Verilog, SystemC, H2 bytecode, microcontroller code
  - some may be fed back as input
- automatic hardware/software partitioning

Orangepath picture



About the H2 compiler

Compiler internals:

- internal representation is H2 machine
  - hierarchically structured imperative machine
  - contains variable declarations, executable code and goals (assertions)
- various refinement algorithms: . . . , SAT-based
- also contains a simulator: generates output traces based on
  - manually-specified stimulus, or
  - rule-constrained pseudo-random stimuli

Orangepath: exciting recent work

Original compiler front-end used custom H2 language

- reminiscent of C + Verilog

Recent work adds .NET CIL “front”-end

- joint work with Satnam Singh
- supports code in (subsets of) any .NET-ported language!

David (2): Pushlogic

Consider a “home area network”. . .

- . . . or any other domain with
  - dynamic population of independent components
  - strong local connectivity
- e.g. car, factory, . . .

Problem: want components to

- interact usefully (“do the right thing”)
  - with a minimum of human effort
- *not* interact harmfully
  - with some automatic level of assurance

Feature interaction

Feature interaction is a common problem. Examples:

- telephony: e.g. call screening + forward-when-busy = ?
- concurrent processes trying to set different TV channels
- vacation message to mailing list

Classification of feature interactions:

- shared trigger (typically race conditions)
- sequential trigger (unanticipated consequence)
- looping (special case of above)
- missed trigger (example?)

The Pebbles model

Idea: separate out *proactive* from *reactive*.

- *reactive* “pebbles” contain core functionality
- *proactive* scripts wire them together into applications
- centralised manager, implementing tuple-space

Require both pebbles and scripts to describe their own behaviour:

- pebbles are controlled by local *bundles* of bytecode
- scripts are also bundles of bytecode
- all of these can be expressed in the Pushlogic language
- safety conditions are described by `always` clauses

Turn behaviour into LTS; safety conditions into CTL; can model-check.

Pushlogic example: CD player (1)

```
def bundle PioneerDvd()
{
  input devices#keypad#now : { Stop : Play Pause Eject Tfwd Trwd};
  output works#cmd : {stop : play pause resume eject};
  output devices#keypad#playled : {0 : 1};
  output devices#keypad#pauseled : {0 : 1};
  output devices#keypad#stopled : {0 : 1};
  input parts#mech#stat#track : {0..99};
  input parts#mech#stat#sec : {0..59};
  input parts#mech#stat#min : {0..99};
  input parts#mech#stat#idx : {0..99};
  output parts#disp#track : {0..99};
  output parts#disp#sec : {0..59};
  output parts#disp#min : {0..99};
  output parts#disp#idx : {0..99};
```

Pushlogic example: CD player (2)

```
with devices#keypad
  if (#now == stop)
  { #(playled, pauseled, stopled)  := (0,0,1);
    works#cmd  := stop;
  }
  else if (#now == play)
  { #(playled, pauseled, stopled)  := (1,0,0);
    works#cmd  := play;
  }
  else if (#now == pause)
  { // This bit is not idempotent - expect a compile time warning!
    if (works#cmd == play)
    { #(playled, pauseled, stopled)  := (1,1,0);
      works#cmd  := pause;
    }
    else
    { #(playled, pauseled, stopled)  := (1,0,0);
      works#cmd  := play;
    }
  }
```

Checking and dynamism

This approach to checking has at least two notable advantages:

- check both proactive and reactive code (c.f. service description/discovery)
- can dynamically check application code (c.f. proof-carrying)

Turning bytecode into LTS

Translation of bytecode into model-checkable LTS:

1. split off start-up path (fork threads, allocate storage, execute ctors, . . .)
   - can run this path before checking
   - determines finite state size
   - requires constraints on CIL
     - s.t. no dyn storage alloc'd outside constructors
2. in main loop, create one state for
   - each blocking CIL primitive, and
   - each program label reachable through more than one path
3. annotate arcs with I/O operations and variable updates

This yields a LTS checkable for various properties. . .

Making model checking scale

Two classes of property may be specified in CTL:

- safety
  - comparatively easy to model-check
- liveness
  - hard to check because of non-monotonicity

Can sometimes express liveness in the form of safety. . .

Make model-checking scale better by

- skipping pebble internals (only bundles are checked)
- skipping state-space local to a pebble (dataflow analysis / ‘cone of influence’)

Atif: binding using ontologies

Want to deploy “canned” (re-usable) scripts over our devices.

Q: How do we bind formal pebbles to actual pebbles?

A: Use ontologies.

- replace previous domain manager with ontology-aware one
- relates concepts among devices; ontology itself is re-used
- can use semantic web rule language (SWRL) alongside Pushlogic

Dynamic binding: rehydration

The dynamic binding mechanism is called *rehydration*. It is triggered when

- an application is launched; or
- a rule (Pushlogic or SWRL) is fired
- a canned ontology is deployed onto the domain
- a quantifier over entities is expanded

Jin: SoC interface automata synthesis

Remember Orangepath?

- one key application is *interface synthesis*
  - i.e. synthesise glue between SoC IP blocks
- higher-level design and simulation $\rightarrow$ faster development

Since most SoC IP blocks are specified in SystemC:

- extract finite-state protocol specs from SystemC
- SystemC $\rightarrow$ LLVM $\rightarrow$ H2
- use various existing algorithms to synthesise adapters
  - e.g. De Alfaro, “Interface Automata”, ESEC/FSE ’01
- can also adapt between RTL and netlist-level interfaces

Henry, Aisha, Tope, Behzad, questions

To close, a shambolic rounding-up of loose ends:

- Yiannis: secure multi-tier web application development
  - collaborating with Andrew Gordon at MSR
- Henry: (take it away)
- Aisha:
- Tope: lots of Pushlogic work. . .
- the mysterious Behzad: requirements engineering, “pharmacological approach to refactoring”

That’s all, folks. Questions?