New-Age Component Linking: Compilers Must Speak Constraints

Alex Shafarenko
Compiler Technology & Computer Architecture Group
University of Hertfordshire

HPC needs software engineeering

- As mentioned by G.Joubert on day 1
- Complexity of design is only mitigated by decomposition and encapsulation
  - Componentisation, local code
- Efficiency requires flexible composition and aggregation
  - Global information for local decisions
- Software crisis: software engineering contradicts efficiency

Classical componentisation

- Computational and HPC s/w has been componentised from the start
- SSPs of the 60s & 70s
  - Collection of subroutines to be used together
  - Linked by name
  - Interfaces must match but are not checked

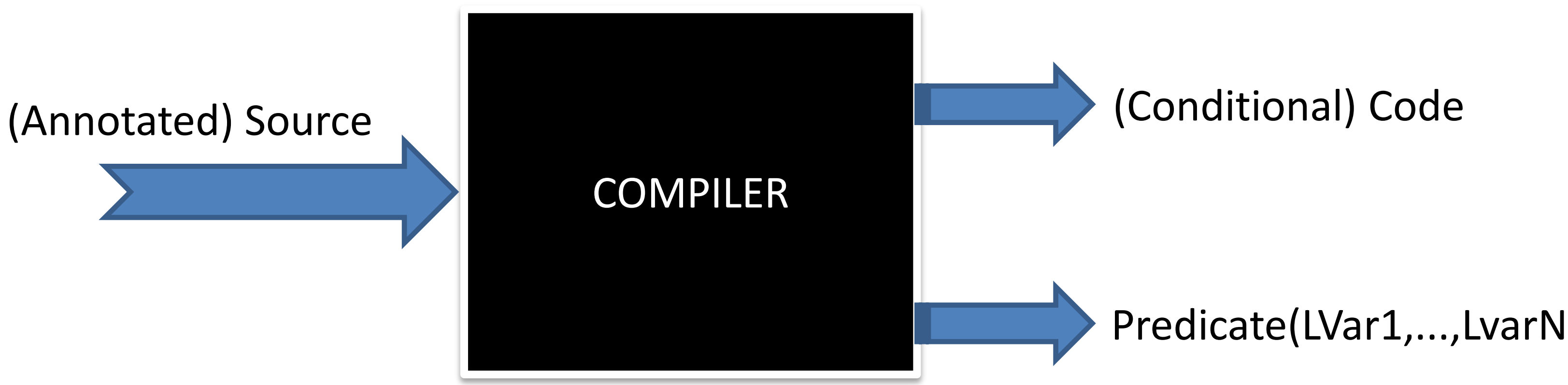
Linking by unification of addresses

- The <u>logic</u> perspective on Fortran and the Linker:
  - External ref in the calling subroutine: **call x(...)**
    - **∃sub: Sub.name="x", Sub.addr =A.** JSR A(...)
  - Exportable address in the callee: **subroutine x(...)**
    - ∃ **Fresh_addr =This.addr, This.name="x", This.code=...**
  - **Result: *Sub* is unified with *This* by the Linker**
  - **BUT THIS logic IS HARDWIRED INTO IT**

The “step forward”

- Object style component technology
  - Collection of classes/methods
  - Hierarchies verified by type/class logic
- A major limitation of the approach
  - Based on type, hence must be provable
    - Or dynamically checkable
    - Not in itself adaptive, approximative and coalescable
  - Still based on unification
- Types = the agenda of the verification cohort
- Still need logic, but different motivation

A new enlightened view



- Compilers communicate external needs using existential qualifiers
- Compilers communicate exportable properties using ground facts and universal quantifiers
- Code and Predicate may share existential LVars.
  - When such LVars are constrained, the code may be specialised

Example: one step beyond the basics

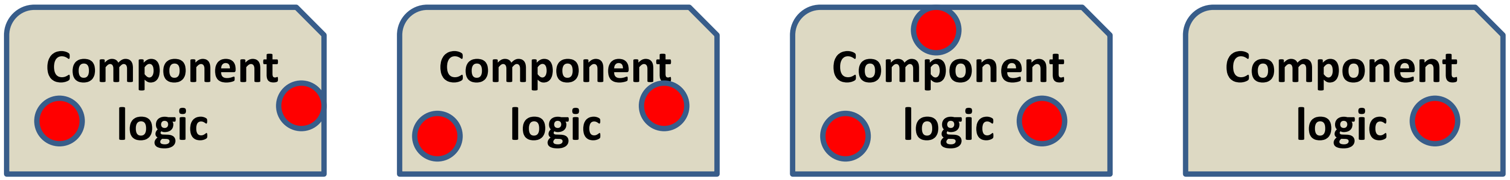
```
subroutine foo
....
do  i=1, 100
    If(x(i)>1.0) x=1.0
    If(x(i)<0.) x(i)=0.
enddo
call sort(100,x)
call sort(256,y)
....
```

∃ **Sub:** Sub.Name="Sort",
Sub.P1=100\/ Sub.P1=256,
Sub.P1=100 ->
(0.<=Sub.P2<=1.)

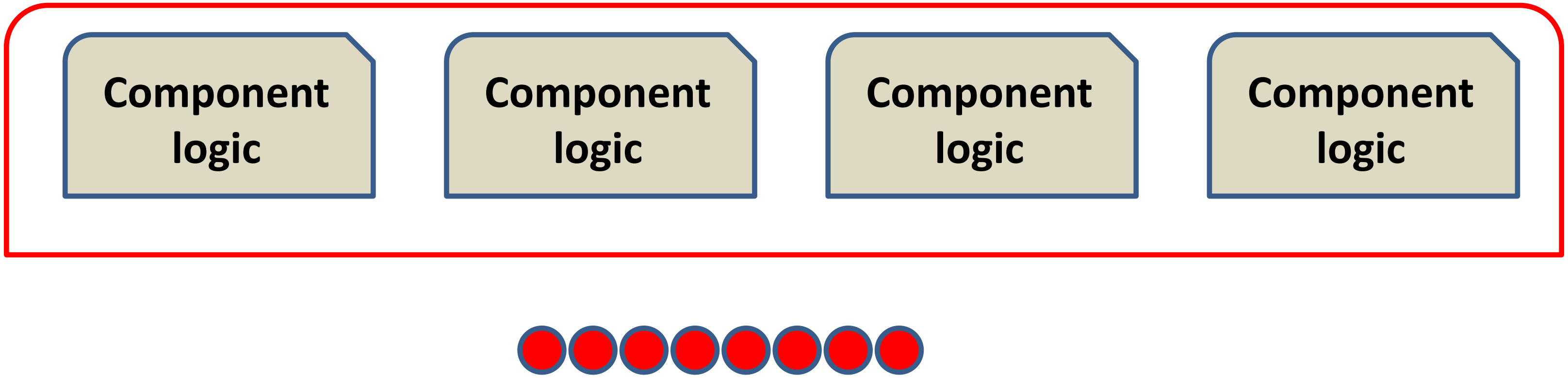
```
Subroutine sort(n,x)
    ovl1: sortPwr2(n,x,low,high)
...
    ovl2: sortGen(n,x)
...
```

Pwr2(This.P1) -> Ovl=**ovl1**;
Ovl=**ovl2**;
∃P,Q: (P<=This.P2<=Q) -> **low**=P,
high=Q; low=-infty,
high=+infty

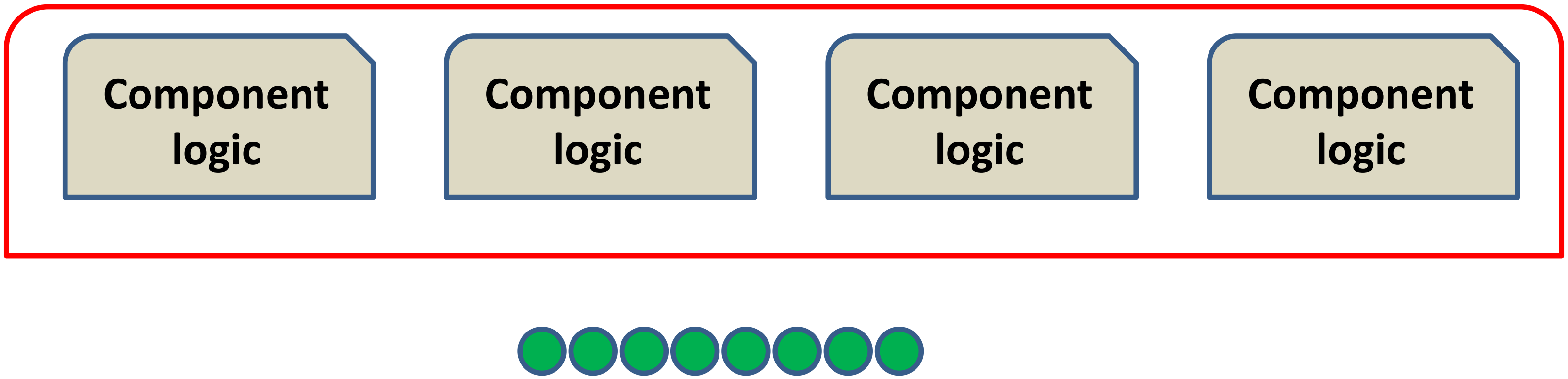
Linker becomes Constraint Aggregator



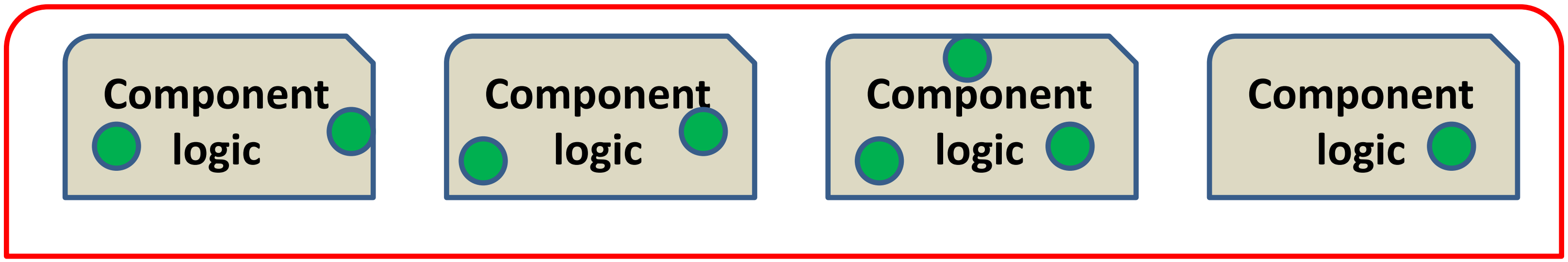
Linker becomes Constraint Aggregator



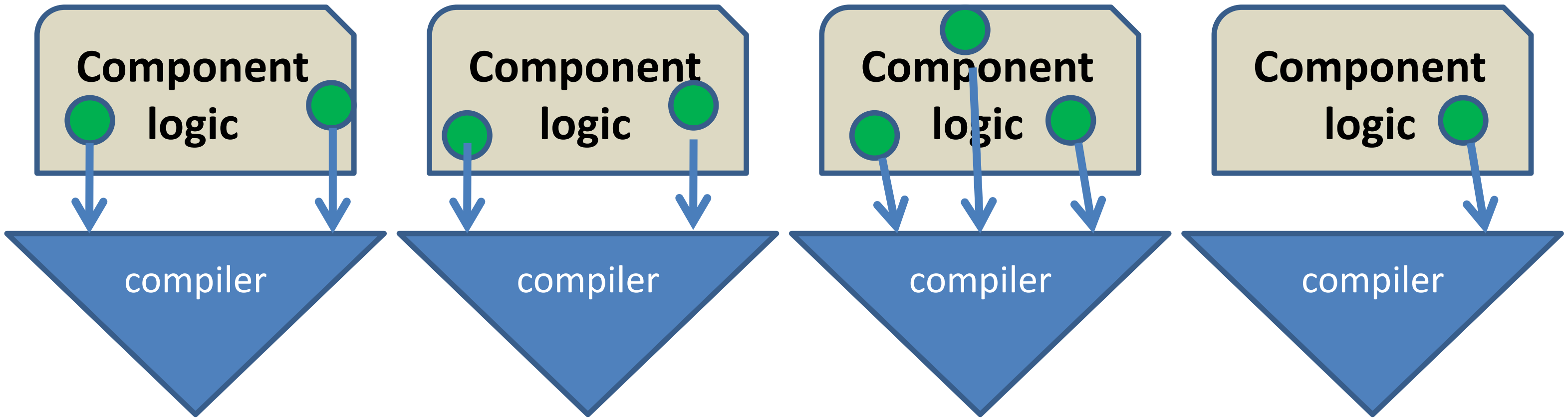
Linker becomes Constraint Aggregator



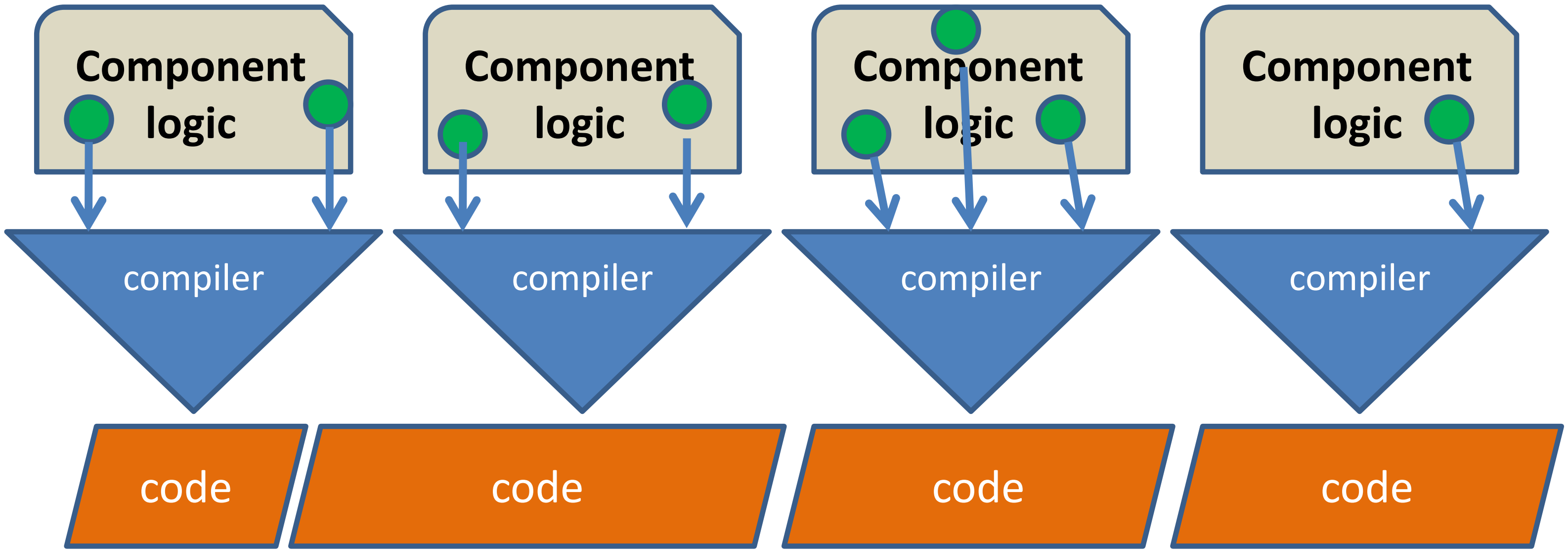
Linker becomes Constraint Aggregator



Linker becomes Constraint Aggregator



Linker becomes Constraint Aggregator

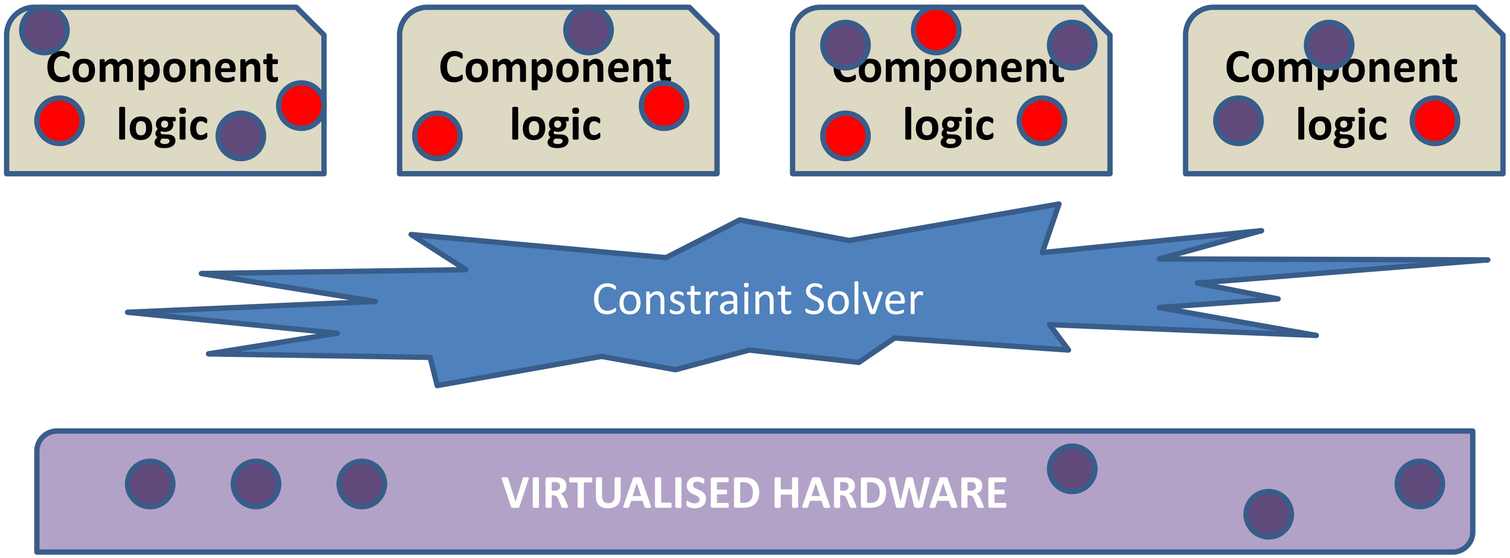


- Classical scheme: deploy+resolve becomes
  - Aggregate
  - Deploy?
  - Resolve

And now to the main point

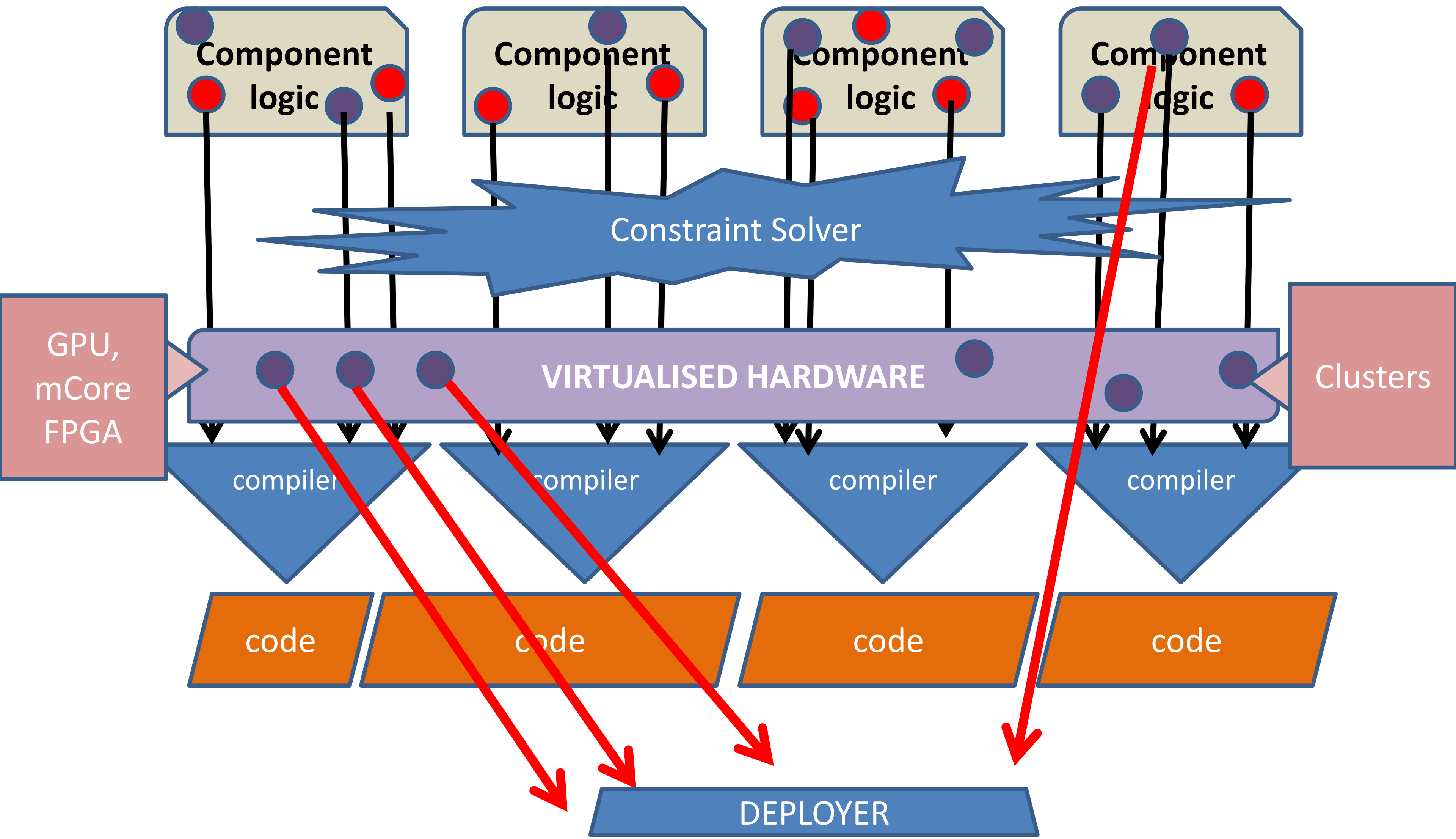
- A heterogeneous, HPC environment
  - Several types of platform: multicore, GPGPU, FPGA
  - Plethora of performance-affecting properties
  - Still need component encapsulation for software engineering purposes
  - Yet wish to "assemble" components intelligently
    - i.e. significantly adapt, customise and tune up
- Hardware intelligence needed as well!

Extrafunctional aggregation



- Evars: environment variables. Things like:
  - Places, addresses, channels, power-limits, memory capacity, throughput, cache structures...
  - Cloud costs

The Scheme



Component logic
Component logic
Component logic
Component logic
Constraint Solver
GPU, mCore FPGA
VIRTUALISED HARDWARE
Clusters
compiler
compiler
compiler
compiler
code
code
code
code
DEPLOYER

Difference between functional & extrafunctional aggregation

- Functional aggregation
  - Follows datapaths
  - Data source asserts properties
  - Data recipient imposes constraints
  - Properties meet constraints -> LVars get assigned
- Extrafunctional aggregation
  - Code specifies requirements by constraining EVars
  - Virtual hardware imposes further constraints
    - Sharing of the resources results in further constraints
  - Aggregation NOT by unification,
    - by constraint solving
    - And – possibly – subsequent optimisation...

System Architecture

- Compiler:
  - Input: component source (+ annot.?)
    - Additionally, Phase 2: LVar bindings (complete), some EVar bindings.
  - Output:
    - Phase 1: Predicates on interface LVars and on local EVars
    - Phase 2: Binary code

System Architecture

- Virtual Hardware
  - Input:
    - Phase 1: compiler generated constraints on EVars
  - Output:
    - Phase 1: Fresh EVars for each component, to be constrained by the compiler, bound EVars for communicating platform properties
    - Phase 2: Bindings of the EVars used for placement; bindings of the EVars used by the compiler.

System Architecture

- Virtual hardware represents placement algorithms as constraints
- Contains a constraint –limited optimiser
  - Placement problem NP complete
  - Heuristics
  - Interface with the general constraint solver

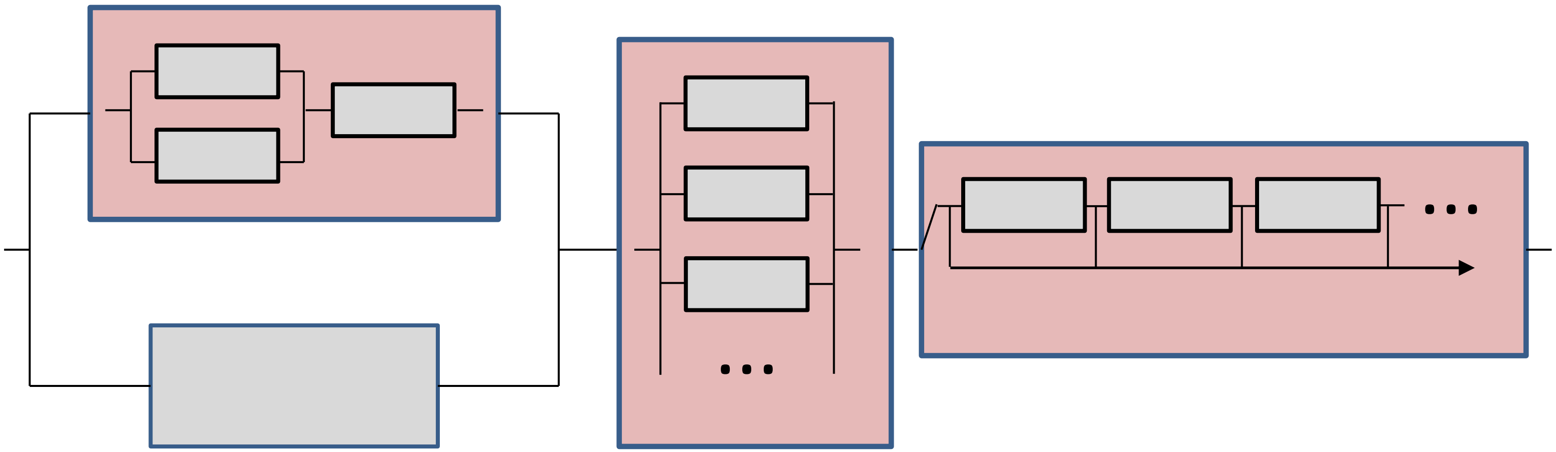
Project ADVANCE

- Framework 7 STREP
  - UH, U St. Andrews, UvA, Utwente
  - SAP (Karlsruhe), Philips (Eindhoven) & 2 SMEs
- Aim: an intelligent software architecture for heterogeneous systems
  - Components communicate STATISTICAL properties
  - Hardware is virtualised via a STATISTICAL model
  - The statistics of data streams are gathered by observation

The Achilles Heel of New Age Linking

- Traceability of control
  - Constraint chaining requires causality
  - Causality necessitates static control flow
    - Otherwise input constraints do not mesh with output constrains
  - This has been the reason of slow uptake
  - Mitigated by coordination

S-Net Coordinated Networks



- Single-Input, Single-Output Boxes
- Type directed routing
- Out of order split-mergers

- Each box a conventional program
- Extremely small API

First phase: CAL

- Constraint Aggregation Language
  - Target representation: a coordinated streaming network (S-Net)
  - Nodes represent components
  - Virtual hardware: distributed multithreaded platform
  - Component compilers speak CAL

CAL example

```
box MYBOX: (a,k) => (b), (c,d)

provided $a :=:
     {Type(array, element($t), rank(2), shape($n,($m,nil)) )} \/ $_ ,
               $k:=: {value($kv), Type(int)} \/ $_
use
        =>  $n1=$n+1,
              $base :=:
              {Type(array, element($t), shape ($n1,($m,nil)) )};                -- Clause 1

        => $b :=: $base \/ {rank(2)},
             $d = $base \/ {rank(3)};                                           -- Clause 2

        $kv > $$nthreads * 100
                =>
        $$T0 :=: $m * log($m)/$$nthreads, $$T1 :=: 1;                           -- Clause 3

        $kv <= $$nthreads* 100
                =>
        $$T1 :=: $m^(3/2); $$M1 :=: 0;                                          -- Clause 4
end
```

CAL vocabularies and protocols

- CAL is a language in which logic protocols can be expressed
  - EVars and LVars for interacting with S-Net form a *vocabulary*
  - Vocabularies are not fixed... Depend on the underlying architecture
  - Constraint structures represent protocol conventions
    - Protocols connect components and solvers

Future Work

- Translation of CAL to a constraint solver language
  - Currently planned YICES
- Functional constraint aggregation (an intelligent component interface)
  - C compiler using EVars for optimisation
- Extrafunctional constraint aggregation
  - Placement optimisations based on constraints

Conclusions

- Compilers should speak symbolic properties
- Linkers should in fact be constraint solvers
- Dynamic linking => dynamic constraint solvers
- Extrafunctional dictionaries should be developed for classes of hardware
- Large scale demonstrator required