

Co-designing a Language, Tool-chain, and Architecture: *Lessons Learnt from the POETS Project*

David Thomas

University of Southampton (2021...)

Imperial College (1996..2021)

POETS : What it is

- EPSRC Programme Grant
 - Running 2016 till 2022
- Four university partners:
 - Newcastle
 - Imperial
 - Southampton
 - Cambridge



This talk is a bit of a retrospective

POETS : The big idea

“Create a new framework for developing and executing event-driven applications using asynchronous algorithms in distributed hardware”

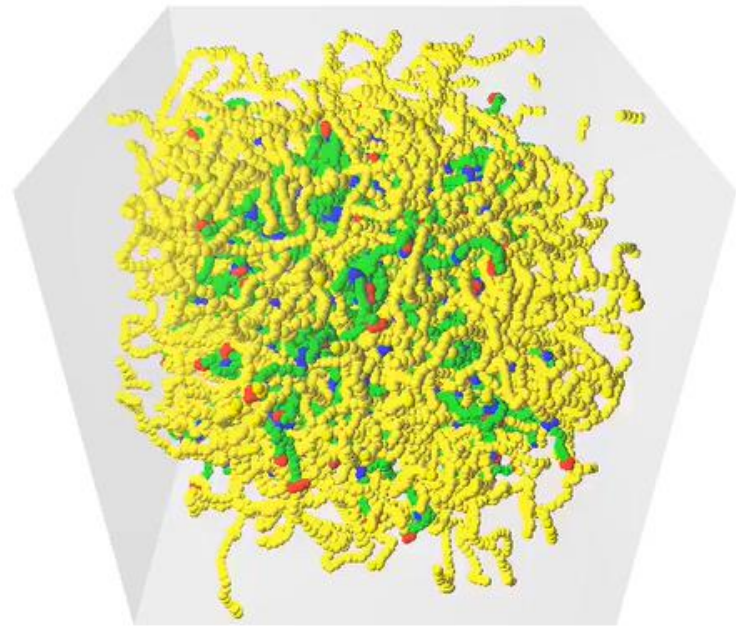
Event-driven = millions of shared-nothing threads sending tiny messages

- Research challenges:
 - **Applications**: what should event-first algorithms look like?
 - **Languages**: what language do we use?
 - **Compilation**: how do we describe and compile such applications?
 - **Hardware**: what does this distributed hardware look like?
- Management challenges:
 - We don't have a target application, language, compiler or architecture
 - How do we even get started?

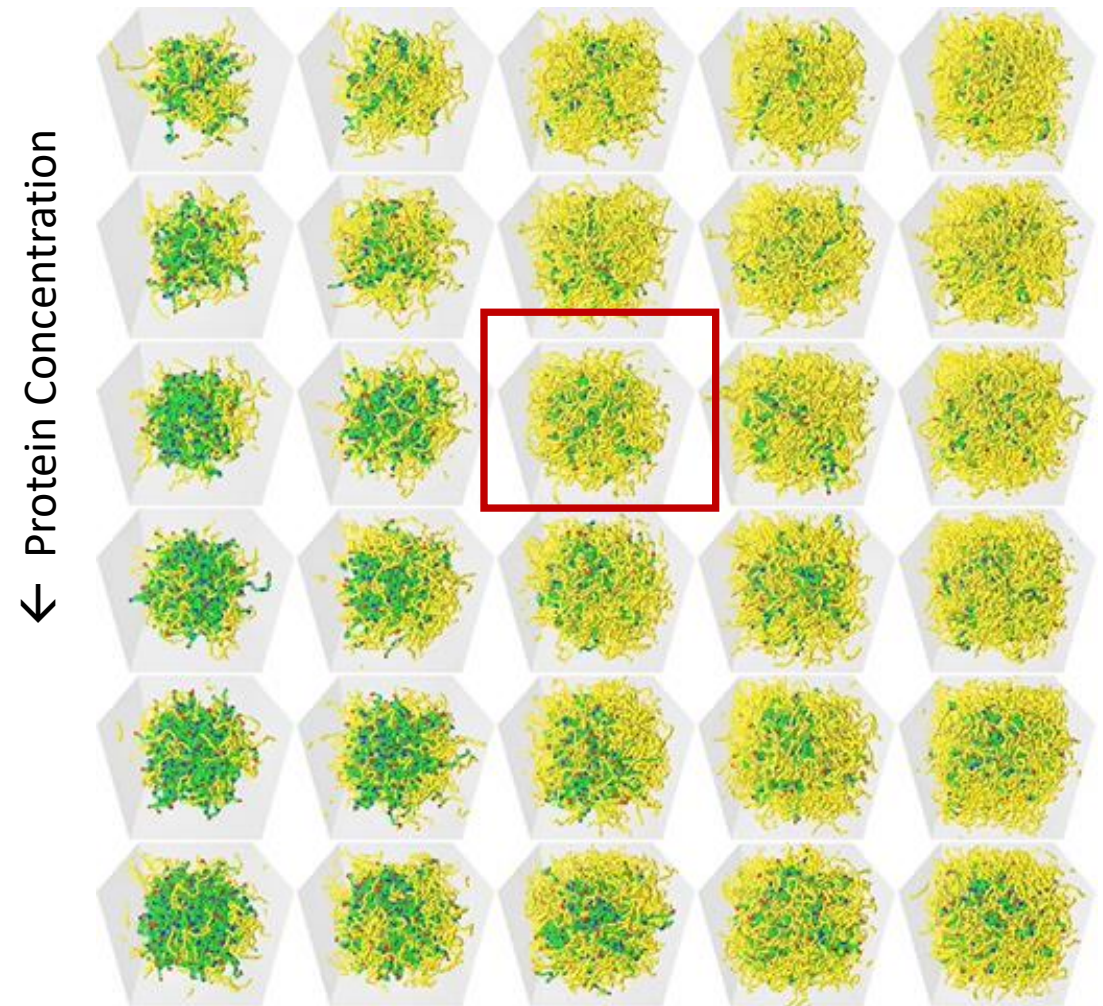
POETS : What we achieved over six years

- **Applications:** portfolio of asynchronous event-based applications
 - Flagship is “Dissipative Particle Dynamics (DPD)”
 - Allowed us to provide speed-up for used in published chemical research

DPD : Exploring phase transitions with POETS



1M particles for 1M time-steps



POETS : What we achieved over six years

- **Applications:** portfolio of asynchronous event-based applications
 - Flagship is “Dissipative Particle Dynamics (DPD)”
 - Allowed us to provide speed-up for used in published chemical research
- **Compilers:** multiple compilers and simulators for one language
 - Main back-end is “The Orchestrator”
 - Performs place and route for applications with 1M+ logical threads
- **Architecture:** bespoke CPU architecture and network called “Tinsel”
 - Custom RISC-V architecture with deeply embedded routed network
 - Currently supports 50K hardware threads on 50 FPGAs

Tinsel : 50K hardware threads per rack



NANDA, 2023/09/05



David Thomas (dbt1c21@soton.ac.uk)



Lessons learnt

Experience of co-designing language, compiler, and hardware stacks

POETS : Lessons learnt

- Specific context:
 - **Co-design**: languages, applications, compilers, and hardware
 - **Multi-partner team**: 12-15 people across four universities
 - **Time-scale**: 6 year project
 - **Changing team**: no post-doc who started with project is still in post
- Who these lessons might be for
 - PhD students considering combined hw+sw research (maybe)
 - Long-term projects with a hardware+software stack
 - Me (writing programme grants)

Lessons Learnt : stating the obvious

1. Abstractions:

1. Set the hardware free
2. Go formal or go home
3. Syntax doesn't matter

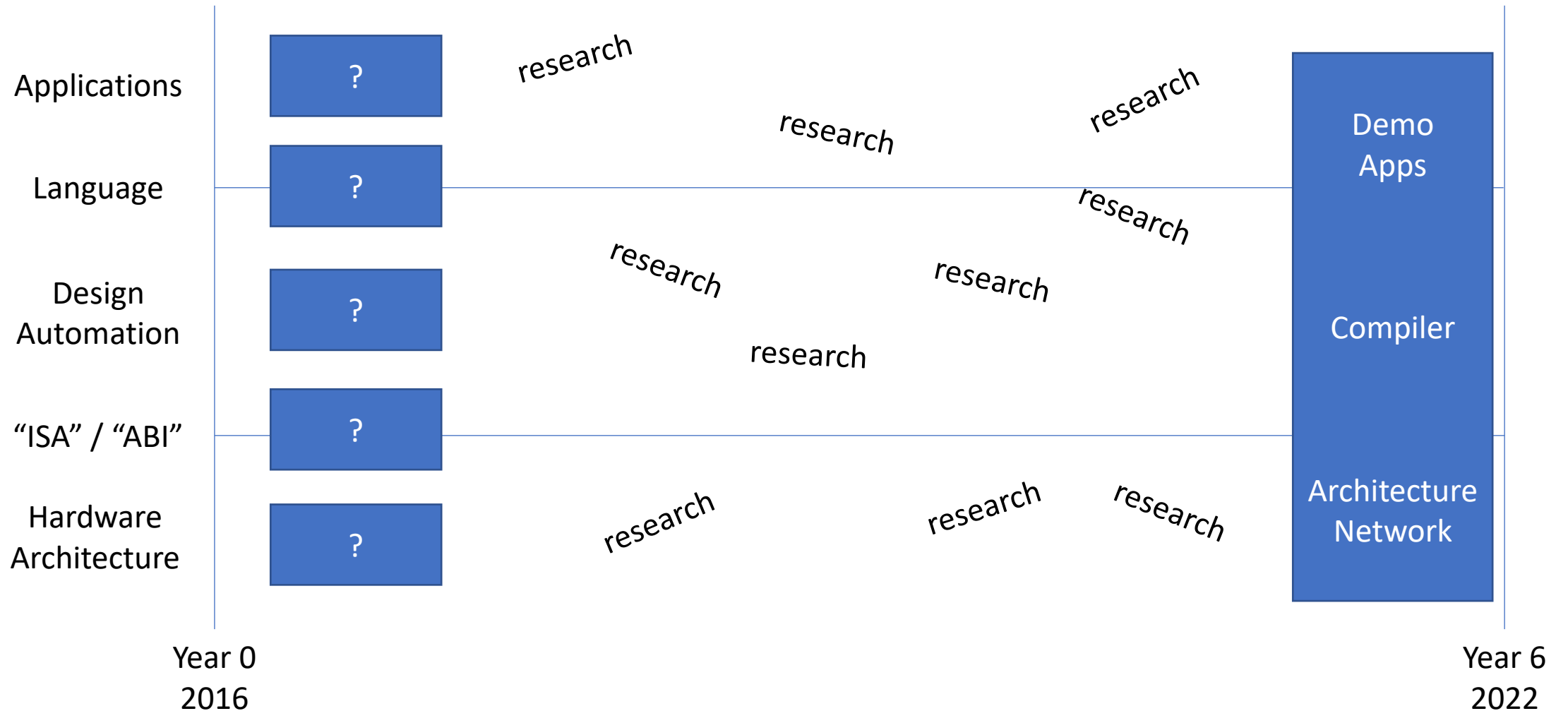
2. Development:

1. Integration tests over unit tests
2. Waterfall sucks
3. Agile sucks
4. Document the “why”; show the “how”
5. Hardware in the loop verification is key

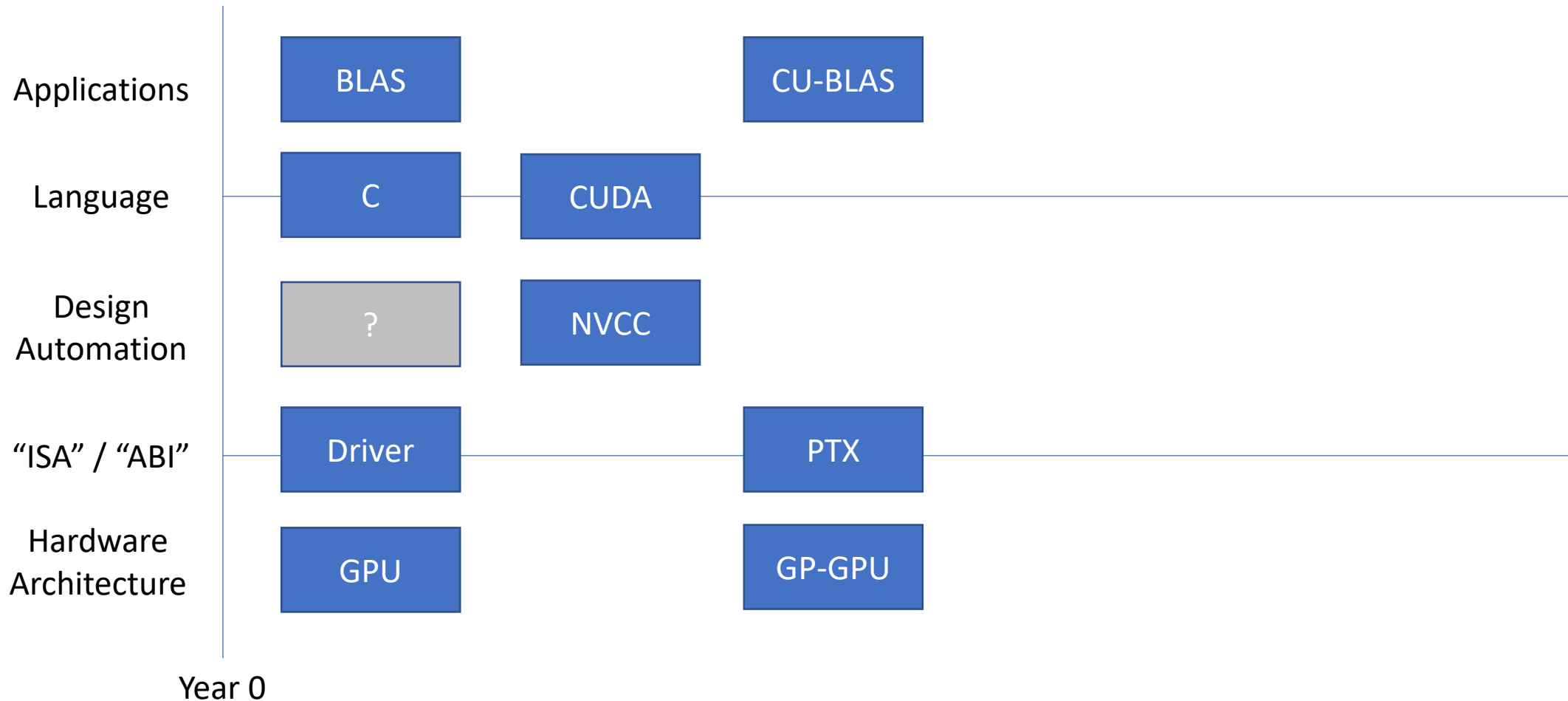
3. Management:

1. Once it all “works” most of the work is ahead of you
2. Everyone can see **all** the repos, **all** the time

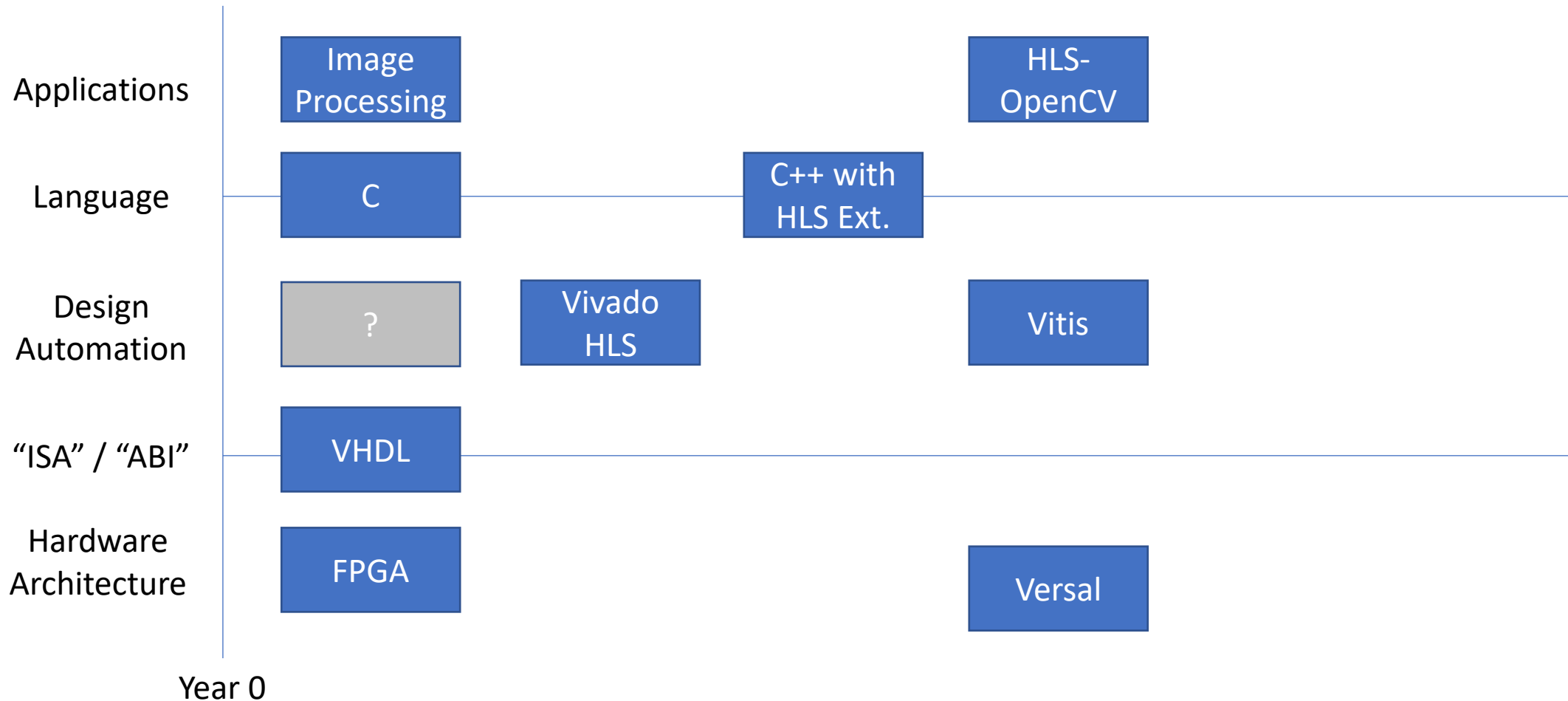
POETS : The management challenge



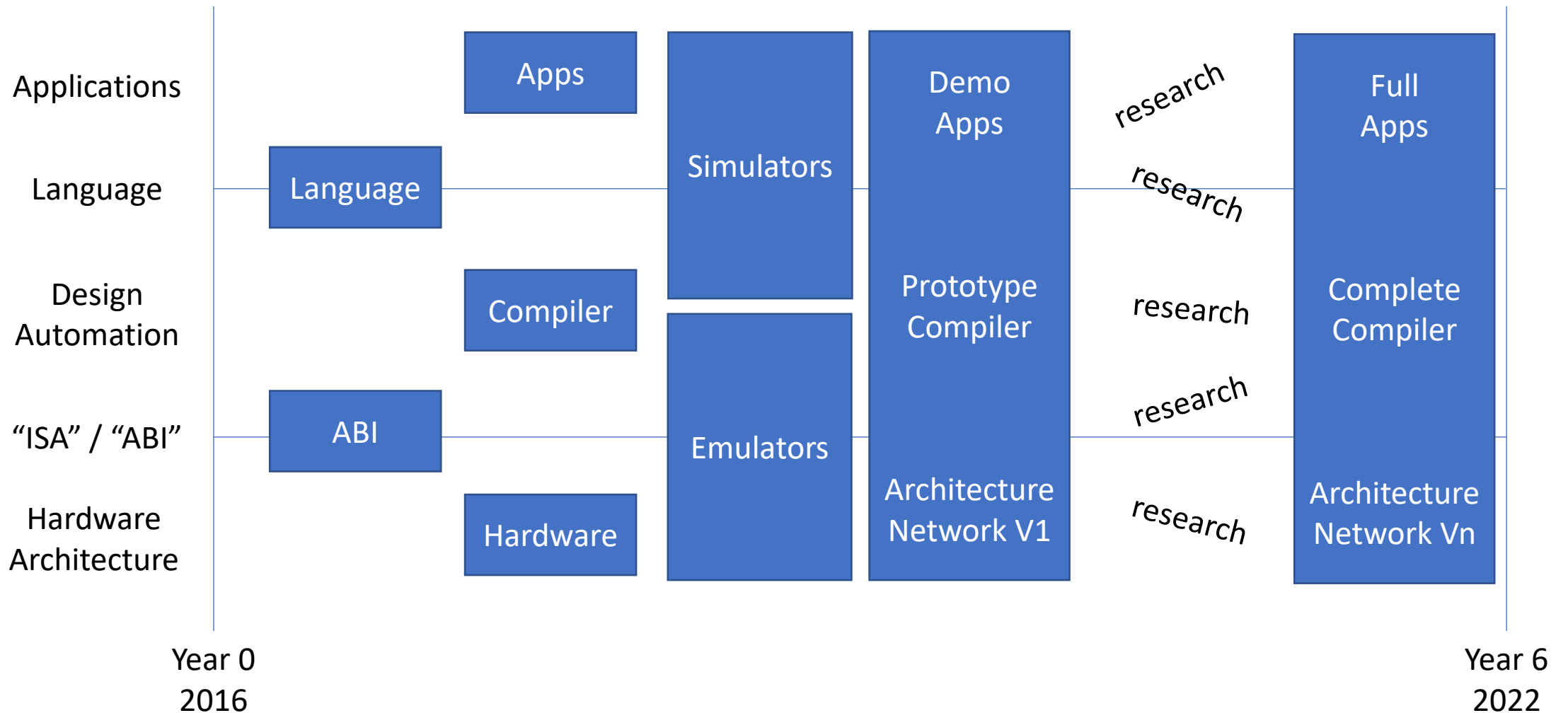
POETS : Compared with CUDA



POETS : Compared with FPGA HLS



POETS : Our approach



Abstractions: set the hardware free

- ***Temptation***: narrow down language to support software
 - Makes it much easier to write and compile applications
 - Makes it easier to verify applications
 - Software+tools people can move faster and break things
- ***Problem***: you're imposing constraints on hardware architects
 - Everything that makes software easier makes hardware harder
 - Initial assumptions become entrenched in apps and compilers
 - You'll end up with hardware that looks like everything else
 - Hardware has a much slower cycle than software

POETS : Applications as graphs of FSMs

Applications are split into graphs of **devices**

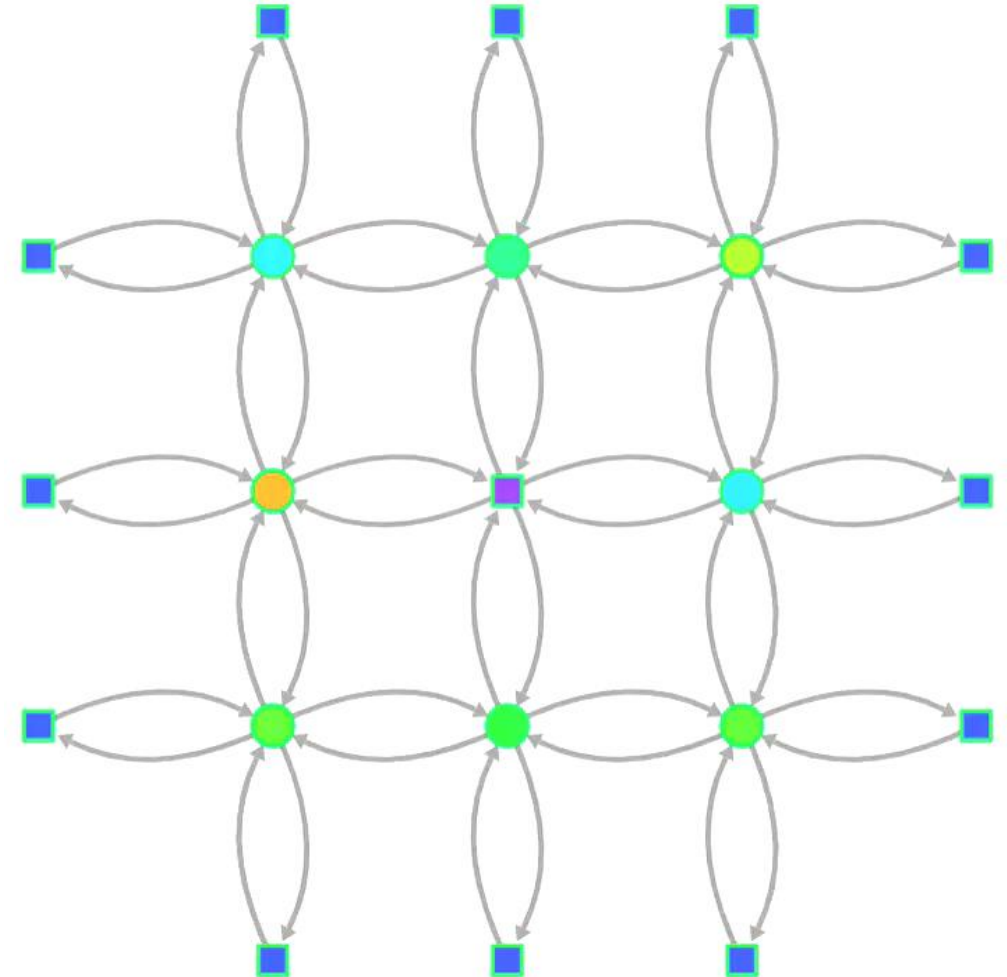
- A device is a finite state machine
- Device state is a tiny part of the global state
- **Only** the device can read and write it's state
- No shared memory – only messages

Receive: message m received by device d

$$d' = \text{receive_handler}(d, m)$$

Send: device d sends a message m

$$(d', m) = \text{send_handler}(d)$$



POETS: Inversion of control

```
class MyDevice
{
    int state;

    void run()
    {
        while(1){
            msg = recv();
            state = receive_handler(state, msg);

            while( more_messages(state) ){
                (state,msg) = send_handler(state);
                send( msg );
            }
        }
    }
};
```

Software is “in control”
Devices are finite state
Hardware buffering is un-bounded

```
class MyDevice
{
    int state;

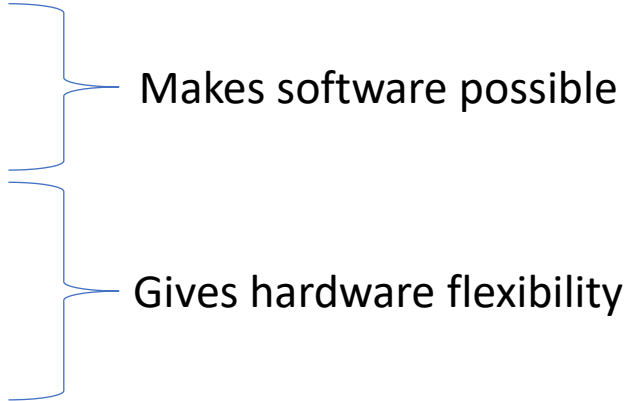
    void on_recv(const Message &msg)
    {
        state = receive_handler(state, msg);
    }

    bool ready_to_send() const
    {
        return more_messages(state);
    }

    void on_send(Message &msg)
    {
        (state,msg) = send_handler(state);
    }
};
```

Hardware+compiler is “in control”
Software must wait for network capacity
Both devices and hardware can be finite state

Abstractions: set the hardware free

- Some key design decisions
 - State changes only occur on send or receive
 - Any message sent will eventually be delivered
 - Devices must wait for an opportunity to send
 - Devices can never delay receipt of a message
 - Messages can arrive in any order
 - This allowed hardware and compiler innovation during project
 - *Network*: changed buffering model and back-pressure
 - *CPU design*: messaging and scheduling primitives changed
 - *Portability*: we were able to compile for other hardware (GPUs and HLS)
- 
- Makes software possible
- Gives hardware flexibility

Abstractions: go formal or go home

- ***Temptation***: define semantics in terms of the implementation
 - Writing the compiler is hard
 - Writing the applications is hard
 - Language semantics are defined by the documentation
 - Test-cases make sure everyone agrees on expected behaviour
- ***Reality***: there are too many corner cases
 - People interpret things differently: applications, compilers, hardware
 - When developing in parallel these will cause problems

POETS : Mis-implementations five years on

```
struct RunTime
{
    vector<Device> devices;
    vector<bool>    ready;

    void run()
    {
        while(1){
            ...
            devices[i].on_rcv(...);
            ready[i] = devices[i].ready_to_send();
            ...

            if(ready[i] && !network_full() ){
                devices[i].on_send(...);
                ready[i] = devices[i].ready_to_send();
            }
        }
    }
};
```

```
struct RunTime
{
    vector<Device> devices;
    vector<bool>    ready;

    void run()
    {
        while(1){
            ...
            devices[i].on_rcv(...);
            ready[i] |= devices[i].ready_to_send();
            ...

            if(ready[i] && !network_full() ){
                devices[i].on_send(...);
                ready[i] = devices[i].ready_to_send();
            }
        }
    }
};
```

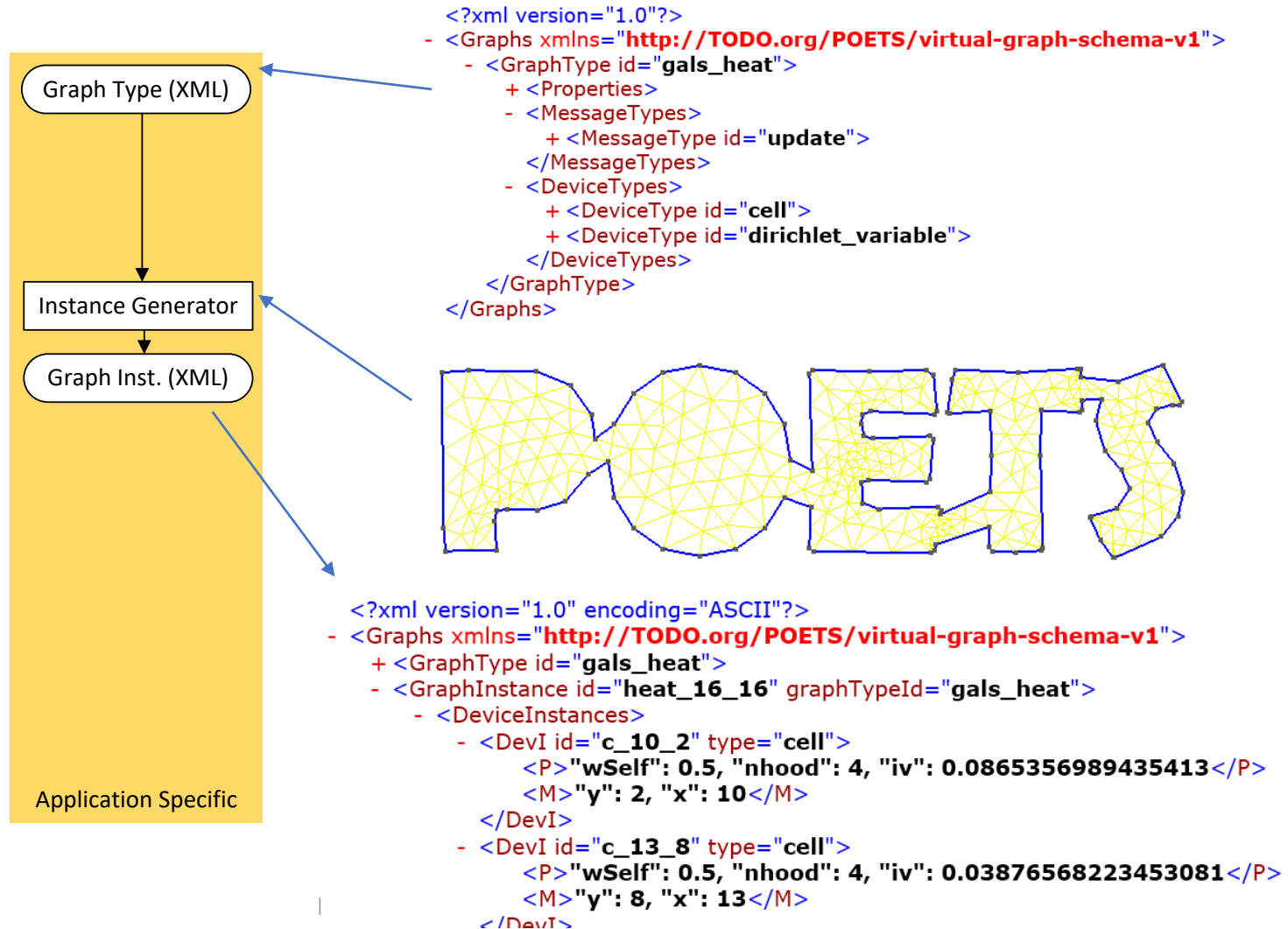
Abstractions: go formal or go home

- Writing formal specs is not enough
 - We *had* formal semantics in year 1
 - Dependently typed in Coq: *a thing of beauty*
 - A more “readable” version in Haskell
 - The formal specs need to be front and centre in the documentation
 - They need to be in a form that everyone can read
 - Five years later: expressed it in python
- *When we used them*, formal semantics had huge benefits
 - Simulation, verification, model checking, equivalence checking, ...

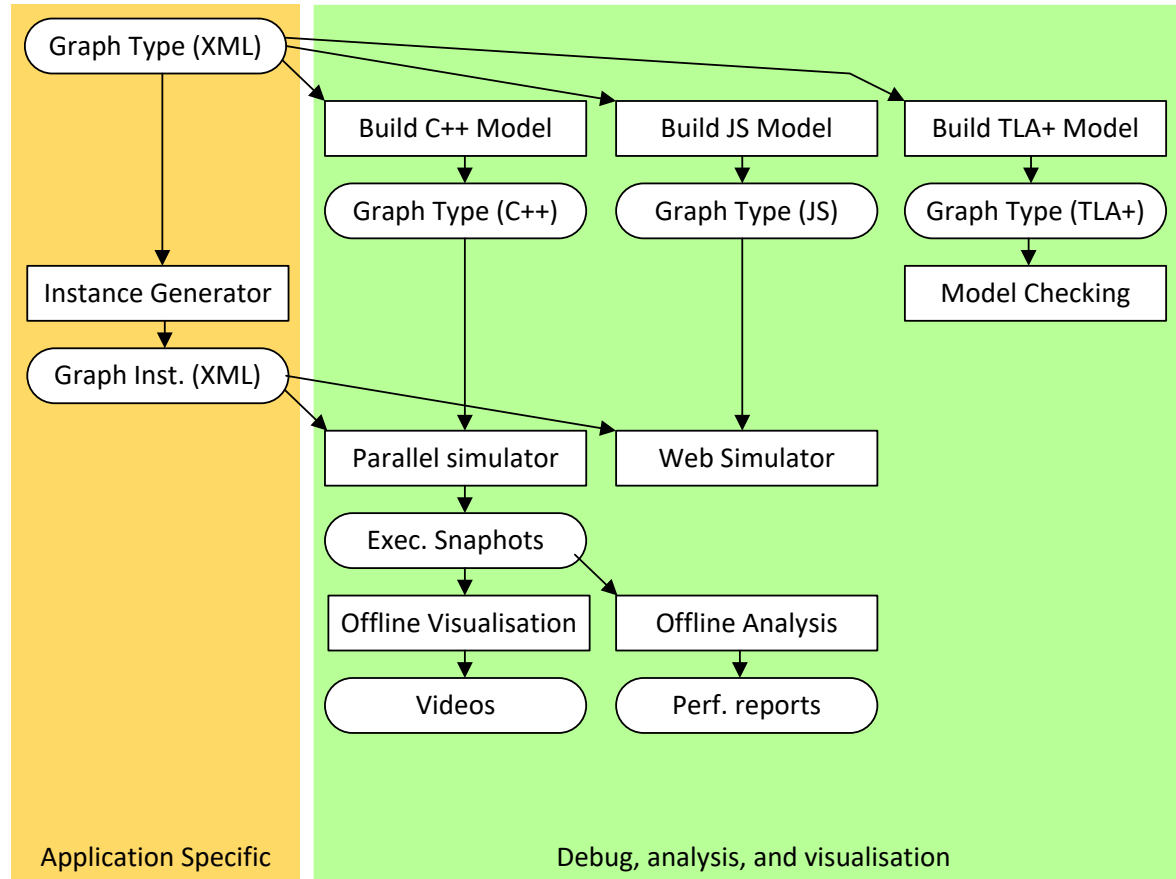
Abstractions: Syntax doesn't matter

- What should the application language look like?
 - Applications are described and specified in this language
 - Compiler will consume language and map into hardware
- ***Temptation***: create a *beautiful* language for graphs and compute
 - Wonderful bespoke grammar and elegant extension points
 - Describes both the functionality of nodes and topology of graphs
 - Implement parser in C++. Then in Python. Then in JavaScript; then...
- ***Practical***: describe it in dumb XML
 - Everyone has an XML parser and generator
 - Automatic versioning support: we went through 4 language revisions
 - Can exploit existing XML schema tools to get free grammar checkers
 - *Downside*: humans hate reading/writing XML, ***but they can if they have to***

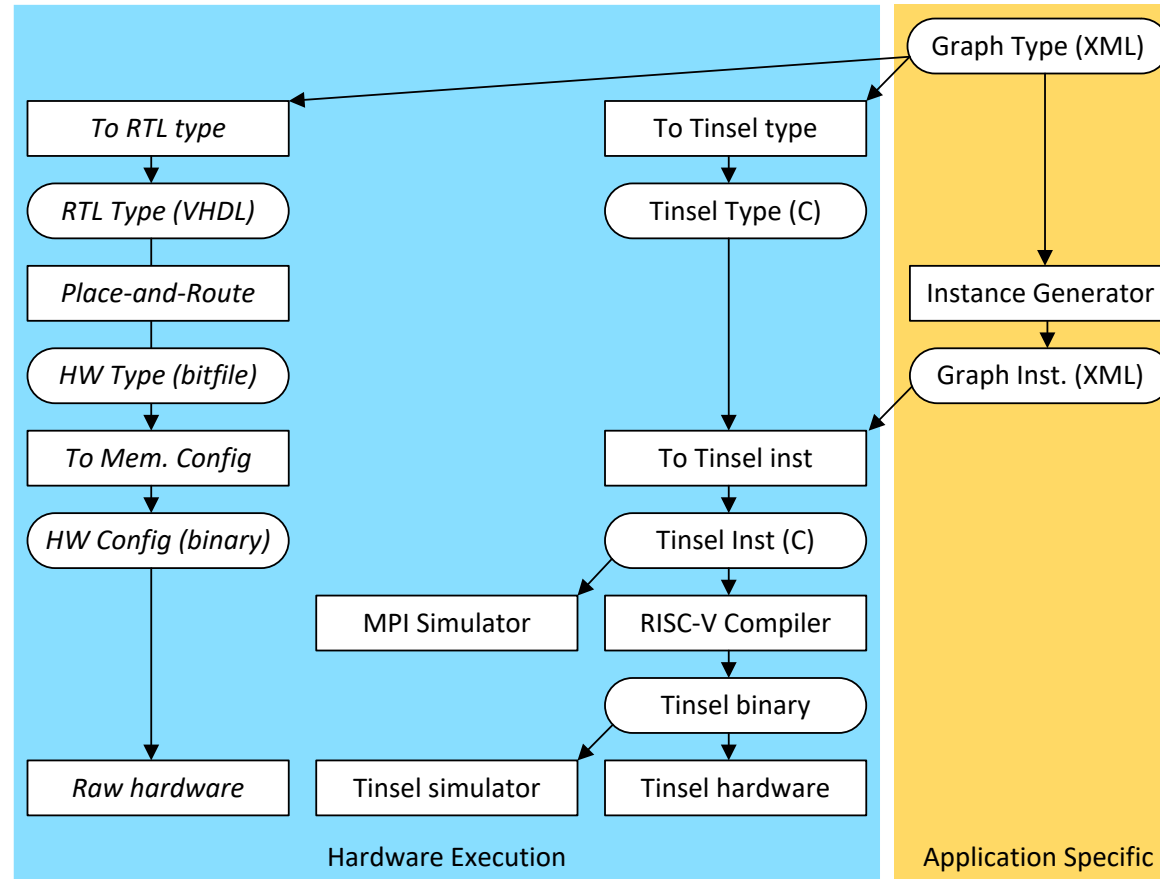
Toolchain : appl. description



Toolchain : development



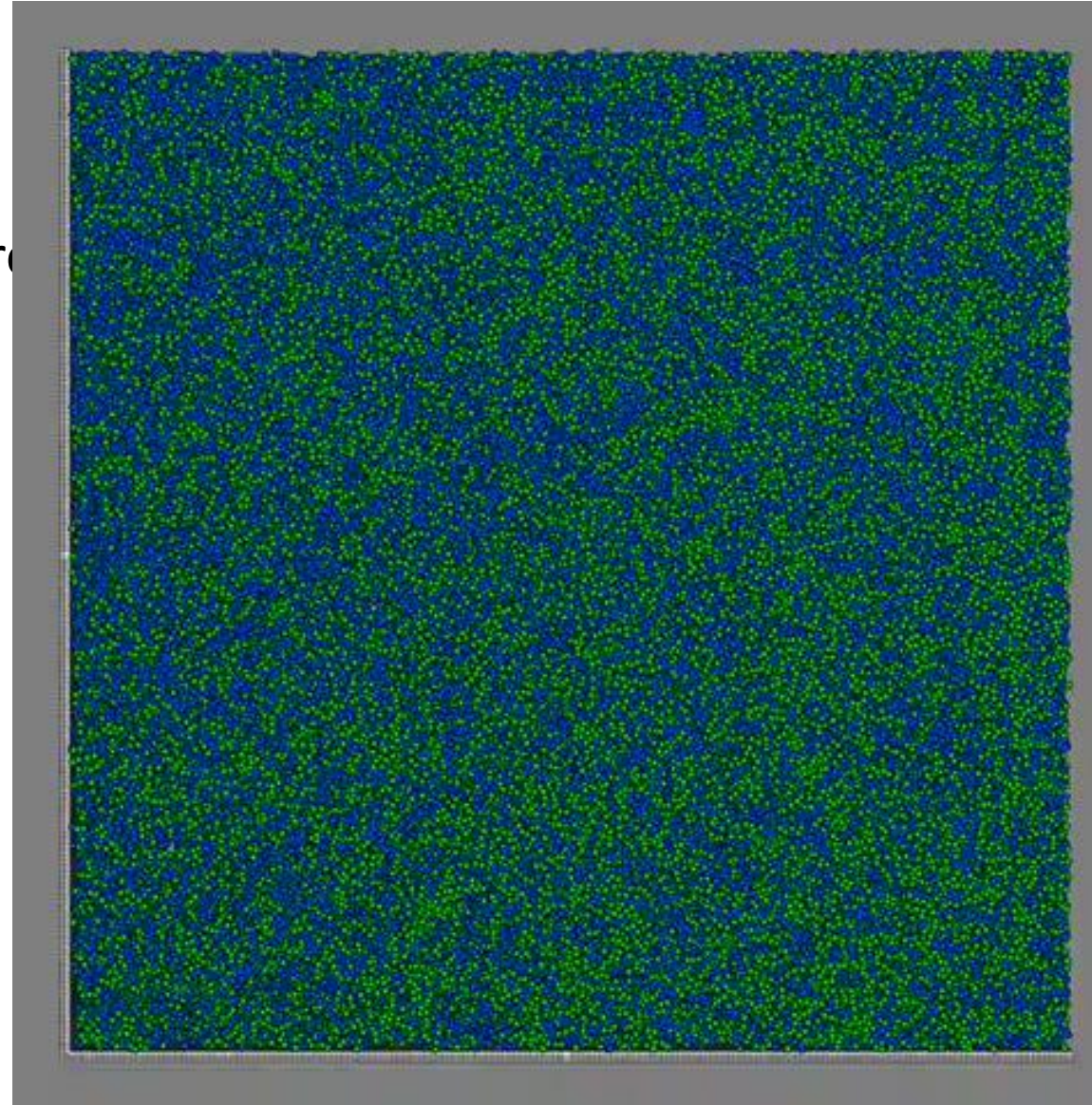
Toolchain : execution



Conclusion

Conclusion (?)

- Genuine co-research in languages and architecture
 - Chicken and egg problem: which one to tackle first?
- Big hardware+software research is complicated
 - Can learn from enterprise management
 - Can put academic research into practice
- POETS got some stuff right, and some wrong
 - We built a complete stack
 - It worked, and we are using it to do science
 - So probably mostly right...?



Abstractions: Integration tests beat semantics

- We have semantics+language: full speed ahead
- ***Temptation***: work independently on apps, compiler, and architecture
 - We all agree on the formal definition, it's bound to work together
- ***Reality***: everyone interprets things slightly differently
 - “Application: always eventually we may send a message”
 - “Compiler: always eventually you will send a message”
- ***Lesson***: full-stack integration tests are incredibly valuable

Development: Waterfall sucks

- We have semantics+language: full speed ahead (again)
- ***Temptation***: each part is a big software/hardware project
 1. Write documentation
 2. Write specifications
 3. Write test-cases
 4. Implement the software/hardware
 5. Unit test the software/hardware
 6. Integration test against other components
- ***Reality***: it might be years before integration tests
 - Very slow feedback for a research project

Development: Agile sucks

- We have semantics+language: full speed ahead (again)
- ***Temptation***: sprints, story-points, stand-ups, oh-my!
 - “We already have weekly meetings: those are stand-ups, right?”
 - “This is research, we need to burst forwards and be incremental!”
- ***Reality***: growing technical debt and unstable interfaces
 - Different research strands have very different time-scales
 - Applications: weeks
 - Compilers: months
 - Hardware: bi-yearly
- ***Recommendation***: set project-wide targets; update them regularly

Development: document “why”; show “how”

- Documentation is awesome in big software+hardware projects
 - Transmits project knowledge over time
 - Transmits project know-how between researchers
- ***Temptation***: write lots of documentation about “how” to do things
 - Long written tutorials used to explain how things are supposed to work
- ***Reality***: this is research; tools and languages are not stable
 - Tutorials get out of date and rot quickly (on a multi-year timescale)
 - We often have to get together and change software+hardware APIs
 - The most important documentation is often *why* it changed
- ***Recommendation***: document “why”; show “how”
 - Have good processes for recording design decisions: git commit logs don’t count
 - We used a process inspired by Python Enhancement Proposals
 - ***Record*** video tutorials on how to use tools and get started
 - Low overhead, and much easier to keep up to date

Development: hardware-in-loop verification

- **Problem:** research created hardware is finicky and scarce
 - There may only be one test-chip or test-installation
 - There is competition for access to the test-hardware
 - It falls over all the time
- **Temptation:** “I’ll run hardware integration manually on commit”
- **Reality:** integration tests are rarely run on test hardware
- **Recommendation:** hardware-in-the-loop continuous integration
 - It is painful to set up
 - It breaks all the time
 - It interferes with other research activities
 - But: you get immediate notice of functional and performance regressions