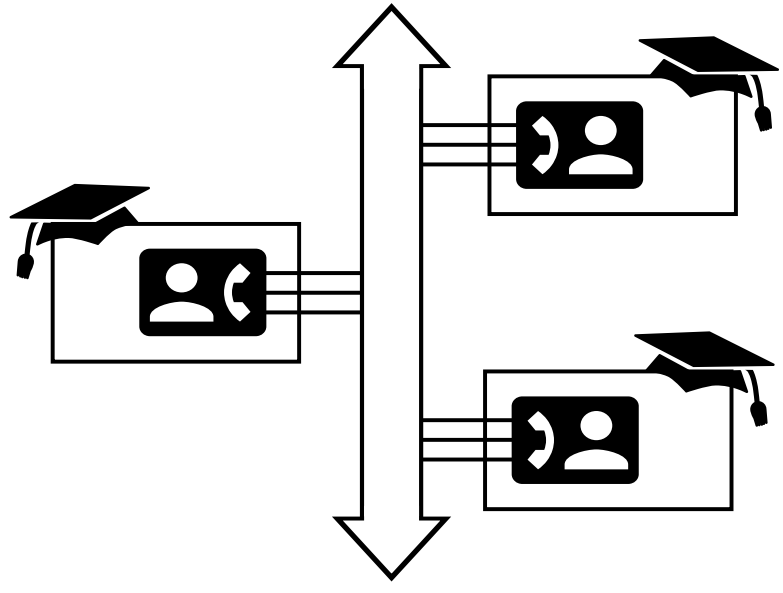


SMART COMPONENTS

Using Session Types to Avoid Late Stage Design Bugs

lenny@cs.stanford.edu


Hardware Design
Type Systems

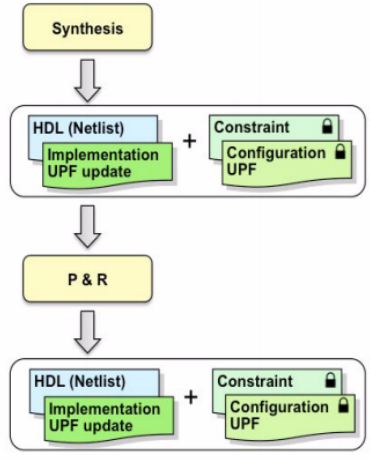
LATE STAGE DESIGN BUGS

Problem: Finding logic issues in gate-level simulations

Goal: Surface these problems earlier (i.e. in RTL)

LOGICAL ERRORS NOT IN RTL?

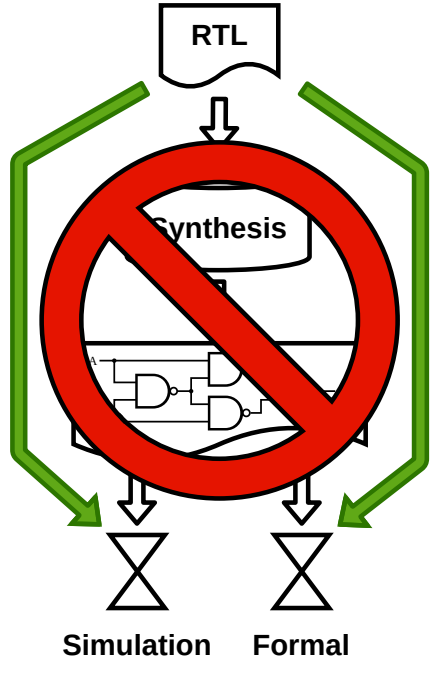
- Some features added after synthesis
  - E.g., power domains
- These features have interactions
  - E.g., BISR and power domains



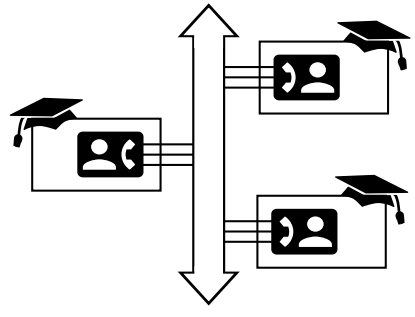
IEEE Standard for Design and Verification of Low-Power, Energy-Aware Electronic Systems

RAISING THE LEVEL OF ABSTRACTION

- Need to verify final netlist to find bugs
  - Lack of abstraction = slow
- **Goal:** surface issues earlier
  - **Abstraction** hides gate-level details
  - **Type system** proves correctness
  - Resolve issues **in RTL**

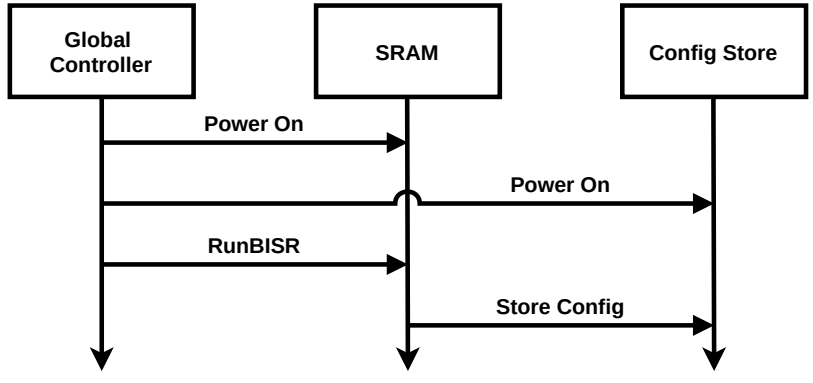


SMART COMPONENTS METHODOLOGY



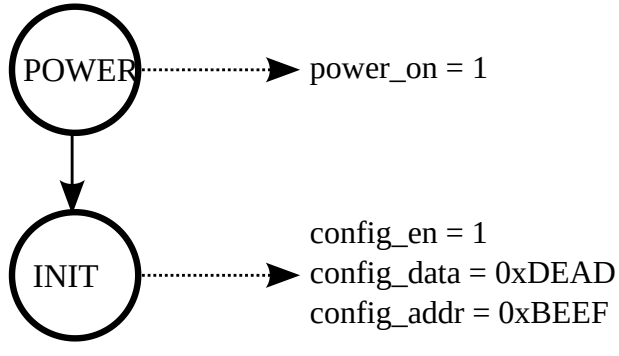
- **Abstract actions** capture component interfaces
- **Unit testing** verifies concrete implementations
- **Session types** verify composition of components

SMART COMPONENT INTERFACES



- RTL interfaces specify component behavior
- Abstraction avoids tech-specific details
- Example actions: power, boot, reset, test, read, write
  - **What are we missing?** Level shifters, DFT?

IP BLOCKS PROVIDE CONCRETE IMPLEMENTATION



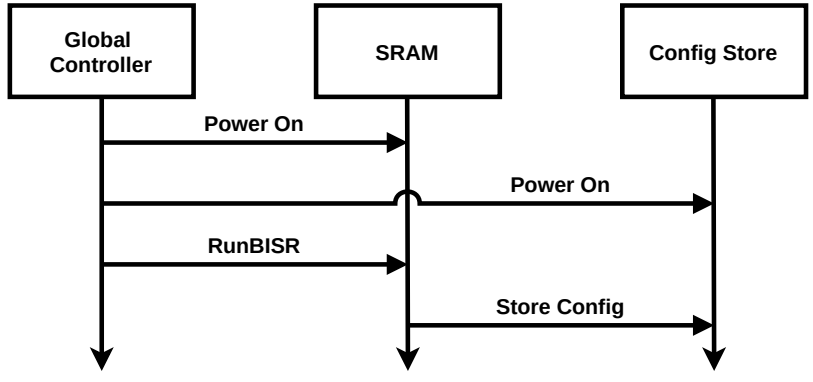
- Modular design facilitates technology changes
- Abstraction facilitates unit level testing

UNIT-LEVEL VERIFICATION

```
tester.send(memory, POWER_ON)
tester.send(memory, BOOT)
config = tester.receive(memory)
tester.write(memory, addr=0xDEAD, data=0xBEEF)
tester.expect_read(memory, addr=0xDEAD, data=0xBEEF)
tester.send(memory, POWER_OFF)
tester.send(memory, POWER_ON)
tester.send(memory, config)
```

- Write tests in terms of abstract actions
- Avoids interaction of concrete components
  - Use abstract implementations of external interfaces and leverage compositional verification

SPECIFYING COMPONENT INTERFACES



- **Design Goals**
  - **Extensible** for user defined actions and protocols
  - Facilitate **unit-level verification** approaches
  - Formally **verify composition** of components
- **Our Choice:** Behavioral type system (session types)

WHY USE A TYPE SYSTEM?

- Types enable **formal proofs** of program properties

$$\text{if } e : \tau \text{ and } e \mapsto e' \text{ then } e' : \tau \text{ (preservation)}$$

- Many useful **typing disciplines** from software
- Magma facilitates developing **new hardware types**
  - Opportunity for **formal methods** (types $\equiv$ logic)

HARDWARE TYPE SYSTEMS

- Basic hardware type systems capture structure
- Advance types capture more semantics
  - E.g., algebraic data types, session types

BASIC HARDWARE TYPE SYSTEM

```
io = m.IO(PCLK=m.Out(m.Clock), PRESETn=m.Out(m.Reset),
          PADDR=m.Out(m.Bits[addr_width]),
          PPROT=m.Out(m.Bit), PENABLE=m.Out(m.Bit),
          ...)
```

- Types capture structure
  - Guarantee matching port names and type
  - **Do not guarantee correct usage of port**

PRODUCT TYPES

```
class CacheReq(m.Product):
    addr = m.UInt[n]
    data = m.UInt[n]
    mask = m.UInt[n // 8]
```

- Abstract fields using names rather than bit ranges
- Ensures consistent interpretation of underlying bits

SUM TYPES (ALA PEAK)

```
if inst[isa.OP].match:
    ...
elif inst[isa.OP_IMM].match:
    ...
```

- A set of bits can have multiple interpretations
- Ensure each variant is handled
- Abstract representation of variants (tag bits)

BEYOND ALGEBRAIC DATA TYPES

- ADTs capture the semantic interpretation of a wire
- Smart components needs to extend this **over time**
  - I.e., interpretation changes throughout a protocol

LOOKING TO SOFTWARE TYPE SYSTEMS

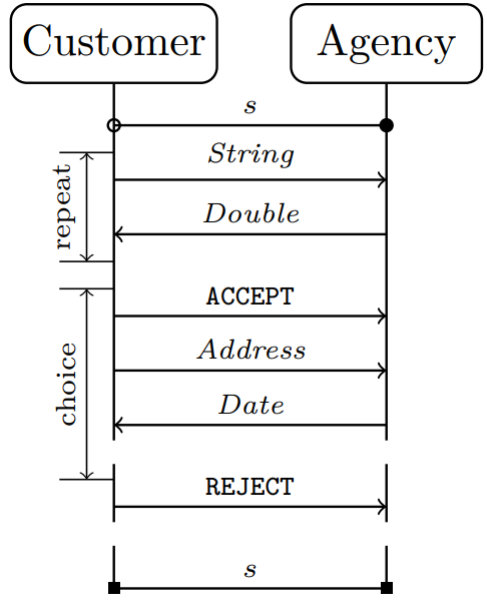
- Integral to certified software (e.g. coq)
- Static systems avoid runtime overhead
- Existing body of work on capturing behavior **in time**

SESSION TYPES

- Capture sequential behavior such as communication protocols

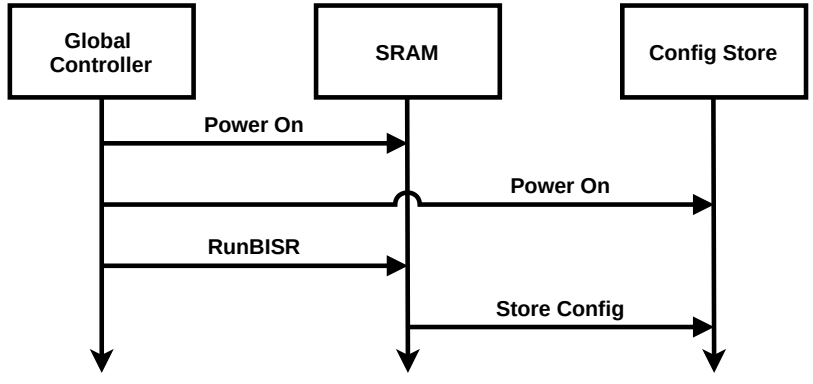
$$T = \bigoplus \left\{ \begin{array}{lcl} \texttt{QUERY} & : & !String.?Double.T \\ \texttt{ACCEPT} & : & !String.?Date.\texttt{end} \\ \texttt{REJECT} & : & \texttt{end} \end{array} \right\}$$

- Statically verify implementations using syntax analysis



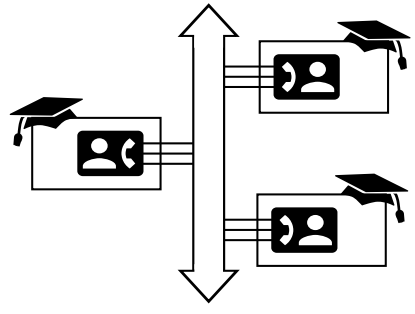
D. Ancona et al.. Behavioral Types in Programming Languages

SESSION TYPES FOR HARDWARE




```
SRAMInit = [
    # SRAM expects someone to power it on
    Receive[Command.PowerOn],
    # SRAM expects someone to tell it to run self repair
    Receive[Command.RunBISR],
    # SRAM expects someone to store its redundancy config
    Write[ConfigData]
]
```

SESSION TYPES VERIFY COMPOSITION

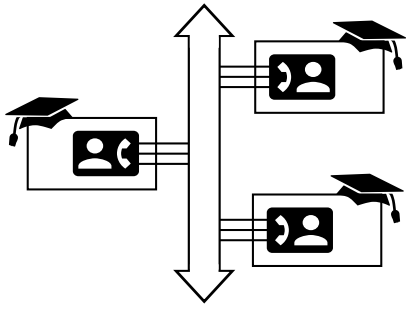


- For a session type to be satisfied, some other component must provide the **dual** action
  - E.g., each send must be satisfied by a receive
- Type checker reveals problems in RTL
  - E.g., SRAM is not configured after power on

TECHNICAL CHALLENGES

- Promoting modularity (avoiding global specs)
  - Can we do this while detecting deadlocks?
- Handling parallelism and concurrency (interrupts)
  - Software exception theory
- Statically verifying implementations
  - Syntax analysis on coroutines

SMART COMPONENTS SUMMARY



Scot Hampton

- **Abstract actions** capture component interfaces
- **Unit testing** verifies concrete implementations
- **Session types** verify composition of components