Avoiding Late Stage Design Bugs using Session Types to Specify Abstract Component Interfaces

Lenny Truong — 4/1/2021 — lenny@cs.stanford.edu

Motivation

- Important features are added late in the design process
    - Testing (BIST), power domains, redundancy
- Two key classes of bugs related to these features
    - Unexpected interactions (e.g. power domains and configuration state)
    - Incorrect use of interfaces (e.g. integrating specific IP block)
- Verification and debugging done with gate level simulations (slow)
- **Challenge:** How can we prevent these bugs from arising late in the design process without breaking causality or requiring the designer to know all the details that will arise at the end of the design?

Memory Example

- Early in the design process, a simple SRAM model is used to represent memory
- Later, it is replaced with a concrete SRAM that adds test and config interfaces
    - Test interface used to find bad columns
    - Config interface stores bad column information (must load before using)
    - Also has power states (shutting off power means we have to reload config)
- **Problem 1:** Integrating the concrete SRAM requires cross cutting design changes
    - e.g., global communication resource needed to drive the interfaces
- **Problem 2:** The SRAM interface specification is incomplete
    - Expensive simulation cycles used to discover desired configuration

Proposal

Use the concept of *abstract actions* to surface the interactions required to use a component (e.g., test, configuration, power) early in the design process

- IP blocks presents an implementation of each action (performing a state change)
- Requires the designer to consider the interacting issues early (e.g., performing the actions requires some global communication resource) without knowing the specific technology used for the components

Issue: Defining a generic set of actions may not be complete

- Power states, boot/restart, configuration, testing, ...???

Methodology

Goal: Capture the required interaction with a component using a type

- Must go beyond structural type checking (i.e. ports are connected properly)
    - A component may be correctly connected to the global communication resource, but the global controller may not perform the required actions

Approach: Structure sequences of actions using a *session type* to ensure that components interact in a type-safe manner

- A user of a component must provide logic to perform the required abstract actions
- A component must provide logic to lower abstract actions into concrete actions
    - Concrete actions correspond to internal state changes

A Brief Introduction to Session Types

Adapted from

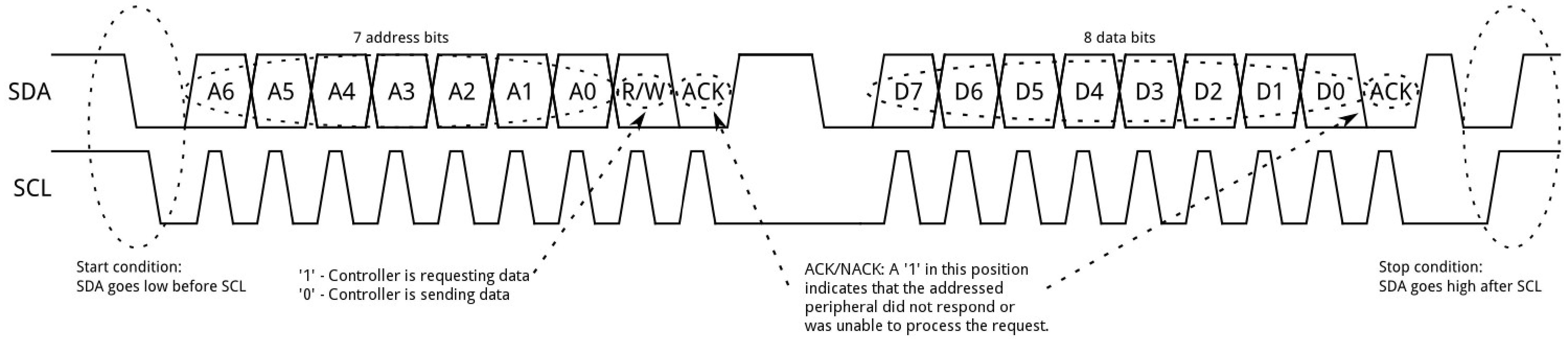
- https://stanford-cs242.github.io/f18/lectures/07-2-session-types.html
- https://munksgaard.me/papers/laumann-munksgaard-larsen.pdf

Session Types Overview

Goal - Structure sequences of reciprocal interactions in a type-safe manner

- A *session* has an associated channel through which all interactions take place and the interactions are modelled by a type -- called a *session type*
- The type system ensures that two processes only communicate via a session if their session types are compatible.

I2C Example


7 address bits
8 data bits
SDA
SCL
A6
A5
A4
A3
A2
A1
A0
R/W
ACK
D7
D6
D5
D4
D3
D2
D1
D0
ACK
Start condition:
SDA goes low before SCL
'1' - Controller is requesting data
'0' - Controller is sending data
ACK/NACK: A '1' in this position indicates that the addressed peripheral did not respond or was unable to process the request.
Stop condition:
SDA goes high after SCL

- Start condition (controller indicates transimission is about to start)
- Address frame (controller chooses peripheral to talk to)
- One or more data frames (8-bit messages)
  - Data flows from controller to peripheral (write) or vice versa (read)
- Stop condition

Session Type Definitions

$$\begin{array}{rcll}
\textsf{SessionType}\ \sigma & ::= & \textsf{recv}\ \tau;\ \sigma & \text{receive message type } \tau \\
& \mid & \textsf{send}\ \tau;\ \sigma & \text{send message type } \tau \\
& \mid & \textsf{choose}\ \{L_0\colon\ (\sigma_{L_0}) \mid L_1\colon\ (\sigma_{L_1}) \mid \ldots\} & \text{choose sub-protocol} \\
& \mid & \textsf{offer}\ \{L_0\colon\ (\sigma_{L_0}) \mid L_1\colon\ (\sigma_{L_1}) \mid \ldots\} & \text{offer sub-protocol} \\
& \mid & \varepsilon & \text{end protocol}
\end{array}$$

Simple I2C Session Type

Only reads/writes one data frame

$$\begin{aligned}
\textsf{I2CStop} &= \textsf{send}\ \texttt{STOP\_COND};\ \varepsilon \\
\textsf{I2CWrite} &= \textsf{send}\ \texttt{Bits[8]};\ \textsf{I2CStop} \\
\textsf{I2CRead} &= \textsf{recv}\ \texttt{Bits[8]};\ \textsf{I2CStop} \\
\textsf{I2CCtrl} &= \textsf{send}\ \texttt{START\_COND};\ \textsf{send}\ \texttt{Bits[7]}; \\
&\quad\ \textsf{choose}\ \{\textsf{read}\colon (\textsf{I2CRead}) \mid \textsf{write}: (\textsf{I2CWrite})\}
\end{aligned}$$

Recursion

$$
\begin{aligned}
\mathsf{I2CWrite} &= \mathsf{send}\ \mathtt{Bits}[8];\ \mathsf{choose}\ \{\mathsf{continue}: (\mathsf{I2CWrite})\ \mid \mathsf{end} : (\varepsilon)\} \\
\mathsf{I2CRead} &= \mathsf{recv}\ \mathtt{Bits}[8];\ \mathsf{choose}\ \{\mathsf{continue}: (\mathsf{I2CRead})\ \mid \mathsf{end} : (\varepsilon)\} \\
\mathsf{I2CCtrl} &= \mathsf{send}\ \mathtt{START\_COND};\ \mathsf{send}\ \mathtt{Bits}[7]; \\
&\quad \mathsf{choose}\ \{\mathsf{read}: (\mathsf{I2CRead})\ \mid \mathsf{write} : (\mathsf{I2CWrite})\}
\end{aligned}
$$

Note: We no longer have an explicit **send STOP_COND**, instead we choose "end"

I2CPeriph

$$\begin{aligned}
\mathsf{I2CWrite} &= \mathsf{recv}\ \mathtt{Bits}[8];\ \mathsf{offer}\ \{\mathsf{continue}\colon (\mathsf{I2CWrite})\ \mid \mathsf{end} : (\varepsilon)\} \\
\mathsf{I2CRead} &= \mathsf{send}\ \mathtt{Bits}[8];\ \mathsf{offer}\ \{\mathsf{continue}\colon (\mathsf{I2CRead})\ \mid \mathsf{end} : (\varepsilon)\} \\
\mathsf{I2CPeriph} &= \mathsf{recv}\ \mathtt{START\_COND};\ \mathsf{recv}\ \mathtt{Bits}[7]; \\
&\quad \mathsf{choose}\ \{\mathsf{read}\colon (\mathsf{I2CRead})\ \mid \mathsf{write} : (\mathsf{I2CWrite})\}
\end{aligned}$$

Duality

$$
\begin{aligned}
\overline{\mathsf{send}\ \tau;\ \sigma} &= \mathsf{recv}\ \tau;\ \overline{\sigma} \\
\overline{\mathsf{recv}\ \tau;\ \sigma} &= \mathsf{send}\ \tau;\ \overline{\sigma} \\
\overline{\mathsf{choose}\ \{L\colon\ (\sigma_L) \mid R\colon\ (\sigma_R)\}} &= \mathsf{offer}\ \{L\colon\ (\overline{\sigma_L}) \mid R\colon\ (\overline{\sigma_R})\} \\
\overline{\mathsf{offer}\ \{L\colon\ (\sigma_L) \mid R\colon\ (\sigma_R)\}} &= \mathsf{choose}\ \{L\colon\ (\overline{\sigma_L}) \mid R\colon\ (\overline{\sigma_R})\} \\
\overline{\varepsilon} &= \varepsilon
\end{aligned}
$$

I2C Dual

$$\mathsf{I2CPeriph} = \overline{\mathsf{I2CCtrl}}$$

How does the type checking work?

Type Checking Primer: A Simple Language

$$
\begin{array}{rcll}
\textsf{Type}\ \tau & ::= & \textsf{int} & \text{integer} \\
 & \mid & \textsf{bool} & \text{boolean} \\
 & \mid & \textsf{unit} & \text{statement type} \\
 & & & \\
\textsf{Expression}\ e & ::= & x & \text{variable} \\
 & \mid & n & \text{integer} \\
 & \mid & b & \text{boolean} \\
 & \mid & e_1 \wedge e_2 & \text{logical and} \\
 & & & \\
\textsf{Statement}\ s & ::= & \textsf{decl}\ x\ \tau & \text{declare variable type} \\
 & \mid & x = e & \text{assignment} \\
 & \mid & s_1;\ s_2 & \text{sequencing} \\
 & \mid & \varepsilon & \text{terminator}
\end{array}
$$

Type Checking Primer: Typing Rules

$$\frac{}{\Gamma \vdash n : \mathsf{int}}\ \text{(T-Int)} \qquad \frac{}{\Gamma \vdash b : \mathsf{bool}}\ \text{(T-Bool)} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{(T-Var)} \qquad \frac{\Gamma \vdash x_1 : \tau \quad \Gamma \vdash x_2 : \tau}{\Gamma \vdash x_1 \wedge x_2 : \tau}\ \text{(T-And)}$$

$$\frac{}{\Gamma \vdash \varepsilon : \mathsf{unit}}\ \text{(T-Epsilon)} \qquad \frac{\Gamma, x : \tau \vdash s : \mathsf{unit}}{\Gamma \vdash \mathsf{decl}\ x\ \tau;\ s : \mathsf{unit}}\ \text{(T-Decl)} \qquad \frac{\Gamma \vdash x : \tau \quad \Gamma \vdash e : \tau \quad \Gamma \vdash s : \mathsf{unit}}{\Gamma \vdash x = e;\ s : \mathsf{unit}}\ \text{(T-Assign)}$$

$\mathsf{decl}\ x\ \mathsf{int};$	(T-Decl)
$x = 2 \wedge 7;$	(T-Int) and (T-And)
$\mathsf{decl}\ y\ \mathsf{bool};$	(T-Decl)
$y = x \wedge \mathsf{False};$	(Fails T-And)
$y = 2 \wedge 7;$	(Fails T-Assign)
ϵ	(T-Epsilon)

Session Type Rules

Basic Idea: Modify the type of the channel in the context as an operation is performed

$$\frac{\Gamma \vdash c : \mathsf{send}\ \tau; \sigma \qquad \Gamma \vdash x : \tau \qquad \Gamma, c : \sigma \vdash s : \mathsf{unit}}{\Gamma \vdash \mathsf{send}(c, x);\ s : \mathsf{unit}} \text{(T-Send)}$$

$$\begin{aligned}
\mathsf{decl}\ c\ (\mathsf{send\ int};\ \mathsf{recv\ int};\ \varepsilon); &\quad (\text{T-Decl},\ c : \mathsf{recv\ int};\ \mathsf{int}; \varepsilon) \\
\mathsf{send}(c, 2); &\quad (\text{T-Send},\ c : \mathsf{recv\ int}; \varepsilon) \\
\mathsf{send}(c, 3); &\quad (\text{Fails, expected recv not send}) \\
\epsilon &
\end{aligned}$$

Code Example

```
I2CStop = Send[STOP_COND, Epsilon]
I2CWrite = Send[BitVector[8], I2CStop]
I2CRead = Receive[BitVector[8], I2CStop]
I2CCtrl = Send[START_COND, Send[BitVector[7], Choose[("read", I2CRead),
                                                     ("write", I2CWrite)]]]

def i2c_controller(c: Channel[I2CCtrl]):
    c.send(START_COND)
    c.send(0xDE)
    c.choose("read")
    result = c.receive()
    c.send(STOP_COND)
    c.close()
```

Adding More Complexity

- Multiparty sessions (more than two entities communicating)
  - Need for bus/NoC protocols
  - Project global type (for all parties) into local type (for one party)
  - Can prevent deadlocks
- Session delegation (one entity hands session over to another entity)
  - One component performs part of the sequence, then finished by another

Applying to a Simple Accumulation Register

Input value is added to an internal register, output value is the current running sum

- Abstract Actions: `PowerOn`, `Boot`
  - If you don't call `PowerOn`, output is `X`
  - If you don't call `Boot`, initial register value is random
  - Calling `Boot` before `PowerOn` is undefined
- Concrete Actions:
  - Initial value is `X` (uninitialized flops)
  - `PowerOn` input bit held high for one cycle sets register to random value
  - `Boot` uses a configuration interface to set initial sum

As a Session Type

$$\mathsf{AccumRegAbstract} = \mathsf{recv}\ \texttt{POWER\_ON};\ \mathsf{recv}\ \texttt{BOOT};\ \varepsilon$$

$$\begin{aligned}
\mathsf{HoldLow} &= \mathsf{recv}\ 0;\ \mathsf{HoldLow} \\
\mathsf{PowerOnConcrete} &= \mathsf{choose}\ \{0\colon (\mathsf{PowerOnConcrete})\ \mid 1 : (\mathsf{HoldLow})\} \\
\mathsf{BootConcrete} &= \mathsf{choose}\ \{(x : \texttt{Bits}[8], 0)\colon (\mathsf{BootConcrete})\ \mid (x : \texttt{Bits}[8], 1) : (\mathsf{HoldLow})\}
\end{aligned}$$

- Using the register requires logic to send the `PowerOn` and `Boot` commands before being able to use the output sum
- Register must provide an adapter to convert from the abstract type to the concrete type for each action

Type Checking Hardware Behavior

- Session types capture the sequential nature of communication
- To apply to structural hardware, we'll need to infer the sequential behavior (FSM)
- Writing a coroutine-style controller would allow standard type checking approaches

Reg Coroutine Controller Example

```python
AccumRegAbstract = Receive[Command.POWER_ON,
                           Receive[Command.BOOT, Epsilon]]
HoldLow = Receive[0, HoldLow]
PowerOnConcrete = Rec("PowerOnConcrete", Choose[(0, "PowerOnConcrete"),
                                                (1, HoldLow)])
x = TypeVar[Bits[8]]
BootConcrete = Rec("BootConcrete", Choose[((x, 0), "BootConcrete"),
                                          ((x, 1), HoldLow)])

class RegController(Controller):
    def __call__(self,
                 abstract: Channel[AccumRegAbstract],
                 power_on: Channel[PowerOnConcrete],
                 boot: Channel[BootConcrete]):

        def wait_for_next_command():
            while ~abstract.receive():
                yield power_on.send(0), boot.send(0)

        yield from wait_for_next_command()
        yield power_on.send(1), boot.send(0)
        yield from wait_for_next_command()
        yield power_on.send(0), boot.send(1)
        while True:
            yield power_on.send(0), boot.send(0)
```

Limitations

- Session types ensure that their is a correct state machine that produces or consumes a required actions sequence
- They ensure that the there is a conversion from abstract and concrete actions
  - Does not ensure that the conversion is correct
- They do not ensure that the behavior of the component is correct after a concrete action (i.e. the correct state change was performed)
  - Could be used to generate assumptions for other verification methods

Conclusion

Goal: Surface the interactions required to use a component (e.g., test, configuration, and power interfaces) early in the design process, without requiring the designer to know which specific IP block they will end up using (abstraction)

Methodology: Specify component interfaces as a sequence of *abstract actions*
Sequences are described as *session types* to ensure that

- Designers provide resources to produce the action sequences
- IP blocks provide logic to lower actions into concrete state changes