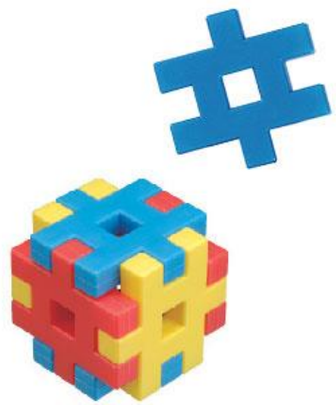
Secure Composition in IoT



Sanjiva Prasad
IIT Delhi

Outline

- Motivation and prescription of IoT programming
- Synchronous Programming as a solution
  - LUSTRE as an answer for IoT
  - Lustre can be compiled to C
  - Lustre provides a simple and efficient verification model
- Extension to Lustre to incorporate security tags and the extended secure flow type-checking
  - Two theorems
    - the security model is correct [wrt **non-interference**]
    - the translation **preserves** security

Lessons from CarShark

- Insecure Public Bus (Open Interface)
  - No security classes, no tags on messages
- Unforeseen Connections between Interfaces
  - UConnect straddled 2 buses, Over-the-air connection
- Decoding of messages (telematics)
- Too much capability: Full Unix, Full network stack
- Reprogrammability: Upgrade and Reboot

Contrast: Security from a High-Level Model

- Security as a basic design element of the model
  - A lattice of security classes; information flow
  - Write-up, Read-down [Bell-LaPadula generalised]
- Secure by construction
  - *Elementary secure blocks*
  - *Composition of blocks is secure*
  - *Explicit and simple interfaces with clear semantics*
- Generate implementation by verified translation
  - Verified compilers or Translation Validation
  - Security properties are maintained in translation

Prescription for Programming IoT

[P. 2015, POPL PLVNet Workshop]

- Reactive Model:
  - Transformational + Interactive + …
  - Environment cannot wait, cannot be ignored!
- Synchronous Streams of Data
  - Timed, i.e., notion of clocks/events
- Declarative (functional or relational)
  - Simple semantics
- Ability to assert (global) properties
- Security metadata tags inherent in all operations:
  - storage, communication, computation

Synchronous Programming

An old solution to a new problem

- View IoT devices as **transducers** of **clocked data streams**
  - Generate sequences of (typed, tagged) values on event-driven clocks
  - Consume sequences
  - Transform tuples of sequences
- Clocked: A clock is an boolean stream (event-driven model)
  - Clocks need not be equal-tempered
  - Clocks can be nested or derived
- Intuitive Dataflow model:
  - *Diagrams of blocks and wires,*
  - *Networks of operators*
  - *Dynamic sampling*
  - *Dynamic behaviour defined by equations*
  - *Time dependencies*
- Standard APIs and clear semantics
- Expressive!
  - Automatic control
  - Systolic Algorithms
  - Hardware Circuits
  - …

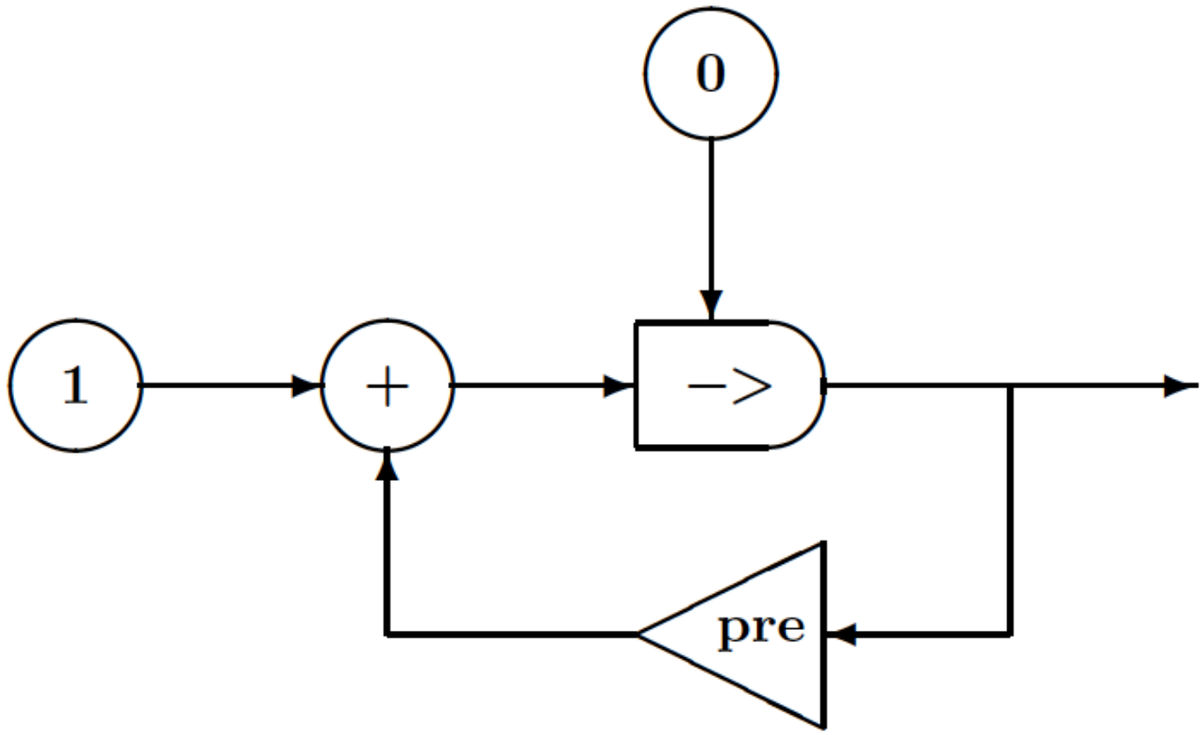
Lustre as a DSL for IoT

Lustre: A successful Synchronous Programming Language

- Declarative Semantics
  - Streams and Relationships between streams given in an equational style
- Deterministic
  - Clear functional semantics
- Multi-clocked
  - Multiple versions of time, simultaneously
  - Nested clocks
  - sampling
- Has an efficient implementation
  - Verified & Certified compilation into C [Leroy et al PLDI 2017]
  - Simulation Environment LURETTE for Test Generation [Halbwachs 1999]
- Can express *assertions* of properties that must hold
- Able to Model Check:
  - Model-checkers: LESAR, KIND
  - Model-checking Safety Properties is EASY!

Lustre Programs as Dataflow Network Diagrams

Example: X = 0 **->** (**pre** X) + 1

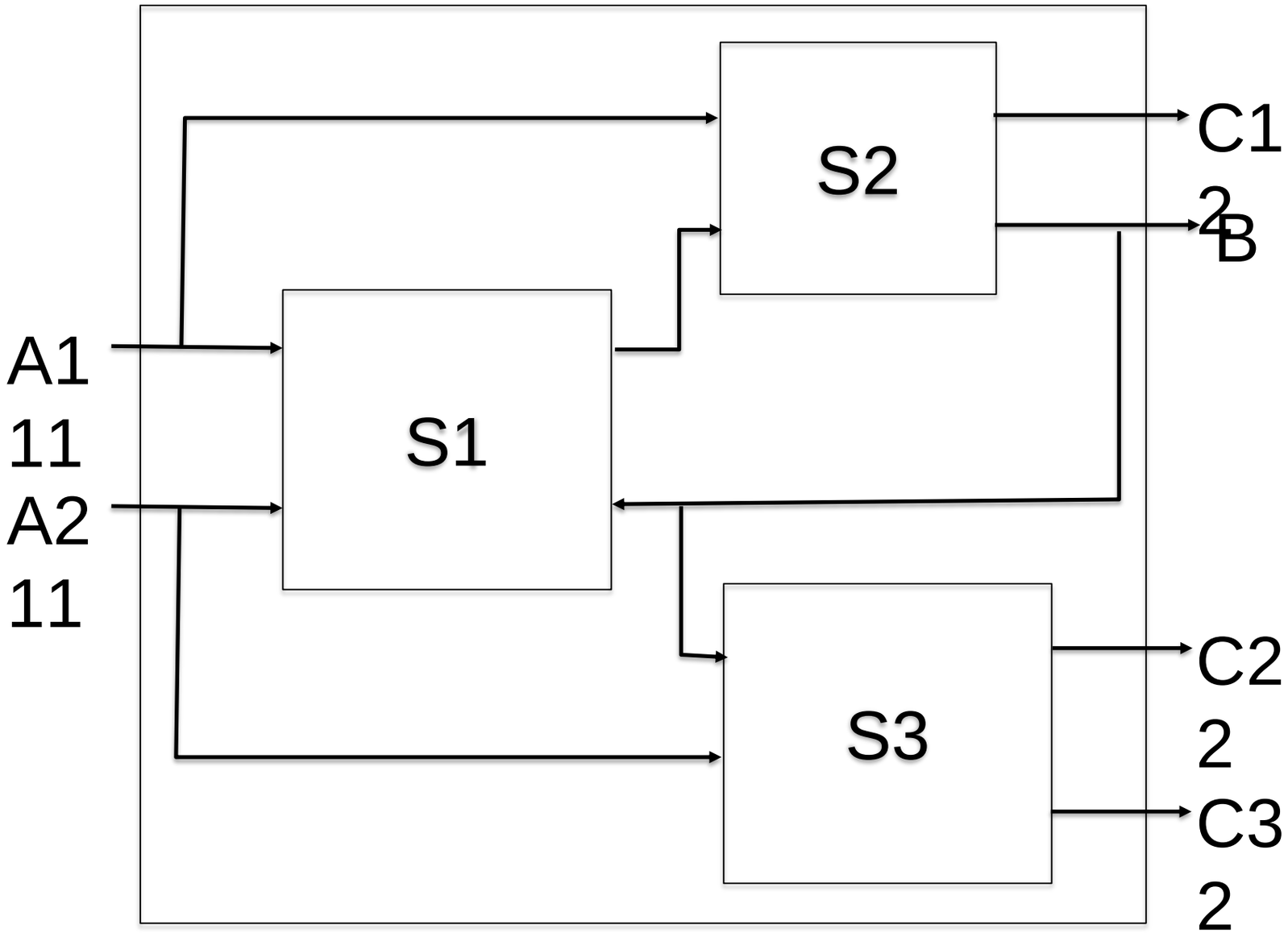


X = [0, 1, 2, …]

Composition: Network Block Diagrams

Nodes may have multiple input and output streams and local streams

- on possibly different clocks
- may have feedback loops



Syntax

Expressions (E on C)

E ::= X (variable — input or defined)

| **c** (constant — integer, boolean, ⋯)

| op(E_1, ⋯, E_n) (stream operations, all streams on the same clock)

| **case** E_0: **[** E_1 **|** ⋯ **|** E_n **]** (conditional)

| **pre** E_1 (previous)

| E_1 **->** E_2 (initialise to E_1 then E_2)

| E_1 **when** C (sample on clock C) in general: **filter**(E_1, C)

| **current** E (interpolate on oversampling) in general: **project**(E_1, C)

| N(E_1, ⋯, E_n) (instantiate a node)

| **ck**(E_1) (clock of E_1)

Equations

X = E

Node Definitions

node N(X_i:T_i **on** C_i)$_{i=1..m}$ **returns** (Z_j:T_j **on** C_j)$_{j=m+1..n}$ **where** { Y_k = E_k }$_{k=m+1..n+p}$

Lustre Semantics

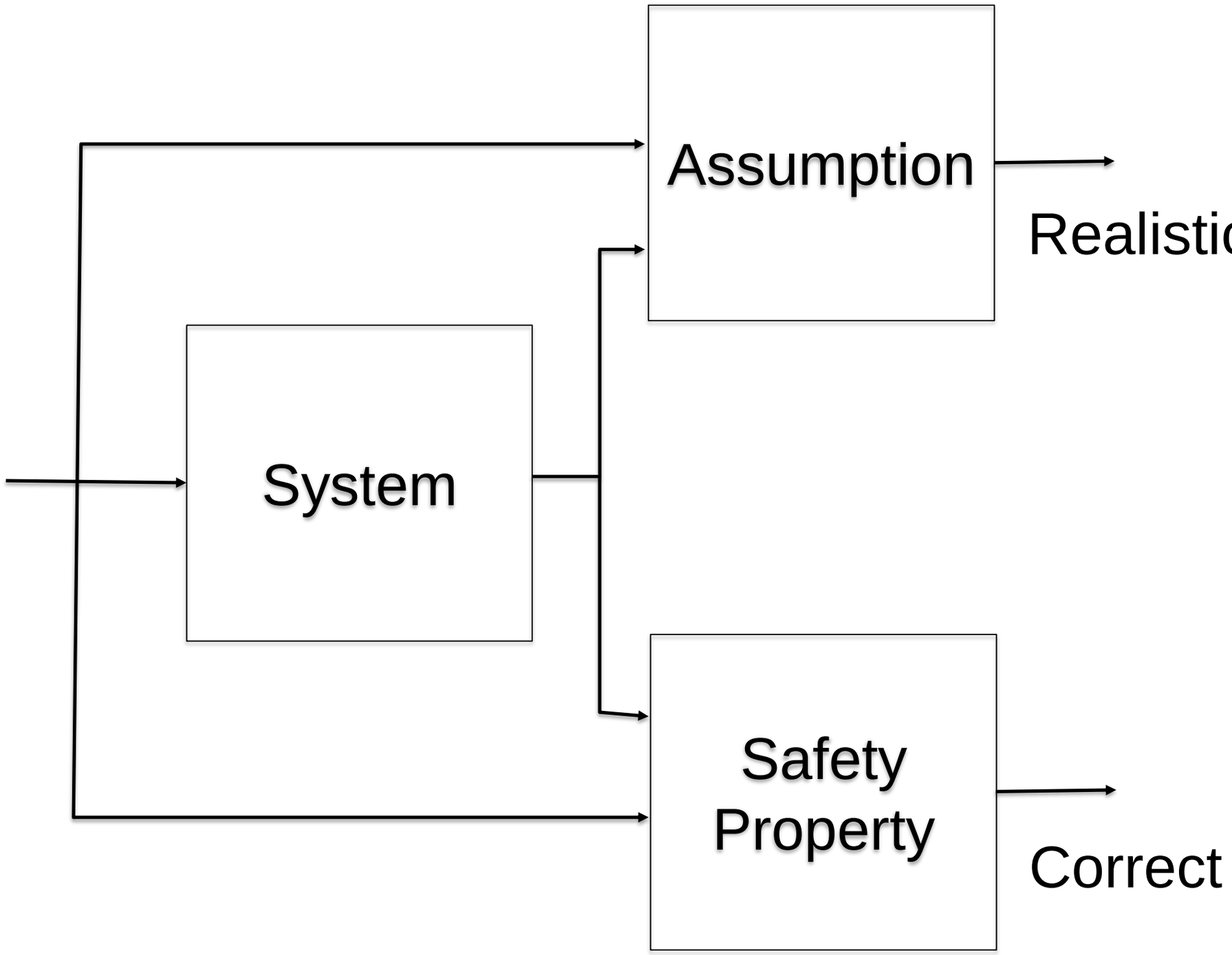
Clocked Streams

E =(	e_1	e_2	e_3	e_4	e_5	$\ldots$)
C =(	tt	ff	tt	tt	ff	$\ldots$)
X=E when C =(	$x_1 = e_1$		$x_2 = e_3$	$x_3 = e_4$		$\ldots$)
time of E, C:	1	2	3	4	5	$\longrightarrow$

Temporal Operators

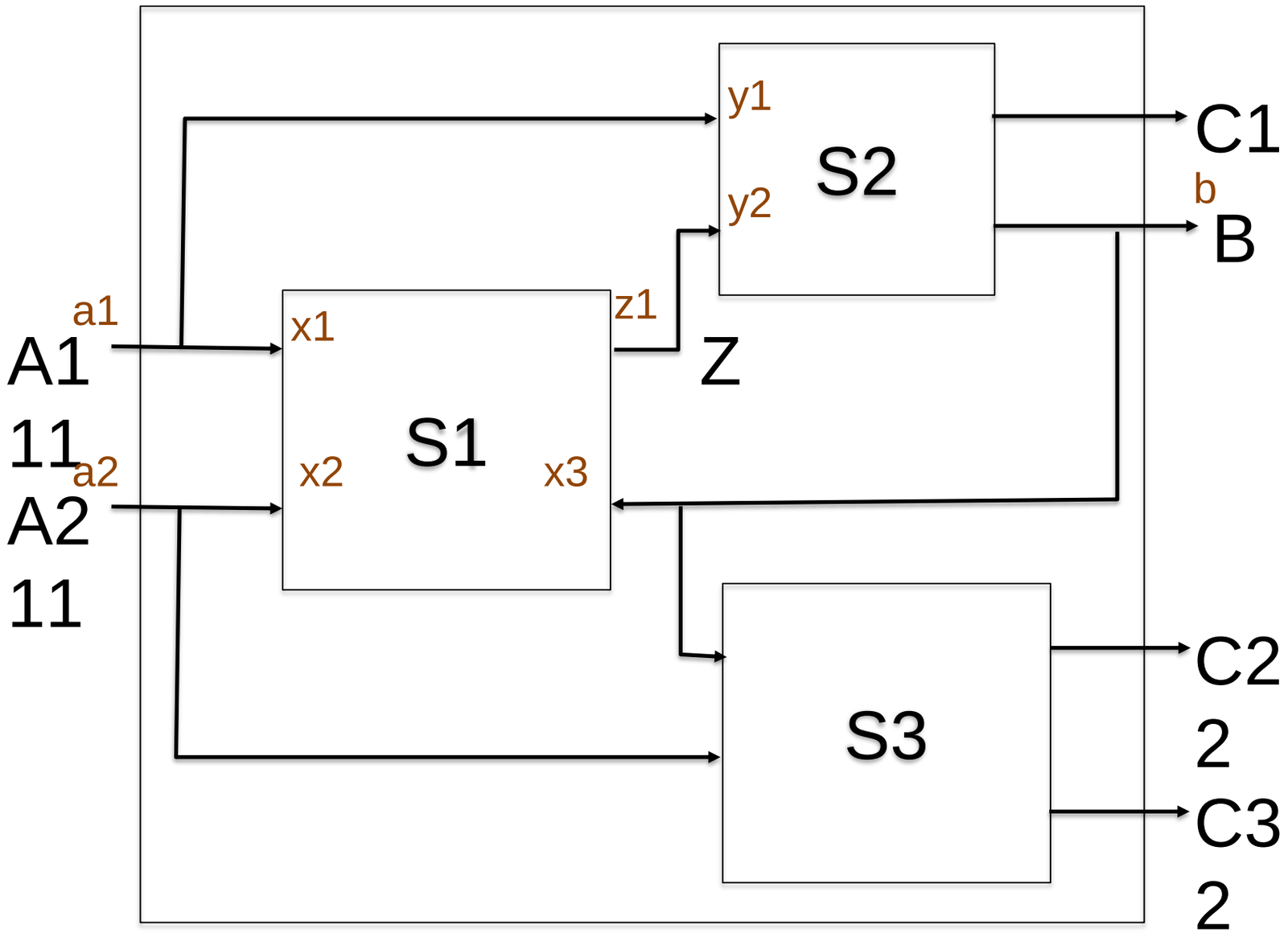
E =(	e_1	e_2	e_3	e_4	e_5	$\ldots$)
C =(	tt	ff	tt	tt	ff	$\ldots$)
X = E when C =(	e_1		e_3	e_4		$\ldots$)
Y = 1 when C -> X =(	1		e_3	e_4		$\ldots$)
Z = current Y =(	1	1	e_3	e_4	e_4	$\ldots$)

Verifying Safety Properties in Lustre



Securing Network Block Diagrams

Input stream security level <= level expected by the Node: a1 <= x1
Output stream security level >= levels of input streams it may read:
z1 >= x1, z1 >= x2, z1 > = x3



New: Evolving Security Classes for Lustre

Denning's Security Lattices [1976]

$$(\mathcal{L}, \leq, \oplus, \bot, \otimes, \top)$$

Clocked streams of security levels

- Partial functions from clock ticks to security levels
- Monotone non-decreasing sequences of security levels
- Intuitive notions of *ordering* and *upper bounds* on these *sequences*
- some auxiliary operators on clocked security-level streams for the temporal operators

Typing Rules on Expressions, Equations and Nodes

- Express the minimal requirements of secure flow from input to output
- Account for both explicit and implicit flows of information (conditionals)
- Ensure that any "taint" from earlier times is carried forward
- Typing rules for (mutually recursive) equations ensures that the outputs have security class no lower than those of the inputs from which explicit or implicit flows may have occurred.

A Secure Flow Framework for Lustre – 1

Simple Security of Expressions

An expression's security level is at least the upper bound of the security levels of all its inputs *{Read only from lower.}*

Equations

X = E

Assuming X has security level *l*, E has the same level *l*.

In general, simple security for systems of mutually recursive equations { $Y_k = E_k$ }

A Secure Flow Framework for Lustre – 2

Confinement Property in Node Definitions

In a node definition: No observable output is given a *lower* security level than the join (upper bound) of the levels of the input streams.

(Note: Purely local streams may be given lower security levels)

Node Instantiation Rule: Secure Composition

Assume a Node Definition is secure (in the confinement sense). Then check that the actual argument input streams have at most the assumed security level. Can guarantee that the outputs have at most the security levels given by the Node Definition rules.

Results – 1

Non-interference Theorem

Suppose a Lustre program is deemed flow secure by the rules proposed by us.

Then, if we change any values on an input stream at a security level *h'*,

we will *not* observe any difference in the output on any stream at security level *l'* such that *not (l' >= h')*.

That is, the *only* outputs that will be affected due to changing inputs are those at security levels to which the input value can legitimately flow. {Write only above, read only from below}

Results – 2

Theorem: Secure Flow Preservation under Compilation

Suppose a Lustre program is deemed flow secure by the rules proposed by us.

Then, when it is compiled into a C program (by the [certified] secure-flow preserving compiler), then the corresponding C program is secure according to Denning's flow security rules.

This result relies on the following facts:

- Data flow equations enforce temporal dependencies
- The security class of an expression is at least that of its clock
- Stream security classes are temporally monotone non-decreasing
- The security typing rules capture all possible explicit and implicit information flows

Thank You