

About the speaker. . .

- PhD dissertation about extensions to **Unity** (communication, true concurrency)
- **Composition** considered but not used to define those extensions (investigation fo works by Collette, Udink, Rao, . . .)
- PostDoc with **Mani Chandy** working on **composition**
- Application to (**our**) compositional techniques to **Unity**
- **Current work** on **composition** (mostly) independent from **Unity**
- Switch from **Unity** to **TLA⁺** in **teaching**

Compositional Reasoning in Unity-like Formalisms

Michel Charpentier
University of New Hampshire

- Compositional Reasoning
 - bottom-up reasoning
 - top-down reasoning and other issues
- Introduction to Unity
 - language, logic and proof system
 - Unity's *Substitution Axiom* (SA)
- Composition in Unity
 - transition-based versus behavior based specifications
 - *Strongest Invariant* (SI) and reachable states
- Composition in TLA
- Composition in general
 - high-level, compositional specifications and component reuse
 - our predicate transformer approach
 - application to Unity: $inv_{\mathcal{U}}$ specifications
- To compose, or not to compose?

“Compositionality”

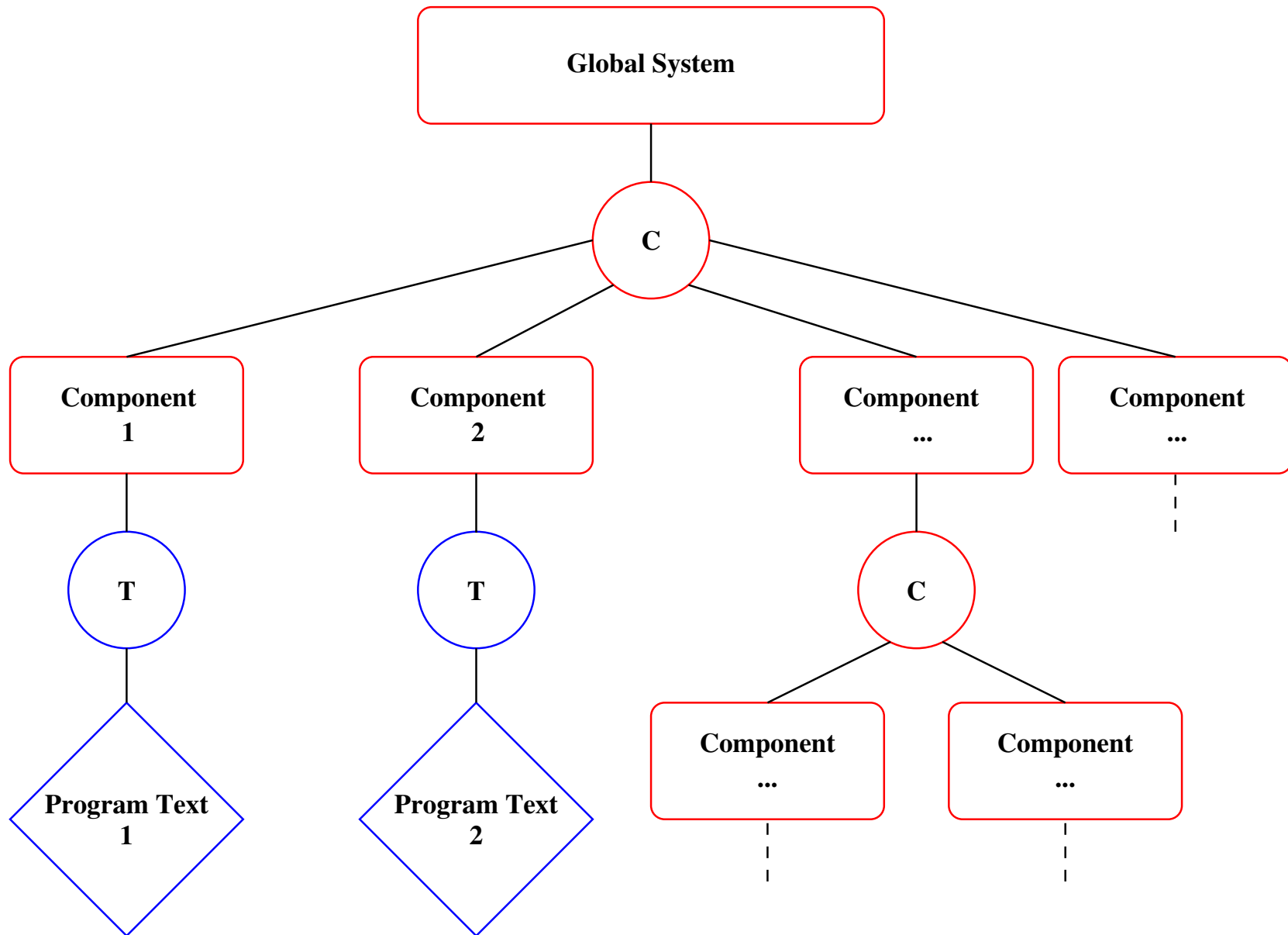
“Under **compositionality**, we include any method by which the **properties** of a system can be **inferred** from **properties** of its constituents, without additional information about the **internal structure** of these constituents.”

Compositionality: The Significant Difference

COMPOS'97

- **property**, **specification**, **property** of a **specification**
- most approaches are **proof-theoretic**
- **top-down** steps?
- abstraction? information hiding? **reuse**?

Compositional Design



Introduction to Unity

- Mani Chandy and Jay Misra (1988), Jay Misra (1995)
- Numerous variants and extensions (composition, communication, refinement, real-time, probabilities, mechanization, ...)
- Based on fair transition systems
- 3 parts:
 - a (model) language for models/programs/algorithms
 - a logic for properties/specifications
 - a proof theory for correctness of programs w.r.t. specifications
- rules for composition
- an infamous Substitution Axiom

Unity language

- syntactic sugar for **transition systems**
- simple **fairness** assumption (uniform weak fairness)
- infinite **computations**, or **behaviors**
- implicit **stuttering**
- simple **datatypes** (booleans, integers, queues, sets, arrays, ...)
- example:

```
Program      F
Declare      x, y, z : integer
Initially    x = 0  $\wedge$  y = 0
Assign       x := x + y
              $\square$       x := x - 1 if x > 0
              $\square$       y := y + 1 if y < 10
              $\square$       z := z + 1 if y = 10
```

$(x, y, z) : \langle (0, 0, -2), (0, 1, -2), (0, 2, -2), (2, 2, -2), (1, 2, -2), (1, 2, -2), (3, 2, -2), \dots \rangle$

Unity logic (\approx Misra, 1995)

- Safety (prevent something bad):
 - p next q : if p is true in one state, q is true in the next state
 - p unless q : if p becomes true, it remains true at least until q becomes true ($p \wedge \neg q$ next $p \vee q$)
 - $\text{stable } p$: if p becomes true, it remains true (p next p or p unless false).
 - $\text{inv } p$: p is true in all states (p is true initially and $\text{stable } p$)
- Liveness (ensure something good):
 - $\text{transient } p$: there is a system transition from p to $\neg p$
 - $p \rightsquigarrow q$: any state that satisfies p is eventually followed by a state that satisfies q
- Mixed specifications (conjunctions of safety and liveness):
 ensures , detects , FP , ...

Unity logic: examples

```
Program       $F$ 
Declare       $x, y, z : \text{integer}$ 
Initially     $x = 0 \wedge y = 0$ 
Assign        $x := x + y$ 
             $\square$             $x := x - 1$  if  $x > 0$ 
             $\square$             $y := y + 1$  if  $y < 10$ 
             $\square$             $z := z + 1$  if  $y = 10$ 
```

$x = 0$ next $x \geq 0$

$\forall k : z = k$ unless $y = 10$

stable $y = 10$

inv $x \geq 0$

transient $x = 1$

true $\rightsquigarrow z > 100$

request = true unless access = true

inv $\forall i, j : \text{excl}(i) \wedge \text{excl}(j) \Rightarrow i = j$

stable $\forall k : \text{count} \geq k$

request = true \rightsquigarrow access = true

$\forall i, j, m : \text{send}(i, j, m) \rightsquigarrow \text{receive}(j, m)$

(inv flag \Rightarrow terminated) \wedge (terminated \rightsquigarrow flag)

Unity proof system

- Basic rules from programs to specifications:

- next:
$$\frac{\forall \tau \text{ transition of } F : \{p\}\tau\{q\}}{p \text{ next } q}$$

- transient:
$$\frac{\exists \tau \text{ transition of } F : \{p\}\tau\{\neg p\}}{\text{transient } p}$$

- ...

- Inference rules from specifications to specifications:

- Conjunction:
$$\frac{\text{inv } p, \text{ inv } q}{\text{inv } p \wedge q}$$

- Progress-Safety-Progress (PSP):
$$\frac{p \rightsquigarrow q, r \text{ unless } s}{p \wedge r \rightsquigarrow (q \wedge r) \vee s}$$

- ...

- A Substitution Axiom

Unity's Substitution Axiom

“If $x = y$ is an **invariant** of a program F , x can replace y in all properties of F . This is a generalization of Leibniz's rule for substitution of equals. A particularly useful form of this axiom is to replace **true** by any **invariant** I , and vice versa.”

- \mathcal{T} : specifications proved **without** the **Substitution Axiom**
- \mathcal{O} : specifications proved **with** the **Substitution Axiom**

$$\frac{\text{inv}_{\mathcal{T}} p}{\text{inv}_{\mathcal{O}} p} \quad \frac{\text{inv}_{\mathcal{T}} I \wedge p}{\text{inv}_{\mathcal{O}} p} \quad \boxed{\text{inv}_{\mathcal{T}} \neq \text{inv}_{\mathcal{O}}}$$

in Unity (1988,1995), only **one** name (invariant): **unsoundness**

Unity's Substitution Axiom (Cont'd)

- \mathcal{T} : properties of atomic transitions

$$p \text{ next}_{\mathcal{T}} q \equiv \forall \tau \text{ transition of } F : \{p\} \tau \{q\}$$

- \mathcal{O} : properties of infinite behaviors

$$p \text{ next}_{\mathcal{O}} q \equiv \forall \sigma \text{ behavior of } F : \forall i : p.\sigma_i \Rightarrow q.\sigma_{i+1}$$

Example:

$$\text{inv}_{\mathcal{T}} x \geq 0 \wedge y \geq 0$$

$$\text{inv}_{\mathcal{O}} x \geq 0$$

$$\text{inv}_{\mathcal{T}} x \geq 0$$

Why consider both \mathcal{T} and \mathcal{O} specifications?

- \mathcal{T} specifications are a **tool** to prove \mathcal{O} specifications
- \mathcal{T} specifications can be **composed**, \mathcal{O} specifications **cannot**

Parallel composition in Unity

\parallel is called **union** and denoted by \parallel

$F \parallel G$ defined by:

- **Union of variables** (no renaming mechanism, no hiding mechanism)
- **Conjunction of initial conditions**
- **Union of transitions**

Composition theorems:

$$\begin{array}{lll} p \text{ next}_{\mathcal{T}} q \text{ in } F \wedge p \text{ next}_{\mathcal{T}} q \text{ in } G & \Rightarrow & p \text{ next}_{\mathcal{T}} q \text{ in } F \parallel G \\ \text{inv}_{\mathcal{T}} p \text{ in } F \wedge \text{inv}_{\mathcal{T}} p \text{ in } G & \Rightarrow & \text{inv}_{\mathcal{T}} p \text{ in } F \parallel G \\ \text{inv}_{\mathcal{T}} p \text{ in } F \wedge \text{stable}_{\mathcal{T}} p \text{ in } G & \Rightarrow & \text{inv}_{\mathcal{T}} p \text{ in } F \parallel G \\ \text{transient } p \text{ in } F & \Rightarrow & \text{transient } p \text{ in } F \parallel G \end{array}$$

No such theorems for $\text{next}_{\mathcal{O}}$, $\text{inv}_{\mathcal{O}}$, \rightsquigarrow , \dots

\mathcal{O} specifications do not compose

Program F
Declare $x, y, z : \text{integer}$
Initially $x = 0 \wedge y = 0$
Assign $x := x + y$
 \parallel $x := x - 1$ if $x > 0$
 \parallel $y := y + 1$ if $y < 10$
 \parallel $z := z + 1$ if $y = 10$

Program G
Declare $x, y : \text{integer}$
Initially $x = 0$
Assign $x := x + 1$
 \parallel $y := y - 1$ if $x > 0$

$\text{inv}_{\mathcal{O}} x \geq 0$ in F

$\text{inv}_{\mathcal{O}} x \geq 0$ in G

but NOT $\text{inv}_{\mathcal{O}} x \geq 0$ in $F \parallel G$

Strongest Invariant (SI): conjunction of all invariants

- $\text{inv}_{\mathcal{O}} p \equiv [\text{SI} \Rightarrow p]$
- $p \text{ next}_{\mathcal{O}} q \equiv p \wedge \text{SI next}_{\mathcal{T}} q$
- **SI**: reachable states (not preserved by composition)

TLA (Temporal Logic of Actions) (Leslie Lamport)

- Programs and specifications are logical formulas of the same language
- Satisfaction is logical implication

$$\begin{aligned} I &\triangleq x = 0 \\ S_1 &\triangleq x' = x + 1 \wedge y' = y \\ S_2 &\triangleq x > 0 \wedge x' = x \wedge y' = y - 1 \\ N &\triangleq S_1 \vee S_2 \\ \text{Fairness} &\triangleq \text{WF}_{\langle x, y \rangle}(S_1) \wedge \text{WF}_{\langle x, y \rangle}(S_2) \\ G &\triangleq I \wedge \Box[N]_{\langle x, y \rangle} \wedge \text{Fairness} \end{aligned}$$

$$G \Rightarrow \Box(x \geq 0)$$

$$\neg G \vee \Box(x \geq 0)$$

$$G \Rightarrow y \in \mathbb{Z} \rightsquigarrow y < 0$$

$$G \Rightarrow \Box(x > 0 \Rightarrow \Box(x > 0))$$

$$G \Rightarrow \Box[x > 0 \Rightarrow x' > 0]_{\langle x, y \rangle}$$

more formal, but closer to Unity than it looks ...

Composition in TLA

Parallel composition is conjunction
(if components are written as open systems)

$$\begin{aligned} I_1 &\triangleq x = 0 \\ S_1 &\triangleq x' = x + 1 \wedge y' = y \\ G_1 &\triangleq I_1 \wedge \Box[S_1]_{\langle x \rangle} \wedge \text{WF}_{\langle x \rangle}(S_1) \\ \\ I_2 &\triangleq \text{true} \\ S_2 &\triangleq x > 0 \wedge x' = x \wedge y' = y - 1 \\ G_2 &\triangleq I_2 \wedge \Box[S_2]_{\langle y \rangle} \wedge \text{WF}_{\langle y \rangle}(S_2) \end{aligned}$$

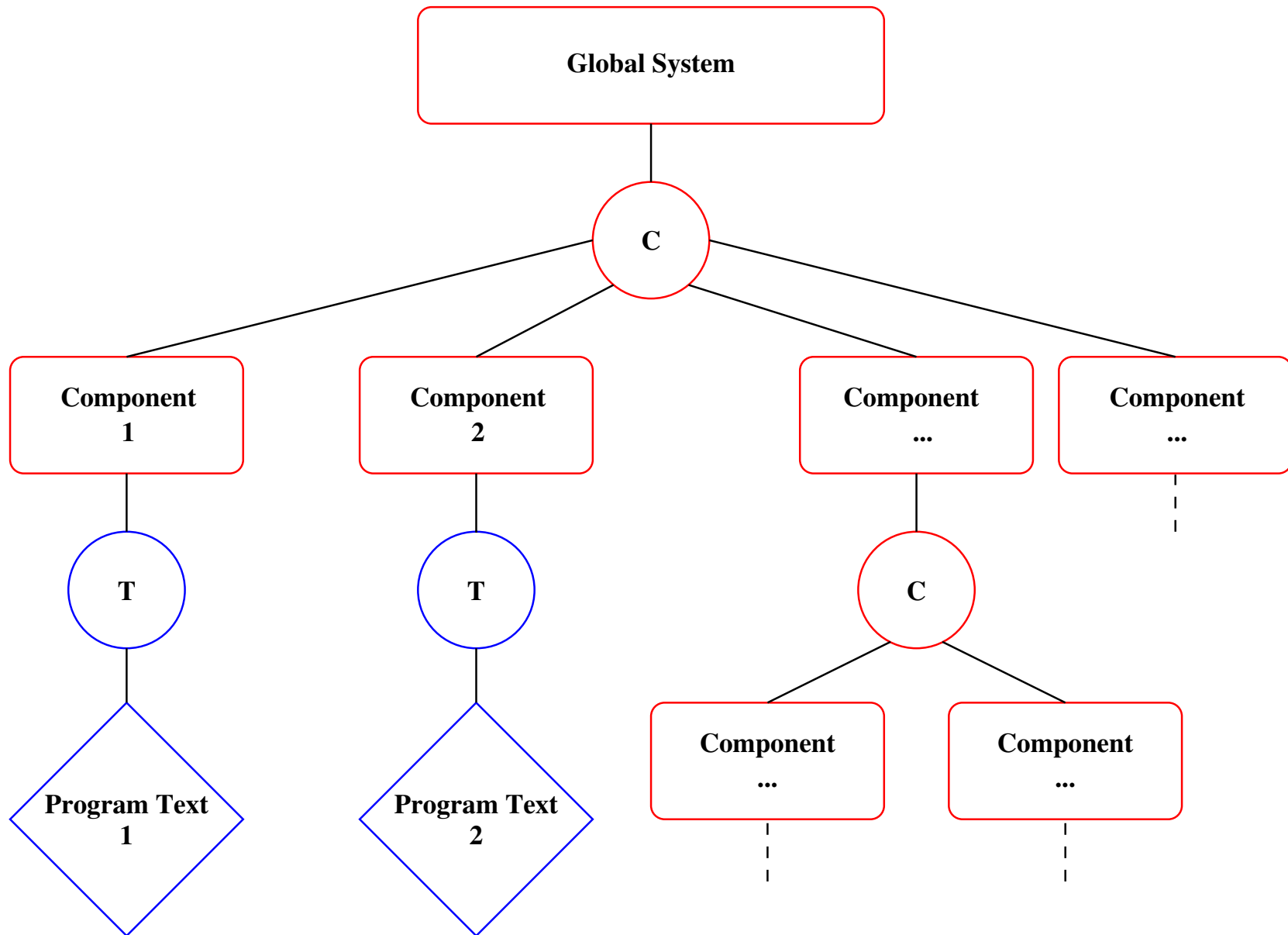
$$G \equiv G_1 \wedge G_2 \quad \text{(by calculation)}$$

Consequence for composition:

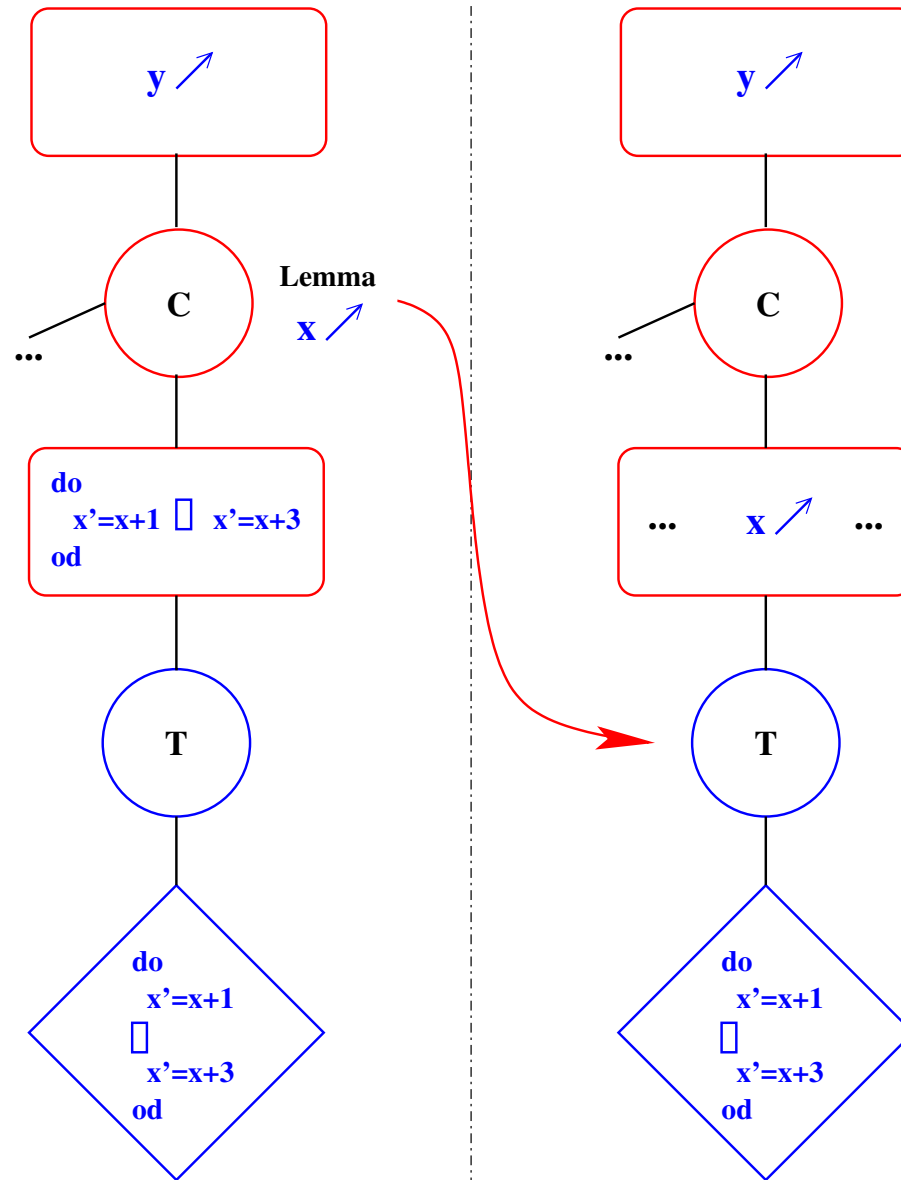
$$\text{Spec in } F \Rightarrow \text{Spec in } F \parallel G$$

for **any** specification **Spec** (including **invariant**, **leads-to**, ...)

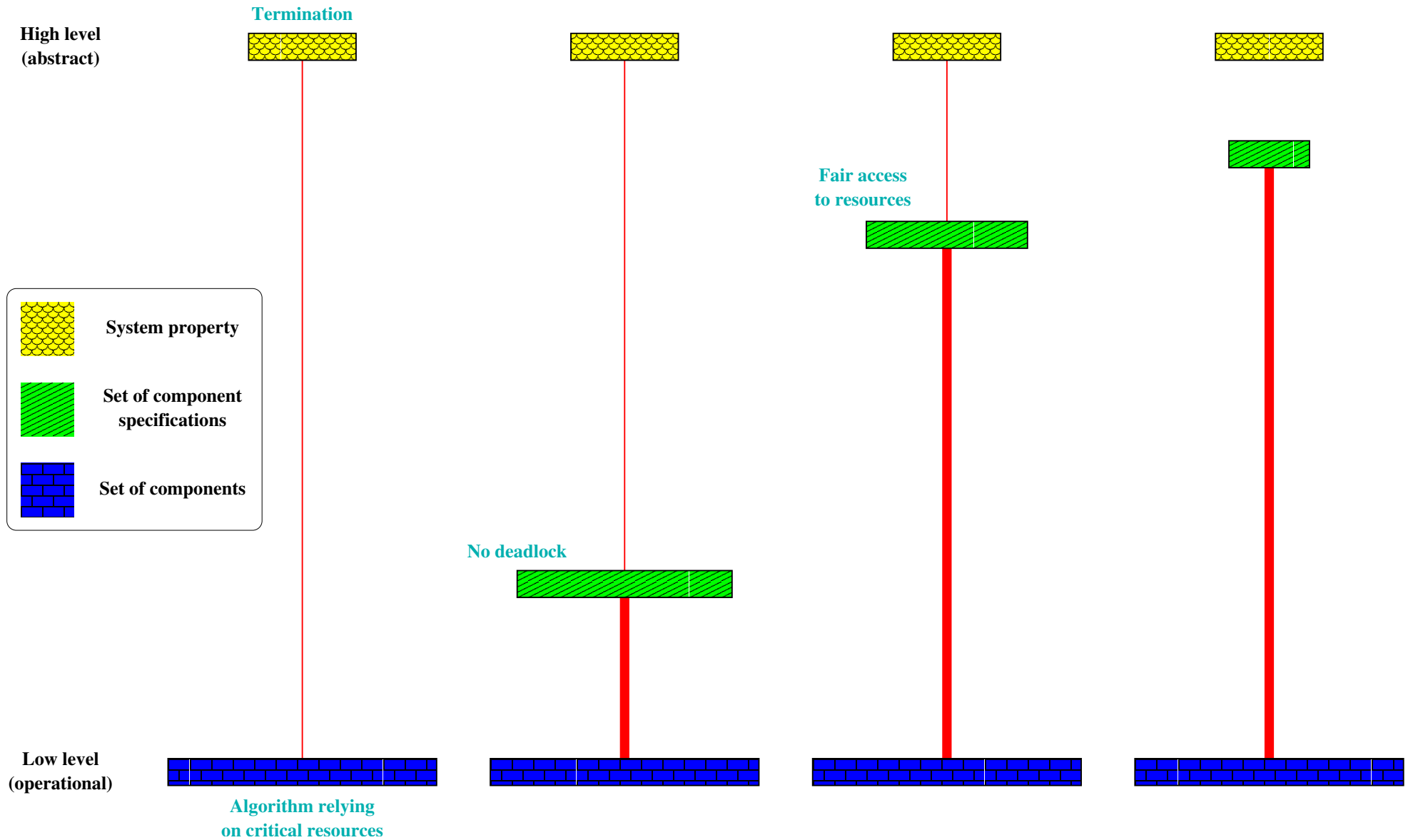
Component and proof reuse



Component and proof reuse (Cont'd)



High-level compositional specifications



Existential and universal specifications

(Mani Chandy, Beverly Sanders)

Existential: $\text{Spec in } F \vee \text{Spec in } G \Rightarrow \text{Spec in } F \circ G$

Universal: $\text{Spec in } F \wedge \text{Spec in } G \Rightarrow \text{Spec in } F \circ G$

Existential: Unity's **transient**, any TLA specification, ...

Universal: Unity's **next_T** and **inv_T**, any **existential** specification, ...

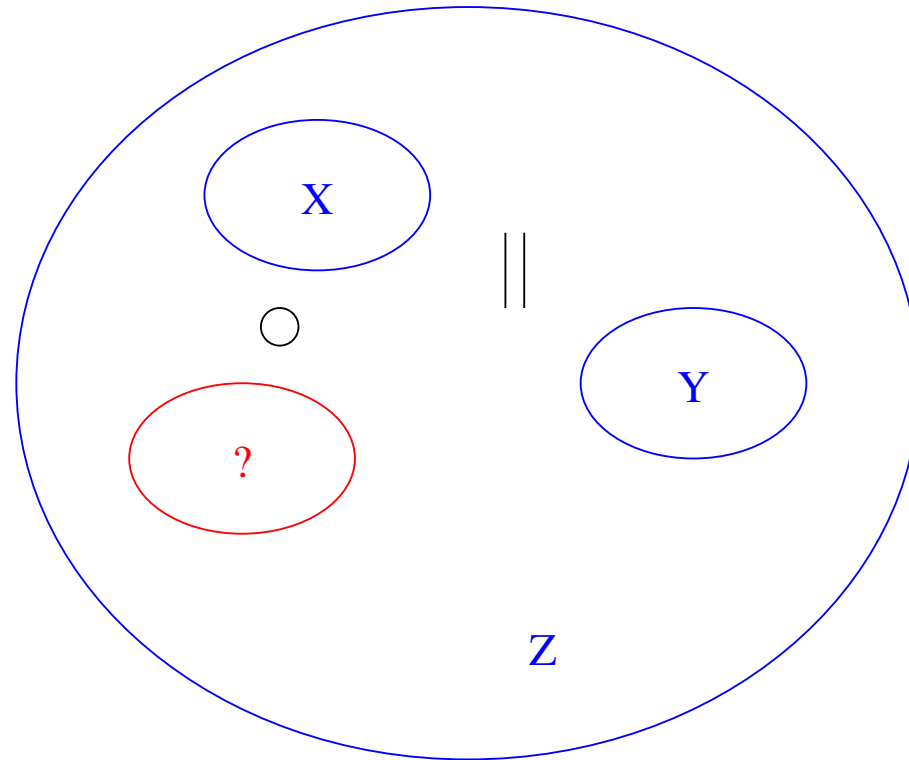
- **semantic** approach
- components can be (almost) **anything**
- law of composition \circ can be (almost) **anything**
- **guarantees** for existential **assumption-commitment** specifications

Predicate transformers for composition

$WE(\text{Spec}) \triangleq$ weakest existential specification stronger than Spec

- $WE(\text{Spec})$ in $F \equiv \forall G, H : \text{Spec} \text{ in } G \circ F \circ H$
(component F “brings” Spec into systems)
- WE is a predicate transformer
(universally conjunctive, monotonic, . . .)
- in some ways, WE is similar to Dijkstra’s *Weakest Precondition*
- a family of other transformers: SE , SU , WE^* , SE^* , SU^* , ~~WU~~
- X guarantees $Y \equiv WE(X \Rightarrow Y)$
- generalization to multiple laws of composition

Beyond “compositionality”



- Knowing X and Y , what can be said about Z ?
- Knowing Z , what can be said about (X, Y) ?
- Knowing X and Z , what can be said about Y ?

(Component **designer** / component **user** (system designer) point of views)

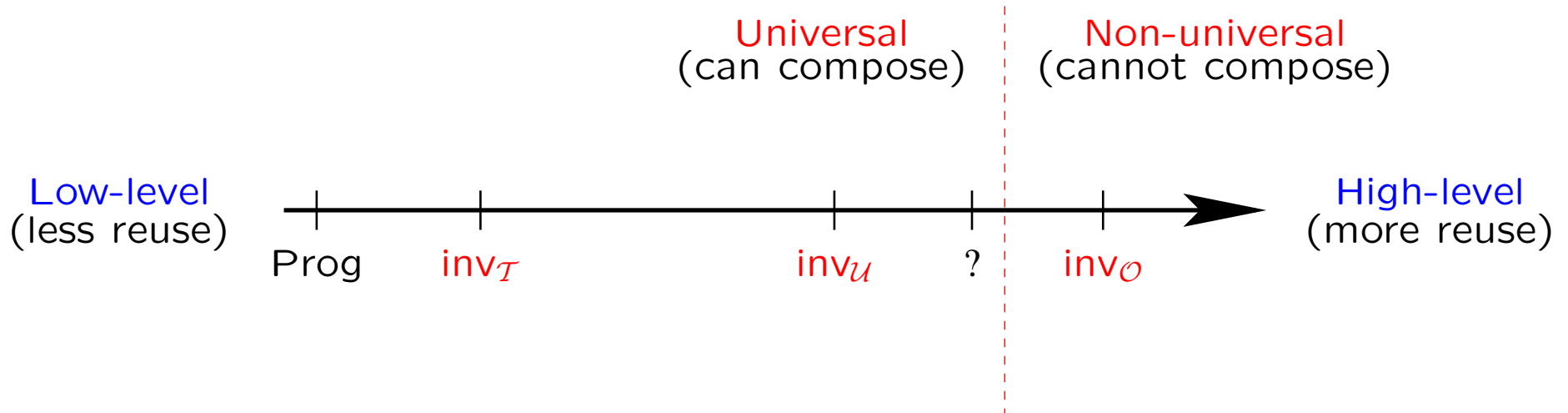
Application to Unity: high-level universal invariants

Strongest Invariant for Composition (SIC):


$inv_{\mathcal{E}} p \triangleq WE(inv_{\mathcal{O}} p)$ (existential specification)


$SIC \triangleq$ conjunction of all p s.t. $inv_{\mathcal{E}} p$ (reachable states with composition)

$$\begin{array}{ll} inv_{\mathcal{O}} p \equiv [SI \Rightarrow p] & inv_{\mathcal{E}} p \equiv [SIC \Rightarrow p] \\ p \text{ next}_{\mathcal{O}} q \equiv p \wedge SI \text{ next}_{\mathcal{T}} q & p \text{ next}_{\mathcal{U}} q \triangleq p \wedge SIC \text{ next}_{\mathcal{T}} q \\ inv_{\mathcal{O}} p \equiv (\text{initially } p) \wedge p \text{ next}_{\mathcal{O}} p & inv_{\mathcal{U}} p \triangleq (\text{initially } p) \wedge p \text{ next}_{\mathcal{U}} p \end{array}$$





Composition: a way to make proofs harder (Lamport)

 difficulties **artificially** introduced by using **ad-hoc** programming notations **disappear** when \parallel is \wedge

 to **come up** with Spec_F and Spec_G and to prove $F \Rightarrow \text{Spec}_F$, $G \Rightarrow \text{Spec}_G$ and $\text{Spec}_F \wedge \text{Spec}_G \Rightarrow \text{Spec}$ **more** difficult than to prove $F \wedge G \Rightarrow \text{Spec}$

 can Spec_F and Spec_G be substantially **simpler/smaller** than F and G ?

 **verify-while-develop** paradigm:
find **bugs** while trying to prove $\text{Spec}_F \wedge \text{Spec}_G \Rightarrow \text{Spec}$
(ignore $F \Rightarrow \text{Spec}_F$ and $G \Rightarrow \text{Spec}_G$)

 **composition** versus **decomposition**:
design and verify **reusable open** components