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

• **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: $\text{inv}_U$ 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

- 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 \( x, y, z : \text{integer} \)
Initially \( x = 0 \land y = 0 \)
Assign \( x := x + y \)
\[ \quad \text{if } x > 0 \]
\[ \quad x := x - 1 \]
\[ \quad \text{if } y < 10 \]
\[ \quad y := y + 1 \]
\[ \quad \text{if } y = 10 \]
\[ \quad z := z + 1 \]

\((x, y, z) : \langle (0, 0, -2), (0, 1, -2), (0, 2, -2), (2, 2, -2), (1, 2, -2), (1, 2, -2), (3, 2, -2), \ldots \rangle\)
Unity logic (\textcopyright Misra, 1995)

- **Safety** (prevent something bad):
  - \( p \text{ next } q \): if \( p \) is true in one state, \( q \) is true in the next state
  - \( p \text{ unless } q \): if \( p \) becomes true, it remains true at least until \( q \) becomes true (\( p \land \neg q \text{ next } p \lor q \))
  - \( \text{stable } p \): if \( p \) becomes true, it remains true (\( p \text{ next } p \) or \( p \text{ unless false} \)).
  - \( \text{inv } p \): \( p \) 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 \text{ \( \rightsquigarrow \) } 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 $x = 0 \land y = 0$

Assign $x := x + y$

| $x := x - 1$ if $x > 0$
| $y := y + 1$ if $y < 10$
| $z := z + 1$ 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 $\Rightarrow z > 100$

request $= \text{true}$ unless access $= \text{true}$

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

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

request $= \text{true}$ $\Rightarrow$ access $= \text{true}$

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

$(\text{inv flag} \Rightarrow \text{terminated}) \land (\text{terminated} \Rightarrow \neg \text{flag})$
• Basic rules from programs to specifications:
  – next: \[
  \forall \tau \text{ transition of } F : \{p\} \rightarrow \{q\} \\
  \frac{p \text{ next } q}{p \text{ next } q}
  \]
  – transient: \[
  \exists \tau \text{ transition of } F : \{p\} \rightarrow \{\neg p\} \\
  \frac{\text{transient } p}{p \text{ transient } p}
  \]
  – ... 

• Inference rules from specifications to specifications:
  – Conjunction: \[
  \text{inv } p, \text{ inv } q \\
  \frac{\text{inv } p \land q}{\text{inv } p \land q}
  \]
  – Progress-Safety-Progress (PSP): \[
  p \rightsquigarrow q, r \text{ unless } s \\
  \frac{p \land r \rightsquigarrow (q \land r) \lor s}{p \land r \rightsquigarrow (q \land r) \lor s}
  \]
  – ... 

• 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 \textit{true} by any invariant \( I \), and vice versa.”

- \( \mathcal{T} \): specifications proved \textit{without} the Substitution Axiom
- \( \mathcal{O} \): specifications proved \textit{with} the Substitution Axiom

\[
\begin{align*}
\text{inv}_\mathcal{T} \ p & \quad \quad \quad \quad \quad \quad \text{inv}_\mathcal{T} \ I \land p \\
\text{inv}_\mathcal{O} \ p & \quad \quad \quad \quad \quad \quad \text{inv}_\mathcal{O} \ p
\end{align*}
\]

in Unity (1988,1995), only one name (invariant): \textit{unsoundness}
Unity’s Substitution Axiom (Cont’d)

• \( T \): properties of atomic \textit{transitions} 
  \[ p \text{ next}_T q \equiv \forall \tau \text{ transition of } F : \{p\} \tau \{q\} \]

• \( O \): properties of infinite \textit{behaviors} 
  \[ p \text{ next}_O q \equiv \forall \sigma \text{ behavior of } F : \forall i : p.\sigma_i \Rightarrow q.\sigma_{i+1} \]

Example:

\[ \text{inv}_T x \geq 0 \land y \geq 0 \]
\[ \text{inv}_O x \geq 0 \]
\[ \text{inv}_T x \geq 0 \]

Why consider both \( T \) and \( O \) specifications?

• \( T \) specifications are a \textit{tool} to prove \( O \) specifications
• \( T \) specifications can be \textit{composed}, \( O \) specifications cannot
Parallel composition in Unity

∥ is called union and denoted by ⌦

\(F ∥ G\) defined by:

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

Composition theorems:

\[
\begin{align*}
\text{next}_T q \text{ in } F \land \text{next}_T q \text{ in } G & \implies \text{next}_T q \text{ in } F ∥ G \\
\text{inv}_T p \text{ in } F \land \text{inv}_T p \text{ in } G & \implies \text{inv}_T p \text{ in } F ∥ G \\
\text{inv}_T p \text{ in } F \land \text{stable}_T p \text{ in } G & \implies \text{inv}_T p \text{ in } F ∥ G \\
\text{transient}_p \text{ in } F & \implies \text{transient}_p \text{ in } F ∥ G
\end{align*}
\]

No such theorems for \(\text{next}_O, \text{inv}_O, \rightsquigarrow, \ldots\)
\section*{Specifications do not compose}

<table>
<thead>
<tr>
<th>Program</th>
<th>F</th>
<th>Program</th>
<th>G</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declare</td>
<td>$x, y, z : \text{integer}$</td>
<td>Declare</td>
<td>$x, y : \text{integer}$</td>
</tr>
<tr>
<td>Initially</td>
<td>$x = 0 \land y = 0$</td>
<td>Initially</td>
<td>$x = 0$</td>
</tr>
<tr>
<td>Assign</td>
<td>$x := x + y$</td>
<td>Assign</td>
<td>$x := x + 1$</td>
</tr>
<tr>
<td></td>
<td>$x := x - 1$ if $x &gt; 0$</td>
<td></td>
<td>$y := y - 1$ if $x &gt; 0$</td>
</tr>
<tr>
<td></td>
<td>$y := y + 1$ if $y &lt; 10$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$z := z + 1$ if $y = 10$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$\text{inv}_O x \geq 0 \text{ in } F$$

$$\text{inv}_O x \geq 0 \text{ in } G$$

but NOT $$\text{inv}_O x \geq 0 \text{ in } F || G$$

\textbf{Strongest Invariant (SI):} conjunction of all invariants

- $\text{inv}_O p \equiv [\text{SI} \Rightarrow p]$
- $p \text{ next}_O q \equiv p \land \text{SI} \text{ next}_T q$
- 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

\[ I \triangleq x = 0 \]
\[ S_1 \triangleq x' = x + 1 \land y' = y \]
\[ S_2 \triangleq x > 0 \land x' = x \land y' = y - 1 \]
\[ N \triangleq S_1 \lor S_2 \]
\[ \text{Fairness} \triangleq \text{WF}_{\langle x, y \rangle}(S_1) \land \text{WF}_{\langle x, y \rangle}(S_2) \]
\[ G \triangleq I \land \Box[N]_{\langle x, y \rangle} \land \text{Fairness} \]

\[ G \Rightarrow \Box(x \geq 0) \]
\[ \neg G \lor \Box(x \geq 0) \]
\[ G \Rightarrow y \in \mathbb{Z} \Rightarrow y < 0 \]
\[ G \Rightarrow \Box(x > 0 \Rightarrow \Box(x > 0)) \]
\[ G \Rightarrow \Box[x > 0 \Rightarrow x' > 0]_{\langle x, y \rangle} \]

more formal, but closer to Unity than it looks . . .
Composition in TLA

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

\[
\begin{align*}
I_1 & \triangleq x = 0 \\
S_1 & \triangleq x' = x + 1 \land y' = y \\
G_1 & \triangleq I_1 \land \Box[S_1]_x \land \text{WF}_x(S_1)
\end{align*}
\]

\[
\begin{align*}
I_2 & \triangleq \text{true} \\
S_2 & \triangleq x > 0 \land x' = x \land y' = y - 1 \\
G_2 & \triangleq I_2 \land \Box[S_2]_y \land \text{WF}_y(S_2)
\end{align*}
\]

\[
G \equiv G_1 \land G_2 \quad \text{(by calculation)}
\]

Consequence for composition:

\[
\text{Spec in } F \Rightarrow \text{Spec in } F \parallel G
\]

for any specification Spec (including invariant, leads-to, \ldots)
Component and proof reuse

Global System

Component 1

Component 2

Component ...

Component ...

T

T

C

Program Text 1

Program Text 2

Component ...

Component ...

Component ...

Component ...
High-level compositional specifications

High level (abstract)

Termination

Low level (operational)

Algorithm relying on critical resources

No deadlock

System property

Set of component specifications

Set of components

Fair access to resources

Termination

No deadlock

System property

Set of component specifications

Set of components
Existential and universal specifications

(Mani Chandy, Beverly Sanders)

Existential: \( \text{Spec in } F \lor \text{Spec in } G \Rightarrow \text{Spec in } F \circ G \)

Universal: \( \text{Spec in } F \land \text{Spec in } G \Rightarrow \text{Spec in } F \circ G \)

Existential: Unity's transient, any TLA specification, \ldots

Universal: Unity's \( \text{next}_T \) and \( \text{inv}_T \), any existential specification, \ldots

- 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

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

- \( WE(\text{Spec}) \) in \( F \equiv \forall G, H : \text{Spec in } G \circ F \circ H \)
  (component \( F \) “brings” \( \text{Spec} \) into systems)

- \( WE \) is a predicate transformer
  (universally conjunctive, monotonic, . . . )

- in some ways, \( WE \) is similar to Dijkstra’s \textit{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):

\( \text{inv}_E p \triangleq WE(\text{inv}_O p) \) (existential specification)

\( \text{SIC} \triangleq \text{conjunction of all } p \text{ s.t. } \text{inv}_E p \) (reachable states with composition)

\[
\begin{align*}
\text{inv}_O p & \equiv [\text{SI} \Rightarrow p] \\
p \text{next}_O q & \equiv p \land \text{SI next}_T q \\
\text{inv}_O p & \equiv (\text{initially } p) \land p \text{next}_O p \\
\text{inv}_U p & \equiv (\text{initially } p) \land p \text{next}_U p
\end{align*}
\]
Composition: a way to make proofs harder (Lamport)

difficulties artificially introduced by using ad-hoc programming notations disappear when \( \| \) is \( \wedge \)

to come up with \( \text{Spec}_F \) and \( \text{Spec}_G \) 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 \( \text{Spec}_F \) and \( \text{Spec}_G \) 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