Compositionality in system design: interfaces everywhere!

Stavros Tripakis
UC Berkeley
Computers as parts of cyber-physical systems

“~98% of the world’s processors are not in PCs but are embedded”

“a premium car today has:
- ~80 computers (ECUs – Electronic Control Units)
- ~100 million lines of code”

Future: smart cars on smart roads, smart power grids, smart hospitals, smart cities, ...
Languages & tools for CPS

Simulink: 1 million licenses in 2004

Key concepts:
- reactive behavior
- concurrency
- timing
- I/O
- ...

Key capabilities:
- simulation
- code generation
- verification

Simulink

Modelica / Dymola

LabVIEW

SCADE
Vision

• These modeling languages of today will become the **system-programming** languages of tomorrow

Rich languages: concurrency, time, robustness, reliability, energy, security, ...

**system compiler**

Complex execution platforms: networked, distributed, multicore, ...

Powerful analyses: model-checking, WCET analysis, schedulability, performance analysis, reliability analysis, ...
Model-Based Design

How to describe what we want?

How to be sure that this is what we want?

How to build it? Automatically Correct-by-construction
Compositional Model-Based Design

Industry: “system integration is key”

How to describe systems modularly?

How to check properties in a compositional way?

Compositionality, modularity

How to synthesize parts of the system independently?
This talk: interfaces

• Interface synthesis

• Interface theories
This talk

• **Interface synthesis**
  - joint with R. Lublinerman, C. Szegedy, E. Lee, M. Geilen, B. Rodiers, D. Bui

• **Interface theories**
model = tree of sub-models
Hierarchy in block diagrams
Hierarchy in block diagrams

Modularization:
hide details, master complexity
Hierarchy benefits

Total number of blocks: ~100
Max. number at any level: ~6

model = tree of sub-models
Can we exploit this hierarchy beyond syntax?

• Can we reuse a submodel?

• Can we treat it as a “black box”?

• Can we build model libraries?

Surprisingly, the answer is often “no” even in state-of-the-art tools
Modular compilation

Enables incremental compilation, IP protection, ..., and libraries!
Modular code generation

Can we do the same for system-design languages?

Code for block A

```plaintext
initialize state;
while (true) do
  await clock tick;
  read inputs;
  compute;
  write outputs;
  update state;
end while
```

Code for block B

```plaintext
initialize state;
while (true) do
  await clock tick;
  read inputs;
  compute;
  write outputs;
  update state;
end while
```

Code (C, C++, Java, ...)

Tripakis
Standard code generation: “monolithic”

```plaintext
P().fire(x1, x2) returns (y1, y2)
{
    y1 := A().fire(x1);
    y2 := B().fire(x2);

    return (y1, y2);
}
```
Problem with “monolithic” code

P\textbf{.fire}(x_1, x_2) \textbf{returns} \ (y_1, y_2);
Problem with “monolithic” code

False I/O dependencies
=>
code not usable in some contexts

Parallel composition of functions is not a function!

P.fire(x1, x2) returns (y1, y2);
Brief Simulink demo
Solution: non-monolithic code

A Mealy machine with multiple output functions

\[ \text{P.fire1( x1 ) returns y1 \{ return A.fire( x1 ); } \]

\[ \text{P.fire2( x2 ) returns y2 \{ return B.fire( x2 ); } \]

Tripakis
Non-monolithic interface

P.fire1( x1 ) returns y1 ;

P.fire2( x2 ) returns y2 ;
Non-monolitholic interface

\[ P.\text{fire1}(x_1) \text{ returns } y_1 ; \]

\[ P.\text{fire2}(x_2) \text{ returns } y_2 ; \]
Non-monolithic interface

interface does not restrict usage

P.fire1(x1) returns y1;

P.fire2(x2) returns y2;
Bottom-up interface synthesis

Given interfaces for sub-blocks A, B, C, compute interface for composite block P.
Simple non-monolithic interface

\[
P.\text{fire1} (\ x_1\ ) \ \text{returns} \ y_1 ;\\
\]

\[
P.\text{fire2} (\ x_2\ ) \ \text{returns} \ y_2 ;\\
\]
What about more complex diagrams?

what about this?

or this?
Interface synthesis for block diagrams

= graph clustering

block diagram

interface
How it’s done

A

B

C

D

P
How it’s done
How it’s done

clustering
How it’s done

Interface for P

P

P.fire1()

P.fire2()

P.fire3()
Different clusterings => different interfaces

A non-monolithic interface for $P$

A monolithic interface for $P$

trade-off: interface size vs. reusability
Different clustering algorithms = different tradeoffs

<table>
<thead>
<tr>
<th>Clustering method</th>
<th>Complexity</th>
<th>Achieves maximal reusability?</th>
<th>Achieves minimal interf. size?</th>
<th>Modularity bound?</th>
<th>Achieves minimal code size?</th>
</tr>
</thead>
<tbody>
<tr>
<td>“step-get”</td>
<td>Polynomial</td>
<td>No</td>
<td>Almost</td>
<td>&lt;=2 functions</td>
<td>Yes</td>
</tr>
<tr>
<td>“dynamic”</td>
<td>Polynomial</td>
<td>Yes</td>
<td>Yes</td>
<td>&lt;=N+1 functions*</td>
<td>No</td>
</tr>
<tr>
<td>“disjoint”</td>
<td>NP-complete</td>
<td>Yes</td>
<td>Yes</td>
<td>?</td>
<td>Yes</td>
</tr>
<tr>
<td>“greedy”</td>
<td>Polynomial</td>
<td>Yes</td>
<td>No</td>
<td>?</td>
<td>Yes</td>
</tr>
</tbody>
</table>

* N = number of block outputs
Bottom-up interface synthesis = automatic abstraction

Efficient + provably optimal
Interfaces for multirate models

Block A fires every 3 rounds

When does P need to fire?
GCD(3,2) = 1: over-approximation too conservative

Interface enriched with deterministic unary automata

3:伤病饭恼

2:伤病饭恼

3 ∪ 2:伤病饭恼

union
Interfaces for dataflow models

A  

B  

FIFO queue

P

monolithic interface
Interfaces for dataflow models

A monolithic interface leads to deadlocks, while the original model does not deadlock.

[ACM TECS 2013]
Interfaces for dataflow models

SDF++:
- shared FIFOs
- but deterministic!

Non-monolithic interface for P
(generated automatically)
This talk

• Interface synthesis

• **Interface theories**
Incremental design

A “steer-by-wire” system:
Incremental design

\[ \nu \in [\nu_{\min}, \nu_{\max}] \]

Latency \( \leq 10\text{ms} \)
Incremental design

How to ensure properties are preserved?

\[ v \in [v_{\text{min}}, v_{\text{max}}] \]

\[ \text{latency} \leq 10\text{ms} \]
Interface theories [Alfaro, Henzinger, et al.]

- **Interface** = component abstraction
- **Interface composition**: $A \cdot B = C$
- **Interface refinement**: $A' \preceq A$

- **Theorems**:

  1. If $A' \preceq A$ and $A$ satisfies $P$ then $A'$ satisfies $P$.
  2. If $A' \preceq A$ and $B' \preceq B$, then $A' \cdot B' \preceq A \cdot B$. 
Substitutability

(1) If $A' \leq A$ and $A$ satisfies $P$ then $A'$ satisfies $P$.

(2) If $A' \leq A$ and $B' \leq B$, then $A' \cdot B' \leq A \cdot B$.

$Z \leq B$ and (1) and (2) $\Rightarrow$ substitutability
Which interface theories for CPS?

– Synchronous relational interfaces
  • Functional properties (correctness)

– Actor interfaces
  • Performance properties (throughput, latency, ...)

\[ v \in [v_{\text{min}}, v_{\text{max}}] \]

\[ \text{latency} \leq 10ms \]
Which interface theories for CPS?

– Synchronous relational interfaces
  • Functional properties (correctness)

– Actor interfaces
  • Performance properties (throughput, latency, ...)

$v \in [v_{\text{min}}, v_{\text{max}}]$

$\text{latency} \leq 10\text{ms}$
Interfaces for correctness

- Akin to “behavioral types” [Liskov et al]
Interfaces for correctness: stateful (dynamic) interface

1-place buffer

write
read

full
empty

Static interface:
(holds at every round)

\[ \neg (\text{empty} \land \text{full}) \land \neg (\text{write} \land \text{read}) \land \text{empty} \Rightarrow \neg \text{read} \land \text{full} \Rightarrow \neg \text{write} \]

Dynamic (state-dependent) interface:

\[ \neg \text{read} \Rightarrow \text{write} \]

\[ \neg \text{write} \Rightarrow \text{read} \]

\[ \text{empty} \Rightarrow \text{full} \]

\[ \text{write} \Rightarrow \text{empty} \]

\[ \text{read} \Rightarrow \text{full} \]
Interfaces for correctness: compatibility checking = type checking

\[ x_2 \geq 0 \iff x_2 \neq 0 \land (x_1 > 0 \land x_2 > 0 \Rightarrow y > 0) \]

Type error!

Can be checked using SAT/SMT solvers
Interfaces for correctness: interface synthesis = type inference

\[ x_2 \geq z \quad \Rightarrow \quad x_2 \neq 0 \land (x_1 > 0 \land x_2 > 0 \Rightarrow y > 0) \]

Automatically synthesized constraint

z > 0
Interface synthesis:
“demonic” vs. standard composition

Standard:
\[ \phi := \phi_1 \land \phi_2 \]

“Demonic”:
\[ \phi := \phi_1 \land \phi_2 \land (\forall y : \phi_1 \Rightarrow \text{in}(\phi_2)) \]
\[ \text{in}(\phi_2) := \exists z : \phi_2 \]
Interfaces for correctness: refinement

\[ \phi' \leq \phi \]

\[ \text{def} \]

\[ = \]

\[ \text{in} (\phi) \Rightarrow \text{in} (\phi') \]

\[ (\text{in} (\phi) \land \phi') \Rightarrow \phi \]
Interfaces for correctness: refinement

\[
\begin{align*}
in (\phi) & \Rightarrow \quad in (\phi') \\
(in (\phi) \land \phi') & \Rightarrow \quad \phi
\end{align*}
\]

• Weaker than Liskov-Wing behavioral subtyping:

\[
\begin{align*}
in (\phi) & \Rightarrow \quad in (\phi') \\
\phi' & \Rightarrow \quad \phi
\end{align*}
\]
Main results

• Preservation of refinement by composition (parallel, serial, restricted feedback):

   If $A' \leq A$ and $B' \leq B$, then $A' \bullet B' \leq A \bullet B$.

• Refinement $\iff$ substitutability:

   $A'$ can replace $A$ in any context iff $A' \leq A$.

“full abstraction”: refinement not stronger than necessary
Which interface theories for CPS?

– Synchronous relational interfaces
  • Functional properties (correctness)

– Actor interfaces
  • Performance properties (throughput, latency, ...)

\[ v \in [v_{\text{min}}, v_{\text{max}}] \]

\[ \text{latency} \leq 10\text{ms} \]
Interfaces for performance

- Relations between I/O timed event streams
  - No values associated with events

\[ \Delta = 4 \]

delay
Interfaces for performance refinement: “the earlier the better”

- Deterministic delay: $\Delta = 4$
- Nondeterministic delay: $\Delta \in [1,3]$

- vs. standard refinement = inclusion of behaviors = implementations more deterministic than specs
What it buys us

time-deterministic model

A → B → C

easier to analyze, verify, ...

preserves worst-case performance (throughput, latency)

VI

time-nondeterministic system
Conclusions

• **Interfaces** = abstractions of components
  – Keep necessary information, hide the rest

• *No unique, one-size-fits-all interface model / theory*

• General principles:
  – Bottom-up interface synthesis for hierarchical languages
  – **Refinement** for incremental design
Thank you

• Questions?

Richer languages: concurrency, time, robustness, reliability, energy, security, ...

System Compiler

more complex execution platforms: networked, distributed, multicore, ...

more powerful analyses: model-checking, WCET analysis, schedulability, performance analysis, reliability analysis, ...