Component-Based System Design with Interfaces

Stavros Tripakis

UC Berkeley and Aalto University

Motivation: computers as parts of systems

How to design dependable systems in an affordable way?

...
Challenges in System Design
(a.k.a. “model-based design”)

How to describe the system that we want?
Simulation, verification, ...
How to be sure that this is what we want?

How to implement it?
Automatically Correct-by-construction
Logic synthesis, code gen,
Also: auto-design some “details” (e.g. controller synthesis)

Component-Based Design

Industry: “system integration is the real issue”

How to model systems modularly?
How to check properties in a compositional way?
Compositionality, modularity
How to synthesize parts of the system independently?
This talk

- Component-based design with interfaces

- Other topics (if there is time)
  - Basic problems in multi-view modeling (composition of overlapping aspects instead of interacting but separate components) [with Reineke]
  - Synthesis from scenarios and requirements (marry Harel + Vardi) [with Alur et al.]
Main messages

• Interfaces are a good thing.
  – They abstract components, keeping important information, and hiding the rest.

• There is not one unique (one-size-fits-all) interface formalism.
  – Different kinds of interfaces are good for different purposes.

Component-based design with interfaces

• **Interfaces for modeling and code generation:**
  – Achieving compositionality in hierarchical models
  – Modular code generation

• **Interfaces for static analysis and verification:**
  – Incremental design using interface theories

• **Interfaces for simulation**
  – Co-simulation of heterogeneous formalisms
  – Modular formal semantics for Ptolemy
Achieving compositionality in hierarchical modeling languages & Modular code generation

Joint work with R. Lublinerma, C. Szeged, E. Lee, M. Geilen, B. Rodiers, D. Bui

Embedded system languages & tools

Simulink

Simulink: 1 million licenses in 2004

LabVIEW

SCADE

UML/SysML

Modelica / Dymola
Hierarchy

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
Problem: these hierarchical formalisms are **non-compositional**

- A submodel cannot be represented as an equivalent atomic model

---

**Case 1: synchronous block diagrams**

![Synchronous block diagram](image)

**Synchronous, deterministic semantics:**

- Inputs
- Outputs
- Inputs
- Outputs
Fundamental model: Mealy machine

- $M = (S, s_0, I, O, \delta, \lambda)$
- where:
  - $S$: set of states
  - $s_0$: initial state
  - $I$: set of inputs
  - $O$: set of outputs
  - $\delta: S \times I \rightarrow S$: transition function
  - $\lambda: S \times I \rightarrow O$: output function

Special case: Moore machine

$\lambda: S \rightarrow O$

Special case: stateless machine

(i.e., a function) $\lambda: I \rightarrow O$

Think of these as an interface

What is the serial composition of two Mealy machines?

\[
\begin{array}{c}
\text{M1} & \text{M2} \\
x_1 & y_1 = x_2 & y_2 \\
\end{array}
\]

\[
\begin{array}{c}
x_1 & \text{M} & y_2 \\
\end{array}
\]

A new Mealy machine
What is the parallel composition of two Mealy machines?

\[ \lambda((s_1, s_2), (x_1, x_2)) = (\lambda_1(s_1, x_1), \lambda_2(s_2, x_2)) \]
What if we add feedback?

\[ \lambda((s_1, s_2), (x_1, x_2)) = (\lambda_1(s_1, x_1), \lambda_2(s_2, x_2)) \]
Motivation: modular code generation

- Source code (e.g., .cc files) → gcc → Standard Compiler (e.g., gcc) → Linker (e.g., ld) → Object code (e.g., .o files) → Executable

Enables incremental compilation, IP protection, libraries, ...

Tripakis

Motivation: modular code generation

Can we do the same for model-based design languages?

- Initialize state; while (true) do await clock tick; read inputs; compute; write outputs; update state; end while

Code for block A

Code for block B

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

Tripakis
State-of-the-art code generation (e.g., Simulink)

• "Monolithic" code generation:

```
A
\downarrow
B
P

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

State-of-the-art code generation (e.g., Simulink)

• Problem with monolithic code:

```
P.fire(x1, x2) returns (y1, y2);
```
State-of-the-art code generation (e.g., Simulink)

- Original diagram has no cycles:

```
P.fire(x1, x2) returns (y1, y2);
```

State-of-the-art tools (e.g., Simulink) suffer from this problem
Solution: non-monolithic code

\[ \text{P.fire1( x1 ) returns y1 { return A.fire( x1 ); } } \]
\[ \text{P.fire2( x2 ) returns y2 { return B.fire( x2 ); } } \]

Non-monolithic interface

\[ \text{P.fire1( x1 ) returns y1 ; } \]
\[ \text{P.fire2( x2 ) returns y2 ; } \]
Non-monolithic interface

interface does not restrict usage

Non-monolithic interface
So, what’s the parallel composition of two functions?

function = stateless Mealy machine  
non-monolithic interface

What is the interface for a Moore machine?

• A DAG (directed acyclic graph)

interface for a Moore machine
So, what’s the parallel composition of two Mealy machines?

A generalized Mealy machine = a DAG of interface functions

Back to modular code generation

• Essentially an interface synthesis problem:
Given interfaces for sub-blocks A, B, C, compute interface for composite block P.

Interface synthesis: sometimes relatively straightforward ...

P.fire1(x1) returns y1;
P.fire2(x2) returns y2;
... but generally non-trivial

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

Interface for A
Interface for B
Interface for C
Interface for D
P
How it’s done

clustering

Interface for P

P

P

P

P

P

P
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>&quot;step-get&quot;</td>
<td>Polynomial</td>
<td>No</td>
<td>Almost</td>
<td>&lt;=2 functions</td>
<td>Yes</td>
</tr>
<tr>
<td>&quot;dynamic&quot;</td>
<td>Polynomial</td>
<td>Yes</td>
<td>Yes</td>
<td>&lt;=N+1 functions*</td>
<td>No</td>
</tr>
<tr>
<td>&quot;disjoint&quot;</td>
<td>NP-complete</td>
<td>Yes</td>
<td>Yes</td>
<td>?</td>
<td>Yes</td>
</tr>
<tr>
<td>&quot;greedy&quot;</td>
<td>Polynomial</td>
<td>Yes</td>
<td>No</td>
<td>?</td>
<td>Yes</td>
</tr>
</tbody>
</table>

* N = number of block outputs
Case 2: modular code generation and interfaces for multi-rate models

Period of block A: 3

What is the period of P?
$\text{GCD}(3,2) = 1$: over-approximation too conservative

Interface enriched with deterministic unary automata

$3 \cup 2:
\rightarrow \rightarrow \rightarrow \rightarrow \rightarrow$

Case 3: modular code generation and interfaces for dataflow models (SDF)

1 2 3 1
P A B P

FIFO queue

monolithic interface
(corresponds to schedule AAABB)
Non-compositionality of SDF

Original model does not deadlock

With monolithic interface: it deadlocks!

Interfaces for SDF

SDF++:
- shared FIFOs
- but deterministic!

Non-monolithic interface for P
(generated automatically)
Component-based design with interfaces

• Interfaces for modeling and code generation:
  ✓ Achieving compositionality in hierarchical models
  ✓ Modular code generation

• Interfaces for static analysis and verification:
  – Incremental design using interface theories

• Interfaces for simulation
  – Co-simulation of heterogeneous formalisms
  – Modular formal semantics for Ptolemy

Tripakis
Modular formal semantics for Ptolemy

Joint work with C. Stergiou, C. Shaver, E. Lee (Berkeley)

Ptolemy tool: heterogeneous modeling and simulation [Edward Lee et al., UC Berkeley]
Actors ...

... and directors
How does Ptolemy work?

- What is a Ptolemy actor?
  - In reality: a Java class implementing a certain Java interface
  - Theoretically (our formalization): some sort of timed Mealy machine

- What is a Ptolemy director?
  - In reality: a Java class ...
  - Theoretically (our formalization): a composition operator turning a composite actor (network of actors) into an atomic actor

- Modular, open architecture: new actors and directors can be added without affecting the rest

The formal interface of Ptolemy actors

- **Fire:** \( F : S \times I \to O \)
- **Postfire:** \( P : S \times I \to S \)
- **Deadline:** \( D : S \times I \to R_+ \cup \{\infty\} \)
- **TimeUpdate:** \( T : S \times I \times R_+ \to S \)

- Of these, Ptolemy in fact uses only the first two: timed actors rely on a different function, `fireAt()`, implemented by the director and called by the actor
Co-simulation of heterogeneous models

Joint work with D. Broman, C. Brooks, E. Lee (Berkeley), L. Greenberg (IBM), M. Masin (IBM), M. Wetter (LBNL)

Problems: tool interoperability? model exchange? co-simulation?

More than just software engineering: semantic heterogeneity!
FMI

- FMI (Functional Mock-up Interface)
- A standard API for model exchange and co-simulation
  - Submodels: “FMUs”
- Promising: backed-up by industry (Daimler, Dassault Systemes, IBM, ...).

Issues with FMI

- Semantics of API (e.g., `doStep(\Delta t)` ) are tricky, because FMUs can reject a time step.
- Unclear how to implement a correct master algorithm.
Problems with FMI

• How to develop a correct, deterministic master algorithm?
  – In what order to execute the FMUs?

• No arbitrary determinism:
  – E.g., from left to right, alphabetically by name, ...

Deterministic co-simulation

• Specified FMI and FMU contracts formally.
• Proposed two master algorithms
  – Proved correctness (termination & determinism & ...)
• Influencing FMI standard.
  – E.g., making some options mandatory.

\[ \text{If } \text{doStep}(\Delta t) \text{ is accepted then } \text{doStep}(\Delta t') \text{ will also be accepted for all } \Delta t' < \Delta t. \]
Component-based design with interfaces

• Interfaces for modeling and code generation:
  – Achieving compositionality in hierarchical models
  – Modular code generation

• Interfaces for static analysis and verification:
  – Incremental design using interface theories

• Interfaces for simulation
  – Co-simulation of heterogeneous formalisms
  – Modular formal semantics for Ptolemy

Tripakis
Incremental design using interface theories

Joint work with B. Lickly, E. Lee, M. Geilen, M. Wiggers, C. Stergiou (Berkeley), T. Henzinger (IST Austria), M. Broy (TU Munich), V. Preoteasa (Aalto)

Incremental analysis

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

How to ensure properties are preserved?

\[ v \in [v_{\min}, v_{\max}] \]
Interface theories

- **Interface** = component abstraction
- **Interface composition**: \( \text{A} \bullet \text{B} = \text{C} \)
- **Interface refinement**: \( \text{A}' \leq \text{A} \)

**Theorems:**

1. If \( \text{A}' \leq \text{A} \) and \( \text{A} \) satisfies \( \text{P} \) then \( \text{A}' \) satisfies \( \text{P} \).
2. If \( \text{A}' \leq \text{A} \) and \( \text{B}' \leq \text{B} \), then \( \text{A}' \bullet \text{B}' \leq \text{A} \bullet \text{B} \).

**Note:**

- Composition is partial \(\Rightarrow\) can specify compatibility (local property)

Substitutability

1. If \( \text{A}' \leq \text{A} \) and \( \text{A} \) satisfies \( \text{P} \) then \( \text{A}' \) satisfies \( \text{P} \).
2. If \( \text{A}' \leq \text{A} \) and \( \text{B}' \leq \text{B} \), then \( \text{A}' \bullet \text{B}' \leq \text{A} \bullet \text{B} \).

\( \text{Z} \leq \text{B} \) and (1) and (2) \( \Rightarrow \) substitutability
Which interface theories for embedded systems?

– Synchronous relational interfaces  
  • Functional properties (correctness)

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

  latency \leq 10\text{ms}

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

Interfaces for correctness

\[
(\text{Real, Real}) \rightarrow \text{Real}
\]

standard type

block

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

relational interface

conjunction ≠ implication

[ACM TOPLAS 2011]

[HSCC 2011]
Interfaces for correctness: compatibility checking = type checking

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

*Type error!*

Can be checked using SAT/SMT solvers

Interfaces for correctness: interface synthesis = type inference

\[ z > 0 \]

\[ x_2 \geq z \]

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

*Automatically synthesized constraint*
Interface synthesis:
“demonic” vs. standard composition

Standard: \( \phi := \phi_1 \land \phi_2 \)

“Demonic”:
\[
\phi := \phi_1 \land \phi_2 \land (\forall y : \phi_1 \Rightarrow in(\phi_2))
\]
\[
in(\phi_2) := \exists z : \phi_2
\]

Interfaces for correctness: refinement

\[
\phi' \leq \phi \quad \text{def} \quad \begin{cases} 
\text{in(} \phi \text{) } \Rightarrow \text{in(} \phi' \text{)} \\
\text{(in(} \phi \text{)} \land \phi') \Rightarrow \phi 
\end{cases}
\]

- Refinement \(\Rightarrow\) substitutability:
  
  \textit{A’ can replace A in any context iff A’} \(\leq\) \textit{A}.

  - i.e., refinement both necessary and sufficient condition for substitutability.
  - Previous similar notions (e.g., Liskov-Wing behavioral subtyping) are sufficient but not necessary, so sometimes too strong.
Interfaces for correctness: stateful (dynamic) interface

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:

\[ \neg \text{write} \]
\[ \neg \text{read} \]
\[ \text{s0} \]
\[ \rightarrow \]
\[ \text{write} \]
\[ \text{empty} \]
\[ \rightarrow \]
\[ \text{read} \]
\[ \text{full} \]

Which interface theories for dynamical systems?

– Synchronous relational interfaces
  • Functional properties (correctness)

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

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

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

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

\[
\begin{array}{c}
\bullet \ \bullet \ \bullet \ \ldots \ \rightarrow \ \Delta = 4 \\
t_1 \ t_2 \ t_3 \ \rightarrow \ \text{delay} \ \rightarrow \ \bullet \ \bullet \ \bullet \ \ldots \\
t_{1+4} \ t_{2+4} \ t_{3+4}
\end{array}
\]

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

**time-nondeterministic system**

preserves worst-case performance (throughput, latency)

VI

easier to analyze, verify, ...

Other topics
Multi-view modeling

Joint work with J. Reineke (Saarland)

Multi-view modeling:
Complex systems => different stakeholders => different concerns

Multi-view modeling: model different aspects of the system (e.g., functionality, performance, energy, cost, ...)

Low-level controllers Simulink

Supervisory controllers Rhapsody/SysML

Physical dynamics Modelica
From (separate) components to (overlapping) views

- **Refinement theories**
  - model interacting but *separate* components

- **Multi-view modeling**
  - Views model *overlapping* aspects

Problem: how to guarantee view consistency?

Example: Geometric (Static) Views

- 3D structure
- 2D views

Inconsistent!
Recent work

- From static to **dynamic** (i.e., dynamical system) views.
- Formalized views and view **consistency** (does there exist a system which could generate a given set of views?).
- Algorithms for:
  - Verification: view consistency checking
  - Synthesis: given set of consistent views, synthesize witness system.

Conclusions

- Model-based system design: key for smart societies of the future
- **Compositionality and interfaces**: key for complex system modeling, analysis, and implementation