Rigorous System Design

Joseph Sifakis
RiSD Laboratory EPFL

PODC 2014
Paris
16-18 July 2014
**RECIPE (Program)**

- Put apples in pie plate;
- Sprinkle with cinnamon and 1 tablespoon sugar;
- In a bowl mix 1 cup sugar, flour and butter;
- Blend in unbeaten egg, pinch of salt and the nuts;
- Mix well and pour over apples;
- Bake at 350 degrees for 45 minutes

**INGREDIENTS (Resources)**

- 1 pie plate buttered
- 5 or 6 apples, cut up
- \(\frac{3}{4}\) c. butter, melted
- 1 c. flour
- \(\frac{1}{2}\) c. chopped nuts
- 1 tsp cinnamon
- 1 tbsp sugar
- 1 c. Sugar

Design is a Universal Concept!
System Design – Two Main Gaps

Correctness?

Correctness?

Requirements (declarative)

Application SW Development

Application SW (executable)

Implementation

System (HW+SW)
Rigorous System Design – The Concept

Considers design as a formal accountable and iterative process for deriving trustworthy and optimized implementations from an application software and models of its execution platform and its external environment.

- **Model-based**: successive system descriptions are obtained by correct-by-construction source-to-source transformations of a single expressive model rooted in well-defined semantics.
- **Accountable**: possibility to assert which among the requirements are satisfied and which may not be satisfied.

Focuses on mastering and understanding design as a problem solving process based on divide-and-conquer strategies involving iteration on a set of steps and clearly identifying:

- points where human intervention and ingenuity are needed to resolve design choices through requirements analysis and confrontation with experimental results.
- segments that can be supported by tools to automate tedious and error-prone tasks.
Rigorous System Design – Four Guiding Principles

**Separation of concerns:** Keep separate what functionality is provided (application SW) from how it is implemented by using resources of the target platform.

**Components:** Use components for productivity and enhanced correctness.

**Coherency:** Based on a single model to avoid gaps between steps due to the use of semantically unrelated formalisms e.g. for programming, HW description, validation and simulation, breaking continuity of the design flow and jeopardizing its coherency.

**Correctness-by-construction:** Overcome limitations of a posteriori verification through extensive use of provably correct reference architectures enforcing essential properties as well as source-to-source transformations.
Rigorous System Design
- Component-based Design
- Correct-by-construction Design

The BIP Framework
- Modeling Interactions
- Modeling Priorities
- Distributed Implementation

Discussion
Any engineering discipline uses a limited number of types of components and rules for interconnecting them. This is a prerequisite for the development of formal frameworks!

- Heterogeneity
Component-based Design – Synchronous vs. Asynchronous

Synchronous components (HW, Multimedia application SW)
- Execution is a sequence of non interruptible steps

Asynchronous components (General purpose application SW)
- No predefined execution step

Open problem: Theory for consistently composing synchronous and asynchronous components e.g. GALS
Component-based Design – Synchronous vs. Asynchronous

UML Model (Rational Rose)
Component-based Design – Interaction Mechanisms

Rendezvous: atomic symmetric synchronization

Broadcast: asymmetric synchronization triggered by a Sender

Existing formalisms and theories are not expressive enough

- use variety of low-level coordination mechanisms including semaphores, monitors, message passing, function call
- encompass point-to-point interaction rather than multiparty interaction
Component-based Design – Programming Styles

- Thread-based programming
- Actor-based programming

Software Engineering
Systems Engineering
Is it possible to express component coordination in terms of composition operators? We need a unified composition paradigm for describing and analyzing the coordination between components in terms of tangible, well-founded and organized concepts and characterized by

- **Orthogonality**: clear separation between behavior and coordination constraints
- **Minimality**: uses a minimal set of primitives
- **Expressiveness**: achievement of a given coordination with a minimum of mechanism and a maximum of clarity

Most component composition frameworks fail to meet these requirements

- Some are formal such as process algebras e.g. CCS, CSP, pi-calculus
- Other are ad hoc such as most frameworks used in software engineering e.g. ADL, or systems engineering e.g. SystemC
The Concept of Glue

Build a component $C$ satisfying a given property $P$, from

- $\mathcal{C}_0$ a set of atomic components described by their behavior
- $\mathcal{GL} = \{gl_1, \ldots, gl_i, \ldots\}$ a set of glue operators on components

Glue operators are stateless – separation of concerns between behavior and coordination
We use operational semantics to define the meaning of a composite component – glue operators are “behavior transformers”

Glue Operators
- build interactions of composite components from the actions of the atomic components e.g. parallel composition operators
- can be specified by using a family of operational semantics rules (the Universal Glue)
**Component-based Design – Glue Operators: Example**

$gl$ is defined by

\[
\frac{q_1 \cdot a \rightarrow q'_1}{q_1 q_2 \cdot a \rightarrow q'_1 q_2}
\]

\[
\frac{q_1 \cdot a \rightarrow q'_1 \quad q_2 \cdot c \rightarrow q'_2}{q_1 q_2 \cdot ac \rightarrow q'_1 q'_2}
\]

\[
\frac{q_1 \cdot b \rightarrow q'_1 \quad \neg q_2 \cdot c \rightarrow}{q_1 q_2 \cdot b \rightarrow q'_1 q_2}
\]

$gl(B_1,B_2)$
A **glue operator** defines **interactions** as a set of derivation rules of the form

$$\{ q_i - a_i \rightarrow i q'_i \}_{i \in I} \quad \text{C}(q_k) \quad k \in K$$

$$(q_1, \ldots, q_n) - a \rightarrow (q'_1, \ldots, q'_n)$$

- $I, K \subseteq \{1, \ldots, n\}$, $I \neq \emptyset$
- $a = \bigcup_{i \in I} a_i$ is an interaction
- $q'_i = q_i$ for $i \notin I$

A **glue** is a set of glue operators
Glue is a first class entity independent from behavior that can be decomposed and composed.

1. Incrementality

2. Flattening
Component-based Design – Glue Operators: Expressiveness

- Different from the usual notion of expressiveness!
- Based on strict separation between glue and behavior

Given two glues $G_1$, $G_2$

$G_2$ is strongly more expressive than $G_1$

if for any component built by using $G_1$ and a set of components $C_0$

there exists an equivalent component built by using $G_2$ and $C_0$
Given two glues $G_1$, $G_2$

$G_2$ is weakly more expressive than $G_1$

if for any component built by using $G_1$ and a set of components $C_0$

there exists an equivalent component built by using $G_2$ and $C_0 \cup C$

where $C$ is a finite set of coordinating components.
Component-based Design – Glue Operators: Expressiveness

- Rigorous System Design
  - Component-based Design
  - Correct-by-construction Design

- The BIP Framework
  - Modeling Interactions
  - Modeling Priorities
  - Distributed Implementation

- Discussion
Correct-by-Construction

Requirements → sat Functional

Application SW

System Model

sat Extra-Functional

≥: refinement relation preserving functional properties

Focus on invariants and deadlock-freedom

Execution Platform
Architectures

- depict design principles, paradigms that can be understood by all, allow thinking on a higher plane and avoiding low-level mistakes

- are a means for ensuring global properties characterizing the coordination between components – correctness for free

- Using architectures is key to ensuring trustworthiness and optimization in networks, OS, middleware, HW devices etc.

System developers extensively use libraries of reference architectures ensuring both functional and non functional properties e.g.

- Fault-tolerant architectures
- Resource management and QoS control
- Time-triggered architectures
- Security architectures
- Adaptive Architectures
An architecture is a family of operators $A(n)[X]$ parameterized by their arity $n$ and a family of characteristic properties $P(n)$

- $A(n)[B_1,..,B_n] = gl(n) (B_1,..,B_n, C(n))$, where $C(n)$ is a set of coordinators
- $A(n)[B_1,..,B_n]$ meets the characteristic property $P(n)$.

Characteristic property: atomicity of transactions, fault-tolerance ….

Note that the characteristic property need not be formalized!
Rule 1: Property Enforcement

Architecture for Mutual Exclusion

Components

Architecture for Mutual Exclusion satisfies Mutex
Feature interaction in telecommunication systems, interference among web services and interference in aspect programming are all manifestations of a lack of composability.

The Refinement Relation \( \geq \)

- \( S1 \geq S2 \) (S2 refines S1) if
  - all traces of S2 are traces of S1 (modulo some observation criterion)
  - if S1 is deadlock-free then S2 is deadlock-free too
  - \( \geq \) is preserved by substitution

- Proof by establishing simulation or bisimulation relations between S1 and S2
Correct-by-Construction – Refinement Preservation

Preservation of $\geq$ by substitution
Correct-by-Construction – Refinement Preservation

C_1  C_2

\( \geq \)

C'_1  D  C'_2

C_1  C_3  C_2

\( \not\geq \)
Rigorous System Design
- Component-based Design
- Correct-by-construction Design

The BIP Framework
- Modeling Interactions
  - Modeling Priorities
  - Distributed Implementation

Discussion
Modeling in BIP – Basic Concepts

Layered component model

Composition operation parameterized by glue IN12, PR12
Modeling in BIP – Atomic Components

- Interface: set of ports and associated variables
- Behavior: set of transitions of the form

\[ z := x^2 - y^2 \]

\[
\begin{array}{c}
\text{Interface: set of ports and associated variables} \\
\text{Behavior: set of transitions of the form}
\end{array}
\]

\[
\begin{array}{c}
\text{(port, guard, assignment)} \\
\end{array}
\]

\[
\begin{array}{c}
in_x \\
in_y \\
x \\
y \\
z \\
out
\end{array}
\]
Modeling Interactions – Connectors

Express interactions by combining two protocols: rendezvous and broadcast

- A **connector** is a set of ports that can be involved in an interaction
- Port attributes (trigger, synchron) are used to model rendezvous and broadcast.
- An **interaction** of a connector is a set of ports such that: either it contains some trigger or it is maximal.

\[ s + sr2 + sr3 + sr2r3 \]
Modeling Interactions – Connectors

Atomic Broadcast:  
a+abc

Causality chain:  
a+ab+abc+abcd

a(1+x)  
x=b(1+y)  
y=c(1+d)
Broadcast
a’bc

Atomic Broadcast
a’[bc]

Causality chain
a’[b’[c’d]]

\[ a(1+b)(1+c) \]

\[ a(1+bc) \]

\[ a(1+b(1+c(1+d))) \]

\[ b(1+c(1+d)) \]

\[ c(1+d) \]
Modeling Interactions – Connectors

\[ (a'b)'c \approx a'bc \]

\[ a'b' \approx a'b + ab' \]
Modeling Interactions – Connectors with Data Transfer

Note: There is no distinction between input and output ports.
Modeling Interactions – Connectors with Data Transfer

(w ← pq).[ x_w := max(x_p, x_q) // x_p := x_w, x_q := x_w]
(r ← ws).[ x_r := max(x_w, x_s) // x_w := x_r, x_s := x_r]

(r ← pqs).[ x_r := max(x_p, x_q, x_s) // x_p := x_r, x_q := x_r, x_s := x_r]
The algebra of connectors

- Axiomatization
- Boolean representation allowing efficient implementation with BDDs


Synthesis of connectors


Dynamic Connectors

Rigorous System Design
- Component-based Design
- Correct-by-construction Design

The BIP Framework
- Modeling Interactions
- Modeling Priorities
- Distributed Implementation

Discussion
Modeling Priorities

Priority rules

<table>
<thead>
<tr>
<th>Priority rule</th>
<th>Restricted guard g₁’</th>
</tr>
</thead>
<tbody>
<tr>
<td>true → p₁ ↯ p₂</td>
<td>g₁’ = g₁ ∧ ¬ g₂</td>
</tr>
<tr>
<td>C → p₁ ↯ p₂</td>
<td>g₁’ = g₁ ∧ ¬(C ∧ g₂)</td>
</tr>
</tbody>
</table>
Modeling Priorities – FIFO policy

PR : \( t1 \leq t2 \rightarrow b1 \langle b2 \)
\( t2 < t1 \rightarrow b2 \langle b1 \)

- idle1
  - a1
  - \( t1 := 0 \)

- ready1
  - b1

- exec1
  - f1

- idle2
  - a2
  - \( t2 := 0 \)

- ready2
  - b2

- exec2
  - f2

- #
Modeling Priorities – EDF policy

PR: $D_1-t_1 \leq D_2-t_2 \rightarrow b_2 \prec b_1$  
$D_2-t_2 < D_1-t_1 \rightarrow b_1 \prec b_2$

```
idle1
  a1
  t1:=0
ready1
  b1
exec1
f1
  t1 \leq D_1
idle2
  a2
  t2:=0
ready2
  b2
exec2
f2
  t2 \leq D_2
```
Modeling Priorities – Composability

\[ \begin{array}{c}
PR2 \\
PR1 \\
\end{array} \quad \neq \quad \begin{array}{c}
PR1 \\
PR2 \\
\end{array} \]

\[ \begin{array}{c}
a \langle {}^1 b \\ c \\
\end{array} \quad \begin{array}{c}
b \langle {}^2 c \\
c \\
\end{array} \]

\[ \begin{array}{c}
a \langle {}^1 b \\
\end{array} \quad \begin{array}{c}
a \\
\end{array} \]

\[ \begin{array}{c}
b \langle {}^2 c \\
\end{array} \quad \begin{array}{c}
\end{array} \]
Modeling Priorities – Composability

We take:

\[
\begin{array}{c|c|c|c|c|}
PR1 & \cdot & \cdot & \cdot & PR2 \\
\cdot & \cdot & \cdot & \cdot & \cdot \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|}
PR1 & \cdot & \cdot & \cdot & \cdot & PR2 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|}
PR1 \oplus PR2 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot \\
\end{array}
\]

PR1\(\oplus\) PR2 is the least priority containing PR1\(\cup\)PR2

Results:

• The operation \(\oplus\) is partial, associative and commutative
• PR1(PR2(B)) \(\neq\) PR2(PR1(B))
• PR1\(\oplus\) PR2(B) refines PR1\(\cup\)PR2(B) refines PR1(PR2(B))
• Priorities preserve deadlock-freedom
Modeling Priorities – Mutual Exclusion + FIFO policy

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1 \leq t_2 )</td>
<td>( b_1 \langle b_2 )</td>
</tr>
<tr>
<td>( t_2 &lt; t_1 )</td>
<td>( b_2 \langle b_1 )</td>
</tr>
<tr>
<td>True</td>
<td>( b_1 \langle f_2 )</td>
</tr>
<tr>
<td>True</td>
<td>( b_2 \langle f_1 )</td>
</tr>
</tbody>
</table>

Diagram:
- \( \text{idle1} \) to \( \text{a1} \) to \( \text{t1} := 0 \) to \( \text{ready1} \) to \( \text{b1} \) to \( \text{f1} \) to \( \text{exec1} \)
- \( \text{idle2} \) to \( \text{a2} \) to \( \text{t2} := 0 \) to \( \text{ready2} \) to \( \text{b2} \) to \( \text{f2} \) to \( \text{exec2} \)
Modeling Priorities – Deadlock Detection

Risk of deadlock: \( PR \oplus PR' \) is not defined
Rigorous System Design
- Component-based Design
- Correct-by-construction Design

The BIP Framework
- Modeling Interactions
- Modeling Priorities
- Distributed Implementation

Discussion
BIP is based on:
- Global state semantics, defined by operational semantics rules, implemented by the BIP Engine
- Atomic multiparty interactions, e.g. by rendezvous or broadcast and priorities

Correct-by-construction translation of BIP models with multiparty interactions and priorities into observationally equivalent S/R-BIP models
- Point to point communication by asynchronous message passing
- No global state - Atomicity of transitions is broken by separating interaction from internal computation
- The BIP Engine is replaced by a set of Engines executing subsets of interactions
- Distributed coordination is orchestrated by an architecture
Distributed Implementation – Single Execution Engine

BIP Model

\[ \alpha_1 < \alpha_2 < \alpha_3 \]

One Engine executes all interactions!
Single Execution Engine

Atomic components

- **busy**: notify involved atoms
- **execute**: launch atom threads
  - execute chosen interaction
- **choose**: choose a maximal interaction
- **filter**: filter using priorities
- **stable state**: compute feasible interactions
  - ready
- **init**
Distributed Implementation – From BIP to S/R-BIP

Transformation of atomic components

Before reaching a state, the set of the offered ports is sent to some Engine

$s_{1\perp}$ and $s_{2\perp}$ are “undefined” states
Distributed Implementation – Multiple Engines

Dispatch interactions across **multiple** engines!
Distributed Implementation – Static Conflict Relation

\[ \alpha_1 \text{ and } \alpha_2 \text{ involve ports of conflicting transitions} \]

\[ \alpha_1 \text{ and } \alpha_2 \text{ share a common port } p \]

\[ \alpha_1 \# \alpha_2: \text{ interactions } \alpha_1 \text{ and } \alpha_2 \text{ may be in conflict} \]
Distributed Implementation – Conflict-Free Multiple Engines

Each engine handles interactions of a class of #*
Distributed Implementation – Limitations

Taking #* may reduce drastically parallelism between interactions
Distributed Implementation – 3-Layer Architecture

[α1,α2][α3,α4]

C1 C2 C3 C4 C5 C6

α1 α2 α3

α4

Conflict Resolution Protocol

Interaction Protocol

α1 α2

port offer

ok fail reserve

Interaction Protocol

α3 α4

port offer

ok fail reserve

offer port offer port offer port offer port offer port offer port

C’1 C’2 C’3 C’4 C’5 C’6
Distributed Implementation – Token Ring CRP

Interaction Protocol
\( \alpha_1, \alpha_2 \)

Interaction Protocol
\( \alpha_3, \alpha_4 \)
Distributed Implementation – Dining Philosophers CRP

Interaction Protocol

α1, α2

Dining Philosophers CRP

reserve2 ok2 fail2

reserve3 ok3 fail3

reserve4 ok4 fail4

Interaction Protocol

α3, α4

C1' C2' C3' C4' C5' C6'

α1 α2 α3 α4
Distributed Implementation – Design Flow

Conflict Resolution Protocol

Partitioning of Interactions

Partitioning of Components

Sockets/C++ Code

MPI/C++ Code

Code Generator

Mapping of Components

Interaction Prot. $\alpha_1 \alpha_2$

Interaction Prot. $\alpha_3 \alpha_4$

Dining Philo. CRP

$\alpha_1 \alpha_2$

$\alpha_3 \alpha_4$

$C_1 \ C_2 \ C_3 \ C_4 \ C_5 \ C_6$

$C_1' \ C_2' \ C_3' \ C_4' \ C_5' \ C_6'$

$\alpha_1 \ \alpha_2 \ \alpha_3 \ \alpha_4$

Dining Philo. CRP

Dining Philo. CRP
Conflict resolution is essential for maintaining overall coherency in particular integrity of resources.

Conflicts between interactions can be resolved locally if they are handled by the same IP. Otherwise, they are resolved by the CRP.

Conflict resolution based on static conflict relation may lead to considerable loss in performance – detect false conflicts based on partial state knowledge.

- Sifakis et al: Knowledge-Based Distributed Conflict Resolution for Multiparty Interactions and Priorities. FMOODS/FORTE 2012: 118-134
- Sifakis et al: Optimized distributed implementation of multiparty interactions with observation. AGERE!@SPLASH 2012: 71-82
Distributed Implementation – Optimization Issues

- Code optimization for components implemented on the same site - replace a composite component by a single flattened component from which sequential monolithic code can be generated


![Graph showing performance comparison](image_url)
Endowing system design with scientific foundations is a major scientific challenge – key ideas:

- Design is formalized as a process of source-to-source correct-by-construction scalable transformations
- Semantic coherency is achieved by using a single expressive component framework
- Architectures are essential for enforcing specific characteristic properties by application of generic coordination principles

There is a huge body of not yet well-formalized solutions to coordination problems in the form of algorithms, protocols, hardware and software architectures. The challenge is to

- Formalize these solutions as architectures and prove their correctness
- Provide a taxonomy of the architectures and their characteristic properties
- Is it possible to decompose any component coordination property as the conjunction of predefined characteristic properties enforced by corresponding architectures?
Discussion

The BIP component framework has been developed for more than 10 years, with Rigorous Design in mind.

- Translation of DSL (Simulink, Lustre, DOL, nesC) into BIP
- Source-to-source transformations proven correct-by-construction
  - taking into account HW resources
  - generating distributed implementations for several platforms
  - code optimization
- Run-times for centralized execution/simulation, distributed execution, real-time execution
- Validation and analysis tools
  - incremental checking for Deadlock-freedom: D-Finder tool
  - statistical Model Checking
- Successful application in many industrial projects
  - software componentization for robotic systems (DALA Space Robot for Astrium)
  - programming multi-core systems (P2012 for STM, MPPA for Kalray)
  - complex systems modeling (AFDX and IMA for Airbus)
Thank You