Seminar 4: Correctness-by-construction

Joseph Sifakis
RiSD Laboratory, EPFL
Computing – Foundations, Goals and Challenges

Next Seminar

December 21, 2012
January 11, 2013

RiSD Laboratory, EPFL, Lausanne
Correct-by-Construction Design

- Correct-by-construction design
  - is at the root of any mature engineering discipline. Is scalable and do not suffer limitations of correctness-by-checking.
  - avoids limitations of a posteriori verification by focusing on system design as a model-based and component-based process.

- System developers extensively use
  - algorithms, protocols and architectures that have been proven to be correct.
  - compilers, to get across abstraction levels and translate high-level languages into (semantically equivalent) object code.

- All of these results and techniques largely account for our ability to master complexity and develop systems cost-effectively.

- Nonetheless, we still lack theory and methods for combining them in principled and disciplined fully correct-by-construction design flows.
Correct-by-Construction Design

Requirements

sat Functional

Application SW

VI

System Model

sat Extra-Functional

sat: satisfaction relation

≥: refinement relation

Execution Platform
Component-based Construction

Correctness-by-Construction

The BIP Component Framework
- Modeling Interactions
- Modeling Priorities
- Compositional Verification
- HW-driven Refinement
- Distribution-driven Refinement

Discussion
Component-based Construction

- Components are indispensable for enhanced productivity and correctness
- Component composition lies at the heart of the parallel computing challenge
- There is no Common Component Model - Heterogeneity
There exists a large variety of mechanisms used to express coordination between components e.g. semaphores, monitors, locks, function call, asynchronous message passing rendezvous, broadcast ….

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 functionality with a minimum of mechanism and a maximum of clarity

None of the existing component composition frameworks satisfies 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 in systems engineering e.g. SystemC
Build a component $C$ satisfying a given property $P$, from:

- $C_0$ 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 coordination mechanisms such as protocols, schedulers, buses
- We need a unified composition paradigm for describing and analyzing the coordination between components in terms of tangible, well-founded and organized concepts
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 derivation rules (the Universal Glue)
A glue operator defines interactions as a set of derivation rules of the form

\[ \left\{ q_i - a_i \rightarrow_i q'_i \right\}_{i \in I} \quad C(q_k)^{k \in K} \]

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

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

Notice that, non deterministic choice and sequential composition are not glue operators

A glue is a set of glue operators
Component-based Construction – Glue Operators: Example

$gl$ is defined by

\[
\begin{align*}
q_1 - a & \rightarrow q'_1 \\
q_1 q_2 - a & \rightarrow q'_1 q_2 \\
q_1 - a & \rightarrow q'_1 \\
q_2 - c & \rightarrow q'_2 \\
q_1 q_2 - ac & \rightarrow q'_1 q'_2 \\
q_1 - b & \rightarrow q'_1 \\
q_2 - c & \rightarrow \\
q_1 q_2 - b & \rightarrow q'_1 q_2 \\
\end{align*}
\]
Glue is a first class entity independent from behavior that can be decomposed and composed.

1. Incrementality

2. Flattening
Component-based Construction – Expressiveness

- Comparison between formalisms and models is done by flattening structure and reduction to behaviorally equivalent models e.g. finite state automaton, Turing machine

- This leads to notions of expressiveness that are not adequate for comparing coordination capabilities of languages and models e.g.
  - all finite state formalisms turn out to be expressively equivalent
  - all modeling and programming languages are Turing complete, while their coordination capabilities tremendously differ

Objective:
- Propose notions of expressiveness based on a strict separation between behavior and coordination
- Compare existing frameworks by using such notions
Component-based Construction – 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 $\mathcal{C}_0$
there exists an equivalent component built by using $G_2$ and $\mathcal{C}_0$
Component-based Construction – Expressiveness

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 Construction

- Correctness-by-Construction

  - The BIP Component Framework
    - Modeling Interactions
    - Modeling Priorities
    - Compositional Verification
    - HW-driven Refinement
    - Distribution-driven Refinement

- Discussion
Correctness-by-Construction for sat

Base elements e.g. atomic components
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 component transformer $A(n)[X]$ and a characteristic property $P(n)$, parameterized by an integer $n$ such that

- $A(n)[C_1,..,C_n] = gl(n) (C_1,..,C_n, D(n))$, where $D(n)$ is a set of coordinating components
- $A(n)[C_1,..,C_n]$ meets the characteristic property $P(n)$.

Characteristic property: atomicity of transactions, fault-tolerance ....
Correctness-by-Construction for sat – Architectures

Rule 1: Property Preservation

Deadlock-free Routing Protocol

Deadlock-free components

Deadlock-free Routing Protocol

Deadlock-free composite component
Correctness-by-Construction for sat – Architectures

Rule2: Property Enforcement

Architecture for Mutual Exclusion

Components

satisfies Mutex
An architecture ensuring a given property can be obtained as the combination of a set of architectures ensuring basic properties.

For example, security architectures are obtained by composition of architectures ensuring

- Antivirus protection
- Intrusion Detection System, Intrusion Protection System
- Sampling
- Monitoring
- Watermarking
- Embedded cryptography
- Integrity checking

**Composability:** We need theory for combining basic architectures and their characteristic properties to obtain an architecture meeting a given global property
Feature interaction in telecommunication systems, interference among web services and interference in aspect programming are all manifestations of a lack of composability.
Correctness-by-Construction for Refinement

The Refinement Relation $\geq$

Rendezvous

$S_1 \geq S_2$ (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

Protocol (Asynch Message Passing)

$S_1 \geq S_2$ (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
Correctness-by-Construction for Refinement

Preservation of $\geq$ by substitution
Correctness-by-Construction for Refinement

Preservation of $\geq$ by substitution

\[
\begin{align*}
\text{bgn}(a) & \quad \text{rcv}(a) \\
\text{fns}(a) & \quad \text{ack}(a) \\
\text{bgn}(b) & \quad \text{rcv}(b) \\
\text{fns}(b) & \quad \text{ack}(b)
\end{align*}
\]
The AS is written in high level languages supporting abstractions such as:

- Atomicity of primitives and interactions between components – in particular multiparty interaction
- A logical notion of time assuming zero-time actions and synchrony of execution wrt to the physical environment

The generated system model is a refinement of the AS generated automatically for a given mapping associating:

- Processes of the ASW → processors of the platform
- Data of the ASW → memories of the platform
- Interactions → execution paths or protocols

Source-to-source transformation in the host language
- Component-based Construction

- Correctness-by-Construction

- The BIP Component Framework
  - Modeling Interactions
  - Modeling Priorities
  - Compositional Verification
  - HW-driven Refinement
  - Distribution-driven Refinement

- Discussion
BIP – Correct-by-Construction Design Flow

1. Application SW
2. Code Generation
3. Execution Platform
4. Hardware Infrastructure
5. Mapping
7. Design Space Exploration
8. Protocols

- Embedding Application SW model in BIP
- Transformation System model in BIP
- Transformation S2S
- Transformation S2S
- Transformation S2S
- Transformation ≥
- Transformation IV
- Transformation ≤
BIP – Component-based Construction

Layered component model

Expressiveness

Composition operation parameterized by glue IN12, PR12
Component-based Construction

Correctness-by-Construction

The BIP Component Framework
  - Modeling Interactions
    - Modeling Priorities
    - Compositional Verification
    - HW-driven Refinement
    - Distribution-driven Refinement

Discussion
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.

\[ s + sr2 + sr3 +sr2r3 \]
Atomic Broadcast: \(a+abc\)

Causality chain: \(a+ab+abc+abcd\)
Modeling Interactions – Connectors

Broadcast
a’bc

Atomic Broadcast
a’[bc]

Causality chain
a’[b’[c’d]]
Modeling Interactions – Connectors

\[ (a'b)'c \approx a'b \quad \text{and} \quad a'bc \approx a'b + ab' \]
Modeling Interactions – The Algebra of Connectors

**Syntax:**

\[
\begin{align*}
  s & ::= [0] | [1] | [p] | [x] \quad \text{(synchrons)} \\
  t & ::= [0]' | [1]' | [p]' | [x]' \quad \text{(triggers)} \\
  x & ::= s | t | x.x | x + x
\end{align*}
\]

where \( P \) is a set of ports, such that \( 0, 1 \notin P \)

\[+\]
\( \text{union} \)

idempotent, associative, commutative, identity \([0]\)

\[.\]
\( \text{fusion} \)

idempotent, associative, commutative, identity \([1]\),
distributive wrt + ([0] is not absorbing)

\([]\), \([\ ]'\)
\( \text{typing} \)

 unary operators

**Semantics:** defined as a function \(|\cdot|: AC(P) \to 2^P\)

**Results** [Bliudze & Sifakis, EmSoft 07]:

- Axiomatization
- Boolean representation allowing efficient implementation
- Component-based Construction
- Correctness-by-Construction
- The BIP Component Framework
  - Modeling Interactions
  - Modeling Priorities
  - Compositional Verification
  - HW-driven Refinement
  - Distribution-driven Refinement
- Discussion
Modeling Priorities

**Priority rules**

<table>
<thead>
<tr>
<th>Priority rule</th>
<th>Restricted guard $g_1'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$true \to p_1 \ll p_2$</td>
<td>$g_1' = g_1 \land \neg g_2$</td>
</tr>
<tr>
<td>$C \to p_1 \ll p_2$</td>
<td>$g_1' = g_1 \land \neg(C \land g_2)$</td>
</tr>
</tbody>
</table>
Modeling Priorities – FIFO policy

PR : \( t_1 \leq t_2 \rightarrow b_1 \langle b_2 \)

\( t_2 < t_1 \rightarrow b_2 \langle b_1 \)

\[
\begin{align*}
\text{idle1} & \quad \text{a1} \\
\quad & \quad \text{start t1} \\
\text{ready1} & \quad \text{b1} \\
\text{exec1} & \quad \text{f1} \\
\text{idle2} & \quad \text{a2} \\
\quad & \quad \text{start t2} \\
\text{ready2} & \quad \text{b2} \\
\text{exec2} & \quad \text{f2} \\
\end{align*}
\]
Modeling Priorities – EDF policy

PR: D1 - t1 ≤ D2 - t2 → b2 / b1

D2 - t2 < D1 - t1 → b1 / b2
Modeling Priorities – Composability

PR2
PR1

PR1
PR2

a \langle 1 \, b

b \langle 2 \, c

c

b \langle 2 \, c

a \langle 1 \, b

c

a

b

c

a

b

c
Modeling Priorities – Composability

We take:

\[ PR1 \oplus PR2 \]

PR1⊕PR2 is the least priority containing PR1∪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:**

- **States:**
  - `idle1`
  - `ready1`
  - `exec1`
  - `idle2`
  - `ready2`
  - `exec2`

- **Transitions:**
  - `a1` from `idle1` to `ready1`
  - `start t1` from `ready1` to `exec1`
  - `f1` from `exec1` to `idle1`
  - `a2` from `idle2` to `ready2`
  - `start t2` from `ready2` to `exec2`
  - `f2` from `exec2` to `idle2`
Modeling Priorities – Example

PR : b1 \langle f2 b2 \langle \{ f1, b1' \} (mutex on R)

PR' : b2' \langle f1 b1' \langle \{ f2, b2 \} (mutex on R')

Risk of deadlock: PR \oplus PR' is not defined
Modeling in BIP – The Language

Priorities

\[ z_4 > 0 \]

\[ p_{123} < r_{34} \]

Interactions

Behavior

\[ v := \max(u, x_3) \]

\[ x_1 := u \]

\[ y_1 := x_1 / 2 \]

\[ x_1 \text{++} \]

\[ y_1 := x_2 \]

\[ y_2 := f_2(x_2) \]

\[ [y_1 < y_2] \]

\[ q_{123} \]

\[ q_1 \]

\[ q_2 \]

\[ q_3 \]

\[ r_3 \]

\[ z_3 \]

\[ r_4 \]

\[ x_1 \]

\[ x_2 \]

\[ x_3 \]

\[ x_4 \]

\[ p_1 \]

\[ p_2 \]

\[ p_3 \]

\[ p_4 \]

\[ p_{1234} \]

\[ p_{12} \]

\[ p_{123} \]

\[ p_3 \]

\[ r_{34} \]
BIP – The Language

// atomic component definition
atomic type Atom(int p, int q, ...)
data int x, y, z, ...
data DataType u, v, w, ...
port MyPort p1(x)
port TypePort2 p2(y, u)

place s1, s2, s3, s4, ...

initial to s1
  do { /* initialization code */ }
  on p1 from s1 to s2
    provided guard1
    do { /* transition code */ }
  on p2 from s2 to s3
    provided x < y
    do { /* plain C code */ }

export port MyPort p1 is rl

end

// connector type definition
connector type Bus (PortType1 p1, PortType2 p2, ...)

define port-expression
data int y

on interaction1 provided guard1
  up { /* interaction code */ }
  down { /* interaction code */ }

... on p1 p2 provided p1.x > 0
  up { y = p1.x + p2.x }
  down { { p1.x = p2.x = y } }

... export port PortType p0(y)

end

// compound component type definition
compound type Compo(int p, ...)

component CompType_1 c1(p, ...)
  ...
component CompType_n cn

connector ConType_1 x1( c1.p, ... c2.q )
  ...
connector ConType_k xk( x1.p0, cn.r )

priority priol
  provided guard
  xi:interaction1 < xj:interaction2

... export port PortType1 c1.p is p
export port PortTypek xk.p0 is q
  ...

end
- Component-based Construction
- Correctness-by-Construction

- The BIP Component Framework
  - Modeling Interactions
  - Modeling Priorities
  - Compositional Verification
  - HW-driven Refinement
  - Distribution-driven Refinement

- Discussion
BIP – Compositional Verification

Verify **global deadlock-freedom** of a system by separate analysis of the components and of the architecture.

Potential deadlock
\[ D = \text{en}(p1) \land \neg \text{en}(p2) \land \text{en}(q2) \land \neg \text{en}(q1) \]

Potential deadlock
\[ D = \text{en}(p1) \land \neg \text{en}(p2) \land \text{en}(q2) \land \neg \text{en}(q3) \land \text{en}(r3) \land \neg \text{en}(r1) \]
Method:
Eliminate potential deadlocks $D$ by computing compositionally
global invariants $\chi$ such that $\chi \land D = \text{false}$

\[ \begin{align*}
B_1 & \models \Box \phi_1 \\
B_2 & \models \Box \phi_2 \\
\psi & \in \mathbb{II}(\gamma(B_1, B_2), \phi_1, \phi_2) \\
\phi_1 \land \phi_2 \land \psi & \Rightarrow \chi
\end{align*} \]
BIP – Compositional Verification

Verification
Component Deadlock-freedom

Abstraction and II generation

Satisfiability II ∨ CI ∨ D

Deadlock confirmation

Deadlock-freedom

Deadlocks

Omega

Yices

BIP model

BIP simulation

---

false-strengthen ≠false-give up
## BIP – Compositional Verification

<table>
<thead>
<tr>
<th>Example</th>
<th>Number of Comp</th>
<th>Number of Ctrl States</th>
<th>Number of Bool Variables</th>
<th>Number of Int Var</th>
<th>Number Potential Deadlocks</th>
<th>Number Remaining Deadlocks</th>
<th>Verification Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature Control (2 rods)</td>
<td>3</td>
<td>6</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>3</td>
<td>3s</td>
</tr>
<tr>
<td>Temperature Control (4 rods)</td>
<td>5</td>
<td>10</td>
<td>0</td>
<td>5</td>
<td>32</td>
<td>15</td>
<td>6s</td>
</tr>
<tr>
<td>UTOPAR (40 cars, 256 CU)</td>
<td>297</td>
<td>795</td>
<td>40</td>
<td>242</td>
<td>--</td>
<td>0</td>
<td>3m46s</td>
</tr>
<tr>
<td>UTOPAR (60 cars, 625 CU)</td>
<td>686</td>
<td>1673</td>
<td>60</td>
<td>362</td>
<td>--</td>
<td>0</td>
<td>25m29s</td>
</tr>
<tr>
<td>R/W(10000 readers)</td>
<td>10002</td>
<td>20006</td>
<td>0</td>
<td>1</td>
<td>--</td>
<td>0</td>
<td>36m10s</td>
</tr>
<tr>
<td>Philosophers (13000)</td>
<td>26000</td>
<td>65000</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>3</td>
<td>38m48s</td>
</tr>
<tr>
<td>Philosophers (10000)</td>
<td>20000</td>
<td>50000</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>3</td>
<td>29m30s</td>
</tr>
<tr>
<td>Smokers (5000)</td>
<td>5001</td>
<td>10007</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>0</td>
<td>14m</td>
</tr>
<tr>
<td>Gas stations (500 pumps, 5000 customers)</td>
<td>5501</td>
<td>21502</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>0</td>
<td>18m55s</td>
</tr>
</tbody>
</table>

Results obtained by using the D-Finder tool: [http://www-verimag.imag.fr/~thnguyen/tool/]
BIP – Compositional Verification

![Graph showing verification time vs gas station size]
- Component-based Construction

- Correctness-by-Construction

- The BIP Component Framework
  - Modeling Interactions
  - Modeling Priorities
  - Compositional Verification
  - HW-driven Refinement
  - Distribution-driven Refinement

- Discussion
HW-driven refinement – The Design Flow

The Design Flow:

1. **Application SW**
   - Application SW Model
   - Instrumentation: API, Observer injection
   - Instrumented System Model
   - Native BIP Simulation

2. **Mapping**
   - Mapping
   - bipWeaver

3. **Architecture**
   - Architecture
   - HW Architecture Model
   - HDL Component Library
   - Code Generation
   - Multi-threaded application code
   - HDS Code

4. **System Model**
   - System Model

5. **Output**
   - Performance Evaluation
   - Code Generation
   - Code Generation
   - Instrumented System Model
   - Native BIP Simulation
   - Performance Results

6. **Input**
   - Application SW
   - Application SW Model
   - HW Component Library
   - HDL Component Library

7. **Tools**
   - dol2bip
   - template gen
   - bipWeaver
   - HDL Component Library

This diagram illustrates the design flow for HW-driven refinement, showing the interaction between application software, mapping, and architecture, leading to the generation of performance results and code.
HW-driven refinement – Building the Application SW Model

1. **Input**
   - Application SW
   - DOL

2. **System Model Generation**
   - Application SW Model
     - dof2bip
     - Application SW SW Model
     - Native BIP Simulation
   - Mapping
   - Architecture
     - Translation
     - HW Architecture Model
   - System Model
     - Transformation
     - Instrumentation: API, Observer injection
     - Instrumented System Model
     - Native BIP Simulation
     - Performance Evaluation

3. **Code Generation**
   - Code Generation
   - Performance Results
   - Multi-threaded application code
   - HDS Code

4. **HW Component Library**
   - HDS Component Library
HW-driven refinement – Building the Application SW Model

Every process and every SW channel are independently translated to atomic components in BIP.

Connectors are generated from connections of the process network generator.

```c
#define IN 1
#define OUT 2
typedef struct _local_states {
    int index;
    int len;
} Square_state;
void square_init(DOLProcess *p) {
    p->local->index = 0;
    p->local->len = LENGTH;
}
int square_fire(DOLProcess *p) {
    float I;
    if (p->local->index < p->local->len) {
        DOL_read((void*)IN, &i, sizeof(float), p);
        i = i*i;
        DOL_write((void*)OUT, &i, sizeof(float), p);
        p->local->index++;
    }
    if (p->local->index >= p->local->len) {
        DOL_detach(p);
        return -1;
    }
    return 0;
}
```
HW-driven refinement – Building the HW Model
Collection of hw-processor, memory and bus components connected as defined in the architecture
- HW-processor and HW-memory are placeholders
- uses HW component library
HW-driven refinement – Building the System Model

Application SW - Mapping - Architecture

DOL

Input

Translation

Application SW Model

Native BIP Simulation

bipWeaver

System Model

HW Component Library

HdS Component Library

Template gen

HW Architecture Model

Performance Evaluation

Instrumentation: API, Observer injection

Instrumented System Model

Native BIP Simulation

Performance Results

Code Generation

Multi-threaded application code

HdS Code

Performance Evaluation

Multithreaded application code

HdS Code
- Transformations defined by the mapping specify how to fill up the HW model
  - fully preserve functional behavior
  - use HdS component library

- Transformation on sw model:
  - Splitting SW-channels
  - Breaking atomic read/write
  - Adding interactions with HW-CPU-Scheduler
  - FIFO buffers mapped to memory

Transformations defined by the mapping specify how to fill up the HW model
- fully preserve functional behavior
- use HdS component library

Transformation on sw model:
- Splitting SW-channels
- Breaking atomic read/write
- Adding interactions with HW-CPU-Scheduler
- FIFO buffers mapped to memory
The MJPEG decoder

- reads a sequence of frames and displays the decompressed frames
- is described as a process network with five processes and nine communication channels
A simplified Multi-Processor ARM (MPARM)
- Five identical tiles and a Shared Memory connected via a Shared Bus.
- Tiles contain a CPU connected to its Local Memory via a Local Bus.
- CPU frequency: 200 Mhz.
- Access times:
  - 2 CPU cycles for local memory
  - 6 CPU cycles for shared memory
## HW-driven refinement – MJPEG decoder: Results

### Process mapping table

<table>
<thead>
<tr>
<th>Mapping</th>
<th>ARM1</th>
<th>ARM2</th>
<th>ARM3</th>
<th>ARM4</th>
<th>ARM5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping1</td>
<td>all</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping2</td>
<td>SS, SF, IQ</td>
<td>MF, MS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping3</td>
<td>SS, SF</td>
<td>IQ, MF, MS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping4</td>
<td>SS, SF</td>
<td>IQ</td>
<td>MF, MS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping5</td>
<td>SS, MS</td>
<td>SF</td>
<td>IQ</td>
<td>MF</td>
<td></td>
</tr>
<tr>
<td>Mapping6</td>
<td>SS</td>
<td>SF</td>
<td>IQ</td>
<td>MF</td>
<td>MS</td>
</tr>
<tr>
<td>Mapping7</td>
<td>SS, SF</td>
<td>IQ</td>
<td>MF, MS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping8</td>
<td>SS</td>
<td>SF</td>
<td>IQ</td>
<td>MF</td>
<td>MS</td>
</tr>
</tbody>
</table>

### SW-Channel mapping table

<table>
<thead>
<tr>
<th>Mapping</th>
<th>Shared</th>
<th>LM1</th>
<th>LM2</th>
<th>LM3</th>
<th>LM4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mapping1</td>
<td>all</td>
<td>all</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping2</td>
<td>C6, C7</td>
<td>C1, C2, C3, C4, C5</td>
<td>C8, C9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping3</td>
<td>C3, C4, C5, C6</td>
<td>C1, C2</td>
<td>C7, C8, C9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping4</td>
<td>C3, C4, C5, C6, C7</td>
<td>C1, C2</td>
<td></td>
<td>C8, C9</td>
<td></td>
</tr>
<tr>
<td>Mapping5</td>
<td>all</td>
<td>all</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping6</td>
<td>all</td>
<td>all</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping7</td>
<td>C6, C7</td>
<td>C1, C2, C3, C4, C5</td>
<td>C8, C9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mapping8</td>
<td>C1, C2</td>
<td>C3, C4, C5, C6</td>
<td>C7</td>
<td>C8, C9</td>
<td></td>
</tr>
</tbody>
</table>
Mapping (1) gives the worst computation time as all processes are mapped to a single processor.

The communication overhead is reduced if we distribute sw-channels to the local memories of the processors.
As more channels are mapped to the local memory, the shared bus contention is reduced. However, this might increase the local memory contention, as is evident for mapping (8).
- Component-based Construction

- Correctness-by-Construction

- The BIP Component Framework
  - Modeling Interactions
  - Modeling Priorities
  - Compositional Verification
  - HW-driven Refinement
  - Distribution-driven Refinement

- 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

Correct-by-construction translation of BIP models 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
Before reaching a ready state, the set of the enabled ports is sent to the Engine
From a ready state, await notification from the Engine indicating the selected port
Distribution-driven Refinement – 3-Layer Architecture

Distributed Mutual Exclusion Protocol

Interaction Protocol for $I_1$

Interaction Protocol for $I_2$

Interaction Protocol for $I_3$

Distributed Execution Engine

Distributed Implementation
Distribution-driven Refinement – Design Flow

Conflict Resolution Protocol

Partitioning of Interactions

Partitioning of Components

Sockets/C++ + Code

Code Generator

MPI/C++ Code

Component 1  129.2.2.1  Core1
Component 2  129.2.2.1  Core3
Component 3  129.2.2.1  Core2

Interaction Prot. \( \alpha_1 \alpha_2 \)

Interaction Prot. \( \alpha_3 \alpha_4 \)

Dining Philo. Conflict Resolution Protocol

Dining Philo. CRP

Dining Philo. CRP

Core1  Core2  Core3  Core4

Core1  Core2  Core3  Core4

CHIP

CHIP

\( \alpha_1 \alpha_2 \)

\( \alpha_3 \alpha_4 \)
Priorities:
BIP model with priorities can be transformed into an equivalent model without priorities. The same implementation principle can be applied.

Optimization issues:
- Building a correct snapshot of the system state is possible, but induces a lot of communication
  - Knowledge-based optimization by detecting false conflicts
  - Optimizing observability of interaction protocols (for priorities)
- 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
 COMPONENT-BASED CONSTRUCTION

CORRECTNESS-BY-CONSTRUCTION

THE BIP COMPONENT FRAMEWORK
- Modeling Interactions
- Modeling Priorities
- Compositional Verification
- HW-driven Refinement
- Distribution-driven Refinement

DISCUSSION
Too much of research in software engineering, systems, formal methods, etc. never made it in practice because it assumed a "design from scratch" approach and correctness-by-checking can only partially contribute to enhancing trustworthiness and optimality. It is limited to systems and properties that can be formalized and checked efficiently, e.g., functional properties of SW components.

Correctness-by-construction does not suffer scalability limitations—correctness for free. It requires formalization of system design as a disciplined incremental component-based process.
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