The BALBOA Project Team

- UC San Diego
  - Rajesh Gupta, Frederic Doucet, Sudipta Kundu, Jeff Namkung, Nick Savoiu (UCI)
- Virginia Tech
  - Sandeep Shukla, Hiren Patel, Gaurav Singh, Said Suhaib
- INRIA/IRISA, France
  - Jean-Pierre Talpin, David Berner
- TIFR, Mumbai, India
  - R. K. Shyamsundar
- Intel, Conexant/Mindspeed, Qualcomm, ST Micro
  - Eric Debes, Mojy Chian, Suhas Pai, Ramesh Chandra
HLM: A personal journey

- It all started as a circuit designer in SC4 c. 1986
  - Life was “Simple”
    - Simulation tool reproduced hardware behavior faithfully
    - Circuits hooked together: modularity and abstraction
    - Designer design automation focused on methodological innovations (split runs, timing calculators, sanity checks)
    - Real simple handoff (of printed C-size sheets)
    - Local verifiability and updates through back annotations
  - Then things changed
    - Design became data, and data exploded
    - Programming paradigm percolated down to RTL
    - Designers opened up to letting go of the clock boundary
- HDL = HLL + Concurrency + Timing + Reactivity + Structure
  - HardwareC, Radha-Ratan, Scenic, BALBOA
Chip Modeling & Building

- Methodological issues are increasingly at the junction of chip and embedded system design

1. Build components
   - model, synthesize, verify
     - Specification-based designs
   - (Automated) synthesis strategies to handle complexity

2. Build systems from components
   - architect, compose, validate
     - Platform-based designs
   - Design reuse, composition, co-simulation strategies

Momentum in applying HLL to HLM
1 Structural Abstraction
   - provide a mechanism for building larger systems by composing smaller ones

2 Reactive programming
   - provide mechanism to model non-terminating interaction with other components
     - e.g., watching (signal) and waiting (condition)
   - exception handling

3 Determinism
   - provide a “predictable” simulation behavior

4 Simultaneity
   - model hardware parallelism, multiple clocks
HLM Enablers

- “Virtualization” of IP blocks through smarts in object oriented (and library based) modeling of system components
  - IP blocks as part of language level libraries
- Virtual system architectures as abstractions of platforms

- Advances in verification techniques
  - HW verifications smarts beginning to drive PL design

- SystemC treading down the path synchronous languages have been before
  - and facing the same problems (solutions)
  - we will discuss one of these problems: causality loops
Compositionality can be achieved

- Component ‘wrappers’
  - Automatic and manual
  - Scripting languages and their integration to modeling languages: SWIG, SysPy (SystemC+Python)
- While integration for simulation is doable
  - Problem becomes immense if model substitutability is concerned
  - Ensuring correctness is not trivial
- Compositional frameworks that rely on models, specifications
  - Heterogeneous MOCs, meta-models
Structure is fundamental to chip design

- Module as a top-level class
- Member functions:
  - model blocks
  - create compound blocks
  - connect component objects
  - set parameters

- A glorified schematic entry
  - set design [new Module]
  - set C0 [$design Component]
  - $design connect C0 C1
  - $design attach_list
  - $design copy_interface
  - $design attach_behavior
  - ...


BALBOA Project

Vision: Focus on Compositionality
- Composability can be achieved through polymorphic interfaces and mixed compiled and interpreted programming components.
- Ensure correctness of the compositional process through static and dynamic validation methods.
- Drive compositionality through advances in interface refinement and substitution.

Project Goals: Algorithms and techniques for
- Composition of IP components for system-level designs
  - Addresses compositional guidance provided by virtual system architectures
- Automated selection of correct IP components and interfaces
  - Addresses port polymorphism and interface adaptor synthesis
- Formal compatibility checks and creation of simulation models
  - Type abstractions, model checking and automated creation of correct interfaces and simulation models.
BALBOA CCF

- **A composition environment**
  - Built upon existing class libraries, to add a software layer for manipulation and configuration of C++ IP models
  - Ease software module connectivity
  - Run-time environment structure

- **A SW architecture that enables**
  - Composition of structural and functional information

- **Current state**
  - SystemC + NS2 + ISS + OS services
Key Technical Decisions

- A layered development and runtime environment
  - **Functionality:** describe & synthesize
  - **Structure:** capture & simulate

- Use an interpreted language for
  - Architecture description
  - Component integration

- Use compiled models for
  - Behavioral description, simulation

- Automatically link the two domains
  - Through a “split-level” interface

- Automatic code “wrapper” generation
  - For component reuse.
Language Layer: Compiled

Component Implementation in C++

- To execute the modeled behavior
- Can use object structure to replicate modeled structures
- Use modeling class library (in SystemC, C++) for
  - Concurrency
  - Bit-level data types
  - Model of time (variants, BFM, ISS etc.)
  - Model of structure
  - OS, Middleware services, abstractions
- Components are implemented by a component library designer, modeling *plus C++ programming*
Language Layers: CIL

- Script-like language based on Object Tcl
- Compose an architecture
  - Instantiate components
  - Connect components
  - Compose objects
  - Build test benches
- Introspection
  - Query its own structure
- Loose typing
  - Strings and numbers

Producer P
Consumer C
Queue Q

P query attributes
⇒ queue_out
C query attributes
⇒ queue_in

P.queue_out query methods
⇒ bind_to read

P.queue_out bind_to Q
…
Language Layers: BIDL

- Describe the component for usage with the CIL
- Exports the interface and internals details:
  - Attributes, Methods
  - Relationships, Non-functional properties
- Configure a Split-Level Interface (SLI)
  - A custom wrapper for manipulation of the C++ compiled object by the CIL
- Generate the Type System Extensions
  - For the CIL introspection and type inference
- (Defines the “meta-level” for reflection)
Internal Component Architecture

CIL Commands

- Split level Interface
- Interpreted OTcl class with variables and methods

Type Adapter Bridge

- Type system information
- Internal compiled object

C++ Objects Interactions
Internal Component Architecture

- **Split-level interface**
  - Link between interpreted and compiled domain
  - Abstracts and manage the underlying C++ object
  - Implements heuristic for type inference
  - Maintains type checking for correct by construction validation
  - Implement the composition model, introspection and reflection
- **Type adapter bridge**
  - Provides a proxy to the internal C++ object
  - Specific for each C++ type
  - Generated by the BIDL
- **Type system information**
  - Specific to the C++ class, generated by the BIDL
- **Interpreted variables and methods**
  - The system architect can add interpreted parts to the component
Language and Run-time Layers

Language

- CIL
- BIDL
- C++

Tools

- Introspection
- GCC

Run-time structure

- Interpreter
- Split Level Interfaces
- Compiled objects

SLI/Type system extension
Reflection
Example

# Instantiate components
Adder a
Register r
connect a.z to r.in

# type introspection
a query type
⇒ Adder

a query type parameters
⇒ DATATYPE (bv10)
a query implementation
⇒ add_fast<bv10>

a query ports
a b cin z cout

# Declare interface
Component Adder/interface {
    Inport a
    Inport b
    Inport cin
    ... Type parameter (DATATYPE)
}

# Declare implementation
Component Adder/Implementation {
    DATATYPE (bv10): add_fast<bv10>
    ... BIDL
}

template<class T>
class add_fast: public sc_module {
    sc_in<bv10> a;
    ... C++
};
Type System

- Compiled types are “weakened” in the CIL
  - Data types are abstracted from signal and ports

- Algorithm for data type inference
  - If a component is not typed in the CIL
    - The SLI delays the instantiation of the compiled internal object
    - Interpreted parts of the component are accessible
  - Verify if types are compatible when a relationship is set
    - If a compatible type is found, the SLI allocates the internal object and sets the relationship
    - If not, the link command is delayed until the types are solved
An adder:

is polymorphic because its ports can have many type mappings:

$$ports(c_1) : \text{int} \quad X \quad \text{int} \quad X \quad \text{bool} \quad X \quad \text{int} \quad X \quad \text{bool}$$

$$ports(c_2) : \text{bv8} \quad X \quad \text{bv8} \quad X \quad \text{bool} \quad X \quad \text{bv8} \quad X \quad \text{bool}$$

$$ports(c_3) : \text{bv16} \quad X \quad \text{bv16} \quad X \quad \text{bool} \quad X \quad \text{bv16} \quad X \quad \text{bool}$$

The $dt_p$ mapping function has 3 choice in assigning the ports to compiled types!

**Mapping can be viewed as an IP selection**
Subtyping & Software Components

Substitutability (polymorphism):

If we replace A by B in the system, will correctness be maintained? (may be a different abstraction, language, required environment)

Problem gets complex as the notion of substitutability is enhanced

Use behavioral types as containers of sequential behavior at the interfaces
Ensuring Compositional Correctness

- Syntactical correctness does not guarantee correct behavior, let alone desired behavior
- How can we compose IP blocks in SystemC so that the system can be further composed
  - (associativity if preserved permits further compositions incrementally)
- Simulation correctness does not imply logical correctness due to
  - Non-coverage (or defining) the complete input environment (input nondeterminism)
  - Behavioral nondeterminism
  - Compositional anomalies: cycles, scheduling order dependencies, 2-level (delta) timing models
  - Problem with delta timing: infinite actions in a finite time (Zeno’s Paradox, Thompson’s lamp)
- How can we carry further with verification methods?
Two-level Timing Models

- Use of delta cycles (like in most HDLs) helps order events that happen within a given scheduling step to preserve deterministic behavior.
- Event notification can be immediate, timed or at delta cycles.
- Delta cycles, even with limited testing for absence of a signal could lead causal cycles.
Example: Checking for event absence forms a cycle

SC_MODULE(M1) {
    sc_in<bool>  e1;
    sc_in<bool>  e3x;
    sc_out<bool> e3;
    sc_out<bool> e1x;

    SC_CTOR(M1) {
        SC_METHOD(p1);
        sensitive << e1 << e3x;
    }
    void p1() {
        if (!e3x.event())
            e3.write(!e3.read());
        e1x.write(!e1x.read());
    }
};

Cyclic loop: three processes themselves
Nondeterministic Behavior

- **non-determinism:**
  - for an input trace, it can be possible to observe different output traces

- **consequence:**
  - can cause synchronization problems
  - missed events, different values, etc

- **where does it come from: four possible sources**
  - mix of concurrency with shared variables
  - mix of concurrency with immediate event notification
  - non-deterministic software models with immediate event notifications
  - un-initialized signals/variables
Nondeterministic Behavior

Event notification can be missed depending on which process gets scheduled first at some arbitrary step.

```c
SC_MODULE(M1) {
    sc_event e;
    int data;

    SC_CTOR(M) {
        SC_THREAD(a);
        SC_THREAD(b);
    }
    void a() {
        data=1;
        e.notify()
    }
    void b() {
        wait(e)
    }
};

SC_MODULE(M2) {
    sc_event e;

    SC_CTOR(M) {
        SC_THREAD(a);
        SC_THREAD(b);
    }
    void a() {
        wait(10, SC_NS)
        e.notify();
    }
    void b() {
        wait(10, SC_NS);
        wait(e);
    }
};
```

at the initial step

at some arbitrary step
Scheduler Dependency

This runs to completion and execute the sc_stop statement
Scheduler Dependency

```cpp
sc_event e;

SC_MODULE(M1) {
    SC_CTOR(M1) {
        SC_THREAD(a); 
    }
    void a() {
        e.notify();
    }
};

SC_MODULE(M2) {
    SC_CTOR(M2) {
        SC_THREAD(b);
    }
    void b() {
        wait(e);
        sc_stop();
    }
};

int sc_main() {
    M1 m1("m1");
    M2 m2("m2");
    sc_start(10);
    return 1;
}
```

inverting the instantiation order makes M2 miss e and block forever

```cpp
int sc_main() {
    M2 m2("m2");
    M1 m1("m1");
    sc_start(10);
    return 1;
}
```

Not really a structural specification!
Of course, we can turn ND to Deterministic SystemC programs

```c
sc_event e;

SC_MODULE(M1) {
    SC_CTOR(M1) {
        SC_THREAD(a);
    }
    void a() {
        e.notify_delayed();
    }
};

SC_MODULE(M2) {
    SC_CTOR(M2) {
        SC_THREAD(b);
    }
    void b() {
        wait(e);
        sc_stop();
    }
};
```

Delayed notification (delta events) can be used to make non-deterministic behavior deterministic.

The delivery of event is delayed until next cycle, introducing a partial order between concurrent events.

However, are these logically correct?
BALBOA Approach

- **A firm semantics (SOS style)**
  - Clear unambiguous understanding of IP block behaviors
- **Static Analysis of SystemC**
  - Check for logical correctness
- **Compositional (modular) Verification Abstractions**
  - Reduces complexity, but requires strong formal foundation
- **Tools for analysis and synthesis**
  - Heterogeneous models, multiple clocks etc
SOS for SystemC

Events emitted

\[(e.\text{notify}(), \Sigma) \xrightarrow{e,\emptyset,\emptyset,\emptyset,1} (\_, \Sigma)\]

termination flag

\[(e.\text{notify\_delayed}(), \Sigma) \xrightarrow{\emptyset,\emptyset,\emptyset,1} (\_, \Sigma)\]

environment

\[(\text{wait}(e), \Sigma) \rightarrow (\text{pause}; \, \text{wait\_cnt}(e), \Sigma)\]

semantic rules

\[e \notin E\]

syntactic rule

\[(\text{wait\_cnt}(e), \Sigma) \xrightarrow{\emptyset,\emptyset,\emptyset,0} (\text{wait\_cnt}(e), \Sigma)\]

If e is not in the environment, wait more

\[e \in E \land e \notin R_P,\]

and identify conditions that lead to compositional anomalies.

\[(\text{wait\_cnt}(e), \Sigma) \xrightarrow{\emptyset,\emptyset,\emptyset,1} (\_, \Sigma)\]

statement by statement, succession (and merging) of environments...

\[(\text{pause}, \Sigma) \xrightarrow{\emptyset,\emptyset,\emptyset,0} (\_, \Sigma)\]

31
Status and plans

- We have SOS and a behavioral type system in place to enable capture of SystemC programs for compositional verification
- Currently working on software architecture to allow
  - Capture and automatic translation of SystemC code
  - Generation of proof obligations (have hand examples working)
- Working on algorithm for refinement checking for simulation efficiency
  - Using model checking to prove flow invariance
Summary

♦ The current movement towards HLM through programming advances holds the promise of modeling and methodology convergence from chip design to embedded systems (software) design
  - Language-level modeling advances now touching new compositional abilities through innovations in design patterns and infrastructure capabilities

♦ However, such advances go hand-in-hand with advances in verification and synthesis tools
  - Yet, good IP-model composability still very much out of reach

♦ BALBOA CCF is a prototype for dynamic composition of IP blocks and their validation through static and dynamic verification.
Related Work

- **Software architecture**
  - Architecture description languages: Wright, EXPRESSION, xADL
    - Component-configuration-connection model
  - Component frameworks
    - Ptolemy
      - Type system in full lattice structure, solving in linear time
      - Interoperation semantics, top down design, Balboa= bottom-up
    - TIMA’s Colif, IBM Coral, JavaCAD
      - Architectural inference, and component selection according to constraints
  - Platform-based design
    - Architectural modeling
  - IP Chinook:
    - Compositional specification with modal processes
    - Weave in new features in the system
    - Problem: no verification
  - Metropolis
    - Formal foundation to system design
    - Not compositional
    - Basic equivalence verification only

- **Split-level programming**
  - Network Simulator (NS)
    - Separate composition concerns from programming
  - Wrapper generation
    - SWIG, CDL (component description languages)
Dynamic Type and Static Type

- **Dynamic type**
  - it defines the transition system of the interface

- **Static type**
  - A static interface of the same code is an abstraction of the dynamic interface, by
    - abstracting the transitions into clock relations,
    - taking closure of the clock relations, and
    - taking transitive closure of the scheduling relations.

- **Verification is through subtype checking (inferred against specified)**
  - Subtype checking in dynamic interface types can be checked by simulation relations
  - Subtype checking for static interfaces can be done using checking trace inclusion
Notationally a minimalistic STS

A multi-clock based type system

<table>
<thead>
<tr>
<th>(value)</th>
<th>v ::= 1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>(location)</td>
<td>l ::= x₀</td>
<td>x</td>
</tr>
<tr>
<td>(clock)</td>
<td>e ::= 0</td>
<td>x = v</td>
</tr>
<tr>
<td>(types)</td>
<td>P ::= (l = v)</td>
<td>x</td>
</tr>
</tbody>
</table>

- Type = set of traces on its signals that satisfy all clock equations in its description.

- Two modules are composable if inference can produce a type for the composition.
Example

```c
void epc::ones () { sc_int<16> idata = 0, ocount = 0;
    while true { wait (epc.lock);
        idata = epc.data;
        ocount = 0;
        while (idata != 0) { ocount = ocount + (idata & 1);
            idata = idata >> 1; }
        epc.count = ocount;
        notify (epc.lock); } }

T2 = ocount;
T3 = T1 & 1;
ocount = T2 + T3;
idata = T1 >> 1;
goto L2;
L3: epc.count = ocount;
    notify (epc.lock);
    goto L1;
```

L1: wait (epc.lock);
    idata = epc.data;
    ocount = 0;
goto L2;
L2: T1 = idata;
    T0 = T1 == 0;
    if T0 then L3;
    :
Behavioral type assignment

Simple SSA block

L2:T1 = idata;
T2 = T1 == 0;
if T2 goto L3;
T3 = icount;
T4 = T1 & 1;
icount = T3 + T4;
idata = T1 >> 1;
goto L2;

Its invariants

L2⇒T1 := idata
T2 := (T1 ≠ 0)
T2 ⇒ L3′
¬T2⇒T3 := icount
T4 := T1&1
icount′ := T3 + T4
idata′ := T1 >> 1
L2′

State-full behavioral types

L2⇒T1 := idata
T2 := (T1 ≠ 0)
T2 ⇒ L3′
¬T2⇒T3 := icount
T4 := T1&1
icount′ := T3 + T4
idata′ := T1 >> 1
L2′

State-less abstraction

L2⇒T1 := idata
T2 := T1
T1 ⇒ T0

L2⇒idata⇒T1

...
Example: Type annotation

Type $P$ of a block consists of synchronous composition of the type associated with every instruction in that block.

<table>
<thead>
<tr>
<th>code</th>
<th>clocks</th>
<th>scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>L2: $T1 = \text{idata};$</td>
<td>$x_{L2} \land \text{idata} \Rightarrow \hat{T1}$</td>
<td>$x_{L2} \Rightarrow \text{idata} \rightarrow T1$</td>
</tr>
<tr>
<td>$T0 = T1 == 0;$</td>
<td>$x_{L2} \land \hat{T1} \Rightarrow T0$</td>
<td>$x_{L2} \Rightarrow T1 \rightarrow T0$</td>
</tr>
<tr>
<td>if $T0$ then $L3;$</td>
<td>$x_{L2} \land T0 \Rightarrow x_{L3}$ (^{\text{(b)}})</td>
<td>$x_{L2} \Rightarrow T0 \rightarrow x_{L3}$ (^{\text{(c)}})</td>
</tr>
<tr>
<td>$;$</td>
<td>$;$</td>
<td>$;$</td>
</tr>
<tr>
<td>goto $L2;$</td>
<td>$x_{L2} \setminus x_{L3} \Rightarrow x'_{L2}$ (^{\text{(a)}})</td>
<td>$;$</td>
</tr>
</tbody>
</table>

Clocks: $x_{L2}, x_{L3}, x_{L2} \setminus L3$

Branches: $T1^\wedge, \ldots$

Data: $T1^\wedge, \ldots$
Static interface abstracts delayed transitions by clock relations.

Type inference function defined by induction on the formal syntax of a Program. Associate a clock with each block to model activation, return.
Type Inference

Table 3: Type inference

<table>
<thead>
<tr>
<th>code</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1: <code>wait (epc.lock);</code></td>
<td>$x_{L1} \land (\text{lock} \neq \text{lock}') \Rightarrow \hat{y}_1$</td>
</tr>
<tr>
<td>\vdots</td>
<td>$x_{L1} \setminus \hat{y}<em>1 \Rightarrow x</em>{L2}' \ldots$</td>
</tr>
<tr>
<td>goto L2;</td>
<td>$\hat{y}<em>1 \Rightarrow x</em>{L2}'$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>code</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3: <code>epc.count = ocount;</code></td>
<td>ocount $\land x_{L3} \Rightarrow \text{count}$</td>
</tr>
<tr>
<td>notify <code>epc.lock);</code></td>
<td><code>goto L1;</code></td>
</tr>
<tr>
<td>\vdots</td>
<td>$x_{L3} \Rightarrow \text{lock}' = \neg \text{lock}$</td>
</tr>
<tr>
<td>\vdots</td>
<td>$x_{L3} \Rightarrow x_{L1}'$</td>
</tr>
</tbody>
</table>

(1) $I[\llbracket \text{block}; \text{pgm} \rrbracket] = I[\llbracket \text{block} \rrbracket]^{\text{L}}_L \mid I[\llbracket \text{pgm} \rrbracket]

(2) $I[\llbracket \text{stm}; \text{block} \rrbracket]^{\text{e}}_L = \text{let } (P)^{e_1} = I[\llbracket \text{stm} \rrbracket]^{\text{e}}_L \text{ in } P \mid I[\llbracket \text{block} \rrbracket]^{\text{e}_1}_L$

(3) $I[\llbracket \text{if } x \text{ then } L_1 \rrbracket]^{\text{e}}_L = \llbracket G_L(L_1, e \land x) \rrbracket^{e \land -x}_L$

(4) $I[\llbracket x = f(y^*) \rrbracket]^{\text{e}}_L = \llbracket E(f)(xy^*e) \rrbracket^{e}_L$

(5) $I[\llbracket \text{notify } x \rrbracket]^{\text{e}}_L = \llbracket e \Rightarrow (x' = -x) \rrbracket^{e}_L$

(6) $I[\llbracket \text{wait } x \rrbracket]^{\text{e}}_L = \llbracket e \setminus (x \neq x') \Rightarrow \hat{y} \setminus \hat{y} \Rightarrow x_{L1}' \rrbracket^{\hat{y}}_L$

(7) $I[\llbracket \text{goto } L_1 \rrbracket]^{\text{e}}_L = \llbracket e \Rightarrow x_{L1}^{\text{exit}} \parallel G_L(L_1, e) \rrbracket$

(8) $I[\llbracket \text{return } \rrbracket]^{\text{e}}_L = \llbracket e \Rightarrow x_{L}^{\text{exit}} \parallel e \Rightarrow -x_{L}' \rrbracket$

(9) $I[\llbracket \text{throw } x \rrbracket]^{\text{e}}_L = \llbracket e \Rightarrow x_{L}^{\text{exit}} \parallel e \Rightarrow \hat{x} \rrbracket$

$I[\llbracket \text{catch } x \text{ from } L \text{ to } L_1 \text{ using } L_2 \rrbracket]^{\text{e}}_L = G_L(L_2, \hat{x} \land x_{L2}^{\text{exit}}) \parallel G(L_2, x_{L2}^{\text{exit}})$