High Level Modeling and Component Compositions

Understanding and Appreciating Structure

Rajesh Gupta
University of California, San Diego.

Intel, November 11, 2005
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
- 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
µSystems Design Context

- **Modeling and design of embedded systems**
  - build "complete" system models
  - explore potential design space
- **Platform-based design**
  - a good solution that can be customized and configured
  - provides known communication architectures
- **Known problems**
  - Component complexity: diversity, interactions
- **Obvious approach**: 
  - raise abstraction, increase reuse of design and verification
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 + Structure**

- HardwareC, Radha-Ratan, Scenic, BALBOA
**µSystems Modeling, Design & Validation**

- 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**
HLM Semantic Necessities

1. **Concurrency**
   - model hardware parallelism, multiple clocks

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. **Structural Abstraction**
   - provide a mechanism for building larger systems by composing smaller ones
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
  - One way to think about these is via MOCs and 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
- 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
1. Composition of IP components for system-level designs
   - Addresses compositional guidance provided by virtual system architectures
2. Automated selection of correct IP components and interfaces
   - Addresses port polymorphism and interface adaptor synthesis
3. Formal compatibility checks and creation of simulation models
   - Type abstractions, model checking and automated creation of correct interfaces and simulation models.
BALBOA CCF Brief Recap

Internal Component Architecture

- Component Integration, CIL
- Split-Level Interface/BIDL
- C++, SystemC
- Compiled
- Interpreted
- System designer

CIL Commands

- Split level Interface
- Interpreted OTcl class with variables and methods
- Type Adapter Bridge
- Type system information
- Internal compiled object

C++ Objects Interactions
Example

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

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

```
Language and Run-time Layers

Language

CIL

BIDL

C++

Tools

Introspection

Interpreter

Split Level Interfaces

Reflection

GCC

GCC

Run-time structure

Compiled objects
Specification in BALBOA

- **Structural specification**
  - components, channels, events, shared variables, connections

- **Behavioral specification**
  - scenarios of observable event sequences

- **Components implementations**

Correctness through type inference and type checking.
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.
- However, delta cycles combined with limited pre-emption, testing for absence of a signal can lead to nondeterministic behaviors.
Example: Checking for event absence forms a cycle

```cpp
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

```
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

```c
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;
}
```

This runs to completion and execute the sc_stop statement
Scheduler Dependency

```c
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

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

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?
Start with Structured Operational Semantics…

- For every syntactic SystemC statement $stmt$, a clean rule to derive observational behavior:

\[
(stmt, \sigma) \xrightarrow{E_O, b} (stmt', \sigma')
\]

where
- $E_I$ is the triggering environment
- $E_O$ is the output environment
- $b$ denotes if the statement terminates in the current instant
- $\sigma$: denotes the state (values assigned to the variables)

- The rules are used to produce a transition system whose language is the observable sequences

...and identify conditions that lead to compositional anomalies.
# SOS Rules for Event Communication

Produce behavior of the form: $E_I \sigma E_O$

<table>
<thead>
<tr>
<th>Rule Type</th>
<th>Description</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>(event-notify)</td>
<td>e.notify() $\xrightarrow{e,\emptyset,\emptyset,1}$ (e in the next environment)</td>
<td>$e$ in the next environment</td>
</tr>
<tr>
<td>(wait-syntactic)</td>
<td>$\text{wait}(e) \xrightarrow{\text{syn}} \text{pause}; \text{wait}(e)$</td>
<td>wait for the next event $e$</td>
</tr>
<tr>
<td>(wait-event-block)</td>
<td>$\frac{e \notin E \land \neg Rv}{\text{wait}(e) \xrightarrow{0} \text{wait}(e)}$</td>
<td>If $e$ not in environment, wait for next instant</td>
</tr>
<tr>
<td>(wait-event-unblock)</td>
<td>$\frac{e \in E \land \neg Rv}{\text{wait}(e) \xrightarrow{1} -}$</td>
<td>If $e$ is in the environment, reduction terminates</td>
</tr>
</tbody>
</table>
**SOS Rules for Sequential Composition**

Produces a behavior of the form: \( E_I \sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 E_O \)

**Assignment**

\[(x := e, \sigma) \xrightarrow{\frac{1}{E}} (_-, \sigma'[x/e])\] Changes the state \(\sigma\) into \(\sigma'\)

**Sequential-composition-1**

\(P_1\) blocks, the whole sequential composition blocks

\[
\begin{align*}
(P_1, \sigma) \xrightarrow{E_O, E_O^\delta, L_1, 0} (P_1', \sigma'_1) \\
(P_1; P_2, \sigma) \xrightarrow{E_O, E_O^\delta, L_1, 0} (P_1'; P_2, \sigma'_1)
\end{align*}
\]

**Sequential-composition-2**

\[
\begin{align*}
(P_1, \sigma) \xrightarrow{E_O^1, E_O^\delta_1, L_1, 1} (P'_1, \sigma'_1) \\
\text{merge}(\langle E_{O_1}, E_{O_2} \rangle, \langle E_{O_1}, E_{O_2} \rangle, \omega) \\
(P_1; P_2, \sigma) \xrightarrow{E_O^1 \cup E_O^2, E_O^\delta_1 \cup E_O^\delta_2, L_1 \cup L_2, b_2} (P'_2, \sigma'_2)
\end{align*}
\]
Compositional Type Checking

- **Types: classify artifacts into sets**
  - “all these cars have 2 doors”
  - “all these cars have keyless entry”
  - In PL, a type defines the domain over which a variable ranges (Domain, Operations)
  - These are fundamentally structural

- **Different notions of behavioral types**
  - Behavioral types describe patterns of interactions
  - Notion generally built upon models of component interfaces
    - An interface type can be a process, a set of sequences, or just a set of observables
    - “events, sequences and transactions”
Types in BALBOA CCF

- A component is a unit of encapsulation
  - Structural types (syntactic types)
  - Behavioral types (semantic types)

What can we observe at the interface of the component?

We use the word “behavioral type” in the sense of interface description.
Structural Types and Inferencing

- Structural types are classic syntactic types
  1. Composite structure
  2. Data type for ports, functions, parameters

- Structural type system used for:
  1. Check type constraints on all channel connections
  2. Fill in the abstracted syntactic details
  3. Find parameter values satisfying all the constraints
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(c₁) : int X int X bool X int X bool
ports(c₂) : bv8 X bv8 X bool X bv8 X bool
ports(c₃) : bv16 X bv16 X bool X bv16 X 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.
Behavioral Types

*Scenario-based specification:* observation of the interactions of many components (cross-cutting the architecture).

*Behavioral type:* captures the part of the scenario, which is local to a component.

Describes what can be observed at the interface of a component.
Component Type

Component types as a pair of:

1. Interface type: with classical types (sets + operations)
2. Behavioral type: set of all possible observable sequences at the interface

Internally, component behavior is whatever one wants it to be as long as it respect the “type contract”
Defining New Component Types

From an interaction diagram: projection and conversion to an automaton

From a block diagram: syntactic translation

From a SystemC component: requires SOS
Example: Central Locking System

The control system for the CLS interacts with many components in the car.

Let us look at the specifications of these interactions.
Example: Central Locking System
Example: Central Locking System

Specification for “transfer driver ID”
Example: Central Locking System

At any times there is a crash, the doors should unlock!
(the specification should be verified to imply this property)
Example: Now combine scenarios to define the behavioral type of the controller.
Example: Central Locking System

Behavioral type for the controller

Scenarios make it simpler to specify this.
Example: CLS

Assume we have this following SystemC implementation of the controller:

Does it correctly implements the behavioral type of the controller?
Example: CLS

```c
void control_process() {
    while(true)
    // doors
        wait(unlock_kf);
        notify(unlock_lm);
        wait(ok);
        notify(lights_door_unlocked);
    ...
    // preferences
        notify(kf_get_id);
        wait(id.default_event());
        preset.write(id.read());
    }
```
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 behavioral type inferencing
  - where the design is entered in an interpreted domain
  - while at the same time avoiding need to separate languages and description by using a layered software architecture and automatic generation of SLI wrappers.
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)