Meta Modeling for Component Compositions

A “Hardware” Guy’s View

Rajesh Gupta
University of California, San Diego.
Introduced April 10, 1989
P446, 1 um, 25 MHz, 4" Wafers
1.2 M transistors

16.5 mm

Pad limited die:
200 pins
52 mm²
>1K dies/wafer
$5/part

50 mm², 50M, 1-10 GHz, 100-1000 MOP/mm², 10-100 MIPS/mW, 300 mm, K units/wafer, 20K wafers/month, $5
Cambrian Explosion in $\mu$Systems

- We have the silicon capacity to do...
  - Multiple cores
  - Multigrain “Programmable” circuit fabrics
  - Coprocessors and accelerators
  - Processor extensions: Short Vector SIMD, Media, Baseband, SDR, …

Stationary Devices

Mobile Devices

Sensor Devices
Computing Efficiencies

- **Watt nodes: Home, Office, Car**
  - Compute intensive platforms
  - Reaching 1 Tops in 5-10W: 100-200 Gops/W
  - 100-1000x more efficient than today’s PCs
  - Programmability must, innovation from domain knowledge

- **MilliWatt nodes: Converged devices**
  - Wireless intensive: radios, networks, protocols, applications
  - Multimedia evolution to SVC leading to 9-36x more CPU than H.264
  - 10-hour battery operation, 1W for 10-100 Gops: 10-100 Gops/W
  - Combination of scaling and duty cycling, computing models
    - Semi houses have to move from components to domain specialization

- **MicroWatt nodes: Immortal devices, ad hoc networks**
  - < 100 microwatts for scavenging, 10 Mops: *very high peak* efficiencies
  - Approach limits on computation and communication
  - Aggressive duty cycling (<1%, 1bps-10kbps).
Coprocessing Is Currently A Favored Way to Improve Efficiencies: W Nodes

- **Automotive: Infineon VIP Platform**
  - 130 nm, 64 mm², 16 SIMD PEs, 200 MHz, OAK DSP, 37 kb eSRAM, 760 mW, 100 Gops @ 8b
  - Each mirror has a camera and a VIP that performs real-time safety related calculations on the image
  - Programming in OAK DSP using its video-related instructions
  - 16 64-bit SIMD processors, each handling 8x8 segment
  - 38 Gops/W

- **Philips Nexperia Platform: Viper2**
  - 50M, 130 nm, MIPS + 2 VLIW Trimedia, 250 MHz, 4W for 104 Gops
  - MIPS controls 60 coprocessors, plus VLIW
  - 26 Gops/W
Intrinsic Power Efficiency of Silicon Substrates

- At 130 nm nodes (ISSCC99 T. Classen)
  - MPU: 100 MOPS/W
  - FPGA: 1-2 GOPS/W
  - ASIC: 10-20 GOPS/W

- We are within 10x of efficiency requirements for custom ASICs (200 MOPS to 200 GOPS per Watt in 65nm)
  - (hardware muxed datapaths with local storage and hw thread control)

- But 500x behind when dealing with SW programmable systems
  - Unless, of course, notion of SW changes underneath..
  - Software development and software-dominated system design is the challenge
    - Particularly for mW nodes where platform architecture uses a mixture of computing fabrics.
Two Pain Points

1. Cost of Design (Verification):
   - architectural innovations versus implementation fabrics
2. How do we program this thing?
   - As node, as network.

SOC Design Cost Model

Courtesy: A. Kahng & ITRS.
Ultimately, it is about programming in building & using Silicon

- Cost reduction through reuse and abstraction: SW to the rescue?
  - “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

- Two fundamental challenges
  - C1 -- Capturing designer “black art”
  - C2 -- Ensuring compositionality of IP, ensuring correctness of compositions

- How to ensure programming for and programmability of Silicon microsystems?
  - Changes in programming models and/or programming language.
Modeling Hardware: 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
Shifts in PM and PL

- **Thesis**: A dramatic shift in PM is in progress!
- **Factors driving this shift**
  1. Rich and diverse computing fabrics (in silicon)
  2. Internet proliferation to end points: data and control integration over wide (and wireless) area with mobility
2. The Internet (= WAN) Influence

- Devices are becoming internet end points and accessing data through multi-level client-server hierarchies
- Yet, it is a
  - different kind of distributed system
  - different kind of data
  - different kind of database

Effect: Explosion in APIs
- XML APIs: for data integration
- Messaging APIs: for flow control across WAN
- S&P APIs: for access control

- Typically, APIs migrate to PL extensions
  - Though there are several emerging challenges and open questions on PL support for new data structures, control flows, assertions.
Emerging Capabilities in Programming: Meta Data Handling

- **Self-documenting/extensible “tags”**
  - Nested tags enable exchange of data (not just documents)
  - Data interchange is the basis for integration of services on the web
- **There is no distinction between data and schema**
  - Simplest XML is a labeled ordered tree with labels on nodes, and possible data values at the leaves.
- **Schema extracted through Data Type Definitions (DTDs)**
  - A DTD is an extended CFG with no terminals:
    - Nonterminals are tags in the XML parse tree
  - Not quite PL data type: Values are not constrained
    - Unordered things are difficult; Inability to separate type from name.
- **New flexible/extensible types and schemas**
  - E.g., regular expressions over trees
  - Ability to talk “about” data / queries through reflection

Example

```xml
<dealer> <UsedCars> <ad>
  <model>Honda</model><yr>92</yr>
</ad></UsedCars>
<NewCars> <ad>
  <model>Prius</model></ad></NewCars>
</dealer>
```

- root: dealer
- dealer ➔ UsedCars, NewCars
- UsedCars ➔ ad*
- NewCars ➔ ad*
- ad ➔ modelyear | model
<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

A → R → Q

Diagram showing a tree structure with nodes labeled R, Q, and A, with branch labels 1, 2, and 1.
Ultimately, it is about programming in building & using Silicon

- Cost reduction through reuse and abstraction: SW to the rescue?
  - “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

- Two fundamental challenges
  - C1 -- Capturing designer “black art”
  - C2 -- Ensuring compositionality of IP, ensuring correctness of compositions

- How to ensure programming for and programmability of Silicon microsystems?
  - Changes in programming models and/or programming language.
C1: IP Composition Need
Introspection Capabilities

- **Structural design**
  - #processes, triggering conditions, #ports, “plumbing”

- **Behavioral design**
  - Model of computation, state machines, patterns of interaction

- **Modeling dimensions**
  - Classes, and constructs for modeling semantics

- **Runtime and tool infrastructure**
  - Static simulation information (setup, coordination/callbacks, tb)
  - Dynamic information (number and type of events in queue)

- ... and so on.

- **Several ways to achieve it:**
  - Component design using patterns, subtyping, composition techniques, using IDLs
Reflection and Introspection: A HW Guy’s Way of Looking At It

- **Component:**
  - A unit of re-use with an interface and an implementation

- **Meta-information:**
  - Information about the structure and characteristics of an object

- **Reification:**
  - A data structure to capture the meta-information about the structure and the properties of the program

- **Reflection:**
  - An architectural technique to allow a component to provide the meta-information to himself

- **Introspection:**
  - The capability to query and modify the reified structures by a component itself or by the environment
Our Approach

1. Start with SystemC descriptions of IP blocks
   - Multi-level (RTL, TL) descriptions

2. Capture meta information of these IP into XML
   - Mostly structural information for now.

3. Generate library of ‘XMLized’ IP blocks
   - Schema to match datatype and protocol type information across IP blocks
   - Create DOM model and constraints for the library

4. Develop methods for IP selection, composition, verification, synthesis
   - Automated methods for IP instantiation, interface generation

IP Selection through an Introspective Composer
- IP matching and connection
  - Insertion of bridges
  - Validation of functionality
  - Create an executable specification
Structural Information

- **RTL-level**
  - SystemC ports (sc_in, sc_out, sc_inout and sc_clock)

- **TL-based**
  - SystemC interface descriptions (sc_interface)
  - Implementation of interfaces (sc_channel, sc_fifo, etc)
  - Interface ports

- **Module-level**
  - Module hierarchy
  - Process description (sc_method, sc_thread, sc_cthread)
  - Sensitivity List and Signal connectivity

- **General information**
  - Type declarations (data type declaration)
  - Type indirection (typedef, extern, #define)
  - Comments (Hints for reflection & constraining)
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.
Structural Compositions in Balboa

- A layered component 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
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

a.cin query type
bv<10>

# 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++
Design Example

Adaptive Memory Reconfiguration & Management

Communication and concurrency refinement

<table>
<thead>
<tr>
<th>Level of Abstraction</th>
<th>CIL #lines</th>
<th>Methods &lt; 30</th>
<th>Queues Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>&lt; 40</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt; 150</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code generation statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ratio of C++/CIL code ~ 10:1</td>
</tr>
</tbody>
</table>

Use a minimum subset of CIL for composing an architecture or a platform.
Types in BALBOA CCF

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

- Automatic inspection of type composition through introspection
  - Data type checks
  - Protocol match checks
  - Adapter synthesis
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
A component:

is polymorphic because its ports can have many type mappings:

\[
\begin{align*}
\text{ports}(c_1) : &\quad \text{int} \quad \times \text{int} \quad \times \text{bool} \quad \times \text{int} \quad \times \text{bool} \\
\text{ports}(c_2) : &\quad \text{bv8} \quad \times \text{bv8} \quad \times \text{bool} \quad \times \text{bv8} \quad \times \text{bool} \\
\text{ports}(c_3) : &\quad \text{bv16} \quad \times \text{bv16} \quad \times \text{bool} \quad \times \text{bv16} \quad \times \text{bool}
\end{align*}
\]

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.
Behavioral Specifications

- Service-based specifications
  - An observable sequence of events that spans across multiple components
  - Compose multiple services to describe the overall interaction behavior

- Algebraic specification similar to process algebra
  - choice: $P_1 + P_1$
  - parallel composition: $P_1 || P_2$
  - hiding: $P_1 \setminus \text{Act1}$
  - relabeling: $P_1[\text{Act1} \rightarrow \text{Act2}]$
  - substitution: $P_1 [a \rightarrow P_2]$
  - invocation: call $P_1$

- Graphical representation using MSCs
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
Example: combine scenarios

to define the behavioral type of the controller

Projection + composition + preemption
Ultimately, it is about programming in building & using Silicon

- Cost reduction through reuse and abstraction: SW to the rescue?
  - “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

- Two fundamental challenges
  - C1 -- Capturing designer “black art”
  - C2 -- Ensuring compositionality of IP, ensuring correctness of compositions

- How to ensure programming for and programmability of Silicon microsystems?
  - Changes in programming models and/or programming language.
C2: Compositional Anomalies Affect Correctness

- Syntactical correctness does not guarantee correct behavior, let alone desired behavior
  - E.g., Use of delta cycles 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 + pre-emption or !signal = non-determinism
  - Problem with delta timing: infinite actions in a finite time (Zeno’s Paradox, Thompson’s lamp)

- Plus many other sources of non-determinism
  - Mostly from combination of concurrency, shared variables and immediate event notifications.
Example: Checking for event absence forms a cycle

```c++
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

Event notification can be missed depending on which process gets scheduled first.

```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)
    }
};
```

```c
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_THREADS(a);
    }
    void a() {
        e.notify();
    }
}

SC_MODULE(M2) {
    SC_CTOR(M2) {
        SC_THREADS(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

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

```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?
Start with Structured Operational Semantics…

For every syntactic SystemC statement $stmt$, a 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

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

<table>
<thead>
<tr>
<th>Rule Type</th>
<th>Rule Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(event-notify)</td>
<td>$e.notify() \xrightarrow{e,\emptyset,\emptyset,1} E \rightarrow_\sigma$</td>
</tr>
<tr>
<td>(wait-syntactic)</td>
<td>$\text{wait}(e) \xrightarrow{\text{syn}} \text{pause}; \text{wait}(e)$ wait for the next event $e$</td>
</tr>
<tr>
<td>(wait-event-block)</td>
<td>$e \notin E \land \neg Rv \quad \text{wait}(e) \xrightarrow{0} \text{wait}(e)$</td>
</tr>
<tr>
<td>(wait-event-unblock)</td>
<td>$e \in E \land \neg Rv \quad \text{wait}(e) \xrightarrow{1} E \rightarrow_\sigma$</td>
</tr>
</tbody>
</table>
SOS Rules for Sequential Composition

Produces a behavior of the form: \[ E_I \sigma_1 \sigma_2 \sigma_3 \sigma_4 \sigma_5 E_O \]

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

\[(\text{sequential-composition-1}) \quad (P_1, \sigma) \xrightarrow{E \ E_O, E_O^\delta, L_1, 0} (P_1', \sigma_1') \]
\[(P_1; P_2, \sigma) \xrightarrow{E \ E_O, E_O^\delta, L_1, 0} (P_1'; P_2, \sigma_1') \]

\[(\text{sequential-composition-2}) \quad (P_1, \sigma) \xrightarrow{E \ E_O^1, E_O^\delta, L_1, 1} (P_1', \sigma_1') \quad (P_2, \sigma_1') \xrightarrow{E \ E_O^2, E_O^\delta, L_2, b_2} (P_2', \sigma_2') \]
\[
\text{merge}(\langle E_O^1, E_O^2 \rangle, \langle E_O^1, E_O^2 \rangle, \omega) \]
\[
(P_1; P_2, \sigma) \xrightarrow{E \ E_O^1 \cup E_O^2, E_O^\delta \cup E_O^\delta, L_1 \cup L_2, b_2} (P_2', \sigma_2') \]
Take-Away Messages

- **Message #1**: Novel computational fabrics approaching intrinsic silicon efficiencies (per mm\(^2\), per watt)
  - The challenge is ensuring programmability and program models
  - Else, it costs too much, takes too long to be useful.

- **Message #2**: Methodology evolution from chip to embedded software design
  - Programming models, methods and language support for building embedded systems (on chip) is critical to exploiting the enormous technology capacities
  - HDLs do not automatically come with composability properties

- **Message #3**: Driven by the ‘rectangles’ and ‘triangles’, the ‘circles’ will likely see maturation of new methods that enable
  - Systematic modeling and exploitation of meta-data in design, verification, synthesis.