Component-Based Software
State-of-the-Art and Lessons Learned

A. Richard Newton
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley

The Productivity Gap

- 58%/Yr. compound Complexity growth rate
- 21%/Yr. compound Productivity growth rate

Source: SEMATECH
NTRS Processor 2010

It’s all about Communication!

It’s all about Concurrency
0.10μm CMOS Process Schematic

“High Performance” Logic

Legend
x is to scale
y is to scale

PECVD Si₃N₄/SiO₂
Passivation Cap.
LoK Polyimide

Cu Metal 1-8
Low K, Planarized
(May have Hard Mask Layers between Levels)

Cu Metal 1
BPSG Ge
SiO₂ Spacer
n⁺ poly gate
NMOS Transistor

Cu Plugs
Cu Barrier/Liner

Elevated CoSi₂ or TiSi₂
Salicide Source/Drain and Gate

p⁺ poly gate
PMOS Transistor

SEMATECH

It’s all about Communication!

It’s all about Power!
Pentium II in NTRS 2010 Silicon

- Year: 2010
- Devices: 7,500,000
- Clock Speed: 1 GHz for 200mW
- Addressable Memory: 64 Gbytes
- Chip Size: 6 mm²

The Productivity Gap

- Logic Transistors per Chip
- Transistor/Staff Month

“How many gates can I get for $N?”

Source: SEMATECH
Value Creation in Product Development
... The Way It Was

e.g. IBM, Digital, Siemens

Product Definition Product Implementation Fundamental Technologies
Markets Technologies

Source: Andy Rappaport, TRG, 1996

Value Creation in Product Development
... The Way It Is Today

e.g. Microsoft, Cisco e.g. Xilinx, 3M

Product Definition Product Implementation Fundamental Technologies
Markets Technologies

Source: Andy Rappaport, TRG, 1996
Design Reuse: Lessons from Software

<table>
<thead>
<tr>
<th>Object-Oriented Approach</th>
<th>Component-Based Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assemble Components to Make an Application</td>
<td>Hard! Still requires OO Programming Skills</td>
</tr>
<tr>
<td>Design and Implement a Component</td>
<td>Hard! Requires OO Programming Skills</td>
</tr>
</tbody>
</table>

Objects versus Components

“Object orientation has failed but component software is succeeding”
(Udell, 1994)

- Definition of objects is purely technical
  - Encapsulation of state and behavior, polymorphism, inheritance
  - Does not include notions of independence or late composition (although they can be added...)
- Object markets did not happen
  - Like the FPGA market-- vendors give the tools away to sell a companion product (e.g. MFC)
- In OO, construction and assembly share a common base
  - Development is very technical, assembly is very technical
  - In CO, construction is technical, but assembly must be open to a wider user base
- Objects are rarely shaped to support “plug-and-play”
- Typically a component has to have sufficiently many uses, and therefore clients, for it to be viable
Where Has a Component-Based Approach Worked?

- **Simple Universal Protocol (SUP)**
  - Unix pipes (character streams only)
  - TCP/IP (only one type of packet; limited options)
  - RS232, PCI
- **Single-Owner Protocol (SOP)**
  - Visual basic
  - Massbus
- **Simple Interfaces, Complex Application (SIC)**
  - When “the spec is much simpler than the code” you aren’t tempted to rewrite it
  - Oracle, SAP, etc.
- Implies there are “natural” boundaries to partition IP and successful components will be aligned with those boundaries.

(*suggested by Butler Lampson*)

---

**Bottom Line: Component Reuse**

- **The Challenge Is Not in the IP Itself, but is in the Component Integration Protocols**
  - It’s not just a “standard bus” problem
  - We need SUPs and SOPs
  - This is true for hardware, software, and so/rdware components
- **Design Validation Remains the Key Bottleneck and is Likely to Get Even Harder**
Component-Based Approach

- Specific vendors will integrate their own families of parameterized components around a SOP
- Special-purpose components will be defined by standard (slowly varying) datatypes and protocols, e.g. MPEG, MP3, TCP/IP, DES, CDMA
- Third-party blocks will be introduced via SUPs, or as “software”
- Each component must be implemented very efficiently and reliably in terms of power, performance, and cost—like for a microprocessor rather than an ASIC
- What makes a Component-Based Approach a Platform-Based Approach?

Platform-Based Approach

“Only the consumer gets freedom of choice; designers need freedom from choice”

- A platform is a restriction on the space of possible implementation choices, providing a well-defined abstraction of the underlying technology for the application developer
- New platforms will be defined at the architecture-microarchitecture boundary
- They will be component-based, and will provide a range of choices from structured-custom to fully programmable implementations
- Key to such approaches is the representation of communication in the platform model
Leads to Conventional View of EDA Development

Behavior

Level of Abstraction

SW/HW

RTL

Today

Tomorrow

Mask

Effort/Value

Leads to Conventional View of EDA Development

Behavior

Level of Abstraction

SW/HW

RTL

Conceptual Gap

Gate-level “platform”

Mask

Effort/Value
Leads to Conventional View of EDA Development

Behavior

Design Entry Level

Hand-off “platform”

Hand-off “platform”

Hand-off “platform”

Hand-off “platform”

RTL

SW/HW

Effort/Value

Mask

Component-Based Programmable Platform Approach

- Application-Specific Programmable Platforms (ASPP)
- These platforms will be highly-programmable
- They will implement highly-concurrent functionality

Intermediate language that exposes programmability of all aspects of the microarchitecture

Integrate using programmable approach to on-chip communication

Assemble Components from parameterized library
New Approach: Compiled Processor Plus Development Tools

Describe the processor attributes from a browser-like interface

Using the processor generator, create...

Tailored, HDL uP core

Customized Compiler, Assembler, Linker, Debugger, Simulator

Use a standard cell library to target to the silicon process

Compact Synthesized Processor, Including Software Development Environment

- Use virtually any standard cell library with commercial memory generators
- Base implementation is less than 25K gates (~1.0 mm² in 0.25µ CMOS)
- Power Dissipation in 0.25µ standard cell is less than 0.5 mW/MHz

to scale on a typical $10 IC (3-6% of 60mm^2)
Bottom Line: Programmable Platforms

- The challenge is finding the right programmer’s model and associated family of microarchitectures
  ▲ Address a wide-enough range of applications efficiently (performance, power, etc.)
- Successful platform developers must “own” the software development environment and associated kernel-level run-time environment
  ▲ “It’s all about concurrency”

Observation: Top-Down versus Bottom-Up

- Clearly, design is some combination of top-down and bottom-up aspects
  ▲ In a top-down implementation emphasis, optimal partitioning drives the process
  ▲ In a bottom-up implementation emphasis, optimal combination of existing components drives the process
- Implies efficient evaluation, composition, and deployment of components (from a variety of sources) is the key emphasis in bottom-up implementation styles
What is ComponentWare in a Software Context?

- Visual Basic: ActiveX controls
- Windows Programmers: DLLs
- JavaBeans
- CORBA
- Microsoft COM (DCOM, COM+)
- i.e. "Binary-Level" reuse

COM Component Market
(excluding Microsoft) $300M in 1997, going to $3B in 2000

(Objects are almost never sold, bought, or deployed!)

Definition of a Software Component

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties”
Components are for Composition

_Nomen est Omen_

- Enables prefabricated ‘things’ to be reused by rearranging them in ever-new composites
- Aren’t most current software abstractions designed for composition as well?
- Isn’t reuse the driving factor behind almost all compositional abstractions?
- Software components are “binary units of independent production, acquisition, and deployment that interact to form a functioning system” Szyperski, “Component Software,” 1998

Component Software

- Requirement for independence and binary form rules out many software abstractions
- Early literature on components (motivated by the ‘software crisis’), refers to them as “Software IC’s” (Mclroy, 1968; Cox 1986).
- Why components?
  - All other engineering disciplines introduce components as they mature
  - And they continue to use them
Component Software

- Does not need to be a single component approach to integration
  - Any successful approach must reach critical mass
  - Critical mass requires sufficient variety and quality
  - Second sources often required as well
- Can also be used to offer a modular approach to products
  (stereos, Sun Solaris)
  - Efficient approach to “special-cases”

![Graph showing development cost and time efficiency vs. % bought with competitive edge, flexibility, ability to adapt]

Important Distinctions

- Between components (abstractions) and their instances
- Like class and object, blueprint and product
- Has been a major issue since the introduction of entity-relationship diagrams
Objects versus Components
“Object orientation has failed but component software is succeeding”
(Udell, 1994)

◆ Definition of objects is purely technical
  ▲ Encapsulation of state and behavior, polymorphism, inheritance
  ▲ Does not include notions of independence or late composition (although they can be added...)
◆ Object markets did not happen
  ▲ Like the FPGA market-- vendors give the tools away to sell a companion product (e.g. MFC)
◆ In OO, construction and assembly share a common base
  ▲ Development is very technical, assembly is very technical
  ▲ In CO, construction is technical but assembly must be open to a wider user base
◆ Objects are rarely shaped to support “plug-and-play”
◆ Typically a component has to have sufficiently many uses, and therefore clients, for it to be viable

Component-Based Design

Today

The component infrastructure

Tomorrow

Reliable, robust, adaptable, and ‘efficient’: “the operating system”

“pass a pointer” implemented on a stack
Component-Based Design

◆ In software languages:
  ▲ Assume we are all on the same “team”
  ▲ Optimize for efficiency, follow-up with debugging to fix problems (fragile interfaces)
  ▲ Doesn’t scale well! (e.g. the Web)
◆ In communication protocols (e.g. TCP/IP)
  ▲ Assume the guy at the other end is brain-dead
  ▲ Assume whatever can go wrong will (links break, etc.)
  ▲ Results in an “architecture” (e.g. packet-based) that is robust under the assumptions

Component-Based Design

◆ What is the “TCP/IP of component assembly”?  
  ▲ In the early days of TCP/IP we needed an IMP to implement the protocol, today it runs in s/w on a laptop
  ▲ Must be reliable, robust, adaptable (“learn”, self-optimizing, self-balancing, negotiate for resources...)
  ▲ Self-verifying (what does that mean?)
  ▲ Self-testing
  ▲ “Queriable”
◆ In many ways, it’s the “OS” of a component-oriented world
◆ Components might be collections of transistor, chunks of software (objects), applications, operating systems, NOW clusters, etc.
Next-Generation Operating Environments

- Advances in hardware and networking will enable an entirely new kind of operating system, which will raise the level of abstraction significantly for users and developers.
- Such systems will enforce extreme location transparency
  - Any code fragment runs anywhere
  - Any data object might live anywhere
  - System manages locality, replication, and migration of computation and data
- Self-configuring, self-monitoring, self-tuning, scaleable and secure

Adapted from Microsoft “Millenium” White Paper
http://www.research.microsoft.com

Next-Generation Operating Environments

- **Seamless Distribution**: System decides where computation should execute or data should reside, moving them there dynamically
- **Worldwide Scalability**: Logically there should only be one system, although at any one time it might be partitioned into many pieces.
- **Fault-Tolerance**: Transparently handle failures or removal of machines, network links, etc.
Next-Generation Operating Environments

◆ **Self-Tuning:** System should be able to reason about its computations and resources, allocating, replicating, and migrating computation and data to optimize performance, resource usage, and fault tolerance.
◆ **Self-Configuring:** New machines, network links, and resources should be automatically assimilated.
◆ **Security:** Allow non-hierarchical trust domains.
◆ **Resource Controls:** Both providers and consumers may explicitly manage the use of resources belonging to different trust domains.

Next-Generation Operating Environments

◆ **No Storage Hierarchy:** Once information is created, it should be accessible until it is no longer needed or referenced.
◆ **Introspection:** The system should possess some aspects of introspection and reflection.
  ▲ Pervasively self-monitoring
  ▲ Reason about its own configuration and performance
  ▲ Suggest improvements
◆ **Just-in-Time Binding:** Sort of like the Internet today, but extended to all object interactions. “Binding-by-Search”
◆ **Tools Emphasis Shifting:** From code-efficiency to rapid application development with wizards automatically generating scaffolding or framework code.
“WebOS”

- The goal is to provide a common set of OS services to wide area applications, including mechanisms for:
  - Resource discovery
  - A global namespace
  - Remote process execution
  - Resource management
  - Authentication
  - Security

- Provide services needed to build applications that are:
  - Geographically distributed
  - Highly available
  - Incrementally scalable
  - Dynamically reconfiguring

Forms of Design-Level Reuse

- “The architecture of a component-based system is significantly more demanding than that of traditional monolithic integrated solutions.”

- Sharing:
  - Consistency: Programming and scripting languages
  - Concrete Solution Fragments: Libraries
  - Contracts: Interfaces
  - Interaction Architectures: Patterns
  - Subsystem Architectures: Frameworks
  - Overall Structure: System architectures
Sharing Consistency: Programming Languages

- A programming language can make some things easy, some difficult, and others impossible
- By doing so, the language encodes a dogma of how things should be done
- Over time, the language dogma combines with a culture of proven ways of doing things using the language: it becomes the lingo of the field
- There is a grow acceptance of the benefits of stringent languages, exemplified by the transition from C++ to Java
- A programming language cannot enforce good design, but it can exclude things that are likely to cause trouble

Sharing Consistency: Programming Languages

- Languages and their implementations can enforce static safety properties
- The primary abstractive, structural, and compositional means of modern programming languages are:
  - Static Type Systems
  - Functions, higher-order functions, and functional composition
  - Blocks or closures
  - Lazy evaluation
  - Procedural abstractions
  - Exceptions and exception handling
  - Classes and implementation inheritance
  - Dynamic type systems
  - Late-binding and type-driven dispatch
  - Support for concurrency and synchronization
  - Module and package systems
Sharing Concrete Solution Fragments: Libraries

- Early programming languages attempted to provide (built-in) all the functions that should ever be used in multiple programs
- Explosion of built-in functions was the result
- Pascal (1971) still had I/O in the language, C (1978) did not
- Since then, clear tendency to take specific functionality out of languages, in favor of structural and abstractional features
- Libraries are naturally layered on top of each other
  ▲ Challenge is to avoid circular dependencies

Sharing Individual Interaction Architectures: Patterns

- Attempt to collect and catalog the smallest recurring architectures in OO software systems
- Gamma, et al (1995) defines a pattern with the following four elements:
  1. A pattern name
  2. The problem the pattern solves, including conditions that must be met for the pattern to be applicable
  3. The solution to the problem brought by the pattern (elements involved, their roles, responsibilities, relationships, collaborations)
  4. Consequences (results and tradeoffs) of applying the pattern
Sharing Individual Interaction Architectures: Patterns

- Gamma identified 23 design patterns (many more have since been identified and documented)

- Example, the Observer pattern:

```
Subject
  Attach(Observer)
  Detach(Observer)
  Notify()

Observer
  Update()

ConcreteSubject
  GetState()
  SetState()
  subjectState

ConcrteObserver
  Update()
  observerState

forall o in observers {
  o->Update()
}

Return subjectState

Observe State = subject->GetState()
```

Design Patterns

- Big Idea #1,324: Design patterns are microarchitectures
- They describe the abstract interaction among objects collaborating to solve a particular problem
- Quite different from software frameworks (see later)
- More abstract and less specialized than frameworks (patterns have no immediate implementation at all)
- Design patterns are smaller architectural elements than frameworks

- Question: could we categorize a set of hardware design patterns as examples from which all implementations are built?
Sharing Architectures: Frameworks

- A framework is a set of cooperating classes, some of which may be abstract, that make up a reusable design for a specific class of software
- Although frameworks are not necessarily domain-specific, they are usually concept-specific
- Frameworks usually keep a number of their classes open for implementation inheritance (i.e. formation of subclasses)
- They often provide default implementations to reduce the burden on lightweight use of the framework
- Most famous framework is the Smalltalk model view controller (MVC) framework (1988), which can be dissected into three principal pattern applications (Observer, Composite, and Strategy)

Sharing Overall Structure: System Architectures

- A few basic principles can be learned from successful software architectures of the past
  - Layering: strict and non-strict
  - Hierarchies or heterarchies
- Benefits of layered architectures have been empirically verified (see: http://www.sel.cmu.edu/technology/architecture/)
- Strictly layered (“onion model”) allows system to be understood incrementally, layer by layer, either bottom up or top down
- But strict layering has several disadvantages:
  - Hinders extensibility
  - Can introduce unacceptable performance penalties
Sharing Overall Structure: System Architectures

- The practice of accessing not only the next lower layer, but any of the lower layers leads to non-strict layering
  - Can solve the extensibility problem
  - Can solve the performance problem
- Has been argues that each level of a description should involve three to seven entities
- Kent Beck (1993) wrote:
  "The first test I apply [in my consulting practice] is whether a staff can explain the system to me in 3-5 objects. If they can't, there is no explicit architecture. More often than not, these projects are in trouble."

Components and Interfaces

- Interfaces are the means by which components connect
  - "A set of named operations that can be invoked by clients"
  - Specification of the interface becomes the mediating middle that lets the two parties work together
- Direct (procedural) and indirect (object) interfaces
  - Object interface introduces an *indirection* called method dispatch
  - Has a big effect when versioning objects, for example
  - Very view solutions to this problem
Interfaces and Standards

- “A component needs to hold a significant portion of a market specific to its domain”
  - Generally drives (quasi) standards
- A standard should specific just as much about interfacing of certain components as is needed to allow sufficiently many clients and vendors to work together (including acceptable deviations and “tolerances”)
- Wiring standards are not enough
  - People can find ways around wiring as needed: adaptors

Interfaces as Contracts

- Not only requirements on the component, but also on the user, hence the term “contract” of “agreement”
- Best captured today by preconditions and postconditions (and perhaps invariants)
  - e.g. Eiffel (Meyer)
  - Hoare triple: {precondition} operation {postcondition}
- Non-functional requirements
  - It shouldn’t fail, it should recover, it shouldn’t take too long,...
- Example of layout checking as a component approach
Nonfunctional Requirements

- Contracts usually state what is done under which provisions
- What about time taken, resources needed, etc?
  - Use of shared resources (e.g. heap)
  - In concurrent RT environment, priorities and their interactions
- Nonfunctional requirements can break components as well as functional ones
- C++ Standard Template Library (Usser & Saini, 96), execution time is bounded
  - Not in seconds, but as a complexity of legal implementation

Formal versus Informal

- Different parts of the interface can be specified more or less formally
- Formalizing wherever possible is a good idea: research
- Keep contracts as simple as possible
- Difficult when dealing with recursion and re-entrance
- Would like to have a compiler or tool check clients and providers for adherence to contracts
Interprocess Communication (IPC)

- Lots of ways:
  - files, pipes, sockets, semaphores, shared memory
  - all scale to networks, except shared memory
- All operate on level of bits and bytes
- Implementing complex operations on top of these mechanisms painful and error-prone
- RPC proposed in 1984 (Bird & Nelson)

Some Potential Key Technologies

- What software technology, or technologies, will play the central role in enabling such a distributed component architecture?
- Java and JavaBeans
- CORBA
- Microsoft COM (COM, DCOM, COM+)
- Jini
CORBA
(Common Object Request Broker Architecture)

◆ A standard for distributed objects being developed by the Object Management Group (OMG).
◆ CORBA provides the mechanisms by which objects transparently make requests and receive responses, as defined by OMG’s ORB.
◆ The CORBA ORB is an application framework that provides interoperability between objects, built in (possibly) different languages, running on (possibly) different machines in heterogeneous distributed environments.

CORBA (1.0 1991, 2.0 1995)

◆ Very open approach: a “wiring” model
◆ Connects a wide variety of languages, implementations, and platforms
◆ CORBA components cannot operate on an efficient binary level, but must engage in expensive high-level protocols
  ▲ e.g. Internet Inter-ORB protocol (IIOP)
  ▲ Visigenic ORB “Visibroker”, part of Netscape browser
◆ Object interface described in a common interface definition language (IDL)
  ▲ All languages must have bindings to OMG IDL
CORBA

Module Example {
    struct Date {
        unsigned short Day;
        unsigned short Month;
        unsigned short Year;
    }
    interface Ufo {
        readonly attribute unsigned long ID;
        readonly attribute string Name;
        readonly attribute Date FirstContact;
        unsigned long Contacts();
        void RegisterContact (Date dateOfContacts);
    }
}

Communication Refinement

- **Separate** *Function* of blocks from inter-block *Communication*
- Substitute lower-level detail for communications behavior

Source: Prof. Alberto Sangiovanni
Microsoft COM Analogy
(Component Object Model)

- Binary and network (DCOM) standard that allows two objects to communicate, regardless of what machine they are running on.
- Can be used from C++, C, VB, Java, Delphi, ...
- Supports three types of objects: In-process (DLL), local (EXE), and remote (DLL or EXE)

COM and DCOM (Microsoft)

- COM is a Binary standard
- It specifies nothing about how particular programming languages may be bound to it
- Does not even specify what a component or an object is
- Neither requires nor prevents the use of objects to implement components
- Fundamental entity it defines is an interface
Communication Refinement

- Issue: Where do we cut? Where are the “standards”?
- Where is the communication burden placed?
- Applies to both hardware and software
COM Interface

- On the binary level, an interface is represented as a pointer to an interface node
- The only specified part of an interface node is another pointer held in the first field of the interface node
- This second pointer is defined to point to a table of procedure variables (function pointers), called vtables

![Diagram of COM Interface]

COM Interface

- A COM component is free to contain implementations for any number of interfaces
- The entire implementation can be a single class, but it doesn’t have to be
- A component can contain many classes that are used to instantiate objects of just as many different kinds
- There is no single object identity that ever leaves the entire COM object
  - How does a client learn about other interfaces?
  - How does a client compare the identity of COM objects?
Accessing COM Objects

- Every COM interface has a common first method named *QueryInterface*
  - Takes the name of an interface, checks if the current COM object supports the named interface, and, if so, returns the corresponding interface reference
  - Error indication is returned if the interface is not supported
  - Interfaces are named using interface identifiers (IDs), which are Globally Unique Identifiers (GUIDs) (a 128-bit number guaranteed to be globally unique)
- If you know one interface, you can get to all the others
- All COM objects implement the *IUnknown* interface (which serves to identify the entire COM object)

Depletion of a COM Object

- Example of an ActiveX document object
- Key points:
  - Multiple interfaces
  - One generic interface
**COM and Reuse**

- Every COM object performs reference counting, either for the object (usually) or for each of its interface nodes
  - AddRef and Release are performed by the user
- As soon as the reference count becomes zero the COM object is unuseable and should self-destruct
  - Leads to recursive destruction of all the objects held exclusively by the object under destruction
  - Can create circular refs
- COM does not support any form of inheritance
- COM supports two forms of object composition to enable object reuse
  - Containment: one object (outer) holds an exclusive reference to another (inner)
  - Aggregation: categories
- Once published, a COM object and its interface may not be changed in any way

**Aspects and Differences**

- Binary interfacing standard
  - Com: core idea
  - Java: JNI, based on COM but Java-specific
  - CORBA: Does not define binary standards
- Source-level standards for compatibility and portability
  - CORBA: strong here (via compatible ORBs)
  - Java: implicit in Java source agreement; standardization of language bindings an issue
  - COM: no concept of source-level standards
- “Grown” versus “Forged” standards
  - All advance without supporting backward compatibility
- Memory management, life cycles, and garbage collection
  - CORBA: Does not offer a solution
  - COM: relies on reference counting
  - Java: relies on garbage collection (with RMI also supports distributed garbage collection)
Aspects and Differences

◆ Concepts for Evolution and Versioning
  ▲ COM: Freeze interfaces and their specs, along with interface ID, once published (solves version and migration problem)
  ▲ CORBA: Does not directly address this issue, but supports the weak notion of minor and major version numbers
  ▲ Java: Addresses versioning at the binary compatibility level, via a painstaking list of rules

◆ Concept of Categories
  ▲ COM: only one that has it

“Only the consumer gets freedom of choice; designers need freedom from choice”

Some Potential Key Technologies

◆ What software technology, or technologies, will play the central role in enabling such a distributed component architecture?

◆ Java and JavaBeans
◆ CORBA
◆ Microsoft COM (COM, DCOM, COM+)
◆ Jini
CORBA (1.0 1991, 2.0 1995)

- Very open approach: a “wiring” model
- Connects a wide variety of languages, implementations, and platforms
- CORBA components cannot operate on an efficient binary level, but must engage in expensive high-level protocols
  - e.g. Internet Inter-ORB protocol (IOP)
  - Visigenic ORB “Visibroker”, part of Netscape browser
- Object interface described in a common interface definition language (IDL)
  - All languages must have bindings to OMG IDL

Java/JavaBeans Analogy

- JavaBeans is a portable, platform-independent component model written in Java.
- It enables developers to write reusable components once and run them anywhere -- benefiting from the platform-independent power of Java.
- JavaBeans acts as a bridge between proprietary component models and provides a seamless means for developers to build components that run in ActiveX container applications.
**Attributes of JavaBeans**

- **Introspection**: enables a builder tool to analyze how a Bean works
- **Customization**: enables a developer to use an app builder tool to customize the appearance and behavior of a Bean
- **Events**: enables Beans to communicate and connect together
- **Properties**: enable developers to customize and program with Beans
- **Persistence**: enables developers to customize Beans in an app builder, and then retrieve those Beans, with customized features intact, for future use

**Communication Refinement**

*Standard interfaces constitute the backbone of an IP market: abstract form the concerns of hardware implementation (multi-target VC), abstract from the concerns of a particular bus (bus-independent VC)*

- System transaction, «ANY» data structure (e.g. video line)
- «ANY BUS» operation (data, address...)
- VSI-Alliance OCB Group, Virtual Component Interface (VCI)
- Communication Interface (e.g. bounded FIFO)
- Physical Bus (e.g. PIBus), fixed bus-width, detailed protocol
- Bus Wrapper

Source: Prof. Alberto Sangiovanni
Automated Interface Synthesis

Source: DARPA ISAT Silicon 2010 Study, 1997
(Randy Harr, Synopsys)

“It’s All About Concurrency”

◆ A global, synchronous model no longer works: neither in hardware nor in software
◆ The majority of errors in modern software development are due to concurrency issues: from Windows NT to Wind River
◆ We are at the beginning of a revolution in embedded runtime support. e.g. Sun Jini, COM+, Universal Plug-and-Play, Ninja
  ▲ Another place for SOPs and SUPs!
◆ Should consider the verification issue up front, and use a verifiable underlying model for concurrency
The Key Elements of an Integrated Approach to SOC

What is the Programmer's Model?

EDA Challenges Move to “Embedded Software”

Traditional Tools

Market

• Many copies
• Bringing hardware “values” to embedded software

“ISA” is new platform

Supplier Model

• Fewer copies
• Higher prices
• Lots of tool services and support

K. Kautzer, A. R. Newton, A. Sangiovanni-Vincentelli, November 1997
The Energy-Flexibility Gap

<table>
<thead>
<tr>
<th>Energy Efficiency (MOPS/mW or MIPS/mW)</th>
<th>Flexibility (Coverage)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embedded µProcessors</td>
<td>0.1</td>
</tr>
<tr>
<td>ASIPs</td>
<td>1</td>
</tr>
<tr>
<td>DSPs</td>
<td>10</td>
</tr>
<tr>
<td>Reconfigurable Processor/Logic</td>
<td>100-200 MOPS/mW</td>
</tr>
<tr>
<td>MUD</td>
<td>10-50 MOPS/mW</td>
</tr>
<tr>
<td>Pleiades</td>
<td>1 V DSP 3 MOPS/mW</td>
</tr>
<tr>
<td>LPArm</td>
<td>0.5-2 MIPS/mW</td>
</tr>
<tr>
<td>Dedicated HW</td>
<td></td>
</tr>
</tbody>
</table>

Source: Prof. Jan Rabaey, UC Berkeley

Challenges of Programmability for Consumer Applications

- Power, Power, Power....
- Performance, Performance, Performance...
- Cost

- Can we develop approaches to programming silicon and its integration, along with the tools and methodologies to support them, that will allow us to approach the power and performance of a dedicated solution sufficiently closely (~2-4x?) that a programmable platform is the preferred choice?
Bottom Line: Programmable Platforms

- The challenge is finding the right programmer’s model and associated family of microarchitectures
  - Address a wide-enough range of applications efficiently (performance, power, etc.)
- Successful platform developers must “own” the software development environment and associated kernel-level run-time environment
  - “It’s all about concurrency”

Value Creation in Product Development
... The Way It Was

e.g. IBM, Digital, Siemens

Product Definition

Product Implementation

Fundamental Technologies

Markets

Technologies

Source: Andy Rappaport, TRG, 1996
Value Creation in Product Development
... The Way It Is Today

![Diagram showing the relationship between markets, technologies, product definition, product implementation, and fundamental technologies.]

Source: Andy Rappaport, TRG, 1996

It’s all about Communication!

It’s actually all about Programmability!

It’s all about Power!