The THINK Component-Based Operating System

Jean-Philippe Fassino - France Telecom, R&D Division

MDE for Embedded System - Brest, September the 6th to 10th 2004
http://www.ensieta.fr/mda/

Partly based on material from T. Coupaye

Overview

I - THINK and CBSE
  › Mastering software complexity
  › Component-Based Software Engineering
  › Expected benefits
  › The THINK initiative

II - Fractal component model
  › Concepts
  › Principles
  › Organisation (open component model)
  › APIs

III - THINK, a Fractal support for Operating System
  › Interfaces
  › Components
  › Development
  › Configuration
  › Deployment

IV - The THINK component library
  › The library
  › Supported hardware
  › Typical composition
  › Evaluations
  › Kernel constructions

V - Conclusion
  › Summary
  › Links
Mastering Software Complexity

» The challenge of software engineering today is to conciliate
  » the mastering of the
    » development
    » deployment
    » integration
    » management (operation)
    » evolution
  » of software systems that are
    » distributed (asynchrony, faults)
    » heterogeneous (equipments, networks, OS, middleware, services)
    » open (topology, dynamicity)
    » partly legacy (third party elements)
  » … while taking into account applicative services needs (QoS)

» Where are we ?
  » “The computer industry has spent decades creating systems of marvelous and ever-increasing complexity. But today, complexity itself is the problem” A. Ganek, T. A. Gorbi (IBM)
  » “We don’t understand something very fundamental about how we build systems” J. Hennessy (Stanford)

Component-Based Software Engineering

» CBSE is a broad effort towards the rational construction and management of complex software systems

» Preliminary definitions*
  » “A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard”

  » “A component model defines specific interaction and composition standards”

  » “A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model”

  » “A component infrastructure is a set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly defined performance specifications”

*from G. T. Heineman, W. T. Council (eds.). Component-Based Software Engineering, Putting the Pieces 0. Addison-Wesley, 2001.
Expected Benefits of CBSE (1/2)

- Adaptation, integration and interoperation of software with bounded effort
  - Arbitrary deployment environments
    - construction of dedicated software infrastructures (e.g. embedded operating systems)
  - Evolution in needs, technical evolution
    - maintainability, durability
  - Organizational evolutions
    - interoperation, integration
  - Dynamic evolution
    - availability, scalability

- Management (administration)
  - Uniform control of software resources: drivers, resources managers, pool, cache, network domain, process, service, …
  - Possibility to instrument resources - including dynamically and automatically

Expected Benefits of CBSE (2/2)

- …more generally, prediction/assessment of certain properties (constraints) on software composition/assembly
  - Reliability, correctness, fault-management, fault-tolerance
  - Security, access control, isolation
  - Scalability, Availability
  - Testability, manageability, maintainability, durability
  - QoS, real-time
  - …

NB: "Components are a way to impose design constraints that as structural invariants yields some useful properties" Conclusions of the 7th International Symposium on Component-Based Software Engineering (ICSE2004-CBSE7), Edinburgh, Scotland, May 2004
The THINK Initiative

➤ THINK Goals
   ➤ Build complex software by assembling, composing « software bricks » a.k.a. components
   ➤ Provide an *homogeneous vision* of operating systems topology
   ➤ No pre-defined kernel philosophy (e.g. mono, micro, exo, …)
   ➤ No pre-defined core functionality (scheduler, memory management, …)
   ➤ Systematic use of components
   ➤ Hardware abstraction components provide direct access to hardware functions
      • Smaller one bring high portability

➤ Process
   ➤ Define *architectural principles* and *concepts* for the construction and management of software systems
   ➤ Realize *implementation* than can be
      • Extended
      • Tooled
      • Composed gracefully (communications, security, QoS, …)

➤ Adopt the Fractal component model

Overview

➤ I - THINK and CBSE
   ➤ Mastering software complexity
   ➤ Component-Based Software Engineering
   ➤ Expected benefits
   ➤ The THINK initiative

➤ II - Fractal component model
   ➤ Concepts
   ➤ Principles
   ➤ Organisation (open component model)
   ➤ APIs

➤ III - THINK, a Fractal support for Operating System
   ➤ Interfaces
   ➤ Components
   ➤ Development
   ➤ Configuration
   ➤ Deployment

➤ IV - The THINK component library
   ➤ The library
   ➤ Supported hardware
   ➤ Typical composition
   ➤ Evaluations
   ➤ Kernel constructions

➤ V - Conclusion
   ➤ Summary
   ➤ Links
Fractal Concepts Overview

» « Classical » concepts
  » Components are runtime entities
  » Interfaces are the only access points to components. Interfaces emit and receives operation invocations
  » Bindings are communication channels. They can be primitive (in the same address space) or composite (to cross over boundaries). In the latter case, they are represented as components and bindings

» More original concepts
  » A component is the composition of a membrane and a content
  » The membrane exercises an arbitrary control over its content. A membrane is made of controllers. It can export control interfaces for some of these controllers
  » The model is recursive at arbitrary levels. The recursion stops with base components which have an empty content. Base components encapsulate entities in an underlying programming language
  » Components can be shared by multiple enclosing components

Components

» Definition
  » A runtime entity exhibiting a recursive structure and reflexive capabilities...

» Components as runtime entities
  » Components are units of
    » Development: design, modelling, implementation
    » Deployment
      » Packaging, installation, configuration, activation
      » Operation (administration)
  » Fractal targets the complete software life cycle but focuses on component as runtime entities
    » Components do exist and can be manipulated as such at runtime for operation purposes

» No predefined « granularity »
  » Fractal components can be of any size
    » e.g. service, resource, data, protocol stack, packet network, binding (stub), drivers, threads, scheduler, …
Interfaces

Definition of (component) interface

- An access point to a component
- An interaction point between components
- A place where operation invocations can be emitted or received

Not to be confused with language interface in relation with typing in Fractal

- Defined as a list of operation signatures, e.g. IDL interfaces
- Also called interface signature in Fractal typing
  - An interface type defines
    - an interface signature
    - some additional properties (constraints) (cardinality, contingency…)
  - Which maps ODP definition:
    "An abstraction of the behaviour of an [component] that consists of a subset of the
    interactions of that [component] together with a set of constraints on when they may occur"

Fractal terminology also uses

- Client versus server interfaces a.k.a role
  - A place where operation invocations can be emitted (client interface) or received (server interface)
- "Functional" versus "control" interfaces
- External versus internal interfaces (detailed later)

Bindings

Abstract definition

- A communication channel between components
- Bindings are “oriented” from a client to a server interface

Bindings are concretized as “primitive” or “composite” bindings

- (Primitive) binding: local communication channel between components (interfaces), i.e. in the
  same address space
  - (Primitive) Bindings are typically implemented by C pointers.
- Composite binding: a configuration of binding components and (primitive) bindings
  - Binding components: specialized component dedicated to communication between
    components to deal with:
    - Distribution: components not in the same address space
    - Communication styles: synchronous, asynchronous...
    - Other communication properties connected to proxy, security, transactions, etc.
Hierarchical Structure

- Fractal is hierarchical at arbitrary levels
  - Components as units of configuration
  - A component controls the components it contains (a component is under the control of its enclosing component(s))

- Composite & primitive components
  - A, B, C are composite
  - D is primitive

- Sub and super relationship
  - C is sub component of B
  - C is super component of D

(Reflexive) Control

- Definition
  - A runtime entity exhibiting a recursive structure and reflexive capabilities. A component is composed of a membrane and a content. A component has well defined access points called interfaces, and provides more or less introspection and control (intercession) capabilities.

- A membrane exercises an arbitrary control over its content. It embodies the control behavior associated to a particular component, e.g:
  - It can provide explicit and causally representation of its content (sub components)
  - It can intercept oncoming and outgoing operation invocations targeting or originating from its content an
    - Superpose a control behavior:
      - Suspending, check-pointing, resuming activities
      - Reifying or changing operation invocations parameters
      - Managing transparently technical services: persistency, security…
      - Managing QoS: memory consumption (garbage collection)…
  - Or it can do nothing at all!
Structural Concepts and Terminology

Architectural Principles

- Components as runtime entities
  - Focus on deployment and management of open systems

- Separation between interfaces and implementations
  - Interfaces are the only access points to components

- Programmatically controllable bindings
  - Bindings are not static nor "hidden in code" but externalized so as to be manipulable by (external) programs

- Hierarchical structure with sharing
  - Encapsulation, resource management

- Arbitrary reflexive control
  - The specification only defines some predefined ("standard") control interfaces dedicated only to structure management (except perhaps life cycle management …)
Benefits

For development
- The model enforces the use of « good principles »
  - Requires programmers to think about architecture
- Separation of concerns : functional vs non functional (management)
- Homogeneous development methods and tools
  - Modelling
  - Test
  - Debugging
- Reusability, portability

For integration, deployment and administration
- Reusability (quick portability)
- Configurability (assemblage, parameterisation)
- Homogeneous administration tools (supervision, monitoring...)
- Reconfigurability, maintainability
  - Safety
  - Availability
  - Scalability
  - QoS

Altogether
- More productivity for programmers
- More durability for software

Organization of the model

Elements of the model specification
- « Levels of control »
  - Foundations
    - Base components with no reflexive capabilities (legacy code)
    - IDL : Fractal is not only for Java
    - Naming and binding API (Name, NamingContext, Binder)
  - Introspection
    - Component and Interface API
    - Introspection of components boundaries
  - Configuration
    - (structural introspection and intercession) Attribute, Content, Binding, Lifecycle control API
    - (Predefined but more generally arbitrary) reflexive control of (white-box) components structure
- Basic Typing : role, contingency (optional, mandatory), cardinality (singleton, collection)
- Instantiation
  - Generic factories : create components of a type given as input
  - Standard factories : “ad-hoc” factories that create components of one type each
  - Templates : “facilities” that create components isomorphic to themselves

Everything is optional and extensible (« open model »)
- Introspection, control interfaces and controllers, factory interfaces, typing
Introspection API

- This level provides external introspection capabilities
- A component at this level owns the Component Fractal control interface for interfaces discovery
  - Component allows discovery of all interfaces owned by a component (server and client, external and internal)
- Interfaces are named
  - The name of an interface is valid in the context of the component that owns this interface
- Interfaces are typed
  - An interface implements both
    - its functional interface signature (expressed in Fractal IDL: e.g. I, J, K)

```java
interface Component {
    any[] getInterfaces ();
    any getInterface (string intfName);}
```

Configuration API (1/5)

- This level provides more introspection and intercession capabilities
- It allows for exposition and control of components internal structure
- It defines 5 « standard controllers » ...
  - BindingController
  - ContentController, SuperController
  - AttributeController
  - LifeCycleController

...used for initial configuration and dynamic reconfiguration

- NB
  - Bindings and content controls are really central to architectural configuration
  - Attributes control is concerned with a restrictive, classical sense of configuration: parametrization
  - Strictly speaking, life cycle control is not concerned with configuration - but it often needed for configuration
**Configuration API (2/5)**

- **BindingController**
  - Used to manage *local bindings* between components
    - Complex bindings (e.g. remote)
      - Must be created using the naming and binding framework
      - May involve binding components
    - Strong semantics of *locality*
      - Bound interfaces must be owned by components in a same direct enclosing component

```java
interface ContentController {
  Component[] getSubComponents();
  void addSubComponent (Component c);
  void removeSubComponent (Component c);
  any[] getInternalInterfaces();
  any getInternalInterface (string c);
}
```

**Configuration API (3/5)**

- **ContentController**
  - Used to manage the hierarchical structure of components
  - Management of bindings between sub and super components (a.k.a. import/export bindings) belongs to content control
Configuration API (4/5)

- **AttributeController**
  - Attributes are configurable properties of components
  - AttributeController interface can be extended
    - Actually, only integer type managed

```java
interface AttributeController {
    int get(string attr);
    void set(string attr, int value);
}
```

Configuration API (5/5)

- **LifeCycleController**
  - Semantics of start and stop are voluntarily as weak as possible
    - May implement usual suspend/resume or start/stop semantics
    - May or may not start/stop sub components
  - The central point is the isolation of 2 states
    - STARTED in which components can accept operations only through their functional interfaces
    - STOPPED in which components can accept operations only through their control interfaces
  - LifeCycleController will often be extended or completely redefined to suit arbitrary life cycles
  - NB: this part of the specification may evolve and even disappear…
Instantiation API

**Template**
- Component description
- Can be instantiated many times
  - Statically (by compiler)
  - Dynamically (through generated factories)

**Factory**
- Dynamically create components of one specific type
  - They are explicitly programmed to do so
  - Or they are generated from template component description

```
interface Factory {
    Component newInstance();
}
```

Overview

**I - THINK and CBSE**
- Mastering software complexity
- Component-Based Software Engineering
- Expected benefits
- The THINK initiative

**II - Fractal component model**
- Concepts
- Principles
- Organisation (open component model)
- APIs

**III - THINK, a Fractal support for Operating System**
- Interfaces
- Components
- Development
- Configuration
- Deployment

**IV - The THINK component library**
- The library
- Supported hardware
- Typical composition
- Evaluations
- Kernel constructions

**V - Conclusion**
- Summary
- Links
Interface Definition Language

- Interface Definition Language (IDL)
  - Similar to ODP
  - Interfaces are expressed with
    - Constant fields
    - Methods
    - Types
      - Primitives
        - [unsigned]{byte, short, int, long}, char
        - any
        - string
      - Interface references
      - Arrays
  - Programming languages mapping
    - C
      - Primitive xxx ⇒ jxxx, unsigned xxx ⇒ juxxx
      - any ⇒ void*
      - string ⇒ char*
      - [] ⇒ *
      - ...

```
interface Test {
    int bar(int arg);
    unsigned int foo();
}
```

Instantiation of interfaces
(cf. C++ Vtable)

```
typedef struct {
    struct MTest* meth;
    void *selfdata;
} RTTest;

struct MTest {
    jint (*bar)(void* this, jint arg1);
    jint (*foo)(void* this);
};
```
Instantiation of components

- A component owns one or more functional interfaces

```
CALL(itf, proc, ...)

itf->meth->proc(itf->selfdata, ...)
```

Interface calling

Architecture Design

- The Hello World example

```
package activity.api;
interface Main {
    void main(int argc, string[] argv);
}
```

```
package video.api;
interface Console {
    void putc(char c);
    void putcs(string str);
    void putxycs(int x, int y, string str);
    void scrollup();
    int cols();
    int rows();
}
```
**Primitive ADL : hwmain**

```c
type MainType {
    provides activity.api.Main as main
}

type hwType extends MainType {
    requires video.api.Console as console;
    attributes position
}

primitive hwmain implements hwType {
    skeleton hw nolifecycle
}

primitive hwmain {
    requires video.api.Console as console
    provides activity.api.Main as main
    attributes position
    skeleton hw nolifecycle
}
```

**Primitive implementation : hwmain**

```c
#include <activity/api/Main.idl.h>
#include <video/api/Console.idl.h>

struct hwmaindata {
    // Imported interfaces
    Rvideo_api_Console* console;
    int position;
};

#ifdef ! defined(ONLYDEFINITION)

/* Main interface method */
static void mainentry(void* _this, jint argc, char** argv) {
    struct hwmaindata* self = (struct hwmaindata*)_this;
    CALL(self->console, putxycs,
         self->position,
         self->position, "Hello World ");
}

struct Mactivity_api_Main hwmain_mainmeth = {
    main: mainentry
};
#endif
```
Providing Fractal Configuration API

» Either delegate configuration API implementation
  » Generic shared tiny code
    - Works on meta-data which encodes internal & external behaviors extracted from ADL
      - Provided & required interfaces
      - Attributes
  » Meta-data can be generated by tools
  » Need to respect some programmatic convention naming

» Or implement our own

Delegation mechanisms

```c
struct exportedskeleton {
    Rfractal_api_Component  myreference;
    unsigned int nbexported;
    struct {
        char* name;
        struct { void* meth, *selfdata;} itf;
    } exported[];
};

struct importedskeleton {
    unsigned int nbimported;
    struct {
        char* name;
        void* itf;
    } imported[];
};

/* BindingController delegation implementation */
static void bind(void* _this, char* name, void* itf) {
    struct importedskeleton *self = _this;
    int i;
    for(i = 0; i < self->nbimported; i++) {
        if(strcmp(self->imported[i].name, name) == 0) {
            *(void**)self->imported[i].itf = itf;
            return;
        }
    }
    // Error: trying to bind an unexisting interface name
}

struct Mfractal_api_BindingController _bindingcontroller = {
    bind: bind,
    ...
};
```
# Generated meta-data : hwmain

```c
#define ONLYDEFINITION

#include <hwmain.c>
static struct hwmaindata component2;
static struct importedskeleton imported2 = {1, {
  "console", &component2.console} }
};
static struct attributedskeleton attributed2 = {1, {
  "position", &component2.position} }
};
struct exportedskeleton hwbase_hw = {
  &_component, &hwbase_hw},
  3, {
    "main", &hwmain_mainmeth, &component2},
    "binding-controller", &bindingcontroller, &imported2},
    "attribute-controller", &attributecontroller, &attributed02}
};
```

---

**Composite component ADL**

- No required code
- Describe
  - Contents (which can be either primitive or composite component)
  - Content bindings
  - Content assigns
  - Controller
    - A static tool (for generate required code) and a dynamic shared part
      - Existing controllers
        - Static - allows only static fixed configuration
        - Dynamic - allows dynamic reconfiguration (adding, removing, changing)
        - Secure - controls all interactions using a Reference Monitor

```adl
composite hwBase implements RootType {
  contains console = arm.sal100.h3600.video.lib.screen
  contains hw = hwmain
  binds this.main to hw.main
  binds hw.console to console.console
  assigns hw.position = 3
  controller org.objectweb.think.controller.Static
}```
**Static controller**

**Aims**
- Manages external bindings
  - No interception realized
- Creates all sub-components
  - This phase was done statically
- Binds sub-components together
  - By subsequent call to Component and BindingController of sub-components
    - A possible optimization was to fill statically dependencies on sub-component structures
- Assigns sub-components attributes
  - By calling AttributeController of sub-components
- Starts all sub-components
  - By traveling sub-components graph to respect order initialization
- Shared code that work on generated meta-data

---

**Generated meta-data:** hwBase

```c
static struct bxmlcomponent kcl[] = {
    {&hwBase_console.myreference, NeedStart},
    {&hwBase_hw.myreference, NeedStart},
};
static struct bxmlbinding kbl[] = {
    {1, 0xff, Mandatory, "main", "main"},
    {0, 1, Mandatory, "console", "console"},
};
static struct bxmlattribute kal[] = {
    {1, "position", (void*)(3)},
};
static struct ecdata hwBase = {
    sizeof(kcl) / sizeof(kcl[0]), kcl,
    sizeof(kbl) / sizeof(kbl[0]), kbl,
    sizeof(kal) / sizeof(kal[0]), kal,
    {&_ECci, &hwBase},
    {&_ECbc, &hwBase},
    {&_EClcc, &hwBase},
};
```
Deployments

```
.IDL compiler

C compiler

Generated binding components

Generated composite components

With enclosure

Linker

Kernel

Boot loader

C compiler

.h

.o

Dynamic Loader

Either by the kernel itself or by another kernel on another CPU

CPU
```

Final composition: hwKernel

```
composite hwh3600kernel {
    contains boot = arm.sal100.boot.lib.boot
    contains main = hwBase
    controller org.objectweb.think.controller.Boot
}
```

```
composite hwh3900kernel {
    contains boot = arm.pxa.boot.lib.boot
    contains main = hwBase
    overloads main/console = arm.pxa.h3900.video.lib.screen
    controller org.objectweb.think.controller.Boot
}
```
"Hello World" for iPaq h3800

Overview

I - THINK and CBSE
- Mastering software complexity
- Component-Based Software Engineering
- Expected benefits
- The THINK initiative

II - Fractal component model
- Concepts
- Principles
- Organisation (open component model)
- APIs

III - THINK, a Fractal support for Operating System
- Interfaces
- Components
- Development
- Configuration
- Deployment

IV - The THINK component library
- The library
- Supported hardware
- Typical composition
- Evaluations
- Kernel constructions

V - Conclusion
- Summary
- Links
The KORTEX library

- Hardware abstraction components (HAL)
  - Boot
  - Exceptions (PIC, timer, ...)
  - MMU (page table, TLB, exceptions, ...) Cache
  - Device drivers
    - Human devices: frame-buffer, keyboard, touch panel, ...
    - Storage devices: Disk, Flash
    - Communication devices: Serial port, Ethernet, Bluetooth

- Operating system services
  - Memory management components (flat & paged memory)
  - Thread and scheduler components (cooperative, round-robin, priority, QoS)
  - Network components (x-kernel)
    - Ethernet, IP, TCP, UDP, SunRPC
  - Bluetooth components
    - HCI, L2CAP, RfComm, HDI, Obex
  - File system components (VFS API)
    - JFFS2, E2FS, NFS, Ramdisk
  - Binding components (local & remote communication)
  - Service components (process, dynamic linker/loader, trader, ...)
  - Posix components (for application portability)

Supported Hardware

- Conform with Fractal components model
  - PowerPC G3 / G4
    - Apple Power Macintoshes (Only New-world architecture)
  - ARM
    - Intel StrongARM (HP/Compaq h3600, h3800)
    - Intel xScale (HP/Compaq h3900, h2200, h5500)
    - Portal Player’s PP5002 (Apple iPod)
    - Motorola Dragon Ball MX 1

- Experimentations (past and future)
  - PC (Intel x86) [INRIA]
  - Lego RCX (Hitachi H8 16Mhz, ROM 16KB, RAM 32KB) [INRIA]
  - Tini Internet Interface (microcontroller DS80C390, ROM 512KB, RAM 1MB) [INRIA]
  - DSP [ST Microelectronics]

- Simulator
  - HAL components built on Unix (Linux, Mac OS X, Cygwin)
  - Run Think kernels on Unix processes
    - Porting and debugging purpose
Bluetooth (1)
[Anthonny Pellerin]

Bluetooth (2)
[Anthonny Pellerin]
Scheduling interfaces

```java
interface Scheduler {
    void initJob(Job job,
        byte* function,
        byte* stackbase,
        int stacksize,
        int arg);
    void destroyJob();
    void addJob(Job job);
    void removeJob(Job job);
    Job getJob();
    void yield();
}
```

```java
interface Semaphore {
    void P();
    int tryP();
    void V();
}
```

```java
interface Mutex {
    void lock();
    int trylock();
    void unlock();
}
```

Component sizes

<table>
<thead>
<tr>
<th>Component Description</th>
<th>Sub components</th>
<th>Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited TCP/IP stack</td>
<td>9</td>
<td>15.5</td>
</tr>
<tr>
<td>Console &amp; graphic drawer</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>H3900 frame buffer (including video memory)</td>
<td>1</td>
<td>160</td>
</tr>
<tr>
<td>H2200 frame buffer (video memory on controller)</td>
<td>1</td>
<td>0.9</td>
</tr>
<tr>
<td>Preemptive priority scheduler &amp; semaphore</td>
<td>11</td>
<td>5.7</td>
</tr>
<tr>
<td>ARM cache &amp; flat memory management</td>
<td>7</td>
<td>3.9</td>
</tr>
<tr>
<td>Think controllers</td>
<td>4</td>
<td>3.1</td>
</tr>
<tr>
<td>C runtime (printf,strupr,memcmp,divisi,...)</td>
<td>n.a.</td>
<td>7.7</td>
</tr>
<tr>
<td>ARM EFL dynamic components loader</td>
<td>4</td>
<td>2.2</td>
</tr>
<tr>
<td>JFFS2 flash reader</td>
<td>3</td>
<td>9.1</td>
</tr>
<tr>
<td>PXA serial port &amp; ymodem</td>
<td>2</td>
<td>1.7</td>
</tr>
<tr>
<td>H3900 touch panel</td>
<td>5</td>
<td>3.9</td>
</tr>
<tr>
<td>Bluetooth stack (HDI, RFCOMM, OBEX,...)</td>
<td>17</td>
<td>83</td>
</tr>
</tbody>
</table>
Performance overheads

- After configuration, only interface calls add overhead
- Interface calling overhead

```c
itf->meth->foo(itf->data);
Void fooimpl(void* _this) {}
```

<table>
<thead>
<tr>
<th>ARM</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface calling</td>
<td>5</td>
</tr>
<tr>
<td>Procedure calling (with context)</td>
<td>3</td>
</tr>
<tr>
<td>Procedure calling (without context)</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PowerPC</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>Time</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Interface calling</td>
<td>6</td>
</tr>
</tbody>
</table>

Memory footprint overheads

- Primitive components

<table>
<thead>
<tr>
<th></th>
<th>Delegation overheads (bytes)</th>
<th>Optimized controller (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exported interfaces</td>
<td>4 + 12 * n</td>
<td>8 * n</td>
</tr>
<tr>
<td>Imported interfaces</td>
<td>4 + 8 * n</td>
<td>0</td>
</tr>
<tr>
<td>Attributes</td>
<td>4 + 8 * n</td>
<td>0</td>
</tr>
</tbody>
</table>

- For the 3KB IP component, overhead is about 2%
- For a 1 KB frame buffer, overhead is about 6%

- Composite components

<table>
<thead>
<tr>
<th></th>
<th>Standard controller (bytes)</th>
<th>Optimized controller (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub components</td>
<td>4 + 8 * n</td>
<td>0</td>
</tr>
<tr>
<td>Sub bindings</td>
<td>4 + 12 * n</td>
<td>0</td>
</tr>
<tr>
<td>Attribute assigns</td>
<td>4 + 12 * n</td>
<td>0</td>
</tr>
</tbody>
</table>

- For the TCP/IP stack with 10 sub-components and 25 sub-bindings, overhead is near 400 bytes.
Building a simple μ-kernel

RPC binding

<table>
<thead>
<tr>
<th>PowerPC G4 500Mhz</th>
<th>Total</th>
<th>Network link</th>
<th>Network driver</th>
<th>Marshalling null call</th>
</tr>
</thead>
<tbody>
<tr>
<td>10baseT</td>
<td>180µs</td>
<td>164.7µs (91.5%)</td>
<td>11.3µs (6.3%)</td>
<td>4µs (2.2%)</td>
</tr>
<tr>
<td>100baseT</td>
<td>40µs</td>
<td>24.7µs (61.7%)</td>
<td>11.3µs (28.3%)</td>
<td>4µs (10%)</td>
</tr>
</tbody>
</table>
Building a Doom kernel

1800f/s on G4 500Mhz
340f/s on iPaq 400Mhz
60f/s on iPod 75Mhz

Less than 2 days to build the specific kernel

Kernel construction illustrations

- Plan-P application specific kernel
  - Programming Language for Active Networks
  - Ethernet Bridge evaluation
    - 86.7 Mbits throughput on Kortex-based kernel
    - 65.0 Mbits throughput on Linux

- Kaffe
  - Looks like a JavaOS
  - 1.6 MB memory footprint (125KB Kortex, 475KB Kaffe)

- Secure kernel (Mohamed-Tahar JARBOUI)
  - By defining a secure composite controller to checks interactions

- QoS kernel (Jean-Charles TOURNIER)
  - By defining resource managers that provide QoS
Conclusion

Effectiveness of component model for building component-based operating system
- "Component-based" does not mean "less efficient"
  - Interface cost == C++ virtual cost
- Operating system development simplification
  - Components and interfaces clearly identified
- Quick system deployment
  - "Lego-like" construction of software by assembling pieces a.k.a. components
- Memory footprint reduction and performance increase
  - Removing unused services
  - Fine resources access allowed

Yes, that works
Links

➤ Think home page
  ➤ http://think.objectweb.org

➤ Fractal home page
  ➤ http://fractal.objectweb.org

➤ Questions ?