

# BALT & CAST

## Principles in Practice

Nick Hawes

Integration Meeting, BHAM, 11/07/2007

# Outline

## First Principles

- Architectures for Intelligent Systems
- The CoSy Architecture Schema

## Toolkit Overview

- BALT & CAST Basics
- Advanced CAST Features

# Architectures are Essential

- ▶ We take the view that architectures are an essential for, and a defining part of, an integrated systems.
- ▶ They play a number of roles:
  - ▶ Provide internal structure.
  - ▶ Define connection and communication patterns.
  - ▶ Provide a principled approach to the overall design.

# Levels of description

- ▶ We will use three different levels of description for architectures:
  - ▶ High-level principles and requirements.
  - ▶ A schema-level realisation of these.
  - ▶ Instantiations of a schema in a concrete design.

# Main Design Principles

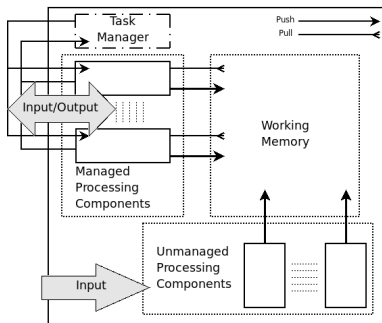
- ▶ An analysis of the CoSy target scenario produced 3 main principles:
  - ▶ Concurrency.
  - ▶ Structure knowledge.
  - ▶ Dynamic control of processing.
- ▶ Multiple components concurrently refining shared data structures to create (distributed) structured information.

# The Schema

## Overview

- ▶ These principles are realised in the CoSy Abstract Architecture Schema.
- ▶ A collection of loosely couple *subarchitectures*.
- ▶ Processing components within subarchitectures can be *managed* or *unmanaged*.
- ▶ These are managed by subarchitecture *task managers*.
- ▶ They share data via *working memories* which are *readable* by all components, but only *writable* by subarchitecture components and *privileged* components.

# The Schema Diagram



# The Toolkit

- ▶ We have implemented the schema in a the CoSy Architecture Schema Toolkit (CAST).
- ▶ This allows the schema to exist in concrete way in our integrated systems (rather than abstractly).
- ▶ Which in turn allows us to separate it from other aspects of the system (and do **science!**).

# Cost & Benefits

- ▶ CAST isn't exactly designed to make all aspects of development easier.
- ▶ **Benefits:** Ease of integration across components, languages machines. Ease of reconfiguration.
- ▶ **Costs:** Slightly odd programming style (event driven). Overheads of communication and translation (small).

## Some facts...

- ▶ Development time of about 12months.
- ▶ 219+ svn commits since move to svn. 5 “versions” before that.
- ▶ Largest system of 7 subarchitectures, 30+ components, 4 machines.
- ▶ 6000 - 8000 read-write-delete cycles/second for same-machine same-language pairs.
- ▶ 100 - 300 read-write-delete cycles/second for same-machine different-language pairs.

# Implementation Principles

Wherever possible:

- ▶ Use of within-process communication.
- ▶ Compile-time checking.
- ▶ No coded-in connection details

BALT & CAST are based on:

- ▶ Java + CORBA (built-in) + Ant
- ▶ C++ + CORBA (omniORB) + CMake

# The Boxes & Lines Toolkit

BALT was designed to:

- ▶ Replace pure CORBA in CoSy systems.
- ▶ Support architecture research.

Designed for rapid prototyping:

- ▶ Data structures specified and changed via IDL.
- ▶ Connections are changed independently of code.

# BALT Components

BALT systems are built from components:

- ▶ Each in component is a thread.
- ▶ Components are connected via typed push and pull connections.
- ▶ Connections end in interface-defined callback methods.
- ▶ **Weakness:** Pull connection *input parameters* are fixed as strings.

# The Basic Component Model

## CASTComponent

All CAST components have some common variables and methods:

- ▶ Process ID: `getProcessIdentifier()`.
- ▶ Output methods: `println()`, `print()`, `log()`, `debug()` (and C++ has `printfln()`).
- ▶ Configuration method: `configure()`.
- ▶ Thread methods: `lockProcess()`, `unlockProcess()`, `sleepProcess()`, `waitForUnlock()`, `runComponent()`.

# SubarchitectureWorkingMemory

All subarchitecture components are connected to a working memory.

- ▶ Effectively an unlimited-length associative-array.
- ▶ Implemented as a hash table along with a vector for tracking the order of adds and overwrites.
- ▶ Every working memory entry is associated with an address:

```
struct WorkingMemoryAddress {  
    string m_id;  
    string m_subarchitecture;  
};
```

# Working Memory Data

## Ontological and Data Types

CAST represents data on two levels: *data types* and *ontological types*.

- ▶ Data types are strings that refer to the data structures themselves. E.g. `LearnInstruction` is a struct from the vision subarchitecture.
- ▶ Ontological types specify the role the data plays in a CAST system. E.g. `LEARN_INSTRUCTION_TYPE` and `RESPONSE_TO_QUERY_TYPE` are two different roles the `LearnInstruction` struct can be used for.
- ▶ Data types exist in a 1-to-n relationship to ontological types, but often there is a 1-to-1 mapping (e.g. `SceneObject`).

# Working Memory Data

Why?

- ▶ Hides the translation of working memory entries into cross-platform code (via CORBA).
- ▶ Allows WM access to happen at conceptual level above raw data structures
- ▶ Allows configuration processes to refer to data as well as processes.

# Working Memory Data

CASTOntology

Ontology objects are used to maintain associations between data and ontological types.

- ▶ Associations are built with `establishObjectMapping`, `establishEnumMapping`, and `establishSequenceMapping`.
- ▶ `CASTCompositeOntology` extends `CASTOntology` to allow multiple ontologies to be combined (e.g. for cross-modal subarchitectures).
- ▶ All standard CAST components must have an ontology.
- ▶ Only the top-level data types that need to be written to WM need to be put in an ontology object.

# The Subarchitecture Component Model

CASTProcessingComponent

Defines a component that exists in a subarchitecture:

- ▶ Subarchitecture ID: `m_subarchitectureID`.
- ▶ Ontology pointer: `getOntology()`, `setOntology()`.

The subarchitecture id is set automatically, the pointer must be set manually (typically in constructor).

# Writing to a Working Memory

`WorkingMemoryWriterProcess`, `WorkingMemoryReaderWriterProcess`

Input to working memory is via three methods:

- ▶ New data:  
`addToWorkingMemory(id, type, data).`
- ▶ Alter data (must respect type):  
`overwriteWorkingMemory(id, type, data).`
- ▶ Delete data: `deleteFromWorkingMemory(id).`
- ▶ IDs can be generated: `newDataID().`

# Reading from a Working Memory

WorkingMemoryReaderProcess

Data is obtained from working memory as an object associated with a an id and an ontological type.

- ▶ Main read method:

```
CASTData<T> getWorkingMemoryEntry(id)
```

- ▶ In Java the returned object may need to be cast to the correct local type (because Java doesn't do templating very well).
- ▶ In C++ this returns a boost smart pointer (this is done to reduce copying of potentially large data structures).

# Reading from Working Memory

## WorkingMemoryChange

Changes to working memory cause all attached processes to receive notifications of the change.

```
struct WorkingMemoryChange {  
    WorkingMemoryOperation m_operation;  
    string m_src;  
    WorkingMemoryAddress m_address;  
    string m_type;  
};
```

This is typically where the input for WM reads come from.

# Receiving Working Memory Changes

WorkingMemoryReaderProcess

Changes to working memory can be routed to component callback objects.

- ▶ Objects are associated with an ontological type and a change operation: `addFilterObject(type,op,object)`.
- ▶ The object must be of type `WorkingMemoryChangeReceiver`.
- ▶ In Java this can be used in association with anonymous inner classes.
- ▶ In C++ there is a class for creating objects that directly call member functions of a class:  
`MemberFunctionChangeReceiver`.

# Subarchitecture Component Classes

`ManagedProcess`, `UnmanagedProcess`

The classes that provide collect all this functionality are as follows:

- ▶ Unmanaged components are currently implemented by sub-classing `UnmanagedProcess`. These components do not receive WM changes, and can only write to WM (not read).
- ▶ Managed components are currently implemented by sub-classing `ManagedProcess`. These components receive change information, can read and write WM, and can also interact with the task manager.

These names (along with loads of other ones!) will soon be changed to match our current theoretical work.

# Managed Components and the Task Manager

## ManagedProcess

A managed component can (should?) ask its subarchitecture task manager for permission to perform a task.

- ▶ Permission is asked for by proposing a task:  
`proposeInformationProcessingTask(id,name)`.
- ▶ Permission is received by callbacks: `taskAdopted(id)`,  
`taskRejected(id)`.
- ▶ When a task is complete, the task manager can be informed:  
`taskComplete(id,status)`.
- ▶ IDs can be generated: `newTaskID()`.

An `AlwaysPositiveTaskManager` is provided.

# Configuring A System

CAST systems are put together using simple config files. Configuration can also be done via code (if required).

- ▶ Config files must specify all the components, their languages, their host machines and any configuration options.
- ▶ Entries in config files are groups of lines specifying subarchitectures.
- ▶ Configuration options written in config files are passed to component's `configure()` method. The options should be passed to the superclass before they are accessed.

# Configuring A System

## A Badly Formatted Example

```
SUBARCHITECTURE vision.sa irobot1
CPP WM VisualWorkingMemory #--log
CPP TM AlwaysPositiveTaskManager
CPP MG cam.server.0 CameraServer -f ./config/cam0.cfg #--log
CPP UM video.server.0 VideoServer -o 0 -cl ./config/cam0.cfg -i 0 --log
#dewey CPP DD video.server.1 VideoServer -o 0 -cl ./config/cam0.cfg -i 2 --downsample 4 CPP MG change.det
ChangeDetector -c -0 -s 4 -d 2 -i "video.server.0" --log
CPP MG roi.detector OriginalSegmentor -r "150 240 620 480" -a 500 -s 0.0004 -o 0.5 --log
```

## Extra Features

As well as the basic operations described above, CAST allows you to do the follow:

- ▶ Configure how change information is propagated across subarchitectures.
- ▶ Create *privileged components* that can write to more than one subarchitecture working memory.
- ▶ Specialise task managers and working memories to solve specific problems for you.

# Integrating BALT Components

Because CAST is implemented using BALT it also allows you connect arbitrary BALT processes to CAST processing components. This allows:

- ▶ Communication without working memories for high-bandwidth data.
- ▶ Creation of system-wide servers for access to resources.

This is how we implemented the `VideoServer`, and possibly how we should approach the robot layer.

# The Hardest Part...

... is actually the system design! Consider:

- ▶ What subarchitectures are necessary.
- ▶ What information should be written to working memories.
- ▶ How this is used and/or altered by other components.
- ▶ What information is going to pass across subarchitecture boundaries.
- ▶ What should be placed outside the system (robot body etc.).
- ▶ ...

# The End

Cheers!