Whither Software Architecture?

Jeff Kramer
Imperial College London

software architecture

with lots and lots and lots of definitions

"... software architecture is a set of architectural (or, if you will, design) elements that have a particular form." (Perry,Wolf)

"The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both." (SEI)

"A software system’s architecture is the set of principal design decisions made during its development and any subsequent evolution." (Taylor,Medvidovic,Dashofy)

Whither software architecture

- how did we get here?
- impact?
- where are we going?

a “soap opera” based on my personal research experience
unintentional stepping on toes

my formative project

CONIC – “configuration programming”

coal mines

Underground, coal mines consist of a number of interacting subsystems:
- coal cutting
- transport
- ventilation
- drainage
- ...

... changes as the mine topography changes.

the CONIC project

Computer Control & Monitoring of underground systems in coal mining.

The investigators:
- Guess Who
- Morris Sloman

The research assistant:
- Jeff Magee
requirements elicitation

- complex
  > large number of interconnected devices, sensors, actuators, controllers, ...
- highly distributed
  > over the mine site both above and below ground
- evolving
  > new coal faces open, old faces close
- robust
  > against failures

Software structure should mirror the physical mine.

engineering distributed software

- **Information Hiding**
  Encapsulation of design behind an interface
  
  *David Parnas, CACM, 1972*

- **Abstraction**
  Programming-in-the-small Vs Programming-in-the-large
  
  *deRemer and Kron, TSE 1975*

- **Composition**
  “Having divided to conquer, we must reunite to rule”
  
  *Michael Jackson, CompEuro 1990*

CONIC research elements

1. distributable components
2. transparent local/remote communication
3. separate configuration description (architecture)
4. construction and modification/evolution (“configuration programming”)

1. distributable components

Key property of context independence.
* communication via a well-defined interface.
* third party instantiation and binding
* reuse in the same system (multiple pumps), and in different systems (other mines).

- input and output ports (indirection)
- parameterised component types
2. local/remote communication

- unidirectional
  - synchronous
  - asynchronous
- bidirectional
- rendezvous

consult a wise guru

3. configuration

Separate explicit description of the **structure** of the system in terms of the **composition** of component instances and connections (i.e., third party instantiation and binding).

Hierarchical composition helps to hide complexity.

4. “configuration programming”

Compile, build and deploy evolved system

Research team: Naranker Dulay, Kevin Twidle, Keng Ng

4. “configuration programming”

structural description

- evolved structural description

change script

system

- evolved system

TSE 1985

4. “configuration programming”

CONIC

- Reusable components
  The control software for a particular coal mine could be assembled from a set of components.

- On-line change
  Once installed, the software could be dynamically modified without stopping the entire system to deal with new coal faces.

Research team: Naranker Dulay, Kevin Twiddle, Keng Ng
CONIC

- The Iron Lady effect!

However ....

- Wider application than coal mining.
- Distributed worldwide to academic and industrial research institutions.
- Exciting and a lot of fun

TSE 1989

CONIC was not general

- .... was programming language dependent (Pascal)
- .... had fixed communications primitives
- .... had simple single message interfaces for bindings

Structural view provides a useful level of abstraction.

---

Darwin - a general purpose ADL

- Component types have one or more interfaces. An interface is simply a set of names referring to actions in a specification or services in an implementation, provided by the component.

- Structural view provides a useful level of abstraction.

- Systems / composite component types are composed hierarchically by component instantiation and interface binding.

Tool support: graphical design and software system generation, deployment, as well as ...

- Model component behaviour
- Compose behaviours using the same structural information as the software architecture

- Compositional reasoning using model checking

ESEC/FSE 1995, FSE 1996
**Process Calculus - FSP**

- **component behaviour**
  
  \[ \text{PUMP} = \text{STOPPED}, \]
  \[ \text{STOPPED} = (\text{cmd.start} \rightarrow \text{STARTED}), \]
  \[ \text{STARTED} = (\text{pump} \rightarrow \text{STARTED} \mid \text{cmd.stop} \rightarrow \text{STOPPED}) . \]

- **model architecture**

\[ ||\text{PUMP\_CONTROL} = (c: \text{CONTROL} \mid | p: \text{PUMP}) \]
\[ /\{ c.\text{cmd}/p.\text{cmd}, \]
\[ \text{level}/c.\text{level}, \]
\[ \text{pump}/p.\text{pump} \}. \]

**Analysis - LTSA**

**fluent** \( \text{RUNNING} = \langle \text{start,stop} \rangle \)

**fluent** \( \text{METHANE} = \langle \text{methane.high, methane.low} \rangle \)

**assert** \( \text{SAFE} = [\text{!RUNNING}] (\text{tick} \rightarrow (\text{METHANE} \rightarrow !\text{RUNNING})) \)

**... in collaboration as always ...**

- **Jeff Magee**
  - Shing-Chi Cheung
    - LTS, CRA & Safety

- **Dimitra Giannakopoulou**
  - Liveness & Fluent LTL

- **Nat Pryce**
  - Animation

- **Emmanuel Letier**
  - AFLTL

- **Sebastian Uchitel**
  - Synthesis

**connector wars**

**pragmatists Vs purists?**

**connector wars**

pragmatists Vs purists?

In the ARES project, Rob van Ommering saw potential of Darwin in specifying television product architectures and developed **Koala**, based on Darwin, for Philips.

First large-scale industrial application of an ADL.

---

**Koala**
Koala - example

Success...
... and is still
in use.

But...

Koala

Not more widely adopted, even in Philips!

- ... despite right level of abstraction
- ... despite compiler + code generation
- ... despite support for diversity

WHY???
Is Koala the only ADL in use?

ROOM
MetaH
AADL
UNICON
WRIGHT
ACME
Rapide
C2
xADL
ArchJava
SADL
UML2?
...

ADLs have not been widely adopted!

Disappointed but not downhearted...

“All hat and no cattle!”

Architecture research is a success

The abstractions pioneered in software architecture research have actually been very influential.

- qualitative aspects
- reviews/style guides
- architectural patterns
- provides and requires
- UML2
- modelling and analysis

Why were ADLs not widely adopted?

Object-Oriented Programming became mainstream!

- focus on class hierarchy
- implicit program structure
- implicit requires interfaces
- objects rather than components
**components vs objects**

- Benefits of a component-oriented view are recognised.
- We can gain the benefits even with objects.

**components from objects**

- Component type as an OO class
- Dependency injection (or inversion of control):
  - "new" and connections are no longer in the component code
  - Supports 3rd party instantiation and binding

**components from objects**

Components can be created from objects with methods/services provided and required. Components can also act as composite components.

**composite components**

```java
public class Leaf {
    public int attribute = 5;
    private Interface port1 = new Interface();
    public Interface getPort1() { return port1(); }
    private Interface port2;
    public void setPort2(Interface i) { port2 = i; }
}
```

```java
public class Composite {
    private Leaf a = new Leaf();
    private Leaf b = new Leaf();
    public Composite() { a.setPort2(b.getPort1()); }
    public Interface getPortA() { return a.getPort1(); }
    public void setPortB(Interface i) { b.setPort2(i); }
}
```
dependency injection

Permits separation of configuration from use

- **current EJB (CDI)** - “... server-side component architecture for Java”
- **Spring** - “... application development framework for enterprise Java”
- **Guice** - “lightweight dependency injection framework for Java 5 and above”
- **Autofac** - “…IOC container for .NET classes by treating them as components.

rays of hope for ADLs

some current practice in programming languages and some application domains

1. software maintenance and evolution
2. adaptive software

1. ADLs for software evolution

*Change* as fundamental in architecture definition – rather than making change management systems aware of architectural concepts.

- add three basic constructs to a Darwin-like ADL (Backbone) to support arbitrary extension:
  - resemblance, replacement, strata
- **Evolve Tool** uses UML2 graphical notation

define new components as a delta from the structure of one or more existing components (ie. reuse)
replacement

A’ globally replaces A in the architecture.

evolution

delta
add
replace ▲ with ▲
delete ❌

combines resemblance and replacement

stratum

extension

depends

include both strata to give extended system

include base stratum to give original system

* packages the definitions
* unit of ownership
* controls visibility

decentralised development

Used by U

Extended by X

Extended by Y

Developed by D
**Evolve demo**

- Evolve design tool
- Backbone ADL

![Evolve diagram](http://www.intrinsarc.com/evolve)

**incremental extension properties**

- **ALTER**
  Allows any possible extension even if unplanned
- **NO IMPACT**
  Others are not impacted by extensions they don't want
- **DECENTRALIZED**
  Supports a fully decentralized environment
- **COMBINE**
  Extensions / upgrades can be combined, problems rectified
- **NO SOURCE**
  Works even without source code!

**conformance**

- “What are the prospects for showing conformance between architecture and code?"
  
  question posed by Garlan and Shaw (ESEC/FSE 2011)

  Generate it!
  
  ... and store it in the code (“exoskeletal software”)

**2. ADLs for adaptive software**

“It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change.”

Charles Darwin
### MAPE cycle

- a single feedback loop?
- response times?
- complexity?

### three layer architecture model

- **Goal Management**
  - Plan synthesis based on a domain model and goals
  - Change Plans

- **Component Control**
  - Decentralised component selection and assembly by transitive closure on components satisfying plan actions
  - Safe operation, including during change (tranquility)

- **Status**

### generating the architecture

- **GoToTask**
- **moteto(t)**

  - **Repository**
  - **Motors**
  - **Location**
  - **Hardware**
    - Already instantiated
  - **SkyCamera**
    - Unavailable, network failure
  - **SLAM**
  - **Camera**
  - **Webcam**

### generating revised plans

- **Plan revision**
  - through model revision using observations and probabilistic rule learning

- **Learning through experience!**
In conclusion...

What does it all mean?

Engineering distributed software

Information Hiding

Encapsulation of design behind an interface

David Parnas, CACM, 1972

Abstraction

Programming-in-the-small Vs Programming-in-the-large

déRemer and Kron, TSE 1975

Composition

"Having divided to conquer, we must reunite to rule"

Michael Jackson, CompEuro 1990

CONIC research elements

1) distributable components
2) transparent local/remote communication
3) separate configuration description (architecture)
4) construction and modification/evolution (configuration programming)

Darwin - a general purpose ADL

Component types have one or more interfaces. An interface is simply a set of names referring to actions in a specification or services in an implementation, provided or required by the component.

Systems / composite component types are composed hierarchically by component instantiation and interface binding.

Evolve - a general purpose ADL

Components have one or more interfaces. An interface is simply a set of names referring to actions in a specification or services in an implementation, provided or required by the component.

Three layer architecture model

1. Planning over abstract domain
2. Assembly of software components to execute plans
3. Component execution and dynamic configuration

A clear separation of concerns

Evolve demo

Koala - example

Dependency injection

Continuing research...

Partial component model synthesis from goals and scenarios for architectural fragments,
- merge overlapping models,
- compose component models according to the system architecture

Requirements elaboration and revision using a combination of model checking and machine learning


Architecture as an Abstraction

... the same architectural description can be used as the structural framework to hang requirements, to compose behaviours for analysis, to compose component implementations for systems, ...
a life of collaborative research

interesting
serendipity
fun

Challenging

rewarding

colleagues

acknowledgement

Whither Software Architecture?

Jeff Kramer
Imperial College London