Towards Understanding Components in Product Line Approaches

For ECE 750 (Component-based Software Systems)
Professor Ladan Tahvildari
Chang Hwan Peter Kim
chpkim@swen.uwaterloo.ca
Generative Programming Lab
University of Waterloo

All the images used in this presentation have been taken from Google search results. They may be copyrighted by their respective owners. This presentation does use the images in any negative manner and it does not endorse the content of any of the images used.

Overview

I. Components in Software Product Line Approaches

II. A survey of components in the state of the art product line approaches
   a) Product Population approach
   b) Software Factories
   c) Generative Software Development

III. Challenges and future work
I. Components in Software Product Line Approaches
(Based on “Chapter 11: Components in Product-Line Architectures”,
Building Reliable Component-Based Software Systems, chapter by
Rob van Ommering and Jan Bosch)

• Essence: reusability, reusability, reusability!
  – D’Souza and Wills (reusable, independent deployment, composition) preferred!
  – Szyperski (explicit context dependencies)

• A thing is not a component if it cannot be reused across multiple applications (but
  it may be a module)
  – Single-product development (which does not develop with reuse in mind), at
    best, has a remote change of yielding components, if their units are accidently
    developed as reusable units
  – But single-product development can reuse existing components
    • Development from scratch, by definition, does not reuse
Essence of component-based software engineering (CBSE)

- Developing with reuse / developing for reuse
  - Single-product development, by definition, does not involve development for reuse

- The day that we no longer need to develop for reuse is
  - The day that the software engineering problem (software crisis) is solved
  - The day that software engineers and researchers lose their jobs

- In the mean time, essence of CBSE: development for reuse
  - Development with reuse may be used to achieve development for reuse, but often this is not feasible
  - Components are developed for reusability

Current Focus of CBSE

- CBSE assumes the existence of 2 component development paradigms
  - Single-product development
    - Top down, single architecture driven
    - Planned use of components (develop only what is needed)
    - All development within a single organization
    - All software developed during the process
  - Component market (e.g. Components Off The Shelf)
    - Bottom-up (no global architecture defined)
    - Use of components is opportunistic (use whatever is available)
    - Component software is available when starting the project
    - It results in inter-organizational reuse

- Single-product development: problems of standard software engineering process
- Component market: problems are not so obvious. After all, this is supposedly the goal of CBSE.
• Architecture problem
  – Component market approach can only work if 1) there is some kind of global architecture, or 2) if we manage to abstract sufficiently from the global architecture
  – VB is successful because it provides a global architecture to which components conform (e.g. ActiveX controls providing interfaces)
    • But ActiveX components are useless outside of Windows or VB
  – To abstract away from the global architecture, we must understand well the semantics of a component and component composition

• Scale
  – Reusability vs. usefulness
  – Small is reusable, but not very useful
  – Large is useful, but not very reusable
  – Solution: VARIABILITY (e.g. through parameterization)

• Resource mismatch
  – Successful component reuse frameworks, like VB require excess of resources that are not available in many products (e.g. in embedded software)

• Categorization
  – 3 kinds of components
    • Core: provides competitive edge, must be developed in-house, secretive
    • Key: could be obtained from others, but are not due to strategic reasons
    • Base: could be and often should be obtained from others
  – Software components will never be generic (as “Core” and “Key” components suggest)
  – Software cannot be made too specific either (as “Base” and “Key” components suggest)

• Economics
  – Selection and reuse of COTS and inter-organizational components → based on market laws (e.g. there must be competition, equilibrium between supply and demand)
  – Not the same for intra-organization → just setting up a reuse division (supplier of components) does not work (no real competition, quality not driven by demand, no real reward, no motivation)
Enter Product Line Approaches

- Need a systematic way of achieving software reuse within an organization
  - Something in between single-product development (developing with reuse) and component markets (developing for reuse)
  - Planned approach within an organization involving top-down and bottom-up construction

- Systematic way of achieving reusability of software for a set of related systems (usually within an organization) with commonalities and variabilities
  - Developing for scoped (intra-organization) reuse

- Components designed (variability) and used (configured) with multiple products in mind

- Approaches can differ depending on the relative amount of commonalities and variabilities
  - System family: considerably more commonality than variability
  - Product population: neither commonality nor variability dominates the other, but both are present in good amounts
  - “Product line”: systems considered as a package through business strategies

- Caveats
  - More work (must develop for multiple systems): a truly development-for-reuse method
  - Hard to have a vision of possible future variations of a product

Dimensions of components in product-line approaches

- Variability
  - Degree to which an existing piece of software can be modified when used in a product (context)
  - Examples (in increasing variability)
    - No variation, or reuse “as is”
    - Parameterization and inheritance
    - Component plug-ins
  - Ability to adapt a component is very important

- Independence (in increasing independence)
  - Reusable
    - Often cause dependence. E.g. certain techniques for reuse (e.g. inheritance) induce dependence
  - Composable
    - Composability of independently developed components

- Ideal: high variability and high independence for maximum flexibility in product-line approaches
II. A survey of components in the state of the art product line approaches

a) Product Population Approach

- "Building Product Populations with Software Components", Rob van Ommering, ICSE 2002
Concepts of Product Populations

- Principles
  - Product population: products with many commonalities and many differences
    - E.g. electronic products (TV and VCR share a video tuner, some electronic products may have same circuit board, etc.)
  - Architecture
    - Single generic architecture is not feasible
    - Arbitrary composition desired
  - Comparison against component-market
    - Components are not generic (functionality not mature enough)
    - Customers are intra-organization (economic laws of component markets do not apply)
    - Unlike generic components, resource constraints are tailored
  - Top-down and bottom-up approach that involves planning of many products and components

- Practicality
  - Philips Research Laboraties
  - Koala component technology (implementation of this product population approach) used to set up a product line architecture for consumer electronic products
  - Product line running since 2000, several products out on the market

---

Koala

- A component model and Architectural Description Language (ADL)
- Component = required/provided interfaces, with constraints
  - Interface = A small set of related functions, a unit of binding, an instance of a type
- Connector = binds interfaces to components
  - Straight connection 1), switch 2) or glue code 3)

• Binding possible as long as provided interface = subset of the required interface
• Triangle = direction of function call
Koala

- Example: a compound component describing a TV platform (drivers and other software)

```cpp
interface ITuner {
    void setFrequency(int f);
    int getFrequency(void);
}

component CTvPlatform {
    provides IProgram pprg;
    requires II2c slow, fast;
    contains
        component CFrontEnd cfre;
        component CTunerDriver ctun;
    connects
        pprg = cfre.pprg;
        cfre.rtun = ctun.ptun;
        ctun.ri2c = fast;
}
```

Koala: straight connection binding

- straight connection: static binding
- component D has a required interface r, which has a function called r_f
- component C has a provided interface p, which has a function called p_f
- when C and D are connected, since the “f” part matches (function names match), the two function references are given the same name (c_p_f)
Koala: glue connection binding

- glue connection: dynamic binding, but static binding possible through partial evaluation (assigning values to variables statically and rewriting the program)

```plaintext
within m {
    cfre.rtun.setFrequency(x) = ctun.ptun.setFrequency(x);
    cfre.rtun.getFrequency() = ctun.ptun.getFrequency();
    cfre.rtun.enableOutput(x) = chip.pout.enableOutput(x);
}
```

- allows a required interface to be mapped to multiple provided interfaces (which is not possible through straight connection)
- more generally, allows arbitrary connections (inter-component variability)

Koala: diversity (variability) mechanism

- Configuring components
  - provides interface: a component provides functions that perform configuration
    - e.g. set methods (setPlatform, setThreshold, setStrategy)
  - required interface: a component requires functions that gives back configuration data
    - e.g. get methods (getPlatform, getThreshold, getStrategy)

- What’s the difference? Which method is preferred?
Koala: diversity (variability) mechanism

• It’s all about separation of concerns!
• providing configuration functions means that the component largely performs configuration
• requiring configuration functions means that the component minimally performs configuration
• of course, no real difference if performing configuration requires virtually no effort
• in reality, configuration = complex and intertwined with underlying component (suggesting provided interfaces)
• but NO excuse: separate concerns ALWAYS WINS
  • Koala’s “diversity” (required) interfaces
• in-between way: an in-between “component” (like glue connection) performs the bulk of configuration

II. b) Software Factories

Based on “Software Factories” (Jack Greenfield and Keith Short, with Steve Cook and Stuart Kent, Wiley 2003)
Concepts of Software Factories

• Existing software product line approaches focus on techniques that manage architecture/design/implementation variability
  – E.g., product populations (esp. Koala) focus on a component composition model based on the "call and return" architectural style

• No framework for systematic variability management throughout the entire software development process (from requirements to product)

• Proposes a vision of such framework in terms of integration of "well-founded" technologies "available" today:
  – Component-Based Development
  – Model-Based Development
  – Software Product Line Development

• Work of Jack Greenfield, Keith Short, Steve Cook, Stuart Kent (architects of Visual Studio team system) at Microsoft Research

Concepts related to CBSE

• Visions (what this new paradigm will trigger)
  – Development by assembly
    • Most development = component assembly, involving customization, adaptation and extension
  – Software supply chains
    • Supply chains will create standard product types and formats
  – Relationship Management
    • Leasing components, receive updates and patches
  – Domain specific assets
    • Configurable tools, processes, and artifacts
  – Mass customization
    • Focus on assembly, rather than writing new code

• Critical innovations (what this new paradigm is really about)
  – Abstraction (from code to requirements)
  – Granularity (lines of code to web services)
  – Specificity (GPL to DSL)
Traditional components: *Catalysis* technique

- D’Souza and Wills (1998)
- 3 main concepts
  - **Type**
    - Specifies external behaviour of an object
    - Interface specification (conforms to an underlying metamodel)
      - Specification type model: relationships between Types
      - Invariants
      - Operations with pre/post conditions
    - E.g. OrderManager type (addOneOrder, changeOneOrder, getOrdersForCustomer, etc.)
  - **Collaboration of objects**
    - E.g. OrderManager and InventoryManager objects involved in “orders-to-cash” joint action
  - **Refinement**
    - Mechanism of relating descriptions of types and collaborations at different abstraction levels
    - “orders-to-cash” joint action refined into
      - “check-stock”, between OrderManager and InventoryManager
      - “check-stock-on-hand”, “find-substitute-product” (InventoryManager)
      - “enter-sale” between OrderManager and AccountManager

*Catalysis* Framework

- **Framework**
  - Recurring pattern of types, collaborations, and refinements
  - Holes in framework match plug-ins at instantiation time
    - Component framework (variability = plug-in, independence = reuse)
- **Criticism**: complexity and reliance on formal specification methods
  - UML Components (Cheesman and Daniels) offers simplified approach
From Components to Service Components

- Component
  - Specification and collaboration is often too low level, missing many of high level functional and non-functional requirements

- Service component
  - Specified in terms of interfaces which are parts of a contract (formally describing how provided interfaces behave with required interfaces)
    - Interface behaviour/obligation specified in terms of Service Level Agreement (SLA)
  - Characteristics
    - Large granularity (provides single business concept or process)
    - Asynchronous interaction (message-based, not call-and-return)
    - Interface specification behaviour fully defined as a state machine of messages

- Business protocols
  - Business protocol = subset of business process describing the message exchange state machine of each party
  - Challenges of designing a protocol (deadlocks, livelocks, reliability, sequencing, etc.)
  - Contracts implement business protocols
    - Parameterized schemas (e.g. Buyer/Seller contract, with Buy and Seller as variation points)
Service component technologies

- Service-Oriented Architecture (SOA)
  - A topology of loosely coupled, large granularity service components
  - Message-based contracts (bi-directional, asynchronous)

- Web services (implementation of SOA)
  - Features
    - Location independent ("write once, run anywhere")
    - Internet standards (open, intranet, internet)
    - True black-box (message interchange)
    - No explicit programming model (connection is the common vocabulary)
  - Specification (Web Service Description Language)
  - Web service data semantics
    - XML schemas (XMI) for data format standard
    - Standard dictionary of terms
    - Semantic web (data reuse framework using Resource Description Framework – uses XML for syntax and URIs for naming)
  - Web services process semantics
    - Language for expressing contracts (Message Exchange Patterns)
    - Business Process Execution Language (BPEL) for web services
      - Specify how a group of web services work to implement a business process

Web Services Description Language (WSDL) [WSDL]

- A specification for defining web services using a common XML grammar
- Contract between service requestor and service provider
- Platform, language independent, mainly for describing SOAP services

```xml
<definitions name="HelloService"...>
  <message name="SayHelloRequest">
    <part name="firstName" type="xsd:string"/>
  </message>
  <message name="SayHelloResponse">
    <part name="greeting" type="xsd:string"/>
  </message>
  <portType name="Hello_PortType">
    <operation name="sayHello">
      <input message="tns:SayHelloRequest"/>
      <output message="tns:SayHelloResponse"/>
    </operation>
  </portType>
  <binding>
    <!-- Use the SOAP HTTP Transport Protocol -->
  </binding>
  <service>
    <!-- URL -->
  </service>
</definitions>
```

1. invoke URL/wsdl/HelloService.wsdl
   sayHello World

2. Retrieve service

3. Request
   ("World")

4. Response
   ("Hello, World!")
II.c) Components in Generative Software Development

- Mapping variability in the problem space to variability in the solution space
- Providing a systematic way of reducing variability in the solution space from choices made in the problem space
• Product populations: architecture/design/implementation components
• Software factories: large granularity service components
• Generative software development: a set of configurable artifacts
  – A superset of components of product populations and software factories
• Aim: to separate variability management mechanism from the artifacts themselves
  – The variability management mechanism specified in a configuration language that is more natural than the language of the artifacts themselves
    • Currently: feature models
    • A feature model = a hierarchy of features and constraints describing valid configurations of features
    • A feature is a symbol plus some properties that may represent an interesting characteristic of a set of systems
• Each component comes with a set of feature models (the provided features at different abstraction levels) that the user of the component can configure
  – Also, each feature model has a set of required features (which may be resolved when two feature models are in the same search space)

Mapping features to models: A template approach

DEMO
III. Challenges and future work

- Components in software product-line approaches
  - Need for a design-for-scoped-reuse approach
  - Variability and independence

- Survey of the state of the art product-line approaches
  - Components in product populations approach: architecture/design/implementation components (Koala)
  - Components in software factories: traditional components to service components (Catalysis, Web services, larger granularity)
  - Components in generative software development: artifacts with variability (feature modeling, model templates)
Challenges and future work

- **Variability**
  - Specification of variation points
  - Separation of the configuration knowledge from components
  - Constraints between variation points

- **Independence**
  - Compositionality: only simple compositions allowed today, and usu. within one language, style, framework, etc.
  - But composition should be allowed between languages, frameworks, etc.
    - Composing frameworks, languages
    - Flexible composition model (e.g. aspect is a composition model – see GluonJ by Dr. Chiba)

- **Levels of abstraction**
  - Component should not just be a part of the software being produced, but also, of the processes, non-executable artifacts, etc.
  - Linking variability at different levels of abstractions through constraints
    - “Multilevel configuration”

- **Views, traceability, and constraints**
  - Components are connected using traceability links (which could also be constraints)
  - New components are derived from existing components by defining views

The Lego promise of CBSE
The Lego promise of CBSE

the vision of software supply chains

the highest level of programming offered by generative software

dvelopment
Timelessly demonstrate our relentless quest to find the Holy Grail

*We will* get there one day…
But only through the convergence of our brilliance

References


