Practical syntactic and extra-functional compatibility for black-box software components

Přemek Brada, Katedra informatiky a výpočetní techniky
FAV ZČU Plzeň

21.5.2012
Agenda

- Software Components: Origins and Core Concepts
- What is Software Component Anyway: From Concepts to Visualization
- Software Component Substitutability: What and How

- Service to People, Institution and Community
- Conclusions
Component Compatibility: Two streams in confluence

> What is Software Component
> How to Evaluate Substitutability

> The Roots and Manifestations of Practicality
  - software architecture research as response to growing complexity of real-world software
  - desire to provide answers that have both formal backing and near-term realistic applicability
  - mostly validating on current cutting edge technology

hmm, isn’t this just low-hanging fruit?
Software Components: Origins and Core Concepts
> **Modularity + Information Hiding**
> - separate compilation units
> - public interface / private implementation

> **Assume-Guarantee**
> - postconditions / functionality guaranteed
>   $\leftrightarrow$ assumptions (preconditions / dependencies) valid

> **Contract**
> - syntax (signature)
> - semantics (ADT), behaviour (traces)
> - extra-functional (QoS)
Foundations (2)

> Compositionality
  - [build systems so that] if property valid on parts
  - it holds on the whole as well

> Software Architectures
  - specify parts
  - compose system by rules (styles)

> Dependency Injection
  - part declares dependency
  - container injects supplier
Core Concepts

- Software Architecture
- Component

- Component Model
- Component Framework
Software Architecture

> “The set of principal design decisions made about a system.”

> Component
> Connector
> Composition rules / styles
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.”

> Technical part
  - independence, contractual interfaces, composition

> Market-related part
  - third parties, [distribution and] deployment
Component Model

> “A component model specifies the standards and conventions imposed on developers of components.” [Bachmann 2000]
  - component types (incl. surface features)
  - interaction scheme
  - composition rules

> “Compliance with a component model is one of the properties that distinguish components (...) from other forms of packaged software.”
Component Framework

> “A component framework is an implementation of services that support or enforce a component model.”

- deployment
- lifecycle
- resource management
What is Software Component Anyway: From Concepts to Visualization
A Historical Perspective / Components

Modularity

1968 McIlroy
“Mass-Produced Software Components”

1972 Parnas
“On the Criteria…”

1981 Misra & Chandy, 1985 Stark
“A Proof Technique for Rely/Guarantee Properties”

1992 Microsoft COM

1992 Perry & Wolf
“Foundations for the Study of Sw Architectures”

1996 Martin
“Dependency Inversion”

1999 Enterprise JavaBeans
CORBA Component Model

1995 Magee & Kramer
“Specifying Distributed Sw Arch”

2000 Medvidovic & Taylor
“A classification and comparison framework for software ADLs”

2002 Szyperski “Component Software” 2nd ed.

2002 Microsoft .NET (assemblies)

2004 Fowler
“Dependency Injection”

2006 OSGi 4

2010 Crnkovic et al.
“A Classification Framework for Software Component Models”

2011 IEEE Software Special Issue

Component-Based Software Eng.

201X Java Platform Module system

Software Architecture

1968 McIlroy
“Mass-Produced Software Components”

1972 Parnas
“On the Criteria…”

1981 Misra & Chandy, 1985 Stark
“A Proof Technique for Rely/Guarantee Properties”

1992 Microsoft COM

1992 Perry & Wolf
“Foundations for the Study of Sw Architectures”

1995 Magee & Kramer
“Specifying Distributed Sw Arch”

2000 Medvidovic & Taylor
“A classification and comparison framework for software ADLs”

2002 Szyperski “Component Software” 2nd ed.

2002 Microsoft .NET (assemblies)

2004 Fowler
“Dependency Injection”

2006 OSGi 4

2010 Crnkovic et al.
“A Classification Framework for Software Component Models”

2011 IEEE Software Special Issue

13
Motivation

- If we don’t understand concepts
  - we cannot communicate
  - things cannot be modeled
  - models cannot be manipulated and visualized

- If we don’t capture an aspect
  (of module interface)
  - we suffer from hidden dependencies
  - modularity breaks

http://www.flickr.com/photos/tobiasschlitt/2736153191/
The endless struggle for perfect terminology

What is a software component? As with the first edition, this book has many pages on that fundamental question. It contains three different definitions that adopt different levels of abstraction: a first one is found at the very beginning of the original Preface; a second in Chapter 4; and a final one in Chapter 20. The existence of more than one definition in this book – and quite a few more cited from related work (see Chapter 11) – has led to some turbulence. Kurzweil, Fein, and Ulrich Eiferschen (2009) in their excellent book, the term “component” for a brief discussion see

“CBSE is a coherent engineering practice, but we still haven’t fully identified just what it is.” [Brown, IEEE Software 1998]
Motivation: What Is a “Component”?

About 17 other definitions 1987-2007

A definition: software component

From the above characterization, the following definition can be formed:

“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.”

This definition was first formulated at the 1996 European Conference on Object-Oriented Programming (ECOOP) as one outcome of the Workshop on Component-Oriented Programming (Szyperski and Pfister, 1997).
A Definition

- We talk about **deployable architectural components**

- A piece of software called a component is
  - black-box (opaque) software element
  - with **completely and precisely specified** features crossing its encapsulation barrier (machine readable way)
  - 3\textsuperscript{rd} party composable and deployable in ways not foreseeable by the developer
  - model conformant (features etc. not arbitrary)
  - + small enough to be reusable, large enough to be business interesting
Why black-box matters

> Software Engineering core concept: modules -> interfaces -> components

> Information hiding enforced

> Goals
  - prevent property leaks
  - localize change effects
  - make software comprehensible
  - make software composable, interchangeable
Case study: OSGi

> Explicit required role
> In-completeness of specification
  - core: don’t declare services
  - declarative services: good & complete, not universal
> Weak specification-implementation consistency
  - core: package resolving only
> Moderate enforcement of black box
  - bind to declared packages and registered services only
  - class leaks from packages deprecated but easy
Outcome: The CoSi Component Model

> Component (bundle)
  - flat, dynamic model
  - OSGi-like packaging
  - Java, Groovy

> Rich base features
  - services, events, EFPs, etc.

> Container
  - simple run-time framework
  - no horizontal services (75kB .jar)
  - lifecycle interceptors, AOP and DI support
Guarantee of Conceptual Properties

> Enforce information hiding on surface
  - no export/import at runtime unless declared

> Prevent unresolvable dependencies
  - **statically** on install, update

```java
public void start(BundleContext context) {
    Integer numValues = (Integer) context.getAttributeValue("sensor.m
    sensor = new TemperatureSensor(numValues.intValue());
    service = context.getService("cz.zcu.fav.kiv.sensorregistry.Sensor
    service.registerSensor(sensor);

    TempSensorSvc tsvc = new TempSensorSvcImpl();
    context.registerService("cz.zcu.kiv.sensors.TempSensorSvc", tsvc);
}
```

```
>install tempsensor-bad-exports.jar
>ps
Id    State          Name
0     Started       systembundle (1.0.0)
1     Started       messagebundle (1.0.0)
2     Started       simpleshell.jar (1.0.0)
3     Started       sensorregistry.jar (1.0.0)
4     Started       windspeedsensor.jar (1.0.0)
5     Started       temperatureSensor.jar (1.0.0)
6     Started       measuringstation.jar (1.0.0)
8     Installed     tempsensor-bad-exports.jar (1.0.0)
>start 8
Bundle tempsensor-bad-exports.jar cannot register service with type cz.zcu.kiv.sensors.TempSensorSvc since it is not exporting proper interface
```
Selected Published Papers


Software Component Substitutability: What and How
Motivation

On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded.

... 3.1 FINDINGS ...

m) The inertial reference system of Ariane 5 is essentially common to a system which is presently flying on Ariane 4. (...) [Its] realignment function (...) was [retained and allowed] to operate for approx. 40 seconds after lift-off.

... p) Ariane 5 has a high initial acceleration and a trajectory which leads to a build-up of horizontal velocity which is five times more rapid than for Ariane 4. [This generated] the excessive value which caused the inertial system computers to cease operation.
HTTP ERROR 500

Problem accessing /system/console/dpsubstverif. Reason:

Caused by:

java.lang.ClassCastException: [Lorg.apache.commons.fileupload.FileItem; cannot
Component Compatibility: Challenges

> "Independent composition by 3rd parties" => need to check compatibility

> Very late binding
  - static architecture: deployment-time
  - dynamic architectures: run-time (or reconfiguration time)

> Can work with distribution form only
  - no sources
  - weak specifications

> Target platform ≠ development one
  - performance, access issues
Checking Compatibility: Options

> Formal – how much from contract to include
  - syntax \(\rightarrow\) type checks
  - semantics (behaviour) \(\rightarrow\) model checking
  - extra-func properties \(\rightarrow\) function evaluation

> Informal – how good is the data
  - version numbers
  - compatibility meta-data

Terminology
- substitutability: \(A \leftrightarrow B\)
- compatibility: \(A \rightarrow A'\)
Informal checks: Analysis of a high-profile OSS OSGi project

“When you only import packages and require some minimal or maximal version it means that the developer of the library has to do very good version management. If he changes an API and does not change the major number it can affect an already deployed application.

“When maven you already have the same problems it compile time but with OSGi it can crash at runtime. We need a whole set of new tools for this problem.”

-- comment on Peter Kriens’ blog, 6/2009

```
<table>
<thead>
<tr>
<th>Original</th>
<th>Semantic</th>
<th>Surface changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0.1</td>
<td>1.0.0</td>
<td>n/a</td>
</tr>
<tr>
<td>1.0.3</td>
<td>1.0.1</td>
<td>(none)</td>
</tr>
<tr>
<td>1.0.4</td>
<td>1.0.2</td>
<td>(none)</td>
</tr>
<tr>
<td>1.2.0</td>
<td>2.0.0</td>
<td>modification</td>
</tr>
<tr>
<td>1.2.1</td>
<td>2.0.1</td>
<td>(none)</td>
</tr>
<tr>
<td>1.2.2</td>
<td>2.0.2</td>
<td>(none)</td>
</tr>
<tr>
<td>1.4.0</td>
<td>2.1.0</td>
<td>extension</td>
</tr>
<tr>
<td>1.4.1</td>
<td>2.2.0</td>
<td>extension</td>
</tr>
</tbody>
</table>
```
Type-Based Compatibility for Software Components

Reconciling
• formal strength
• practical aspects

(Semantic and Behavioural)
• intensive research
• “state explosion problem” $\rightarrow O(e^n)$
Formal checks: Type-Based

> Assignment analogy

```
Vehicle v := (Car) ford;
```

- triggers type check
- dynamic type checking allows unforeseen subtypes

Instances of type $T_1$ can be bound to variables declared to be of type $T$ if $T_1 <: T$

- short <: long
- Car <: Vehicle
Component as a Type

> Formalized (and simplified) model
> Captures
  - collection of elements (their types)
  - element role (provided, required)

\[ C = (E^P, E^R) ; E = \{ e_i \mid e=(n, T, r, o, a) \} \]
  - declared component type
  - elements – name, type, role, opt, arity

> Instance \( \gamma : C ; \gamma.P \subseteq E^P \) etc.
  - not all elements may be always present
Type-Based Compatibility: Variations

> **Strict**
  - on component type representations
  - used for 1:1 any-time compatibility

> **Contextual**
  - consider actual use of component instance
  - include in comparison with $A'$ or $B$
Strict Compatibility

> Standard subtyping

\[ A^r <: A^c \text{ when} \]

\[ \forall p^r \in A^r.P, \forall p^c \in A^c.P \quad \cdot \quad p^r <: p^c \]

\[ \forall r^r \in A^r.R, \forall r^c \in A^c.R \quad \cdot \quad r^r :> r^c \]

- covariance for provided part
- contravariance for required one

> “A substitute component should be usable whenever the current one was expected, without the client noticing it.”

[Wegner, Zdonik, 1988]
Goal
- evaluate through recursion
- store for future reference
- read

Rules
- \(\text{none} < \text{ins} \mid \text{del} < \text{spec} \mid \text{gen} < \text{mut}\)
- \(\text{ins} \oplus \text{del} \rightarrow \text{mut}, \ \text{ins} \oplus \text{spec} \rightarrow \text{spec}, \ldots\)

\[\text{diff}(a,b) = \text{spec} \iff b <: a\]
Outcome: Type-Safe Component Updates

- Initial bindings and Updates that ensure application consistency
- Relation to framework checks (lifecycle)
Application and Measurements: OSGi Resolving Integration

<table>
<thead>
<tr>
<th></th>
<th>Null app</th>
<th>Parking</th>
<th>CoCoME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meta-data</td>
<td>28</td>
<td>58</td>
<td>234</td>
</tr>
<tr>
<td>Type check</td>
<td>256</td>
<td>1345</td>
<td>10437</td>
</tr>
</tbody>
</table>

Applications
- Null = 2 bundles
- Parking = 6 bundles
- CoCoME = 15/37 bundles

Felix resolver hook

Desktop + Android implementation
Industrial Applications

> Luminis (NL)
  - conceptual cooperation
  - Apache ACE project enhancements

> Openmatics s.r.o.
  - strict compatibility for API + 3rd party OSGi application verification
  - simulation tests of extra-functional property limits
Type-Based Compatibility: Variations

> **Strict**
  - on component type representations
  - used for 1:1 any-time compatibility

> **Contextual**
  - consider actual use of component instance
  - include in comparison with $A'$ or $B$

> **Prerequisites**
  - deployment context as a type
Deployment Context as a Type

> Deployment Context (D) = rest of architecture
  - components, bindings (relations, actual types)
  - architectural consistency

> Contextual Complement of $\alpha: A = \alpha$’s view of D

1. obtain actual usage
2. invert effective type
Deployment Context as a Type

> Deployment Context \( (D) = \) rest of architecture
  - components, bindings (relations, actual types)
  - architectural consistency

> Contextual Complement of \( \alpha:A = \alpha^{'}s \ view \ of \ D \)

1. obtain actual usage
2. invert
Contextual Substitutability

\[ \forall x^p \in P \, \exists p \in P \cdot p <: x^p \]
\[ \forall r \in R \, \exists x^r \in R \cdot x^r <: r \]

\[ \Rightarrow A^r <: D \, A^c \]
> Contextual complement: may provide less, require more
  ▪ $A_D.P \subseteq \alpha.P$ – not all provisions need be used, etc.

> Effective type & context time-dependent (!)

> Substitutability: strict ensures contextual in any context
  ▪ $A' <: A$ will always fit into $D(\alpha)$
Type-Based Substitutability: Practical Considerations

> How to obtain replacement component type
  - binary package, e.g. bytecode [Bauml, Brada 2010]
  - $E'$ not included, obviously

> How to obtain context (-> complement)
  - query component framework

> Subtyping vs. language rules
  - e.g. Java binary compatibility
Compatibility of Extra-Functional Properties

EFPs = qualitative characteristics
- performance, resource consumption, reliability
- maintainability, security, usability

Motivation and challenge
- properties “same” but values “context dependent”
- lack of normalization (esp. in CBSE)
- rudimentary support beyond RT and HA domains

Closest model: Palladio
Based on Generic EFP Meta-Model (K. Ježek)

> Component-model independent
  - **generic** meta-model
  - primitive, complex and derived properties
    
    $E = \{e \mid e = (n, T, E_d, \gamma, M)\}$
  - assignment and evaluation framework

> Usage context independent
  - **declaration** (type) – global repository
    
    (time_to_process, integer, \{unit:‘ms’, names: \{low, avg, high\}\})
  - **definition** (value) – local repositories
    
    mobile/GPRS:  
    time_to_process: low = 10, high=5000, ...
    data_transferred: low = 1, high=100, ...

desktop/10GEth:  
    time_to_process: low = 1, high=250, ...
    data_transferred: low = 1000, high=1000000, ...
EFFCC: Extra-functional properties
Featured Compatibility Checks

alignment ("time_to_process" always the same property)
EFP Compatibility Checks

- Works on complete (to be) composed architecture graph

1. Create graph
   - uses element/feature meta-types, types, roles

2. Find values (depth-first)
   - assign direct values
   - compute derived and function-defined values (recursion on “R” features)

3. Compare and evaluate
   - quality vector using the $\gamma$ function
   - results pair-wise and for the whole composition

Can always say “low < high”
Dealing with Complexity

> Use incremental evaluation for
  - large applications
  - architecture reconfiguration (substitution)
Goal
- integrate EFFCC to OSGi => provide EFP support

Implementation
- framework-tied “assignment module”
- metadata extensions
Selected Published Papers


“... were omitted for brevity”
Components

> Meta-modeling
  - The ENT meta-model

> Visualization
  - Advanced Interactive Visualization Approach
  - Vieport in Diagrams
Compatibility

> Simulation-based Approaches
  - componentized simulations
  - EFP verification

> Meta-data for Resource Constrained Scenarios
  - the CRCE repository
Service to People, Institution and Community
Grants

> (GAČR 1999-2001 „Developing software components for distributed environment“)

> **GAČR 2008-2010** „Methods and models for consistency verification of advanced component-based applications“

> **GAČR 2011-2013** „Methods of development and verification of component-based applications using natural language specifications“

> PhD student grants: Kamil Ježek M/cr, TALENT
Education and Institutional

> **Master-level courses (2001-today)**
  - Principles of / Advanced Software Engineering, Modern Trends in Software Engineering (seminar)
  - Programming Internet Applications, Java Enterprise Technologies

> **Bc-level courses, master theses**

> **PhD students (2006+)**

> **Institutional involvement**
  - Head of Software Engineering and Info Systems (2011)
Community Involvement

> Program Committees
  - Euromicro SEAA (2007+)
  - SOFSEM (2011+)
  - QUASOSS, CNSI, Objekty

> Industry Liaisons
  - guest lectures, CZJUG
  - Enterprise Software Engineering Competence Center (2011)
Conclusion
The Road(s) Ahead

> Making compatibility possible in resource-constrained scenarios
  - algorithm optimizations
  - rich meta-data use
  - pre-computed results of computationally expensive checks

> Visualization of complex software architectures
  - data-supported graph layouts and interaction
  - interaction and usability aspects
With a big Thank You to

Lukáš Valenta, Jarda Bauml
Kamil Ježek, Jarda Šnajberk, Lukáš Holý
+ all those Bc and Ing students

František, Širo, Ralf, Ivica,
Kung-Kiu, Tomáš, Petr
+ their colleagues

Jana + j + t + k

Him

http://www.flickr.com/photos/sushiraider/6893083210/