Autonomous Points in Component Composition

Vladimír Mencl, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, Czech Republic, mencl@nenya.ms.mff.cuni.cz

Abstract

Current technologies provide only a primitive support for managing configuration of applications composed of software components; advanced facilities are needed, namely for specifying the points in an component hierarchy suitable for initiating an update or acquisition from an independent vendor. *Autonomous points* proposed in this paper reflect these requirements.

Component Updating Deficiencies

- only primitive support for managing configuration in current systems (replacing the whole application, replacing only a single library, . . .)
- updating is necessary at the component level
- not all components in the application/component hierarchy are suitable for initiating an update
  - component are likely to be updated in continuous blocks
  - components designed to be a root of an update must have a precise specification to reliably decide on substitutability

Depth of Component Specification: Multiple Levels

- common methods for component interface specification
  - syntactical specification as interface declarations (e.g., Darwin [9])
  - semantic specification as method invocations sequences [3, 7] (usually take a form of regular expressions or CSP description)
  - such specifications are not sufficient to reliably decide on substitutability of a component
  - rather serve as an aid in the application design and design verification
- specifications of a depth sufficient for deciding on substitutability would be too costly on human resources
  - would rather resemble an API specification
  - specifications automatically generated from code can merely provide the same information as the descriptions based on method invocation sequence
  - the component vendor might not be interested in creating such deep specifications (and neither in releasing the specification to the public)
- for using design architecture specifications, the developers should not pay the penalty of having to specify the use-specifications at the highest available level (possibly guaranteeing algorithmic decision on substitutability)
- a need arises for multiple levels of specification depth

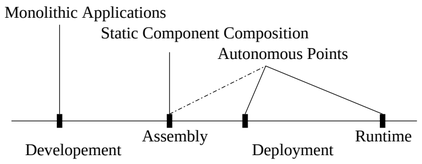
Component Life-cycle with Respect to Composition

- generally recognized component life-cycle stages include *development*, *assembly*, *deployment* and *run-time*
- architecture roles involved at each stage (similar to EJB [6])
  - key roles are *component assembler* and *component deployer*
- selection of components is typically done at the assembly stage

Modes of Composition – Autonomous points

- selection of a suitable component implementation may have to be postponed from the assembly stage to a later stage
  - either the *deployment stage* or the *runtime stage*
- by postponing the selection, the developer delegates the responsibility for the selection to a different architecture role (e.g., the component deployer)
- components to be autonomously composed have to be carefully selected; interface specification of such components must have depth sufficient to reliably decide on substitutability
- composition may be postponed by the developer, e.g., when he
  - wishes to provide an opportunity for customization
  - customization may be necessary, as certain implementations may have limitations on distribution due to e.g., patents or export restrictions; e.g. the RSA library.
  - component is being independently developed by a third party
  - component is provided as a service (with specified QoS)
  - component is provided as standalone services
    - ∗ the service provider may wish to hide changes made to components realizing the service
      - · if properties of the service are held, the provider should not be obliged to reveal the changes made
    - ∗ this way, web services can be seen as components
    - ∗ corresponds to postponing the selection as far as to the runtime stage
- we address the issue by introducing different mode of composition is used for such components, we call them *autonomous points* in the application/component hierarchy

Component Selection in Component Life-cycle



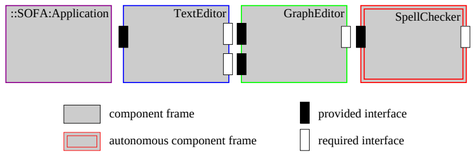
Autonomous points in the SOFA Component Model

- SOFA is a hierarchical component model – components are either *compound* (composed of other components) or *primitive* (units of implementation)
- Black-box view of a component (called a *frame*) consists of services provided and required and a behavior protocol
- Grey-box view describes the component at the first level of nesting – frames of subcomponents and their interactions (called an *architecture*
- The assembly process consists of selection an architecture for each frame, recursively.
- Autonomous points have been realized as a part of SOFA

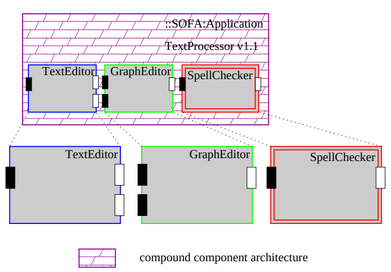
Example – Development of an Autonomously Composed Application in SOFA

- development in SOFA and the use of autonomous points illustrated on a hypothetical text-processing application

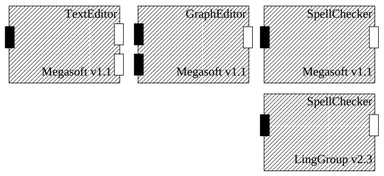
1. black-box specifications created (by specification vendors – possibly the same body as the application developer).
   - functionality is split into the following components.
   - SOFA::Application frame is the interface of each standalone application.



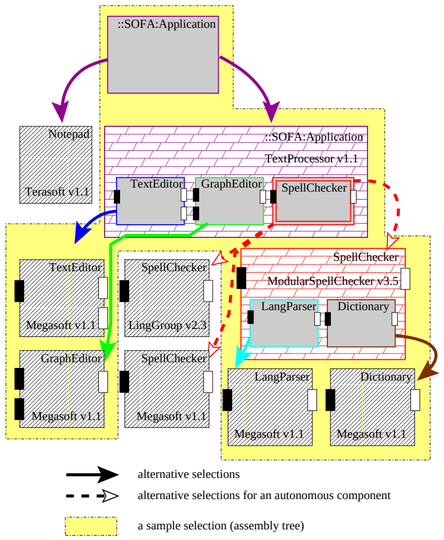
2. component architectures are created
   - composing functionality of subcomponents into a more complex component
   - only frames of subcomponents are considered
   - implementations not required at this time



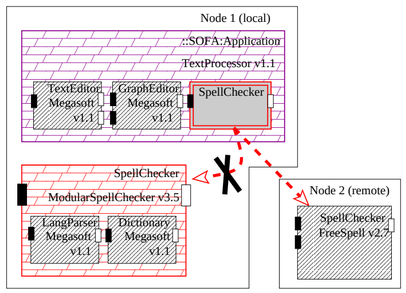
3. implementations of primitive architectures are created.



4. asssembly stage
   - an architecture (implementation) to be selected for each frame
   - multiple choices may be available
   - recursively for each frame of a compound architecture selected
   - a hypothetical scenario for the TextProcessor application
   - yellow area demonstrates a sample selection



5. the deployment and runtime stages
   - at the deployment stage, adjustments can take place in the autonomous points
     - a default implementation may be provided with the application
     - a completely different implementation may be integrated at the deployment stage (e.g., a remotely deployed web-service)
   - analogous changes may take part at the runtime stage
     - such changes have dynamic nature and are not recorded in the deployment descriptor



Example: Source Code

- Component Definition Language (CDL) used for describing components
- sample source code corresponding to the above example
  - `SpellChecker` component declared with the **auto** modifier – the component is developed autonomously

```
module TextEdit {
  struct Hint   { ... } ;    // data structure definitions
  typedef sequence <string> Text;
  typedef sequence <Hint> SpellResponse;

  interface SpellService { // interface to the spellchecker service
    void selectLanguage ( in string Language);
    SpellResponse checkText(in Text TextToCheck);

    protocol : ( selectLanguage ; checkText∗)∗
  };

  auto frame SpellChecker {
    provides:
      SpellService SpellC;
    protocol : ( SpellC.selectLanguage ; SpellC.checkText∗)∗
  };

  interface GraphService { ... };
  frame GraphEditor { ... };

  frame TextEditor { ... };
};

module SOFA {
  system frame Application { };
};

architecture TextProcessor implements SOFA::Application {
  auto inst TextEdit :: SpellChecker SC;
  inst TextEdit :: TextEditor TE;
  inst TextEdit :: GraphEditor GE;

  bind TE.GS to GE.GS;
  bind TE.SpellC to SC.SpellC;
};
```

- assembly descriptor and deployment descriptor stored as XML documents
- samples (corresponding to the above example)
  - both statical and an autonomous composition demonstrated
  - a default binding provided for autonomously bound `SpellChecker`

1. sample assembly descriptor

```
<assembly name="com.megasoft/TextEdit::TextProcessor">
  <frame ref="::SOFA::Application!1.1" />
  <architecture composition="static" archtype="compound"
      ref="::TextProcessor!1.1" />
    <!−− the architecture to be used for the toplevel
      −− Application component frame −−>
  <assembly>
    <!−− nested assembly descriptor for the TextEditor subcomponent
      −− ( statically composed) −−>
    <frame ref="::TextEdit::TextEditor!1" />
    <architecture composition="static" archtype="primitive"
        ref="com.megasoft/TextEdit::TextEditorImpl!1.1" />
  </assembly>

  <assembly>
    <!−− nested assembly descriptor for the SpellChecker subcomponent
      −− (autonomously composed) −−>
    <frame ref="::TextEdit::SpellChecker!1" />
    <architecture composition="autonomous" defaultarchitecture="yes"
        archtype="primitive"
        ref="com.linguasoft/TextEdit::SpellChecker!1.1" />
  </assembly>
  <!−− .... −−>
</assembly>
```

2. sample deployment descriptor
   - selection of a concrete implementation for `SpellChecker` postponed from the assembly to the deployment stage

```
<assembly ref="com.megasoft/TextEdit::TextProcessor">
  <unit>
    <location>sofa:local<location> <!−− specify the deployment location −−>
    <architecture composition="static" archtype="compound"
        ref="::TextProcessor!1.1" />
    <assembly>
      <frame ref="::TextEdit::TextEditor!1" />
      <!−− local configuration and parameters may go here −−>
    </assembly>

    <assembly>
      <frame ref="::TextEdit::SpellChecker!1" />
      <unit>
        <location
            ref="http://www.freespell.org/spellchecker?version=1.1"/>
        <architecture composition="autonomous" defaultarchitecture="yes"
            archtype="primitive"
            ref="org.freespell/TextEdit::SpellChecker!2.7" />
        <!−− for the autonomously composed SpellChecker subcomponent,
          −− an alternative location with a remotely deployed (web) service
          −− has been specified −−>
      </unit>
    </assembly>
  </unit>
</assembly>
```

Evaluation & Conclusion

- need to support multiple levels of specification depth pointed out
- only designated components should be allowed for assembly modifications
  - different modes of composition introduced – *autonomous points*
  - postponing the selection of concrete implementation to a later stage in the component life-cycle
- a proof-of-the-concept realization in the SOFA component model [1].
- tools & implementation
  - CDL compiler and a repository of specifications (data structures compliant to the concept of autonomous points) [4]
  - `http://nenya.ms.mff.cuni.cz/thegroup/SOFA/tools/`

References

[1] Plášil, F., Bálek, D., Janeček, R.: *SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*, Proceedings of ICCDS'98, May 4-6, 1998, Annapolis, Maryland, USA, IEEE CS Press 1998

[2] Bálek, D., Plášil, F.: *Software Connectors and Their Role in Component Deployment*, Proceedings of DAIS'01, Krakow 2001, Kluwer 2001 (in print), 18 p.

[3] Plášil, F., Višňovský, S., Bešta, M.: *Bounding Component Behavior via Protocols*, In proceedings of TOOLS USA '99, pp. 387-398, Aug 1999.

[4] Mencl, V., Hnětynka, P.: *Managing Evolution of Component Specifications using a Federation of Repositories*, Tech. Report No. 2001/2, Dept. of SW Engineering, Charles University, Prague

[5] Object Management Group: *CORBA Component Model Specification*, orbos/99-07-01,

[6] Matena, V., Hapner, M.: *Enterprise JavaBeans 1.1 Specification*, Sun Microsystems Inc., August 10, 1999

[7] Medvidovic, N., Rosenblum, D. S., Taylor, R. N. : *A Language and Environment for Architecture-Based Software Development and Evolution*, ICSE-21, Los Angeles, May 1999

[8] Levans, G.T., Sitamaran, M.(eds): *Foundations of Component-Based Systems*, Cambridge University Press, 2000, ISBN 0-521-77164-1

[9] Magee, J., Dulay, N., Kramer, J.: *Regis: A Constructive Development Environment for Distributed Programs*, In Distributed Systems Engineering Journal, September 1994

Other sources of information

Author

- Vladimír Mencl
- mencl@nenya.ms.mff.cuni.cz
- http://nenya.ms.mff.cuni.cz/~mencl/

Distributed Systems Research Group

- http://nenya.ms.mff.cuni.cz/thegroup

Keywords

software components, updating, architecture description languages (ADL), software configuration management