Reusable Services for Automotive Software

Advanced Automotive Software and Systems Development: Model-Driven Development of Reliable Automotive Services
San Diego, CA (USA), March 15-17, 2006

Jun.-Prof. Dr. Holger Giese
Software Engineering Group
University of Paderborn
Warburger Str. 100
D-33098 Paderborn, Germany
hg@upb.de
Outline

(1) Motivation: Why are Components not enough?
(2) Reusable Services: An evolutionary Perspective.
(3) Life Cycle Issues:
   - Requirements Engineering
   - Architectural Design
   - Detailed Design and Implementation
(4) Verification & Synthesis
(5) Summary
(1) Motivation

Why are Components not enough?

Problems Automotive SE has to face:

- Cooperation via networks results in dramatically increasing complexity
- Product variants differ w.r.t. to their support for complex cooperation
- Reuse of components is restricted to a given architectural context (product lines)
- Reuse of components (even in product lines) does not cover verification beyond module testing
Motivation

Observation for Automotive SE:

- High potential for reuse but currently limited to product lines and supplier know-how
- Top-down product line approach does not fit to the supplier chains (integration of alternative supplier solutions often very difficult)
- Decreasing development times and increasing complexity render more reuse the only feasible solution

⇒ Reuse “solutions” rather than the “boxes”, but how?
(2) Reusable Services

Example: Advanced Energy Management System

- **eBalance** from Hella KGaA Hueck & Co.
- **Background:** ongoing electrification creates a plus in functionality but also results in a steadily increasing need for electrical energy

**Specific function considered:**
- If the energy provided by the generator is not sufficient, the battery has to be used to handle the overload
- If a critical load status of the battery and such an overload is detected, the loads due to comfort functions have to be shut down gradually
The eBalance Architecture

- The **core** coordinates the producer and consumer
- **Driver** bridge the gap between the eBalance protocol and the specific components.
- Driver can be **located** on the node of the consumer resp. producer or operate via network
- We have
  - std. interfaces and
  - application specific interfaces
Pattern-based Architecture

Patterns:
- **Idioms** (class level)
- **Design pattern** (multiple classes)
- **Architecture pattern** (overall architecture)

Here:
- Std. interfaces can be made reusable assets in form of pattern
Service-Based Architecture

- Service: "the interaction among entities involved in establishing a particular functionality." (Krüger et al. MoDELS 2005)
- a generalization of the pattern concept
- Enables reuse of complex networks of collaborating roles including complete components (core)
- Represents the essential solution provided by the eBalance system
- The rest are wrapper and integration efforts which are specific for each system

✦ Support reuse of services and provide support to minimize the integration efforts
My “Service” Definition

• Feature (User View)
• Function (may realize a feature)
  – Function of the Advanced Energy Management (as a whole)
• Service (may realize a function)
  – Cooperation of different roles to provide the required Advanced Energy Management
  – Focus is the interaction between the roles and not the functional decomposition
  – Roles are not necessary functions (they are often describe no behavior which is useful/understandable on its own)
• Services are not necessary
  – Provide RPC like interaction
  – Offer their functionality through an interface
(3) Life Cycle Issues

- To make the service-oriented approach applicable, we also require support for other life cycle phases (not only architectural design)
- We will therefore look into:
  - Requirements Engineering
  - Architectural Design
  - Detailed Design and Implementation
  - Verification & Validation
Scenarios are a standard means to capture requirements
Scenarios are focusing similar to services on the cooperation between roles
⇒ Services are a natural perspective during requirements engineering!
Architectural Design (1/3)

Macro Architecture:
- **Assign** roles to components
- Assigned roles (and their states) constrain the component behavior (**role invariants**)
- A component usually fulfill **multiple** roles

⇒ Consistent **integration** of the roles is essential
Architectural Design (2/3)

Micro Architecture:

- **Controller:** provides the permanent service (e.g., a feedback control algorithm)
- **Operator:** takes care of the mode management of the component, failure management, and the communication with other components

⇒ Operator mediates between roles and the controller
Architectural Design (3/3)

The integration problem for multiple roles in the automotive domain is mainly restricted to the following three cases:

- **Reading of data**: map to the provided data of the embedded controller (may be simple or very complex; observer)
  
  *Only local problem!*

- **Writing of data**: map to provided must-values of the underlying controller (but conflicts are possible and must be resolved!)
  
  *Only local problem!*

- **Synchronization behavior**: handle mode-dependent read and write operations and command messages
  
  *Requires non-local understanding*

  *Synchronization is usually very error prone*

  - automatic support for synchronization (**correctness by construction**)
  
  (discussed in more detailed later)
Detailed Design and Implementation

- **Reading of data**: realize mapping methods with programming languages or CAE tools with code generation.
- **Writing of data**: realize mapping methods with programming languages or CAE tools with code generation.
- **Synchronization behavior**: automatically generate code from the state machine models derived during the architectural design.
(4) Verification & Synthesis: Pattern

**Purpose:**
- Safe shut down of consumers

**Roles:**
- Interface
- RT Statechart
- Invariants

**Pattern:**
- Set of roles
- Connector
- Properties

Model checking (timed models with UPPAAL or RAVEN):
- checks compatibility
- takes channel behavior into account (fault tolerance, fail safe, …)

\[ \phi = AG \neg (consumer.on \land core.disabled) \]
Synthesis of Synchronization

Behavior:
- Component synchronization must **refine** the related role protocol
- **Synthesis** must provide appropriate component synchronization

Results (symbolic synthesis; no timed models yet):
- 7 role
- 1,2 Mio state
- 1,5 sec
(BDD encoding grows only linear with the number of roles!)
Verification: Composed Correctness?

**Compositionality Result:** All systems
- that cooperate only via **correct patterns** *(model checked for each pattern)*
- and consist only of **correct components** *(holds by construction due to the synthesis step)*

will **behave correctly**.

Can be extended to **services**!

[Holger Giese]
(5) Summary & Future Work

- The **high potential for reuse** in automotive software cannot be fully exploited today as components and standard interfaces do not really represent the stable part of the solutions.
- Services often provides **better reusable assets** than components as they capture the stable part of the “solution” (and are already implicitly employed!).
- Exploiting the domain specific restrictions, techniques for **automating** the hard parts of **the integration problem** for multiple roles can be derived.

**Planned future work:** evaluating the whole envisioned approach in cooperation with industry