A Case Study on Behavioural Modelling of Service-Oriented Architectures

Marek Rychlý

Department of Information Systems
Faculty of Information Technology
Brno University of Technology
(Czech Republic)

4^{nd} IFIP TC2 Central and East European Conference
on Software Engineering Techniques,
October 12–14, 2009

Outline

1. Introduction
   - Service-Oriented Architecture (SOA)
   - Component-Based Development (CBD)
   - Case Study: Specification and Architecture
2. Behavioural Modeling of Services
   - A Calculus of Mobile Processes (π-Calculus)
   - Case Study: Behaviour of Services
   - Case Study: Behaviour of Components
3. Current Results and Future Work

Service-Oriented Architecture (SOA)

Definition (Service-Oriented Architecture)

SOA represents a model in which functionality is decomposed into small, distinct units (services), which can be **distributed** over a network and can be combined together and reused to create **business applications**.

[Thomas Erl, SOA: Concepts, Technology, and Design, 2005]

SOA can be described at three levels of abstraction:

1. **business processes**
   (a system is a hierarchically composed business process, represents sequence of steps in accordance with some business rules leading to **a business aim**)
2. **services**
   (an implementation of **a business processes** and their parts with well-defined interfaces and interoperability for the benefit of the business)
3. **components**

Component-Based Development (CBD)

Definition (Software Component)

A software component is a unit of composition with contractually specified **interfaces** and explicit **context dependencies** only. It can be deployed independently and is **subject to composition** by third parties.

[Clemens Szyperski, Component Software: . . . , 2002]

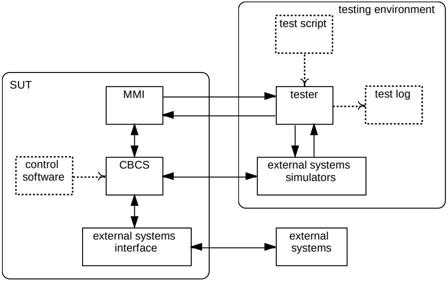
1. **Primitive components**
   (realised directly, beyond the scope of architecture description)
2. **Composite components**
   (decomposable on systems of subcomponents at the lower level)

The (dynamic) architecture of component-based system can evolve:

- **functional interfaces** can be (re)bound via **control interfaces**,
- **mobile components** can be moved into different contexts,
- (composite) components can change their functionality.

Case Study Specification

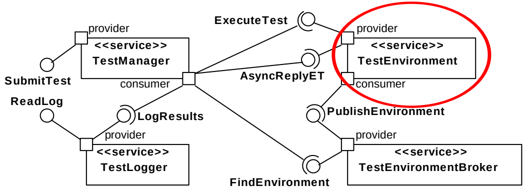
- **Testing environment**
- **Tester**
- Set of **external system simulators**
- **System under testing** (SUT)

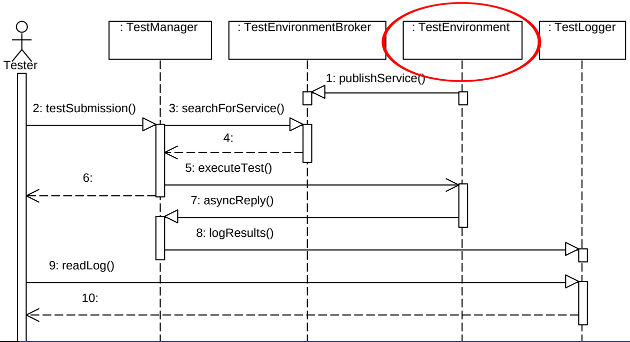


Testing environment is described as a composition of a **tester** and a set of **external system simulators**. **Tester** automatically executes specific **test scripts** and coordinates the SUT via a **man machine interface** (MMI) and the **external system simulators**. Set of **external system simulators** interact with SUT and simulate a tested environment (e.g. a behaviour of field objects as points, track circuits, coloured signals, etc.). **Computer based control system** (CBCS):

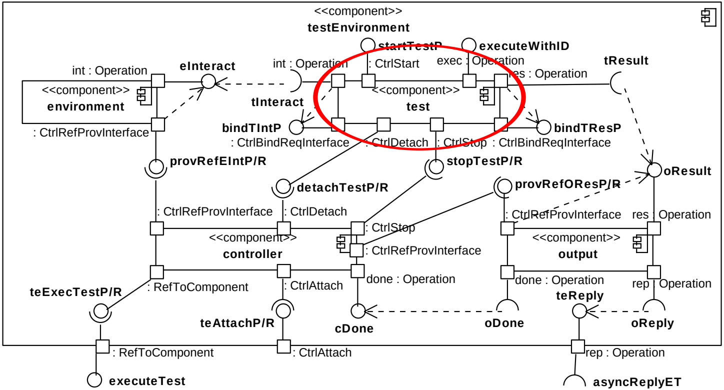
- runs the **control software**,
- interacts with operators via the **man machine interface** (MMI),

Service-Oriented Architecture of Testing Environment





TestEnv. Service as Component-Based System



Component “testEnvironment” is able to receive component “test” (a test script) and to attach it as its sub-component via component “controller”.

A Calculus of Mobile Processes (π-Calculus)

- Algebraic approach to description of a system of concurrent and mobile processes.
- Two concepts: **agents** (communicating processes) and **names** (communication channels, data, etc.).

$\overline{a}\langle b\rangle.P$	output prefix	$P + Q$	sum of capabilities of processes
$a(c).P$	input prefix	$P \mid Q$	composition of processes
$\tau.P$	unobservable prefix	$!P$	an infinite composition of the process
$(c)P$	restriction of scope		

$$
\begin{aligned}
P &::= M \mid P \mid P \mid (c)P \mid !P \\
M &::= 0 \mid \pi.P \mid M + M \\
\pi &::= \overline{a}\langle b\rangle \mid a(c) \mid \tau
\end{aligned}
$$

Reduction, Abstraction and Application

Communication defined as a **reduction relation** $\rightarrow$, the least relation closed under a set of the reduction rules.

$$\text{R-INTER}\ \frac{}{(\overline{a}\langle b\rangle.P_1 + M_1) \mid (a(c).P_2 + M_2) \rightarrow P_1 \mid P_2\{b/c\}}$$

$$\text{R-PAR}\ \frac{P_1 \rightarrow P_1'}{P_1 \mid P_2 \rightarrow P_1' \mid P_2} \qquad \text{R-RES}\ \frac{P \rightarrow P'}{(c)P \rightarrow (c)P'} \qquad \text{R-TAU}\ \frac{}{\tau.P + M \rightarrow P}$$

$$\text{R-STRUCT}\ \frac{P_1 \equiv P_2 \rightarrow P_2' \equiv P_1'}{P_1 \rightarrow P_1'} \qquad \text{R-CONST}\ \frac{}{K\lfloor\tilde{b}\rfloor \rightarrow P\{\tilde{b}/\tilde{a}\}}\ K \triangleq (\tilde{a}).P$$

- An **abstraction** of arity $n \geq 0$ is an expression of the form $(a_1, \ldots, a_n).P$, where the a_i are distinct.
- A **pseudo-application** of an abstraction $F \stackrel{def}{=} (\tilde{a}).P$ is an expression of the form $F\langle\tilde{b}\rangle$, a process $P\left\{\tilde{b}/\tilde{a}\right\}$.
- A **constant application** of a process constant $K \triangleq (\tilde{a}).P$, is an expression of the form $K\lfloor\tilde{b}\rfloor$, reducible according rule R-CONST. It allows **recursive definitions**.

Behavioural Description of Services in SOA

- Behaviour of the testing environment:

$$
\begin{aligned}
System &\stackrel{def}{=} (st, rl).(et, ar, lr, pe, fe) \\
&\quad (TM\langle st, fe, lr\rangle \mid TE\langle et, ar, pe\rangle \mid TL\langle lr, rl\rangle \mid TEB\langle pe, fe\rangle)
\end{aligned}
$$

- Behaviour of “TestEnvironmentBroker” service:

$$
\begin{aligned}
TEB &\stackrel{def}{=} (pe, fe).(q)(TEB_{pub}\lfloor q, pe\rfloor \mid TEB_{find}\lfloor q, fe, pe\rfloor) \\
TEB_{pub} &\triangleq (t, pe).pe(i, d).(t')(\overline{t}\langle t', i, d\rangle \mid TEB_{pub}\lfloor t', pe\rfloor) \\
TEB_{find} &\triangleq (h, fe, pe).h(h', i, d).(TEB_{find}\lfloor h', fe, pe\rfloor \mid (\overline{fe}\langle i\rangle.\overline{pe}\langle i, d\rangle \ + \ d))
\end{aligned}
$$

- Behaviour of “TestEnvironment” service:

$$
\begin{aligned}
TE &\stackrel{def}{=} (et, ar, pe).TE_{init}\langle et, ar, pe\rangle.TE_{impl}\langle et, ar\rangle \\
TE_{impl} &\stackrel{def}{=} (et, ar).(s_0, s_1, ar^s, et^g) \\
&\quad (\overline{ar^s}\langle ar\rangle \mid (d, t)(\overline{et^g}\langle t\rangle.t(p).Wire\lfloor et, p, d\rfloor) \mid TE_{comp}\langle s_0, s_1, et^g, ar^s\rangle \\
TE_{init} &\stackrel{def}{=} (et, ar, pe).\overline{pe}\langle et, ar\rangle
\end{aligned}
$$

- . . . see the conference proceedings. . .

Behavioural Description of Components

- Interface references and binding, import and export, control of the component's life-cycle, in component "testEnvironment":

$$
\begin{aligned}
TE_{comp} \overset{def}{=}\ & (s_0, s_1, p^g_{executeTest}, p^s_{asyncRepltET}).(p_{executeTest}, r_{teExecTest}, \\
& p^s_{teExecTest}, r_{asyncRepltET}, p_{teReply}, p^g_{teReply}, p_{teAttach}) \\
& (Ctrl_{Ifs}\langle p_{executeTest}, p^g_{executeTest}\rangle \mid Ctrl_{Ifs}\langle r_{teExecTest}, p^s_{teExecTest}\rangle \\
& \mid Ctrl_{Ifs}\langle r_{asyncRepltET}, p^s_{asyncRepltET}\rangle \mid Ctrl_{Ifs}\langle p_{teReply}, p^g_{teReply}\rangle \\
& \mid Ctrl_{EI}\langle p_{executeTest}, r_{teExecTest}\rangle \mid Ctrl_{EI}\langle p_{teReply}, r_{asyncRepltET}\rangle \\
& \mid Ctrl_{SS}\langle s_0, s_1, p_{teAttach}\rangle \mid TE'_{comp}\langle p_{teAttach}, p^s_{teExecTest}, p^g_{teReply}\rangle)
\end{aligned}
$$

- Core behaviour of composite component "testEnvironment":

$$
\begin{aligned}
TE'_{comp} \overset{def}{=}\ & (p_{teAttach}, p^s_{teExecTest}, p^g_{teReply}).\ldots \\
& (Ctr\langle s^{ctr}_0, s^{ctr}_1, p^g_{cDone}, p^g_{teExecTest}, r_{teAttach}, r_{detachTest}, r_{stopTest}, \\
& \quad r_{provRefEInt}, r_{provRefORes}\rangle \mid Env\langle s^{env}_0, s^{env}_1, p^g_{eInteract}\rangle \\
& \mid Out\langle s^{out}_0, s^{out}_1, p^g_{oResult}, p^s_{oDone}, p^s_{oReply}\rangle \mid \ldots)
\end{aligned}
$$

- …see the conference proceedings…

Current Results and Future Work

Current Results

- The behaviour is described as a single π-calculus process abstraction.
  (e.g. process abstraction $(st, rl).System$)
- It describes dynamic architecture with component mobility.
  (e.g. service “TestEnvironmentBroker”, component “test” in “testEnvironment”)
- Evolution of the architecture can be invoked by functional requirements.
  (e.g. processing test scripts invoke changes in component “testEnvironment”)
- Verification of properties of the behaviour and model-checking in SOA.
  (ensures compatibility of services, limits evolution of architecture, etc.)

Further work

- Integration with modelling tools, based on metamodel.
- Design-time verification and model-checking, service and component modelling with constraints.

Thank you for your attention!