LABORATOIRE D'INFORMATIQUE DE NANTES ATLANTIQUE

Composing Components with Shared Services in Kmelia

P. André, G. Ardourel, C. Attiogbé

LINA UMR CNRS 6241

March, 2008

Outline

- Introduction
- An overview of the Kmelia Component
  - Service Specification and Service Composition
- Multipart Interaction
  - Shared Services and Composition
  - Architecture, composability
- Illustration
- Summary and Perspectives

Context of the work

The context of the Kmelia Model: sound basis

- to develop correct software within CBSE (components, composition)
- to propose techniques for property verification.

The goal of the Kmelia Project:

- to provide developers with component models and guidance,
- to build practical toolbox (COSTO Toolbox).

Motivation

One missing feature:

- User-friendly multipart communication
  (for weakly coupled interacting systems)

For instance, modelling a chat system:

- One server, several asynchronous clients
- Synchronisation between the server and the current clients
- Clients may join the discussion at any time
- The clients share the server services

$\Rightarrow$ Motivation for the extension of the Kmelia model.

Our approach of the problem

The article deals with:

sharing services/component composition/multipart interactions:

- extension of the Kmelia component model with shared services
- extension of the language with additional interaction primitives

Policy

- keep the Kmelia model simple, extensible, mechanisable
  - building on existing formal basis: services
  - extending the existing framework: COSTO toolbox

Kmelia Abstract Component Model (from 2006)

```
Component C1
Interface       <Interface descr>
Types           <Type Defs>
Variables       <Var list>
Invariant
                <Predicate>
Initialisation
  ...           // var. assignments
Services
  ...           // described at side
end
```

```
Provided aService_1 ()
  Interface     <Interface descr>
  Pre           <Predicate>
  Post          <Predicate>
  Behaviour     // eLTS
  init          aStateI
  final         aState
  { state_i —label—> state_j
    ... }
end
Required aService_2 ()
    ...         // in the same way
```

Service Behaviour Specification: an eLTS

- States, initial state, final states
- Transitions: `source--label-->target`

```
label ::= [guard] actions*
```

Actions:

- elementary action
- communication: *service call/response* or message *communication*.

```
Communication ::= channel(!|?|!!|??)message(param*)
```

```
Channel ::= SELF | CALLER | RequiredServiceName
```

The behaviour of a service *ss* $\langle S_{ss}, L_{ss}, \delta_{ss}, \Phi_{ss}, S_{0_{ss}}, S_{F_{ss}} \rangle$.

Service Composition

Service Composition via

- Horizontal Structuring: Interaction between Services
  - linking required and provided services
    (internally, by the caller, by a third component)
  - service calls/responses handled with communication mechanisms:
    !! for a service call `channel!!message(param*)`
    ?? for a service wait `channel??message(param*)`
- Vertical Structuring: State Annotation, Transition Annotation

Composition with Shared Services

- Previous version of Kmelia

  one-to-one connection
  simple architecture
  behavioural property verification

- New version

  one-to-many connection
  Impact on architecture, communications, verification

Composition with Shared Services/ new features

New features

- Making explicit the use of:
  - Component type/ Component

    c: CT, c[n] : CT
  - Assembly type/Assembly
- Introducing Shared Provided/Required Services with the related communication aspects

Architectural Aspects

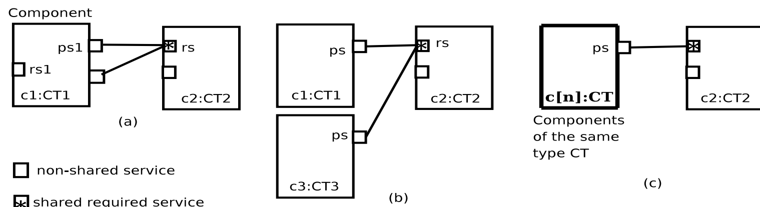

Figure: Shared Required Services

Architectural Aspects

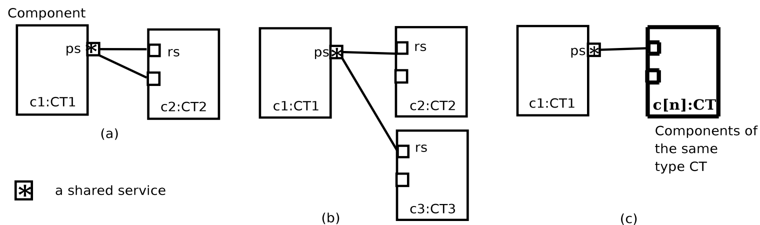

Figure: Shared Provided Services

Communication aspects (Service Interaction)

- Standard binary communication operators do not suffice
- Handling multipart interaction (between required/provided)
- Synchronisation of several services instead of two
- Need of specific communication primitives

Communication aspects

The communication actions have been extended:

`channel[<selector>](!|?|!!|??)message(param*)`

The values of `<selector>` are: `ALL`, `i` and `:i`.

`CALLER[i]!msg(val)`	Emission of `msg(val)` to the caller `i`
`CALLER[ALL]!msg(val)`	Broadcast of `msg(val)` to all the callers
`CALLER[i]?msg(x: x_Type)`	Reception of a value from the caller `i`
`CALLER[:i]?msg(x: x_Type)`	Reception of a value from any caller `i`; the other received values are not taken into account.
`CALLER[ALL]!!subSerb(val)`	Broadcast of a sub-service call to the callers
`tab_x := CALLER[i]??subServ(x: x_Type)`	Wait the return of a sub-serv. from the caller `i`
`CALLER[ALL]??subServ(x:x_Type)`	Wait the return of all sub-services from all the callers
...	...

Table: Communication actions from a shared provided service to its callers

Multipart Interaction

Compatibility of behaviours: the correct interactions between the caller service and the called service behaviours (B_i, B_j).
Simultaneous state-based examination of two flattened LTS.

From current states, the output transitions are checked:
- Interleaving of independent actions
- Matching actions (with identical channel):

send(!)	receive(?)
call service(!!)	start service(??),
emit service result(!!)	get service result(??)

until final states without blocking(deadlock)

This algorithm *compatible*(B_i, B_j) is now extended to several interacting services

Multipart Interaction : verification

For each component C_i, the triple (s_i, req, s_j) constitutes the analysis context to check each service of C_i.

- s_i a service of C_i, req one required service of s_i, B_i the behaviour (labelled transitions) of s_i.
- A component C_j, one service s_j that is linked to req; B_j the behaviour of s_j.

To check the composability at service and component level, we consider one service s_i, one required service req of s_i, and S_J the set of services linked to req: (s_i, req, S_J).

Behavioural Compatibility: verification

The behavioural compatibility of (s_i, *req*, S_J), results in:
i) checking (s_i, *req*, s_j) for each $s_j \in S_J$;

$$\text{compatible_gen}(s_i, S_J) \Leftrightarrow \forall s_j \in S_J \mid \ compatible(B_i, B_j)$$

ii) checking *one-to-n matching* between s_i and S_J.
the following matching conditions are required:

s_i **performs**	**each** s_j **in** S_J **performs**
`_req[ALL]?msg(...)`	`CALLER!msg(...);`
`_req[ALL]!msg(...)`	`CALLER?msg(...);`
`_req[ALL]??srv(...)`	`CALLER!!srv(...);`
`_req[ALL]!!srv(...)`	`CALLER??srv(...)`

Synchronous multipart communication

Formally, a synchronous communication between n entities.
The formal specification of one-to-n_matching(s_i, S_J):

$$\frac{\begin{array}{c} s_i \mathrel{\widehat{=}} \langle S_{s_i}, L_{s_i}, \delta_{s_i}, \Phi_{s_i}, \{cst_i\}, S_{F_{s_i}} \rangle \ \wedge \\ ((cst_i,\ \texttt{_req[ALL]?msg(...)}),\ nst_i) \in \delta_{s_i} \wedge \\ \forall s_j \in S_J \mid s_j \mathrel{\widehat{=}} \langle S_{s_j}, L_{s_j}, \delta_{s_j}, \Phi_{s_j}, \{cst_j\}, S_{F_{s_j}} \rangle \ \wedge \\ ((cst_j,\ \texttt{CALLER!msg(...)}),\ nst_j) \in \delta_{s_j} \end{array}}{\text{one-to-n_matching}(s_i, S_J)}$$

Then the behavioural compatibility is generalised to (s_i, req, S_J) with:

$$\frac{\text{compatible_gen}(s_i, S_J) \wedge \text{one-to-n_matching}(s_i, S_J)}{\text{beh_compatible_gen}(s_i, S_J)}$$

Illustration

```
COMPONENT CHAT_SRV
INTERFACE
provided: {connection, interac-
tion}
required: {}
SERVICES
provided connection()
...
shared provided interaction()
// sends 'news'
// receives 'msg', 'close'
...
news ()
...
END_SERVICES
```

```
COMPONENT CHAT_CLT
INTERFACE
provided: {chat_session}
required: {interaction}

SERVICES

required interaction()
// receives 'news'
// sends 'msg', 'close'
...

provided chat_session()
...
END_SERVICES
```

Illustration

The assembly is specified in Kmelia as follows.

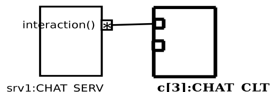

Figure: Chat System : Architecture

```
COMPOSITION
{    srv1: CHAT_SERV
clt[3]: CHAT_CLT
}
{  (p-r srv1.interaction,
clt[3].interaction)

}
```

Illustration

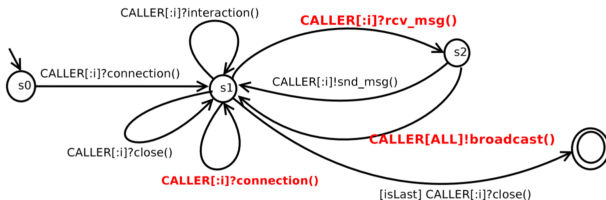

Figure: A part of the behaviour of the `interaction` service of the chat server

Illustration

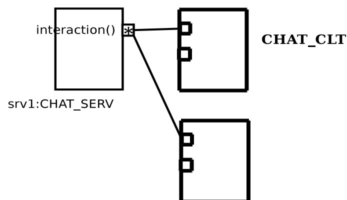

Figure: Possible architecture

explicit management of channels wrt linked clients

Conclusion

Summary

- The Kmelia model is extended to deal with shared services
- Communication actions are (re)defined
- The Kmelia model is updated and supports multipart interaction
- Behavioural property analysed

Ongoing Work

- Service/Component interruption
- Extending the COSTO Toolbox to deal with shared services
- Redefinition of the bridge with Lotos n-ary parallel composition
- Extending the Kmelia data language

Thanks for your attention!

Questions, Please