SERVICE BROKERAGE IN PROLOG

Cheun Ngen Chong, Sandro Etalle, and Pieter Hartel
University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands.

Rieks Joosten and Geert Kleinhuis
TNO Telecom Groningen
P.O. Box 15000, 9700 CD Groningen, The Netherlands.

Keywords: Service brokerage, rights expression language, conceptual modelling, service management.

Abstract: Service brokerage is a complex problem. At the design stage the semantic gap between user, device and system requirements must be bridged, and at the operational stage the conflicting objectives of many parties in the value chain must be reconciled. For example why should a user who wants to watch a film need to understand that due to limited battery power the film can only be shown in low resolution? Why should the user have to understand the business model of a content provider? To solve these problems we present (1) the concept of a packager who acts as a service broker, (2) a design derived systematically from a semi-formal specification (the CC-model), and (3) an implementation using our Prolog based LicenseScript language.

1 INTRODUCTION

A *service* is a combination of an application and its maintenance. The application implements the functionality required, e.g. making available a communication channel, playing a song. The maintenance ensures availability e.g. fast delivery, high bandwidth, 24 hour access. Services are characterized by a wide variety of parameters, for example the capability of the service delivery (e.g. bandwidth), and the restrictions on the service usage (e.g. device limitation). These parameters make service brokerage a complex problem.

Users have a wide variety of service demands (e.g. to use their services anywhere and anytime); on the other hand, service providers have their own requirements (e.g. to control user's access rights). We present the concept of a packager who acts as a service broker, and we present an implementation as part of the *Residential Gateway Environment* (RGE) project (Joosten et al., 2003). Our contribution is two-fold: (1) During the design stage, we show how to derive the complex infrastructure for the service management from a semi-formal high-level description: the "Calculating with Concepts" (CC) method (Dijkman et al., 2001). We encode all aspects of service brokerage in LicenseScript (Chong et al., 2003). (2) During the operational stage, we show how LicenseScript handles the diverse requirements of all parties involved.

LicenseScript is based on Prolog and multiset rewriting and allows one to express *licenses*, i.e. conditions of use on dynamic data. Prolog has the advantage of combining an operational semantics (needed, e.g., in negotiations) with a straightforward declarative reading. Our addition of multiset rewriting to Prolog allows to encode in an elegant and semantically sound way the *state* of a license. The semantics of LicenseScript is given in terms of traces (Chong et al., 2003).

Section 2 introduces the overall infrastructure of the RGE service management and its CC model. Section 3 derives LicenseScript from the CC model. The last section concludes and presents future work.

2 CC MODEL OF RGE

We present the overall infrastructure of RGE service management together with the CC method. The RGE architecture supports three main roles: the residential gateway (RG), the packager (P) and the service providers (SP). Service providers provide services (S), e.g. access to music, videos, but also bandwidth. The packager behaves as a service broker, being able to manipulate and integrate the services provided by the various SPs. The residential gateway is where the services actually run. A power user (PU) of the RG is allowed to (un)subscribe to services. All users (U)

Ngen Chong C., Etalle S., Hartel P., Joosten R. and Kleinhuis G. (2005).
SERVICE BROKERAGE IN PROLOG.
In *Proceedings of the Seventh International Conference on Enterprise Information Systems*, pages 409-412
DOI: 10.5220/0002517804090412
Copyright © SciTePress

are allowed to use the services.

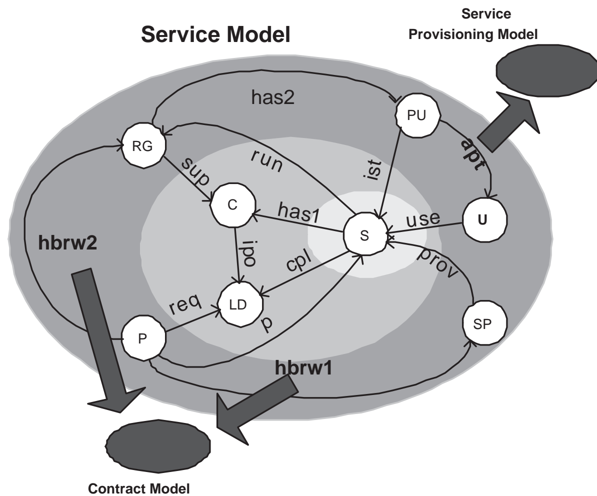

Figure 1: Three of many CC models of the RGE service management infrastructure.

To develop the RGE infrastructure, den Hartog et al. (den Hartog et al., 2004) use the *Calculating with Concepts (CC)* method, which can be seen as an extension of Entity-Relationship diagrams. The basic ingredients of a CC model are (a) entities, (b) relations and (c) restrictions. The rationale behind the CC-method is that every engineer involved in a project has a different interpretation of the system requirements. The CC method is then used in group discussions to iron out these differences, and thus help to develop a consistent frame of reference. Figure 1 presents a simplified CC-model for the RGE architecture centered on service brokerage. There are many other, similar models centering on other, relevant aspects. The models are related through the use of a common vocabulary for entities, relations and restrictions.

Table 1: The CC entities of the Service Model

	Entity
C	Service characteristic, e.g. the quality, etc
LD	List of demands, which a service must comply with
P	Packager
RG	Residential gateway
S	Service
SP	Service provider
U	Normal user who uses the service
PU	Power use who possesses the administrative power on RG

The roles of the RGE service management infrastructure are represented by the CC entities (see Table 1). These roles interact through the entities service (S), characteristics (C), and list of demands (LD). Relationships and restrictions are described in further detail in 2 and 3, respectively. The restrictions listed in Table 3 largely determine the semantics of the CC model (Joosten et al., 2003).

Table 2: The relations between the entities of the Service Model

Abbr.	Relation
apt	Power user assigns permission(s) to user.
cpl	Service complies with list of demands.
has1	Service has characteristic.
has2	Residential gateway has (is owned by) power user.
hbrw1	Packager has a business relation with service provider.
hbrw2	Packager has a business relation with residential gateway.
ipo	Characteristic is part of list of demands.
ist	Power user has subscribed to service.
p	Packager permits service.
prov	Service provider provides service.
req	Packager requires list of demands.
run	Service runs on residential gateway.
sup	Residential gateway supports characteristic.
use	User uses service.

Table 3: The CC restrictions of the Service Model

Restrictions
Every user is created by one and only one power user.
Every user has been assigned permissions by one and only one power user.
Every residential gateway has one and only one power user.
Every packager permits at least one service.
Every packager has a business relation with at least one service provider.
Every packager has a business relation with at least one residential gateway.
Every service is provided by at least one service provider.
Every service has at least one characteristic.
For every list of demands, there is at least one characteristic that is part of that list of demands.
Every packager requires at least one list of demands.

3 LICENSESCRIPT DERIVATION

We now briefly introduce the LicenseScript language, then we will show how to derive a LicenseScript specification from the CC model just presented. LicenseScript (Chong et al., 2003) is a formalism that can be used to specify access control and manipulation of licenses on digital content like music, video, software etc. The unique feature of LicenseScript is that licenses actually carry Prolog code (representing access and usage conditions) together with bindings, that can be used to store the *state* of the license.

In LicenseScript we work with *objects* (licenses) and *rules*. Objects have the form:

```
object_name(Content,Clauses,Bindings)
```

Here `object_name` is the name of the object; `Content` is a content identifier which is associated to this object; `Clauses` is a set of Prolog clauses, and `Bindings` is a set of attributes pertaining to the object. Rules have the form:

```
rule_name(arguments):
    lhs -> rhs     <== Condition
```

Here, `lhs` and `rhs` are multisets of objects. `Condition` is a logical formula that may refer to the clauses defined in the objects contained in `lhs`. Because of this, rules are second-order constructs; objects are first order.

Intuitively, objects are pieces of enhanced (mobile) Prolog code, while rules are there to manipulate the objects and to query the code they carry. Rules are *not* mobile, and can be thought of as being the interface between the devices and the mobile code. The use of multiset rewriting allows us to model in a logical yet effective way the presence of mutable resources that not only can be modified, but also created and destroyed.

We now show how to derive LicenseScript code from the CC model. We propose a set of informal derivation rules to map the various CC components onto LicenseScript objects and rules, and/or the content, clauses and bindings of the objects:

1. We start from the service (entity *S*, inside the innermost circle of Figure 1) because this is the central entity in RGE service management. Each instance of *S* is then mapped onto the *content* part of an appropriate LicenseScript object.
2. Entities are split into two groups:
   - *Objects*, i.e., the entities on which actions are being performed (*C* and *LD* in the middle circle). These are mapped onto LicenseScript *objects*.
   - *Subjects*, i.e., the entities that perform actions upon the objects (*RG*, *P*, *SP*, *U*, and *PU* in the outer circle). These are mapped onto LicenseScript *bindings*.
3. Relations between the entities are mapped onto LicenseScript *clauses*, the body of which must reflect the cardinality restrictions of the relation.
4. Restrictions are captured by LicenseScript *multiset rewrite rules*.

Objects *LD* becomes `demands(S,C,B)` while *C* is mapped onto `characteristics(S,C,B)`. In addition, to communicate with the external world, (Contract Models and Service Provisioning Models, in Figure 1), we use the objects `license(S,C,B)` and `contracts(S,C,B)`, where `S` represents the service; `C` denotes a set of clauses; and `B` is a set of bindings.

Clauses In principle, derivation rule #3 maps each CC relation onto a separate clause. To improve efficiency, we map more than one CC relation onto a single clause. For instance, we use the clause `cangrant(·)` to capture both relations *has2* and *apt*. This clause allows the power user to assign the license (i.e. grants the usage permissions) to normal users:

```
cangrant(Blic1,Blic1',Blic2,Poweru,User) :-
    get_value(Blic1,power_user,Pu),
    authenticate(Pu,Poweru),
    set_value(Blic1,user,User,Blic1').
```

Here `Blic` and `Blic1'` are `bindings`. To access these bindings we use the primitives below to get (resp. set) the value associated with `Name` in `Bindings`:

```
get_value(Bindings,Name,Value)
set_value(Bindings,Name,Value,NewBindings)
```

As a second example, the clause `canuse(·)` allows to capture the relations *run* and *use*. `canuse` authenticates and checks that the service actually runs on the residential gateway (the binding `enabled`):

```
canuse(Blic,Blic',User) :-
    get_value(Blic,user,U),
    get_value(Blic,enabled,E),
    E == true, authenticate(U,User).
```

We conclude this part by showing a more complex example. `canpermit(·)` authenticates the packager to ensure its genuineness, before enabling the service to the residential gateway; relations *p* and *ist* are captured here.

```
canpermit(Bdem,Bdem',Blic,Clic,Pack) :-
    get_value(Bdem,packager,P),
    get_value(Bdem,service_provider,S),
    get_value(Bdem,license_clauses,Clic),
    authenticate(Pack,P),
    set_value(Bdem,enabled,true,Blic).
```

Rules Rules provide the necessary interface between the outside world and the LicenseScript objects. The simplest example of rule is `use`, which is invoked by the user to actually use a service. The rule just has to check for the presence of a license:

```
use(Service,U) :
    license(Service,Clic,Blic1)
->  license(Service,Clic,Blic2)
<=  Clic |- canuse(Blic1,Blic2,U)
```

`canuse(Blic1,Blic2,U)` is queried in `Clic` (a failure of the query would indicate that the license is no longer valid; e.g. it might have expired); after successful completion of the query, the license is replaced by another one with a the new set of bindings `Blic2`.

A more complex rule is `grant`, which duplicates a license. The power user would execute `grant` to grant some permissions/rights to a normal user:

```
grant(Service,U1,U2) :
    license(Service,Clic,Blic1),
->  license(Service,Clic,Blic1'),
    license(Service,Clic,Blic2)
<=  Clic |-  cangrant(Blic1,Blic1',Blic2,U1,U2)
```

This rule generates a new `license(·)` for the user.

Finally we present the rule `permit`, with which the packager generates a license for some service to be run on the residential gateway:

```
permit(Service,P,S) :
    demands(Service,Cdem,Bdem),
    characteristics(Service,nil,Bcha)
->  demands(Service,Cdem,Bdem')
    characteristics(Service,nil,Bcha'),
    license(Service,Clic,Blic)
<=  Cdem |-  canpermit(Bdem,Bdem',Blic,Clic,P),
    Cdem |-  cancomply(Bdem,Bdem',Bcha,Bcha')
```

The object `demands(Service,Cdem,Bdem)` indicates that a user has requested `Service`; `Cdem` and `Bdem` are respectively a set of clauses and a set of bindings that – combined – specify extra side conditions such as the maximum bandwidth, the price the user is willing to pay, etc. By calling `canpermit`, the packager checks if permission can be granted. `canpermit` also returns the clauses that will be used in the new license. On the other hand, `cancomply` validates the service request (See below).

3.1 Service Requirements Validation

We now define how the packager validates a service request, first using a simplified version of the `cancomply(·)` clause:

```
cancomply(Bdem,Bdem',Bcha,Bcha') :-
    get_value(Bdem,bandwidth,X1),
    get_value(Bcha,bandwidth,X2),
    get_value(Bdem,quality,Y1),
    get_value(Bcha,quality,Y2),
    get_value(Bdem,billing,Z1),
    get_value(Bcha,billing,Z2),
    X1 >= X2, Z1 = Z2, Y1 =< Y2.
```

The last line shows the use of constraints to ensure that the maximum bandwidth of the user's device meets the minimum bandwidth required for the service; that the billing status of the user meets the requirement of the service provider; and that the quality measure required by the user does not exceed the offered quality.

Alternatively, one can use a parametric approach, in which the list of requirements to be complied with is stored in the license:

```
cancomply(Bdem,Bdem',Bcha,Bcha') :-
    get_value(Bcha,requirements,Reqs),
    meets_requirements(Reqs).
meets_requirements([]).
meets_requirements([[Reqn,Reqv]|Reqs]):-
    check_requirement(Reqn,Reqv),
    meets_requirements(Reqs).
```

Recall that (see rule `permit` above) the query `cancomply` is fired in the set of clauses `Cdem` specified in the user's demand `demands(Service,Cdem,Bdem)`. Therefore `cancomply` can check that the service specification meets the the constraints set out in the user's demand.

4 CONCLUSIONS AND FUTURE WORK

We present one of the central concepts of the LicenseScript-RGE demonstrator, i.e. the *packager*, which acts as a service broker. We derive its implementation in our Prolog based LicenseScript language, using a systematic derivation from a semi-formal specification (the CC-model).

The combination of Prolog and multiset rewriting proves to be a very suitable platform for implementing a complex broker such as the one we have presented, in particular: (1) To represent complex services in a flexible and efficient manner one needs to employ executable (mobile) code of some kind. (2) To manipulate services it is therefore necessary to employ a second-order system. Prolog is perfect for this.

REFERENCES

Chong, C. N., Corin, R., Etalle, S., Hartel, P. H., Jonker, W., and Law, Y. W. (2003). LicenseScript: A novel digital rights language and its semantics. In Ng, K., Busch, C., and Nesi, P., editors, *3rd International Conference on Web Delivering of Music (WEDELMUSIC)*, pages 122–129, Los Alamitos, California, United States. IEEE Computer Society Press.

den Hartog, F. T. H., Baken, N. H. G., Keyson, D. V., Kwaaitaal, J. J. B., and Snijders, W. A. M. (2004). Tackling the complexity of Residential Gateway in an unbundling value chain. In *Proceedings of XVth International Symposium on Services and Local AccesS (ISSLS 2004)*, page Published Electronically. IEE.

Dijkman, R. M., Pires, L. F., and Joosten, S. M. M. (2001). Calculating with Concepts: a technique for the development of business process support. In Evans, A., France, R., Moreira, A., and Rumpe, B., editors, *Proceedings of the UML 2001 Workshop on Practical UML-Based Rigorous Development Methods*, volume 7 of *Lecture Notes in Informatics*, pages 87–98. GI-Edition.

Joosten, R., Knobbe, J.-W., Lenoir, P., Schaafsma, H., and Kleinhuis, G. (2003). Specifications for the rge security architecture. Technical Report Deliverable D5.2 Project TSIT 1021, TNO Telecom and Philips Research, The Netherlands.