Service Brokerage with Prolog

Cheun Ngen Chong[1], Sandro Etalle[1,3], Pieter Hartel[1],
Rieks Joosten[2], and Geert Kleinhuis[2]

[1] University of Twente, P.O. Box 217, 7500 AE Enschede
{chong,etalle,pieter}@cs.utwente.nl
[2] TNO Telecom Groningen, P.O. Box 15000, 9700 CD Groningen
{H.J.M.Joosten,G.Kleinhuis}@telecom.tno.nl
[3] Center for Mathematics and Computer Science (CWI), P.O. Box 94079, 1090 GB Amsterdam
Sandro.Etalle@cwi.nl

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 [17], for example the capability of the service delivery (e.g. bandwidth), the flexibility of the service access (e.g. availability of the service 24 hours), 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 requirements. For instance, they want to: have their services delivered promptly and installed properly; use their services anywhere and anytime; have a wide choice of services with various prices, qualities, etc; and they want to ensure that the technical limitations of their devices are taken into account when they acquire (purchase, lease etc) the service. In addition, users would like to protect their privacy. On the other hand, service providers have their own requirements. They need to have their services published, promoted, and more importantly paid for promptly. They also need to control the access rights of the services according to contracts established with the users. Therefore, with these different requirements from both sides of the value chain, service management becomes a complex issue.

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 [14,

13]. 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 Concept" (CC) [9]. We encode all aspects of service brokerage in LicenseScript [3, 4] using logic programming. (2) During the operational stage, we show effi ciently 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. Elsewhere, Prolog has also proven itself to be suitable for describing complex access policies, as demonstrated by the security language Binder [8]. 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 [4]. Here, we demonstrate the practical value of LicenseScript by using it as intelligent messaging middleware for the RGE project. The result is a large distributed software platform which we describe in this paper. The platform consists of the following main components (we will elaborate these components later in section 4): Tomcat Server, MySQL database, JDBC database interface, etc: 50 JSP (Java Server Page) fi les, 20 SWF (Shockwave Flash) fi les. In addition we have the Prolog-based components: the ECL^{i}S^e Prolog inference engine, the LicenseScript meta-interpreter, Java user interface and RMI interfacing with RGE components: 6 Java and 2 Prolog source fi les.

Section 2 introduces the overall infrastructure of the RGE service management and its CC model. Section 3 derives LicenseScript from the CC model. Section 4 describes the RGE implementation. Section 5 discusses related work and the last section concludes and presents future work.

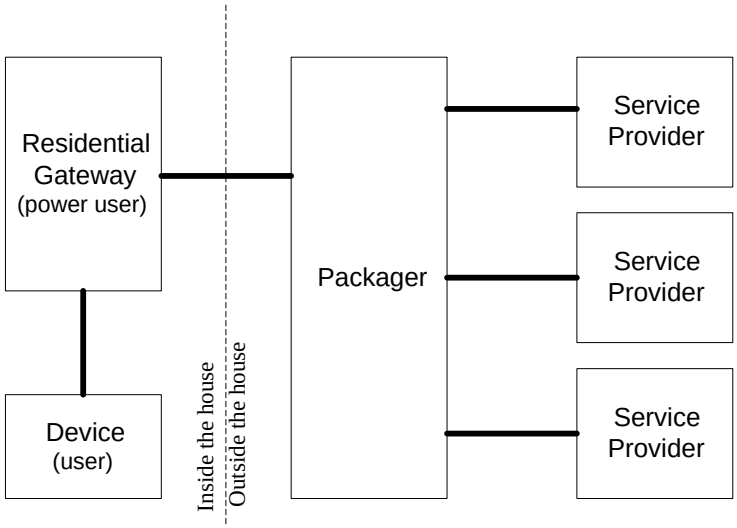

Fig. 1. The service management architecture of the RGE.

2 RGE Service Management Infrastructure

We present the overall infrastructure of RGE service management together with the CC method. As shown in Figure 1, 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) are allowed to use the services.

The packager has some control (on behalf of the SPs) over the services that are made available on the residential gateway. More importantly, the packager tries to match service characteristics (C) to user demands (LD). To achieve this, the packager has to have a business relation with the RG on one hand and with the service providers at the other end of the value chain.

In the rest section, we briefly introduce the CC method and we specify the RGE service management infrastructure.

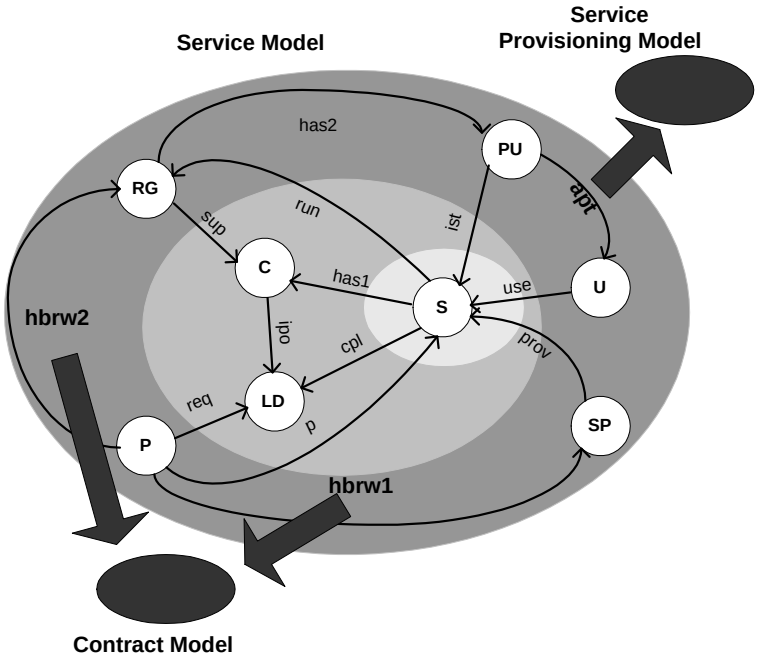

Fig. 2. Three of many CC models of the RGE service management infrastructure.

2.1 CC Model

To develop the RGE infrastructure, Hartog et al [7] 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 2 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.

The roles of the RGE service management infrastructure are represented by the CC entities RG, SP, PU, U, and P (see Table 1). These roles interact through the entities service (S), characteristics (C), and list of demands (LD). Entities, relationships and restrictions are described in further detail in Tables 1, 2 and 3, respectively. Notice that the restrictions listed in Table 3 largely determine the semantics of the CC model. For details of the CC model derivation process, the reader may refer to [14].

Abbr.	**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

Table 1. The CC entities of the Service Model.

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 [3] 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)
```

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 2. The relations between the entities of the Service Model.

#	Cardinality Restrictions
1	Every user is created by one and only one power user.
2	Every user has been assigned permissions by one and only one power user.
3	Every residential gateway has one and only one power user.
4	Every packager permits at least one service.
5	Every packager has a business relation with at least one service provider.
6	Every packager has a business relation with at least one residential gateway.
7	Every service is provided by at least one service provider.
8	Every service has at least one characteristic.
9	For every list of demands, there is at least one characteristic that is part of that list of demands.
10	Every packager requires at least one list of demands.
#	**Other Restrictions**
11	If a service s complies with list of demands ld, and characteristic c is part of ld, then s has c.
12	If a user u uses service s, then u has been assigned a permission by power user pu that is subscribed to s.

Table 3. The CC restrictions of the Service Model.

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. Consider as an example the following rule:

```
offer(Service,S,P) :
    contracts(Service,Ccon,Bcon),
    characteristics(Service,nil,Bcha1)
->  contracts(Service,Ccon,Bcon),
    characteristics(Service,nil,Bcha1'),
    characteristics(Service,nil,Bcha2)
<=  Ccon |- canoffer(Bcon,Bcha1,Bcha1',Bcha2,S,P)
```

This rule rewrites the multiset:

```
contracts(Service,Ccon,Bcon),
characteristics(Service,nil,Bcha1)
```

into the following multiset:

```
contracts(Service,Ccon,Bcon),
characteristics(Service,nil,Bcha1'),
characteristics(Service,nil,Bcha2)
```

The proviso is that the query `canoffer(·)` succeeds when fired in the set of clauses `Ccon`.

Deriving Licensescript

We now show how to derive LicenseScript code from the CC model. We propose a set of derivation rules to map the various CC components (i.e. entities, relations and restrictions) 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) 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:
   - Those that have a *direct* relationship with *S* (*C* and *LD* in the middle circle) are mapped into LicenseScript *objects*.
   - Those that have an *indirect* relationship with *S* (*RG*, *P, SP* and *U*, in the outer circle) are mapped into LicenseScript *bindings*.
3. Relations between the entities are mapped onto clauses, the body of which must reflect the cardinality restrictions of the relation.
4. Other general CC restrictions are captured by LicenseScript multiset rewrite rules.

LicenseScript Object	Description
`characteristics(S,nil,B)`	Represents characteristics of a service.
`demands(S,C,B)`	Represents list of demands required.
`license(S,C,B)`	Represents permissions/rights.
`contracts(S,C,B)`	Represents business relations.

Table 4. The LicenseScript objects representing CC entities, where `S` represents the service; `C` denotes a set of clauses; and `B` is a set of bindings.

Objects Objects are the result of mapping entities of the middle circle: *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 2), we use the objects `license(S,C,B)` and `contracts(S,C,B)`. Table 4 provides a summary. The reader may refer to the technical report [14] for more information.

Clauses In principle, derivation rule #3 maps each cardinality relation onto a separate clause. To improve effi ciency, 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).
```

The parameter `Clic` allows to extract a whole new set of clause from the bindings and to create a new LicenseScript object with it.

Other clauses are derived in the same way from the CC model. A summary of the definitions is given in Table 5.

Clause	Object	Relations	Restrictions
`cancomply(·)`	`demands(·)`	has1, cpl, ipo, sup	8, 9, 10
`canpermit(·)`	`demands(·)`	p, ist	4
`canoffer(·)`	`contracts(·)`	prov	5, 7
`canrequest(·)`	`contracts(·)`	req	6
`cangrant(·)`	`license(·)`	has2, apt	2, 3
`canuse(·)`	`license(·)`	use, run	1

Table 5. The LicenseScript clauses that capture the relations in Table 2 and the conditions of success for the restrictions in Table 3.

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,Requirements),
    meets_requirements(Requirements).

meets_requirements([]).
meets_requirements([[Req_name,Req_value]|Reqs]):-
    check_requirement(Req_name,Req_value),
    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.

Related to this, Corin et al [6] have demonstrated that LicenseScript allows one to define flexible payment policies that may be set by users. This is non-trivial as there may be more than one service provider interacting with one packager.

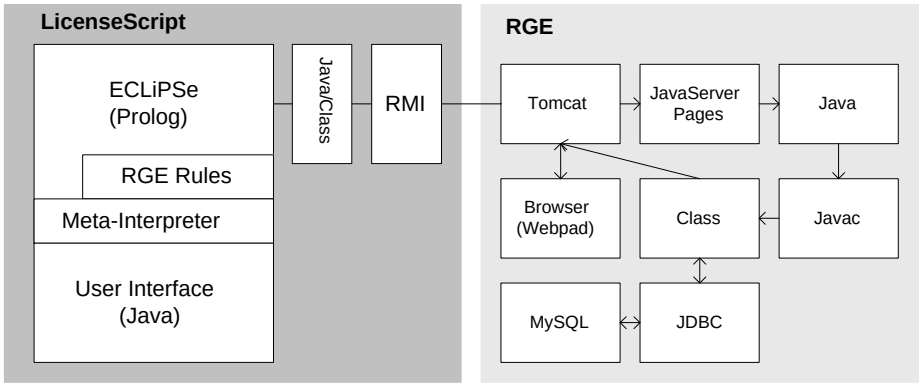

Fig. 3. Configuration of the RGE demonstrator software components.

4 RGE Demonstrator

We have finally come to the RGE demonstrator proper. As shown in Figure 3, the demonstrator contains a number of off-the-shelf software components. On the right hand side we find among others the Tomcat server and the MySQL database.

The left hand side of Figure 3 zooms in on the specific LicenseScript components. The basic one is the LicenseScript meta-interpreter, which is implemented using ECL^{i}PS^e (`http://www.icparc.ic.ac.uk/eclipse/`).

The core of the LicenseScript meta-interpreter is a simple extension of the vanilla meta-interpreter presented in [15]:

```
solve([],_).
solve([Query|Queries],Program) :-
    copy_term(Program,Temps),
    member((Query:-Body),Temps),
    solve_body(Body),
    solve(Queries,Program).

solve_body(true).
solve_body((Goal1,Goal2)) :-
    solve_body(Goal1),solve_body(Goal2).
solve_body(Goal) :-
    call(Goal).
```

The only difference lies in the fact that LicenseScript runs a query in a specific `Program`. Multiset rewriting has also been implemented in Prolog [3].

The meta-interpreter is interfaced with the Tomcat server via a Java interface, which is called by remote method invocation (RMI).

Figure 4 reports a snapshot of the (Java) user interface for the LicenseScript components. Various buttons allow the user to view the LicenseScript objects in the various devices of the multiset. The text area underneath the buttons logs and displays the status of the execution of the meta-interpreter and the ECL^{i}PS^e engine. The multiset viewer

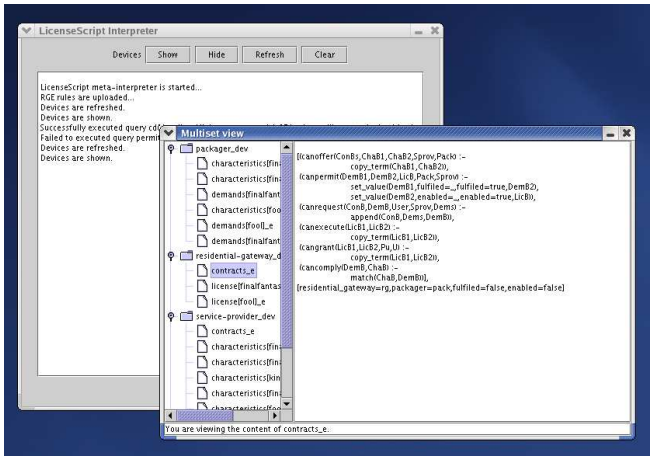

Fig. 4. The LicenseScript Interpreter user interface.

allows the users to observe the status of the LicenseScript objects before and after the execution.

Fig. 5. A dialog shows that the demands and the characteristics of the service do not match.

As shown in Figure 5, when the demands and the characteristics of the service do not match, a dialog notifi es the packager.

To conclude this section, we show a snapshot of the RGE demonstrator in Figure 6. The leftmost Web browser represents the service provider, the middle is the packager and the rightmost is the residential gateway.

5 Related Work

We discuss related work on Web service management and service brokerage in section 5.1 and section 5.2, respectively.

5.1 Web Service Management

Web services are services provided over the Internet. Web services are described using the XML-based Web Services Description Language (WSDL) (`http://www.w3.`

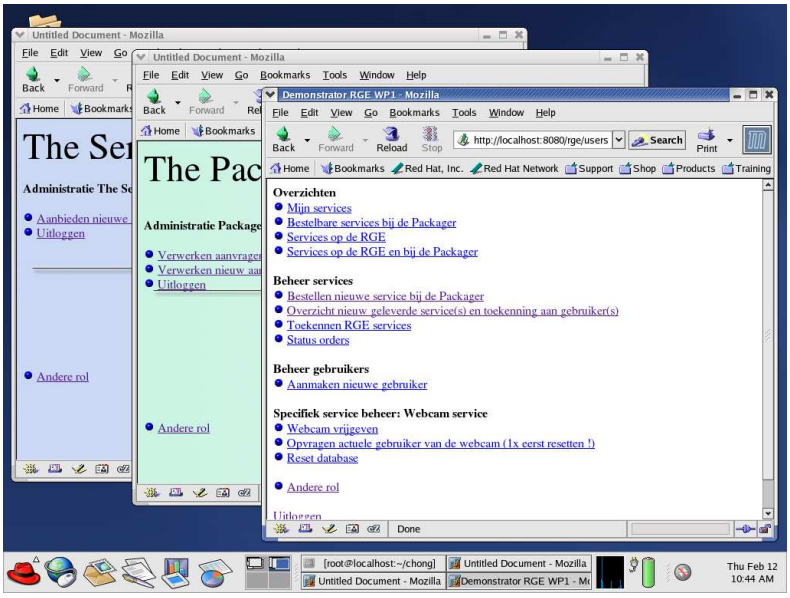

Fig. 6. The RGE demonstrator.

`org/TR/wsdl`). WSDL provides a simple way for service providers to describe the *basic format of requests* (made by the users) to their systems regardless of the underlying protocol and the message encoding format. The underlying protocol of WDSL is the Simple Object Access Protocol (SOAP) (`http://www.w3.org/TR/SOAP/`). SOAP is used to describe the envelope and message format. In addition, SOAP provides a basic request/response handshake protocol that for exchanging structured information. WSDL/SOAP have emerged as a standard for handling Web service architecture. They have been deployed in some current Web service frameworks, e.g. IBM WebSphere (`http://www.ibm.com/websphere`) and Microsoft .Net (`http://www.microsoft.com/net`).

WSDL describes the Web services merely as a set of end-points (of the connections) operating on the messages. This makes it diffi cult to match parameterised service requests to offerings because matching requires flexibility in making the connections. This is precisely what LicenseScript is good at. On the other hand LicenseScript provides no facility for the description of message transmission format and message exchange protocol, as WSDL/SOAP does; these facilities are beyond the scope of LicenseScript.

As mentioned earlier, there exist several frameworks for Web service management, e.g. Java 2 Platform Enterprise Edition (J2EE) (`http://java.sun.com/j2ee`) (which is integrated in the IBM WebSphere) and Microsoft .Net. These frameworks provide facilities for *service description*, *service implementation*, *service publishing*, *discovery and binding*, as well as *service invocation and execution* [10]. In LicenseScript, we have

concentrated on service description. Actually, the overall RGE project has addressed the service implementation, service invocation and execution.

5.2 Service Brokerage

We have seen how LicenseScript can be used for *service brokerage*. We now discuss related work on this topic.

Bichler and Segev [1] present a framework for service brokerage, where the broker is represented as an *agent*. Typically, an agent is programmed to search and aggregate the information from the Internet. An advanced agent is able, for example, to perform price comparison on some products (e.g. BargainFinder [11]); or to negotiate and participate in an online auction on behalf of its owner [12].

While agents provide an elegant and convenient way to model tasks such as service brokerage, their deployment can raise privacy concerns and security problems [2]: agents need to be protected against possible malicious host, and vice versa.

Our licenses are mobile and contain code (Prolog clauses) and data (Bindings); in this sense licenses can be interpreted as agents. However, we make a number of assumptions to alleviate security problems. In particular we assume that the rules and the prolog engine can be trusted (they represent the firmware of our devices).

6 Conclusions and Future Work

We present one of the central concepts of the LicenseScript-RGE demonstrator, i.e. the *packager*, who 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).

Prolog proves to be a very suitable platform for implementing a complex broker such as the one we have presented, in particular:

- To represent complex services in a flexible and efficient manner one needs to employ executable (mobile) code of some kind. To manipulate services it is therefore necessary to employ a second-order system. Prolog is perfect for this.
- Services should not only be executable, but should have a clear and concise semantics (after all, they are *licenses*). The close relation between operational and the declarative semantics of Prolog is an invaluable advantage.
- Prolog is ideal to match requirements, and good at resolving conflicts. Therefore it is a natural platform for service brokerage.

Thanks to Prolog's expressive power – the LicenseScript engine consists of just a few dozens of lines of code. Also services (which are represented as objects, containing Prolog code), usually require only few Prolog lines to be described.

In the future, we are planning to implement the concept of authorized domain [16] in the RGE. We would also like to enhance the security of LicenseScript objects by using some tamper-resistant hardware [5].

Acknowledgement

We would like to thank the RGE project members for support, particularly Igor Passchier and Nico Zornig. This work was partially supported by the Telematica Institute.

References

1. M. Bichler and A. Segev. A brokerage framework for internet commerce. *Distributed and Parallel Databases: Special Issue on E-Commerce*, 7(2):133–148, April 1999.
2. D. M. Chess. Security issues in mobile code systems. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes for Computer Science*, pages 1–14. Springer-Verlag Berlin, 1998.
3. C. N. Chong, R. Corin, S. Etalle, P. H. Hartel, W. Jonker, and Y. W. Law. LicenseScript: A novel digital rights language and its semantics. In K. Ng, C. Busch, and P. Nesi, editors, *3rd International Conference on Web Delivering of Music (WEDELMUSIC)*, pages 122–129, Los Alamitos, California, United States, September 2003. IEEE Computer Society Press.
4. C. N. Chong, R. Corin, S. Etalle, P. H. Hartel, and Y. W. Law. LicenseScript: A novel digital rights language. In *Int. Workshop for Technology, Economy, Social and Legal Aspects of Virtual Goods*, page To appear, Ilmenau, Germany., May 2003.
5. C. N. Chong, Z. Peng, and P. H. Hartel. Secure audit logging with tamper-resistant hardware. In D. Gritzalis, S. D. C. di Vimercati, P. Samarati, and S. K. Katsikas, editors, *18th IFIP International Information Security Conference (IFIPSEC)*, volume 250 of *IFIP Conference Proceedings*, pages 73–84. Kluwer Academic Publishers, May 2003.
6. R. Corin, C. N. Chong, S. Etalle, and P. H. Hartel. How to pay in LicenseScript. Technical Report TR-CTIT-03-31, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, July 2003.
7. F. T. H. den Hartog, N. H. G. Baken, D. V. Keyson, J. J. B. Kwaaitaal, and W. A. M. Snijders. 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 TBA. IEE, March 2004.
8. J. DeTreville. Binder, a logic-based security language. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 105–113. IEEE Computer Society, March 2002.
9. R. M. Dijkman, L. F. Pires, and S. M. M. Joosten. Calculating with Concepts: a technique for the development of business process support. In A. Evans, R. France, A. Moreira, and B. Rumpe, 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, October 2001.
10. P. Fletcher, M. Waterhouse, and M. Clark. *Web Services Business Strategies and Architectures*. APress, July 2003.
11. A. R. Greenwald and J. O. Kephart. Shopbots and pricebots. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 506–511. Morgan Kaufmann Publishers Inc., 1999.
12. R. H. Guttman and P. Maes. Agent-mediated integrative negotiation for retail electronic commerce. In *Selected Papers from the First International Workshop on Agent Mediated Electronic Trading Agent Mediated Electronic Commerce*, volume 1571 of *Lecture Notes in Computer Science*, pages 70–90. Springer-Verlag, 1998.
13. B. Hillen, J. Kwaaitaal, A. van Neerbos, I. Passchier, and D. S. Rivero. Management requirements for residential gateways. Technical Report Deliverable D1.1 Project TSIT 1021, KPN Research and TU/e, The Netherlands, August 2002.

14. R. Joosten, J-W. Knobbe, P. Lenoir, H. Schaafsma, and G. Kleinhuis. Specifications for the rge security architecture. Technical Report Deliverable D5.2 Project TSIT 1021, TNO Telecom and Philips Research, The Netherlands, August 2003.
15. Leon Sterling and Ehud Shapiro. *The Art of Prolog (Second Edition)*, chapter 17, pages 319–357. The MIT Press, Cambridge, Massachusetts 02142, Uniter States, 1994.
16. S.A.F.A. van den Heuvel, W. Jonker, F.L.A.J. Kamperman, and P.J. Lenoir. Secure content management in authorised domains. In *The World's Electronic Media Event IBC 2002*, pages 467–474, September 2002.
17. S-J. Yang and H-C. Chou. Adaptive QoS parameters approach to modelling Internet performance. *International Journal of Network Management*, 13:69–82, 2003.