SWS for Financial Overdrawn Alerting

José Manuel López-Cobo, Silvestre Losada, Oscar Corcho, Richard Benjamins, Marcos Niño, and Jesús Contreras

Intelligent Software Components, S.A. (iSOCO)
C/ Pedro de Valdivia, 10. 28006 Madrid, Spain
{ozelin,slosada,ocorcho,rbenjamins,
marcosn,jcontreras}@isoco.com

Abstract. In this paper, we present a Notification Agent designed and implemented using Semantic Web Services. The Notification Agent manages alerts when critical financial situations arise discovering and selecting notification services. This agent applies open research results on the Semantic Web Services technologies including on-the-fly composition based on a finite state machine and automatic discovery of semantic services. Financial Domain ontologies, based on IFX financial standard, have been constructed and extended for building agent systems using OWL and OWL-S standard (as well as other approaches like DL or f-Logic). This agent is going to be offered through integrated Online Aggregation systems in commercial financial organizations.

Keywords: Semantic Web Services, Ontologies, Composition, Intelligent Agent.

1 Introduction

The objective of the distributed system described in this paper (the Customer Notification Agent) is to provide added value to customers of financial services. This added value consists in a fully customizable and configurable set of aggregations and estimation functionalities on account balance evolution, as well as SMS and email alerts (among others), which will allow customers to have more efficient information about his financial position.

This system reuses existing technology for aggregation available at our company (iSOCO GETsee ®), and migrates it to Semantic Web Services technology. The integrated use of Semantic Web technologies and Web Services allows us to describe and reason with pieces of code understandable for machines, discharging the sometimes tedious task of checking the online accounts to a software system. This system is able to engage with other commercial solutions for aggregation and to detect at run-time and raise alerts if some conditions are detected (for example, a possible overdrawn of a customer saving account, due to the future payment of an invoice).

We have developed different ontologies to express the needed knowledge for this application. These ontologies are divided into three groups: general ontologies, which

S.A. McIlraith et al. (Eds.): ISWC 2004, LNCS 3298, pp. 782–796, 2004.

represent common sense knowledge reusable across domains; domain ontologies, which represent reusable knowledge in a specific domain; and application-dependent ontologies, which represent the application-dependent knowledge needed.

We have defined three high-level services for performing the task of the Customer Notification Agent. The *GETseeSWS Service* accesses the online accounts of the customer and the invoices associated with them, and calculates the balance for these accounts. The *NotificationService* notifies customers with different types of messages (discharging in 3rd party providers the execution of the actual notification) and finally, the *EstimationService* estimates, using different kinds of arithmetical functions, the expectable amount of an invoice for preventing an overdrawn situation.

One of the main innovations of our systems is the proposal of a finite state diagram to represent the composition of atomic processes into composite ones using conditions as a way to choose between different choices. Such an approach allows at runtime the discovery and invocation of services which comply with the conditions defined for the transition from one state to another. This allows describing a composite process at design-time by defining its behaviour and leaving the selection of the specific service to the execution time. This is an innovation with respect to other approaches where the selection of the specific services is done also during the design time.

The paper is organized as follows. Section 2 describes a sample scenario where the Notification Agent can be used, showing the main actors and agents involved in the overall process and the steps usually followed by them. Section 3 describes the ontologies that we have developed, either from scratch or by reusing other ontologies or vocabularies already available elsewhere. Section 4 describes the Semantic Web services created for the system, which have been implemented using OWL-S, DL and f-Logic. Section 5 describes one of the main contributions of this paper, namely the proposal for service composition using finite state diagrams. Finally, section 6 provides some conclusions of our work and future lines of research.

2 Scenario Description

Let us suppose that we are working on the scenario presented in figure 1. In this scenario we have a customer with several banking accounts where he/she has different amounts of money. This customer has also contracts with some consumer goods companies such as a telephone company, and gas and electricity providers, among others.

Everyday, the Customer Notification Agent will detect whether any of the customer accounts is going to be in an overdrawn situation. Bank accounts may have different invoices associated from different consumer good companies. If the amount of the invoice is bigger than the amount of money of the account, there could be an overdrawn situation. To help the customer, the system calculates an estimation of the

amount of every invoice expected for that account before its value date and notifies the customer if the balance of the saving account is less than the expected invoice amount. The system will choose any of the notification channels available for the customer and will notify him/her about the overdraw possibility.

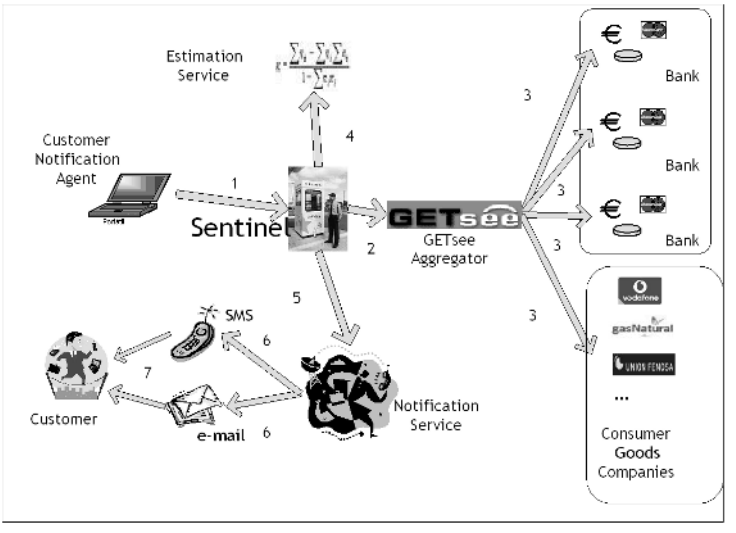

Fig. 1. Sample scenario diagram for the Notification Agent.

In this scenario, the following actors are involved: the customer, the banks, and the consumer goods companies. And the following agents are involved: customer notification agent (CNA), Sentinel and some estimation services. Finally, the iSOCO GETsee® application is at the core of this scenario, in charge of the aggregation of data from bank accounts and consumer goods companies.

The following steps will be normally done:

Step 1: Everyday, the Customer Notification Agent dynamically configures and invokes the Sentinel Service. This agent has the entire customer's information needed for invoking the composed service (online username, password and other data). The update frequency of this agent can be customized.

Step 2: The Sentinel Service uses iSOCO GETsee® for collecting information from the customer's accounts.

Step 3: iSOCO GETsee® collects the amount balance of all the customer's accounts (of banks B1, B2, …, Bn). In one (or more) of this accounts some consumer goods companies (E1, E2, …, En) can charge invoices. The invoices have their notification and value dates. The frequency of those invoices is always the same (weekly, monthly, bimonthly, annually).

Step 4: For each invoice of consumer goods companies (E1, E2, …, En) associated with the account, the Estimation Service estimates the probable amount at the end of the period, Ae (estimated amount) in terms of heuristics or mathematical models. Ae has a relationship with a consumer good company (Ee) and an account of a bank (ABe). If the Ae is less than the established threshold for the account, then an alert has to be raised.

Step 5: The Notification Service looks in a (de)centralized registry different ways to communicate with the user. It can find different services involving many different devices (phone calls using VoIP, SMS, electronic mail, telegram) and personal data (phone number, cell phone number, e-mail, postal address). The services discovered must have the ability to perform the action defined in the Notification Service.

Step 6: The invocation engine sorts in terms of cost, time to deliver, etc., the different possibilities and chooses the first service in this particular ranking. Some data mediation could be needed if terms of the ontology used differ from the one used by the Notification Service. If the service chosen has an irrecoverable mismatching of process or data, or some communication error occurs in the invocation, the service has to be able to choose another service and invoke it.

Step 7: The service chosen is invoked and the user is notified.

In summary, the objective of the Notification Agent is to provide added value to the user including a fully customizable and configurable set of aggregations and estimation functionalities on balance evolution as well as SMS and email alerts, allowing the customer to have more efficient information about his financial position in the incoming time period.

Several estimation functionalities allow calculating balance evolution on different accounts according to expected invoices and payments. The foreseen value of account balances will allow firing alert rules defined by the user and managed by the Notification Agent application. Those alerts could let him anticipate any trouble that could occur in his accounts or avoid missing any business opportunity.

3 Ontology Structure for the CNA

In this section we describe briefly the ontologies that model the domain presented in our scenario, and which will be used by the Semantic Web services developed for it and described in section 4. These ontologies have been implemented in OWL [3] using the Protégé-2000 ontology tool [5], with the OWL-plug-in [6]. A graphical outline of the main relationships between these ontologies is presented in figure 2.

According to the classifications of Van Heijst and colleagues [1] and of Mizoguchi and colleagues [2], we can distinguish the following types of ontologies:

- *General ontologies*, which represent common sense knowledge reusable across domains. In this group we can include our ontologies about users and notifica-

tions, which include basic concepts related to persons and their contact information, and our ontology about estimation parameters, which is based on statistical concepts.

- *Domain ontologies*, which represent reusable knowledge in a specific domain. In this group we can include our ontologies about financial products and financial services.

- *Application ontologies*, which represent the application-dependent knowledge needed. In this group we can include our ontologies about saving accounts and invoice payments, and our ontology about overdrawn situations. The reason why we classify them under this group does not mean that they might not be reusable in other ontology-based applications; instead it means that we have not designed them taking into account such objective.

We will first describe the general ontologies. The ontologies about users and notifications include basic concepts related to persons (users of information systems), such as name, surname, birth date, etc, and related to their contact information , such as email addresses, postal addresses, phone and fax numbers, etc. The same message can be sent in many different types of notifications, using the same or different physical devices. For instance, if you want to communicate with someone sending him a fax and an e- mail, the receiver will have two different communications, one in the facsimile device and the other in his e-mail inbox.

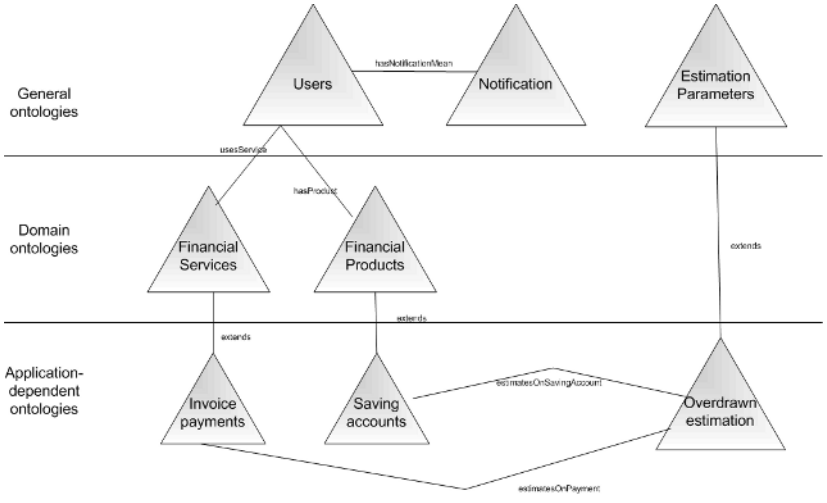

Fig. 2. Ontologies developed for the Customer Notification Agent for Financial Overdrawn.

With regard to the ontology about estimation parameter, it describes the basic arithmetical functions that can be used, among others, to estimate the amount of the spending of an invoice (or whatever other numerical concept which has an historical

evolution). This ontology considers parameters related to linear estimation factors, statistical information, heuristics, etc.

Regarding the domain ontologies defined, we have two different ones, as shown in figure 2: financial services and financial products. These ontologies are based on the IFX financial standard [12], so that they will be easier to reuse by other ontology-based applications.

The ontology about financial products contains the different types of products provided by a bank (loans, investment accounts, saving accounts, investment funds, etc.). In all of them the bank and the customer sign a contract where the bank stores or lend money from or to the customer. Each financial product has their own specific attributes, and is related to the corresponding user(s) of the ontology about users. Each product can be owned by many holders and vice versa.

The ontology about financial services represents those services that banks can provide to their customers and which are not financial products. These financial services provide added value to the relationship between a bank and their customers. They include loyalty cards, paying invoices by direct debit, Internet connection advantages, information provision about stock markets, etc.

The application-dependent ontologies describe more specific concepts and relationships related to our system. One of these ontologies is the one related to invoice payment, which represents the service that the bank offers to their customers, allowing to charge directly to a saving account of the customer the payment of many different things (taxes, shopping, subscriptions, consumer goods companies consumes like gas, water or phone). The ontology related to saving accounts includes concepts related to the types of saving accounts that can be contracted with the banks with which we are working.

Finally, the last application-dependent ontology extends the general ontology about estimation parameters, focusing on the specific case of overdrawn situations like the ones presented in the previous section.

4 Discovery of Notification Semantic Web Services

The following top-level services are available, as shown in figure 3: GETsee Service, Notification Service and Estimation Service.

Besides, the figure shows how the GETsee Service is decomposed into five atomic services (openSession, getAccounts, getInvoices, getBalance, closeSession). These five services are annotated using the same ontology as the GETsee service (although this is not mandatory in our approach). Those atomic services invoke other services, which are annotated according to other ontologies. In these cases, data mediation is needed for the exchange of messages, although this is out of the scope of this paper.

At last, the Notification Service looks for a service able to notify something to a person and finds at least two services (notification by SMS and notification by e-mail), which might be annotated according to other two more ontologies.

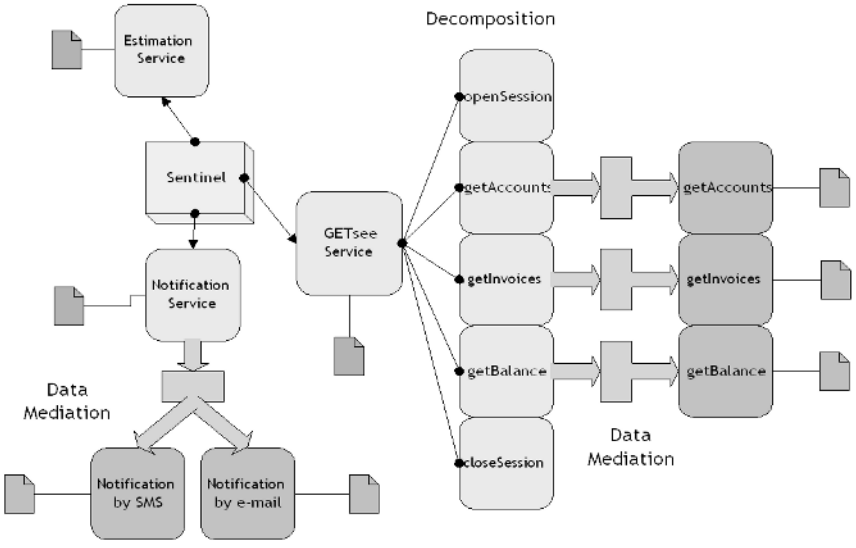

Fig. 3. A diagram of the Semantic Web services used for our notification scenario.

As commented in the previous section, the Semantic Web services used in our scenario have been annotated with OWL-S, DL and f-Logic [4, 10]. OWL-S uses the class Service as a complete description of the content and behaviour of a service. It has three differentiated parts. First of all, the Service Profile explains "what the service does". The Process Model describes "how the service works" and finally the Service Grounding maps the content and format of the messages needed to interact with the service. It provides a mapping from the semantic form of the messages exchanged as defined in the Process Model, to the syntactic form as defined in the WSDL input and output specifications.

For a further understanding about how is supposed to work the Discovery of Notification Services [17], we put the description (using DL) of two services (defined by their capabilities) and a Request from a User and depict how they will be matched.

Some domain-level facts:

$$\texttt{Notification} \sqsubseteq \texttt{Action}$$
$$\texttt{EmailNotification} \sqsubseteq \texttt{Notification}$$
$$\top \sqsubseteq =1\ \texttt{from}$$

Capability and a Request:

$$\texttt{Cap}_A \equiv$$
$$\texttt{EmailNotification} \sqcap \exists\ \texttt{from.User} \sqcap \exists\ \texttt{to.User} \sqcap \forall \texttt{to.User} \sqcap$$
$$\exists \texttt{usedProvider.}\{\texttt{Provider}_A\} \sqcap \exists \texttt{sendingTime.Timestamp} \sqcap \exists \texttt{content.String} \sqcap$$
$$\forall \texttt{acknowledgement.}=_F \sqcap \exists \texttt{cost.}\ =_5$$

$$\begin{aligned}&\mathtt{Req} \equiv \\ &\mathtt{ElectronicNotification} \sqcap \exists\ \mathtt{from.\{User_X\}} \sqcap \exists\ \mathtt{to.\{User_Y\}} \sqcap \exists\ \mathtt{to.\{User_Z\}} \sqcap \\ &=\mathtt{2to} \sqcap \forall\ \mathtt{usedProvider.Provider} \sqcap \exists\ \mathtt{sendTime} \leq_{\mathtt{200406250900}} \sqcap \exists\ \mathtt{con-} \\ &\mathtt{tent.String} \sqcap \forall\ \mathtt{acknowledgment} =_{\mathtt{T}} \sqcap \forall\ \mathtt{cost} \leq_{\mathtt{5}}\end{aligned}$$

With respect to the ontology schema introduced above the DL-based discovery component will match requests and capabilities using DL inferences. The basic idea of the DL-based discovery matching is to check whether the conjunction of a request and a capability is satisfiable, i.e. there can at least be one instance which they have in common. If $\mathtt{Request} \sqcap \mathtt{Capability_X} \sqsubseteq \bot$ holds true there is no such common instance and the request cannot be fulfilled by this capability.

Other useful approach would be use f-Logic and a reasoner for describe capabilities and goals [8] and make queries for matchmake capabilities and goals. For the goal we model the postcondition (the state of the information space that is desired). We express this by a fact in f-logic (here we use the flora2 syntax [16]).

```
myGoal:goal[
  postCondition->myNotification].
myNotification:notification[
  ntf_userToBeNotified -> johndoe,
  ntf_date-> d040606:date[dayOfMonth->5, monthOfYear->5,
                          year->2004],
  paymentMethod -> creditCard, cost -> 0.2,
  ntf_body -> "Your Account Z will be in minus in 2 weeks",
  ntf_from -> sentinel].
johndoe:user[
  nif ->123, name -> "John Doe", password -> "p", login -> "l",
  firstPreference -> jdMobile,
  contacts ->>
   {jdEmaill:eml_account[eml_account->"jon@doe.com"],
    jdMobile:phone[phn_number->"0123456", phn_type->mobile],
    jdHome:phone[phn_number->"6543210", phn_type->home]}].
sentinel:user[
  name -> "Sentinel System",
  contacts ->> {jdEmaill:eml_account[
     eml_account->"sentinel@isoco.com"]}].
```

The capability postcondition describes the state of the information space the service has after its execution. Here we use some prolog build in predicate e.g. '//' which is an integer division, but that might also be replaced by more declarative predicate names like "integerDivision(X,Y,Z)".

```
smsProvider[postcondition] :-
 _AnyNotification:notificationSMS[
    phn_number -> _X:phone[phn_type->mobile],
    ntf_receiptAcknowledgement -> false,
    ntf_time -> Time:dateAndTime,
    content -> AnyMessage:message,
    payment -> Payment],
 is_charlist(
  AnyMessage.msg_body, AnyMessageLength)@prolog(),
  AnyMessageLength < 800,
  Tokens is '//'(AnyMessageLength,160)@prolog()+1,
  Cost is Tokens * 0.05,
```

```
Payment.cost >= Cost,
(Payment.paymentMode=creditCard; Payment.paymentMode=account),
secondsBetween(currentDate,Time,X),   X < 5*60.
```

In the F-Logic approach for discovery we are checking if the capability entails the goal (capability ≤ goal). Current limitations with respect to available reasoners led to the current modelling, where we have the goal-postcondition as a fact (which may not be fully specified) and the capability-postcondition as a rule.

We would like to extend this approach on the one hand to overcome the limitations due to the modelling of the goal as fact (i.e. that makes it hard to express ranges and constraints) and on the other hand to extend it to other matching semantics (e.g. if the intersection is satisfiable like in the DL approach).

5 Composition Using Finite State Diagram

The functionality of the non-atomic processes could be decomposed in a structured (or not) set of atomic processes for performing the same task. This composition (or decomposition, viewed from the opposite side) can be specified by using control constructs such as *Sequence* and *If-then-else*. Such decomposition normally shows, among other things, how the various inputs of the process are accepted by particular subprocesses, and how its various outputs are returned by particular subprocesses.

A CompositeProcess must have a *composedOf* property by which is indicated the control structure of the composite, using a *ControlConstruct*

```
<rdf:Property rdf:ID="composedOf">
  <rdfs:domain rdf:resource="#CompositeProcess"/>
  <rdfs:range rdf:resource="#ControlConstruct"/>
</rdf:Property>
<owl:Class rdf:ID="ControlConstruct"/>
```

Each control construct, in turn, is associated with an additional property called *components* to indicate the ordering and conditional execution of the sub processes (or control constructs) from which it is composed. For instance, the control construct, *Sequence*, has a *components* property that ranges over a *ProcessComponentList* (a list whose items are restricted to be *ProcessComponents*, which are either processes or control constructs).

This property allows managing the control flow of the execution of a Composite-Process but, in counterpart, binds the ontologies to contain information about the data and control flow, what is not always desirable [13,14].

For that reason, in our system we have developed a mechanism to describe finite state machines (finite state diagrams). The situation calculus introduces first-order terms called *situations*, [15]. The intuition behind the situation calculus is that the world persists in one state until an *action* is performed that changes it to a new state. Time is discrete, one action occurs at a time, time durations do not matter, and actions are irreducible entities. Actions are conceptualized as objects in the universe of discourse, as are states of the world. Hence, states and actions are reified. All changes to

the world are the result of *actions,* which correspond to our atomic processes. The situation that holds on entry to an action is different to that which holds on exit. The exit situation is said to be the *successor* of the entry situation. Sequences of actions combine to form histories that describe composite situations – in essence the state that holds at the end of the sequence. Given this interpretation we can clarify the meaning of preconditions. A precondition is a condition that must be true of the situation on entry to an atomic process. However, sometimes these preconditions cannot be computed in terms of the input that is in terms of the domain ontology. That kind of preconditions are also called *assumptions*.

So, speaking in terms of Semantic Web Services, each *state* can be seen as a situation, stable, after or before any *action*. The set of preconditions that must be true in this state are part of the preconditions of the atomic processes that make change that state. Following in this interpretation, transitions in the state diagram represent each atomic process needed for fulfil part of the goal, as is presented in figure 4.

At this very moment there are several efforts to describe preconditions, postconditions, effects and assumptions in the research area, but few consensus has been reached to determine a final good candidate (SWRL, F-Logic, OWL, DRS). In order to describe our current needs we define a naïve solution to model conditions. Of course, making use of the reuse, we can import references to other conditions expressed in other ontologies.

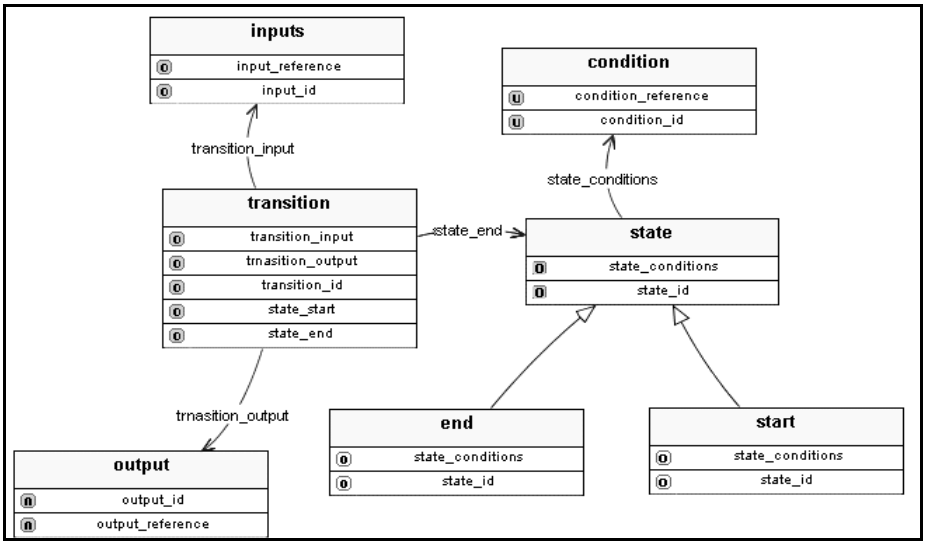

Fig. 4. Finite state diagram ontology.

The class *Condition* represents conditions, that is, statements which can be true or false in a particular state of the world (depicted in the state diagram). These conditions could be expressed in the same way (in fact, they are exactly the same) that we use to describe conditions in Semantic Web Services. Conditions of a State are modelled as instances of this class (or subclasses defined by an Ontology designer). This

class is defined as subclass of […] Process.owl#Condition and […] Process.owl#Effect defined in the OWL-S Process Ontology for model conditions and effects. Using this technique, expressing conditions in the state diagram in the same way as in the Services will favor any attempt to matchmake Services with Transitions.

```
<owl:Class rdf:ID="condition">
  <rdfs:subClassOf rdf:resource="Process.owl#Condition"/>
  <rdfs:subClassOf rdf:resource="Process.owl#Effect"/>
</owl:Class>
```

5.1 State

The class State models a state inside a state diagram. A state is represented as a node of the Graph which models a state machine. Each node is labelled with conditions, using the relationship *state_conditions*. Besides, each node is identified with a unique id, the slot *state_id*. A state in a Service Composition represents an intermediary step in the execution of two services.

The states (when we are talking of a concrete State Diagram) are represented as instances:

```
<state rdf:ID="estimated">
   <state_conditions rdf:resource="#logged_in"/>
</state>
```

5.2 Input and Output

The classes Input and Output defines the desired input and output of each transition. The specific inputs and outputs are modelled as subclasses of these classes. This is because the messages exchanged by the services (or viewed from the State Diagram point of view, the inputs needed for performing an action and the outputs derived from this actions) are, at last, classes or parts of some domain ontology. For a successful matchmaking it could be desirable Data Mediation for helping the Discovery Service to find services with similar inputs and outputs. The specific subclasses of Input and Output can be described in the same Ontology or they could inherit from other Ontologies (multiple inheritance) allowing to express the input and output of a Transaction in terms of the inputs and outputs of Services.

5.3 Transition

The class Transition models *actions* in a State Diagram. These actions are responsible for building a conversation in terms of the domain knowledge. From a stable situation (a state) and in presence of some conditions (which are true), some action is performed and some transition from the previous state to his successor is made. In a state diagram this transition is represented using an arrow from the starting state to the ending state. In a Composite Service framework, a Transition models the execution of an operation (in terms of Semantic Web Services this could be done by an Atomic Process or by another Composite Process).

The class Transition has the following attributes and relationships:

- State_start, State_end: They are the starting and ending state of the transition. They are instances of the class State. Each state is labeled with conditions which serve to refer to the preconditions and effect of the transition.
- Transition_input, Transition_output: Defines, in the domain ontology, the desired input and output for the transition. They references to subclasses of the class Input or Output (described before) or they could be a simple data type. This restriction makes mandatory the description of this ontology in OWL-full because OWL-DL doesn't allow this kind of description.

There are two special states labelled in a special way to denote what is the starting state and the ending state. Doing this, we always know what the first subgoal which can be achieved is and what is the final subgoal. With this information, some reasoning could be done forward or backward. To be able to transit from one state to another, The Discovery Service has to be able to find some Semantic Web Service with the same set of preconditions, effects, inputs and outputs which has the instance of Transition representing the transition between the states in the following terms:

- Preconditions: These conditions label the starting state.
- Effects: They are the conditions present on the ending state but missing in the starting state.
- Inputs: Define which part of the domain ontology need the service to be executed. Some data mediation could be needed if there are 3rd party services using other ontology.
- Outputs: Define which part of the domain ontology is the result of the execution of the service. Some data mediation could be needed if there are 3rd party services using other ontology.

For obtaining a more precise understanding of the relationship between the State Diagram and the Services (for the sake of matchmaking), see the figure 5.

This is the state diagram which models the functionality of Sentinel. It could be easily translated to the State Diagram Ontology, previously described. With this ontology and the description of the Service, an agent could accomplish the task described with the state machine. The agent will need to make some decision about what transition to take (i.e. what service has to execute) and some reasoner (with storage functionalities) will be needed to perform the control flow. Two instances of transitions can be seen below.

```
<transition rdf:ID="KB_044630_Individual_84">
   <state_end rdf:resource="#Logged"/>
   <state_start>
      <stateStart rdf:ID="initState"/>
   </state_start>
   <transition_id>GETseeSWSlogin</transition_id>
   <transition_input rdf:resource="#input_user"/>
   <transition_output rdf:resource="XMLSchema#boolean"/>
</transition>
```

```
<transition rdf:ID="KB_044630_Individual_92">
   <transition_input rdf:resource="#input_output_savingAccounts"/>
   <transition_input rdf:resource="#input_user"/>
   <transition_id>GETseeSWSgetinvoices</transition_id>
   <transition_output rdf:resource="#output_InvoicesPayments"/>
   <state_start rdf:resource="#AccountsLoaded"/>
   <state_end rdf:resource="#InvoicesLoaded"/>
</transition>
```

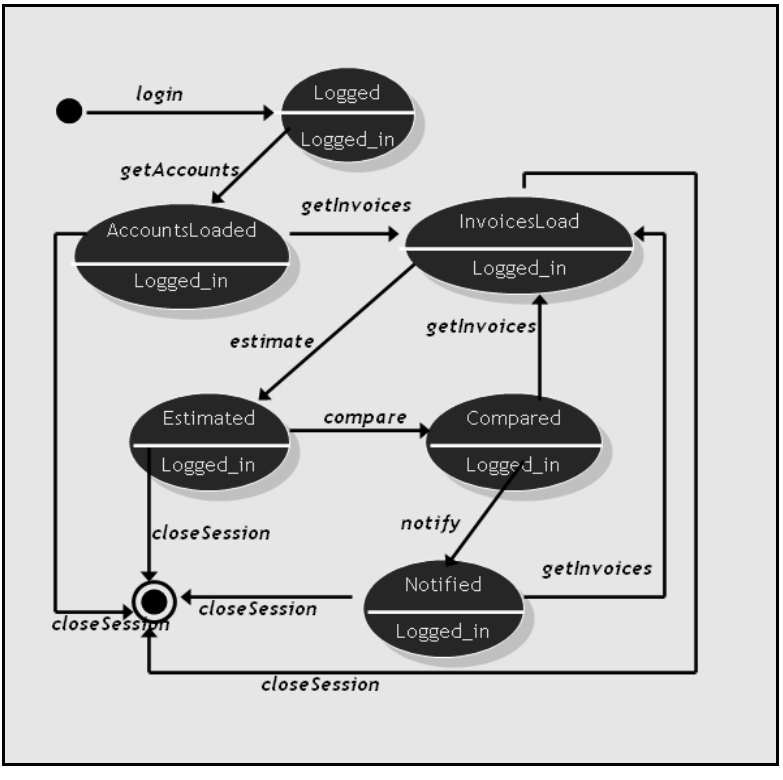

Fig. 5. Relationships between the state diagram and the Sentinel service.

6 Conclusions

We have described the Customer Notification Agent which makes use of an aggregation system, developed by iSOCO, called GETsee. ISOCO GETsee® application is able to aggregate information coming from different sources. It can be financial information (saving accounts, credit cards, investment funds, etc.), different invoices from consumer goods companies, loyalty cards, insurance companies or e-mail accounts from different Web Portals.

The Customer Notification Agent focuses on the dynamic configuration of a system that generates notifications and suggestions for transactions to the customer related to conciliation between financial accounts and pending invoices.

Integration of applications is one of the most ambitious goals of the Semantic Web Services. The existence of different agents or legacy applications must not interfere in the shared use of information. Exploiting the advantages of semantic interoperability and loose-coupled services will allow us to interconnect different applications and integrate data and information through messages. So, the system to be built leans upon an existing iSOCO's commercial application and others agents or services built *ad hoc*.

For adding semantics to the system, we have defined, or reused, different ontologies to express the needed knowledge. Besides, we have defined three services for perform the task of the Customer Notification Agent. The GETseeSWS Service access the online accounts of the customer, the invoices associated with them and calculates the balance for these accounts. The Notification Service notifies the user any message and finally, the Estimation Service estimates, using some arithmetical functions, the expectable amount of an invoice for preventing an overdrawn situation.

A Finite state diagram has been used for representing the composition of Atomic Processes, allowing in run-time the discovering and invocation of services which comply with the conditions defined for transition from a state to another. This allows us to describe a Composite Process in design-time, defining its behaviour and leaving the selection of the particular service to the execution time.

Some open research issues have been explored in this work as the composition on-the-fly and the discovery of Services using different approaches. These approaches will contribute to the testing of the contents of WSMO [8] and the SWWS-CA [18]. The projects SWWS and DIP, supporting this work, are devoted to the contribution and dissemination of WSMO.

Acknowledgements. This work is supported by the IST project SWWS (IST-2001-37134) and the IST project DIP(FP6 – 507483). We would like to thank Pablo Gómez, Andrés Cirugeda and Ignacio González for their contributions.

References

1. van HeijstG, Schreiber ATh, Wielinga BJ (1997). *Using explicit ontologies in KBS development* .International Journal of Human-Computer Studies 45:183-192
2. Mizoguchi R, Vanwelkenhuysen J, Ikeda M (1995). *Task Ontology for reuse of problem solving knowledge*. In: Mars N (ed) Towards Very Large Knowledge Bases: Knowledge Building and Knowledge Sharing (KBKS'95). University of Twente, Schede, The Netherlands. IOS Press, Amsterdam, The Netherlands.

3. OWL. Web Ontology Language.
http://www.w3.org/TR/2004/REC-owl- features- 20040210/
4. OWL-S. OWL for Services. http://www.daml.org/services/owl-s/1.0/
5. Protégé 2000. Stanford Medical Informatics. http://protege.stanford.edu/
6. OWL Plugin: A Semantic Web Ontology Editor for Protégé.
http://protege.stanford.edu/plugins/owl/
7. ezOWL Plugin for Protégé 2000. http://iweb.etri.re.kr/ezowl/plugin.html
8. WSMO. Web Service Modeling Framework.
http://www.nextwebgeneration.org/projects/wsmo/
9. SWRL: A Semantic Web Rule Language Combining OWL and RuleML.
http://www.daml.org/2003/11/swrl/
10. Michael Kifer, Georg Lausen, James Wu , Logical Foundations of Object Oriented and Frame Based Languages. Journal of ACM 1995, vol. 42, p. 741-843
11. DRS: A Set of Conventions for Representing Logical Languages in RDF. Drew McDermott, January 2004. http://www.daml.org/services/owl-s/1.0/DRSguide.pdf
12. IFX. Interactive Financial eXchange. http://www.ifxforum.org
13. D. Berardi, F. De Rosa, L. De Santis, and M. Mecella. Finite state automata as conceptual model for e-services. In Proc. of the IDPT 2003 Conference, 2003. To appear.
14. S. Narayanan and S. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proc. of WWW 2002*.
15. H. Levesque, F. Pirri, R. Reiter. Foundations of the Situation Calculus. Linköping Electronic Articles in Computer and Information Science, Vol 3, nr 18.
http://www.ep.liu.se/ea/cis/1998/018. December 1998
16. *FLORA*-2: An Object-Oriented Knowledge Base Language. http://flora.sourceforge.net
17. S. Grimm, H. Lausen. Discussion document SWWS Service Description / Discovery. May 2004
18. C. Priest. SWWS-CA A Conceptual Architecture for Semantic Web Services. May 2004