Prototyping a requirements specification through an automatically generated concurrent logic program*

Patricio Letelier Pedro Sánchez Isidro Ramos

Department of Information Systems and Computation
Valencia University of Technology, 46020 Valencia (Spain)
{letelier, ppalma, iramos}@dsic.upv.es

Abstract. OASIS is a formal approach for the specification of object oriented conceptual models. In OASIS conceptual schemas of information systems are represented as societies of interacting concurrent objects. Animating such models in order to validate the specification of information systems is a topic of interest in requirements engineering. Thus a basic execution model for OASIS specifications has been developed. Concurrent Logic Programming is a suitable paradigm for distributed computation allowing a natural representation of concurrence. Using Concurrent Logic Programming, OASIS specifications are animated according to OASIS execution model. In this work, we show how OASIS concepts are directly mapped into concurrent logic programming concepts. To illustrate our ideas, an example of a bank account codified in KL1 is given and parts of the program that animates its corresponding OASIS specification are shown. This work has been developed in the context of a CASE tool supporting the OASIS approach. Our aim is to build a module for animation and validation of specifications. A preliminary version of this module is presented.

1 Introduction

Conceptual models, representing the functional requirements of information systems, are a key factor when linking the problem and solution domains. Building a conceptual model is a discovery process, not only for the analyst but also for the stakeholders. The most suitable strategy in this situation is to build the conceptual model in an iterative and incremental way, through analyst and stakeholder interaction. Conceptual modeling involve four activities: elicitation of requirements, modeling or specification, verification of quality and consistency, and eventually, validation.

Formal methods for conceptual modeling provide improvements in soundness and precision for specifications, simplifying their verification. However, when considering elicitation and requirements validation, prototyping techniques are more used. Hence, it is interesting to obtain a combination of both approaches.

* This research is supported by the "Comisión Interministerial de Ciencia y Tecnología" (CICYT) through the MENHIR proyect (grant no. TIC97-0593-C05-01).

This work uses OASIS [9, 13] (**O**pen and **A**ctive **S**pecification of **I**nformation **S**ystems) as a formal approach for object-oriented conceptual specification of information systems. This is a step forward in a growing research field where validation of formal specifications through animation is being explored [18]. In this sense, some other proposals close in nature to OASIS are [7] and [6]. The differences, though, between these works and ours are basically determined by features of the underlying formalisms and the offered expressiveness . According to the presented results, the state of the art is similar and is characterized by preliminary versions of animation environments.

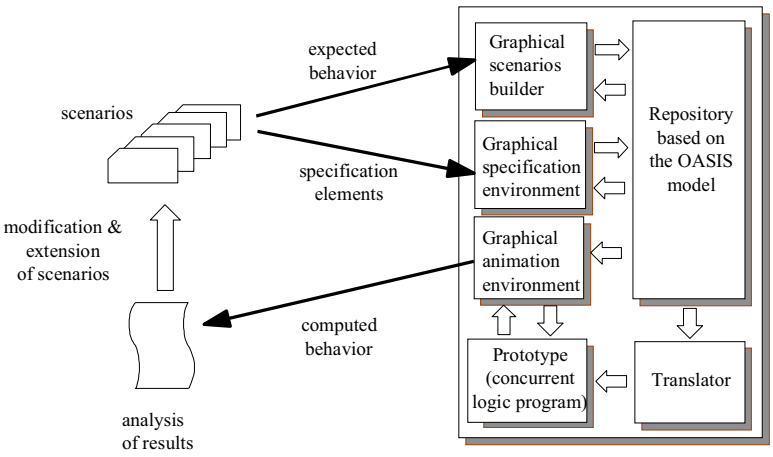

Fig. 1. A framework for incremental specification of requirements

Fig.1 shows our framework for elicitation, modeling, verification and validation of requirements. Elicitation is achieved using scenarios [15]. The expected behavior and elements of a given specification are extracted from scenarios. The graphical scenario builder helps defining scenarios in a suitable way. Functional requirements are modeled using a graphical specification module based on OASIS. Conceptual models can be verified according to OASIS formal properties. At each stage of the requirements specification process it would be possible to validate the behavior of the associated prototype against the expected behavior. This comparison could lead to updates or extensions of existing scenarios. This cycle continues until the requirements are compliant with the proposed set of scenarios.

Experiments have been carried out using Object Petri Nets [16] and Concurrent Logic Programming [8] as semantic domains for OASIS specifications. These efforts have resulted in a basic execution model [10]. This model is used to animate OASIS specifications implemented over concurrent programming environments.

In this context, we will present the mappings between OASIS concepts and clauses in concurrent logic programming. Those mappings have been implemented in a translator program. The translator takes an OASIS specification stored in the repository and generates automatically a concurrent logic program that constitutes a prototype for the corresponding conceptual model. Furthermore, through a preliminary version of the graphical animation environment the analyst can interact with the prototype in a suitable way. We have worked with the concurrent logic languages Parlog [3] and KL1[2] which have similar features. The implementation showed in this work is KL1 code.

2 OASIS

An OASIS specification is a presentation of a theory in the used formal system and is expressed as a structured set of class definitions. Classes can be simple or complex. A complex class is defined in terms of other classes (simple or complex). Complex classes are defined by establishing relationships among classes. These relationships can be aggregation or inheritance. A class has a name, one or more identification mechanisms for its instances (objects) and a type or template that is shared by every instance belonging to the class. We present next the basic concepts of OASIS.

Definition 1. ***Template or type****. A class template is represented by a tuple* $\langle Attributes, Events, Formulae, Processes \rangle$.

Attributes is the alphabet of attributes. For all $att \in Attributes$ exists the function:

$$att : sort\ of\ names \rightarrow sort\ of\ values$$

Events is the alphabet of events. For all $e \in Events$ is possible to get $\underline{e} = \theta e$ being θ a basic substitution of the parameters of the event. $Formulae$ is a set of formulae which are organized in sections and their underlying formalism depends on the section where they are used. *Processes* is the set of process specifications, classified in protocols and operations.

Definition 2. ***Service****. A service is either an event or an operation. The former is an instantaneous and atomic service. An operation is a non-atomic service and, in general, has duration.*

Definition 3. ***Action****. An action is a tuple* $\langle$*Client,Server,Service*$\rangle$*. It represents the action in the client object, associated to requiring the service, as well as the action in the server object, associated to providing the service.*

For each class we assume the implicit existence of A, a set of actions obtained from the services that objects in the class can request (clients) or provide (servers). For all $a \in A$ is possible to obtain $\underline{a} = \theta a$, being θ a basic substitution of client, server and service.

Definition 4. ***Object state***. *An object state is a set of evaluated attributes. It is expressed by well-formed formulae in First Order Logic.*

Definition 5. ***Step***. *A step is a set of actions occurring simultaneously in an object's life.*

Definition 6. ***Object life or trace***. *An object's life or trace is a finite prefix of object steps.*

2.1 OASIS expressed in Dynamic Logic

In [11] Deontic Logic [1] is described as a variant of Dynamic Logic [5]. The definition of deontic operators in Dynamic Logic is:

$\psi \rightarrow [a]false$ "the occurrence of a is forbidden in states where ψ is satisfied".
$\psi \rightarrow [\neg a]false$ "the occurrence of a is obligatory in states where ψ is satisfied".
$\psi \rightarrow [a]\phi$ "in states where ψ is satisfied, immediately after of the a occurrence, ϕ must be satisfied"

where ψ is a well-formed formulae that characterizes an object's state when the action a occurs and $\neg a$ represents the non-occurrence of the action a (i.e., only other actions different from a could occur). Furthermore, there is no state satisfying the atom $false$. This represents a state of system violation. Thus, one action is forbidden if its occurrence leads the system towards a violation state, and one action is obligatory if its non-occurrence leads the system towards a violation state. The OASIS *Formulae* and *Processes* are mapped to the formulae previously presented.

These formulae constitute a sublanguage of the language proposed and formalized in [20]. In [9] OASIS is presented as a specification language with a well defined syntax. Here is an example of part of a simple bank system using the OASIS syntax. This example will be used in the rest of the paper.

```
conceptual schema simple_banking_system
class account
identification
    number:(number);
constant attributes
    number:nat; name:string;
variable attributes
    balance:nat(0); times:nat(0); pin:nat(0); rank:nat(0);
derived attributes
    good_balance:bool;
derivations
    good_balance:={balance>=100};
events
    open new; close destroy;
```

```
    deposit(Amount:nat);
    withdraw(Pin:nat,Amount:nat);
    pay_commission;
    change_pin(Pin:nat,NewPin:nat);
    change_rank(Rank:nat);
valuations
    [deposit(Amount)] balance:=balance+Amount, times:=times+1;
    [withdraw(Pin,Amount)] balance:=balance-Amount, times:=times+1;
    [self:pay_commission] balance:=balance-1;
    [::pay_commission] times:=0;
    [change_pin(Pin,NewPin)] pin:=NewPin;
    [change_rank(Rank)] rank:=Rank;
preconditions
    withdraw(Pin,Amount) if (pin=Pin and balance>=Amount) or
                    (pin=Pin and balance<Amount and rank=2);
    change_pin(Pin,NewPin) if (pin=Pin);
    close if (balance=0);
triggers
    self::pay_commission when
                    (times>=5 and good_balance=false and rank=0);
end class

class customer
identification
    name:(name);
constant attributes
    name:string;
events
    add new; remove destroy;
end class

interface customer(someone) with account(someone)
    services(deposit,withdraw,change_pin);
end interface

interface account(someone) with self
    services(pay_commission);
end interface
end conceptual schema
```

In this example there are two classes: `customer` and `account`. The objects in both classes are active. An `account` object is forced to self-trigger an action with the event `pay_commission` whenever the trigger condition is satisfied. Although `customer` objects do not have explicit triggers, they have an interface with `account` objects enabling to require actions associated with the visible events. Thus `customer` objects are active objects as well. Furthermore, by default, there is always an object called `root`. This object will be responsible for activating the events that do not have an explicit client in the specification. In the example, the `root` object can require actions with the event `add` and `remove` to the `customer`.

2.2 An execution model for OASIS

Our execution model is an abstract animator for formulae of obligation, permission and change of state associated to an object. Next we briefly describe the concepts included in the execution model proposed in [10].

The sequence of steps in an object's life is sorted by time. We assume there is a "clock object" sending special actions — called *ticks* — to every object in the system. The received *ticks* by an object are correlative with natural numbers, t_1, t_2, etc. Hence, being i and j natural numbers then $i < j \iff t_i < t_j$.

Definition 7. ***Mailbox.*** *A mailbox is the set of actions that can be included in a step executed by one object at one tick. The mailbox at instant t_i is denoted by $Mbox_i$.*

Definition 8. ***Object's state at tick.*** *An object's state at tick is denoted by $State_i$ and represents the object's state in the interval $[t_i, t_{i+1})$. That is, between t_i (included) and t_{i+1} the state is considered constant.*

The processing of the actions inside the mailbox implies their classification. Next we present all possible actions that might be present in a mailbox. They are characterized as subsets of $Mbox_i$ (that is at instant t_i) .

- **Obligated actions**: these are actions associated to obligated service requests (which have as a client the object itself) and **have to** occur. The set of obligated actions is denoted by $OExec_i$. These actions are determined by obligation formulae, that is, their form is: $\phi[\neg a]false$.
- **Non-obligated actions:** these are actions corresponding to services requested by other objects (or itself) which **could be** provided or not depending on prohibitions established over those actions and verified in $State_i$. The set of non-obligated actions is denoted by $\overline{O}Exec_i$.
- **Rejected actions**: these are non-obligated actions whose occurrence is prohibited in $State_i$. The set of rejected actions is denoted by $Reject_i$. These actions are determined by prohibition formulae, that is, their form is: $\phi[a]false$.
- **Candidate actions:** these are non-obligated actions whose occurrence is permitted in $State_i$. The set of candidate actions is denoted by $Cand_i$.
- **Executed actions:** these are actions forming the step executed at t_i. The set of executed actions is denoted by $Exec_i$. A step is composed by $OExec_i$ joined with a subset of $Cand_i$.
- **Actions in conflict:** These are a subset of $Cand_i$ formed by actions in conflict[1] with some obligated or candidate action just selected. The set of actions in conflict is denoted by $Conf_i$.

A simple criterion is used in order to choose actions from $Cand_i$: when two actions are in conflict, the action which first arrived to the mailbox will be chosen. The actions in conflict $Conf_i$ are "copied" to the next mailbox ($Mbox_{i+1}$).

[1] Two actions are in conflict if they could change the value of non-disjoint set of attributes.

The object behavior is characterized by an algorithm (detailed in [10]) that manipulates each mailbox (at each *tick*) obtaining the subsets previously defined and producing the change of the object state.

3 Concurrent logic programming and OASIS

Concurrent logic languages arise as an attempt to improve the efficiency of logic languages by exploiting the stream AND parallelism. Besides, they are high level programming languages and very convenient for parallel and distributed systems. A concurrent logic program is a set of Horn Clauses with Guards.

$$H \leftarrow G_1, ..., G_n : B_1, ..., B_m \qquad n, m \geqslant 0$$

A goal has the following form:

$$\leftarrow M_1, ..., M_k \qquad k > 0$$

All M_i are evaluated in parallel (AND parallelism), using the program clauses for their reduction. For each M_i the clauses that can reduce it are examined in parallel (OR parallelism), selecting only one and avoiding backtracking. The criterion of selection is that the M_i unifies with the head of the clause, and the conjunction of guards $G_1, ..., G_n$ will be satisfied (evaluated also using AND parallelism, if they exist). If more than one clause could be chosen, then a sequential search can be established in textual order from top to bottom.

The integration of Concurrent Logic Programming and the OO modeling has generated a great deal of research. We are interested in modeling objects as perpetual processes according to the OASIS execution model. Modeling objects as perpetual processes is an approach initiate by Shapiro and Takeuchi [17], in which an object is implemented as a tail-recursive process that passes the update state of the object as arguments in the recursive call. The identity of the object is the name of an input stream argument of the process. Works in this direction are principally based on making OO extensions to concurrent logic languages. Within this approach some proposals are: Polka [4], L2||O2 [14] and A'UM [19]. Although some implementation aspects are common, our motivation is to generate automatically a concurrent logic program corresponding to an OASIS conceptual model.

3.1 Objects and classes in concurrent logic programming

Now we will sketch the essential features that allow considering an OASIS specification as a KL1 program. Details of intermediate clauses and clause bodies will be omitted to facilitate their reading. In Concurrent Logic Programming a

society of objects can be seen (at run time) as a goal to solve, where each object is a subgoal. Each object is evaluated using AND parallelism (inter-object concurrence).

$$\leftarrow object_1(In_1, Out_1, State_1), ..., object_k(In_k, Out_k, State_k) \qquad k > 0$$

In Concurrent Logic Programming, the partial instantiation of logical variables enables to use them as communication channels. In_i is the input channel for receiving actions, it is a merger of the messages received from itself and from other objects. Also, In_i is used as the object *oid*, so anybody knowing the value of *oid* could instantiate partially this variable, that is, send an action to that object. Out_i is the output channel for sending actions. $State_i$ is the list of terms $att(Attribute, value)$ for each attribute of the object.

In general, creation (and destruction) of objects at run time is required. Subgoals that have the capacity of generating (in their reduction) a new instance of an object should exist. These subgoals correspond in a natural form to the notion of class. Thus, classes are implemented as other objects of the society. A class has an attribute called *Population*, which is a list of pairs (In_i, Key) for each class instance. In_i is the object *Oid*, Key is a list of constant attributes that allows referring to the object. The predicate name of a class goal will be the name of the corresponding class. For an object goal the predicate name will be the name of its class with "`o_`" at the beginning.

Example 1. The class `account` in our bank system (at run time) is represented as a goal in Concurrent Logic Programming. Classes are goals at the beginning of the execution. Thus the class `account` appears as the following subgoal:

```
:- ...,account(In,Out,[att(population,[])]),...
```

3.2 Object behavior

Each object attempts to reduce itself using the clauses that represent its specification. Each clause recognizes one action and is able to reduce the object to a set of subgoals. Below an object goal and clauses with which this object goal could be reduced are shown. Considering $p, q, s, t \geqslant 0$ and $r > 0$:

$$: -..., o\_class_i(In^*, Out^*, State^*), ...$$

$$o\_class_i([action_{i1}|In], Out, State) : -$$
$$G_{i1}, ..., G_{ip}|$$
$$B_{i1}, ..., B_{it},$$
$$o\_class_i(In, NOut, NState).$$

...
$o\_class_i([action_{ir}|In], Out, State) :-$
$G_{i1}, ..., G_{iq}|$
$B_{i1}, ..., B_{is},$
$o\_class_i(In, NOut, NState).$

In^*, Out^* and $State^*$ represent the logical variables In, Out and $State$ at a given moment in the reduction of the goal o_class_i.

Object creation and destruction Classes are implemented as subgoals inside the initial goal. We obtain the instances by means of reduction of class goals whenever the action occurs with the event of creation. The creation of an object implies that the following subgoals are generated during the class goal reduction:

...
$NOut = \{Out, ObjectOut\},$
$o\_class(Oid, ObjectOut, Attributes),$
...

Out is the output channel for the class goal. This channel is separated into two new logical variables. *NOut* will be used as the new output channel for the class goal. *ObjectOut* will be the output channel for the object created. *Oid* is another new logical variable that will be used as object *Oid* and input channel.

Example 2. The clause to which the class `account` reduces when the action of creation occurs (action with the `open`[2] event). In the body of that clause there is the creation of a new object (goal) `account`.

```
account([action(Client,s(account),e(open,Attributes))|RestActions],
            Out,State) :-
    Out={NOut,ObjectOut},
    o_account(Oid,ObjectOut,Attributes),
    get_attributes(Attributes,[att(number,Number)]),
    NPopulation=[object(Oid,Number)|Population],
    update_state(State,[chg(population,NPopulation)],NState),
    ...
    account(RestActions,NOut,NState).
```

`get_attributes` is a predicate that extracts some attribute values from a list. `update_state` modifies a list of attributes producing a new one. When the event `open(101,john,0,0,1234,0,false)` is received the goal `account` becomes:

```
account(action(Client,c(account),
    e(open,[101,john,0,0,1234,0,false]))|Rest],Out,State)
```

This goal is reduced in the following two subgoals, that is, a new object `account` has been created.

[2] An event of creation has implicitly the constants and variable attributes of the object as arguments.

```
...
o_account(Oid,ObjectOut,[att(number,101),att(name,john),
    att(balance,0),att(times,0),att(pin,1234),
    att(rank,0),att(good_balance,false)]),
account(Rest,NOut,),[att(population,[object(Oid,101)]]).
```

The destruction of an object is obtained by the reduction carried out by selecting a clause whose body does not contain the same object as subgoal. Hence, the execution of the object ends.

Change of state We say that an object is implemented as a perpetual process because among the subgoals in which an object is reduced the same object appears. This produces the effect of continuity in the object life. Whenever the object goal is thrown as subgoal in the reduction, some of its attributes may be modified. Thus a change of state due to the occurrence of the associate action is represented. The effect "to execute action" is obtained in the reduction when the new input channel is used as the original one without considering the last executed action. The formulae that define the change of state when the event is executed are subgoals that assign new values to the attributes of the object.

Example 3. The execution of the action associated with the event `deposit(10)` sent to the recently created object is represented as the reduction of the goal:

```
o_account([action(Client,s(account,id(number,[att(number,101)])),
    e(deposit,[10]))|RestIn],Out,[att(number,101),att(name,john),
    att(balance,0),att(times,0),att(pin,1234),att(rank,0),
    att(good_balance,false)])
```

This goal is reduced using the following clause:

```
o_account([action(Client,Server,e(deposit,[N]))|Rest],Out,State) :-
    get_attributes(State,[att(balance,Balance),att(times,Times)]),
    NBalance:= Balance+N,
    NTimes:= Times+1,
    LExp1:= NBalance,
    RExp1:= 100,
    test_condition([[c(ge,LExp1,RExp1)]],NGood_balance),
    update_state(State,[chg(balance,NBalance),chg(times,NTimes),
    chg(good_balance,NGood_balance)],NState)
    o_account(Rest,Out,NState).
```

`test_condition` is a predicate that evaluates a list of conjunctions representing a well-formed formulae in the specification. If all the conjunctions are satisfied the second argument of `test_condition` is instantiated to *true*, otherwise it is $false$. Thus, in this case, the goal is reduced to the following goal, in this way, the object has changed its state.

```
o_account(Rest,Out,[att(number,101),att(name,john),att(balance,10),
    att(times,1),att(pin,1234),att(rank,0),att(good_balance,false)])
```

Prohibitions Action preconditions are implemented through intermediate predicates in the body of the object clauses.

Example 4. Here is the clause that verifies preconditions when the event `withdraw(1234,10)` is attended.

```
check_one_o_account_condition(action(Client,This,e(withdraw,[P,N])),
            State,Result,Msg) :-
    utility:get_attributes(State,[att(balance,Balance),
            att(rank,Rank),att(pin,Pin)]),
    LExp1:= Balance, RExp1:= N,
    LExp2:= Balance, RExp2:= N,
    LExp3:= Rank, RExp3:= 2,
    LExp4:= P, RExp4:= Pin,
    test_condition([[c(eq,LExp4,RExp4),c(ge,LExp1,RExp1)],
        [c(eq,LExp4,RExp4),c(lt,LExp2,RExp2),c(eq,LExp3,RExp3)]],
        Result),
    Msg=not([[c(eq,LExp4,RExp4),c(ge,LExp1,RExp1)],
        [c(eq,LExp4,RExp4),c(lt,LExp2,RExp2),c(eq,LExp3,RExp3)]]).
```

This time, the predicate `test_condition` is used to instantiate to *true* or *false* the variable `Result`. When the precondition is not satisfied another action is sent to the client including the message `Msg`.

3.3 Inter-object Communication

The communication mechanism offered by Concurrent Logic Programming is to share variable among subgoals, interpreted as a communication channel among objects. In order to allow many objects to communicate with a determined object, the architecture showed in Fig.2 has been implemented.

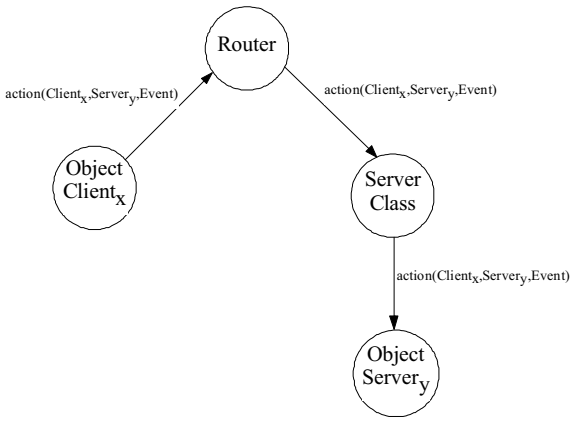

Fig. 2. Communication between objects

In general, if a client object $client_x$ has to send an action to a server object (or itself) $server_y$, $client_x$ instantiates partially its variable *Out*. The *Out* variable is merged with the rest of the output variables of objects and is used as input variable for an additional goal called *router*. The *router* goal has as many output variables as classes in the system. Whenever the goal *router* receives an action in its input, it will put the action in the corresponding output channel according to the server of the action.

The first goal in a KL1 program is the atom `main`. Thus, in the body of this predicate the global schema of communication is established.

Example 5. The main predicate defined for our bank system.

```
main:-
    take_active_actions(ExternActions),
    UserActions=[action(c(tester),s(user),e(add,[att(userid,root)]))
        |InUser],
    user(InUser,OutUser,[att(population,[])]),
    customer(InCustomer,OutCustomer,[att(population,[])]),
    account(InAccount,OutAccount,[att(population,[])]),
    generic:new(merge,{ExternActions,OutUser,OutCustomer,OutAccount},
        Actions),
    router(Actions,InUser,InCustomer,InAccount).
```

`take_active_actions` is a predicate that takes the external actions during the session of animation. There is always a class called `user` and an object of this class whose attribute `userid` has the value `root`.

Obligations In OASIS the object communication is activated by obligations. In this work we will focus on the fundamental communication mechanism, represented by triggers in the OASIS syntax. Triggers represent asynchronous communication. Hence, triggers are implemented executing $Out = [Action|NOut]$ without waiting for an answer. Triggers are generated due to a change of state. Thus, in the body of each clause that represents a change of state related with a condition in the trigger, a goal that evaluates the condition and could shoot the trigger will be added. This sentence will appear in the body of a selected clause in the reduction of a client object. *Out* is the identifier of the shared variable that constitutes the output channel of actions for the client object. This is a term containing the client reference. *NOut* is the new variable for the output channel of the client object.

Actions must first arrive to the router that sends the action to the corresponding input channel of class. If the action server is a class instance, the associated class searches in its *Population* attribute the input channel of the server object.

Example 6. A goal representing an object `account` reaching a new state:

```
o_account(In,Out,[att(number,101),att(name,john),att(balance,80),
    att(times,5),att(pin,1234),att(rank,0),att(good_balance,false)])
```

The possibility of trigger activation in the new state (that will be reached) was detected during the previous change of state. The clause doing this work is:

```
verify_o_account_triggers([t1|RestTriggers],State,Triggers) :-
    utility:get_attributes(State,[att(times,Times),
                att(good_balance,Good_balance),att(rank,Rank)]),
    LExp1:= Times,    RExp1:= 5,
    LExp2=Good_balance,    RExp2=false,
    LExp3:= Rank,    RExp3:= 0,
    utility:test_condition([[c(ge,LExp1,RExp1),
        c(eq,LExp2,RExp2),c(eq,LExp3,RExp3)]],Answer),
    utility:get_attributes(State,[att(number,Number)]),
    utility:put_trigger(Answer,action(this,self,e(pay_commission,[])),
        Triggers,NTriggers),
    verify_o_account_triggers(RestTriggers,State,NTriggers).
```

`put_trigger` is a predicate that puts the action in the trigger list if `Answer` is instantiated to *true*. In this case because `test_condition` returns *true* in `Answer`, we are sure that the object $Mbox$ contains the action:

```
action(this,self,e(pay_commission,[]))
```

This is an action that must be triggered because the client term is `this`. Thus it is put in the output channel using:

```
Out=[action(This,self,e(pay_commission,[]))|NOut]
```

Now `This` is the client specification referring to its identification.

4 A graphical animation environment

In the OASIS context, the system behavior is determined by the behavior of its objects. An object behavior can be observed by analyzing the actions occurred and the states reached by the object. In this sense, the animation of an OASIS specification allows examining actions and states of the objects. Following the guides mentioned before, a translator from OASIS to KL1 has been implemented. This translator takes as input a system specification from an OASIS repository and produces a KL1 program that is compiled in order to obtain the prototype. The translator and the prototype are programs running in a Unix workstation. The interface has been implemented in Tcl/Tk [12] using the Tcl plug-in for Netscape and establishing a socket connection to the Unix workstation.

Fig.3 shows the interface for a preliminary version for an environment of animation. The idea is to make easier the use of the prototype. The object society is drawn in the upper left corner, on the right the traces of actions of an object (or object group) are listed. The list of traces can be filtered according to the kind of actions defined ($OExec_i$, $\overline{O}Exec_i$, $Conf_i$, $Exec_i$ and $Rejected_i$). In the state area the state of the object is presented (only when one object is selected). Buttons *play*, *pause*, *stop*, *forward* and *review* are provided in order to control the session of animation. When the animation is paused it is possible to explore the traces of actions and states at previous instants. Eventually, the two entry widgets allow building an external action sent by the analyst in representation of one object in the system.

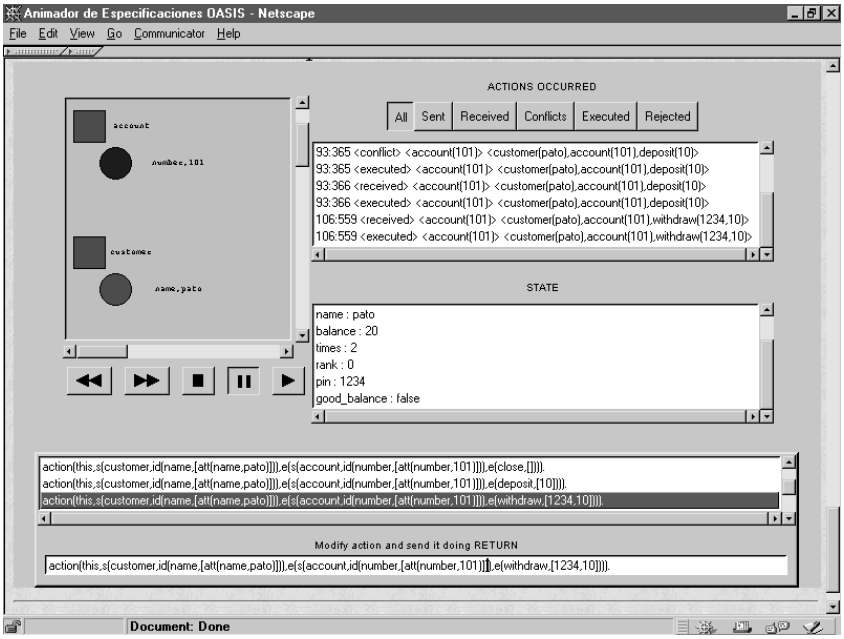

Fig. 3. An animation session

5 Conclusions

We have shown how the main features of OASIS can be naturally and directly represented in a concurrent logic program. Using the execution model of OASIS as a guide we can obtain a useful animation of the OASIS specification. Our animation is only applied to purposes of requirements validation and do not claim to be the final software product. The fidelity of the obtained concurrent logic program in relation to the OASIS system specification is a matter that is still being studied. In this case the verification and demonstration tasks would be supported by three important factors: firstly at the conceptual level the model is described in a formal language, secondly the abstract execution model is inspired by the semantics of that formal language and eventually there are some proposals dealing with formalization of concurrent programming languages. These factors do not determine the required justification but give a way of formalization on which we are working.

We have built a translator program to obtain a concurrent logic program automatically from an OASIS specification using the established correspondences. This work is being integrated into a CASE tool for system modeling supporting the OASIS model.

References

1. L. Åqvist. Deontic logic. In D.M. Gabbay and F.Guenthner, editors, Handbook of Philosophical Logic II, pages 605-714. Reidel, 1984.
2. T. Chikayama. KLIC User's Manual. Institute for New Generation Computer Technology, Tokyo JAPAN, 1995.
3. T. Conlon. Programming in PARLOG. Addisson-Wesley, 1989.
4. A. Davison. Polka: A Parlog object-oriented language, Ph.D. thesis, Department of Computer Science, Imperial College London, 1989.
5. D. Harel. Dynamic Logic. In Handbook of Philosophical Logic II, editors D.M.Gabbay, F.Guenthner; pages 497-694. Reidel 1984.
6. P. Heymans. The Albert II Specification Animator. Technical Report CREWS 97-13, Cooperative Requirements Engineering with Scenarios, http://sunsite.informatik.rwth-aachen.de/CREWS/reports97.htm.
7. R. Herzig and M. Gogolla. An animator for the object specification language TROLL light. In Proc. Colloq. on Object-Orientation in Databases and Software Engineering, Montreal 1994.
8. P. Letelier, P. Sánchez and I. Ramos. Animation of system specifications using concurrent logic programming. Symposium on Logical Approaches to Agent Modeling and Design, ESSLLI'97, Aix-en-Provence, France, 1997.
9. P. Letelier, I. Ramos, P. Sánchez and O. Pastor. OASIS 3.0: Un enfoque formal para el modelado conceptual orientado a objeto. SPUPV-98.4011, Servicio de Publicaciones Universidad Politécnica de Valencia, 1998.
10. P. Letelier, I. Ramos and P. Sánchez. Un modelo de ejecución para especificaciones OASIS 3.0 Infórme Técnico DSIC-II/36/98, Universidad Politécnica de Valencia, 1998.
11. J.-J.Ch. Meyer. A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic. In Notre Dame Journal of Formal Logic, vol.29, pages 109-136, 1988.
12. J. Ousterhout. Tcl and the Tk Toolkit. Addison-Wesley, 1994.
13. O. Pastor and I. Ramos. OASIS version 2 (2.2) : A Class-Definition language to model information systems using an object-oriented approach, SPUPV-95.788, Servicio de Publicaciones Universidad Politécnica de Valencia, 1995.
14. E. Pimentel. L2||O2: Un lenguaje lógico concurrente orientado a objetos, Tesis Doctoral, Facultad de Informática, Universidad de Málaga, 1993.
15. C. Rolland, C. Ben Achour, C. Cauvet, J. Ralyté, A. Sutcliffe, N.A.M. Maiden, M. Jarke, P. Haumer, K. Pohl, E. Dubois and P. Heymans. A Proposal for a Scenario Classification Framework, Technical Report CREWS 96-01, http://sunsite.informatik.rwth-aachen.de/CREWS/reports96.htm.
16. P. Sánchez, P. Letelier and I. Ramos. Constructs for Prototyping Information Systems using Object Petri Nets, Proc. of IEEE International Conference on System Man and Cybernetics, pages 4260-4265, Orlando, USA, 1997.
17. E. Shapiro and A. Takeuchi. Object oriented programming in concurrent prolog, en New Generation Computing, vol.1, pages 25-48, 1983.
18. J. Siddiqi, I.C. Morrey, C.R. Roast and M.B. Ozcan. Towards quality requirements via animated formal specifications. Annals of Software Engineering, n.3, 1997.
19. K. Yoshida and T. Chikayama. A'UM: A string based concurrent object-oriented language, In Proc. of the Int.Conf. on FGCS, ICOT, pages 638-649, 1988.
20. R.J. Wieringa and J.-J.Ch. Meyer. Actors, Actions and Initiative in Normative System Specification. Annals of Mathematics and Artificial Intelligence, 7:289-346, 1993.