Animating formal specifications with inheritance in a DL-based framework[1]

Pedro Sánchez Patricio Letelier Isidro Ramos

Department of Information Systems and Computation
Valencia University of Technology
Camino de Vera, s/n, 46071 Valencia
email {ppalma | letelier | iramos}@dsic.upv.es

http://www.dsic.upv.es/users/oom

Keywords: Requirements engineering, object oriented methods, inheritance, animation of specifications.

Abstract

Dynamic Logic (DL) provides a suitable formal framework to model actions and reasoning about them. OASIS is a language for the specification of object oriented conceptual models. In our model, specialization is a relation between classes that defines an inheritance mechanism through static and dynamic partitions. A variant of DL (including the Deontic operators for permission, prohibition and obligation) is the formalism used in OASIS to deal with changes of state, triggers, preconditions, protocols and operations. The animation of conceptual models in order to validate the specification is an interesting topic. We have worked on translating OASIS specifications automatically to concurrent environments in order to obtain a prototype useful to validate specifications by animation. The aim of this paper is to show that it is feasible to translate static and dynamic partitions automatically into dynamic logic formulae. Thus, using the same developed schema of animation it is possible to execute OASIS specifications including inheritance.

1 Introduction

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

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

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

This work uses *OASIS* [10,7] (**O**pen and **A**ctive **S**pecification of **I**nformation **S**ystems), a formal approach for object oriented conceptual specification of information systems. This is a step forward in a growing research field where the validation of formal specifications through animation is being explored [16]. In this sense, some other proposals close in nature to *OASIS* and to the work we are carrying out using *OASIS* are TROLL [5], ALBERT [6] and OBLOG [13]. The differences 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 art is similar and is characterized by preliminary versions of animation environments.

Validation and verification play an important role in the quality of the final product. By means of validating formal specifications stakeholders can fasten the correspondence between formal specifications and user prospects. The verification process allows to verifying the correctness between implementations and specification. Errors can be propagated towards the design and implementation process if the validation and/or verification are not done exhaustively. Basically, validation and verification techniques can be classified in two groups:

- Statics: reasoning about system properties regarding a set of predefined rules.
- Dynamics: executing some kind of implementation of the system.

As Feenstra points out in [1] two approaches are considered when animating specifications: to reason about scenarios (what requires an automatically generated prototype obtained from formal specifications) and to reason about reachability properties (what requires software capable of solving queries automatically). In our work the animation process focuses on the first approach showing the effects of actions in the system by testing scenarios.

Figure 1 shows a framework based on *OASIS* for elicitation, modeling, verification and validation of requirements. Elicitation is achieved by using scenarios [14]. The elements and expected behavior of a given specification are extracted by analysts from scenarios. 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-Oriented Petri Nets [15] and Concurrent Logic Programming [8] as concurrent environments for *OASIS* specifications. Correspondences between *OASIS* and these environments have been included in a translator program. This translator takes an *OASIS* specification stored in the repository and generates automatically a Concurrent Logic Program or a Petri Net that constitute 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 obtain CodeSign code automatically

from *OASIS* specifications. This work is being integrated into a CASE tool for system modeling supporting the *OASIS* approach. We have addressed validation through animation focusing on *OASIS* but this work could be extended to other similar languages. Basically, the obtained prototypes allow to animate sets of a variant of Dynamic Logic formulae [12] (including permissions, valuations and obligations) representing *OASIS* specifications. Our aim is to be able to animate (using the same translator) *OASIS* specifications including inheritance.

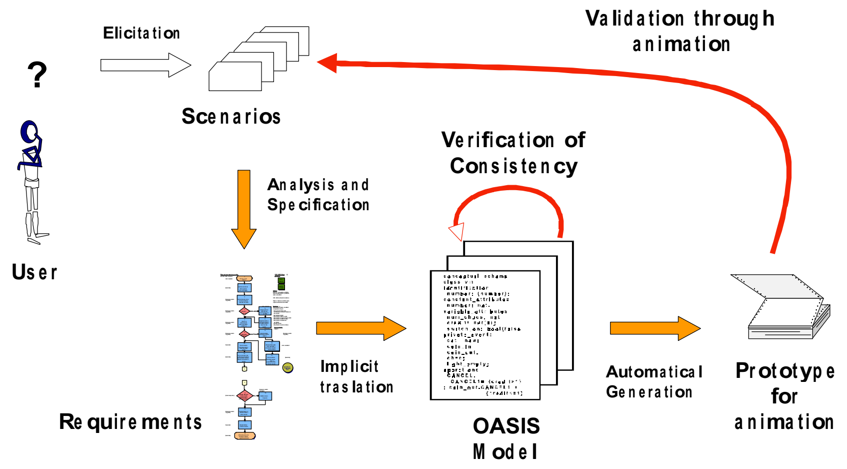

Figure 1: Animation Environment.

Nowadays, an increasing work in validation through animation of conceptual models has been developed. Many CASE tools have integrated an special module for animation. Most of tools related to requirements validation do not offer much help for animation. OBLOG[2] has an animation module but it is not still available. In Rhapsody[3] and ObjectTime[4] a graphical representation (a sequence diagram) is included as a result of the animation and interaction between objects. In this way, Aonix[5] offers a module called "Animator" in which the edition and execution of scenarios are supported. In BridgePoint[6] validation is done by monitoring in several levels the generated prototype but it is not easy to contrast between the obtained and expected results. In [3] an environment about TROLL in which is possible to study the change in the state of objects is showed. In a similar way, in [6] the architecture and functionality of an animation module for ALBERT specifications is presented. In all these proposals near to *OASIS*, the state of the art is clearly characterized by preliminary versions of animation modules. The aim of this paper is to show that is feasible to translate automatically the *OASIS* specialization in *DL* formulae. Thus using the same developed schema of animation it is possible to execute *OASIS* specifications including inheritance.

[2]http://www.oblog.pt

[3] http://www.ilogix.com

[4] http://www.objectime.com
[5] http://www.aonix.com
[6] http://www.projtech.com

The rest of this paper is organized as follows: in section 2 we introduce the basic concepts of *OASIS*. Section 3 gives the formal framework of specialization in *OASIS*. Section 4 gives a short description of the used *DL*. Section 5 establishes a representation mechanism for specialization. Section 5 summarizes the paper and outlines further work. Finally, an appendix introduces a full case study.

2 Basic concepts of *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) by establishing relationships among classes. These relationships provide **aggregation** or **inheritance** mechanisms. 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. Each object has an unique identifier (*oid*) set by the system, however, objects are referred by their identification mechanisms belonging to the problem space. A function establishes a mapping between the identification mechanisms and the *oid*. The type or template describes the structure and behavior of every object.

Thus each object encapsulates its own state and behavior rules. As usual in object oriented environments, objects can be seen from two points of view: static and dynamic. From the static perspective, the attributes are properties describing the object structure. The object state in a given instant is the set of structural properties values. From the dynamic perspective, the evolution of objects is characterized by the “change of state” notion. The occurrence of actions implies changes (by means of **valuations** and **derivations**) in the values of the attributes. Object activity is determined by a set of rules: **preconditions** (as forbidden actions in certain states), **triggers** (as obligations to be fulfilled in certain states), **protocols** (as allowed sequences of actions in object life) and **operations** (as obligated sequences of actions). A **step** is the set of actions executed at the same instant by the object.

Inheritance is a mandatory characteristic of the object oriented paradigm. Inheritance is a mechanism through which subclasses inherit properties of superclasses. The inheritance mechanism is an unquestionable help in incremental construction and reuse, but researches rarely agree on its meaning and usage [17]. In *OASIS*, inheritance is used in a disciplined way under the concept of **specialization**.

3 Specialization in *OASIS*

In *OASIS* to specialize a class means to create one or more partitions for it. Each partition is a set of new classes that divides the original one taking into account some criterions, thus many partitions can coexist. One object, in a given instant, is instance of only one subclass in every partition. Next we give the characteristics of each kind of specialization in *OASIS*, it is, **static partitions** and **dynamic partitions**.

Definition 1 ***Subclasses and superclasses.*** *When a subclass is a specilization of another class, the former is said to be a subclass and the latter a superclass. The template of a subclass is derived from the templates of other classes (superclasses).*

3.1 Static partitions

A static partition [18] divides completely the possible instances set of the partitioned class into disjoints subsets. Thus, static partitions divide the space of objects. Each object is permanently created as an instance of a given subclass in static partitions.

Each instance of a given subclass is at the same time one instance of its superclass and vice versa.

Example 1 Two static partitions of class `vehicle`:

```
truck, car, other
       static specialization of vehicle;
gas, diesel, other
       static specialization of vehicle;
```

3.2 Dynamic partitions

A dynamic partition [18] includes subclasses to which an object can belong during its lifetime. Object migration between classes is produced by the occurrence of actions. Thus, a dynamic partition divides the possible states of objects allowing to objects to change their subclasses.

From the point of view of the *OASIS* language, we can specify the transition between subclasses in a partition in two manners (being equivalent): by means of a migration process (related to occurring actions) or by attribute values.

Example 2 A dynamic specialization of class `car` produced by the occurrence of the actions `new_car`, and `repair_car` in *OASIS*:

```
working, broken_down dynamic
specialization of car
       migration relation is
  car = new_car.working;
  working = break_down.broken_down;
  broken_down = repair.working;
```

The creation of a `car` instance implies that it starts belonging to the class `working`. As an instance of `working`, actions from `car` and `working` templates can be recognized. The action `break_down` implies to leave the subclass `working` and to migrate towards `broken_down` class.

From a theoretical point of view, the process representing the life of an instance of `car` is the joint of processes defined in every subclass. The connections among subclasses are given by the actions included in the migratory process.

Example 3 A dynamic partition of the class `person` under the attribute approach:

```
child where {age < 13}
teenager where {13 < = age and age < = 19}
```

```
adult where {19 < age}
dynamic specialization of person;
```

In this case, whenever the attribute `age` changes, depending on the established conditions one instance of person could migrate between subclasses.

Eventually, in *OASIS* is not allowed to define static partitions from dynamic ones.

3.3 Species and multiple inheritance in *OASIS*

The Cartesian product between leaf classes of a given specialization hierarchy gives the set of **species** mixing all properties of those classes. Furthermore, emergent properties can be specified in their templates.

Example 4 Species of the hierarchy `vehicle`:

```
truck*diesel, car*gas, car*diesel
other*diesel, other*gas, truck*gas
```

Example 5 The class `truck*diesel` may have an emergent attribute named `tonnage`. This attribute must be specified in the `truck*diesel` template.

4 *OASIS* formalized in *DL*

Dynamic logic (DL) [4] has been traditionally considered to describe and reason about dynamic systems. The aim of this logic is the study of mathematical properties of programs and their behaviour. An object in *OASIS* is able to pass from one state to another. Due to the state change, the truth values of the formulas describing the state also change. The objective in our work is to define a logical basis to be able to express our reasoning about *OASIS* specifications. To each *OASIS* object we associate an accessibility relation in such a way that a pair of state (*s*,*t*) is in that relation if and only if there is a computation of the program (execution of a set of actions by the object) transforming the state *s* into the state *t*.

In [12] Deontic Logic is described as a variant of *DL* [4]. The definition of deontic operators in *DL* is:

$O(a) \Leftrightarrow [\neg a]$ false	"the occurrence of *a* is obligatory".
$F(a) \Leftrightarrow [a]$ false	"the occurrence of *a* is forbidden".
$P(a) \Leftrightarrow \neg F(a)$	"*a* is permitted if and only if *a*
	is not forbidden".

Where *false* denotes no reachable world. It is usual to find in related works the special atom V of violation representing a problematic or "non-ideal" situation. In our work the atom false expresses the impossibility of performing a transition.

We are using a sublanguage of the language presented by Meyer in [12] including the following kind of formulae:

$\psi \rightarrow [a]\mathit{false}$	“the occurrence of *a* is forbidden
	in states where ψ is satisfied”
$\psi \rightarrow [\neg \mathrm{a}]\mathit{false}$	“the occurrence of *a* is obligatory
	in states where ψ is satisfied”
$\psi \rightarrow [\mathrm{a}]\phi$	“immediately after *a* occurrence,
	ϕ must be satisfied
	in states where ψ is satisfied”

These formulae are named **prohibition**, **obligation** and **valuation**, respectively. Additionally, ψ is a well-formed formula that characterizes the state of an object 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*. 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.

An *OASIS* class template is represented by the tuple $\langle$*Atr*, *Ev*, *Formulae*, *Processes*$\rangle$, that is, attributes, events, *DL* formulae (valuations, prohibitions and obligations) and process specifications (protocols and operations). In [9] it is showed how Processes can also be interpreted as a set of *DL* formulae. Thus, the class template can be seen as $\langle$*Atr*, *Ev*, *DLForm*$\rangle$ where *DLForm* includes all *DL* formulae establishing the behavior of objects. Next section details how, through a translation process, static and dynamic partitions can be integrated in the same formal framework of *DL*.

5 Specialization in *DL*

As mentioned in previous sections species involve classes in the same hierarchy. Likewise, the classes constituting species have a common ancestor class which has been partitioned more than once. Figure shows a specialization hierarchy with the following species:

C_3*C_1	C_3*C_6	C_3*C_7
C_4*C_1	C_4*C_6	C_4*C_7
C_5*C_1	C_5*C_6	C_5*C_7

But, for instance, the following are not species: C_3*C_4,C_1*C_7, $C_3*C_5*C_1$, etc.

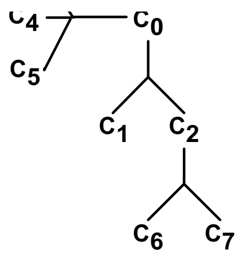

Figure 2: Specialization hierarchy.

We are interested in establishing the properties of species because in *OASIS* objects are always instances of species. In order to calculate these properties we will see some simplified case studies. Next we will analyze the equivalent in *DL* of any static partition.

5.1 Static partition hierarchies

Given a specialization hierarchy with only static partitions the set of properties describing the behavior of any leaf class in the hierarchy is the union of properties from root class to the leaf class. In static and dynamic partitions of *OASIS* we have behavioral compatibility between superclasses and subclasses. Thus, the set of properties of any leaf class C_i is given as follows:

i. Set of attributes:

$$Atr_i = \cup Atr_k \qquad \forall C_k = C_i \text{ hasta } C_k = C_{root}$$

ii. Set of events:

$$Ev_i = Ev_k \qquad \forall C_k = C_i \text{ hasta } C_k = C_{root}$$

iii. Valuation formulae in subclasses can only modify emergent attributes[7] (and also attributes without valuations in superclasses), thus the set of valuation formulae is the union of valuations from leaf class to root class (i.e. the *DL* formulae union).
iv. If a prohibition formula is modified in the subclass then the new condition must imply the superclass condition, thus the set of properties is the union of the properties from leaf class to root class (i.e. the *DL* formulae union). In this way, only are considered the most specific preconditions.
v. Obligation formulae are established in the same way as prohibition formulae.

After deriving both structural and behavioral properties from every leaf class, it is immediate to calculate the properties of species.

Definition 2 ***Template of a species*** *(C_0*...*C_n) is a class template with the following elements:*

$$Atr_0 \cup \ldots \cup Atr_n \cup Atr_e$$

[7] That is, attributes defined in the subclass specification.

$$Ev_0 \cup \ldots \cup Ev_n \cup Ev_e$$

$$DLForm_0 \cup \ldots \cup DLForm_n \cup DLForm_e$$

where Atr_i, Ev_i and $DLForm_i$ are attributes, events and *DL* formulae of class C_i; Atr_e, Ev_e and $DLForm_e$ are the added attributes, events and *DL* formulae being specified for the species ($C_0 * \ldots * C_n$). In case of overlapping of properties the calculus of properties follows the constraints previously mentioned.

5.2 Dynamic partition hierarchies

Whenever dynamic partitions are included, the process to determine properties of species is slightly different.

The idea is to translate all the subclass properties in every dynamic partition into an ascending direction. Figure 3 shows an example in which class C_2 has been dynamically partitioned. Thus we want to extend the dynamic formulae of class C_2 of Figure 3 separating the formulae due to C_6 from those due to C_7. Next specific considerations when deriving the *DL* formulae set are described here.

Migration by action occurrence

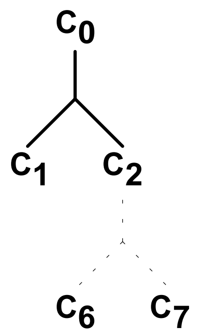

Figure 3: Dynamic partition.

Given the dynamic partition $\{C_1,\ldots,C_n\}$ of the class C_j, then the template of C_j including properties of subclasses for each class C_i belonging to the partition, is given as:

i. Set of attributes: $Atr = Atr_j \cup Atr_1 \cup \ldots \cup Atr_n$.
ii. Set of events: $Ev = Ev_j \cup Ev_1 \cup \ldots \cup Ev_n$.
iii. The set of valuations formulae is the union of valuation formulae of classes C_j and C_1 to C_n. Each valuation must have a condition that controls when it is used.
iv. The set of prohibition formulae is the union of prohibition formulae from the classes C_j and C_1 to C_n. Taking into account the behavior compatibility restriction, if C_i redefines a prohibition of C_j then the condition of C_i prevails.
v. Obligation formulae are established in the same way as prohibition formulae.
vi. Migration processes defined in partitions will be included in class C_j as a protocol containing valid action sequences. Prohibition formulae are directly incorporated into the set of prohibition formulae of C_j. However the behavior

property of the class C_i must add an occurrence predicate that evaluates the current state of the process. Thus, this such a protocol will be translated into prohibition formulae and valuation formulae in the same way as protocols.

vii. Repeat the procedure from i) to vi) for each dynamic partition of C_j.

Example 6 Given the following dynamic partition in *OASIS*:

```
C6, C7
dynamic specialization of C2
migration relation is
  C2new.C6
  C6a6.C7
  C7a7.C6
```

where we also have the following valuation formulae:

$\Phi_2 \rightarrow [a_2]\Psi_2$	in class C_2
$\Phi_6 \rightarrow [a_6]\Psi_6$	in class C_6
$\Phi_7 \rightarrow [a_7]\Psi_7$	in class C_7

then, when translating these properties into the class C_2 , the previous formulae are substituted by the following ones[8]:

$\Phi_2 \rightarrow [a_2]\Psi_2$
$((p = C_6) \wedge \Phi_6) \rightarrow [a_6]\Psi_6$
$((p = C_7) \wedge \Phi_7) \rightarrow [a_7]\Psi_7$

and the following formulae are included:

$(p = C_6) \rightarrow [a_6](p = C_7)$
$(p = C_7) \rightarrow [a_7](p = C_6)$
$\neg(p = C_6) \rightarrow [a_6]\text{false}$
$\neg(p = C_7) \rightarrow [a_7]\text{false}$

Migration by attribute value

In this case *DL* formulae are now defined as follows:

i. Set of attributes: $Atr = Atr_j \cup Atr_1 \cup ... \cup Atr_n$.
ii. Set of events: $Ev = Ev_j \cup Ev_1 \cup ... \cup Ev_n$. Taking into account that an emergent event of a subclass must not be available if the object is not instance of that subclass, then it is necessary to add a *DL* formulae that disable the occurrence of that event (see next example).
iii. The set of valuation formulae is the union of valuation formulae of classes C_j and C_1 to C_n.

8 Where p is a variable representing the current state of the migration process. Thus p is modified by valuation formulae.

iv. The set of prohibition formulae is the union of prohibition formulae of classes C_j and C_1 to C_n. Taking into account the behavior compatibility restriction, if C_i redefines a prohibition of C_j then the condition of C_i prevails.

v. Obligation formulae are established in the same way as prohibition formulae.

vi. Each formula obtained from C_i must add an occurrence predicate referring to the current state of the object.

vii. Repeat from i) to vi) for each dynamic partition of C_j.

Given the following dynamic partition in *OASIS*:

```
C6 where {atr1 < 0},
C7 where {atr1 >= 0}
dynamic specialization of C2
```

where we also have the following valuation formulae:

$\Phi_2 \rightarrow [a_2]\Psi_2$	in class C_2
$\Phi_6 \rightarrow [a_6]\Psi_6$	in class C_6
$\Phi_7 \rightarrow [a_7]\Psi_7$	in class C_7

then, when translating these properties into the class C_2 , the previous formulae are substituted by the following ones:

$\Phi_2 \rightarrow [a_2]\Psi_2$
$((atr_1 < 0) \wedge \Phi_6) \rightarrow [a_6]\Psi_6$
$((atr_1 >= 0) \wedge \Phi_7) \rightarrow [a_7]\Psi_7$

and the following formulae are included:

$\neg(atr_1 < 0) \rightarrow [\, a_6]false$
$\neg(atr_1 >= 0) \rightarrow [\, a_7]false$

5.3 Translation process summary

The translation process from class specification with specialization to *DL* formulae is summarized in the following steps:

- The translation process is applied for each specialization hierarchy.
- The translation process in dynamic partitions (if they exist) starts from the bottom level. All properties defined in dynamic subclasses are added to the superclass. The translation process continues until reaching a static partition or until reaching the top class in the hierarchy (in this case, the translation process finishes).
- When all dynamic partitions have been substituted the properties for each leaf class containing only static partitions are determined.
- Eventually, properties for species can be determined from leaf class properties. This hierarchy flattening in separate species is the final representation. Thus, all species will be independent classes at the same level and without specialization relationships.

5.4 Animation support

From an ideal point of view the representation in *DL* formulae finishes when all properties of species are specified. However, we also have to give a suitable support for taking into account the existing polymorphism in our *OASIS* model. Thus, an action in *OASIS* may reference a server class not being a species. In this situation the system must determine the full properties at the lower level. This implies a separation in two levels in our animation environment. On the one hand we have objects whose behavior is given by one species in which they were created. On the other hand we also have a class representing each original hierarchy class. In this additional class we keep a reference for each object belonging to it[9].

Figure 4 shows how every class is considered at the same level (without hierarchy) and objects are associated to species. However, there are mappings from each class to their object instances. For example obj_1 is an instance of class A and B and it is also instance of all species like B∗, etc.

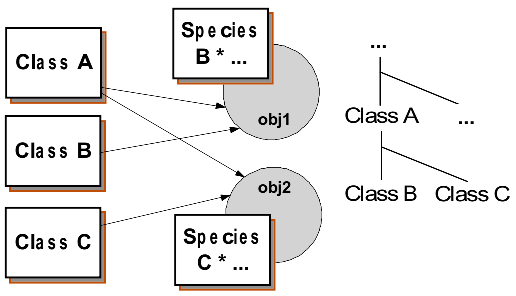

Figure 4: Representation example.

6 Conclusion

OASIS is a formal approach for the specification of object oriented conceptual models. In *OASIS* conceptual schemes 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.

Using inheritance we can specialize (or generalize) properties defined in classes. Specialization is an important modeling mechanism. There are two kinds of specializations in *OASIS*: by defining static and dynamic partitions. Objects in static partitions belong to a given specialized class during their whole lifetime. Besides, objects in dynamic partitions can migrate from one subclass to another one. The migration between subclasses of the same partition may be due to action occurrence or change in attribute values. A process migration represents the possible transitions among subclasses that an object can do. In addition, class templates of *OASIS* can be expressed as a set of *DL* formulae.

Static and dynamic specialization constructs in conceptual modeling allow expressing directly some patterns. For example, in [2] the dynamic classification is implemented

[9] A mapping between identifying mechanisms and *oids*.

using the pattern named *state* and the role mechanism may be implemented using the pattern named *role object*.

This paper gives the required steps to translate static and dynamic partitions in the same framework of *DL* used for class template. Thus, an *OASIS* specification with specialization can be interpreted automatically as an equivalent specification without specialization in our *DL*-based framework. In this way, the current animation environment, working with valuation, prohibition and obligation formulae, can be used to animate specifications with specialization.

We have built a translator program to obtain a prototype from *OASIS* specifications automatically. The translator should be extended in order to include the mappings established in this paper. This work is being integrated into a CASE tool for system modeling supporting the *OASIS* approach and the methodology OO-METHOD [11] defined for *OASIS*.

References

[1] Feenstra R.B. and Wieringa R.J. *Validating Specifications of Dynamic Systems using Automated Reasoning Techniques*. In J.C. Bioch and Y.-H. Tan (Eds.), Proceedings of the Seventh Dutch Conference on Artificial Intelligence, NAIC'95, pages 105-114, 1995.

[2] Gamma E., Helm R., Johnson R. and Vlissides J. *Design Patterns: elements of Reusable Object Oriented Software*. Professional Computing Series, Addison-Wesley, Reading, MA, 1994.

[3] Grau A. and Kowsari M. A. *Validation System for Object-Oriented Specifications of Information Systems*. In Manthey R. and Wolfengagen V. (Eds.), Proceedings of the First East-European Symposium on Advances in Databases and Information Systems (ADBIS'97), St. Petersburg, Electronic Workshops in Computing, Springer, September 2-5, 1997.

[4] Harel D. *Dynamic Logic*. In Handbook of Philosophical Logic II, editors D.M. Gabbay, F. Guenthner, pages 497-694, Reidel 1984.

[5] Grau A., Kuester Filipe J., Kowsari M., Eckstein S., Pinger R. and Ehrich H.-D. *The TROLL Approach to Conceptual Modelling: Syntax, Semantics and Tools*. In Ling T.W., Ram S. and Lee M.L. (Eds.) Proc. of the 17th Int. Conference on Conceptual Modeling (ER'98), Singapore, pages 277-290, Springer, LNCS 1507, November 16-19, 1998.

[6] Heymans P. *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] Letelier P., Ramos I., Sánchez P. and Pastor Ó. *OASIS 3.0: A Formal Approach to Object-Oriented Conceptual Modeling*. Servicio de Publicaciones de la Universidad Politécnica de Valencia (SPUPV-98.4011), ISBN 84-7721-663-0, 1998. (in Spanish)

[8] Letelier P., Sánchez P. and Ramos I. *Prototyping a requirements specification through an automatically generated concurrent logic program*. Gupta (Ed.) Practical Aspects of Declarative Languages, Lecture Notes in Computer Science LNCS 1551, pages 31-45, Springer-Verlag, 1998.

[9] Letelier P., Sánchez P. and Ramos I. *Process Specification for Objects interpreted into Dynamic Logic*. III Jornadas de Ingeniería de Software (JIS'98), Murcia, pages 281-292, 1998. (in Spanish)

[10] Pastor O. and Ramos I. *OASIS versión2(2.2): A Class-Definition Language to Model Information Systems Using an Object-Oriented Approach*. Servicio de Publicaciones Universidad Politécnica de Valencia, SPUPV-95.788, 1995.

[11] Pastor O., Insfrán E., Pelechano V., Romero J., and Merseguer J. *OO-METHOD: An OO Software Production Environment Combining Conventional and Formal Methods*. Proceedings of Conference on Advanced Information Systems Engineering, CAiSE '97, pages 145-158, Barcelona, 1997.

[12] Meyer J.-J.Ch. *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.

[13] OBLOG Software S.A. *The OBLOG Software Development Approach*. (White Paper). 1999, `http://www.oblog.pt/Download/Documentation.exe`

[14] Rolland C., Ben Achour C., Cauvet C., Ralyté J., Sutcliffe A., Maiden N.A.M., Jarke M., Haumer P., Pohl K., Dubois E. and Heymans P. *A Proposal for a Scenario Classification Framework*. Technical Report CREWS 96-01, http://sunsite.informatik.rwth-aachen.de/CREWS/reports96.htm.

[15] Sánchez P., Letelier P. and Ramos I. *Constructs for prototyping information systems with object Petri Nets*. Proceeding of the IEEE International Conference on SMC'97, pages 4260-4265, Orlando, 1997.

[16] Siddiqi J., Morrey I.C., Roast C.R. and Ozcan M.B. *Towards quality requirements via animated formal specifications*. Annals of Software Engineering, n.3, 1997.

[17] Taivalsaari A. *On the Notion of Inheritance*. ACM Computing Surveys, Vol. 28, N. 3, September 1996.

[18] Wieringa R., Jonge W. and Spruit P. *Roles and dynamic subclasses: a modal logic approach*. Vrije U., The Netherlands, 1995.

A An example

In this example a vending machine (vm) accepts coins (coin_in) increasing the customer credit (credit). This credit decreases (coin_out) when a product is given (give). The machine also allows canceling the operation (CANCEL) and then returning all the stored coins. The machine has a warning light (light_empty) that is switched on if the action switch_on occurs. The *OASIS* 3.0 specification of vm is the following:

```
class vm
identification
 number: (number);
constant attributes
 number :nat;
variable attributes
 credit :nat(0);
 light_empty :bool(false);
events
 set new;
 coin_in;
 coin_out;
 light_empty;
 give;
valuations
 [::switch_on]light_empty:=true;
 [coin_in]credit:=credit+1;
 [::coin_out]credit:=credit-1;
 [give]credit:=credit-1;
preconditions
 CANCEL if {credit > 0};
 give if {credit > 0}
operations
 CANCEL:
 CANCEL1= {credit > 1}::coin_out.CANCEL1
  + {credit=1}::coin_out;
end class
```

This machine is specialized in chocolate machines (choc_vm) and others (other_vm). This is a static partitioning, thus the *OASIS* specification is the following:

```
choc_vm, other_vm
        static specialization of vm;
```

An object belonging to the class choc_vm has chocolates as products and their quantity is num_chocs. This quantity is reduced by one unit when giving a chocolate (give). It is not possible (a prohibition) to obtain a chocolate if there is neither credit nor chocolate. In *OASIS* we have:

```
class choc_vm
variable attributes
 num_chocs nat(0);
valuations
 [give] num_chocs:=num_chocs-1;
preconditions
 give if {credit > 0 and num_chocs > 0}[10]
end class
```

We will now consider a dynamic partitioning. The idea is to make a distinction between chocolate machines with products (with_choc) and without products (no_choc). A machine of type no_choc must switch on the warning (light_empty). In *OASIS* we have:

```
no_choc {num_chocs=0},
with_choc {num_chocs > 0}
 dynamic specialization of choc_vm;
class no_choc
triggers
 ::switch_on when {light_empty=false};
end class
```

We will also include a static partition of class choc_vm. Thus we make a distinction between bounded[11] (bounded_choc) and unbounded (unbounded_choc). The following *OASIS* specification gives the detailed information of this partition:

```
bounded_choc, unbounded_choc
 static specialization of choc_vm;

class unbounded
protocols
 GETCHOC:
  GETCHOC1=coin_in.GETCHOC2;
  GETCHOC2={credit=1}give.GETCHOC1 + {credit > 1}give.GETCHOC2
      + coin_in.GETCHOC2 +   {credit=1}::coin_out.GETCHOC1
      + {credit > 1}::coin_out.GETCHOC2;
end class

class bounded
protocols
 GETCHOC:
 GETCHOC1=coin_in.GETCHOC2;
 GETCHOC2=give.GETCHOC1+ coin_in.GETCHOC3 +::coin_out.GETCHOC1;
```

[10] It is mandatory in *OASIS* to repeat the full inherited condition and then to expend the formula in the subclass specialization.

[11] A bounded machine has a limited admission of credit.

```
 GETCHOC3=give.GETCHOC2+ coin_in.GETCHOC4 +::coin_out.GETCHOC2;
 GETCHOC4=give.GETCHOC3 + ::coin_out.GETCHOC3;
end class
```

Eventually, vm is dynamically partitioned in working and broken_down. After the creation action (set) the machine starts in the working state. A machine may be repaired (repair) to reach the working state. When the action break_down occurs, the machines moves to the state broken_down where the events cancel, coin_in and give are not habilitated. The specification in *OASIS* (see the Figure 5) is the following one:

```
broken_down, working
 dynamic specialization of vm
 migration relation is
        vm = set.working;
        working = break_down.broken_down;
        broken_down = repair.working;

class broken_down
events
 repair;
preconditions
 coin_in if {false};
 CANCEL if {false};
 give if {false};
end class

class working
events
 break_down;
end class
```

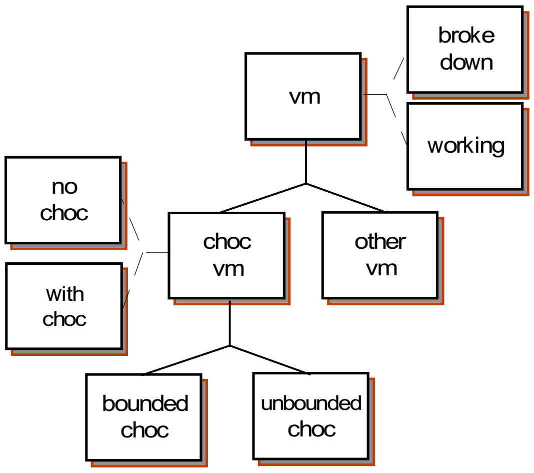

Figure 5: Specialization hierarchy.

A.1 *DL* for each *OASIS* class

We give here a *DL* formulae set for each class mentioned above:

<u>class vm</u>

```
credit=N →[coin_in] credit=N+1
credit=N →[::coin_out] credit=N-1
```

```
credit=N →[give] credit=N-1
[::switch_on] light_empty=true
[set] cancel=0
[CANCEL] cancel=CANCEL1
cancel=CANCEL1 ∧ credit > 1 →[::coin_out] cancel=CANCEL1
cancel=CANCEL1 ∧ credit=1 →[::coin_out] cancel=0
¬(credit > 0) →[give] false
¬(credit > 0) →[CANCEL] false
(cancel=CANCEL1 ∧ credit > 1) ∨ (cancel=CANCEL1 ∧ credit=1)
→[¬::coin_out] false
```

class choc_vm

```
num_chocs=N →[give] num_chocs=N-1
¬(num_chocs > 0 ∧ credit > 0) →[give] false
```

class bounded_choc

```
[set] getchoc=GETCHOC1
getchoc=GETCHOC1 →[coin_in] getchoc=GETCHOC2
getchoc=GETCHOC2 →[coin_in] getchoc=GETCHOC3
getchoc=GETCHOC3 →[coin_in] getchoc=GETCHOC4
getchoc=GETCHOC2 →[choc] getchoc=GETCHOC1
getchoc=GETCHOC3 →[choc] getchoc=GETCHOC2
getchoc=GETCHOC4 →[choc] getchoc=GETCHOC3
getchoc=GETCHOC2 →[::coin_out] getchoc=GETCHOC1
getchoc=GETCHOC3 →[::coin_out] getchoc=GETCHOC2
getchoc=GETCHOC4 →[::coin_out] getchoc=GETCHOC3
¬((getchoc=GETCHOC1)∨ (getchoc=GETCHOC2)
∨ (getchoc=GETCHOC3))→[coin_in] false
¬((getchoc=GETCHOC2) ∨ (getchoc=GETCHOC3) ∨ (getchoc=GETCHOC4))→[choc]
false
¬((getchoc=GETCHOC2)∨ (getchoc=GETCHOC3)
∨ (getchoc=GETCHOC4))→[::coin_out] false
```

class unbounded_choc

```
[set] getchoc=GETCHOC1
getchoc=GETCHOC1 →[coin_in] getchoc=GETCHOC2
getchoc=GETCHOC2 ∧ (credit=1) →[give] getchoc=GETCHOC1
getchoc=GETCHOC2 ∧ (credit > 1) →[give] getchoc=GETCHOC2[12]
getchoc=GETCHOC2 →[coin_in] getchoc=GETCHOC2
getchoc=GETCHOC2 ∧ (credit=1) →[::coin_out] getchoc=GETCHOC1
getchoc=GETCHOC2 ∧ (credit > 1) →[::coin_out] getchoc=GETCHOC2
¬((getchoc=GETCHOC1) ∨ (getchoc=GETCHOC2))→[coin_in] false
¬((getchoc=GETCHOC2) →[give] false
¬((getchoc=GETCHOC2) →[::coin_out] false
```

class no_choc

```
light_empty=false →[¬::switch_on] false
```

12 This formula and next one may be deleted due to not modifying the variable value of the process.

class broken_down

```
¬(false) →[coin_in] false
¬(false) →[CANCEL] false
¬(false) →[give] false
```

A.2 Representation of dynamic partitions

As mentioned above, properties of dynamic partitions need to be included in the partitioned class. The properties of the class choc_vm after incorporating the properties of classes no_choc and with_choc are the following ones:

```
num_chocs=0 ∧ light_empty=false →[¬::switch_on] false[13]
num_chocs=N →[give] num_chocs=N-1
¬(num_chocs > 0 ∧ credit > 0) →[give] false
```

The inclusion of properties of the subclasses broke_down and working into the class vm plus the migratory process, gives the following set of *DL* formulae:

```
credit=N →[coin_in] credit=N+1
credit=N →[::coin_out] credit=N-1
credit=N →[give] credit=N-1
[::switch_on] light_empty=true
[set] cancel=0
[CANCEL] cancel=CANCEL1
cancel=CANCEL1 ∧ credit > 1→[::coin_out] cancel=CANCEL1
cancel=CANCEL1 ∧ credit=1→[::coin_out] cancel=0
¬(vm=working) →[give] false[14]
¬(vm=working) →[CANCEL] false
¬(vm=working) →[coin_in] false
(cancel=CANCEL1 ∧ credit > 1) ∨ (cancel=CANCEL1 ∧ credit < =1)
→[::coin_out] false
[set] vm=working[15]
vm=working ∧¬(credit > 0)→ [give] false
vm=working ∧ ¬(credit > 0)→ [CANCEL] false
vm=working → [break_down] vm=broken_down
vm=broken_down → [repair] vm=working
¬((vm=working) →[break_down] false
¬((vm=broke_down) →[repair] false
```

A.3 Representation of static partitions

After dealing with dynamic partitions we need to derive the *DL* formulae for each path from each leaf class to the top class.

[13] This formula is moved here from the dynamic subclass.

[14] This permission and the next two ones replace the prohibitions of vm due to be more restrictive.

[15] This formula and the following one are related to the migratory process of the dynamic partition

class vm - choc_vm - bounded

```
num_chocs=N → [give] num_chocs=N-1
credit=N → [coin_in] credit=N+1
credit=N → [::coin_out] credit=N-1
credit=N → [give] credit=N-1
[::switch_on] light_empty=true
[set] getchoc=GETCHOC1
getchoc=GETCHOC1 → [coin_in] getchoc=GETCHOC2
getchoc=GETCHOC2 → [coin_in] getchoc=GETCHOC3
getchoc=GETCHOC3 → [coin_in] getchoc=GETCHOC4
getchoc=GETCHOC2 → [choc] getchoc=GETCHOC1
getchoc=GETCHOC3 → [choc] getchoc=GETCHOC2
getchoc=GETCHOC4 → [choc] getchoc=GETCHOC3
getchoc=GETCHOC2 → [::coin_out] getchoc=GETCHOC1
getchoc=GETCHOC3 → [::coin_out] getchoc=GETCHOC2
getchoc=GETCHOC4 → [::coin_out] getchoc=GETCHOC3
[set] vm=working
vm=working → [break_down] vm=broken_down
vm=broken_down → [repair] vm=working
[set] cancel=0
[CANCEL] cancel=CANCEL1
cancel=CANCEL1 ∧ credit > 1 → [::coin_out] cancel=CANCEL1
cancel=CANCEL1 ∧ credit=1 → [::coin_out] cancel=0
¬(vm=working) → [give] false
¬(vm=working) →[CANCEL] false
¬(vm=working) → [coin_in] false
¬((vm=working) →[break_down] false
¬((vm=broke_down) →[repair] false
¬((getchoc=GETCHOC1)∨ (getchoc=GETCHOC2)
∨ (getchoc=GETCHOC3))→[coin_in] false
¬((getchoc=GETCHOC2)∨ (getchoc=GETCHOC3) ∨ (getchoc=GETCHOC4))→ [choc]
false
¬((getchoc=GETCHOC2)∨ (getchoc=GETCHOC3)
∨ (getchoc=GETCHOC4))→[::coin_out] false
¬(num_chocs > 0 ∧ credit > 0) → [give] false[16]
(cancel=CANCEL1 ∧ credit > 1) ∨ (cancel=CANCEL1 ∧ credit=1)
→ [::coin_out] false
num_chocs=0 ∧ light_empty=false → [¬::switch_on] false
```

class vm - choc_vm - unbounded

```
num_chocs=N → [give] num_chocs=N-1
credit=N → [coin_in] credit=N+1
credit=N → [::coin_out] credit=N-1
credit=N → [give] credit=N-1
[::switch_on] light_empty=true
[set] vm=working
vm=working → [break_down] vm=broken_down
vm=broken_down → [repair] vm=working
[set] cancel=0
[CANCEL] cancel=CANCEL1
```

16 This prohibition formulae associated to get event what have just been mentioned just above can be mixed into one formula by using the logic connector '∨'.

```
cancel=CANCEL1 ∧ credit > 1→ [::coin_out] cancel=CANCEL1
cancel=CANCEL1 ∧ credit=1→ [::coin_out] cancel=0
¬(vm=working) → [give] false
¬(vm=working) →[CANCEL] false
¬(vm=working) → [coin_in] false
¬((vm=working) →[break_down] false
¬((vm=broke_down) →[repair] false
¬(num_chocs > 0 ∧ credit > 0) → [give] false
(cancel=CANCEL1 ∧ credit > 1) ∨ (cancel=CANCEL1 ∧ credit=1)
→ [::coin_out] false
num_chocs=0 ∧ switch_on=false → [¬::light_empty] false
[set] getchoc=GETCHOC1
getchoc=GETCHOC1 →[coin_in] getchoc=GETCHOC2
getchoc=GETCHOC2 ∧ (credit=1) →[give] getchoc=GETCHOC1
getchoc=GETCHOC2 ∧ (credit > 1) →[give] getchoc=GETCHOC2[17]
getchoc=GETCHOC2 →[coin_in] getchoc=GETCHOC2
getchoc=GETCHOC2 ∧ (credit=1) →[::coin_out] getchoc=GETCHOC1
getchoc=GETCHOC2 ∧ (credit > 1) →[::coin_out] getchoc=GETCHOC2
¬((getchoc=GETCHOC1) ∨ (getchoc=GETCHOC2))→[coin_in] false
¬(getchoc=GETCHOC2) →[give] false
¬(getchoc=GETCHOC2) →[::coin_out] false
¬(getchoc=GETCHOC1) ∨ (getchoc=GETCHOC2) →[coin_in] false
¬(getchoc=GETCHOC2) → [give] false
¬(getchoc=GETCHOC2) → [::coin_out] false
```

<u>class vm - other_vm</u>

```
credit=N →[coin_in] credit=N+1
credit=N →[::coin_out] credit=N-1
credit=N →[give] credit=N-1
[::switch_on] light_empty=true
[set] cancel=0
[CANCEL] cancel=CANCEL1
cancel=CANCEL1 ∧ credit > 1→[::coin_out] cancel=CANCEL1
cancel=CANCEL1 ∧ credit=1→[::coin_out] cancel=0
¬(vm=working) →[give] false
¬(vm=working) →[CANCEL] false
¬(vm=working) →[coin_in] false
(cancel=CANCEL1 ∧ credit > 1) ∨ (cancel=CANCEL1 ∧ credit < =1)
→[::coin_out] false
[set] vm=working
vm=working → [break_down] vm=broken_down
vm=broken_down → [repair] vm=working
¬((vm=working) →[break_down] false
¬((vm=broke_down) →[repair] false
```

A.4 Properties of species

[17] This formula and the next one can be deleted although we prefer to keep them in order to make the translation process clearer.

As mentioned above, it is possible to deduce the properties (as *DL* formulae) for each species as the union of the respective properties. In the example we distinguish the following species involving only static partitions[18]:

```
other_vm
bounded_choc
unbounded_choc
```

Theses species correspond with each class mentioned above. In order to complete the calculus of properties we present here the set of identification mechanisms, attributes and events deduced from the specialization hierarchy of our example.

A.4.1 Identification mechanisms

The unique available identification mechanism (number) has been defined in the class vm. This mechanism is inherited by every species.

A.4.2 Attributes and events

We present the attribute and event sets for each species. We assume that the possible conflicts caused by coincidental names have been previously adjusted as a part of the consistency verification process of the specification.

Specie vm - choc_vm - bounded

```
Atr={number, credit, switch_on, num_chocs}
Ev={set, coin_in, coin_out, light_empty, give, CANCEL, repair,
break_down}
```

Specie vm - choc_vm - unbounded

```
Atr={number, credit, switch_on, num_chocs}
Ev={set, coin_in, coin_out, light_empty, give, CANCEL, repair,
break_down}
```

Especie vm - other_vm

```
Atr={number, credit, switch_on}
Ev={set, coin_in, coin_out, light_empty, give, CANCEL, repair,
break_down}
```

[18] In this example, taking into account that there is only one hierarchy and that there is only one double static partition of the same class, species are formed by only one class. In a more general case it should be necessary to connect the properties of each class.