Preserving behaviour: Why and How

Fabienne Cathala[1,3] Pascal Poncelet[2,3]

[1] Cemagref division Aix-en-Provence - [2] IUT Aix-en-Provence

[3] LIM - URA CNRS 1787 - Université de la Méditerranée
Faculté des Sciences de Luminy, Case 901, 163 Avenue de Luminy,
13288 Marseille Cedex 9, FRANCE
E-mail: cathala@lim.univ-mrs.fr

Abstract

In this paper, we propose a generic method for elaborating the behavioural specification dictionary of applications. It could apply in the context of various conceptual modelling approaches and take advantage of functionalities provided by associated CASE tools. The method is based on a meta-schema abstracting the behavioural concepts by using the structural abstractions of the chosen modelling approach. Once storage structures are generated from this meta-schema, they can be populated in an automated way by examining dynamic schemas specified by designers. The method is intended for dealing with particular applications in which behaviour must be preserved.

1 Motivations

There are some kinds of applications in which behaviour must be represented not to be simulated or enhanced (through some executive programs or even active rules [Ha90, LNR87, TPC94b]) but to be preserved.
Reasons behind "storing behaviour" are various. First of all, by keeping the whole trace of the application dynamics, scope of queries can be extended to the application history. But a main difference between our concern and temporal or historical database approaches [LZ88] is that we aim to capture much more semantics about application behaviour. For instance we would like to express the following queries: what happened when such event occurred? Which were the reactions of such or such object? Another motivation is storing "behaviour patterns", i.e. particular possible behaviours which could be either critical or good behaviours. Whenever objects adopt such behaviours, they must be detected automatically, thus it is possible to control the application evolution over time or to anticipate critical situations. For instance, in a library management application, users could be classified according to their behaviour over time in different categories such as "good or bad borrower", "occasional", ... We imagine that, depending on their category, they could be offered some special privileges (or in contrast privileges could be revoked).

Adressing the issue of storing behaviour requires dealing with two different representation models. On one hand, dynamic conceptual models are really suitable for representing application behaviour, as well as behaviour patterns. On the other hand, the only models available for storage are database models, i.e. structural models. In this paper, we propose a generic method for preserving behaviour. It could be used with various modelling approaches and yields the behavioural specification dictionary in form of instanciated storage structures (based on such or such

database model). The proposed method has two interesting qualities: its simplicity and a non expensive enhancement particularly when integrated within the CASE tool supporting the chosen modelling approach.
The paper is organized as follows. Section 2 gives the general principles of our method. These principles are enforced through two different experimentations. The first one, summarized in section 3, is based on the OMT approach [RBP$^+$91]. It serves as an illustration of the method feasability. The groundwork for the second experimentation, described in section 4, is the IFO$_2$ approach [PTCL93, TPC94a]. We take advantage of the associated CASE tool and complement it by an additional functionality. Section 5 proposes a brief survey of related work, in particular through a comparison with our method.

2 Preserving behaviour: general principles

Behaviour representation has been addressed by various conceptual approaches among which we could quote OMT, OOD, OOA-OOD, ... [Boo91, RBP$^+$91, Som91, SM88, SM92, SSE87, Saa91]. They provide designers with a high-level and object-oriented model. It is complemented by tranformation mechanisms applying to conceptual schemas and yielding specifications which could be implemented using various target systems (classical and object DBMSs or languages) [RBP$^+$91, BM91, PTCL93]. For describing the dynamics of applications, these approaches frequently adopt state-transition diagrams.

Given our particular concern of preserving behaviour, conceptual approaches provide a dynamic representation for abstracting application behaviour, along with structural representation facilities. The former capability meet our need of specifying behaviour patterns, or possible behaviours; the latter offers high-level abstractions and mechanisms for modelling and designing database schemas. In such a context, our approach (its general principles are described in figure 1) is based on a meta-schema describing the dynamic concepts by using the structural abstractions of the model. Meta-data is frequently required for capturing meaning, content, organization or purpose of data, and various approaches make use of meta-schemas. Some of them are summarized in [SM91].

Once the meta-schema is specified, any application behaviour modelled by designers is seen as an instance of the meta-schema. On an implementation level, storage structures can be generated from the meta-schema, by using provided transformation mechanisms. Such facilities must be complemented in order to deal with instances. In fact, an automatic instanciation can be performed. It applies to dynamic conceptual descriptions and populates object classes or relations generated from the meta-schema.
Thus, the dynamic dictionary of the application is achieved and can be handled by application programmers merely through the query language of target systems. Such a facility could be an interesting tool in software engineering, in particular for cooperative work, because it is the basis of a dynamic repository (with such or such underlying storage model) while offering a high-level description of behaviour (interfaces with aided tools supporting the chosen conceptual model can be easily developped).

Following from this principle, we examine the enhancement of our method with two different conceptual models: OMT and IFO$_2$. OMT is representative of several modelling approaches. It proposes an object-oriented extension of the Entity/Relationship model and its dynamic model is based on statecharts [Har88]. On

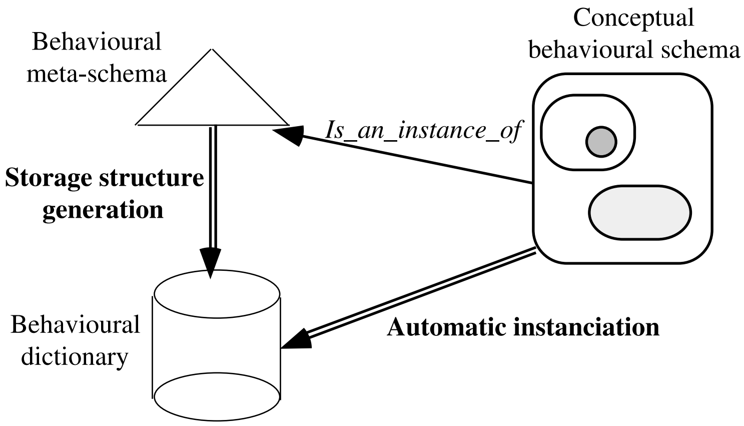

Figure 1: General principles of the approach

the other hand, IFO_2 adopts a "whole-object" and "whole-event" philosophy by describing the static and dynamic aspects of applications.
In these two different contexts, we apply the described approach as detailed in the following sections.

3 Enhancement with OMT

In OMT [RBP$^+$91], object dynamics is modelled through state-transition diagrams. Basically, being in a given state, an object could evolve by performing a transition and enter another state. A transition, triggered by event, is constrained by a condition or guard and can perform actions. State-transition diagrams could be nested (state generalization): a state can be expanded in a lower-level diagram offering a refined vision of a behaviour part, in terms of states and transitions.
Furthermore, diagrams could be splitted into concurrent sub-diagrams and diagrams, at the highest level, reflect behaviour of object classes, inherently concurrent (state aggregation). Synchronization between objects is based on event sending.

Exemple 1 As an illustration, we consider the behaviour of two kinds of objects in the library application: "Book-Copies" and "Users".
These behaviours are captured through the diagrams in figure 2. Let us consider the state diagram describing the user's behaviour, as regards the registration fees of the library. His current situation is captured through the attribute "status". The proposed diagram encompasses three states: "Registred", "Waiting" and "Excluded" corresponding to the possible values of "status". We imagine that a user changes from "Registred" to "Waiting" if he is late in paying for registration. From the state "Waiting", the transition to "Registred" can be performed, if the user decides to pay, else, after a certain delay, the user's privileges are revoked and the object state becomes "Excluded" (corresponding to the "User" diagram exit point).
To illustrate the concurrent sub-diagram mechanism, let us now examine the diagram "Registred"; it corresponds to the entry point of the diagram "User" and it is splitted into two sub-diagrams "Subscriver-State" and "Borrow-State". When the transition "t4" is performed, the concerned object enters a state in each sub-diagram. "Subscriver-State" is expanded in a lower-level diagram, encompassing the states "User" (no current loan), "Borrower" and "Reminded". This state is entered whenever a user do not return the borrowed books in time. user borrowing books. A very mere state diagram of "Book-copy" objects is also proposed

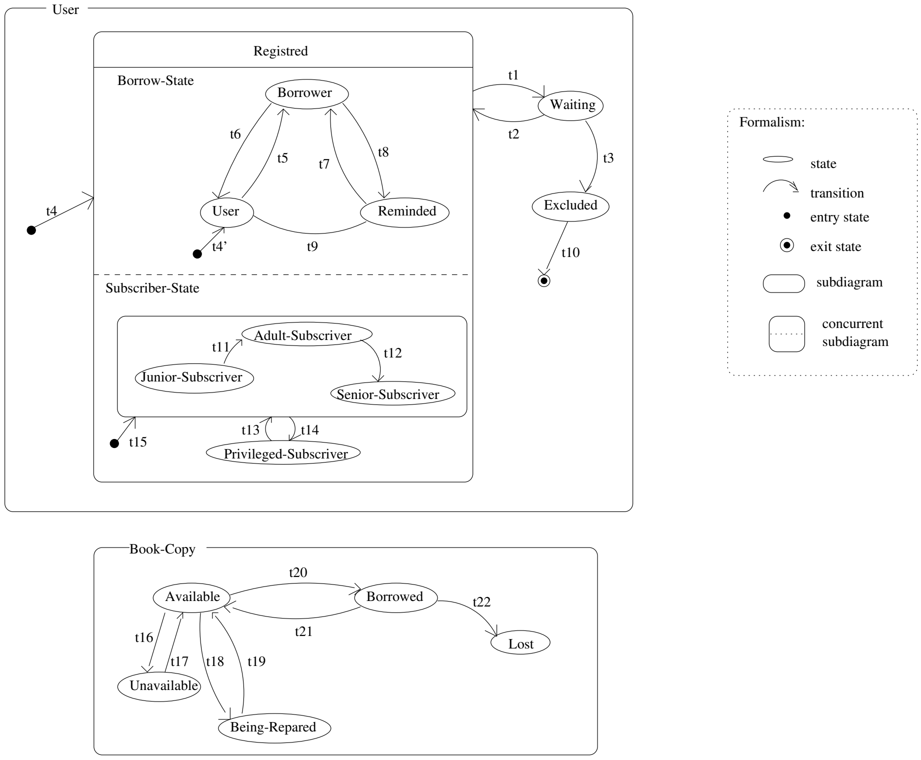

Figure 2: Library application dynamics with OMT

The behavioural meta-schema, to be specified, must capture not only the basic concepts but also the abstractions of state generalization and aggregation. The meta-schema is partially depicted in figure 3. In fact, we focus on the more interesting aspects when abstracting the dynamic model and do not detail the description of all the concepts.

Due to its recursive feature, state generalization requires suitable structural abstractions and a clear cut vision of activity propagation through the various description levels, since diagrams could be arbitrarily nested. For reflecting the nesting mechanism, we use the recursive aggregation of the object model, and adopt a uniform vision of states and diagrams. More precisely, a DIAGRAM could be specialized in a simple STATE (at the lowest level) or in a SUB-DIAGRAM (at whatever higher level). A SUB-DIAGRAM is described as an aggregation of transitions, and in its turn TRANSITION is modelled as a recursive aggregation having the following components: INITIAL, FINAL, EVENT, CONDITION, and ACTION. The two former components correspond to the diagram exited or entered by the transition, thus, at the lowest-level, they are merely the initial and final states of the transition. This representation makes it possible to capture not only transitions between states or between higher-level diagrams but also transitions indicating entry or exit points in a diagram. When specified, latter transitions must be necessarily triggered to enter or exit a diagram. They can correspond to the beginning or the end of object life cycle, and they play an interesting part in transition inheritance from a higher level diagram. In fact, when a diagram is entered possible entry points are the only subdiagrams (or states) which can be reached. On the other hand, exit points must be necessarily reached before performing a transition exiting the diagram. For these particular transitions, the INITIAL (or FINAL) component has no instance. The

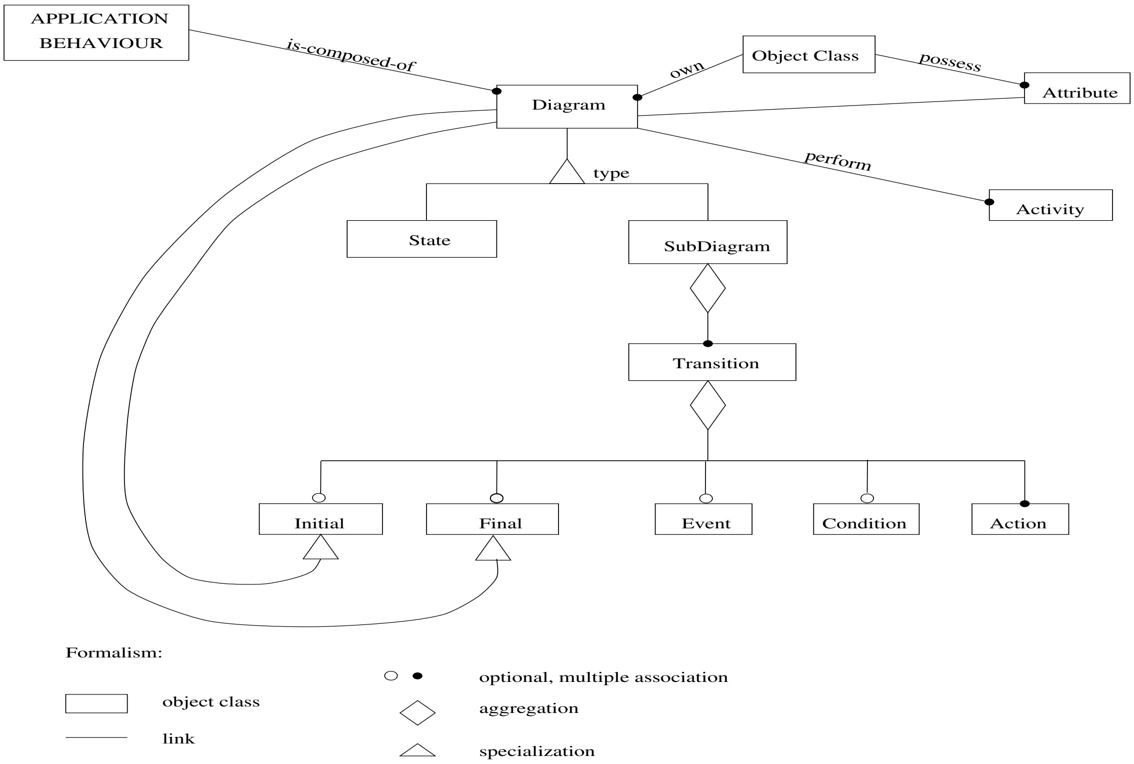

Figure 3: Behavioural meta-schema with OMT

other components, describing Transition, i.e. Event, Condition, and Action, capture the possible event triggering a transition, its guard if any, and the actions to be performed.
The meta-schema being defined, the second step of our approach can be applied by using the transformation mechanisms defined for the object model of OMT. We choose the relational model as transformation target, and achieve from the behavioural meta-schema, the relational schema partially given below.

Application (Name, ...);

Diagram (Id_d, *A_id*, Type, *O_id*, *Name*);

Transition (Tr_id, Initial, Final, E, C, *D_id*);

Attribute (A_id, ...);

Object_class (O_id, ...);

Action (A_id, Tr_id, ...);

When performing such a mapping, we assume that diagrams in the behavioural specifications are identified through the combination of their name and the attribute which is abstracted. Transitions are provided with an artificial identifier (prefered to the composite candidate key combining the attributes Initial, Final and Event).

Relations reflecting the meta-schema are then automatically populated from dynamic representations specified by designers. Thus the underlying mechanism performs the instanciation of the behavioural dictionary of an application. Before describing the corresponding algorithms we need to introduce the following definitions.

Preliminary definitions

- Let D be the set of dynamic diagrams in the designer specifications.
  $\forall d \in D$, d is defined as a couple $(d_i d, T_d)$, where T_d is the set of transitions in d.
  $\forall t \in T_d$, t is defined as a tuple $(t\_id, init, fin, E, C, A)$ where $init$ and end $\in D$, stand for the diagrams exited and entered by t, and E, C, A symbolize the event, condition and action clauses of t.
- Let $S \subset D$ be the set of states such that: $\forall d \in S$, $T_d = \emptyset$;
  and $H \in D$, the set of highest-level diagrams, such that:
  $d \in H$, $\nexists d' \in D$ and $\nexists t' \in T'_d$ / $t'.init = d$ or $t'.fin = d$ where $t'.init$ ($t'.fin$ resp.) is the initial (final resp.) diagram of t'.
- $Marked\_Diag$ is an intermediary set used to store diagrams already examined.

Instanciation algorithm
The instanciation mechanism starts from the highest-level diagrams by performing for each one the procedure Diag-Instanciation. This procedure captures the diagram semantics (merely by operating suitable insertions) and then examines its transitions. In order to insert the tuple describing a transition, diagrams entered or exited by the transition must be captured. This is done by applying once again the procedure Diag-Instanciation to the initial and final diagrams of the considered transition. Thus, if transition is defined between high level diagrams, it is inserted only when the diagrams are fully described (with their own sub-diagrams and transitions).

```
prog Schema_Instanciation
  for each d in H do
    Marked_Diag ← ∅
    Diag_Instanciation(d)
  endfor
endprog


proc Diag_Instanciation(d)
  Insertion(DIAGRAM,d)
  if d ∉ S then
    for each t in T_d do
      if t.init ≠ ∅ and t.init ∉ Marked_Diag then
        Marked_Diag ← Marked_Diag ∪ {t.init}
        Diag_Instanciation(t.init)
      endif
      if t.fin ≠ ∅ and t.fin ∉ Marked_Diag then
        Marked_Diag ← Marked_Diag ∪ {t.fin}
        Diag_Instanciation(t.fin)
```

```
          endif
          Insertion(TRANSITION,t)
        endfor
      endif
    endproc
```

Through this section, we show the feasability of our method by using OMT. The chosen dynamic model is really suitable for representing how objects evolve over time. Nevetheless, it is not adapted to set-oriented process. This is why we propose a second experimentation in the following section.

4 Enhancement with IFO_2

IFO_2 is a conceptual approach encompassing both structural and behavioural representation capabilities. Its originality is to offer symetric concepts for the static and dynamic descriptions. One of the main difference with OMT is that dynamics is described in an overall way (and not object class by object class), by specifying events of various semantics and their relationships (synchronization or triggering). Basic event types are proposed (external, temporal, operation invocation). To express synchronization, they could be combined using various event constructors (aggregation, sequence, grouping, union). Event types, either basic or complex, are organized by using the key concept of event fragment. A fragment describes the system reactions when faced by particular circumstances (or events). It necessarily encompasses an event type, called the fragment heart, possibly related to other types by mean of triggering functions. A precedence function must be specified whenever events of the heart type must be preceded by other events. In short, a fragment describes the causality relationships between a precedent type, a heart type and triggered types. Fragments are integrated within event schemas by using represented types and IS_A links. A represented type can symbolize any fragment since it stands for its heart. It could be specified for a twofold reason: re-using, in a fragment, the description of another fragment or refining the latter description (i.e. specializing it through additional precedent or triggered types).

Exemple 2 Figure 4 represents the library application dynamics. This schema is composed of five fragments related by IS_A links. The fragments "Subscription_Application" and "Loan_Application" capture external events: user's request for library registration and borrowing request, along with operations which are performed when such events occur. The precedence relationship between the two external types is modelled through the represented type "Registred_Subscriver" related with a precedence function to "Loan_Application". The loan management is based on the fragment "Loan" (represented with a constructor sequence), which successively trigger the simple types "Unavailable" (the corresponding method applies to the borrowed book which becomes unavailable) and "Init-Loan" (which actually creates the loan). When a "Loan" event occurs, the number of current loans is increased. (the simple type "Inc-Nb-Loan" performs a method). When a given delay completes, the user could be sent a "Reminder" (perhaps several times). The fragment "Return" describes what happens when a user returns a borrowed book. And finally, the fragment "Closure" models how a loan completes: the borrowed book becomes available or the user is excluded.

The behaviour meta-schema is specified by using the structural concepts of IFO_2 (formally given in [PTCL93]), which are very close to the behavioural concepts.

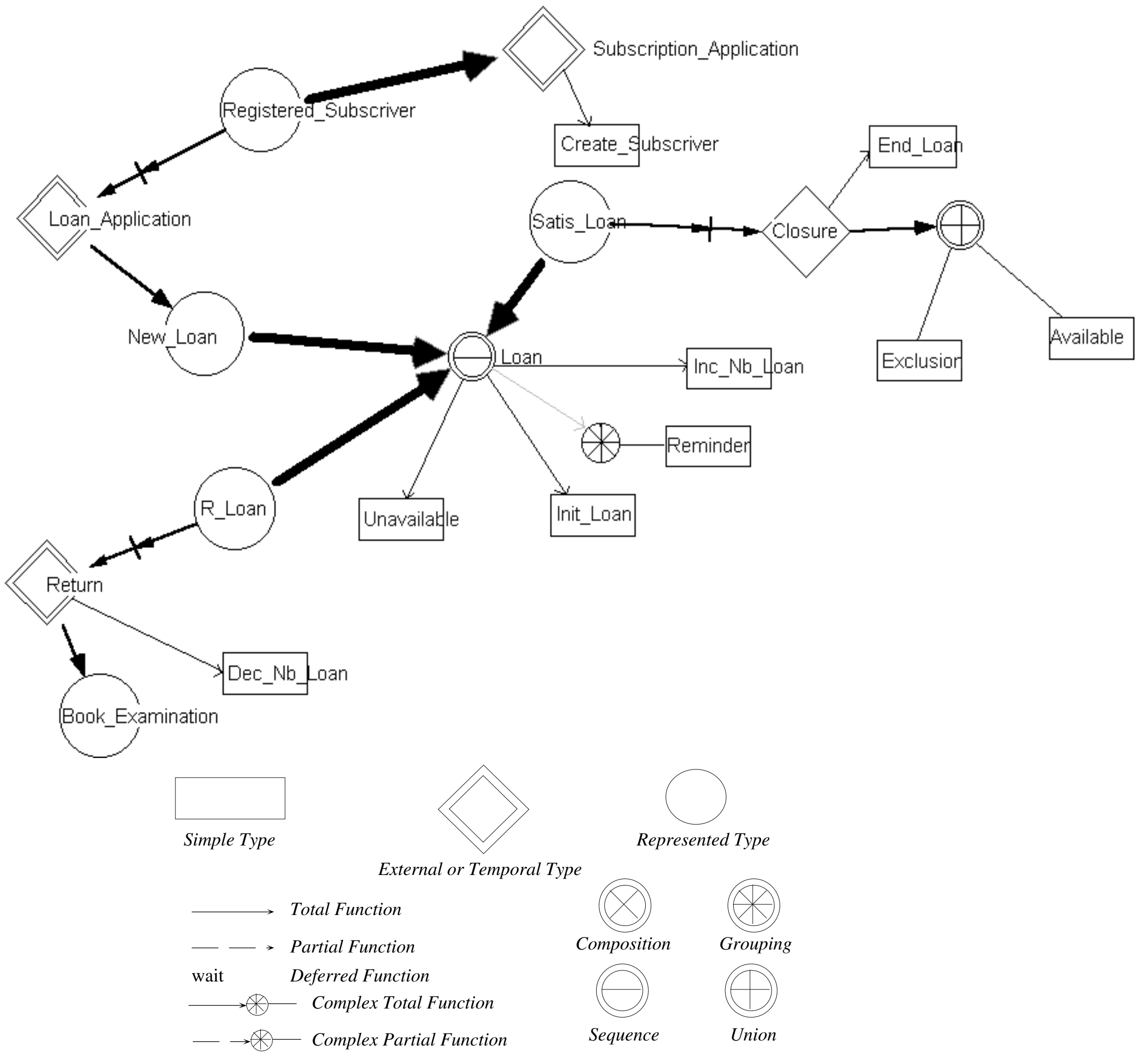

Figure 4: Library application dynamics with IFO_2

More precisely, object types in IFO_2 could be either basic (simple (attribute) or abstract (entity)) or complex. For the latter, various object constructors are defined. Object types are organized through fragments: a principal type, called the heart, is described with its properties, i.e. other object types. Object fragments could be re-used or refined by introducing represented types which stand for the fragment hearts, and IS_A links.

The meta-schema, depicted in figure 5, encompasses two main fragments: Fragment and Type. Additional fragments are specified for specialization or re-using concerns. The fragment Type is devoted to specifying an IFO_2 event type, which is characterized by its name, its category and its parameters. This description is refined in four ways according to the category of the event type in question (thus four additional fragments are introduced).(i) The event type can be simple. In that case, the method invoked must be preserved along with its nature (update operation, request, ...). (ii) When the considered event type is abstract, its nature must be kept (are events of this type external, temporal or internal?). (iii) Let us consider

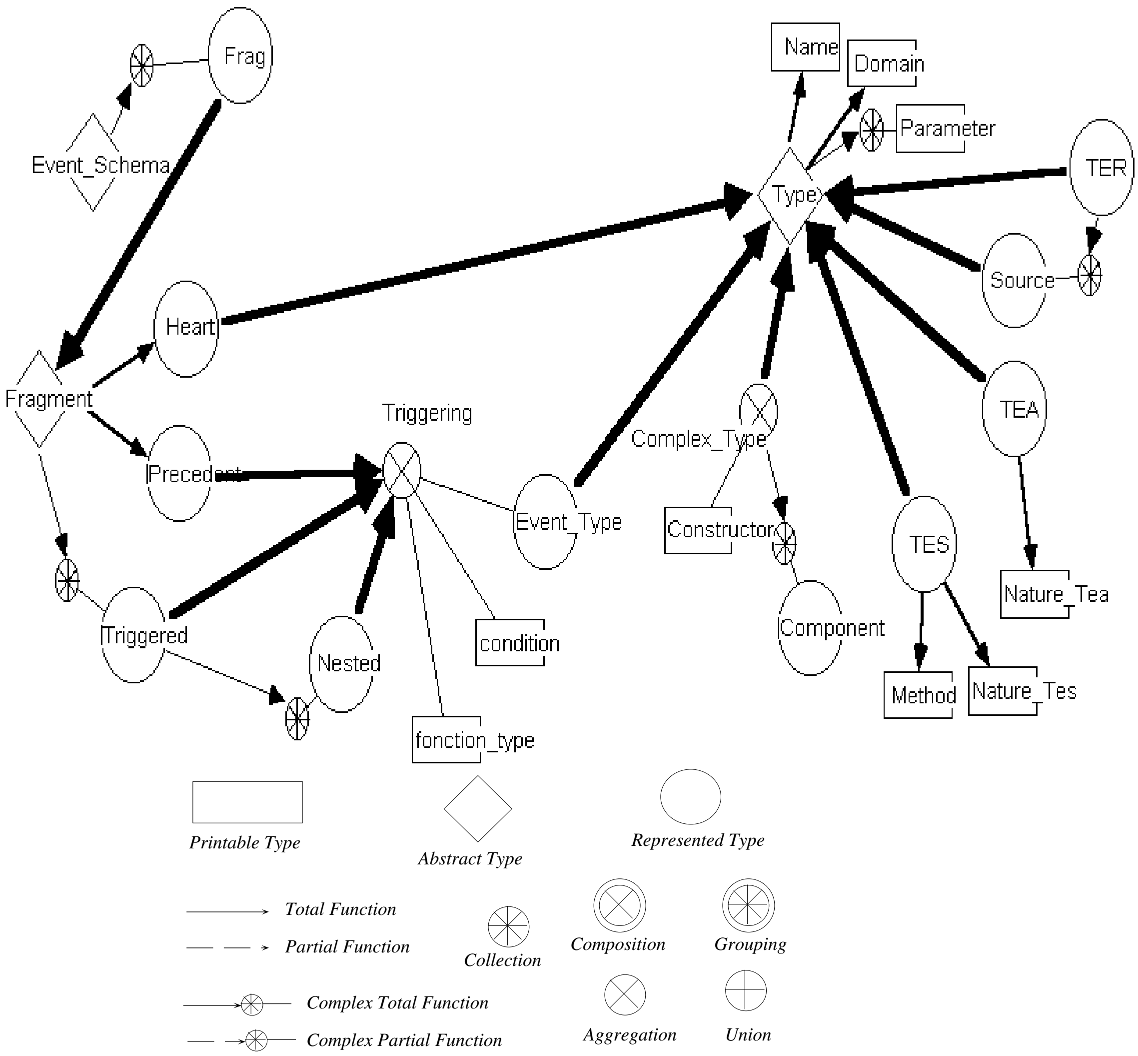

Figure 5: Behavioural meta-schema with IFO_2

now an event type which is a represented type. It has a particular feature since it symbolizes another type (or several) called its source(s). This feature is captured, in the fragment TER, by specifying a necessary and possibly multi-valued property: SOURCE. The latter is a represented type which itself stands for TYPE. (iv) Finally, when the event type under consideration is a complex type, it is abstracted through a recursive construction. A COMPLEX_TYPE is seen as the aggregation of a CONSTRUCTOR applying to a set of COMPONENT; each of which being itself an object of TYPE. With this modelling style, arbitrarily complex types can be represented. The fragment of heart FRAGMENT describes the organization of event fragments. Event fragments are seen as abstract entities characterised by the following properties:

- a necessary and mono-valued property: HEART, which is a represented type standing for a TYPE;
- a mono-valued property: PRECEDENT, is specified only if the considered event fragment encompasses a precedent type;

- a multi-valued property: Triggered, which captures the possible types triggered by the heart. Each triggered type could be itself heart of a sub-fragment. This means that it can, in its turn, trigger other types, called Nested.

The fragment Triggering is introduced in the meta-schema for a single reason: sharing a part of specifications. In fact, when describing a precedent, triggered or nested event type, we need to capture not only the type in question but also the nature of the precedence or triggering function and the condition under which triggering is actually performed. Such a description is encapsuled within the fragment Triggering, merely reduced to its heart (specified as an aggregated type).

```
class C_SchemaEvt
public    type tuple(
          Fragments: set(C_Fragment))
end;

class C_Fragment
public    type tuple(
          heart: C_TE,
          precedent: C_Triggering,
          trig: set(
                    tuple(triggered: C_Triggering,
                              set(C_triggering))))
end;

class C_Triggering
public    type tuple(
          function_type: string,
          condition: string,
          TypeEvt: C_TE)
end;

class C_TE
public    type tuple(
          name: string,
          domain: string,
          parameter: set(string))
end;

class C_Complex_Type inherit C_TE
public    type tuple(
          constructor: string,
          component: set(C_FTE))
end;

class C_TES inherit C_TE
public    type tuple(
          method: string,
          nature: string)
end;

class C_TEA inherit C_TE
public    type tuple(
          nature: string)
end;

class C_TER inherit C_TE
public    type tuple(
          set(C_TE))
end;
```

Figure 6: O_2 classes

From IFO_2 structural schemas, mechanisms for generating code could be enhanced, for relational systems, the Object-Oriented DBMS O_2 or C++. As an illustration, we chose IFO_2 transformation (described in [PTCL93]), which yields when applied to the meta-schema the O_2 inheritance class hierarchy, given in figure 6.

For populating O_2 classes, we define an instanciation mechanism which successively examines event fragments and creates required objects in the suitable classes.

Preliminary definitions

- Let $\mathcal{F}$ be the set of dynamic fragments in the designer specifications. $\forall f \in \mathcal{F}$, f is defined as a couple (T, F_u) where T is a set of types and F_u is a set of functions;

- $\forall t \in T$, t is defined as a tuple $(ident, domain, heart, parameters, category)$ where *ident* is the type identifier, *heart* a boolean, and *category* symbolizes the category type (simple, abstract, ...). Its structure is type-dependent, i.e. holds all relevant items for category instanciation;
- $\forall f_u \in \mathcal{F}_\mathcal{U}$, f_u is defined as a tuple (source_type, target_type, type_fonction, condition).

Instanciation algorithm

```
prog Schema_instanciation
  for each f in 𝓕 do
    Frag_instanciate(f)
    S ← S + {f}
  endfor
endprog


proc Frag_instanciate(f)
  Insertion(FRAGMENT,f)
  for each t in T do
    Type_insertion(t)
    if t.heart=TRUE then
      f.heart ← t.identifier
    endif
  endfor
  for each f_u in 𝓕_𝓤 do
    trigger_insertion(f_u)
    if f_u.type_fonction=precedence then
      f.preced ← f_u.source_type
    else if f_u.type_fonction=triggering then
          f.succ ← f.succ + {f_u.source_type}
        else f.imbr ← f.imbr + {f_u.source_type}
        endif
    endif
  endfor
endproc


proc Type_insertion(t)
  Insertion(TE,t)
  if t.domain=TES then Insertion(TES,t) endif
  if t.domain=TEA then Insertion(TEA,t) endif
```

    if t.domain=complex then Insertion(complex,t) endif
    if t.domain=TER then Insertion(TER,t) endif
endproc

An interesting feature of the IFO_2 model is that it provides the concept of trace (initially introduced in [Hoa85]). The trace is the sequence of all the events ever occurred during the application life. Thus it proposes a chronological vision of IFO_2 event schema instances. Mapping the structure of trace into an O_2 class provides us with a suitable mechanism for storing actual behaviours, only in terms of events (with their time-stamp, parameters, ...). Any object of the class TRACE describes an event and the behavioural dictionary gives the circumstances in which it happens. Thus using the O_2 query language, historical requests can be expressed for retrieving particular events but also causality relationships between events.

5 Related work

Motivated by needs of particular applications, our method addresses a specific issue. However its general objective could be compared to that of historical or temporal database approaches [LZ88, CI94, JSS94]. Actually, in both cases, the motivation is to preserve objet history. Nevertheless, our method boosts this objective by fully capturing the object behaviour, and not only its various states over time. In fact, it could be seen as an additional tool for an improved management of temporal databases for several reasons. First of all, the behavioural meta-schema provides the precise context in which temporal data could be better interpreted. Complementing temporal data, the behavioural dictionary could be used for extending query capabilities. Finally our method can be seen as a modelling aid for defining the schema of a temporal database. Through a static analysis of behavioural specifications along with simulations of real behaviours, relevant states which must necessarily be preserved can be exhibited.
On the other hand, temporal database approaches provide us with possible target models for enhancing our method, without altering its general principles.

In spite of different motivations, let us notice a similarity between our method and research work in data mining field, interested in exhibiting sequential patterns [AIS93, AS94, SA95]. For such a discovering, sequences of events describing the behaviour and actions of users or systems are collected. This list of transactions, each of which encompassing various database operations resemble more closely to the trace structure used when enhancing our method with IFO_2. But of course these approaches place major emphasis on defining suitable mechanisms and non expensive algorithms for extracting knowledge.

6 Conclusion

This paper presents a method for elaborating the behavioural specification dictionary of application. It could be used with different modelling approaches such as OMT or IFO_2. A meta-schema describing behavioural abstractions is defined with the structural concepts of the chosen modelling approach. From this meta-schema, storage structures for various target systems could be yielded applying transformation mechanisms. Instanciation of the behavioural dictionary could then be performed in an automated way.
The motivation behind our proposal is dealing with particular applications in which object behaviour must be carefully observed. Of course the next step is now to define

manipulation capabilities in order to compare real behaviours to patterns. Finally, we intend to define an aided tool for exhibiting behavioural patterns from actual bahaviours. This last perspective really meets a data mining objective.

References

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceeding of the ACM-SIGMOD*, pages 207–216, Washington, D.C., May 1993.

[AS94] R. Agrawal and R. Srikant. fast association for mining association rules. In *Proceeding of the 20th International Conference on Very Large Data Bases (VLDB'94)*, Santiago, Chile, September 1994.

[BM91] M. Bouzeghoub and E. Métais. Semantic Modelling of Object-Oriented Databases. In *Proceedings of the 17th International Conference on Very Large Data Bases (VLDB'91)*, pages 3–14, Barcelona, Spain, September 1991.

[Boo91] G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cumming Comp, 1991.

[CI94] J. Clifford and T. Isakowitz. On the semantics of (bi)temporal variable databases. In *Proceedings of 4th Conference on Extending Database Technology (EDBT'94)*, volume 779 of *Lecture Notes in Computer Science*, pages 215–230, Cambridge, UK, March 1994. Springer Verlag.

[Ha90] D. Harel and al. STATEMATE: A working environmentfor the development of complex reactive systems. *IEEE Transaction on Software Engineering*, 16(4):403–414, 1990.

[Har88] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[Hoa85] C.A.R Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[JSS94] C. Jensen, M. Soo, and R. Snodgrass. Unifying temporal data models via a conceptual model. In *Information Systems*, volume 19 of *7*, pages 513–547, 1994.

[LNR87] J.Y. Lingat, P. Nobecourt, and C. Rolland. Behaviour management in database application. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB'87)*, Brighton, UK, September 1987.

[LZ88] P. Loucopoulos and R. Zicari. Special issue on temporal databases. *Data Engineering Bulletin*, 11(4), December 1988.

[PTCL93] P. Poncelet, M. Teisseire, R. Cicchetti, and L. Lakhal. Towards a Formal Approach for Object-Oriented Database Design. In *Proceedings of the 19th International Conference on Very Large Data Bases (VLDB'93)*, pages 278–289, Dublin, Ireland, August 1993.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[SA95] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceeding of the 21th International Conference on Very Large Data Bases (VLDB'95)*, Zurich, Switzerland, September 1995.

[Saa91] G. Saake. Descriptive Specification of Database Object Behaviour. *Data & Knowledge Engineering*, 6:47–73, 1991.

[SM88] S. Shlaer and S.J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data.* Yourdon Press, Prentice Hall, 1988.

[SM91] M. Siegel and S. Madnick. A Metadat Approach to Resolving Semantic Conflicts. In *Proceedings of the 17th International Conference on Very Large Databases (VLDB'91)*, Barcelona, Spain, September 1991.

[SM92] S. Shlaer and S.J. Mellor. *Object Lifecycles Systems Analysis: Modeling the World in State.* Yourdon Press, Prentice Hall, 1992.

[Som91] I. Sommerville. *Software Engineering.* Addison-Wesley, 1991.

[SSE87] A. Sernadas, C. Sernadas, and H. D. Ehrich. Object-Oriented Specification of Databases: An Algebraic Approach. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB'87)*, pages 107–116, Brighton,UK, August 1987.

[TPC94a] M. Teisseire, P. Poncelet, and R. Cicchetti. Dynamic Modelling with Events. In *Proceedings of the 6th International Conference on Advanced Information Systems Engineering (CAiSE'94)*, volume 811 of *Lecture Notes in Computer Science*, pages 186–199, Utrecht, The NetherLands, June 1994. Springer Verlag.

[TPC94b] M. Teisseire, P. Poncelet, and R. Cicchetti. Towards Event-Driven Modelling for Database Design. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB'94)*, Santiago, Chile, September 1994.