The Story of the IDEA Methodology*

Stefano Ceri and Piero Fraternali

Dipartimento di Elettronica e Informazione, Politecnico di Milano
Piazza Leonardo da Vinci 32, I-20133 Milano, Italy
ceri/fraterna @elet.polimi.it

Abstract. The IDEA Methodology is a comprehensive approach to the design of information systems; its main focus is the use of innovative features of database technology, in particular the object-oriented and rule-based paradigms.
The development of the IDEA Methodology started three years ago, in the framework of the IDEA Esprit project. The methodology was first conceived in our university, then experienced by several industrial partners of the IDEA Consortium, then consolidated. In the summer of 1997, the IDEA Methodology is widely available through a variety of media and sufficiently stable to permit us to attempt a critical evaluation of this three-years story. Of course, this is a preliminary evaluation: the methodology will have its most significant assessment after its dissemination and usage, which is currently taking place.
In order to set the context for the discussion, the paper introduces the main features of the IDEA Methodology through a case study.

1 Introduction and Case Study

As most of the object-oriented methodologies, the IDEA Methodology includes the three classical phases of *Analysis*, *Design*, and *Implementation*. In addition, it includes *Prototyping* as an intermediate phase, placed between design and implementation, and dedicated to the assessment of the output of design. To introduce the features of the IDEA Methodology, we present a small-scale case study, called the *Job-shop Management System*, through a description of its requirements and of selected methodological documents.

1.1 Requirements

The *Job-shop Management System* (*Job-shop*, for short) is a simplified version of an industrial application developed by one of the partners of the IDEA Consortium. For the purpose of illustrating the IDEA Methodology, we will consider the following requirements:

The job-shop is made of a set of working centers, which are either mills or lathes. The shop processes parts, which must be picked up from the inventory, turned, milled, and deposited back to the inventory. Parts are moved around the

* Research presented in this paper is supported by Esprit project P6333 IDEA

shop by shuttles. A shuttle may be free, or occupied carrying a part. The shop foreman may create a new working order, related to a specific part. Processing the order requires picking up, turning, milling, and depositing the associated part. To start an order there must be a free shuttle and the new order must not exceed the maximum number of pending orders, which is set to 1.5 times the number of working centers. A shuttle is assigned to an order for the entire duration of the working process and waits for the order's part to finish each working phase. A shop labourer may declare a working phase of an order (pick up, turning, milling, deposit) completed. The completion of a phase makes the part available for the next operation.

1.2 Analysis Documents

Analysis is devoted to the collection and specification of requirements at the conceptual level. This phase aims at a natural and easy-to-understand representation of the "universe of discourse", and uses conceptual models with an associated graphical representation which are well established in the software engineering practice, such as the Object Model [7] (a variation of the well-known Entity-Relationship model) and Statecharts, a widely used visual language for specifying finite state systems [6].

Figure 1 shows the Object Model of the Job-shop case study, which contains six classes (*part, order, workingCenter, lathe, mill, shuttle*), one inheritance hierarchy between class *workingCenter* and classes *lathe* and *mill*, two relationships (*ord-part, transport*), and one integrity constraint (*capacity*) limiting the number of pending orders to avoid overflowing the job-shop capacity. Operations (or methods, in the object oriented terminology) are specified as arrows pointing to a class box, annotated by the operation's name and parameters.

The black triangle on the inheritance hierarchy denotes a total and exclusive hierarchy (mills and lathes constitute a partition of the set of workcenters), and the annotation STATIC DEFERRED in the constraint oval dictates the evaluation semantics of the constraint, which is a predicate on a single database state (STATIC) and must be checked only at the end of the transaction (DEFERRED).

The dynamic behavior of the classes described in the Object Model is detailed by the Dynamic Model, which includes the statechart illustrated in Figure 1.2. Objects of class *order* may be in six different states (*pending, pickup-op, lathe-op, mill-op, deposit-op*, and *completed*), which correspond to the various phases of an order's production. Some states (*pickup-op, lathe-op, mill-op, deposit-op*) are further decomposed into two OR-substates, to show that the respective phase may be in process or terminated. The statechart also specifis the default state for new orders (*pending*), and the transitions that make an order progress in the production workcycle upon the occurrence of production events (for example, event *completePhase* signals the completion of a working or transport operation).

The Application Model collects a number of *application sheets*, one of which is shown in Figure 1.2. The sheet describes application *Production Progress Management* by illustrating its data dependencies, preconditions, postconditions, and

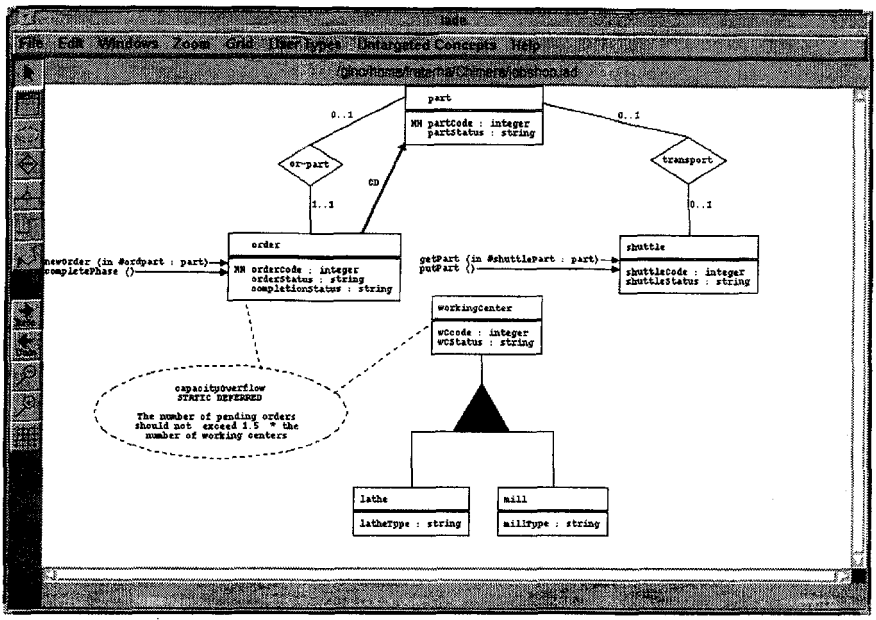

Fig. 1. The Object Model of the Job-Shop Case Study

produced events, without including the algorithmic details, which are not the focus of the IDEA Methodology and can be specified with the techniques offered by existing object-oriented methodologies.

1.3 Design Documents

Design is the process of translating requirements expressed by the Object Model, Dynamic Model, and Application Model into documents written in Chimera that provide a precise, unambiguous specification of the application. The process is divided into schema design, concerned mostly with types, classes, relationships, and operations; and on rule design, further subdivided into deductive rule design and active rule design.

We start by showing in Figure 4 the Chimera code corresponding to the Object Model of Figure 1. Each class of the Object Model is mapped into a Chimera class; the inheritance hierarchy is represented by the *superclasses* clause in the definition of class *lathe* and *mill*; relationships are transformed into object-valued attributes in the involved classes (e.g., the relationship *transport* associating a part and its shuttle yields an attribute *shuttlePart* in class *shuttle*); the signatures of operations are introduced in the class definitions. Note that additional attributes have been added (for example *orderStatus* and *completionStatus* in class *order*) to provide a design structure to the dynamic states of classes described in the statecharts.

The *capacity* constraint in the Object Model is mapped into a Chimera untargeted constraint, i.e., a view with an implicit meaning: if the view is non-empty

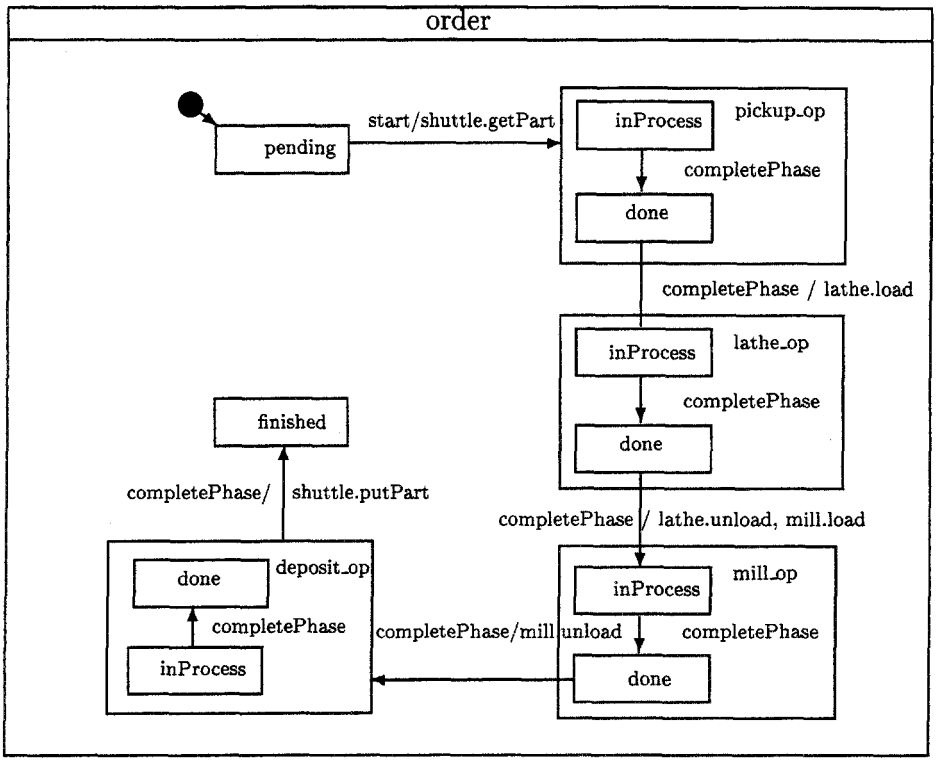

Fig. 2. The Dynamic Model of the Job-Shop Case Study

Application:	Production Progress Management
Description:	Declares a production phase completed.
Reads:	The order that has completed the phase.
Changes:	The order's status, the status of the resources working the order, and the allocation of a part to a resource.
Sends:	Event *completePhase* is raised
Assumes:	The order is not finished and its current working phase is in progress.
Result:	The current working phase is declared terminated.

Fig. 3. Examples of Application Sheet

```
define object class part
attributes partCode: integer,
           partOrder: order,
           partStatus: string,
           partWC: workingCenter
end;

define object class order
attributes orderCode: integer,
           orderPart: part,
           orderStatus: string,
           completionStatus: string
operations newOrder(in #ordpart:part),
           completePhase()
end;

define object class shuttle
attributes shuttleCode: integer,
           shuttleStatus: string,
           shuttlePart: part
operations putPart()
           getPart(in #shuttlePart:part),
end;

define object class workingCenter
attributes wCcode: integer,
           wCstatus: string
end;

define object class lathe
superclasses workingCenter
attributes latheCode: integer
end;

define object class mill
superclasses workingCenter
attributes millCode: integer
end;
```

Fig. 4. The Chimera schema of the Job-shop case study

in some database state, then that state violates integrity and the view tuples identify the objects responsible of the violation. The definition of constraint *capacity* is an example of a Chimera deductive rule:

```
define deferred constraint capacity(orderNum:integer,centerNum:integer)
 capacity([I,L])<-integer(I),I=card(W where workingCenter(W)),
                  integer(L),L=card(O where order(O),
                                    O.status=''pending''),
                  L>1.5*I
end;
```

The constraint consists of a signature and an implementation: the signature defines the constraint type as a record with two integer fields, and the constraint evaluation mode as deferred; the implementation is a deductive rule, whose head introduces a predicate with the same signature as the constraint, and whose body is a declarative formula binding the logical variables in the head to the number of orders and working centers that produce the violation. The constraint predicate can be used in the course of the transaction to retrieve constraint violations and possibly repair them, e.g., by deleting an appropriate number of pending orders; if violations remain at transaction commit, the transaction is rolled back.

Active rule design is mainly concerned with developing Chimera triggers implementing the busisness rules that lay behind the applications documented by the Application Sheets. For example, the *Production Progress Management* ap-

plication admits several business rules which govern the evolution of the production process after an operation is completed. We illustrate a rule implementing a *pull* production policy, where the availability of a lathe *" pulls"* the next order to be produced, selected as the one having the oldest order-code, and starts its production on a lathe. A similar rule manages the completion of a milling operation.

```
when a lathe changes its availablity status
if   it becomes free and there are pending orders
then assign the earliest order to the lathe and
     update the status of the lathe, order, and part.
```

More sophisticated policies can be added to tackle urgent orders, failures of various resources, and so on. We now show how the business rule for lathe assignement can be written in Chimera.

```
define immediate trigger assignLathe for lathe
 events     modify(wCstatus)
 condition occurred (modify(wCstatus), Self),Self.wCstatus= "free",
           order(O), part(P), P=O.orderPart,
           O.orderCode=min(PO.orderCode where order(PO),
                                  PO.orderStatus="pickup_op",
                                  PO.completionStatus= "done",
                                  PO.orderPart.partStatus= "raw" )
 actions    modify (lathe.wCstatus, Self, "working"),
           modify (order.orderStatus, O, "lathe_op"),
           modify (order.completionStatus, O, "in_process"),
           modify (part.partWC, P, Self)
end;
```

The trigger is defined as immediate, which means that it is considered for execution during the course of the transaction. The event part lists all the events which may trigger the rule (in this case, the modification of an existing lathe's working status). The condition contains a declarative formula which must be satisfied for the rule to be executable; conditions are limited to conjunctions of atomic formulas, positive or negated. The first subformula in the condition of rule *assignLathe* is a so called *event formula*, built from the special predicate *occurred*, which binds to the logical variable *Self* ranging over class *lathe* only those objects which have been modified since the last rule execution; next the condition tests that the modified lathe is free, retrieves and binds to the logical variable O the order with the minimum value of *orderCode* that is in the *pickup_op* status, has completed the transport phase, and is associated to a raw part, and binds the part associated to that order to variable P. Variables Self, O, and P are then used in the action in order to modify the status of the lathe, modify the status attributes of order O, and link part P to the lathe that is working it.

1.4 Prototyping Documents

Rapid Prototyping is the process of testing, at a conceptual level, the adequacy of design results with respect to the actual user needs. A variety of formal

transformation methods can be applied to improve the quality of the design, verify its formal properties, or transform design specifications into equivalent specifications which exhibit different features. The IDEA tools, discussed in Section 5, assist the automatic generation and analysis of active rules, and enable the prototyping of applications written in Chimera.

In particular, we focus on termination analysis of active rules, performed by a tool named *Arachne* (Active Rule Analyzer for Chimera). Arachne builds the triggering graph of a rule set, that is a graph representing rules as nodes, and potential rule interactions as arcs: there is an arc from rule A to rule B if A's execution can trigger B. A cycle in the triggering graph denotes the risk of nontermination. Two levels of analysis are supported: *syntactic*, where only the event and action part of rules are considered, and *semantic*, where greater accuracy is obtained by considering also rule conditions.

Termination analysis, performed on the case study with Arachne, produces the following results: 21 cycles are discovered by syntactic analysis, of which only three are retained after semantic analysis. By focusing on these, termination is easily checked manually. Figure 5 shows the remaining cycles after semantic analysis.

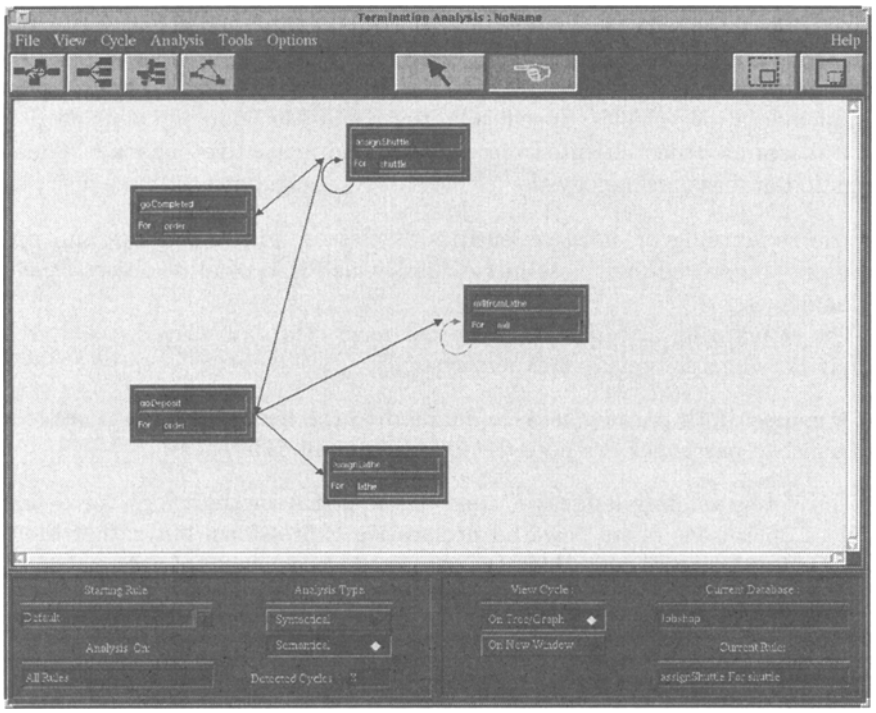

Fig. 5. The Triggering Graph after Semantic Rule Analysis

1.5 Implementation Documents

Implementation is the process of mapping conceptual specifications into schemas, objects, and rules of existing database platforms; the process is influenced by the features of the specific target environments. To exemplify the needed implementation techniques, the IDEA methodology uses five different systems: **Oracle, Illustra**, and **DB2**, three classic *relational* products supporting triggers; **ODE**, an *object-oriented* database available on the Internet to universities and research institutes; and **Validity**, the first *deductive and object-oriented* database system that will be soon brought to the market. The mapping to Oracle is presently assisted by *Pandora*, a tool (described in Section 5), producing Oracle 7 code from Chimera.

After this short presentation of the Idea Methodology, we turn to the critical assessment of its applicability.

2 Suitability of applications

The first discussion concerns the identification of the class of applications to which the IDEA Methodology should be applied. IDEA is an object oriented approach (one of the second-generation object-oriented methodologies) but its applicability is restricted to data-intensive applications, i.e., applications in which data management is the main focus and database management systems are used as implementation vehicles. Specifically, the IDEA Methodology aims at giving to databases an object-oriented organization and a reactive behavior. Thus, it brings to database technology the richness of two neighbour software discipline:

- The structuring of database entities as classes, with attributes and operations, organized into class hierarchies, which is typical of object-oriented languages.
- The structuring of computation in the form of a data-driven, reactive behavior, which is typical of expert systems.

Of course, both paradigms were adapted to the needs of data-intensive applications. In particular, we note the following main differences:

- The "programming language" used in the IDEA methodology for defining class operations offers powerful declarative expressions, but rather limited procedural expressions, which are restricted to sequences of actions (no control statements).
- The "reactive knowledge" supported by the IDEA Methodology is rather simple, especially for what concerns knowledge acquisition from experts and explanation of the reactive behavior.

These limitations are quite acceptable in the context of the IDEA Methodology; in particular, absence of control structures in the programming language eases the mapping to database query languages. Operations of objects are normally

rather simple (they are mainly constructors, accessors, and transformers), while more complex computations are encoded within applications.

For what concerns rules, the absence of an explanation facility is consistent with current limitations of database technology, where active rules are executed together with applications and cannot be easily traced. We expect that active rules are not too many and not too strongly interacting, while an expert system can easily grow to hundreds or thousands of rules. Obviously, we expect instead that the size of the database is significant, and that requirements such as reliability, availability, and efficiency of access are dominant factors.

3 Suitability of models and targets

In this section, we critically revise our choices about the models of the IDEA Methodology, distinguishing the analysis phase and the design phase. We also present the rationale for the choice of exemplifying implementation through certain target systems.

3.1 Analysis Models

The main choice is the *lack* of features: object interaction diagrams à la Booch Method, and informal specifications of reactive behavior were considered but eventually have been omitted from the set of analysis models.

Object interaction diagrams have been omitted because we expect that analysts use the IDEA Methodology to give a high level specification of the dependencies and interactions of their applications with the database objects, sufficient to design the shared objects and knowledge. For the specification of the computational aspects of an application (for example, the user interface or complex elaborations), the IDEA Methodology may be integrated to "traditional" object-oriented methodologies, whose models can be used to specify the interaction between main memory application objects and database shared objects.

In retrospective, the weakest choice in the analysis models concerns the lack of a model for specifying general business rules in an informal way during analysis. After some experience, we verified that structured-text annotations similar to those introduced in Section1.3 are a valuable addition to the analysis models and a good basis for writing business rules in Chimera.

3.2 Design Model

Chimera is a complex language, which includes selected features from deductive languages (declarative expressions), object-oriented languages (inheritance, methods, path expressions), and procedural languages (data manipulation primitives, sequence iterator). It has also a number of innovative features, like, the uniform binding passing mechanisms from conditions to actions of operations, active rules, and transactions.

The design of Chimera was mainly inspired by our desire of *expressing a large amount of knowledge* in the database, instead of keeping such knowledge procedurally encoded within applications. This feature can be regarded as an additional level of independence for applications, called **knowledge independence**, achieved by factoring knowledge out of the applications and expressing it in the schema, in particular within objects and rules. We recall that traditionally databases provide physical independence (from the actual storage implementation) and logical independence (from the structure of the schema).

The design language had to respond to other requirements: we wanted it to be *formal, executable, easy to use* for the designer, and *easy to map* into concrete program and data structures.

In the following, we analyze several technical features of the language and discuss our most controversial choices.

- We originally included into the language *value types* which, similarly to object types, could be organized into hierarchies and have cardinality constraints. Then, we simplified this notion which users found difficult to distinguish from that of object class.
- Similarly, we initially allowed constraint and view predicates to be typed arbitrarily, and then constrained them to be records of atomic types, because references to parameters of arbitrary structure in views or constraints are syntactically quite difficult.
- We originally included class properties (attributes, operations, and constraints); for instance, we could model the cardinality of a class as a class attribute, and have an accessor operation for this piece of information, as well as a constraint on the maximum number of instances of the class. Then we decided to remove them, because users were confused by expressions mixing variables ranging on class attributes and "regular" attributes, and also because integrity constraints referring to class properties could be easily represented using the aggregate operators of the Chimera language.

While we introduced several simplifications in the object-oriented model, we retained a good deal of complexity in the active rule language, which is the most innovative feature of Chimera. In retrospective, not all the advanced features of triggers were equally understood and used by designers:

- The most difficult feature is the difference between *event-consuming* and *event-preserving* rules. In the former case, each "event" (e.g., the update to an object) is processed by each rule only once, at the first consideration of the rule after the event's occurrence. In the latter case, all occurred events are seen by a rule at each consideration. The consumption mode also influence the semantics of the *old* function, used to evaluate terms on past database states. We observed that the second option is very rarely required and typically remains unused; luckily, the first option is also the default one, but in retrospective we could simplify the language and omit event-preserving triggers.

- Another advanced feature of the language is the use of *event predicates*, by means of which the objects affected by an event are selected in the condition and used in the action part of the rule. There are two variants of an event predicate: with the first one, an object is selected regardless of its evolution after the event; with the latter, called *net-effect* variant, the object is selected only if the event affecting it is not invalidated by a subsequent event (e.g., the object is created and not subsequently deleted). We experienced that the use of event predicates is quite natural, however the distinction between the two cases is most often neglected by users, who always use one version. Moreover, net effect is not easily supported by commercial implementation targets, so this feature, although useful, could be simplified as well.
- A final consideration concerns the execution semantics of rules and operations in Chimera, which is inherently set-oriented, in spite of the prevalence of tuple-oriented semantics in commercial systems. We found that the set-oriented execution of Chimera triggers and the notion of rule action atomicity, whereby cascading rule triggering is enabled only at the end of action execution, greatly enhances the understandability of large rule sets, because rule interactions are more apparent and the designer can reason on the effect of a rule execution without worring about side-effects and nondeterminism caused by recursive activation of other rules *during* the processing of the action.

3.3 Selection of the Implementation Target

In the implementation part, the IDEA Methodology is quite different from classical database design methodologies, as it goes deep in the mapping from design documents down to specific database systems. The choice of mapping to systems rather than to more abstract descriptions, such as the current standards, was quite controversial. For instance, in the methodology of [3], the mapping process included a model-independent logical design followed by a model-specific (but not product-specific) logical design, tailored to the relational, hierarchical, and network models. Our decision was imposed either by the deficiencies of the standards in the description of advanced database features (e.g., of SQL3 for active rules), or by the discrepancies between the standards and the products (e.g., between ODL and OQL and most object oriented database products).

Another trade-off concerned the selection of the products and prototypes to use; we were influenced by the desire to cover a variety of different concrete situations with as few mapping schemes as possible. We selected 3 relational systems: Oracle, Illustra, and DB2. Oracle and DB2 represent rather classical systems and thus demonstrate that the IDEA technology can be implemented on the most established platforms, while Illustra is particularly strong in the support of objects and rules. Next we chose one object-oriented system (ODE), which is a research prototype available on the Internet to academic and research-oriented institutions. Finally we included Validity, the first industrial-strenght product featuring deductive and object-oriented technology, developed by BULL, the main contractor of the IDEA Project.

On the positive side of our choice, the mapping to products is more valuable to users than the mapping to standards or to abstract models: users find guidelines which are sufficiently precise to let them produce DDL and DML code that can be executed by the selected database kernels. On the negative side, the IDEA Methodology becomes bound not only to products, but also to their versions as of the beginning of 1997, and thus will need a constant revision.

4 Suitability of the process

We next discuss the process of designing applications with the IDEA Methodology, and the main alternatives we had to face.

4.1 Analysis

The most important task of analysis is the production of the Object Model, which is a classical Entity-Relationship diagram, with a greater emphasis on the specification of the declarative properties of the application domain, expressed as integrity constraints.

As already mentioned, the IDEA Methodology introduces reactive properties only during design. The rationale for this choice is that rules express "procedural knowledge" (reactions) and as such they should be produced by starting from higher level specifications, like declarative integrity constraints, or statecharts. In this way, the analyst is free to concentrate on semantics, rather than being forced to consider the intricacies of reactive behavior.

This approach is certainly adequate for the so-called *internal* and *extended* applications of active rules, that is, those situations where active rules are used as implementation devices for a functionality that should be supported by an advanced DBMS (e.g., integrity), or by a DBMS extended to support a specific database-intensive application (e.g., workflow management).

However, in this way rules expressing the business policies of the enterprise remain unexpressed for the entire duration of analysis, and are postponed until design. In retrospective, this choice increases the risk of overlooking or "forgetting" important business rules, which developers may later be forced to encode into applications, thus losing the benefits of knowledge independence. To reduce this danger, we suggest a more flexible development process where business rules are gathered during application or business process analysis.

4.2 Design

The IDEA Methodology approaches schema design first, followed by deductive rule design, and by active rule design. The rationale for this sequence is to give priority to the structural and behavioral knowledge associated to objects and classes; next, declarative knowledge is defined by means of deductive rules; finally, procedural knowledge is defined by means of active rules.

Schema design is a sequence of transformations that progressively consider the features of the Object Model and express them by means of Chimera. The main schema design activity is the transformation of relationships into object-valued attributes that serve as references between objects. This transformation is required because Chimera does not support relationships explicitly. In object-oriented models, relationships are turned into object-valued attributes; in the relational model, relationships are modeled as pairs of attributes on the same domain, enabling join operations connecting the related tables. Our solution meets the requirements of object-oriented models and can be considered as a first step towards a relational solution; however, it also forces the designer to a decision which may influence and constrain the implementation, and as such it was criticised by some of our users.

Deductive rule design is driven by the specifications collected during analysis, i.e., fixed-format constraints and annotations for data derivations and generic integrity constraints. During design, we suggest that the designer express all the available declarative knowledge; when the design is consolidated, prior to implementation, data derivations and integrity constraints can be reviewed to discard features that are redundant or that are too costly to implement.

During active rule design, three sources of specifications are considered for active rules: integrity constraints, business process specifications, and the dynamic behavior of objects.

In the IDEA process, active rules for integrity maintenance are considered an evolution of deductive rules for specifying database integrity, in which the designer adds a procedural description of the actions that repair an integrity violation. Reactive integrity maintenance enables the correction of erroneous actions performed by transactions; however, the declarative nature of the constraint specification is lost. The iter from declarative to precedural integrity maintenance has been initially criticised by IDEA users, who found it redundant and possibly confusing. However, for sizeable applications and non trivial integrity constraints, it soon became clear that a declarative specification of integrity constraints is a necessary prerequisite for the subsequent formulation of reactive maintenance rules, since declarative constraints are easier to manage and understand, and provide a clear roadmap to identify all the maintenance rules needed to enforce integrity.

Business rule design is surely the feature of the IDEA Methodology that was accepted with most favor by IDEA users. Without business rules, "enterprise policies" must be encoded within applications, with the risk of replications and inconsistencies; this increases during system evolution because business rules change quite rapidly. We believe that this positive result is mostly due to the "conceptual level" in which active rule design occurs in the IDEA process; although designers must implement their applications on relational platforms supporting triggers, they do not use them to *specify* business rules.

The transformation of statechart into a design structure was a less problematic issue. We proposed two alternative strategies for mapping statecharts, one more "object-oriented", the other more "rule based". We observed that most

often designers started with the former option, in which state transitions are mapped into class operations, and migrated to the latter, in which the finite state machine is implemented by triggers, when they got more acquainted with active rules. In any case, the presence of a standard method for mapping an event-driven model into objects and rules in the database was generally considered an improvement with respect to implementing ad hoc solutions in the application programs.

4.3 Prototyping

Prototyping is a methodological phase in which design results are implemented on a small scale, typically with rapid prototyping software, and their adequacy and conformity with respect to requirements are evaluated by designers and by users. For this purpose, we provide a set of tools for rapid prototyping, described in Section 5. In the IDEA Methodology, prototyping has an additional unconventional meaning: we want to analize complex rule sets in order to assess their interaction. This task is the most challenging, since in rule-based systems the difficult problem is not designing rules which individually behave well, but designing collections of rules which alltogether have a correct behavior and that can be easily debugged, maintained, and evolved. This aspect of prototyping is called *knowledge design in the large.*

Prototyping of deductive rules is focused on achieving the two properties of *stratification* and *satisfiability.* Stratification is concerned with determining the evaluation order of deductive rules in the presence of negation and sets; satisfiability is concerned with the absence of contradictory integrity specifications.

Similarly, prototyping of active rules is focused on achieving the two properties of *termination* and *confluence.* The former guarantees the termination of rule processing after any input transaction, and the latter guarantees that the final state computed by active rules is the same regardless of their evaluation order. We consider termination as the most important property, and provide analysis and rule modularization techniques which enable a designer to decompose a large rule set and prove its termination.

Our experience with rule prototyping has shown that this phase is essential for acquiring greater familiarity with the rule based style of computation. Playing with rules on small scale examples, and reasoning on collective rule behavior in an organized way are fundamental activities which give the necessary insight to designers on the behavior of their rules.

4.4 Implementation

Although the IDEA book addresses implementation on five distinct database systems, the mapping processes have certain common features, because in all cases we need to map the schema, integrity constraints, operations, deductive, and active rules.

Schema mapping depends on whether the target supports generalization hierarchies; if not two mapping schemes are possible (called the *horizontal* and *vertical* mapping), which implement generalization hierarchies by means of several, related tables. Fixed-format integrity constraints typically map into integrity checking clauses available in most relational systems and in some nonrelational system. Some relational systems and most object-oriented systems do not support referential integrity and generic integrity constraints, and in such case we model these features by means of active rules.

Declarative expressions are typically well supported by relational query languages, and conversely procedural operations are well supported by object-oriented methods.

Deductive rules can be mapped easily only in those systems (DB2, partially, and Validity) which support them directly; in other cases, relational views and object-oriented methods can be used to provide an acceptable approximation. Alternatively, materialization of derived attributes and views can be supported by active rules which are syntactically derived from deductive rules; this translation is a platform-independent technique, especially valuable for systems strong in active rule support and weak in deductive rule support.

The mapping of active rules is the most difficult and problematic, because of the operational nature of active rule semantics, which is quite different in each system, and also because of the heavy limitations that each product imposes with respect to Chimera active rules. In the end, two approaches have emerged.

- With *meta-triggering*, we use the native active rule engine in order to detect events, but then extend it to support the semantics of conceptual rules; in this way, we can reproduce all the conceptual features of Chimera triggers. Typically, we program the extensions to the active engine by using stored procedures and imperative language attachments; this solution is application-independent and fully defined in the IDEA Methodology.
- With *macro-triggering*, we use instead the native rule engine available on each target system; conceptual triggers are possibly aggregated to constitute macro-triggers which respect priorities defined in Chimera; these aggregates in some situations do not have the same semantics as the original Chimera active rules, but are simpler to obtain.

Vertical schema mapping, operation mapping, deductive rule mapping or their transformation into active rules, meta-triggering and macro-triggering constitute a collection of techniques which, although defined on our five representative targets, can be adapted for reuse on any other database platform.

5 Tools

In our experience with IDEA users we observed that the lack of tools for rule generation, analysis, and run-time monitoring is the main obstacle to the widespreading of active database applications.

At the end of the IDEA Project, an integrated tool environment is available, which supports the various phases of the IDEA Methodology, provided by Politecnico di Milano and Bonn University; we focus on the management of active rules, the specific research interest of Politecnico within the IDEA Project. The IDEA environment comprises tools for schema design, active rule generation, analysis, prototyping, debugging, browsing, and for mapping Chimera into Oracle.

Iade is a classical CASE Tool used during analysis, enabling the graphic editing and annotation of Object Model diagrams, which are semi-automatically mapped into schema declarations, constraints, and triggers written in Chimera.

Arachne supports the compile-time termination analysis of a set of Chimera triggers, determining the potential causes of infinite executions due to the mutual interaction of active rules. Arachne supports also rule modularization and termination analysis at the module level.

Algres Testbed is an execution environment which permits the rapid prototyping of the design specifications. It is based on Algres, a system developed at Politecnico di Milano which provides complex objects and an extended relational algebra. The testbed supports the full Chimera language (with the exception of deductive rules) and can be used for monitoring the execution of transactions and active rules.

Pandora automatically translates Chimera applications into Oracle, by exploiting the mapping techniques described in Section 4.4, and in particular by using meta-triggering to implement Chimera triggers.

Building these tools gave us the opportunity of getting a deep insight into both the theoretic and technological aspects of rule based technology, with positive feedbacks on the methodology itself. From the making of Arachne we learnt the importance of mastering rule interactions at design time, which inspired the notion of active rule stratification, which is at the basis of the IDEA approach to active rule design in the large. Two different implementations of Chimera (the first tightly coupled to the Algres systems, the second loosely coupled to Oracle) challenged our language design choices and inspired general-purpose, target-independent implementation techniques, like meta-triggering, that can be used to extend the capabilities of any existing active database system.

At present, the effort to support the IDEA Methodology has taken a novel direction: we are porting the IDEA Tools on the Internet, to allow perspective users to directly experiment the IDEA Methodology by creating their own applications, from the editing of Object Model diagrams to the production of the final Oracle code. To this end, Iade, Arachne, Pandora, and the Algres Testbed are being reimplemented in Java and will available in the so-called *IDEA Web Lab* by April 1997.

6 Outlook and Conclusions

In this paper, we have discussed a number of issues and problems that have characterized the birth and development of the IDEA Methodology. Some solu-

tions to these problems appear satisfactory, while other are controversial. With the full dissemination of the IDEA Methodology, fostered by the availability of a number of IDEA-related resources on the Internet, we expect to collect more substantial answers and feedbacks.

References

1. E. Baralis, S. Ceri, and S. Paraboschi. Improved rule analysis by means of triggering and activation graphs. In T. Sellis, editor, *Proc. of the Second Workshop on Rules in Databases Systems*, LNCS 985, pages 165–181, Athens, Greece, Sept. 1995.
2. E. Baralis, S. Ceri, and S. Paraboschi. Modularization techniques for active rules design. *ACM Transactions on Database Systems*, March 1996. (to appear).
3. C. Batini, S. Ceri, and S. B. Navathe. *Conceptual Database Design, an Entity-Relationship Approach.* Benjamin Cummings, 1993.
4. S. Ceri and P. Fraternali. *Designing Applications with Objects and Rules: the IDEA Methodology.* Addison Wesley Longman, Great Britain, 1997.
5. S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Active rule management in Chimera. In J. Widom and S. Ceri, editors, *Active Database Systems.* Morgan-Kaufmann, San Mateo, California, 1995.
6. D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
7. J. Rambaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.