Tools for Chimera: An Environment for Designing and Prototyping Advanced Applications in an Active DOOD Model

Ulrike Griefahn, Thomas Lemke, Rainer Manthey

Department of Computer Science III
University of Bonn
Römerstraße 164, D-53117 Bonn, Germany
e-mail: idea@informatik.uni-bonn.de

Abstract

During the last five years, the object-oriented data model Chimera has served as a focus of numerous research activities within the European cooperation project IDEA. Chimera particularly emphasizes the use of deductive and active rules during object-oriented modeling. Apart from a full application design methodology based on Chimera, a number of tools supporting the different phases of this methodology have been developed by the IDEA partners. In this paper, the tools developed at the University of Bonn will be introduced. They support schema design, prototyping and evolution as well as deductive rule analysis and automatic generation of triggers for analyzing updates of derived data.

1 Introduction

Chimera [5, 6] is a novel database model and language combining active and deductive database concepts in a coherent object-oriented context. Till now, there are only very few attempts which aim at combining both rule paradigms with an object-oriented data model. At present, only the Rock&Roll language [1] goes into the same direction similarly providing such a hybrid database language. However, there are numerous approaches to integrating an object model with the one or the other form of rule individually. DOOD systems introduce derived concepts – such as derived classes, virtual objects, or derived attributes – into an object-oriented context. Active rules can be used to provide objects with a limited degree of autonomous behaviour. Thus Chimera has been called an active DOOD model.

The motivation to invest effort in the integration of the three paradigms comes from the conviction that this particular combination of functionality presents a kind of “critical mass” for the development of more “intelligent” database technology. Whether this hypothesis will turn out to be valid remains to be proved, of course. However, a first step towards this kind of systems has been made by a group of European database researchers who have been cooperating over five years in the IDEA project, sponsored by the ESPRIT program of the European Union. The acronym IDEA stands for “Intelligent Database Environment for Advanced Applications”. Chimera has been designed during the course of this research project as a joint linguistic focus to be supported by the various systems developed by the individual partners.

The main results of the project are a comprehensive methodology, two sets of tools supporting the various phases of this methodology, and a collection of case studies performed by the industrial partners. “The IDEA Methodology” [3] provides a framework for analyzing, designing, and prototyping database applications with objects and rules. The tool environments have been developed by teams at the Politecnico di Milano and at the University of Bonn, respectively. Whereas the POLI team has been concentrating on aspects of active rule management, our team at Bonn focussed more on deductive rules and on change management in an OO setting. Although the two tool sets have not been integrated into a uniform system supporting the IDEA methodology in a coherent manner, the various phases of application development are well covered by tools from the one or the other group. It is the purpose of this paper to provide an overview of the goals and the main characteristics of those tools which have been developed at Bonn.

At the core of this tool environment there is an implementation of the full Chimera model and language on top of a relational back-end (a non-commercial, academic prototype DBMS, programmed in persistent Prolog and supporting deduction and triggers as well [9]). This implementation is not intended to serve as an efficient DBMS for managing large databases, but as a prototyping environment for Chimera applications. For this reason it has been called the Chimera Prototyping Tool (CPT). In this context, the notion “prototyping” has been used in the technical sense suggested by the IDEA methodology, where it follows the analysis and design phase. During analysis, a requirements specification expressed by means of an object-oriented static and a statechart-based dynamic meta-model are compiled and mapped to a formal representation in Chimera. Various design steps are then applied in order to transform the initial Chimera schema into a more consistent, reliable, and adequate form. In the prototyping phase, properties of this schema are further investigated – possibly leading to several other rounds of revision and upgrade – by employing small, but meaningful sets of test data to which in particular the rules of the original schema can be applied and validated. The methodology then suggests to map the final version of the resulting schema to one of many possible commercial object-oriented or relational platforms for implementation purposes.
CPT comes with an advanced Graphical User Interface (CPT/GUI) for the design of Chimera schemas. Thus, CPT does not only support the prototyping phase, but also the design phase of the IDEA methodology. Whereas the methodology starts schema design from a separate application representation in the meta-model mentioned, CPT skips this phase and directly enables designers to define applications in Chimera. Due to the various forms of deductive rules offered by Chimera (supporting class and attribute derivation, interclass views, and integrity constraints) and the rather powerful, set oriented trigger facilities, the design of a Chimera schema is a far more complex task than the pure design of a class hierarchy. In particular, schema design in such a context may last over a considerable period of time, thus requiring special support for incremental schema development. Various techniques developed for the graphical manipulation of this kind of advanced database schemas have been developed and integrated into CPT/GUI. Independent from this aspect, the solutions developed for CPT in order to map object-oriented Chimera schemas to a relational platform deserve some attention. Particularly the use of relational rules on the back-end side suggests a number of new and original techniques. CPT, its graphical interface, and its implementation will be discussed in Section 3.

The other tools developed in our group are closely integrated with CPT as far as the style of interaction (design of interface, ease of use) is concerned. There are two tools devoted to the analysis of the deductive rules contained in a Chimera schema. In Chimera terminology, deductive rules are also called "passive rules" in order to contrast them with active rules. The Passive Rule Design Tool (PRDT) offers a selection of services for checking various desirable properties of deductive rule sets (such as safeness and stratifiability), explaining the reasons for potential violations of these properties (which are often far less obvious in an object-oriented model than in a relational one) and to suggest concrete steps for "repairing" such design mistakes. PRDT is discussed in Section 4. The second tool is called Propagation Rule Compiler (PROP). It supports schema designers in deriving active rules which are able to compute consequences of base data updates on derived data. At least for incremental adaptation of materialized classes or attributes as well as for incremental integrity checking it is important to be able to compute such induced changes efficiently. Designing propagation triggers by hand is a hard and error-prone affair, such that tool support for this phase of schema design is strongly recommended. PROP is presented in Section 5.

Last not least, there is a tool which supports designers of complex Chimera applications during schema evolution. This Schema Evolution Assistant (SEA) makes use of the same graphical means for evolving existing schemas (combined with populated databases) as are available for incrementally designing schemas from scratch. To do so is quite obvious in principle as a stepwise schema design process can be viewed as a sequence of schema evolution steps applied to an empty database. Whereas problems related purely to the management of meta-data are the same during design and evolution, the problems arising from the need to adapt existing data to the new schema structure are the real hard aspects of schema evolution. In a complex model like Chimera, a dedicated tool for checking and resolving conflicts, explaining choices and offering alternatives is even more necessary than in the case of more conventional data models. SEA provides a first step into this direction. It is discussed in Section 6. Figure 1 presents an overview of the architecture of the tool environment.

Figure 1: Architecture of the Chimera tool environment

Up till now, the tools addressed here have been mainly documented in a number of technical reports written during the course of the IDEA project. The purpose of the present paper is to introduce the tool environment as a whole and to discuss the rationale behind our approach. Publications focusing on the individual tools or on special issues related to these tools will follow. The source code of the tools (implemented in Eclipse Prolog and in Tcl/Tk) as well as the reports mentioned are available over the WWW from http://ww.cs.uni-bonn.de/~idea.

2 Chimera: An Active, Deductive, Object-oriented Data Model

Chimera is the name of a famous mythical creature composed from a lion, a goat and a serpent. The choice of this name for the common model and language of the activities in the IDEA project has been motivated by the emphasis on the integration of concepts from three of the most active and productive areas of current research, namely active, deductive and object-oriented databases. When designing Chimera, the goal was not to invent new, original features, but to properly select important and well-established concepts and to blend them in a seamless manner into a unified whole. During the course of the project, Chimera has been used by many different individuals in industry and academia both as an implementation target as well as for coding a wide variety of sample applications. The language proved to support a lot of requirements in a quite satisfactory manner, though naturally a couple shortcomings of the original
design have been identified over time. Apart from the document defining the language in the form originally designed by the IDEA consortium [5], a comprehensive introduction into Chimera can be found in the book presenting the IDEA methodology [5].

Chimera consists of a conceptual model (called Chimera Model, short: CM), providing object-oriented modeling facilities, and of a conceptual language (called Chimera Language, short: CL), providing data definition commands, declarative queries, procedural primitives for database manipulation, as well as various forms of rules and constraints. Chimera has been designed in such a way, that it can either serve as an interface to a stand-alone database system or as a database sublanguage embedded in various procedural host languages. Whereas the Chimera implementation developed in Milano has been embedded in C++, our own version of the language is hosted by Prolog. Certain basic linguistic decisions in CL, such as primitive value types, syntax of primitive values, conventions for distinguishing variables, constants or functions, and so on, may thus vary from one embedding to the other due to the particular conventions of the host language.

CM is based on \textit{values} and \textit{objects}, classified into value and object \textit{classes} and \textit{types}. Objects have identity and state. Object identifiers are special, system-generated values, used for expressing links between objects. States are records of (complex) values possibly referring to other objects. Object classes can be organized in hierarchies supporting (multiple) inheritance and overriding. Apart from \textit{attributes}, several other features can be employed in order to characterize semantics as well as behaviour of the objects in a particular class: \textit{constraints} impose conditions on legal states of the respective class, \textit{operations} represent specialized methods applicable to objects of that class, and \textit{triggers} can be used for expressing behavioural patterns by which the DBMS may react to manipulations of these objects. Attributes, constraints, operations, and triggers targeted to a particular object class in this manner can be viewed as modules of a Chimera schema. In a class definition, first the \textit{signatures} (i.e., domains of attributes and parameters) of the respective concepts are introduced.

Figure 2, exhibits an example of a class hierarchy – modeling the lines and connections in a railway network – in the graphical style offered to users of the schema design facilities of one of our tools (see below). Existence of targeted features in the definition of the respective classes is indicated by special icons. On clicking, they unfold into boxes listing the signatures of the individual targeted attributes, constraints, or the like.

In addition to targeted concepts, a Chimera schema may contain a number of untargeted concepts as well. Untargeted constraints can be used in order to express global constraints, imposing conditions on instances of more than one class. Using this device, it is possible to express conditions which could not otherwise be expressed in a targeted manner, because the classes involved are not connected via an attribute link. But even if such links exist, designers may prefer not to designate any of the classes as the “target” of the constraint. In a similar way, untargeted operations and triggers can be introduced. Another global feature in CM is the \textit{view} concept. Views are (derived) predicates on objects of several classes which can be used for expressing certain logical relationships without being forced to objectify the instances of that relation by forming an additional class. Thus view instances don’t have OIDs. Last not least, user-defined value types (introducing implicit subsets of the predefined types of the language) and value classes (explicitly enumerating instances of an application-specific type) may be defined.

The deductive aspects of Chimera are formulated by means of expressions of the declarative sublanguage of CL. CL combines linguistic features of Datalog with an SQL (or better: OQL) style of attribute reference. Formulas of this declarative sublanguage are used to construct \textit{deductive rules} as in Datalog. They are employed for defining the population of derived classes, values of single- or set-valued attributes, instances of views and value classes, and integrity constraints. A derived class in Chimera has to be a subclass of at least one extensional class, the instances of which have been explicitly and individually created. Thus Chimera doesn’t allow for virtual objects (identified by derived or invented OIDs). This omission of a concept already proposed in various other DOOD languages is motivated by the inherent theoretical complexity of the issue which we didn’t master by the time Chimera was designed. Derived attributes may be introduced by rules defining them either as single-valued – in which case run-time checks for value uniqueness are required – or as set-valued – in which case individual instances of the set constituting the attribute value are derived rather than the set as a whole. Thus Chimera does not suffer from grouping problems for derived, set-valued attributes as occurred, e.g. in \textit{EELC} [2]. As usual, deductive rules in a schema are required to be safe and stratifiable.

Deductive rules defining various derived components of a class definition are introduced in the DDL of CL in a separate section, called the \textit{implementation} of the respective class. Apart from these deductive rules defining attributes and constraints, an implementation section also contains the code of each targeted operation as well as the active rules implementing targeted triggers. The active rule language of Chimera is based on a set-oriented semantics similar to that of the Starburst rule system [19]. The event part of an \textit{active rule} consists of one or more event pattern specifying situations in which the rule is to be activated. At present, only occurrences of data manipulation requests concerning the class under consideration are viewed as events. Event patterns may not be parameterized as they are intended to react to more than one instance of that pattern occurring since the respective rule has been triggered last. The condition part of an active rule is a declarative expression evaluated over the current database as well as over an event log recording the instances of the events occurred. Thus, parameter values of triggering events can be accessed declaratively within the rule condition. Action parts are sequences of data manipulation commands expressed in the procedural sublanguage of CL. The semantics of active rule processing can be controlled by a variety of parameters included in an active rule, such as coupling mode (immediate/deferred), event consumption mode (consuming/preserving), or priority declarations.

The procedural features of CL include means for querying and updating databases. There are two basic modes of data manipulation. In URI mode (user friendly interface) commands are interactively input at the screen and answers are output to the same device. In this mode, query results are either directly displayed, or passed to subsequent update commands. In API mode (application programming interface) DML statements are embedded into a host language program. Query results are collected into sets or lists and are bound to program variables for further manipulation. The update facilities of Chimera comprise creation and deletion of objects, modification of extensional attribute values, object migration from sub- to superclasses (and vice versa)
as well as population of value classes. Chimera supports a transaction concept as well. Transactions are composed of chains of individual data manipulation commands or operation calls, linked by either a set-oriented or an instance-oriented sequence operator. Each such chain is called a transaction line. A transaction is a sequence of transaction lines, where the end of each line constitutes a potential rule triggering point during the course of the transaction. Thus active rules can be executed either immediately after the end of the transaction line containing the triggering events, or at the end of the entire transaction during which it has been activated.

In summary, Chimera provides a quite powerful combination of standard features of object-oriented models and modestly innovative versions of established techniques from both areas of database rule research, blended into a uniform language in an original manner.

3 Chimera Prototyping Tool (CPT)

The Chimera Prototyping Tool (CPT) is an experimental implementation of the Chimera data model and language. It supports users during schema design and prototyping. CPT can be accessed using a Command Interface (CPT/CI) or a Graphical User Interface (CPT/GUI) offering quite elaborate facilities for incremental schema design. CPT has been implemented on top of a newly developed database programming language, called Phoenix [9], which combines declarative, imperative, and reactive features in the context of the relational model. Phoenix and thus CPT have been designed as single-application systems, i.e., for each application, a new database has to be created containing exactly one schema (plus data conforming to this schema).

In the following we will first give a short overview of CPT/GUI and then briefly explain how Chimera schemas and data manipulation commands are mapped onto Phoenix.

3.1 Graphical User Interface (CPT/GUI)

The graphical user interface of CPT [11] provides intuitive, but powerful means for visualizing and developing Chimera schemas. It supports advanced graphical schema design without burdening the user with the odds of applying purely textual data definition commands. However in addition, CPT/GUI offers a fully synchronized text-based schema editor. Any schema change done within the graphical interface is immediately visible in the textual editor and vice versa. Furthermore, CPT/GUI allows to save and load schemas in a temporary format such that incremental schema design over multiple sessions is possible. As far as data manipulation is concerned, only a very basic textual interface is provided at the moment. The development of a more sophisticated, graphical DML interface is planned for the next future.

CPT/GUI does not try to represent all aspects of an entire schema in one view but – for the sake of clarity – uses different graphical means for schema visualization. Within CPT/GUI object classes and their relationships are the main focus of attention. When developing a new Chimera application, the user may first complete the design of the entire class hierarchy neglecting all details of the individual classes. Afterwards, the classes can be treated separately by implementing their targeted concepts (i.e. attributes, constraints, operations, and triggers). Untargeted concepts (i.e. value classes and types, views, constraints, and triggers) are defined in a separate window not connected with the object class hierarchy.

CPT/GUI offers three different modes for visualizing the object class hierarchy. These include two kinds of tree presentations, the layout of which is either automatically computed by the system or may be freely designed by the user (Fig. 2). In addition, the schema may be illustrated by means of a structured list resembling the representation of directories in a file manager. With any of these representation modes, a virtual view is available providing a bird’s-eye.
view on large hierarchies which do no longer fit into the main window. Furthermore, a schema navigator can be invoked which provides a complete overview (by means of list boxes) of the schema under revision and thus allows to quickly reach any component of the schema.

Starting from the graphically represented class hierarchy, the designer may easily browse through the (partially) existing schema and add or change implementation details. Double-clicking on any of the object class icons as shown in Figure 2 will open a view on various implementation cards which are available for defining the attributes, constraints, operations, and triggers of the selected class. Figure 3 shows the attribute definition card which is in the foreground when selecting an object class.

New attributes can be introduced by establishing a connection between the object class icon in the center of this card and the type the attribute is supposed to have. All types that are currently available in the schema under construction are presented in the list box. Complex types can be constructed using the structure icons at the lower left side of the definition card. Double-clicking on any of the attribute names will open the implementation window providing all the details of the selected attribute, especially the deductive rules of derived attributes.

Furthermore, CPT/GUI allows to open the implementation windows directly from the class hierarchy representation. To achieve this, the attribute and constraint icons in Figure 2 tell the user whether attributes and/or constraints have been defined for a particular class. Double clicking on such an icon presents a box including the names of the existing concepts. Double-clicking on any of these entries will directly lead to its implementation.

When designing CPT/GUI we took care that the various definition cards present a homogeneous view to the user. For this reason, the steps to be performed for implementing attributes have been adapted for most of the other targeted and untargeted concepts. Only the implementation of triggers and external procedures did not fit into this scheme.

3.2 Mapping Chimera onto Phoenix

The present implementation of Chimera has been realized by developing a compiler mapping Chimera DDL statements statically, and Chimera DML statements dynamically onto data structures and programs in Phoenix. The main challenges during the design of the compilation strategy have been caused by two problems: the mapping of a class hierarchy (involving classes, attributes, and operations) to relations and procedures, and the mapping of Chimera active rules to trigger-reaction rules in Phoenix. The compilation of value types, value classes, views, and integrity constraints is rather straightforward, because all these structures directly correspond to relational concepts. In the following we briefly sketch the main issues related to the problems mentioned; a full treatment can be found in [10].

Class hierarchy: The basic idea underlying the implementation of Chimera schemas is to store each object in exactly one extensional relation corresponding to the most specific class of the object. From these base relations the overall set of instances of a class will be derived by means of deductive rules.

For an object class $c$, the extensional attributes are summarized within one derived relation, called $c$ as well, while each derived attribute $a$ is defined by an individual derived relation, denoted $c.a$. In order to relate attribute values with a particular object, all these relations include an additional argument for representing the identifier of this object. Hence, for an object class $c$ having $n$ extensional and $m$ derived attributes, we define one $n+1$-ary relation, i.e. $c$, and $m$ binary relations. The most specific objects of $c$ are stored in a relation, named $\text{ext}_{c}$, having $n+1$ arguments as well.

The inheritance relationships of Chimera classes are implemented by means of deductive rules appropriately linking the derived relations to the extensional ones. The relation $c$ representing all the instances of the object class $c$ is derived by summarizing the most specific objects in $\text{ext}_{c}$ and all objects which are included in the direct extensional subclasses $c_i$ of $c$ where $i = 1, \ldots, n$. Hence, CPT generates the
following rules:

\[ c(a, x_1, \ldots, x_n) \rightarrow \text{ext}_c c(a, x_1, \ldots, x_n) \]
\[ c(a, x_1, \ldots, x_n) \rightarrow c_1(a, x_1, \ldots, x_n, y_1, \ldots, y_m) \]

where \( y_1, \ldots, y_m \) are variables referring to the extensional attributes which are locally defined for the subclass \( c_1 \) of \( c \).

One has to note that derived classes can be ignored during this compilation, as in Chimera derived classes are virtual subsets of a superclass not introducing any additional objects.

When implementing derived attributes, we have to take care that the computation of an attribute value is performed wrt. the implementation that is valid for the respective object, i.e., that belongs to the most specific class of the object. Hence, we define the rules:

\[ c.a(a, x) \leftarrow \text{mac}_c c(a), \$c.a'(a, x) \]
\[ c.a(a, x) \leftarrow \text{mac}_c c(a, x) \]

where the predicate \( \$c.a' \) refers to the implementation of the attribute \( a \) corresponding to class \( c \). The relation \( \text{mac}_c \) represents the local instances of the object class \( c \) which have to be derived from the respective extensional relations \( \text{ext}_c \). Unfortunately, the \( \text{ext} \)-relations do not always correspond to the local extension of a class \( c \). This discrepancy arises because the local objects of derived classes are stored in the extensional relation corresponding to the extensional class from which they are derived. Hence, a relation \( \text{ext}_c \) summarizes the most specific objects of the class \( c \) as well as of all its derived subclasses. For this reason, we use separate relations, \( \text{mac}_c \), which – in the case of extensional classes – are derived from the “\( \text{ext} \)”-relations by excluding all objects that belong to a derived subclass. In case of derived classes, they are derived according to the respective (translated) population rules.

Note that the compilation of operations is performed following a quite similar technique. Operations are mapped onto Phoenix procedures which use conditional statements for determining the most specific class of the object for which the operation is called. Then the implementation corresponding to this class is chosen and executed.

**Chimera triggers:** The compilation of Chimera triggers into Phoenix trigger-reaction rules involves several problems which are due to the considerable differences between the active rule semantics of both systems. Active rule processing in Chimera is set-oriented, i.e., triggers are processed after several events have taken place. A Chimera trigger is eligible if this set of events includes at least one instance matching an event specification of the trigger. The execution of triggers is set-oriented as well which means that a trigger is executed only once for all its triggering instances (cf. [5]). In contrast to this, active rule processing in Phoenix is instance-oriented, i.e., it is invoked immediately after a single event occurs and the reaction of a triggered rule is executed separately for each triggering instance (cf. [9]).

In order to realize the behaviour of Chimera triggers in Phoenix we distinguish two phases of rule processing in Chimera. Each phase will be implemented by adequate Phoenix trigger-reaction rules. In the first phase, the triggering process of Chimera active rules is simulated, i.e., even if more than one matching event occurs, the reaction to be performed is triggered only once. As Phoenix rules react to each triggering event, we have to generate trigger-reaction rules that ignore all triggering events except one. In the second phase, the condition of the Chimera trigger is evaluated and the reaction is executed.

4 Passive Rule Design Tool (PRDT)

One of the main difficulties arising while designing DODD schemas is caused by the problem of controlling logical dependencies between the concepts in a schema. Such dependencies are on the one hand introduced via deductive rules. On the other hand they originate from the implicit assumptions inherent in the class hierarchies of an object-oriented model, such as inclusion of class extensions, inheritance and overriding (cf. [3, chap. 9 and 11]). The Passive Rule Design Tool (PRDT) [15] aims at supporting Chimera application designers in analyzing whether the deductive rules in a Chimera schema are syntactically and semantically well-formed. It explains and visualizes errors which might have occurred, and points out possible actions to be performed in view of overcoming deficiencies of the design.

The schema designer activates PRDT from within CPT in order to check the deductive rules defined so far. For individual rules, the standard syntactic criteria for deductive databases [4] are checked, i.e., the type of each term occurring within a rule is deduced and checked for its correct use, and the finiteness of the set of possible bindings for each variable (safeness) is determined.

Whereas stratifiability is a well-known concept from relational deductive databases, further difficulties arise from object-orientation as this introduces additional implicit dependencies between derived concepts. The approach underlying PRDT is to make explicit these dependencies by (virtually) extending the set of deductive rules of a schema, and then to analyze this extended set by standard stratifiability checks.

As an example, consider the subtle implicit dependency between derived attributes and classes redefining such attributes. Assume that the derived attribute \( a \) is introduced in the object class \( C \), where it is defined by a set of rules of the form

\[ \text{Self}.a = X \leftarrow \text{body}_c. \]

This attribute – including its implementation – is inherited by each subclass \( SC_i \) of \( C \) not redefining it. If the attribute is accessed for an instance of this class, the rule body as defined within \( C \) is evaluated, with \( \text{Self} \) bound to the current object. Assume however that a subclass \( SC_i \) redefines the attribute implementation as \( \text{Self}.a = X \leftarrow \text{body}_{y_{SC}}. \). Attribute accesses for objects of this class must be evaluated using the local definition. The local definition has to be taken into account although the instances of \( SC_i \) are also instances of \( C \) due to upward instance inheritance. This implies that the body of the original rule definition in \( C \) has to be extended in a way such that it is never evaluated for instances of \( SC_i \), i.e., it has to be extended to

\[ \text{Self}.a = X \leftarrow \neg SC_i(X), \text{body}_c. \]

The implicit negation resulting from overriding has been made explicit and may influence the stratifiability of the deductive rule set defined within that schema.

The result of rule analysis is graphically presented by PRDT by displaying the dependencies between derived concepts (Fig. 4). This is done in a style tightly integrated into CPT/GUI and similar to the way propagation paths are displayed in PROP. Since dependencies among concepts are however defined by the rules which implement them, a further inspection of those rule relationships is possible.

If a dependency belongs to a critical, non-stratifiable cycle, the schema designer has to change his schema in order to make the rule set stratifiable. PRDT supports the schema
5 Propagation Rule Compiler (PROP)

PROP is a tool supporting developers of Chimera applications during schema design and prototyping. Its purpose is to analyze all deductive rules implementing derived concepts (attributes, classes, views, and constraints) in a particular Chimera schema in view of determining all implicit changes which are induced by these rules when an explicit update occurs. Based on this analysis, the tool automatically generates Chimera triggers that are able to compute the differences between two consecutive database states.

The computation of such differences is one of the key tasks a deductive database system is expected to automatically perform. Information about the changes of a database can be exploited for a variety of purposes, such as integrity checking, constraint enforcement, or incremental maintenance of materialized views. Furthermore, it can also be used for supporting triggers that may react to updates of derived data. If change computation is automatically provided by a Chimera system, application designers are liberated from the exhausting and difficult task of programming a considerable amount of complex triggers themselves.

The triggers generated by PROP are called propagation rules, as during active rule processing the updates are propagated in a bottom-up manner from the extensional concepts to the dependent derived ones. In deductive databases the process of update propagation constitutes a complex multi-stage inference process, as a single change of some base data may affect the evaluation of various deductive rules and thus lead to induced updates possibly affecting further rules.

The motivation for using Chimera triggers in order to implement update propagation is twofold: On the one hand, each propagation step can be very naturally interpreted as being driven by an update to be propagated, and as causing a follow-up modification of a rule-defined concept. The specific conditions under which the propagation step is performed make up the condition part of a trigger. On the other hand, using a system-component which exists anyway – in this case a trigger interpreter – for implementing internal services of the database system is likely to reduce size and complexity of the resulting overall system.

However, update propagation is not the only task PROP is responsible for. In addition, it offers a kind of explanation facility for making explicit in which way the various concepts of the schema under consideration depend on each other. Within a graphical interface, the logical dependencies and their interactions with the various forms of updates on them are presented to an application designer and can be “animated” by simulating updates and their consequences on an abstract level. This way the designer is supported in getting a much more thorough understanding of the inherent complexity introduced via deductive rules.

In the following we will first briefly address the course of rule compilation and then illustrate the main features of the graphical explanation component. For a complete technical documentation on rule compilation we refer to [13] while a more detailed description of the explanation component can be found in [12].
5.1 Rule Compilation

The update propagation approach underlying PROP essentially transfers techniques developed for relational deductive databases, e.g., [7, 18], to the DOOD case. However, due to the richer set of concepts and assumptions present in object-oriented models, there are numerous new and non-trivial problems which require the original techniques to be modified, refined, and extended. Among others, problems arise from model-inherent assumptions which are implicit in the definitions of deductive concepts. For instance, the special variable Self implicitly ranges over all instances of the class to which the rule belongs. Further peculiarities result from additional data manipulation primitives (e.g. object migration) not known in relational databases, and from the possibility to specify class hierarchies and to redefine derived attributes.

PROP takes as input a Chimera schema and generates Chimera DDL statements defining object classes, views, and triggers which provide the basis for an automatic computation of all follow-up modifications resulting from an arbitrary update of extensional data. These DDL statements are used to extend the original schema such that the resulting one has the same semantics with respect to the user-defined application, but additionally meets all requirements for an automatic propagation of updates.

In order to represent the various kinds of updates, PROP generates so-called delta classes the objects of which represent individual modifications of the Chimera database. Based on these delta classes, delta views are defined which are necessary for preventing multiple propagations of the same update. The task of update propagation is thus to create delta objects correctly and completely describing the real changes of the database. Hence, for each possible change of extensional data, the compiler constructs an initial propagation rule which is responsible for automatically creating a delta object describing this base update. In order to compute all subsequent modifications which are directly induced by some known update, PROP generates so-called general propagation rules. These rules react to the creation of delta objects, compute the induced updates, and act by creating further delta objects describing the computed changes. For each deductive rule the compiler generates several general propagation rules depending on the different modifications that may directly influence the evaluation of the body and thus the value of the derived concept. In addition, the execution of further tasks (as discussed above) can be expressed by means of triggers as well, denoted final propagation rules. These rules react to the creation of objects in delta classes, and then perform a specified action. However, for the time being the generation of final propagation rules is not supported by PROP, but completely left to the application designer.

5.2 Explanation Component

As pointed out above, the Propagation Rule Compiler consists of two components, one being responsible for the creation of update propagation rules, the other offering an explanation facility that can be applied during schema design or evolution. Based on the generated propagation triggers, the purpose of the explanation component is to graphically illustrate the logical dependencies established by the deductive rules and to point out all possible consequences a database change may cause. (For the time being only dependencies referring to derived classes and attributes can be visualized.)

Starting from a graphically represented database schema (having the same layout as in CPT, cf. Figure 2), an abstract update can be applied and its propagation through the deductive rules will be simulated step by step. This update will affect a particular schema concept (class, attribute, etc.) which may be referenced in several deductive rules. Each logical dependency corresponding to one of the rules will be depicted by means of an arrow connecting the affected concept with the concept implemented by this deductive rule.
Immediately after the abstract update has been “applied”, all direct logical dependencies will be represented. To this end, each derived concept affected by the initial update appears on the screen (if it was not already present) and an arrow connects it with the previously affected concept in order to emphasize the dependency. Afterwards, each of these newly appeared concepts may be selected in order to simulate the next level of update propagation. On demand, each logical dependency can be explained in further detail by double-clicking on the corresponding arrow. Then the deductive rule is displayed, together with the trigger that has been generated for this particular update propagation step. Figure 5 shows the railway application and all direct and indirect dependencies arising when an object for the object class line is created. The logical dependency between the object class path and its derived attribute length is explained in the smaller window on the right.

6 Schema Evolution Assistant (SEA)

Schema evolution is the modification of database schemas during the lifetime of an application, in particular if the database has already been populated. The Schema Evolution Assistant (SEA) [14] has been developed in order to support a schema designer in the evolution of a schema. The designer interacts with SEA using CPT/GUI which provides a flexible means for handling even large and complex application schemas. SEA informs the user about consequences of a schema change the user might not be aware of and might not intend to do.

6.1 A Reflective SEA

To perform schema evolution, SEA has to have access to the schema of the current application. Instead of tightly integrating SEA with the underlying Chimera DBMS, we originally decided to follow a reflective approach for schema management, because using Chimera itself for schema management immediately provides us with all advantages of an advanced data model and language [16]. We thus developed the reflective Schema Evolution Assistant, based on the representation of Chimera schemas as instances of a Chimera meta schema.

The reflective SEA provides a variety of benefits: Data manipulation primitives can be used without changes and without restrictions for schema objects too. The integrity of schemas can be enforced by defining corresponding Chimera integrity constraints which are automatically checked by CPT. Chimera operations can be used for the management of reflectively represented schemas. Chimera triggers can be defined which automatically execute required actions in response to the modification of schema objects. Last not least, the entire CPT can be used for schema management, e.g., for graphically representing and browsing the meta schema, for querying the application schema and so on.

The experience we gained with reflective schema management and evolution showed that Chimera is a very powerful means for structurally representing Chimera schemas, but fails to adequately represent behavioural aspects. The use of derived classes and derived attributes for implicit concepts allows, e.g., the derivation of inherited attributes, when the local attributes of each class and the class hierarchy are given. Integrity constraints can be used for checking the consistency of the represented application schema.

Another very important aspect of the reflective schema representation is granted for free with the transaction concept of Chimera: Hypothetical schema reasoning can be done without any risk because in the case of a resulting inconsistency, the entire transaction containing the schema object modifications can be rolled back without having changed the current schema or data.

Due to the availability of constraints, operations, and triggers within a Chimera schema, the meta schema not only describes the structure of schema objects but also relevant parts of their behaviour. One goal of the consequent application of the reflective approach to schema management for Chimera would be to incorporate further parts of schema behaviour into the meta schema. This way, the behaviour of a schema is made explicit and, if permissions are granted, can thus be modified by a Chimera user.

However, such a far reaching integration can not been done in the current meta schema because it would require the extension of Chimera towards a complete database programming language with meta level constructs – an extension which we decided not to perform due to compatibility and complexity considerations. Without these extensions, the implementation of any non-trivial code in Chimera heavily relies on external procedures, violating the reflectivity principle and turning it ad absurdum when most interesting procedural code consists of external procedure calls.

We therefore decided to suspend our work on the reflective Schema Evolution Assistant, realizing that this very powerful approach is still desirable but not reasonable with the Chimera model and CPT currently available.

6.2 A Non-Reflective SEA

Trying to keep the advantages of a reflective architecture, we designed the non-reflective Schema Evolution Assistant which uses some of the components originally written for the reflective version of SEA but uses the CPT data dictionary for schema representation instead of a reflective schema database. We divided the non-reflective SEA into two main components, planning respectively executing schema evolution requests.

During the planning phase, SEA takes into account the effects of the requested schema modification on the consistency of the schema itself, and on the instances already existing in the database. In this phase, the schema designer may be asked for strategic decisions if alternatives are available. The result of the planning phase is a script which is presented to the designer for acknowledgment and is finally executed on the current Chimera database.

The transaction concept which is not available in the non-reflective SEA is partially replaced by the separation of schema evolution into the two phases. There is, however, no other way for testing whether the schema resulting from script execution will be consistent or not than explicit testing during the planning phase. Once acknowledged by the user for execution, a script is not an atomic schema change “transaction” like in the reflective case, but executed step by step, possibly omitting low-level changes which cannot be performed successfully. However, omitted changes are recorded and can be reacted upon by the schema designer.

The advantage of the non-reflective SEA, however, is that with the renunciation of reflectivity, full computational power becomes available to the SEA routines. Especially the planning component of the SEA supports the flexible creation of scripts, taking into account actual database states as well as user decisions concerning strategies or alternatives.

Regarding the central problem of schema evolution, the adaptation of existing instances to the new schema, SEA
actually supports the immediate conversion of affected instances [20]. We currently investigate how to integrate other strategies (delayed conversion [8], schema versioning [17]) into SEA such that they coexist and can flexibly be chosen for a specific application. However, for such high-level schema and strategy management we still prefer to use a reflective approach after appropriately extending Chimera respectively the CPT.

7 Conclusion

The Chimera tools environment provides a flexible, powerful means for supporting designers of advanced applications in a very expressive database language. The complexity of the language naturally leads to a particular need for tool support during schema design. A detailed analysis of intermediate and final schema versions as well as extensive prototyping is required. Based on a full application design methodology, our tools support various key phases of Chimera application development.

In this paper, we briefly described the Chimera data model and language as well as our prototyping tool (CPT) which implements Chimera on top of a relational back-end system. CPT constitutes the basis for a variety of tools we developed for supporting schema design (CPT/GUI), passive rule analysis (PRDT), generation of propagation triggers (PROP), schema prototyping (CPT), and schema evolution (SEA). All tools are closely integrated, thus leading to a homogeneous Chimera tool environment.

Future work will aim at consolidating and extending the Chimera tool environment as well as the individual tools.

Acknowledgments

The authors would like to thank the master students Elmar Ascheid, Ulrich Daugs, and Thomas Rath for their efforts in implementing PRDT, CPT/GUI, and PROP.

References


