# CONSOLIDATED SPECIFICATION OF CHIMERA (CM and CL)

Summary: This document constitutes the consolidated definition of Chimera; it is a revised version of the T21D2 deliverable presented in June 1993

Author(s): Stefano Ceri, Rainer Manthey

Contributor(s): Elena Baralis, Elisa Bertino, Christoph Draxler, Ulrike Griefahn, Danilo Montesi, Andrea Sikeler, Letizia Tanca

Issued by: Stefano Ceri
Dipartimento di Elettronica e Informazione
Politecnico di Milano
Piazza L. Da Vinci, 32
20133 Milano
ITALY

Authorised by: S. Ceri

Issue number: 1

Status: DEFINITIVE

Date of issue: 29.11.1993

Reference: IDEA.DE.2P.006.01

Number of pages: ??

# CONTENTS

# 1 INTRODUCTION

This document defines **Chimera**, the conceptual interface of IDEA. It constitutes a revised version of the first specification of Chimera, issued in June 1993 [1]. The major differences between the two specifications have been summarized in a short supplementary document [2]. Chimera consists of a conceptual model (also called **Chimera Model**, short: CM), providing object-oriented modelling facilities, and of a conceptual language (also called **Chimera Language**, short: CL), providing data definitions, declarative queries, procedural primitives for database manipulation, as well as various forms of rules and constraints. Thus, Chimera supports object-oriented, deductive, and active database features.

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, such as, e.g., Prolog, C, C++, or Peplom. In the former case, we refer to CL as the language of a "user-friendly interface" (short: UFI), whereas in the latter case CL can be viewed as an "application programming interface" (short: API). CL syntax will be identical in both modes, except for a few of its basic query primitives. Due to the diversity of programming environments present in the IDEA consortium, aspects of procedural, general-purpose programming have been omitted from CL as far as possible. This issue has been left to the respective programming language into which Chimera is going to be embedded by the respective partners. Certain basic linguistic decisions in CL, such as primitive value types, syntax of primitive values, conventions for distinguishing variables, constants or functors, and so on, may vary from one embedding to the other due to the particular conventions of the host language. In this document we provide a "default" syntax (see Appendix 1) that should be followed as closely as possible in order to ensure a high degree of compatibility between various implementations of Chimera.

The style of this document is mostly informal, based on textual definitions and characteristic examples rather than on formal definitions. It has been the goal of the authors of this document to provide a definition which is complete, unambiguous and clear. However, any informal definition of a language will necessarily be limited in its degree of precision. Therefore this document will probably not be suited as an ultimate reference for all questions concerning Chimera. Moreover, there are several aspects of Chimera that definitely require more careful and intensive consideration in the future. This is particularly the case for active rules, to be refined within WP2, T2 (reactive processing). The Conclusion Section summarizes the major changes proposed in this document, and proposes some considerations about Chimera's interaction with Peplom.

# 2 OVERVIEW OF CM

This initial section of the CM description is intended to provide a "bird eye's view" of the concepts to be found in CM together with an outline of the rationale behind the general design choices made. Each of the main concepts of Chimera is briefly introduced and the relationships between the individual parts of the model are sketched. In the subsequent three sections, each of the concepts in turn will be discussed in depth, data definition primitives of CL will be introduced, and representative examples will be provided.

## 2.1 Objects and Values

CM is an object-oriented data model. **Objects** are abstractions of real world entities (such as persons, companies, or machines). Every entity of an application domain that is to be represented in a database should be modeled as an object. Objects are distinguished from each other by means of unique **object identifiers** (OIDs).

**Attributes** are functions mapping an object to a uniquely defined value. Any semantically meaningful information about a particular object has to be associated with that object by means of one of its attributes. Attribute values of an object may change over time, without changing the identity of that object. The collection of all attribute values associated with an object is called the **state** of that object. The state of an object is represented in Chimera as a record of attributes.

Objects can be manipulated by means of **operations**, defined and implemented by the designer of a particular application according to needs. An operation is essentially one of the procedural primitives offered by Chimera (see Section **??**) or a procedure written in the procedural host language of the environment in which Chimera is embedded. Rather than identifying Chimera operations and procedures one to one, a level of indirection has been introduced: operations are related to procedures by means of "guarded" declarations (see Section **??**). When invoking an operation, the corresponding procedural code is executed only, if the "guard" (a declarative condition) is satisfied in the state of the database reached when the call is processed. In this way, operations are clearly separated from procedures conceptually, and control over operation invocations is assigned to the Chimera run-time system.

A **value** can be either atomic or structured. Atomic values are atomic, printable symbols, such as numbers or strings, or OIDs (i.e., references to objects). Structured values are built from atomic values by recursively applying one of the predefined constructors for sets, lists, or records. Chimera provides a number of predefined **operators** applicable to values, such as arithmetic operators (e.g., *, +, or sqrt) applicable to numbers, or selectors (like head and tail), applicable to constructs.

Objects are the essential components of a database, representing real world entities that are characteristic for the respective application domain. In contrast, values cannot be described by means of attributes and cannot be manipulated by means of operations. Values serve as a means of describing objects, but are not essential components of a database.

## 2.2 Types and Classes

Chimera supports both the notion of type and of class, with a one-to-one relationship between them [1]. Whereas the notion of "type" emphasizes the structural and behavioural similarity of objects, the notion of "class" emphasizes membership of these objects in a common set of instances. In Chimera, most types remain implicit, while classes are defined, populated, and deleted explicitly, thus underlining the database character of Chimera: the main purpose of the language and the model is to provide an interface to a persistent store of collections of elements of the same type.

All objects of a Chimera application must belong to an **object class**. Classes must be defined first, and then objects may be inserted into classes; type definitions (i.e., definitions of structure and behavior) are inferred automatically from class definitions. Whereas attributes as introduced in the previous subsection associate values with individual objects, Chimera provides a means for associating values with entire classes as well. **Class attributes** are functions mapping an entire class to a unique value. Examples of possible class attributes are cardinalities or statistical values such as average age or average salary for a class of person objects. Analogously, **class operations** manipulate an entire class rather than individual instances.

Object classes may be recursively specialized into **subclasses**, resulting in a taxonomic hierarchy of arbitrary depth. Multiple superclasses are possible, but multiple inheritance is subject to restrictions (see Section **??**). A subclass inherits all attributes and operations from its superclasses, but may redefine their implementation. Moreover, a subclass may introduce additional attributes which are applicable to the objects in that subclass only. Subclass definitions are acceptable only if the corresponding (implicit) types are compatible. We will introduce a sophisticated set of subtyping conditions to be satisfied whenever a class is defined as being subclass of another class.

Values are organized by means of types and classes as well. Most atomic values, such as integers, reals, or characters, are predefined and provide a (possibly infinite) "pool" of possible atomic symbols. These values cannot be manipulated by the database user, i.e., they cannot be inserted or deleted. Therefore, we do not regard concepts like *integer* or *string* as classes of Chimera, but as types only. Similarly, structured values are described by means of type constructors (record, set, and list) applied to value types. Structured value types also provide a (possibly infinite) "pool" of structured values. Once a type is defined, it is available in Chimera (for reuse in other object descriptions).

However, Chimera users may introduce user-defined collections of values, called **value classes**. These are populated explicitly, by means of insert operations, according to the needs of applications. In analogy to object classes, value classes are associated with a unique, but implicit type; they do not have attributes or operations. Value classes are used in Chimera as **active domains**: whenever the attribute of an object has a type which corresponds to a value class, the only allowed attribute values for that object must

[1] This major design decision in Chimera has been motivated by our wish of controlling the complexity of the model and especially of achieving maximal compatibility of Chimera with Peplom, the database programming language of IDEA.

belong to the value class.

## 2.3 Object Identifiers and Object References

In order to be able to express object-valued attributes as well as operations and constraints accepting objects as parameters, Chimera explicitly distinguishes between objects and object identifiers. Object identifiers are concrete symbols referencing a particular (abstract) object, but are not identical with that object. Thus, an object-valued attribute, e.g., managers of employees (being employees themselves) is in fact "OID-valued". OIDs do not carry any meaning, but serve purely as internal surrogates. They are generated and manipulated by system software and cannot be retrieved by database users. In addition to OIDs, users of a Chimera application may, however, define their own **external names** for individual objects of particular interest.

This design choice leads naturally to an *object sharing semantics for object-valued attributes* (in which multiple objects may reference one and the same object by means of different attributes), rather than a copy semantics (in which objects referred to by attributes of a given object would have to be copied in the state of that object).

## 2.4 Targeted and Untargeted Definitions

A **schema definition** in CL is a collection of targeted and untargeted definitions. Each type or class definition is a **target**, that is, a unit of abstraction and modularization. Features (such as attributes and operations) which are defined in the context of a given target have a scope that is limited to that target. Thus, targets enable a modular design and some degree of information hiding that is typical of object-oriented design (this is further supported by the separation between the definitions of signatures and implementations of each target, see later).

However, some information in the schema cannot be targeted; for instance, **views** combining information from several classes, or **triggers** affecting multiple classes, or **constraints** relating the state of objects from several classes. Therefore, some definitions cannot be expressed in the context of types and classes; these are called **untargeted definitions**.

Given that targeted definitions are usually easier to understand, control, and evolve, a good design principle for Chimera applications is to choose an appropriate collection of targets, so that most of the definitions in the schema can be targeted.

## 2.5 Derived Attributes, Derived Classes, Views

Passive rules of the CL language may be used for deriving information, as it is normally done in deductive databases. Passive rules can be applied in the following situations:

- A subset of attributes, called **derived attributes**, can have values which are defined by means of rules instead of individual update operations over time. Derived attributes are part of the object's state.
- In the context of hierarchies of object classes, populations of some subclasses, called **derived classes**, may also be defined by means of passive rules, which generate the subclass population implicitly when certain properties of the superclass hold.
- Finally, untargeted **views** are always defined by means of passive rules which combine information from one or more classes of Chimera. Each view is introduced independently of a particular type or class definition and is "implemented" by one or more passive rules.

## 2.6 Constraints and Triggers

In addition to the basic structural and behavioural concepts of attributes and operations, Chimera provides two more concepts: constraints and triggers. Both concepts can be either targeted or untargeted.

**Constraints** are a means of restricting the contents of the database. Constraints consist of conditions (expressed in CL). They have a name, and they may have output parameters. In case of a violation of a constraint, these output parameters return values specific to the cause of the particular violation. Constraints that are targeted to a particular class may either restrict the extent of the respective class or restrict the set of legal values of its attributes; some constraints may be targeted to value types. Untargeted constraints restrict the set of legal database states and usually relate two or more classes.

**Triggers** are a means of introducing specific reactions to particular events relevant to the database. Such events are currently restricted to database specific operations (i.e. queries and updates), operation calls, and constraint violations. Other events could be time-related or external events [2]. Reactions are calls to procedures written in either CL or the embedding host language. The execution of reactions is subject to conditions on the database state reached whenever an event is monitored; further, triggers are prioritized, so that when several of them can be fired a partial order is imposed. Triggers are named like constraints, but do not have parameters. A trigger that refers to a single class in each of its components (i.e., event, condition, and reaction part) can be introduced together with the definition of that particular class, as a targeted concept. All other triggers have to be introduced individually. Triggers are synonymously referred to as **active rules** within this document, because both notions are commonly used in the active database community, and in order to emphasize there relationship with passive rules (the former being imperative, the latter declarative specifications of certain autonomously initiated behaviour of the database system).

[2] The exact nature of all events supported in Chimera will be investigated in more detail in WP2 T2 (reactive processing).

## 2.7 Signature and Implementation

When defining a new class, all attributes and operations of the new class have to be specified together with the corresponding domains (i.e., the class from which the attribute values or the operation parameters have to come). In addition, names and parameters of those constraints and triggers targeted to the particular class have to be introduced. The entirety of these name and domain definitions makes up the **signature** of the respective class.

Concepts introduced and "typed" in the signature of a class, then have to be "implemented". There are different ways of implementing a concept, depending on its particular nature. Classes can be populated either by explicit, individual creation of all instances, or by implicitly and collectively defining their instances by means of passive rules. The same applies in principle for attributes and class attributes: attribute values can either be introduced individually during object creation, or collectively by means of passive rules. Constraints (and class constraints) are always implemented by passive rules. Operations are implemented by means of a "guarded procedure body", as mentioned earlier. Finally triggers are implemented by active rules. The association of attributes, classes, constraints, operations, and triggers with the expressions implementing them is performed after the respective signatures have been introduced. The entirety of these concept definitions is called the **implementation** of the respective class.

Both signature and implementation are considered part of the class definition. However, CL provides different data definition operations for introducing signature and implementation, in order to reflect that implementations are usually introduced after signatures have been defined.

# 3 VALUES

In Chimera, values play a purely descriptive role: they are regarded as a means of describing objects, but are not intended to represent main entities of interest in a specific application domain themselves. As opposed to objects, values are not abstractions, but concrete symbols or structures composed from symbols. Values cannot be described by attributes, as opposed to objects. Consequently, there is no need to change their "state" by means of operations and no need for an "identity" to be preserved under state changes. A value represents itself wherever it occurs. Values can occur as attribute values of objects and as parameter values of operations and constraints.

Object identifiers are concrete symbols referencing a particular (abstract) object. Consequently, we include object identifiers into the set of possible values. Values can be classified as follows:

- atomic values
    - object identifiers

  - symbols
- structured values
  - sets
  - lists
  - records

Atomic values are regarded as simple, undecomposable items. All atomic values are predefined in Chimera, i.e., users cannot create new values. This is reasonable as normally the collection of printable symbols is determined by the programming environment and/or the host language of a particular implementation anyway, and is not at the disposal of users or application designers. We assume that there is a finite, but practically unlimited stock of object identifiers predefined by the system. Thus creating a new object means choosing one of the possible OIDs (automatically, of course) rather than generating a new OID.

Structured values are complex terms composed by applying set, list, or record constructors to atomic values or (recursively) to structured values. Thus, nested structures of arbitrary depth can be obtained. The individual components of a structured value can be accessed by special operators, such as predefined, structure-specific functors like head and tail of a list, or user-defined selectors in a record.

## 3.1 Predefined Value Types

All symbols that may appear in any Chimera expression belong to one of the six predefined basic value types of Chimera:

- **oid**
- **integer**
- **real**
- **boolean**, i.e. the values {true, false}.
- **char**
- **string** or **string(n)**, where $n$ is a number denoting the fixed length of a string.

*Oid* denotes the generic type of all possible object identifiers and depends on the conventions chosen for the particular implementations of Chimera. In contrast to the other predefined value types, *oid* cannot be directly used in the definition of a type or a class, where it is instead possible to indicate the name of an object class. Such an object class will however be represented by means of *oids*; implicitly, all object identifiers assigned to objects of a given class constitute a value type with the same name as the respective object class.

Other predefined types (such as byte streams for representing images or text) may be supported, but at least the six types listed above are suggested as a kind of "minimal" set of basic types in every implementation of the interface. The exact definition of the respective basic value type will be left to the individual implementations, in order to avoid any mismatch between the basic types of the particular host language and those of the version of Chimera embedded into it.

The special value *null* is a polymorphic element of every value type used to indicate an unknown element of the respective type.

For any value type T, predefined or user-defined, we assume that the type of all finite sets and the type of all finite lists of T-values are available as structured value types, too. This is automatically the case without requiring any explicit definition by the user. Analogously, a potentially unlimited stock of finite record types composed from already introduced component types and from freely chosen labels is at the disposal of users without any prior definition. These "predefined" structured value types are recursively defined as follows:

- If T is a value type, then set-of(T) and list-of(T) are structured value types.
- If $L_1, L_2, ..L_n$ are distinct names and $T_1, T_2, ..T_n$ are value types (not necessarily distinct), then record-of($L_1 : T_1, L_2 : T_2, ..L_n : T_n$)
  is a structured value type, where $L_i : T_i$ is the i-th *component* of the record type, $L_i$ is the label and $T_i$ is the type of the component.

For record types, some or even all labels can be omitted. In this case the unlabelled components are assumed to be implicitly labelled by their relative position, i.e., an implicit label $L_i = i$ is assumed. Examples of legal structured value types are:

```
set-of(integer)
list-of(boolean)
record-of(re:real, im:real)
record-of(integer,integer)
set-of(record-of(day:integer, month:integer, year:integer))
```

## 3.2 User-defined Value Types

Apart from using the predefined value types offered by Chimera, designers of a particular application schema may introduce application-specific names for certain value types. Such a desire will probably arise for record types more frequently than for set or list types. Examples are *complex* for denoting records of two real values representing complex numbers or *date* for denoting records of three integers representing dates according to some ordering convention of day, month, and year. Such user-defined names are introduced in CL by means of **type definitions**, introducing a new name and associating it with a predefined value type:

```
define value type complex:
          record-of(re-part:integer, im-part:integer)
end

define value type date:
          record-of(day:integer, month:integer, year:integer)
end
```

In case of complex numbers, the newly defined type name *complex* has been introduced as a synonym for (and an abbreviation of) the predefined record type, in particular fixing a certain set of labels indicating the meaning of the respective component within the respective application domain. In the other example, *date* is actually intended to denote a proper subtype of the type of all possible records of three integers. In order to cope with situations like this, user-defined value types may be restricted by means of constraints on the respective type. Constraints are named and may be parametrized.

Name and parameter domains of a constraint are defined as part of the type definition to which it applies. The actual conditions expressing the restrictions are introduced by means of passive rules in a separate **implementation definition**. Format and syntax of these definitions in CL is identical for value and object types and will be discussed in more detail in connection with object classes in the next section; however, note that the only parameter of constraints defined over value types is exactly the name of the value type. For the example, a constrained definition for type *date* is as follows:

```
define value type date:
         record-of(day:integer, month:integer, year:integer)
  constraints  improperDate(Date: date)
end

define implementation for date
  constraints  improperDate(Date) <- Date.day < 0;
               improperDate(Date) <- Date.day > 31;
               improperDate(Date) <- Date.month < 0;
               improperDate(Date) <- Date.month > 12;
               improperDate(Date) <- Date.year < 0;
               improperDate(Date) <- Date.day = 31,
                                     Date.month in {2,4,6,9,11};
               improperDate(Date) <- Date.day = 30,
                                     Date.month = 2;
               improperDate(Date) <- Date.day = 29, Date.month = 2,
                                     not (Date.year mod 4)=0;
end
```

If an attribute is defined as ranging over a user-defined object class $O$, this means that the actual values taken by this attribute are the OIDs of the objects in $O$ (i.e., values) rather than the entire objects as such. As an example of a user-defined value type which includes an object type in its definition, consider the *assignment* record combining a project, a start date and a termination date, where *project* denotes an object class:

```
define value type assignment:
  record-of(task:project, startDate:date, terminationDate:date)
end
```

This value type can be used in the context of object class definitions, for instance for defining the attribute *yearActivities* as a list of *assignments*.

## 3.3 User-Defined Value Classes

Predefined and user-defined value types are considered as "abstract domains" in Chimera, in the sense that the set of instances of the type is never made explicit. For such value types, only the existence of a suitable routine for type checking is assumed. However, neither in the database nor somewhere inside the system an explicit enumeration of the individual instances of the type is assumed.

In case users would like to control the extent of a user-defined value type by enumerating its values, a **value class** instead of a value type has to be defined. By doing so, a value type with the same name and definition is automatically generated, but in addition storage for an explicit extent is allocated. As an example of a value class populated explicitly (and possibly incrementally over time) by means of individual insertions, consider postal codes of cities:

```
define value class postalCode:integer end
```

Constraints may be defined on value classes as well, thus restricting their explicit extent rather than their implicitly assumed range of instances. For instance, postal codes in Germany are restricted to being integers consisting of four digits, where only certain such combinations have been actually assigned to cities, towns and villages in the country. This situation is modeled as:

```
define value class postalCode:integer
   constraints   improperCode(Code: postalCode)
end

define implementation for postalCode
   constraints   improperCode(Code) <- Code < 0 ;
                 improperCode(Code) <- Code > 9999
end
```

Having defined *postalCode* as a value class rather than a value type has the following effect: only those four-digit integers explicitly introduced by subsequent insertions will be accepted as legal values for *postalCode*, rather than all possible positive integers below 10000.

Note that the definition of a value class with integrity constraints is separated into two components, a signature and an implementation, too.

# 4 OBJECTS

Objects are abstractions, internally represented by means of an immutable object identifier (OID) assigned to the object automatically at object creation time. The characteristic property of objects is to have a state that may change over time due to the application of operations to the objects. The state of an object is a record consisting of all attribute values of that particular object, labelled by the respective attribute names. Operations applicable to an object are the only means of expressing state changes. Chimera offers generic update operations applicable to every object, but more specific operations, usually combining several individual occurrences of generic update operations, may be specified.

Each object is assigned to a class at creation time, but may be inserted into any subclass or backed to a more general superclass of its initial class during its lifetime without changing its identity. CL offers generic procedures for creating and deleting objects (with persistent or temporary lifetime; see below) as well as for moving and removing existing objects from classes which are related by means of a generalization hierarchy. Moving an object from one class $C$ to another class which is neither a subclass nor a superclass of $C$ is not possible.

Besides this explicit assignment to one class, each object implicitly belongs to all superclasses of its class due to the semantics of subclassing essentially based on subsetting. Objects in a subclass **inherit** all attributes, operations, and constraints defined for any of its superclasses. The possibility of redefining inherited concepts exists. Details on inheritance will be elaborated below as well.

The class of an object implicitly determines its type, comprising all the state-independent components of the class definition, such as, e.g., attributes, attribute domains, operations, constraints and so on. Object types are automatically derived from class definitions.

## 4.1 Object Classes

When defining an object class, all concepts related to individual instances of that class (attributes, operations, and constraints) have to be introduced simultaneously. In addition, several concepts related to the entire class (but not to individual instances) are introduced as well (namely class attributes, class operations, class constraints and triggers). Finally, all superclass relationships between the newly defined class and previously defined classes have to be stated at class definition time. Thus, object classes are the principal targets for developing schemas in Chimera.

As already indicated in the section on values, in CL the definition of the **signature** of a class (i.e., of all names and domains associated with that class) and the definition of the **implementation** of the concepts associated with the class are introduced separately. This choice seems to be appropriate if taking a certain incrementality during the design of an application schema into account. Once a class has been entirely defined (i.e., both signature and implementation of all components of the signature have been introduced), no further modifications of the class definition are accepted. That means that for the time being Chimera does not offer any means for incremental schema evolution. Instead, complete redefinition of the entire object class is necessary when a component of its original definition is to be modified or if new components are to be added.

### 4.1.1 Signatures

As mentioned above, signatures of all concepts associated with an object class are introduced together with the name of the new class (causing an automatic generation of a type with the same name). In this paragraph, we discuss the format of object class definitions. Each object class definition consists of eight parts introducing:

- Superclasses (all direct superclasses of the newly defined class);
- Attributes (of individual instances of the class);
- Operations (causing changes of states of individual objects);
- Constraints (conditions restricting attribute values of individual objects);
- C-attributes (associating values with the entire class rather than with individual instances);
- C-operations (changing the state of c-attributes without affecting instance attributes);
- C-constraints (restricting the extent of the entire class without affecting individual instances);
- Triggers (establishing reactions to arbitrary events affecting class members).

If any of these eight components is missing in a class definition, the respective part is considered empty (e.g., attributes, but no c-attributes). Each of the components consists of a set of signatures of the concepts belonging to the respective category:

- The superclasses part consists of a set of object class names.

- The attributes part consists of a set of attribute signatures, each being a pair:

  ```
  attribute name: domain
  ```

  where *domain* is the value type from which the attribute may take its values.

- The operations part consists of a set of operation signatures of the form:

  ```
  operation name ( list of parameters )
  ```

  where the list of parameters may be empty and each parameter is a triple:

  ```
  role parameter name: domain name
  ```

  Role may be either *in* (for input parameters) or *out* (for output parameters). Parameter names have to start with an uppercase letter because they correspond to variables in the implementation of operations.

- The constraints part consists of a set of constraint signatures of the form:

  ```
  constraint name (list of output parameters)
  ```

  where output parameters may be missing; each output parameter is a pair:

  ```
  parameter name: domain name
  ```

- The c-attributes, c-operations, and c-constraints parts are structured in an analogous way as the attributes, operations, and constraints parts.

- The triggers part consists of a set of (unparametrized) trigger names.

Parameters for operations are fairly conventional, whereas parameters for constraints require some further explanation. Constraint evaluation binds the named output parameters whenever the constraint is **not** satisfied. Parameters can be used, e.g. in the context of explanation support (Task 3 of WP2), for *reporting* of conditions which cause the constraints' violation; they can also be used in the context of transactions, when constraint evaluation can be explicitly requested, to allow for an *early identification* of potential violations. Some output parameters of constraints may have object attributes as domains; these parameters have the same value type as the respective attribute.

Beyond user-defined constraints, Chimera supports some **generic constraints** which are extensively used. Two generic constraints are targeted (i.e., expressed in the context of a class definition); they are introduced by the keywords *key* and *notnull*. Intuitively, keys denote sets of attributes whose "collective" value cannot be identical in two distinct objects belonging to the same class; notnull attributes cannot assume the polymorphic value *null*, they must instead assume a different legal value for their type. Key attributes must not be structured by means of sets or lists.

The following is a first representative example of an object class definition. It has been written in a syntactic style that is used in the UFI mode of Chimera, where we expect a window-based interface or a screen editor to be available which is sensitive to a multi-line, indentation-based style of arranging input. For embedding a similar definition into code of the API, a sequential style based on braces and separators rather than line feeds and indentation has to be used.

```
define object class person
  attributes    name:string(20)
                birthday:date
                vatCode:string(15)
                age:integer
                income:integer
                profession:string(10)
  operations    changeIncome(in Amount:integer)
  constraints   tooLowIncome(N:name)
                key(V:vatCode)
  c-attributes  averageAge:integer
                lifeExpectancy:integer
  c-operations  changeLifeExpectancy(in  Delta:integer,
                                     out NewValue:integer)
  c-constraints invalidLifeExpectancy(I:integer)
end

define object class employee
  superclasses  person
  attributes    emplNr:integer
                mgr:employee
                salary:integer
                dependents:set-of(employee)
  constraints   exceedsMgrSalary(Nr:emplNr)
  c-attributes  maximumSalary:integer
  triggers      adjustSalary
end
```

### 4.1.2 Implementation Definitions

Once each concept associated with an object class has been introduced by means of a signature definition, the individual concepts can be implemented. The term **implementation** stands for specification of the meaning of the respective concept by means of passive or active rules, or by operation implementation. A first rough classification of the different means of implementation is obtained from the following scheme:

- Attributes and c-attributes are implemented by means of passive rules specifying the values of the respective attribute by means of a declarative expression. If so, the attribute is called a **derived attribute**. Otherwise the attribute is called **extensional**, and attribute values of individual objects have to be inserted or modified by means of individual operations.

- Constraints and c-constraints are implemented by means of passive rules associating with a parametrized constraint name a condition that should not hold in any state of the database (thus constraints in Chimera are formulated negatively, stating what should *not* be the case for any legal database state, rather than expressing invariant conditions of the database). When an update turns the value of any such condition to true (i.e., when the positive version of the resp. constraint would be violated), the parameters of this constraint are bound to values characterizing the "violated " instance of the constraint. The implementation of generic constraints does not need to be indicated (although, for uniformity reasons, their semantics could be explained by giving them a generic implementation).

- The implementation of an operation or c-operation in **Chimera** is an expression of the form:

  ```
  Operation-name: Condition -> Operation-Code
  ```

  Operations are explicitly activated by means of an operation call. At calling time, all *input* parameters of the operation must be bound to a value (either a constant or a variable binding). After the end of the operation execution, all output parameters are guaranteed to be bound to a value (possibly null), too. The condition is any declarative expression of Chimera. It guarantees declarative control ("guard") upon operation execution, specified inside Chimera. The operation code is a transaction line of Chimera (see Section **??**); in particular, it can be an arbitrary sequence (or pipeline) of calls to procedures (either one of the procedural primitives of Chimera or written in the host language) executed as reaction to the operation invocation.

- Triggers are implemented by active rules specifying triggering events, conditions, and reactions; in addition, triggers may include priorities indicating in which order the triggers should be fired.

Apart from implementing individual concepts such as attributes or constraints by means of rules, the extent of a class may be defined by means of passive rules as well, i.e., the class may be populated by means of rules referring to other classes from which the objects of the newly defined class are chosen. Such rule defined classes are called derived as well. Class population by means of rules is only possible when a subclass is defined by specializing its superclasses (e.g., *employee* being a *person* whose profession is "employee") [3]. Instead, a superclass is always, implicitly populated from its subclasses (e.g., *person* is populated from *teacher, student*, and *secretary*).

[3] If a population is derived, updates to the attribute values used in the derivation rule may cause insertions or deletions of objects into the derived class. Therefore, the updates mentioned above should be carefully performed.

Population rules as well as implementations of attributes, constraints, operations, c-attributes, c-constraints, and c-operations are introduced altogether in an **implementation definition**. Only trigger implementations are introduced individually, as there are too many separate components of a trigger for being included in a joint definition of all implementations. An implementation definition in CL consists of seven parts (each one may be missing):

- A population implementation, consisting of a set of passive rules defining the class extent.
- An attributes implementation, consisting of a set of passive rules defining values of derived attributes.
- A constraints implementation, consisting of a set of passive rules defining all constraints.
- An operation implementation, consisting of a set of implementations defining each operation.
- Analogous parts defining c-attributes, c-constraints, and c-operations.

The exact syntax of trigger definitions will be introduced later in this document within a separate section entirely devoted to triggers and their execution (see Section **??**). As representative example of an implementation definition we continue defining the person and employee classes, the signature of which has already been introduced above:

```
define implementation for person
  attributes    Self.age=X <- X=1993 - Self.birthday.year
  operations    changeIncome(Amount):
                  integer(New),New=Self.income+Amount ->
                  modify(person.income,Self,New)
  constraints   tooLowIncome(N) <-
                  Self.income<500000, N=Self.name
  c-attributes  Class.averageAge=Y <-
                  integer(Y), Y=avg(X.age where person(X))
  c-operations  changeLifeExpectancy(Delta,New):
                  integer(New), Delta<10,
                  New=Class.lifeExpectancy + Delta ->
                  modify(person.lifeExpectancy,Class,New)
  c-constraints invalidLifeExpectancy(I) <-
                  I = Class.averageAge - Class.lifeExpectancy,
                  abs(I) > 5
end

define implementation for employee
  population    employee(X) <- person(X), X.profession="employee"
```

```
  attributes    X in Self.dependents <- employee(X), X.mgr=Self;
                X in Self.dependents <- employee(X), employee(Y),
                                        Y.mgr=Self, X in Y.dependents
  constraints   exceedsMgrSalary(Nr) <-
                  Self.salary>Self.mgr.salary, Nr=Self.emplNr
end

define trigger  adjustSalary for employee
  events        create;
                modify(salary)
  condition     Self.salary>Self.mgr.salary
  actions       modify(employee.salary,Self,Self.mgr.salary)
end
```

In these implementations, the special variables *Self* and *Class* represent the particular object and the particular class, resp., to which the attributes or class attributes apply.

The fact that an attribute or c-attribute is derived has to be declared in the signature of the resp. attribute; similarly, a derived object class has to be defined as derived. In our example, the complete signature definitions must therefore be changed as follows:

```
define object class person
  attributes    ...
                age:integer, derived
                ...
  c-attributes  averageAge:integer, derived
                ...
end

define object class employee, derived
  attributes    ...
                dependents:set-of(employee), derived
                ...
end
```

## 4.2 Subtypes, Subclasses and Inheritance

In Chimera, an object class can be declared as subclass of one or more already existing object classes when introducing the class. By doing so, an implicit subtype relationship is induced on the object types automatically generated from the respective class definitions. For user-defined value types (and classes), however, a subtype relationship with one of the predefined types is always introduced in the definition of the respective type, but no further means for creating hierarchies of user-defined value types is provided.

By applying type constructors to user-defined subtypes of the basic predefined types, a more sophisticated notion of subtype for values is induced, which also forms the basis of redefinition of attributes and operations of objects. Therefore we formally define this important notion at the beginning of this section:

- If a user-defined value type T has been introduced by means of a type definition of the form *define value type T:*$T_1$ or *define value class T:*$T_1$, then T is a subtype of $T_1$.
- If T is a subtype of $T_1$, then set-of(T) is a subtype of set-of($T_1$).
- If T is a subtype of $T_1$, then list-of(T) is a subtype of list-of($T_1$).
- If T = record-of($l_1 : T_1, l_2 : T_2, \ldots, l_n : T_n$) is a record-type, then every record type record-of($l_1 : D_1, l_2 : D_2, \ldots, l_n : D_n$) is a subtype of T for which every component type $D_i$ is either identical with or a subtype of the corresponding component type $T_i$.

The basic policy for defining subclass relationships between object classes can be summarized as follows:

- The extent of a subclass is a subset of the extent of all its superclasses.
- The attributes, operations, constraints (both on the class as well as the instance level), and triggers defined on a superclass apply to all its subclasses as well.
- Apart from the inherited concepts for each subclass additional attributes, operations, constraints, and triggers applicable to single instances of the subclass or to the suclass as a whole can be introduced.
- Inherited concepts may be redefined (or overwritten) in a subclass definition subject to a number of restrictions, which will be discussed below.
- An object may belong to more than one most specific subclass within a subclass hierarchy, e.g., a person may at the same time belong to subclasses *male* and *employee*, even though the two subclasses are uncomparable with respect to the subclass relationship. In CL expressions referring to objects in the intersection of such most specific, but unrelated subclasses, one of them has to be distinguished for type checking purposes, and determines which implementation of shared attributes is to be applied.
- Multiple inheritance of $C$ from classes $C1, C2, ..Cn$ is allowed, provided that there exists a superclass $CS$ as common ancestor of $C1, C2, ..Cn$. Conflicts due to names of attributes, operations, constraints and triggers which are inherited by $C$ from multiple superclasses have to be resolved by prefixing the inherited concept with the name of the intended superclass.

For an illustration of these basic assumptions concerning subclassing and inheritance, consider the subclass relationship between *employee* and *person* introduced in the previous sections. The state of each instance of *employee* consists of ten components: the six attributes *name, birthday, vatCode, age, income, profession* are inherited from the superclass *person*, the four attributes *emplNr, mgr, salary, dependents* are applicable to employees only, but not to persons in general. In the same manner, three class attributes are applicable to the subclass *employee*, two inherited from *person* - *averageAge* and *lifeExpectancy* - one specially defined for the *employee* class - *maximumSalary.* All operations and constraints defined for *person* are applicable to *employee* too.

### 4.2.1 Redefinition of Inherited Concepts

An inherited concept can be redefined in a subclass definition of CL in the following ways:

1. The signature of an attribute or an operation can be redefined by specializing (or refining) the domain of the attribute or of one or more parameters of the operation, respectively. Specialization of domains of attributes or result parameters of operations can be done by replacing the domain defined in the superclass definition (which always is a value type) by a proper subtype of this domain *(covariance rule)*. Input parameters of operations can instead be refined by replacing the domain defined in the superclass definition by a proper supertype of this domain *(contravariance rule)* [4]. A redefined signature is indicated by adding a qualifier *redefined* to the respective component of the subclass definition. Note that both extensional and derived attributes can be redefined, but they cannot be redefined so that a derived attribute becomes extensional or viceversa.

2. The implementation of an attribute or an operation can be redefined by introducing a different implementation of the respective concept, which "overrides" the inherited definition. Redefinition of implementation can be combined with or without redefinition of the signature. Implementation redefinition has to be performed by means of a special *redefine* statement, in very much the same way as implementations have to be defined after signatures have been introduced.

As an example for redefinition consider a further refinement of the person-employee example. A subclass *fiatEmployee* of *employee* is introduced which redefines both signature and implementation of the attribute *dependents* as well as the signature of the attribute *mgr*:

[4]The use of contravariance for the refinement of input parameters of operations is assumed here mostly for compatibility with Peplom, where it is motivated by the need of safe compile-time checking of programs; however, it is not clear at this stage of development of the IDEA project whether this motivation holds for Chimera programs. Therefore, the (more natural) alternative of using covariance for input parameters of operations as well could be reconsidered in future versions of Chimera and/or in testbeds.

```
define object class fiatEmployee
     superclasses   employee
     attributes     mgr: fiatEmployee, redefined
                    dependents: set-of(fiatEmployee), redefined
                    tax: integer
end

redefine implementation for fiatEmployee
     attributes     X in Self.dependents <-
                      fiatEmployee(X), X.mgr=Self
                    X in Self.dependents <-
                      fiatEmployee(X), fiatEmployee(Y),
                      Y.mrg=Self, X in Y.dependents
end
```

Redefinition of constraints and triggers is currently not allowed in Chimera [5].

## 4.3 Lifetime of Objects

Chimera is a database sublanguage. Therefore objects in Chimera are normally expected to be **persistent**, i.e., to remain reachable in persistent store after the termination of an interactive session and to disappear only on explicit request from the database user. However, Chimera objects may also be **temporary**, in which case they are not entered to persistent store, but are automatically deleted at the end of a session. The following items summarize Chimera's policy concerning lifetime of objects:

1. Object classes may include both persistent and temporary instances. Thus, the granularity of persistence is the object level.
2. The lifetime of an object is determined at object creation time. There are two different creation procedures in CL: *create* generates persistent objects, *create_tmp* generates temporary objects.
3. A temporary object may become persistent by applying the CL procedure *make_persistent*; however, persistent objects cannot become temporary.
4. All objects referenced (by means of attributes) from persistent objects are automatically made persistent at the moment the reference is established.
5. An object belonging to several classes has the same lifetime in all its classes.

[5] Redefinition of constraints is intellectually challenging, as it enables to introduce *exception mechanisms*; it is not clear, however, whether this semantics is the most desired for Chimera. Redefinition of triggers requires to identify redefinition rules for each component (event, condition, action, and priorities as defined by means of a partial order, see below); this issue will be further analyzed in the context of WP2 T2 (reactive processing).

6. Temporary objects are automatically deleted at the end of a session.
7. The lifetime of an object can be queried by means of an additional, system-generated attribute **pstatus** applicable to objects of any class. At object creation time, the value of *pstatus* is automatically initialized by the system to either *persistent* or *temporary*.

For a detailed definition of the various procedures mentioned in this section refer to the CL description in Section ??.

## 4.4 External Names

In Chimera each object can be associated with a unique system-wide external name, which can be used syntactically instead of OID variables. Such names are associated with an OID by means of a bijective "name" function name(OID,Name) which is introduced in a separate **name definition**.

```
assign name(X,stefano)
```

where `X` is bound to an OID. External names cannot be changed during the lifetime of the object.

The introduction of external names is motivated by the resulting user-friendly style of addressing individual objects. The execution of the statements

```
create(person,[name:"Stefano Ceri",wife:teresa],X),
assign name(X,stefano)
```

will first create[6] a person with name "Stefano Ceri" returning a new OID X where `teresa` is an already existing external name and the name `stefano` is created by means of the `define name` statement.

Each class is implicitly associated with a system-wide external name (which is the class name), so that this name can be used for accessing class attributes.

```
person.averageAge
```

Thus external names provide a solution to the problem of referencing class attributes, operations, and constraints, syntactically homogeneous with the handling of properties of individual class instances.

[6]The create statement will be defined in Section ??

# 5 UNTARGETED CONSTRAINTS, VIEWS, AND TRIGGERS

We have already seen that constraints and triggers can be targeted and untargeted. Targeted constraints and triggers can be considered as *local* to the class (or type) in which they are defined. Untargeted constraints and triggers, as well as views, can be considered as *global*, and define "interactions" among classes. Signature and implementation of untargeted concepts are given by means of a unique definition.

## 5.1 Untargeted Constraints

Untargeted constraints are global contraints which must be satisfied by the whole database. As with targeted constraints, they can have output parameters (used to report the instance responsible for the integrity violation after their execution) and may not be refined. One example of such a constraint, connecting object classes *person* and *car*, expressed in CL, is:

```
define constraint tooLowLiability(Name: person.name,
                                  Plate: car.plate,
                                  Liability: integer)
   tooLowLiability(Name, Plate, Liability)  <-
                         person(X), car(Y), X = Y.owner,
                         Name = X.name, Plate = Y.plate,
                         Liability = Y.insuranceLiability*0.7,
                         Liability < 500000
end
```

The output parameters are the person's name $Name$, the car's license plate $Car$, and the car's liability *Liability* such that the person owns the car and the car's insurance liability is too low. Note that, in the context of untargeted contraints, attribute names used for output parameters must be prefixed by the name of the class to which they belong.

Among untargeted constraints, Chimera supports another **generic constraint** which correlates attribute values of different classes. This constraint is introduced by the keyword *inverse*, as follows:

```
define constraint inverse(X:c1.a1,Y:c2.a2) end
```

Inverse attributes relate pairs of extensional, object-valued attributes $a1$ and $a2$, defined respectively for the two classes $c1$ and $c2$, such that $a1$ maps objects of $c1$ to objects of $c2$ ($a1$'s type is $c2$ or set-of($c2$)), $a2$ maps objects of $c2$ to objects of $c1$ ($a2$'s type is $c1$ or set-of($c1$)), and $a1$ is the inverse mapping of $a2$. One example of inverse constraint is:

```
define constraint inverse(L1:car.owner,L2:person.car) end
```

This constraint has the following intuitive semantics: for each mapping from a car to its owner established by the *owner* attribute, there must be a mapping from a person to his/her owned car, established by the *car* attribute, and vice versa. This requires a careful management of insert, delete, and update operations, that is not further discussed in this document. The implementation of this constraint does not need to be defined.

## 5.2 Untargeted Views

Views are used in order to build derived concepts from concepts already present in the database (thereby playing the role of presenting information in a "format" that is most suited to a particular user or user group) [7]. Each view has a signature given by a user-defined value type and an implementation given by a collection of passive rules.

As example of a view, consider the case of an object class *person* with attributes *spouse* and *sex*:

```
define object class person
  attributes        spouse: person
                    sex:    string(10)
end
```

Then, it is possible to define a view *marriage* on *person* relating two married persons, as follows:

```
define view marriage: record-of(husband:person,wife:person)
   marriage((husband:X,wife:Y)) <- X.spouse=Y, X.sex=male
end
```

Views can be considered as "predefined queries", expressed by means of passive rules, whose definition is included in the schema of a Chimera database [8].

## 5.3 Untargeted Triggers

Triggers are means of introducing specific reactions to particular events relevant to the database. Untargeted triggers can be seen as *global triggers* which must react to particular events relevant to the whole database. Trigger definition will be discussed in more detail in section **??**.

[7] For the time being, we are not discussing authorization and protection mechanisms in Chimera; it is expected that views be used for that purpose as well.

[8] The possibility of materializing views (i.e., of evaluating and maintaing their extent) is left to the Chimera compiler as an optimization.

As an example of an untargeted trigger expressed in CL, consider the situation where *employee* objects are *members* of *departments*, they have a *salary*, and a department has allocated a given *budget for salary*. We require that the budget for salary at a department remains greater than (or equal to) the sum of all employee salaries allocated at that department. To do so, a trigger *raiseBudget* is defined, which monitors events like inserting employees, changing their salary, changing their allocation, or changing the department's budget. The trigger's condition is true whenever there is a violation of the above requirement. The action consists in setting the department's budget equal to the sum of the salaries of the employees in the department:

```
define trigger raiseBudget
    events     create(employee)
               modify(employee.salary)
               modify(dept.members)
               modify(dept.salaryBudget)
    condition  dept(D), integer(I),
               I=sum(E.salary where employee(E), E in D.members),
               I>D.salaryBudget
    actions    modify(dept.salaryBudget,D,I)
    after      employee.adjustSalary
end
```

This trigger has a lower priority than the other trigger *adjustSalary*, defined in the context of the employee class; therefore, if both are triggered at the same time, *adjustSalary* should be considered first, possibly performing "local" adjustments of salaries in the context of the employee class, and *raiseBudget* should only be considered when *adjustSalary* is no longer triggered.

Note that the above trigger can be informally called an *active constraint*; it reacts to the violation of the following constraint:

```
define constraint budgetViolation(D:dept, I:integer)
  budgetViolation(D,I) <- I=sum(E.salary where employee(E),
                                               E in D.members),
                          I > D.salaryBudget
end
```

Note that the trigger *raiseBudget* satisfies the following requirements w.r.t. the constraint *budgetViolation*:

- It includes as events *all* the state changes that may lead to a violation of the constraint.
- Its condition corresponds to the body of the passive rule implementing the constraint.

- Its action produces a state where the constraint is satisfied.

One of the future research activities inside WP2, T2 (reactive processing) concerns the management of "passive" constraints by means of active constraints expressed in the form of triggers.

# 6 OVERVIEW OF CL

CL serves two main purposes:

- Definition of implementations of schema components (expressed by means of passive rules, active rules, and operation implementations).
- Manipulation of data by means of queries updates and transactions (expressed by means of either the UFI, or the API interface).

This section provides a "bird eye's view" of the concepts to be found in CL together with an outline of the rationale behind the general design choices made. In the subsequent sections, each of the concepts in turn will be discussed in-depth and representative examples will be provided.

## 6.1 Declarative Expressions

Cl is a logic-based language, supporting declarative queries, declarative rules for data definition (called passive rules) as well as declarative conditions for controlling imperative (active) rules and imperative operations. Logical languages are classically composed of two main syntactic categories: terms and formulas. Terms denote individuals in the respective domain of interpretation of the language. In the Chimera context this means that a CL term denotes either a value or an object. Formulas express propositions about individuals, being either true or false.

**Terms** are atomic or complex. Atomic terms include constants and variables; complex terms include functional terms built from constructors (set, list, record) or evaluable terms built by means of functions available in Chimera (attributes, selectors, predefined operators).

**Formulas** are either atomic, or complex as well. Atomic formulas are composed of a predicate symbol and a list of parameter terms; they include class, type, event, constraint, membership, and comparison formulas. Complex formulas are constructed from atomic formulas by means of connectives expressing conjunction and negation.

Formulas are evaluated over a database state according to the classical assumptions of first-order semantics, which we do not elaborate on in this document, but take for granted.

In order to avoid syntactically valid formulas which denote infinite set of instances, we impose that formulas be range-restricted. When compared to other *Datalog-like* logic languages, CL offers a richer collection of mechanisms for building terms and formulas; these however enable to implement and use all the features available through CM, which is a rich object-oriented data model.

## 6.2 Passive Rules

Passive rules are one of the key concepts of CL. They are used for declaratively defining class instances or attribute values, and for the implementation of views and constraints. A **passive rule** is an expression of the form:

```
Head  <-  Body
```

where the *head* is an atomic formula, the *body* is an arbitrary formula, and each variable in the head occurs in the body. Rules are stratified with respect to sets and negation, thereby ensuring that the computation of their fixpoint converges to a unique, minimal model. These limitations do not allow us to express certain semantics by means of rules (for instance, we exclude locally stratified rules); moreover, our choice for a "standard" semantics for stratified rules excludes the possibility of choosing other semantics for more general rules, such as inflationary semantics. However, our design choice is motivated by the fact that stratified rules satisfy the requirements of most applications and have a more intuitive meaning compared to nonstratified rules.

## 6.3 Procedural Expressions

CL does not aim at being a full-fledged programming language. However, there is the need for expressing certain database-related imperative actions under the control of the database system, and thus for incorporating a certain degree of procedural syntax into CL.

Procedural expressions in CL are composed of primitive database statements, i.e., of updates, queries and operation calls. The only general means of forming more complex statements is to build chains of primitive statements. Due to the database nature of these primitives, CL provides two different chaining operators: one for passing sets of variable bindings from one component statement to the other (the *sequence* operator) and one for passing individual bindings (the *pipe* operator). In addition, the UFI mode of interaction provides two syntactic variants which express iteration over sets of objects either explicitly or implicitly.

## 6.4 Queries, Updates, and Transactions

Chimera supports rather conventional notions of query, update, and transaction.

**Queries** in Chimera are either submitted from a *user-friendly interface* (UFI) or from an *application program interface* (API). Queries supported from in UFI mode include *display* and *find*; queries supported from an API include *select* and *next*.

In essence, all four query primitives are very similar; each of them consists of a Chimera formula F and a target list T. In all cases, the formula F is evaluated over the current state of the database, returning either individual bindings to the variables in T (in the case of *find* and *next*) or the set of all the bindings to these variables (in the case of *display* and *select*). Thus, the rationale of query primitives is to provide both tuple-oriented and set-oriented access from both kinds of interfaces.

**Updates** in Chimera support object creation and deletion, object migration from one class to another, state change or change of persistency status of objects, and value class population and modification. This collection of primitives enables all possible class and persistency updates which can be envisioned in Chimera, with the only exception of turning an object from persistent to temporary.

Finally, Chimera supports a conventional notion of **transaction**, where user-controlled *commit* and *rollback* primitives allow to either atomically execute all changes defined inside the transaction boundaries, or to restore the transaction's initial state.

## 6.5 Active Rules and Operation Implementations

Similarly to the way how passive rules serve as a declarative means of implementing certain Chimera concepts, there are two categories of imperative constructs for implementing other concepts in Chimera as well. In early versions of Chimera there was just one uniform such mechanism, called active rule, used to implement both triggers and operations. Due to the rather strong differences in syntax and semantics, we now call only those constructs implementing triggers an *active rule*, whereas operations are defined by means of an *operation implementation*.

Both categories of constructs contain a procedural CL expression as their "body", expressing a sequence of database actions that are to be executed. In both cases, trigger as well as operation, this execution only takes place, if a certain declarative (pre-)condition is satisfied over the current state of the database. The difference between the two categories of constructs is the style of invocation. In case of operations, an explicit invocation (operation call) is required and control is locally transferred from one operation call to the next. In case of triggers, invocation is implicit, controlled by a system component monitoring actions and determining appropriate reactions. Triggers have a rather complex semantics and will thus be explained in more detail in a separate section **??**. The semantics of operation calls is a rather straightforward one and has already been dicussed in Section **??**.

# 7 DECLARATIVE EXPRESSIONS

Declarative expressions are ubiquituous in Chimera:

- Every query in CL essentially is a declarative formula.
- The body of every passive rule (used for implementing derived classes and attributes as well as constraints) is a declarative formula.
- The condition part of every active rule is a declarative formula.
- The condition part of every operation implementation is a declarative formula, too.

In the following sections, we will introduce the various constituents of declarative Chimera expressions in detail.

## 7.1 Terms

Terms in a logical language can be classified into four main categories:

- atomic terms
  - constants
  - variables
- complex terms
  - constructs
  - evaluable terms

In this section, we introduce syntax and semantics of the different categories of terms.

**Constants** in CL are predefined atomic values of Chimera coming from the six basic value types of CM: oid, integer and real numbers, characters and strings of the respective host language, the Boolean values *true* and *false*, and the polymorphic null value *null*. They denote themselves.

**Variables** in CL are denoted by *variable names*; in Chimera (as in many other logic programming languages), variable names are finite alphanumeric strings beginning with an uppercase letter. All variables must be declared by means of class or type formulas (see below).

A variable X declared in a class formula *c(X)* ranges over the extent of the class *c*, i.e., X can be *instantiated* during the evaluation of the formula containing X by instances of the class $c$ only. Variables ranging over value classes are bound to individual values; variables ranging over object classes are conceptually bound to individual objects; internally, object-valued variables are bound to OIDs. A variable X declared in a type formula *t(X)* ranges over the (possibly infinite) set of allowed values for that type.

As objects are abstractions, users may never access the bindings of object-valued variables at the interface level, but may only use such variables for expressing relationships between objects inside a formula. When users want to retrieve information about individual objects, they should instead retrieve the values of particular attributes which uniquely characterize it. In particular, attributes constituting a *key* have such a property.

**Constructs** in a logical language are functional terms built from constructor functions, i.e., functions mapping a list of terms to a new, complex term containing each of the argument terms as its components. According to the three different type constructors of CM there are three different term constructors in CL:

- Set terms are constructed using curly brackets { and }.
  If $t_1, ..., t_n$ are terms, then $\{t_1, ..., t_n\}$ denotes the set consisting of the terms denoted by the $t_i$.
- List terms are constructed using square brackets [ and ].
  If $t_1, ..., t_n$ are terms, then $[t_1, ..., t_n]$ denotes the list consisting of the terms denoted by the $t_i$ in the specified order.
- Records are constructed from round brackets ( and ) and colons : separating the label of each component from the respective component entry.
  If $t_1, ..., t_n$ are terms and $l_1, ..., l_n$ are labels, then
  $(l_1 : t_1, ..., l_n : t_n)$ denotes the record with components $l_i : t_i$.

Constructs may only occur as values of attributes or as parameters of operations defined over a structured value domain, e.g.:

```
X.divisors = {2,3,4,6}.
Y.pay = (amount:2500, unit:"ECU")
Y.birthday = (day:20, month:7, year:1953)
Y.birthday.month = 3
T.authorList = ["Ceri","Gottlob","Tanca"]
```

**Evaluable terms** are constructed from an evaluable function symbol and a list of parameters, which are terms again. Evaluable functions return a value which is in one way or the other related to their input parameters, but does not have to be composed from these input parameters (as is the case for constructors). Evaluable functions are represented in CL by finite, alphanumeric strings starting with a small letter, or they are special symbols representing a particular predefined operator on values (such as $+$ or $*$). There are three kinds of evaluable functions in CL:

- attributes
- selectors
- operators

**Attributes** are unary functions applied in postfix dot notation, as shown in the above example where the left hand side of each of the five equations constitutes an attribute term [9].

**Selectors** are also unary functions, accessing individual components of a structured value. Individual components of a record are normally accessed by means of the corresponding label. If for a particular component no explicit label has been introduced, implicit labelling by integers indicating the relative position of the respective component within the record is assumed. For illustration, let the variable T stand for the complex term (empl: "Smith", salary: 100000, 1992). Then T.empl denotes the string "Smith", T.salary denotes the number 100000, and T.3 denotes the (unlabeled) third component value 1992. Individual components of list terms are accessible by means of integer selectors indicating relative positions as well, e.g., [a,b,c].2 denotes the character b. Attributes and selectors are both written in postfix notation, because the state of every object is regarded as a record, the labels of which are the individual attributes of the object class in the order they have been introduced in the respective class definition.

**Operators** are predefined functions of various arities applying to values. The following predefined operators ought to be supported by every Chimera implementation:

- *Arithmetic operators*: addition (+), subtraction (-), multiplication (*), division (/).
- *Aggregate operators* applicable to sets and lists (in some cases restricted to integer and real elements): *min, max, sum, count, avg*.
- *Set operators*: union(+), difference(-), intersection(*).
- *List operators*: concatenation (//), head (hd), tail (tl).

Additional operators may be provided by individual implementations. Examples of evaluable terms composed from operators are:

```
X.age + 3
(X.salary.amount * X.salary.unit.exchangeToEcu) * 12
count(X.authors)
hd([a,b,c,d]) // tl(X)
X.friends + Y.friends
```

[9] Attribute terms can be applied recursively, yielding to path expressions.

In addition to the aggregate operators defined above, CL supports also more **complex aggregate operators**, obtained by applying an aggregate operator (*avg*, *sum*, *min*, *max*, *count*) to pairs (T where F), where T is a term and F is a formula (see Section **??** below). T and F share common variables C in such a way that these variables must not occur outside the aggregate complex function operator, and any variable in T must also occur in F. F may have additional variables A. Such operators have the following semantics: they are applied to the multiset of bindings of the variables C for each distinct binding of the variables A computed by evaluating the formula F; the variables A must either be bound before the evaluation of the aggregate function or occur within a positive subformula of F to be bound by the evaluation. A null value is returned if the evaluation of F fails. Examples are:

```
avg(X.age where person(X))
sum(X.salary where X in Y.members)
sum(X.salary where employee(X), manager(Y),
                   Y.name = "Mauricio Lopez",
                   X in Y.dependents)
```

(note that the variable Y in the second example is bound before the evaluation of the aggregate function, see the active constraint in Section 5.3; in the third example, instead, the variable Y is bound by the evaluation). These complex operators enable aggregate computations on multisets of values arbitrarily extracted from sets of objects belonging to the same class, and are similar to aggregate functions evaluated on relational attributes, supported by relational query languages.

External names are CL terms, too. They can be viewed as parameterless functions, denoting oids. Thus every occurrence of an external name within a declarative CL expression ought to be viewed as functional term, too.

Finally, CL supports a special function *choose(S1,N,S2)* which should be evaluated with the first argument $S1$ bound to a set of elements of arbitrary type $T$ and with the second argument $N$ bound to an integer; the third argument $S2$ is bound by the evaluation to a set of at most $N$ elements of type $T$, nondeterministically chosen among the elements of $S1$.

## 7.2 Formulas

Like terms, formulas of a logical language are either atomic or complex. Atomic formulas are composed of a predicate symbol and a list of parameter terms. Predicates are either finite, alphanumeric strings starting with a small letter (like function symbols) or special characters such as $<$ or $==$ representing special predefined relations. There are five kinds of atomic formulas in CL:

- **Class or type formulas**, built from class or type names (including view names) representing unary predicate symbols, e.g., *person(X)*, *marriage((X,Y))*, or *com-*

*plex((re:X,im:Y))*. These formulas are used for declaring the type of a variable within a complex formula.

- **Constraint formulas**, built from constraint names representing predicate symbols, e.g., *tooLowIncome(X)*. These formulas are used in the head of passive rules which implement constraints (see below).

- **Event formulas** declaratively referring to the parameters of an event observed. Event formulas can be used within the condition part of an active rule only and are explained in more detail in the section on triggers **??**.

- **Comparisons** built from predefined, binary predicate symbols $<$, $<=$, $>$, $>=$ and the two equality symbols $=$ and $==$, e.g., $2 < 3$.

- **Membership formulas** built from the predefined, binary predicate symbol *in* used for accessing elements of a set or list term, e.g., *X in* $\{2,3,4\}$ or *Y in X.children* or *Y in [56,45]*.

A second form of membership formula is offered for expressing the fact that a value or an object belongs to multiple classes. For doing so, a class name can be used on the right-hand side of the "in" operator instead of a set or list term, e.g., *X in employee*. The meaning of such formula is the same as that of the corresponding class formula, e.g., *employee(X)*, with the exception that membership formulas do not establish types for variables. Such membership formula is applied when an object may belong to more than one most specific subclass within a subclass hierarchy, e.g., when we want to select persons which at the same time belong to subclasses *male* and *employee* and we like to use *employee* as the unique type:

```
employee(X),X in male, X.salary=5
```

In the above example, X is is of type *employee*, and any property (attribute, operation, constraint) of *employee* can therefore be referenced through X.

Classically, there are two fundamentally different notions of equality that are supported by object-oriented systems:

- **Equality by identity** means that two compared terms, ranging over an object class, denote the same object. Equality by identity is possible only between terms which range over the same class or over two classes related by inheritance.

- **Equality by value** means that two compared terms denote objects with the same attribute values (but not necessarily the same OIDs); this form of equality can be further refined in two ways:

  - **Shallow equality** considers all direct attributes (including those represented by means of OIDs) of an object which are either directly defined on the respective class or inherited from superclasses.

- **Deep equality** considers, in addition to the above attributes, all attributes of objects which are recursively reached by means of OID-based references.

In CL, the equality symbol = denotes identity, whereas the symbol == denotes shallow equality. Deep equality comparisons have to be expressed explicitly on a case by case basis by means of formulas. As an example, assume the following object class definitions:

```
define object class person
   attributes name:string(20)
              spouse:person
end

define object class student
   superclasses  person
end
```

Then, the following three examples impose identity, shallow equality, and deep equality on two variables X and Y ranging over *student* and *person*, resp.:

```
student(X), person(Y), X=Y
student(X), person(Y), X==Y
student(X), person(Y), X==Y, X.spouse.name=Y.spouse.name
```

(Note: the above examples use compound conjunctive formulas, see below).

Complex formulas are constructed from atomic formulas or negated atomic formulas by means of conjunction, expressed by the "," connective; quantifiers remain implicit, i.e., all variables are assumed to be implicitly quantified in the same way as in Datalog [10].

It is well-known that certain syntactically valid formulas may be satisfied by an infinite set of instances. As infinite answers to queries (or infinite extensions of rule-defined classes) are semantically useless, the usual way out is to identify a decidable subset of all possible formulas which is guaranteed to be satisfied by finite sets of instances if applied to a finite database; such formulas are called **range-restricted**. All valid CL formulas must be range-restricted.

Before introducing range-restricted formulas we define the notion of a range-restricted variable for which it is guaranteed that it can only be bound to a finite set of values.

A variable X is range-restricted in an arbitrary formula F, if

- X occurs in at least one class, constraint, or event formula of F,

[10] Use of explicit quantifiers and/or disjunction is envisioned as one of the directions for extending Chimera.

- or X occurs in any equation X=T or T=X in F where T is a ground term, or where all variables in T are range-restricted in F

  (if T is an aggregate term *agg*(T1 where F1) all local variables of T must be range-restricted in F1 and all global variables range-restricted in F),

- or X occurs in any membership formula (X in S) in F where S is a ground set or all variables in S are range-restricted in F.

The variable X is range-restricted in each of the following atomic formulas:

```
X = 1
X in {1,2,3}
X in Self.dependents
X = sum(E.salary where E in D.members)
```

(provided D and Self, resp., are bound somewhere in the syntactic context, too.)

A formula F is range-restricted iff all variables of F are range-restricted in F. The following are examples of range-restricted formulas:

```
integer(X), X in {1,2,3}
employee(X), employee(Y), not X=Y, X.salary>Y.salary
employee(X), real(Y), Y=0.8*X.gross-salary
employee(X), integer(Z), integer(W), Z=X.salary, W=Z+500
employee(X), modified(employee.salary,Y), X.salary<0.5*Y.salary
```

whereas the following is not range-restricted:

```
employee(X), integer(Y), X.salary>Y
```

Some variables may be range-restricted by their signature definition, e.g.:

```
define view marriage: record-of(husband:person,wife:person) as
   {marriage((husband:X,wife:Y)) <- X.spouse=Y, X.sex=male}
end
```

In this case, class formulas *person(X)* and *person(Y)* are implicitly added to the formula in the rule's body; an explicit repetition of the above class formulas could, however, improve the rule's readability (especially if signatures and implementations are separately defined).

Note that the special variables Self and Class are implicitly range-restricted.

An important restriction imposed on complex formulas is that for every variable there may be at most one type/class formula in order to retain assignment of a unique type for each variable, thus facilitating type checking. Thus, a formula like:

```
male(X), employee(X), X.age>45
```

is illegal and has to be reformulated by means of a membership formula replacing one of the two class formulas, e.g.,

```
male(X), X in employee, X.age>45.
```

# 8 PASSIVE RULES

A passive rule is an expression of the form:

```
Head  <-  Body
```

Where:

- The *head* is an atomic formula.
- The *body* is an arbitrary formula.
- Each variable in the head occurs in the body.

Passive rules are means for defining the extent of a subclass by "filtering" out certain objects from the superclass based on a declarative condition and for defining the value of an attribute intentionally, i.e. without enumerating every instance separately; these rules are *targeted* to specific classes [11]. Furthermore, passive rules are the (only) means of defining views and constraints.

Examples of passive rules are:

- for a subclass:

```
engineer(X) <- employee(X), X.profession="engineer"
```

- for a view:

```
works-for((X,Y)) <- employee(X), employee(Y),
                    Y=X.department.boss
```

[11] Note that attributes defined by the list constructor cannot be implemented by means of passive rules, because declarative expressions in the body of passive rules cannot determine the order of elements in a list.

- for attributes:

```
Self.salary=200000 <- engineer(Self), Self.age < 35

Y in Self.children <- person(Self), person(Y),
                      Y.father=Self
```

Any concept may be defined by several rules, expressing different cases of the overall definition. This multi-case form allows incremental definition of a concept, as in:

```
X.salary=20000 <- engineer(X), X.age < 35;
X.salary=25000 <- engineer(X), X.age >= 35;
X.salary=50000 <- manager(X), leads((X,Y)), group(Y);
X.salary=60000 <- manager(X), leads((X,Y)), department(Y)
```

where "leads" is considered as a binary view.

## 8.1 Stratification and Safety

For each set of rules recursively referring to each other, we impose the restriction that this rule set has to be *stratified* with respect to both negation and sets.

Intuitively, in a stratified rule set no recursive cycle may involve negation or sets. An example of a rule set with stratified negation is:

```
Y in X.descendants <- person(X), Y in X.children
Y in X.descendants <- person(X), Z in X.children,
                      Y in Z.descendants
Y in X.non-descendants <- person(X), person(Y),
                          not (Y in X.descendants)
```

An example of a rule set with stratified use of sets is:

```
Y in X.descendants <- person(X), Y in X.children
Y in X.descendants <- person(X), Z in X.children,
                      Y in Z.descendants
X.number-Desc = N  <- person(X),
                      N = card(X.descendants)
```

In addition to stratification, we expect each set of rules to be *safe*. In a safe rule, every variable in the head of the rule occurs in at least one positive atomic sub-formula of the rule body, thus guaranteeing that bindings for variables can be computed from those data sets addressed in the respective rule only. In addition, we expect the rule bodies to be range-restricted, guaranteeing that variables local to the rule body can be properly evaluated, too. An example of an unsafe rule is:

```
male(X) <- not female(X)
```

Such a rule would imply that every value or object not belonging to the class *female* belongs to the (derived) class *male*. This is, of course, semantically unintended and is due to a missing specification of the range of X. The above example rule could be turned into a safe one by explicitly restricting X to range over class *person* only:

```
male(X) <- person(X), not female(X).
```

This form of safeness is called *local*, as it can be decided by analyzing rules one by one independently of each other.

Passive rules defining attributes, constraints, or class instances are targeted to the respective class. A special variable *Self* is implicitly ranging over the instances of the target class. The following rules define two derived attributes using the *Self* device:

- Salary is computed from the gross-salary;
- Manages includes the set of employees which are recursively (i.e. directly or indirectly) managed by one employee.

```
define object class employee
 attributes  name: string(20)
             gross-salary: integer, derived
             dependents: set-of(employee)
             salary: real
             manages: set-of(employee), derived
end

define implementation for employee
 attributes Y = Self.salary <- real(Y), Y=0.8*Self.gross-salary;
            Y in Self.manages <- employee(Y), Y in Self.dependents;
            Y in Self.manages <- employee(Z), Z in Self.dependents,
                                 employee(Y), Y in Z.manages
end
```

Similarly, a special variable *Class* is implicitly ranging over a target class, to be used in the definition of derived c-attributes. The following rule defines a derived class attribute (the *average age* for the *person* class) by using the *Class* device:

```
define object class person
  ....
  c-attributes averageAge: integer
  ....
end

define implementation for person
  ....
  c-attributes Class.averageAge=Y <- integer(Y),
                                     Y=avg(X.age where person(X))
  ....
end
```

Beyond the limitations on passive rules (such as range-restriction of formulas, safety of passive rules, stratification of passive rule sets), rules defining implementations must be type-compatible with the signatures they refer to. Therefore:

- The head of a passive rule used in the implementation of either an attribute, a constraint, the population of a class, or a view must be compatible with its respective signature. In particular:
  - In the case of an attribute, the head consists of either an equality or a membership formula where one of the terms is the respective attribute term, so that bindings are generated for that attribute.
  - In the case of object class population, view or constraint definition, the head is a class or constraint formula, respectively.
- Event formulas mentioned in triggers which are targeted to a class can refer only to query or update primitives of that class.

# 9 QUERIES

In Chimera, two different sets of query primitives are provided, one for each of the two interaction modes mentioned earlier. In the *UFI* mode, queries are issued through a user-friendly stand-alone interface, probably a window-based tool interacting with a Chimera interpreter directly. In the *API* mode, Chimera queries are considered as special procedure calls embedded into an application programming language of a particular Chimera implementation.

In essence, all four query primitives are very similar in syntax. Each of them consists of a Chimera formula F and a target list T, listing variables and selector terms. The basic semantic policy is that formula F is evaluated over the current state of the database, returning either individual bindings to the T-variables or a set of all bindings that can be found (or computed).

## 9.1 UFI Queries

The UFI queries allow to display the content of the database or to extract information into temporary variables.

### 9.1.1 Display

The **display** operation is a set-oriented query primitive in the user-friendly interface. A call *display(T where F)* causes all bindings to T resulting from evaluation of F to be displayed in a suitable style on the screen (the respective style of display being left to the individual implementation of the UFI). An example is:

```
display(X.name where marriage((husband:X,wife:Y)), Y.name="Teresa")
```

A display is executed by first evaluating the formula F, resulting in the computation of a set of bindings for the variables occurring in F. The target list T refers to some of these variables; however, when variables assume OID values, the target list cannot refer to them directly, but it must instead include the *projection* of the corresponding objects on suitable attributes or attribute components; these projections should not include OIDs [12].

The answer actually displayed by a query can be a multi-set of instances of the target list, rather than a set. As an example consider the query:

```
display(X.name,Y.make where person(X), car(Y), Y.owner=X)
```

If a certain person owns more than one car of the same make, the same pair [Name,Make] will be displayed twice.

Some variables occurring in F may not appear in T, e.g. the variable Y in the following example:

```
display(X.name, Z where person(X), car(Y),
                        string(Z), Y.owner=X, Z=Y.make).
```

In this case the variable(s) missing in the target list are assumed to be implicitly existentially quantified; this means that an implicit grouping is applied when constructing the answer. In the example, each pair [Name, Make] would appear only once, independent of the number of Y instances associated with the same X and Z. The possibility of omitting

[12] We recall that OIDs cannot be *printed*, as they carry internal information which is not relevant to users; hence the above restriction. A possible alternative, considered by some object-oriented databases, is to design a key attribute as the one that should be printed instead of the OID; this solution is currently left as an option for testbeds.

variables from the target list allows to display the result of computation of aggregate functions, e.g., the following query displays for each employee the average salary of his dependents:

```
display(X.name,Y where employee(X), integer(Y),
                       Y=avg(Z.salary where employee(Z),
                                            Z in X.dependents))
```

It is also possible to display the target list according to user-controlled ordering, by means of an additional parameter introduced by the keyword *order-by*. The parameter consists of a sequence of ordering clauses of the form:

```
ascending(T1)
descending(T2)
```

Where T1, T2 are variables or terms evaluable for each variable binding extracted after the evaluation of F such that their lexicographical order is well defined (of course, they cannot be OIDs); for example:

```
display(X.name,X.age where person(X),X.age>18
                     order-by ascending(X.name),descending(X.age))
```

### 9.1.2 Select

In addition to the *display* primitive, the UFI interface supports also a *select* primitive, which is used to extract individual bindings into atomic temporary variables. A call *select(T where F)* causes bindings to T resulting from evaluation of F to be associated to the variables which are named in T; this enables their use in subsequent primitives which are submitted from the UFI (primarily update primitives, see Section **??**). Examples of select are:

```
select(X where employee(X), X.salary<5000)
select(X,Y where person(X), car(Y), Y.owner=X)
```

The evaluation of a *select* primitive is identical to the evaluation of a *display*: the formula F is evaluated, resulting in the computation of bindings for the variables occurring in F; next, the target list is computed from these bindings. However, the target list may include OID values in this case. Variables which are named in the target list become *visible* in the UFI and can be used in building transactions; this will be further explained in Section **??**.

An additional style is possible for the UFI, motivated by the wish of making the implicit iteration over bindings produced by the select statement more explicit. In this case, a **foreach** operation is introduced, which defines the scope and iteration of variables extracted into the target list. The syntactic format of this query primitive is *foreach(T where F) do S*, where S is a sequence of procedure calls which may use variables in the target list T as input parameters. This is equivalent to specifying *select(T where F)* and then listing the procedure calls *S*. The rationale of allowing both styles will be further discussed in Section **??**. An example is:

```
foreach(X where postalCode(X),X<999) do delete(postalCode,X)
```

where first all bindings for X (i.e., values to be deleted) are obtained by evaluating the formula F (i.e., all values less than 999) and only then the deletion of the respective postal codes is performed.

Two variants of the *select* primitive allow to extract collective bindings into structured temporary variables, built by means of set or list constructors, respectively. Such structured temporary variables are introduced by the keywords *into-set* or *into-list* respectively, which are followed by a variable name; that variable name can be used in the UFI transaction in order to refer to the collection of extracted bindings. For instance, in:

```
select(X into-set  S where employee(X), X.salary<5000)
select(X into-list L where employee(X), X.salary<5000
                     order-by ascending(X.salary))
```

Variable S denotes in the UFI the set of extracted employees, while variable L denotes the list of extracted employees in ascending salary order.

## 9.2 API Queries

In the embedded version of CL, queries result in answers to be returned as bindings to program variables of the embedding host language program. The passing of results to program variables is done by means of an assignment statement of the host language (which we express in this text for illustration purpose in the classical := style without fixing any particular syntactic solution this way). Thus, Chimera queries will be expressions of the host language evaluated on the right hand side of the assignment.

Set-oriented queries are expressed by means of the **select** function, to be used in an assignment to a set-valued variable S: *S := select(T where F)* or list-valued variable L: *L := select(T where F order-by O)*. The types of the program variables have to be compatible with the types obtained as set/list of the target list record T in this case.

Individual solutions are obtained using a cursor mechanism expressed by an assignment of the form *next(T into V where F)* or *next(T into V where F order-by S)* using the Chimera **next** function; this function must also return an error condition which becomes true when the cursor is emptied, in a manner that is specific to the particular embedding into a "host" programming language. The type of the program variable V has to be compatible with the type of the target list T in this case; thus, T must contain attributes which do not include set or list constructors.

The following two examples illustrate the use of *select* or *next* queries. In the first case, the type system of the "host" programming language (e.g. Peplom) supports sets, with no *impedance mismatch* with Chimera; thus, $S$ is a set variable directly assigned to the result of the *select* query:

```
S := select(X.name,X.empNum where employee(X),X.mgr.name="Jean-Marie")
```

The second example illustrates a possible API for the "C" programming language [13], where the function *next* returns a negative error code when the cursor is emptied; a suitable "C" structure V is assigned to the bindings produced for T after each *next* call, until the cursor is emptied.

```
struct V-type
  { char name(20);
    int  num;
  } V;

while (true)
{
  int I;

  I=next(X.name,X.empNum into V
         where employee(X),X.mgr.name="Jean-Marie"
         order-by ascending(X.empNum));
  if (I<0) break;
  printf(V.name,V.num);
}
```

## 9.3 Support for External Procedures

External procedures that can be called from inside Chimera need to be defined, so that their interface becomes available for use in either queries (*apply* primitive) or reactions. The format of such a definition could be:

```
  define external procedure procedure-name(list of parameters) end
```

Procedures written in the "host" programming language perform any application-related function; however, in order to inspect the content of the database or to modify the state

[13] This example is by no means prescriptive of how the actual implementation of *next* should be done in a "C" language interface.

of the database, they must in turn use the API query primitives and update primitives, defined below. A further analysis of the interaction between Chimera and the "host" programming environment will be developed in the context of testbed implementations.

# 10 UPDATES

Chimera provides a number of predefined procedures for performing basic database manipulations. There are five main data manipulation operations in Chimera, which we will discuss in turn:

- Object creation and deletion;
- Object migration from one class to another;
- State change of an object, i.e., attribute modification;
- Change of persistency status of an object;
- Value class population and modification.

## 10.1 Object Creation and Deletion

Persistent objects are created by means of the **create** operation. The general format of a *create* command is *create(C,T,O)*. A class name $C$ and a record term $T$ are expected as input, resulting in the creation of a new object of type $C$ with state $T$; $O$ is the OID of the newly created object, returned as output. When $C$ is a subclass in a generalization hierarchy, $T$ must include all attributes defined in all the superclasses of $C$ [14]. The operation is successful only if the structure of $T$ and the types of all components of $T$ are compatible with the attribute signatures defined for $C$.

Temporary objects are created by means of the *create_tmp* operation, which does not place the new object into persistent store, but keeps it in the workspace of the current session.

As an example for object creation, assume that the object class *project* is defined as follows:

```
define object class project
  attributes        name:string(50)
                    acronym:string(10)
```

[14] Consider two classes $C1$ and $C2$ such that $C2$ is a derived subclass of $C1$; an instance can be inserted into $C2$ either by means of a *create* primitive on $C2$, or be automatically inserted into it by a passive rule after a create on $C1$; in the latter case, the extensional attributes which are special to $C2$ are initially set to null.

```
                         number:integer
                         duration:integer
                         commencement:date
                         members:set-of(employee), derived
   end
```

A syntactically correct creation command resulting in the creation of a new (persistent) instance of *project* is:

```
  create(project,
         (name:"Intelligent DB Environment for Advanced Applications",
          acronym:"IDEA",
          number:6333,
          duration:4,
          commencement:(day:1,month:6,year:1992)),X)
```

where derived attributes are omitted.

Objects can be deleted by means of the **delete** operation, taking a class name $C$ and an object identifier $O$ as input parameters, the format being *delete(C,O)*. The effect of a deletion command is to delete the object identified by $O$ from its class $C$ and also from all other classes $C'$ to which it belongs due to inheritance.

Moreover, an implicit referential integrity constraint is established: whenever an object $O$ is deleted from class $C$, the OID of the deleted object is dropped from all attribute values which refer to $O$ in other objects $C'$. Therefore:

- If $C$ is the type of an atomic attribute of $C'$ (either defined individually or a record component), its value is set to *null*.
- If $C$ is the type of the element of an attribute of $C'$ built by means of set or list constructors (e.g., $set - of(C)$), then $O$ is deleted from the set or list; this may result in producing an empty set or list.

An example of deletion is:

```
   delete(project,X)
```

with X bound by the declarative expression

```
   project(X), X.name = ''STRETCH''.
```

All references to the deleted project are also set to *null* or deleted.

## 10.2 Object Migration

If an existing object in a given class is to be inserted to a more specific subclass or inversely moved back to a more general superclass, the OID of the object (which means its "identity") remains unchanged, but only those attributes that exclusively belong to the more specific class have to be added or removed from the objects state. As an example, consider the promotion of an employee to the manager level (*manager* being a subclass of *employee* with some extra attributes like *extra-benefit* or *official-car*). The inverse - admittedly rather unlikely, but nevertheless not completely excluded - case is the transfer of a manager back to normal employee status (which means the loss of extra benefits and official cars).

CL offers another pair of operations for the two forms of object migration. Specialization of an object to a subclass is done by means of the **specialize** operation, taking two class names $C1$ and $C2$, an object identifier $O$ and a record term $T$ as its input parameters. The result of executing the command *specialize(C1,C2,O,T)* is that object $O$, initially belonging to class $C1$, is inserted into class $C2$ as well, and that its state is extended by concatenating its old state (containing values for those attributes that now are inherited ones) with $T$, specifying values for those attributes that are special to $C2$. Of course, $O$ remains an instance of $C1$ due to subclassing.

An example of use of specialization, related to the signature definitions of object classes *employee* and *fiatEmployee* of Section **??**, consider:

```
specialize(employee,fiatEmployee,X,(tax:32))
```

where X is bound by a declarative expression.

The inverse process is expressed by means of the **generalize** operation, which takes only three parameters: two class names $C1$ and $C2$ again, and an identifier $O$. The result of performing *generalize(C1,C2,O)* is that object $O$ is removed from class $C1$, while it remains a member of the superclass $C2$ of $C1$; therefore, all attributes special to $C1$ are dropped. For example, consider:

```
generalize(employee,person,X)
```

where X is bound by a declarative expression; as effect of this operation, attributes *emplNr, mgr, salary*, and *dependents* are dropped for the particular object identified by the OID value bound to X.

Explicit specialization and generalization operations between classes $C1$ and $C2$ is not possible if $C1$ is derived from $C2$. When a class $C$ inherits from multiple classes $C1, ..Cn$, an instance of any of the superclasses $Ci$ can be inserted into $C$ by means of a specialization only if that instance is already included into all classes $C1, ..Cn$ [15].

[15] However, instances of $C$ can be directly created by means of a *create* primitive, which requires the specification of the entire state of $T$ (e.g., all attributes defined in all superclasses of $C$).

## 10.3 Attribute Modification

Every attribute of an object has to be initialized (possibly with *null*) at object creation time. The initial values are specified within the state parameter of the respective create command. Later changes of an attribute value can be performed by means of the **modify** operation. The syntax of this operation is *modify(C.A,O,V)*, where A is an attribute of object class $C$, $O$ is an object identifier of an instance of $C$, and $V$ is the new value of $O$, replacing whatever old value had been in existence. A modify command returns an error if either the attribute is not compatible with the definition of $C$, or the identifier $O$ does not refer to an object in $C$, or if the value $V$ does not have the type required for $A$ values in class $C$. An example of modification is:

```
modify(employee.salary,Self,Self.mgr.salary)
```

## 10.4 Change of Persistency Status

Temporary objects can be made persistent by calling the **make_persistent** operation. Two parameters are required, a class name and an object identifier: *make_persistent(C,O)* causes the pstatus of $O$ in class $C$ to be changed from *temporary* to *persistent*. If $O$ already is persistent, the command will succeed but no action will be performed. If $O$ does not exist as a temporary instance of $C$, the operation will return an error.

## 10.5 Value Class Population

User-defined value classes may be populated by explicitly inserting (or deleting) individual values. This concept is similar to the enumerated types found in many programming languages, but causes explicit persistent storage to be generated for the extension of the respective value class. The CL operations for adding a predefined value to or deleting it from a user-defined value class are **add** and **drop**. Both operations expect the class name as first parameter, and the value to be added to / dropped from that class as second parameter. Recalling the value class *postalCode* of Section **??**, examples of additions and deletions are:

```
add(postalCode,3001)
drop(postalCode,X)
```

where X is again bound by a preceding declarative expression.

# 11 PROCEDURAL EXPRESSIONS

Procedural expressions are ubiquituous in Chimera, too:

- The operation code of every operation implementation is a procedural expression.
- Every transaction is a procedural expressions.
- The reaction part of every trigger is a procedural expression.

In the following sections, we will introduce the various constituents of procedural Chimera expressions in detail.

## 11.1 Procedure Calls

Procedures that can be called are either query primitives (*display* and *select*), or update primitives (as defined in Section **??**), or operations (Section **??**), or externally defined procedures (Section **??**). These procedures have typed input and output parameters; therefore, interaction among procedures is achieved by variables, of suitable type, which are produced as output parameters by given procedures and consumed as input parameters by subsequent procedures.

Output parameters are produced by:

- *create* primitives, returning OID variables of newly created objects.
- *select* queries, returning either individual or collective bindings.
- *operations* and *externally defined procedures*, returning output parameters.

Input parameters are consumed by:

- *update primitives* (including *create*, which may require input parameters for the state definition).
- *operations*, requiring an OID and possibly other input parameters, and *externally defined procedures*, requiring input parameters.

Clearly, the type of variables used as input parameters in a procedure call must be compatible with the type expected by the parameter's signature. All variables used as output parameters in a procedure call must be unbound.

## 11.2 Transaction Lines

Transaction lines are *chains* of one or more procedure calls. During the discussions on the semantics of transaction lines in Chimera, the need for an explicit distinction between a set-oriented and an object-oriented style of chaining has been identified. Therefore, CL offers two different connectors for composing chains of procedure calls. Apart from a set-oriented **sequence connector** one can also use an object-oriented **pipe connector**.

**Sequences** of procedure calls are constructed by means of the sequence connector which is denoted by ",". When a variable is defined by means of a *select* statement and then used by a given procedure $P$ (e.g., an update primitive), the procedure call is iterated over all the bindings produced by the select statement itself. This results in an object-oriented semantics of application of procedures (including update primitives). However, the UFI optimizer may analyze such a sequence to detect the *select* statement where a given variable is introduced, and then process each update primitive in a set-oriented way.

```
select(X where train-number(X), X>400),  // iteration of drop
drop(train-number,X)                     // applied to a value class

select(X where employee(X), X.mgr.name="Manthey"), // iteration of
modify(employee.salary, X, X.salary + 5000)       // modify applied to
                                                  // an object class

select(X where employee(X), X.name="Ceri"),  // example of use
X.raiseSalary(10)                                // of operations
```

The same chains can also be expressed by using as query primitive *foreach* rather than *select*:

```
foreach(X where train-number(X), X>400) do
    drop(train-number,X)

foreach(X where employee(X), X.mgr.name="Manthey") do
    modify(employee.salary, X, X.salary + 5000)

foreach(X where employee(X), X.name="Ceri") do
    X.raiseSalary(10)
```

The semantics of execution of sequences is well understood in the general case. Each procedure $P$ is iterated over the cartesian product of bindings determined for variables $V1, ..Vn$ previously introduced as output parameters. For example, in the following sequence:

```
select(X,Y where F1(X,Y)), P1(X),
select(Z,W where F2(Z,W)), P2(X,Y,Z)
```

- The sequence consists of 4 procedure calls (*select*, $P1$, *select*, $P2$) which are executed in sequence. Each select causes the evaluation (and storage) of a set of bindings, that can be later referenced by means of variable names.
- Procedure $P1$ is iteratively executed for each projection on the first argument of the set of bindings produced by the first *select*.
- Procedure $P2$ is iteratively executed for each element of the cartesian product of the set of bindings produced by the first *select* with the projection on the first argument of the set of bindings produced by the second *select*.

**Pipelines** are constructed by means of the object-oriented pipe connector which is denoted by "%". The semantic difference between this and the sequence connector is illustrated by the following example.

```
select(X where employee(X), X.mgr.name="Manthey"),
modify(employee.salary, X, X.salary + 5000)%
X.changeIncome(X.salary + 5000)
```

In order to better understand the example, assume that the call of the select statement bind the variable X to an implicit set of values XO. Note that XO could be explicitly built as a set variable as introduced in the formulation `select(X into-set XO where employee(X), X.mgr.name="Manthey")`. The pipe connector "%" corresponds to the following equivalent formulation using the explicit loop construct of Chimera:

```
foreach(X where X in XO) do
    modify(employee.salary, X, X.salary + 5000),
    X.changeIncome(X.salary + 5000)
```

If the sequence connector "," would be used instead of the pipe connector, then the "modify" statement would be applied to all elements of XO first, and only then the operation "changeIncome" would be called. Using the "foreach" style, an equivalent formulation would be:

```
foreach(X where X in XO) do
    modify(employee.salary, X, X.salary + 5000),
foreach(X where X in XO) do
    X.changeIncome(X.salary + 5000)
```

The motivation for introducing the pipe connector comes from the challenge of embedding programming languages, like Peplom, into Chimera. The issue was to find a natural way of passing bindings between selections and subsequent procedure calls in a transaction line. The possibility of applying two sequence connectors in Chimera complies with the set-oriented style of Chimera's declarative services as well as with the "guided by objects" style of Peplom. Using the sequence connector is very natural in a set-oriented context while using the pipe connector is very natural in standard programming languages.

**Transaction lines** are chains of one or more procedure calls where subsequent procedure calls are connected by "," (sequence) or by "%" (pipeline). Variables shared between subsequent procedure calls in a transaction line must be type compatible. Variables which are used for linking output to input parameters have a name and type which is assigned to them by the procedure calls where they are first introduced. All output variables introduced in the same line must have different names.

We expect that bindings between input and output parameters normally required by the UFI will be simple, as in the above examples; we also conjecture that each implementation of UFI interface will introduce restrictions wrt the general case. If a procedure is called with an unbound input variable, that procedure is not executed (but this is not considered an error).

# 12 TRANSACTIONS

A transaction in Chimera is a sequence of transaction lines which are presented to the Chimera database server, linguistically encapsulated by a (possibly implicit) *begin-transaction* and a *commit* or *rollback* statement. At the beginning of a UFI session or API application, a database is opened by means of an *opendb* primitive which indicates the database name; a database can be created by indicating a new name. A symmetric *closedb* primitive indicates that the database is released by the session or application. The notion of transaction in Chimera has three facets.

- It serves to define atomic units of execution from the viewpoint of reliability and persistency.
- It allows to encapsulate actions which are subject to particular modes for reactive processing.
- It enables to define interactions between query and update primitives, by means of either UFI variables or programming language variables.

This section illustrates each of these facets separately.

## 12.1 Transactions as Units of Persistency and Reliability

Chimera supports a rather conventional notion of transaction, whereby a collection of database changes can be committed atomically. A transaction is initiated by the *begin-transaction* primitive (that may be implicit in some interfaces); it is committed by means of a *commit* primitive, and aborted by means of a *rollback* primitive. Transactions are normally isolated from each other by means of a concurrency control system.

The effect of a successful commit primitive is to apply all schema and instance updates atomically; these are persistent. The effect of a rollback primitive is to suspend the transaction's execution and to restore the transaction's initial state.

## 12.2 Transactions in the UFI interface

A transaction submitted from the UFI is a *sequence of transaction lines*. In the design of the UFI interface, we recognized the need of interaction among calls; this interaction requires the definition and use of temporary variables, whose type, scope, and legal use needs to be precisely defined, as well as a precise definition of the semantics of procedure executions, in particular with respect to the iteration over variable bindings; this section attempts a first characterization.

We restrict the scope of a variable to one *transaction line*, syntactically recognized in the UFI (e.g., lines are separated by a suitable delimiter). The following is an example of a transaction. Transaction lines are separated by the ";" delimiter:

```
>
> begin-transaction
>
> select(X where train-number(X), X>400),   // iteration of drop
> drop(train-number,X);                    // applied to a value class
>
> select(X where employee(X), X.mgr.name="Manthey"), // iteration of
> modify(employee.salary, X, X.salary + 5000);      // modify applied to an
>                                                  // object class
>
> display(X where employee.exceedsMgrSalary(X));  // use of output
>                                                // parameters of
>                                               // constraints
>
> create(employee,(..),X),                      // connection of create
> create(employee,(..),Y),                     // statements; newly created
> create(project,(..workers:[X,Y]..),Z);     // employees X and Y are
>                                           // inserted into project Z
>
>
```

```
> select(X into-list L
>          where employee(X),                // creation of a list L;
>                X.mgr.name="Willem Jonker"  // use in the state of
>          order-by ascending(X.salary)),   // a create object
> create(project,(..workers:L..),Z);
>
> select(X where employee(X), X.name="Stefano Ceri"), // example of use
> X.raiseSalary(10);                                   // of operations
>
> commit
```

## 12.3 Transactions in the API interface

Also a transaction submitted from the API is a *sequence of procedure calls* for the Chimera database; however, the interaction between calls is normally regulated by the use of constants and variables defined according to the type system of the embedding programming language. We recall that basic types in each Chimera testbed are compatible with those of the embedding programming language.

Further, transaction execution can be controlled by means of conventional *control statements*, which determine alternative flows of execution based on conditional or iterative statements. A transaction must be *well-formed*, namely, along each possible flow of execution after the begin-transaction statement there must be exactly one command forcing either the commit or the abort of the transaction.

# 13 TRIGGERS

**Chimera** supports *set-oriented triggers*, which are activated by database operations and perform reactive computations. The distinguishing choice of **Chimera** is to map an object-oriented data model with set-oriented triggers, e.g., triggers responding to collective operations.

Triggers in Chimera follow the *event-condition-action* paradigm of active databases:

- *Events* correspond to database accesses for retrieval or manipulation.
- Each *condition* is a declarative formula, to be evaluated in the state before activation of the reaction.
- Each *reaction* is a chain (sequence or pipeline) of one or more procedure calls, which can perform any computation on the database.

**Events** are a uniform interface for defining patterns of actions the observation of which can trigger a reaction. Such patters can be:

- *Queries* performed over object classes. A query is any retrieval operation which occurs during the evaluation of a term in the context of a passive rule. Events are denoted by the name of the target of the query (either a class or an attribute of a class):

```
query(class-name)
query(class-name.attribute-name)
```

- *Updates* performed over object classes. Events are denoted by the name of the resp. update operation and the target (class name, possibly attribute name) of this operation. Based on the CL update operations discussed in Section **??**, the following events are available in Chimera [16]:

```
create(class-name)
create_tmp(class-name)
delete(class-name)
specialize(class-name, class-name)
generalize(class-name, class-name)
modify(class-name.attribute-name)
make_persistent(class-name)
```

Each trigger is defined on a set of *triggering events*; the set is associated to a disjunction semantics (the trigger becomes active if *any* of its triggering events occurs). We exclude, for the time being, to support within **Chimera** a more complex event calculus. Note that we do not support triggers on *value classes*.

The **condition** is a declarative formula written in CL; it serves the purpose of monitoring the execution of the reaction part. The **reaction** is a chain of procedure calls; procedures can be either update primitives of Chimera (see Section **??**), or *display* primitives (see Section **??**), or operations, or externally defined procedures, or the transactional command *rollback*. Conditions and reactions may share some *atomic variables* that are used in order to relate them; in addition, conditions may use special formulas *occurred* and *holds* (defined in the sequel) in order to identify objects which have been the target of one of the above events.

Syntactically, active rules must be *safe*, that is, the variables occurring as input parameters of some procedures in the reaction part of the rule must be present in some positive literals of the condition part of the same rule (or be defined as output parameters of precedent procedures).

As an example, consider the trigger *adjustSalary* for class employee, requiring that, whenever an employee's salary is greater than his manager's salary, the employee's salary be set equal to that of his manager:

[16]Currently we have not yet considered the possibility of *time-based triggering events* in Chimera; this issue is left for investigation in WP2 T2 (reactive processing).

```
define trigger  adjustSalary for employee
  events        create
                modify(salary)
  condition     Self.salary > Self.mgr.salary
  actions       modify(employee.salary,Self,Self.mgr.salary)
end
```

## 13.1 Informal Description of Trigger Semantics

In this section we describe informally the semantics of triggers in **Chimera**. Triggers are activated by events such as queries or updates; therefore, multiple triggers can be activated as effect of the same event type. When one of the events of an active rule occurs, we say that the active rule is *triggered*. At given points of time (defined in the sequel) *active rule processing* is started. Rule processing consists of the iterative execution of *rule processing steps*, until certain termination conditions hold.

Each rule processing step consists of a sequence of *selection*, *consideration*, and possible *execution*. During trigger selection, one of the triggers which are triggered is chosen; the choice is explicitly influenced by priorities (also defined below). The selected rule is considered, by evaluating its condition; if the condition is satisfied (either because it produces the boolean value *true* or because it produces some bindings for the variables which are shared between the condition and the reaction), then the reaction is executed; this completes a rule processing step. The evaluation of the condition part of a trigger is similar to the production of a set of bindings through a *select* or *foreach* query primitive in the context of a transaction line; indeed, condition and reaction can be considered as part of the same transaction line from a transactional perspective.

Considered rules are no longer triggered, unless their triggering event occurs again as effect of some rule's reaction; note that a rule's reaction can trigger itself. In general, the set of triggered rules may change at each new active rule processing step. Active rule processing continues until no rule is triggered [17]; we call such a state a *quiescent state*.

## 13.2 Immediate and Deferred Triggers

Each trigger is defined as either *immediate* or *deferred*:

- The processing of *immediate triggers* is started immediately after the completion of the transaction line which has caused the triggering event.

[17] Clearly, the possibility of infinite rule processing due to chains of active rules that trigger each other indefinitely exists in Chimera; the study of sufficient conditions for ensuring termination is one of the topics of WP2 T2 (reactive processing). Regardless of this, each Chimera implementation should control the run-time execution of active rule processing, possibly by forcing the abort of the underlying transaction when a given threshold on the number of triggered rules is reached.

- The processing of *deferred triggers* is started at the end of the transaction, as part of the execution of the transactional command *commit*.

Whenever the execution of a transaction unit is completed, immediate triggers are processed according to the semantics described in the previous section, until a fixpoint is reached (called a *quiescent state* wrt immediate triggers); at commit time, active rule processing is applied to *all* triggers, until a *final state* is reached. The transactional command *savepoint* forces rule processing over all triggers (including deferred ones); rule processing started by a *savepoint* command produces an intermediate transaction state which is quiescent wrt all triggers.

## 13.3 Targeted and Untargeted Triggers

A trigger may be *targeted* to a specific class, when its triggering events are restricted to queries and updates over a specific class; otherwise, the trigger is *untargeted*. Targeted triggers are defined in the signature and implementation of their respective class. The syntax of implementations of targeted triggers may be simplified, because the class name is understood by the context, and therefore can be omitted from events and event formulas (next defined).

## 13.4 Priorities between Triggers

In order to control their execution, triggers have two additional components: the *before* and *after* declarations. These definitions are only useful if two triggers are "triggered" at the same time, and indicate which one of them should be considered first; by effect of these specifications, triggers are partially ordered.

- The **before** declaration includes the name of all triggers that have lower priority than the trigger being defined.
- The **after** declaration includes the name of all triggers that have higher priority than the trigger being defined.

We assume a default partial order where all immediate triggers precede all deferred triggers, and all targeted triggers precede all untargeted triggers; the first criterion is predominant over the second one. However, user-defined priorities may override this default ordering.

When many triggers are triggered at the same time, the system should consider them in decreasing priority order; when two triggers have the same priority, the choice between them is nondeterministic. Acyclicity of the precedence relation between triggers should be checked when a new trigger is defined.

## 13.5 Event Formulas

Conditions of triggers may include *event formulas*, particular formulas supported by the declarative language of **Chimera**, built by means of the binary predicates *occurred* and *holds*.

Syntactically, these predicates have two arguments: a set of events and a variable name. Events which may appear in the first argument of event formulas of a given trigger must be a subset of its triggering events. Observable events in Chimera include: *query, create, create_tmp, modify, delete, specialize, generalize, make_persistent*; *query* and *modify* can refer to specific attributes. An additional event *change* enables to refer to OIDs of all class instances that were affected by all the triggering update primitives of a given rule. Class names in event formulas can be omitted when they are clear from the context, e.g. in targeted triggers.

The variable appearing as second argument of the event formula must range over the objects of the class affected by the event, and becomes bound to OIDs instances which were subject to any of the events which are specified in the first argument; each OID bound by the computation of an event formula is called an *event instance*.

For instance, consider a trigger targeted on class C, a variable X over that class, and the event formula:

```
occurred((create,make_persistent),X)
```

After the evaluation of the event formula, X is bound to instances of class C which were either created or made persistent.

## 13.6 Net Effects for Event Predicates

The distinction between predicates *occurred* and *holds* is that in the former case all events which originally caused rule triggering are bound, while in the latter case some events are excluded: precisely, those events whose effect was compensated by subsequent events on the same object, thus computing the *net effect* of event instances. Compensations are performed as follows:

- A sequence of *create* and *delete* primitives on the same object, possibly with an arbitrary number of intermediate *modify* primitives on that object, has a null net effect.
- A sequence of *create* and several *modify* primitives on the same object has the net effect of a single create operation.

- A sequence of several *modify* and a *delete* primitive on the same object has the net effect of a single delete operation on that object [18] [19].

To illustrate the difference between *occurs* and *holds*, suppose that a condition is evaluated after the creation of an employee, with OID $e1$, and its subsequent deletion; then:

- After evaluating the formula *occurred(create(employee),X)*, X is bound to $e1$.
- After evaluating the formula *holds(create(employee),X)*, X is not bound.

Note that the net effect is computed at a given time during active rule processing, and we cannot exclude that other events will occur in the remainder of the transaction, yielding a different final composition of net effects.

### 13.6.1 Event Consumption Modes

Two distinct *event consumption modes* are possible for each trigger; this feature is relevant when a given trigger is considered multiple times in the context of the same transaction.

- Events can be *consumed* after the consideration of a rule; in this case, each event instance is considered by a rule only at its first execution, and then disregarded.
- Alternatively, events can be *preserved*, i.e., all events since the transaction start are considered at each rule consideration.

To understand the difference between consuming and preserving triggers, consider the following example. Let us assume that an active rule be initially triggered by events $E_1$ and then re-triggered by events $E_2$; consider its second triggering. Then bindings produced for the event formula *occurred(event,X)* at the second execution of $r_1$ are restricted to the set $E_2$ if events are consumed, and they are given by the set $E_1 \cup E_2$ if the events are preserved.

Both alternatives are semantically viable, as one trigger might respond to temporary inserts, e.g. for tracking change history, while another one might only respond to actual inserts, e.g. for consistency checking.

Event consumption modes and net effect evaluation are orthogonal; therefore, if the event formula *holds(event,X)* is used instead of *occurred(event,X)*, then event instances which are selected as indicated in this subsection, and next the net effect is computed over them.

[18] The composition of two subsequent *modify* events referring to the same object is automatically achieved by object orientation, since all updates on the same object correspond to the same OID and yield a unique binding.

[19] Other update primitives of **Chimera**, such as *create_tmp, make_persistent, specialize,* and *generalize* could be considered for net effect. Our current choice is to limit the definition of net effect to the primitives *create, delete,* and *modify*, which are more significant.

### 13.6.2 Reference to Old State

Finally, it is sometimes useful in triggers to refer to past database states. This feature is supported in **Chimera** by the function *old*, that can be applied to atomic formulas, indicating that the respective formula is to be evaluated in a previous database state. The chosen state depends on the event consumption mode:

- If the rule is event-consuming, then the old state is the one holding at the last consideration of the rule; prior to any consideration of a rule, the old state refers to the state at the transaction start.
- If the rule is event-preserving, then the old state refers to the state at the transaction start.

By offering all the above alternatives, we have designed a very rich trigger language, by means of which trigger semantics can be adapted to specific application needs rather than being forced to use one specific behavior, often "buried" inside the implementation of active rule systems. However, we are aware that such a rich language should be carefully used by application designers; in particular, we advocate an approach where triggers can be automatically generated by declarative specifications.

## 13.7 Defaults and Examples

Defaults for execution and consumption modes are set to *deferred* and *consuming*; alternative modes must be explicitly indicated by prefixing the keyword "trigger" by any of the keywords "immediate" or "event-preserving".

The following active rule provides the implementation of the **targeted, deferred** trigger *adjustSalary* for employees, requiring that, whenever an employee salary is greater than his manager's salary, then employee's salary be set equal to that of his manager:

```
define trigger  adjustSalary for employee
  events        create
                modify(salary)
  condition     Self.salary>Self.mgr.salary
  actions       modify(employee.salary,Self,Self.mgr.salary)
end
```

Note that in the above example all event formulas relate implicitly to the target object; in facts, event formulas of targeted triggers can refer only to events (queries or updates) defined on the target class.

The following **targeted, immediate** trigger is identical to the previous one, but it is considered at the end of the execution of the transaction line which raises the triggering event, rather than at the end of the transaction:

```
define immediate trigger immAdjustSalary for employee
  events        create
                modify(salary)
  condition     Self.salary>Self.mgr.salary
  actions       modify(employee.salary,Self,Self.mgr.salary)
end
```

The next example is a **untargeted, deferred** trigger which was introduced in Section ??. The trigger reacts to any change to salaries affecting a given department, whose condition produces some bindings (for the shared variables I and D) when the salary budget of a given department is exceeded by the sum of all salaries of the employees working at that department, and whose reaction is to adjust the salary's budget. This trigger has a lower priority with respect to the trigger which was introduced before. Note that this untargeted triggers refers to events defined over multiple different classes.

```
define trigger raiseBudget
    events     insert(employee)
               modify(employee.salary)
               modify(dept.members)
               modify(dept.salaryBudget)
    condition  dept(D), integer(I),
               I=sum(E.salary: employee(E), E in D.members),
               I>D.salaryBudget
    actions    modify(dept.salaryBudget,D,I)
    after      employee.adjustSalary
end
```

The next example illustrates the use of event formulas and of the function *old*. The **targeted, deferred, event-preserving** trigger selects all employees who get, in the course of the transaction, a high salary raise (possibly caused by small salary raises due to individual modify operations). Note that the rule is event-preserving, therefore all modifications since the transaction start are accumulated at each rule consideration; further, note that the condition part evaluates the salary difference between the state before transaction's execution and the new state determined at active rule processing time. The reaction consists in calling the external procedure *monitorSalary*.

```
define event-preserving trigger modifySpecialEmp for employee
  events        modify(salary)
  condition     occurred(modify(salary),X), integer(Y),
                Y=X.salary - old(X.salary), Y > 50000
  actions       monitorSalary(X,Y)
end
```

Finally, the next example illustrates the use of net effects; the **targeted, deferred** trigger is triggered by creation of employees, and the event formula excludes bindings to objects which were created and subsequently deleted by the transaction. The reaction consists in specializing the selected employees, by adding them to the subclass *specialEmp* of *employee*; the state of *specialEmp* includes the (new) attribute *raise*.

```
define trigger createSpecialEmp for employee
events        create
condition     net_occurred(create,X), integer(Y),
              Y=X.salary - old(X.salary), Y > 50000
actions       specialize(employee,specialEmp,X,(raise:Y))
end
```

## 13.8 Innovative Features of Triggers in Chimera

Innovative features of trigger management in **Chimera** include the introduction of *event formulas* (introduced by predicates *occurred* and *holds*), of alternative *consumption modes* for event instances, and of the meta-predicates *old*. In this section, we illustrate the rationale of our choices.

- We have excluded to support an *event calculus* for triggers: they can only monitor the disjunction of several simple events.
- We have excluded a *super-immediate* semantics of immediate triggers (i.e., starting their processing immediately after the raising of triggering events) because we prefer a priority-based iterative rule processing semantics to a recursive semantics. With our choice, the bindings to variables which are used in a transaction unit cannot be changed as effect of rule processing, thereby easing the understanding of transaction units themselves.
- Consequent to our exclusion of super-immediate semantics, the raising of triggering events is separated from the processing of relevant triggers. Thus, we do not bind event instances at the raising of events (e.g., by event formulas in the event part), but we bind them at rule consideration time (by means of event formulas in the condition).

- The notions of immediate/deferred and event consuming/preserving are orthogonal; they are motivated by specific applications. Their definition is set at the rule level because they determine execution semantics, and in particular the sets of event instances relevant to each rule consideration.
- Event formulas *occurred* and *holds* are used for accessing event instances; the use of either predicate indicates whether net effect should be applied or not. We decided that the elimination of event instances due to net effect composition should *not* result in de-triggering of rules; this would have heavily affected the semantics of rule processing. Instead, net effect is simply *computed* during rule consideration, possibly resulting in empty sets of bindings. For stressing this feature, we preferred to use two distinct predicates for event formulas rather than adding another mode to active rules.
- Priorities define a unique partial order, holding for all triggers (regardless of whether they are immediate or deferred); such unique partial order permits the run-time selection component to choose one of the triggered triggers at each rule processing step.

The rich offering of alternatives for triggers is motivated by their large spectrum of applications; we show that all the features provided by triggers in **Chimera** are required by some class of applications.

- *Default setting* (consuming, deferred) and use of net effects is suited for checking static integrity constraint at transaction commit; with this choice, the database is allowed to be invalid at an intermediate state of the transaction.
- *Event-preserving rules* are required for checking of dynamic integrity constraints; event preservation is required for detecting sequences of events that collectively violate the constraints, but not individually.
- *Immediate active rules* are required for triggers whose effect should "soon" become visible. However, **Chimera** users should be aware that immediate really means "at the end of the transaction line" rather than "after the tuple-level operation" (as in many tuple-oriented trigger mechanisms). With some care, we could use immediate active rules for supporting view materialization or data derivation, of for the early checking of integrity constraints.
- The transactional command *savepoint* is issued by an application when it needs to check constraints or alert users at an intermediate state of the transaction, rather that at commit; if savepoints are *recoverable* (i.e., associated to transactional-level persistence), then state consistency may indeed be required.
- Events *without net-effect composition* are required by triggers used for book-keeping, such as transactions which perform logging on the database journal, as they must track each individual change.

- Finally, triggers for *alerting* are normally event-consuming (i.e., they monitor each event only once) and require several of the alternative semantics which are offered: they can either be *immediate* (when they alert immediately after the occurrence of the monitored event) or *deferred* (when they alert at commit time); they can monitor *net effects* (reacting only if the event is still present), or instead monitor events *without composition*, when they track an event regardless of subsequent evolution.

## 14 SCHEMA UPDATES

Updating Chimera schemas is an argument left for future investigations. In this section we illustrate the current rationale for disallowing schema updates, and indicate the kind of difficulties that have to be solved in order to attach this problem successfully.

Currently, targets must be defined (both signature and implementation) before being populated by means of objects or values. In this way, a Chimera database produced as result of succesfully commited transactions is certainly consistent wrt integrity constraints. This is additionally guaranteed by the fact that constraints cannot be specialized; thus, adding a subclass does not change the consistency requirements on the elements of the superclass. Note that:

- There is no guarantee that constraints be satisfiable.
- Integrity may be enforced by means of active rules which react to constraint violations and restore consistency, as suggested in the literature of constraint management.
- There is no guarantee of termination of active rule processing.

Schema evolution is an important feature of modern database systems, as it enables to adapt to changes in the application requirements. Normally, there is a trade-off between supporting evolution and integrity; Chimera highlights the importance of supporting integrity, hence a careful management of schema updates is required.

Various alternatives for schema evolution may apply to Chimera objects. In general, it is easier to add new elements to an object, rather than changing or deleting existing elements; it is normally easier to change an implementation rather than a signature; and it is easier to add derived attributes rather than extensionally defined ones. Adding integrity constraints may be critical, because the the database state might be inconsistent with respect to the new constraint. Based on these considerations, the following schema update primitives could be first considered for Chimera:

```
add derived attribute attribute-name: domain
    to class class-name as {attribute implementation}
```

```
add operation operation-name (list of input parameters): output domain
    to class class-name as {operation implementation}

add trigger trigger-name
    to class-name as {trigger implementation}

change implementation for operation operation-name
    in class class-name as {new operation implementation}
```

In addition to targeted schema updates, it is possible to support untargeted schema updates; untargeted elements should only be added or deleted, but not updated. The addition of untargeted schema elements is currently supported in Chimera, as we do not restrict their definition time; therefore, untargeted views, constraints, and triggers can be added even if they refer to classes with a population. Note, however, that the addition of an untargeted constraint requires to check the validity of the current database with respect to the new constraint.

We do not expect to support the update or deletion of value types (they can only be added incrementally), or the schema update of value classes. It should instead be possible to destroy entire value or object classes from the database. This operation, however, requires care, because these classes may be mentioned in other targeted or untargeted schema elements, which then would become invalid; therefore, the operation should only be allowed for classes which are *not* referenced. A syntax for this schema update is:

```
destroy class class-name
```

Further investigation is needed in order to decide whether schema update should be supported at all in Chimera, possibly giving precedence to the schema update primitives whose syntax has been given above.

# 15 SUMMARY OF CL PRIMITIVES

In this final section, we summarize informally the features of the **Chimera Language** that have been introduced so far; a precise description of CL's syntax is given in Appendix 1.

## 15.1 Data Definition Language

We start by summarizing the distinction between schema and instance, and distinguishing exensional from intensional instances. A **schema** in Chimera consists of definitions of value types, value and object classes, and untargeted constraints, triggers, and views. Only classes are associated to persistent extents; therefore it is possible to store information about real wold concepts only if these are modeled as classes; this information constitutes the **extensional instance** of a Chimera database. From the extensional instance, it is possible to derive data through a variety of mechanisms; all derived data associated to a given extensional instance constitute the corresponding **intensional instance**.

### 15.1.1 Targeted Definitions

Targeted definitions include the definition of value types, value classes, and object classes. Each of them is given by distinguishing the signature from the implementation. Their syntax is sketched in the following:

```
define value type vtname: domain
  [constraints   constraint-definitions]
end

define value class vcname: domain
  [constraints   constraint-definitions]
end

define object class ocname
  [superclasses  superclasses-names]
  [attributes    attribute-definitions]
  [operations    operation-definitions]
  [constraints   constraint-definitions]
  [c-attributes  c-attribute-definitions]
  [c-operations  c-operation-definitions]
  [c-constraints c-constraint-definitions]
  [triggers      trigger-names]
end

define implementation for vtname
  [constraints   constraint-implementations]
end

define implementation for vcname
  [constraints   constraint-implementations]
end

define implementation for ocname
  [population    population-rules]
  [attributes    attribute-implementations]
  [operations    operation-implementations]
  [constraints   constraint-implementations]
  [c-attributes  c-attribute-implementations]
  [c-operations  c-operation-implementations]
```

```
    [c-constraints c-constraint-implementations]
  end

  define [optional-modes] trigger trigger-name for ocname
    events        event-definitions
    condition     condition-definition
    actions       action-definitions
    [before        trigger-names]
    [after         trigger-names]
  end
```

### 15.1.2 Untargeted Definitions

Untargeted schema elements include views, constraints, and triggers. Each of them is given by combining in the same definition both the signature and the implementation, as follows:

```
  define view vname: domain
    view-implementation
  end

  define constraint constraint-definition
    constraint-implementation
  end

  define [optional-modes] trigger trigger-name
    events        event-definitions
    condition     condition-definition
    actions       action-definitions
    [before       trigger-names]
    [after        trigger-names]
  end
```

## 15.2 Data Manipulation Language

Data manipulation primitives in Chimera consist of queries and updates. Both are either submitted from a *user-friendly interface* (UFI) or from an *application program interface* (API).

### 15.2.1 Queries

Queries supported from a UFI include *display* and *select* or *foreach*, whose syntax is roughly the following:

```
display(target-list-definition
        where complex-formula-definition
        [order-by order-specification])
select(target-list-definition [into-set|into-list]
       where complex-formula-definition
       [order-by order-specification])
for_each(target-list-definition
         where complex-formula-definition)
         do procedure-calls
```

Queries supported from an API include *select* and *next*, whose syntax is roughly the following:

```
select(target-list-definition
       where complex-formula-definition
       [order-by order-specification])
next(target-list-definition into-host-variable
     where complex-formula-definition
     [order-by order-specification])
```

### 15.2.2 External Procedures

The signature of external procedures can be defined in CL as follows:

```
define external procedure procedure-name(parameter-list) end
```

### 15.2.3 Updates

Updates in Chimera support object creation and deletion, object migration from one class to another, state change or change of persistency status of objects, and value class population and modification. Their rough syntax is:

```
create(class-name, state, object-variable)
create_tmp(class-name, state, object-variable)
delete(class-name, object-variable)

specialize(class-name, class-name, object-variable, state)
generalize(class-name, class-name, object-variable)

modify(class-name.attribute-name, object-variable, attribute-value)

make_persistent(class-name, object-variable)
```

```
add(value-class-name, value)
drop(value-class-name, value)
```

### 15.2.4 Transactions

Transactional commands supported in Chimera include:

```
opendb
closedb
begin-transaction
commit
rollback
```

## 16 CONCLUSIONS

This consolidated version of the Chimera specification has been prepared after a six-months phase of reconsideration of old, and investigation into new issues. Particularly those aspects related to active rule processing have matured since the first specification, mainly due to intensive discussions within the team at Politecnico di Milano. A brief summary of the main changes is contained in the companion document [8] and will not be repeated here.

Chimera is certainly an intricate conceptual development, but this is consequent from Chimera's intrinsic nature of merging, within a unique model and language, the main features of object-oriented, deductive, and active databases. We envision that each of the research groups involved in either basic research or testbed development will consider this document as a starting-point for setting research goals or determining/revising priorities driving their testbed implementations.

## REFERENCES

[1] S. Ceri, R. Manthey: *First Specification of Chimera (CM and CL)*, IDEA.DD.2P.004.02, May 1993.

[2] S. Ceri, U. Griefahn, R. Manthey: *A Summary of New Features of Chimera*, IDEA.DE.2P.005.01

[3] M. Novak, G. Gardarin, and P. Valduriez: *Flora, A Functional-Style Language for Object and Relational Algebra*, IDEA.DD.4N.001, February 1993.

[4] E. Baralis, E. Bertino, F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca: *"Issues in the design of the IDEA Conceptual Interface"*, IDEA.DD.2P.001.01, July 1992.

[5] P. Bayer, S. Bressan, A. Sikeler, B. Wüthrich: *"Combining Object-Oriented and Deductive Database Definition and Manipulation Languages"*, Internal Report ECRC-92, September 1992.

[6] P. Dechamboux, M. Lopez, C. Roncancio: *"The Data Model of the Peplom DBPL"*, IDEA.WP.1B.001, October 1992.

[7] P. Bayer, J. Fox: *"State-of-the-Art Report on Reactive Processing"*, IDEA.DD.2E.001, June 93.

[8] S. Ceri, and J. Widom. *"Deriving Production Rules for Constraints Maintenance"*. In *Proc. of the 16th International Conference on Very Large Data Bases,* pages 566-577, Brisbane, Australia, August 1990.

## A1 SYNTAX OF THE CHIMERA LANGUAGE

In this appendix an EBNF syntax of the Chimera Language is presented. Every feature of CL is included, with the only exception of the API. The grammar is organized in sections:

1. Top-level productions.
2. Signatures of targeted concepts.
3. Implementation of targeted concepts.
4. Untargeted concepts.
5. Operations.
6. External procedures.
7. Transaction primitives.
8. Triggers.
9. Actions.
10. Query language primitives.
11. Update primitives.
12. Expressions, terms and operators.
13. Formulas.

The above sections of the grammar define sublanguages of Chimera, that may help in the identification of compilation units. Normally, productions of sublanguages can "call" productions of lower-level sublanguages; for instance, productions of Section 8 (triggers) "call" productions of Section 9 (actions), which in turn "call" productions of Section 10 (queries) and 11 (updates). Finally, formulas, expressions, terms and operators may be "called" by most of higher levels. "IDENTIFIER" is a user-supplied name beginning with a lower-case letter which cannot be a keyword. "VARIABLE" is a user-supplied name beginning with a capital letter. "NUMBER" is a natural number.

With respect to the previous version of the Syntax, the main changes we introduced are:

- separation in the grammar of ddl and dml transactions,
- a new organization of the expression/term section (in order to make easier the subsequent semantic control phase),
- the introduction of a new symbol for the record construct ("[" instead of "(") and for the list construct (" < " instead of "[") to solve a syntactical ambiguity,
- the introduction of the "define name" primitive in the action section to associate an oid to a user-defined name,
- some changes to the syntax of the triggers ("holds", "change") and the transactional-commands ("savepoint"),
- a new organization of the top-level productions to implement the concept of "transaction-line",
- the new format of the operation-call,
- the use of the word "end" to close each definition section.

## A1.1 Top-level Productions

```
chimera ::= begin_section transaction {transaction} "close_db"

begin_section ::= "create_db" | "open_db"

transaction ::= "begin" "transaction" transaction_body end_transaction

end_transaction ::= "commit" | "rollback"

transaction_body ::= ddl_transaction | dml_transaction

ddl_transaction ::= ddl ";" {ddl ";"}

ddl ::= targeted | untargeted | external
```

```
targeted ::= signature | implementation

dml_transaction ::= dml ";" {dml ";"}

dml ::= action {connector action}

connector ::= "," | "%"
```

## A1.2 Signatures of Targeted Concepts

```
signature ::= object_definition
            | value_definition

object_definition ::=
   "define" "object" "class" class_name class_reference_body "end"

class_reference_body ::=
   [[derived_option] superclasses]
   [attributes]
   [operations]
   [constraints]
   [class_attributes]
   [class_operations]
   [class_constraints]
   [triggers]

superclasses ::= "superclasses" superclass_list

attributes ::=  "attributes" attribute_list

operations ::= "operations" operation_list

constraints ::= "constraints" constraint_list

class_attributes ::=  "c_attributes" attribute_list

class_constraints ::=  "c_constraints" constraint_list

class_operations ::=  "c_operations" operation_list

triggers ::= "triggers" trigger_list
```

```
superclass_list ::= class_name {"," class_name}

attribute_list ::= attribute {"," attribute}

attribute ::= attribute_name ":" type_structure [ options ]

options ::= redefined_option [derived_option]
          | derived_option [redefined_option]

derived_option ::= "derived"

redefined_option ::= "redefined"

type_structure ::= named_type
                 | structured_type

named_type ::= atomic_type
             | user_named_type

structured_type ::= list_type
                  | record_type
                  | set_type

atomic_type ::= "boolean"
              | "char"
              | "integer"
              | "real"
              | "string"
              | "string" "(" "NUMBER" ")"

user_named_type ::= class_name | value_name

list_type ::= "list_of" "(" type_structure ")"

record_type ::= "record_of" "(" fields ")"

set_type ::= "set_of" "(" type_structure ")"

fields ::= field { "," field }

field ::= [label_name ":"] type_structure

constraint_list ::= constraint_element {"," constraint_element}

constraint_element ::= constraint_name "(" parameter_list ")"
```

```
parameter_list ::= parameter { "," parameter }

parameter ::= variable_name ":" domain_name

operation_list ::= operation_element {"," operation_element}

operation_element ::=
   operation_name "(" [op_parameter_list] ")" [redefined_option]

op_parameter_list ::= op_parameter {"," op_parameter}

op_parameter ::= op_token variable_name ":" domain_name

op_token ::= "in" | "out"

trigger_list ::= trigger {"," trigger}

trigger ::= trigger_name

value_definition ::= value_type_definition
                   | value_class_definition

value_class_definition ::= "define" "value" "class" value_name
                           ":" type_structure [ constraints ] "end"

value_type_definition ::= "define" "value" "type" value_name
                          ":" type_structure [ constraints ] "end"

domain_name ::= named_type

attribute_name ::= "IDENTIFIER"

constraint_name ::= "IDENTIFIER" | "key" | "notnull"

class_name ::= "IDENTIFIER"

label_name ::= "IDENTIFIER"

trigger_name ::= "IDENTIFIER"

value_name ::= "IDENTIFIER"

variable_name ::= "VARIABLE"
```

## A1.3 Implementation of Targeted Concepts

```
implementation ::= object_implementation
                 | targeted_trigger
                 | value_implementation

object_implementation ::=
   implementation_alternative "implementation" "for" class_name
   [population_implementation]
   [attribute_implementation]
   [operation_implementation]
   [constraint_implementation]
   [c_attribute_implementation]
   [c_operation_implementation]
   [c_constraint_implementation]
   "end"

implementation_alternative ::= "define" | "redefine"

population_implementation ::= "population" passive_rule_list

attribute_implementation ::= "attributes" passive_rule_list

operation_implementation ::= "operations" active_rule_list

constraint_implementation ::= "constraints" passive_rule_list

c_attribute_implementation ::= "c_attributes" passive_rule_list

c_constraint_implementation ::= "c_constraints" passive_rule_list

c_operation_implementation ::= "c_operations" active_rule_list

active_rule_list ::= operation_rule {";" operation_rule}

passive_rule_list ::= passive_rule {";" passive_rule}

passive_rule ::= head "<-" body

head ::= atomic_formula

body ::= formula

value_implementation ::=
   "define" "implementation" "for" value_name
```

```
   [constraint_implementation]
   "end"

targeted_trigger ::= targeted_trigger_rule
```

## A1.4 Untargeted Concepts

```
untargeted ::= untargeted_constraint | untargeted_trigger
             | untargeted_view

untargeted_constraint ::=
   "define" "constraint" constraint_name "(" parameter_list ")"
   passive_rule_list "end"
 | "define" "constraint" "inverse"
   "(" variable_name ":" qualified_attribute_name "," variable_name
   ":" qualified_attribute_name ")" "end"

untargeted_trigger ::= untargeted_trigger_rule

untargeted_view ::=
   "define" "view" view_name ":" type_structure passive_rule_list "end"

view_name ::= "IDENTIFIER"
```

## A1.5 Operations

```
operation_rule ::= operation_name argument_list ":" formula
                   "->" list_of_actions

operation_name ::= "IDENTIFIER"
```

## A1.6 External Procedures

```
external ::= "define" "external" "procedure"
             procedure_name "(" [parameter_list ] ")" "end"
```

```
procedure_name ::= "IDENTIFIER"
```

## A1.7 Triggers

```
targeted_trigger_rule ::=
   "define" [trigger_options] "trigger" trigger_name "for" class_name
   "events" t_trig_events
   "condition" t_condition_formula
   "actions" reactions
   [priority_option]
   "end"

untargeted_trigger_rule ::=
   "define" [trigger_options] "trigger" trigger_name
   "events" unt_trig_events
   "condition" unt_condition_formula
   "actions" reactions
   [priority_option]
   "end"

trigger_options ::= trigger_consumption_token [trigger_execution_option]
                  | trigger_execution_token [trigger_consumption_option]

trigger_consumption_option ::= trigger_consumption_token

trigger_consumption_token ::= "event_consuming"
                            | "event_preserving"

trigger_execution_option ::= trigger_execution_token

trigger_execution_token ::= "deferred"
                          | "immediate"

t_trig_events ::= t_event { t_event }

unt_trig_event ::= unt_event { unt_event }

t_event ::= "create"
        | "create_tmp"
        | "delete"
        | "make_persistent"
        | "generalize" "(" class_name ")"
        | "specialize" "(" class_name ")"
```

```
        | "modify" [ "(" attribute_name ")" ]
        | "query" [ "(" attribute_name ")" ]
        | "change"
        | operation_name

unt_event ::= "create" "(" class_name ")"
        | "create_tmp" "(" class_name ")"
        | "delete" "(" "class_name" ")"
        | "make_persistent" "(" "class_name" ")"
        | "generalize" "(" class_name "," class_name ")"
        | "specialize" "(" class_name "," class_name ")"
        | "modify" "(" class_name [opt_attrib] ")"
        | "query" "(" class_name [opt_attrib] ")"
        | "change "(" "class_name" ")"
        | class_name "." operation_name

opt_attrib ::= "." attribute_name

t_condition_formula ::= t_simple_condition_formula_list
                      | "true"

unt_condition_formula ::= unt_simple_condition_formula_list
                        | "true"

t_simple_condition_formula_list ::= t_simple_condition_formula
                            { "," t_simple_condition_formula }

unt_simple_condition_formula_list ::= unt_simple_condition_formula
                             { "," unt_simple_condition_formula }

t_simple_condition_formula ::= pos_or_neg_atomic_formula
                             | t_event_formula
                             | "not" t_event_formula

unt_simple_condition_formula ::= pos_or_neg_atomic_formula
                               | unt_event_formula
                               | "not" unt_event_formula

t_event_formula ::= event_token "(" t_event_list "," variable_name ")"

unt_event_formula ::= event_token "(" unt_event_list "," variable_name ")"

event_token ::= "holds" | "occurred"

t_event_list ::= t_event { "," t_event }
```

```
unt_event_list ::= unt_event { "," unt_event }

reactions ::= reaction { connector reaction }

reaction ::= action
           | "rollback"

priority_option ::= "after" qualified_trigger_list [before_option]
                  | "before" qualified_trigger_list [after_option]

before_option ::= "before" qualified_trigger_list

after_option ::= "after" qualified_trigger_list

qualified_trigger_list ::=
   qualified_trigger_name {"," qualified_trigger_name}

qualified_trigger_name ::= [class_name "."] trigger_name
```

## A1.8 Actions

```
action ::= query_cmd
         | update
         | operation_call
         | procedure_call
         | name_definition
         | "savepoint"

procedure_call ::= procedure_name "(" [exp_list] ")"

operation_call ::= var_or_oid_name "." operation_name
                   "(" [exp_list] ")"

name_definition ::= "assign" "name" "(" variable_name ","
                     oid_name ")"

oid_name ::= "IDENTIFIER"
```

## A1.9 Query Language Primitives

```
query_cmd ::= display_cmd | for_each_cmd | select_cmd

display_cmd ::=
   "display" "(" start_with_var_or_id_list [where_option]
    [ordering_option] ")"

start_with_var_or_id_list ::= start_with_var_or_id
                            { ","  start_with_var_or_id }

start_with_var_or_id ::= start_with_variable | start_with_identif

where_option ::= "where" formula

select_cmd ::=
   "select" "("  var_list  [select_into_var_option]
   "where" formula [ordering_option] ")"

var_list ::= variable_name { ","  variable_name }

select_into_var_option ::= "into_set" variable_name
                         | "into_list" variable_name

for_each_cmd ::=
   "for_each" "(" var_list "where" formula ")" "do"
    list_of_actions "end_do"

list_of_actions ::= action { connector action }

ordering_option ::= "order_by" ordering_option_list

ordering_option_list ::= ordering_term {"," ordering_term}

ordering_term ::= ordering_op "(" start_with_variable ")"

ordering_op ::= "ascending" | "descending"
```

## A1.10 Update Primitives

```
update ::= create_cmd
```

```
          | create_tmp_cmd
          | delete_cmd
          | change_persistency_cmd
          | generalize_cmd
          | specialize_cmd
          | modify_cmd
          | population_cmd

create_cmd ::=
   "create" "(" class_name "," record "," variable_name ")"

create_tmp_cmd ::=
   "create_tmp" "(" class_name "," record "," variable_name ")"

delete_cmd ::=
   "delete" "(" class_name "," variable_name ")"

change_persistency_cmd ::=
   "make_persistent" "(" class_name ","  variable_name ")"

generalize_cmd ::=
   "generalize" "(" class_name "," class_name "," var_or_oid_name ")"

specialize_cmd ::=
   "specialize" "(" class_name "," class_name ","
   var_or_oid_name "," record ")"

modify_cmd ::=
   "modify" "(" start_with_identif "," var_or_oid_name "," expression ")"

population_cmd ::=
   "add" "(" class_name "," expression ")"
 | "drop" "(" class_name "," expression ")"

var_or_oid_name ::= oid_name | variable_name
```

## A1.11 Expressions, Terms and Operators

```
exp_list ::= expression { "," expression }

expression ::= sum_exp { add_op expression }

add_op ::= "+" | "-"
```

```
sum_exp ::= mult_exp { mult_op sum_exp }

mult_op ::= "*" | "/" | "mod" | "//"

mult_exp ::= "(" expression ")" | term

term ::= set_or_list | record | simple_term

set_or_list ::= set | list

set ::= "{" [ term { "," term } ] "}"

list ::= "<" [ term { "," term } ] ">"

record ::= "[" record_fields "]"

record_fields ::= record_field { "," record_field }

record_field ::= [ label_name ":" ] term

simple_term ::= "abs" "(" expression ")"
      | "-" minus_follows
      | "old" "(" var_or_oid_name [ opt_attrib ] ")"
      | list_unary_operator
      | constant
      | start_with_variable
      | start_with_identif
      | aggregate_operator

minus_follows ::= simple_term | "(" expression ")"

list_unary_operator ::= "hd" "(" sum_exp ")" | "tl" "(" sum_exp ")"

constant ::= "CHAR" | "STRING" | "INTEGER" | "REAL" | boolean_value
           | "null"

boolean_value ::= "true" | "false"

start_with_variable ::= variable_name { dot_notation }

start_with_identif ::= "IDENTIFIER" { dot_notation }

dot_notation ::= "." dot_follows

dot_follows ::= "IDENTIFIER" | "NUMBER"
```

```
aggregate_operator ::= agg_op "(" expression [ where_option ] ")"

agg_op ::= "max" | "min" | "sum" | "avg" | "card"
```

## A1.12 Formulas

```
formula ::= pos_or_neg_atomic_formula {"," pos_or_neg_atomic_formula }

pos_or_neg_atomic_formula ::= atomic_formula | "not" atomic_formula

atomic_formula ::= class_formula
                 | type_formula
                 | constraint_formula
                 | comparison_formula
                 | membership_formula
                 | choose_formula

class_formula ::= class_name "(" variable_name ")"
                 | class_name "(" record ")"

type_formula ::= named_type "(" variable_name ")"
                 | named_type "(" record ")"

constraint_formula ::= [class_name "."] constraint_name argument_list
                      | "inverse" "(" class_name "." attribute_name ","
                                      class_name "." attribute_name ")"

argument_list ::= "(" exp_list ")"

comparison_formula ::= expression comparison_op expression

comparison_op ::= "<" | "<=" | ">" | ">=" | "=" | "!=" | "=="

membership_formula ::= expression "in" expression |
                       expression in class_name

choose_formula ::= "choose" "("  choosing_set  ","  cardinality  ","
                   variable_name ")"

choosing_set ::= expression

cardinality ::= expression
```